Security Audit Report

Hydration Peg Drift Stableswap

v1.0

May 2, 2025

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Intergalactic Limited to perform a security audit of Hydration Peg Drift Stableswap Security Audit.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/galacticcouncil/hydration-node
Commit	`253132a7b089b79158463bbf43870e78a70d8860`
Scope	Scope of the tests was limited to components listed below: ● `math/src/ratio.rs` ● `math/src/stableswap` ● `pallets/stableswap` ● `pallets/ema-oracle`
Fixes verified at commit	`f671b1c51461842936684a40cf8d1685a45b8080`

	Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Hydration is a DeFi protocol built on Substrate, offering advanced trading, liquidity provision, and price discovery mechanisms for the Polkadot ecosystem. It integrates various financial primitives including stableswap AMMs, omnipool liquidity aggregation, EMA oracles, and cross-asset routing capabilities to create a robust foundation for efficient asset exchange and yield generation.

The scope of audit was limited to `swableswap` and `ema-oracle` pallets, with mathematical concepts supporting it.

The `stableswap` pallet is a Curve/stableswap-style Automated Market Maker (AMM) designed for highly efficient and low-slippage trades between assets of similar value. The pallet supports advanced features including multi-asset pools, dynamic fee adjustment, weighted pegging mechanisms, and comprehensive liquidity operations.

The `ema-oracle` pallet provides exponential moving average (EMA) oracles of different time periods for price, volume, and liquidity data across various asset pairs. The oracle data is accessible through standardized interfaces and is designed to be integrated with other pallets to support price discovery and financial operations.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	The code is self-explanatory, extrinsics have extensive specification in the form of comments, but additional documentation and functional assumptions have not been provided.
Test coverage	Low	Test coverage for `pallets`, `primitives`, and `runtime` reported by `cargo tarpaulin` is `5.67%`.

Summary of Findings

No	Description	Severity	Status
1	Potential denial of service by storing non-whitelisted pairs from Bifrost oracle	Major	Acknowledged
2	Liquidity providers can be blocked from exit	Major	Acknowledged
3	Liquidity of any asset can be inflated or deflated by the registry owner	Major	Acknowledged
4	Unexpected behavior results from inability to remove oracle data set during genesis	Minor	Acknowledged
5	Missing slippage protection in `add_liquidity` extrinsic	Minor	Resolved
6	Missing protection for rapid amplification changes	Minor	Acknowledged
7	Pool destructure procedure is not complete	Informational	Resolved
8	Silent duplication error handling during liquidity removal	Informational	Resolved
9	Insufficient pool fee validation	Informational	Acknowledged
10	Missing staleness monitoring for oracle data	Informational	Acknowledged
11	Potential for optimization in shares calculation	Informational	Acknowledged
12	Possible optimization in `create_pool` function	Informational	Resolved
13	Inconsistency in `update_amplification` specification	Informational	Resolved
14	Misleading error messages	Informational	Acknowledged
15	Missed invariant verification	Informational	Resolved
16	Redundant balance validations	Informational	Acknowledged
17	Unresolved `TODO` comments in the codebase	Informational	Acknowledged

Detailed Findings

1. Potential denial of service by storing non-whitelisted pairs from Bifrost oracle

Severity: Major

At `pallets/ema-oracle/src/lib.rs:363`, data from the Bifrost source is accepted and stored in the `Accumulator` even if the pair is not whitelisted. This can lead to a potential Denial of Service scenario.

If Bifrost continuously pushes updates for non-whitelisted pairs, they consume the `MaxUniqueEntries` capacity, preventing valid (whitelisted) entries from being recorded. Since these unwanted entries are never removed (the pairs are not whitelisted), the pallet effectively becomes cluttered with irrelevant data, locking out legitimate updates.

Recommendation

We recommend rejecting `Bifrost` updates for non-whitelisted pairs.

Status: Acknowledged

2. Liquidity providers can be blocked from exit

Severity: Major

In `pallets/stableswap/src/lib.rs:614-620`, the function `remove_liquidity_one_asset` validates that either the liquidity provider attempts to perform full exit from the pool by comparing total `share_issuance` and requested `share_amount`, or that the amount of liquidity left in the pool after the withdrawal is greater than `MinPoolLiquidity`.

However, this validation generates scenarios where liquidity providers cannot leave the pool. For example:

1. A pool is created and `MinPoolLiquidity` is set to 100$.
2. Liquidity providers Alice and Bob enter the pool with shares 100$ each.
3. Alice removes 99$ of liquidity, rendering the pool to have 101$ of liquidity.
4. Now, Bob cannot withdraw the full amount of his 100$.
5. Bob can withdraw only 1$.

As it is implemented in the function `remove_liquidity`, pools are destroyed whenever full liquidity is withdrawn by a single liquidity provider. However, the pool destruction is not triggered when it falls below `MinPoolLiquidity` but is still above zero. This limit also can prevent providers from full exit.

Recommendation

We recommend checking `current_share_balance` and allowing any user to liquidate their share if they intend to exit completely.

Status: Acknowledged

3. Liquidity of any asset can be inflated or deflated by the registry owner

Severity: Major

In `math/src/stableswap/math.rs:752-767`, the function `normalize_value` is implemented which normalizes balances of assets having different decimals. For instance, if the number of decimals is less than `TARGET_PRECISION`, the corresponding balance is multiplied by 10^K where K is the difference between the two decimals. The function `normalize_value` is used in most core calculations related to the pool operations.

However, the function `normalize_value` is not utilized in the `do_add_liquidity` function defined in `pallets/stableswap/src/lib.rs:1347-1465`. Specifically, on line `1385` when new liquidity is summed up with the current amount stored in the variable `reserve`.

It might look at first glance that decimals of assets stored in liquidity pools cannot change. However, as it is seen in `pallets/asset-registry/src/lib.rs:460-468` decimals of already registered assets can be updated by the registry owner. In case of such an update, all existing liquidity pools including the updated asset become corrupted — all the liquidity already deposited in such pools will be implicitly multiplied or divided by 10^K.

The most realistic scenario resulting in such a pool corruption is denomination performed by on-chain governance when the consequence on the pools is not obvious. Furthermore, if the registry owner is a centralized administration entity, there is a high systemic risk since such an entity can inflate or deflate liquidity in the pools to any number, at will or by mistake.

Recommendation

We recommend tracking asset decimals uniformly across all pools and normalizing liquidity levels after each decimals update.

Status: Acknowledged

4. Unexpected behavior results from inability to remove oracle data set during genesis

Severity: Minor

In the `ema-oracle` pallet at `pallets/ema-oracle/src/lib.rs:224`, genesis oracle entries are stored within the pallet's storage. However, the associated assets are not automatically added to the `WhitelistedAssets` variable. This creates a scenario where an oracle supporting a specific asset pair cannot be removed if support for that asset pair is later discontinued. The `remove_oracle` extrinsic enforces a check that only oracles associated with whitelisted assets can be removed.

This behavior prevents authorized users from removing outdated or unwanted oracles from the pallet. This results in the accumulation of unnecessary data within the pallet and introduces the risk of unintended oracle usage within the stableswap pallet.

Recommendation

We recommend that during genesis assets should be included in the `WhitelistedAssets`. This will enable the removal of the oracle by authorized origin at a later time.

Status: Acknowledged

5. Missing slippage protection in `add_liquidity` extrinsic

Severity: Minor

Unlike other liquidity functions in the pallet, the `add_liquidity` function defined in `pallets/stableswap/src/lib.rs:516` does not implement slippage protection.

This is inconsistent with both other liquidity functions in the implementation (like `add_liquidity_shares` and `remove_liquidity_one_asset`) and with Curve Finance's implementation, which does include a `min_mint_amount` parameter.

Users adding liquidity may receive significantly fewer shares than expected due to front-running attacks or market volatility between transaction creation and execution, potentially resulting in financial losses.

Recommendation

We recommend adding a minimum shares parameter working as a slippage protection mechanism.

Status: Resolved

6. Missing protection for rapid amplification changes

Severity: Minor

The `update_amplification` function in `pallets/stableswap/src/lib.rs:448` lacks a minimum timeframe requirement that must elapse between consecutive amplification parameter changes.

Unlike Curve Finance, which enforces a minimum timeframe (`MIN_RAMP_TIME` constant set to 86400 seconds/1 day), this implementation allows rapid sequential amplification changes. A malicious actor with `AuthorityOrigin` privileges could manipulate the amplification parameter in rapid succession, causing pool instability and creating arbitrage opportunities to extract value from the pool.

Recommendation

We recommend implementing a minimum timeframe constant similar to Curve's implementation and enforce validation that sufficient time has elapsed since the previous amplification change.

Status: Acknowledged

7. Pool destructure procedure is not complete

Severity: Minor

In `pallets/stableswap/src/lib.rs:1089-1092`, the `remove_liquidity` function implements the pool destruction. However, corresponding peg data expressed as value of type `PegInfo` is not removed from the storage. This leaves stale entries in `PoolPegs` even though the pool is no longer active.

Recommendation

We recommend removing the pool's peg information together with the pool itself.

Status: Resolved

8. Silent duplication error handling during liquidity removal

Severity: Informational

The `remove_liquidity` function does not explicitly validate that asset IDs in `min_amounts_out` are unique. In `pallets/stableswap/src/lib.rs:1044` there is a validation checking, if `min_amounts_out` length is the same as `pool.assets` length, however it does not clearly ensure that unique assets amount is equal.

When duplicate assets are provided, the function will fail later when attempting to access a missing asset, but the error message (`IncorrectAssets`) doesn't clearly communicate the specific issue.

Recommendation

We recommend adding explicit validation for duplicate assets before conversion to BTreeMap, providing a more specific error message.

Status: Resolved

9. Insufficient pool fee validation

Severity: Informational

In `pallets/stableswap/src/lib.rs`, the `update_pool_fee` function is defined that allows updating pool fees to new values. However, several important validations are missing:

- The `pool.fee` variable of type `Permill` is assigned to the new fee value without any validation of its range. This means the fee could be set up to `100%` preventing liquidity providers from any profits. Such an extreme fee could break normal pool operations.
- The new fee value is not compared against the current value. This allows redundant updates.

Recommendation

We recommend introducing a maximum allowed fee to avoid excessively high fees and also validate that the update is not redundant.

Status: Acknowledged

10.Missing staleness monitoring for oracle data

Severity: Informational

Within `pallets/stableswap/src/lib.rs:1782`, when the code fetches oracle data via `get_raw_entry`, it does not verify whether the oracle feed is updating regularly. Nor does it log any warnings if updates stop for an extended period. As a result, the pallet may unknowingly rely on an outdated price—especially if, for instance, the Bifrost oracle feed ceases to provide fresh data.

Recommendation

We recommend adding a staleness check and log messages (or alerts) if oracle data is older than a configured threshold.

Status: Acknowledged

11. Potential for optimization in shares calculation

Severity: Informational

In `math/src/stableswap/math.rs:174`, within the `calculate_shares` function, the function call to `calculate_d` can be optimized for the case when `share_issuance` is `0`.

In such a case, `Some((updated_d, fees))` value can be returned even before computing the value of the `adjusted_reserves` variable.

Recommendation

We recommend streamlining implementation of the function `calculate_shares`. See the Appendix.

Status: Acknowledged

12. Possible optimization in `create_pool` function

Severity: Informational

It was observed that the `create_pool` function defined in `pallets/stableswap/src/lib.rs` takes an `assets` parameter of type `Vec<T::AssetId>`. Although the actual length of that vector is validated to not exceed a defined threshold and `MaxAssetsExceeded` error is returned otherwise at line `1301`, such validation is not executed right away.

There is no security impact associated with doing it later, however, executing it at the beginning of the function can reduce the unnecessary execution time in cases when it's known to fail.

An even better approach could be changing the type of the `assets` parameter to `BoundedVec` which by default cannot be longer than a specified length. This approach is already applied in other functions, e.g. `add_liquidity`.

Recommendation

We recommend changing the `assets` parameter type to `BoundedVec` if possible. Alternatively, asserting the length of `assets` vector early should be implemented.

Status: Resolved

13. Inconsistency in `update_amplification` specification

Severity: Informational

The function comments for `update_amplification` in `pallets/stableswap/src/lib.rs:448` do not match the actual implementation parameters. The comments reference different parameter names and structures.

This inconsistency may cause confusion during code maintenance and for developers integrating with the pallet.

Recommendation

We recommend updating the function documentation to accurately reflect the parameter names used in the implementation.

Status: Resolved

14. Misleading error messages

Severity: Informational

Throughout the codebase, the `ArithmeticError::Overflow` error is used incorrectly to report general validation issues. For example, in `pallets/stableswap/src/lib.rs:1414`, it is used when the `calculate_shares` function returns `None`.

Other locations where `ArithmeticError::Overflow` is used:

- `pallets/stableswap/src/lib.rs:633, 726, 1233, 1276, 1414, 1503, 1610`
- `pallets/stableswap/src/trade_execution.rs:54, 128, 174, 205, 325`

Additionally, in `pallets/stableswap/src/lib.rs:1369`, the `IncorrectAssets` error is returned after detecting a duplicate in the input vector of assets. A more precise message would be `DuplicatedAssets`.

Recommendation

We recommend correcting error messages for better clarity and maintainability.

Status: Acknowledged

15.Missed invariant verification

Severity: Informational

There are several invariants verified during test runs and enabled by the `try-runtime` flag. One of such invariants is `ensure_remove_liquidity_invariant`, used in `pallets/stableswap/src/lib.rs:1103`. This invariant is verified in both `remove_liquidity` and `remove_liquidity_one_asset`, but not in `withdraw_asset_amount` which is a liquidity removal, too.

Recommendation

We recommend verifying the `ensure_remove_liquidity_invariant` invariant in the `withdraw_asset_amount` function.

Status: Resolved

16.Redundant balance validations

Severity: Informational

There are several redundant assertions have been discovered, both involving user balances:

- During liquidity withdrawals, e.g. in `pallets/stableswap/src/lib.rs:1032`, it is validated that `current_share_balance` is greater than `share_amount` and error `InsufficientShares` is returned otherwise. However, if the user does not have enough shares, later the attempt to burn them using `T::Currency::withdraw(pool_id, &who, share_amount)` will fail anyway. This validation is implemented in both `remove_liquidity` and `remove_liquidity_one_asset`, but is missing in `withdraw_asset_amount`.
- During liquidity deposits, in `pallets/stableswap/src/lib.rs:1364`, it is validated that the user has more than `asset.amount` tokens. Similarly, to the previous point, if it is not the case, the deposit will fail during `T::Currency::transfer(asset.asset_id, who, &pool_account, asset.amount)`.

Redundant assertions reduce code readability and maintainability.

Recommendation

We recommend removing the above mentioned assertions.

Status: Acknowledged

17. Unresolved `TODO` comments in the codebase

Severity: Informational

The codebase demonstrates good in-line documenting practices and does not use `TODO` comments in general. However, two such comments have still been identified within the given scope of this audit (excluding tests):

- `pallets/stableswap/src/lib.rs:235`
- `pallets/stableswap/src/lib.rs:742`

In general, `TODO` and `FIXME` comments tend to accumulate without resolution and often become outdated, hence reducing codebase maintainability and readability.

Recommendation

We recommend resolving the listed `TODO` comments, or moving them to the proper task tracker system.

Status: Acknowledged

Appendix A: Suggestions

1. Optimized code for "Potential for optimization in shares calculation"

```
if share_issuance == 0 {
      return Some((updated_d, fees))
};

let adjusted_reserves = updated_reserves
      .iter()
      .enumerate()
      .map(|(idx, asset_reserve)| -> Option<AssetReserve> {
      let (initial_reserve, updated_reserve) =
to_u256!(initial_reserves[idx].amount, asset_reserve.amount);
      let ideal_balance = d1.checked_mul(initial_reserve)?.checked_div(d0)?;
      let diff =
Balance::try_from(updated_reserve.abs_diff(ideal_balance)).ok()?;
      let fee_amount = fee.checked_mul_int(diff)?;
      fees.push(fee_amount);
      Some(AssetReserve::new(
            asset_reserve.amount.saturating_sub(fee_amount),
            asset_reserve.decimals,
      ))
      })
      .collect::<Option<Vec<AssetReserve>>>()?;

let adjusted_d = calculate_d::<D>(&adjusted_reserves, amplification, pegs)?;
let (issuance_hp, d_diff, d0) = to_u256!(share_issuance,
adjusted_d.checked_sub(initial_d)?, initial_d);
let share_amount = issuance_hp.checked_mul(d_diff)?.checked_div(d0)?;
let shares_amount = Balance::try_from(share_amount).ok()?;

Some((shares_amount, fees))
```

Security Model

The security model has been carefully crafted to delineate the various assets, actors, and underlying assumptions of Hydration Peg Drift Stableswap. They will then be analyzed in a threat model to outline high-level security risks and proposed mitigations.

The purpose of this security model is to recognize and assess potential threats, as well as the derivation of recommendations for mitigations and counter-measures.

There is a limit to which security risks can be identified by constructing a security/threat model. Some risks may remain undetected and may not be covered in the model described below (see disclaimer).

Assets

The following outlines assets that hold significant value to potential attackers or other stakeholders of the system.

Token assets

Pool reserves - The actual token balances controlled by pool accounts that users trade against. These represent the primary value held within the protocol and are a critical target for protection.

Share tokens - Unique tokens minted for each pool representing liquidity provider ownership. These tokens establish claims to portions of the pool reserves and carry significant value.

Rebasing tokens - Tokens whose balances can change proportionally across all accounts that hold them. The stableswap pools in Hydration will include rebasing tokens similar to AAVE's "A" tokens. These tokens create unique security considerations because:

- When a token in an AMM rebases, its quantity changes in a way not governed by AMM rules,
- Assuming the AMM references account balances as reserves, rebases will change AMM reserves,
- For stableswap, rebasing changes both the invariant and the spot price,
- Assumptions about invariant stability, share-to-reserves ratios, or price stability between user interactions may no longer hold,
- Design considerations or safety assurances based on these assumptions may be invalidated.

Bridged assets - Wrapped tokens from other chains that may exist within pools. These cross-chain assets require special security considerations due to their interactions with external properties.

Protocol state data

Stableswap pool state - Configuration data structures maintaining asset details and pool parameters. This state data determines how pools operate and must be protected from unauthorized modifications.

Pool peg information - Pegged values stored in `pallet_stableswap::PoolPegs`. These values are crucial for maintaining price equilibrium and directly impact trading behavior.

Oracle data - Time-weighted price data from `pallet_ema_oracle::Oracles`. This data influences trading decisions and must remain accurate and tamper-resistant.

Oracle accumulator - Temporary storage for data before block finalization. This component ensures consistent processing of price data across block boundaries.

Code & logic assets

Stableswap implementation - The code implementing the invariant model in `hydra_dx_math::stableswap`. This mathematical foundation must be implemented correctly to ensure fair and expected trading behavior.

EMA implementation - Code handling time-weighted average calculations in `hydra_dx_math::ema`. This logic ensures accurate price averaging over time.

Fee calculation code - Implementation for determining and distributing trading fees in `pallet_stableswap::calculate_target_fee.` This code directly impacts protocol economics and must function correctly.

Rebasing handler code - Implementation for detecting and handling token supply changes `pallet_stableswap`. This logic ensures accurate accounting during supply adjustments.

Control mechanisms

Parameter setting authority - The capability to modify protocol parameters like fees and amplification via `update_pool_fee` and `update_amplification` functions. This capability must be carefully controlled and monitored.

Emergency controls - The mechanisms allowing trading to be halted in emergencies through `set_asset_tradable_state` and other emergency functions. These safeguards must be accessible when needed but protected from abuse.

Upgrade mechanisms - The infrastructure allowing protocol code to be modified. These mechanisms directly impact protocol evolution and security.

Access controls - The systems controlling who can perform privileged operations, implemented through `Origin` checks and permission validation in functions like `create_pool`. These controls define the security perimeter for sensitive protocol functions.

Stakeholders/Potential Threat Actors

The following outlines the various stakeholders or potential threat actors that interact with the system.

Privileged actors

AuthorityOrigin (stableswap) - Can call `create_pool`, `update_pool_fee`, and `update_amplification`. This role has significant control over pool creation and parameters.

UpdateTradabilityOrigin - Can modify asset tradability via `set_asset_tradable_state`. This role controls which assets can be traded.

AuthorityOrigin (ema-oracle) - Can add or remove asset pairs from the oracle. This role influences price discovery mechanisms.

Rebalance authority - If manual intervention is needed, the protocol may need an on-chain role to execute emergency rebalancing. This addresses extreme market conditions.

Rebase-oriented oracle operators - Ensuring accurate price data for rebasing tokens. These specialized operators handle dynamic token supplies.

External actors & market participants

BifrostOrigin - Special origin authorized to update Bifrost oracle data. This provides external price information.

TargetPegOracle - External price oracle providing asset data. This influences pool pricing.

Arbitrage bots - Exploiting price differences between Hydration pools and external markets. These help maintain price efficiency. Bots profiting from price discrepancies across parachains via XCM. These operate in the cross-chain environment.

MEV bots - While Polkadot lacks traditional MEV, front-running can occur via collators modifying transaction ordering. These extract value from transaction ordering.

Bridge operators - Entities managing cross-chain stablecoin transfers for non-native Polkadot assets. These facilitate cross-chain asset movement.

Users & economic actors

Users may be at risk of attacks such as social engineering, phishing, unauthorized access to wallets, or receiving fraudulent information. Users may fall into various categories:

Traders - Users swapping assets via the stableswap pool. These participants execute sell and buy extrinsics to exchange tokens, potentially exploiting price inefficiencies and driving the pool toward equilibrium. Their trading activities directly impact price discovery and generate fees for the protocol and LPs.

Liquidity providers (LPs) - Users adding/removing liquidity via `add_liquidity` and related extrinsics. These are the core participants providing trading depth. LPs may provide various types of assets, including:

- Standard tokens received directly from external sources
- Rebasing tokens (similar to AAVE's "A" tokens) which they received by providing liquidity to lending protocols
- Wrapped or bridged assets from other chains

Many LPs participate in yield farming strategies, where they actively manage their liquidity positions across multiple protocols to maximize returns through:

- Trading fees from the stableswap pools
- Protocol incentive rewards (token emissions)
- Secondary yield from deposited assets (such as rebasing tokens that accrue interest)
- Strategic liquidity reallocation based on changing APYs across the DeFi ecosystem

This creates a multi-layered liquidity provision ecosystem where the same user may be an LP across multiple protocols (lending, swapping, etc.), compounding their exposure and risk profiles. The interaction between these different LP positions (especially with rebasing tokens that change in quantity over time) introduces complex security considerations for pool accounting and share token valuation.

Protocol DAOs & governance participants - Stakeholders voting on fee structures, incentives, and security upgrades. These influence protocol evolution.

Privileged roles & risky counterparties

Pool curators/managers - Entities managing parameters like amplification factors. These have direct influence over pool behavior.

Whitelist & KYC gatekeepers - If pools enforce whitelisting, someone must approve addresses. These control access to the protocol.

Supply chain

The technical supply chain includes libraries, dependencies, and compiler infrastructure. If compromised, these components could introduce vulnerabilities even if the protocol's core code is secure. Historical precedents like the Vyper compiler vulnerability that affected Curve Finance demonstrate the impact of supply chain compromises.

Assumptions

The following outlines various assumptions upon which the system's functioning is predicated.

Pool configuration invariants

Pool composition limit - Maximum of 5 assets (`MAX_ASSETS_IN_POOL`) must be enforced per pool. Violation could lead to excessive computational complexity or gas costs.

Amplification constraints - All amplification values must remain within defined AmplificationRange (`NonZeroU16`). Values outside this range could destabilize pools or enable exploitation.

Token standardization - All assets in a pegged pool must maintain identical decimal places for accurate price calculation. Inconsistent decimals could create precision errors exploitable by attackers.

LP token representation - Share tokens must always accurately reflect proportional ownership of the pool, accounting for rebasing events. Inaccurate representation would lead to unfair value distribution and manipulable states which could be exploited by attackers.

Liquidity operation constraints

Initial liquidity provision - First LP must establish minimum viable liquidity across all pool assets simultaneously. Imbalanced bootstrapping could create exploitable initial conditions.

Liquidity addition mechanism - Subsequent liquidity additions are restricted to maintain pool balance. Unrestricted additions could manipulate pool ratios.

Withdrawal flexibility - LPs should maintain ability to withdraw either proportionally across all assets or selectively from specific assets. Restricted withdrawals would harm user experience.

Withdrawal impact limitations - Large withdrawals must not destabilize pool equilibrium beyond acceptable thresholds. Unlimited withdrawals could trigger liquidity crises.

Oracle & data integrity requirements

Oracle price boundaries - All price updates via `update_bifrost_oracle` must remain within `MaxAllowedPriceDifference`. Updates could indicate manipulation or extreme market conditions.

External data validation - `TargetPegOracle` must provide consistently accurate data within `max_peg_update` constraints. Inaccurate external data would compromise pricing.

Temporal consistency - `BlockNumberProvider` must ensure correct timestamps for EMA decay calculations. Inconsistent timing would disrupt time-weighted averaging.

Rebase detection - The protocol must detect and account for all rebasing events before trade execution. Missed rebase events would create exploitable accounting discrepancies.

Mathematical & operational safety

Minimum viable operations - `MinTradingLimit` and `MinPoolLiquidity` must be enforced to prevent dust attacks. Subminimal operations would enable attacks or burden the system.

Account protection - Pool accounts must maintain whitelisted status to prevent unintended asset removal. Compromised accounts would threaten user funds.

Value constraints - All calculations involving shares and tokens must prevent negative or implausible values. Mathematical edge cases could be exploited if not properly bound.

Rebase boundaries - Upper and lower limits on recognized rebase percentages must be enforced to mitigate volatility risks. Extreme rebases without limits could destabilize pools.

Emergency controls - Circuit breakers must activate during extreme market conditions affecting rebasing tokens. Without these safeguards, market disruptions could cause permanent damage.

Numerical safety - Calculations must not overflow, underflow, or divide by zero under any input conditions. Mathematical failures would corrupt the state or enable exploits.

Invariant preservation - The stableswap invariant must be maintained throughout all operations. Breaking this fundamental property would compromise pricing and enable value extraction.

Liquidity incentivization - While by design, the stable swap strives to reduce certain economic attack vectors, such as liquidity removal fee, liquidity incentives may render these attack vectors profitable again.

Network & integration assumptions

Blockchain reliability - The underlying blockchain must provide reliable transaction finality. Network issues would create inconsistent protocol state.

Parachain stability - The protocol depends on reliable parachain operations and cross-chain communication. Network disruptions would affect critical protocol functions.

Resource estimation - Extrinsic weights must accurately reflect computational costs. Inaccurate estimates would enable DoS attacks or economic losses.

Integration boundaries - As the protocol connects with other DeFi components or protocols, these integrations must respect security boundaries. Improper integration would introduce vulnerabilities.

Protocol composability - Interactions with external protocols must be secure and predictable. Unexpected behavior in connected protocols would affect system stability.

Threat Model

Process Applied

The process performed to analyze the system for potential threats and build a comprehensive model is based on the approach first pioneered by Microsoft in 1999 that has developed into the STRIDE model (https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)).

Whilst STRIDE is aimed at traditional software systems, it is generic enough to provide a threat classification suitable for blockchain applications with little adaptation (see below).

The result of the STRIDE classification has then been applied to a risk management matrix with simple countermeasures and mitigations suitable for blockchain applications.

STRIDE Interpretation in the Blockchain Context

STRIDE was first designed for closed software applications in permissioned environments with limited network capabilities. However, the classification provided can be adapted to blockchain systems with small adaptations. The below table highlights a blockchain-centric interpretation of the STRIDE classification:

Spoofing	In a blockchain context, the authenticity of communications is built into the underlying cryptographic public key infrastructure. However, spoofing attack vectors can occur at the off-chain level and within a social engineering paradigm. An example of the former is a Sybil attack where an actor uses multiple cryptographic entities to manipulate a system (wash-trading, auction smart contract manipulation, etc.). The latter usually consists of attackers imitating well-known actors, for instance, the creation of an impersonation token smart contract with a malicious implementation.
Tampering	Similarly to spoofing, tampering of data is usually not directly relevant to blockchain data itself due to cryptographic integrity. It can still occur though, for example through compromised developers of the protocol that have access to deployment keys or through supply chain attacks that manages to inject malicious code or substitutes trusted software that interacts with the blockchain (node software, wallets, libraries).
Repudiation	Repudiation, i.e. the ability of an actor to deny that they have taken action is usually not relevant at the transaction level of blockchains. However, it makes

	sense to maintain this category, since it may apply to additional software used in blockchain applications, such as user-facing web services. An example is the claim of a loss of a private key and hence assets.
Information Disclosure	Information disclosure has to be treated differently at the blockchain layer and the off-chain layer. Since the blockchain state is inherently public in most systems, information leakage here relates to data that is discoverable on the blockchain, even if it should be protected. Predictable random number generation could be classified as such, in addition to simply storing private data on the blockchain. In some cases, information in the mempool (pending/unconfirmed transactions) can be exploited in front-running or sandwich attacks. At the off-chain layer, the leakage of private keys is a good example of operational threat vectors.
Denial of Service	Denial of service threat vectors translates directly to blockchain systems at the infrastructure level. At the smart contract or protocol layer, there are more subtle DoS threats, such as unbounded iterations over data structures that could be exploited to make certain transactions not executable.
Elevated Privileges	Elevated privilege attack vectors directly translate to blockchain services. Faulty authorization at the smart contract level is an example where users might obtain access to functionality that should not be accessible.

STRIDE Classification

The following threat vectors have been identified using the STRIDE classification, grouped by components of the system.

	Spoofing	**Tampering**	**Repudiation**	**Information Disclosure**	**Denial of Service**	**Elevated Privileges**
Governance operations	Proposals manipulation through the social engineering	Tampering with governance processes to push malicious proposals	Denial of malicious governance actions by using multisigs or anonymous actors	-	Blocking governance operations via excessive proposals or sybil governance attacks	Amplification parameter manipulation for rebasing pools

	Spoofing	**Tampering**	**Repudiation**	**Information Disclosure**	**Denial of Service**	**Elevated Privileges**
Privileged role exploits	Impersonation of privileged roles (e.g., `AuthorityOrigin`) to alter amplification parameters Creating malicious smart contracts that mimic legitimate ones Creating malicious tokens for registration in pool	Tampering with compliance data to allow restricted asset Upgrade of Substrate pallets with malicious or malfunctioning code Altering mathematical models (e.g., amplification values) to distort execution	-	Public exposure of governance participants leading to targeted attacks	Disabling or pausing the protocol	Unauthorized regulatory rule changes enforced via privileged access Unauthorized access to restricted actions Unauthorized changes in fee structures impacting economic stability Elevating fee structures or governance parameters for personal gain Gaining unauthorized access to system-wide emergency controls
Oracle manipulati	Manipulated oracles	Malicious updates to	-	Oracle-internal price	Spamming oracles	Unauthorized oracle

	Spoofing	**Tampering**	**Repudiation**	**Information Disclosure**	**Denial of Service**	**Elevated Privileges**
on	feeding incorrect data into EMA-based systems Impersonating legitimate oracles to feed manipulated price data	on-chain oracle values altering price calculations		inconsistency attack	with excessive updates to block genuine transactions	price updates manipulating trade execution
Economic attacks	Spoofed AMM calculations causing incorrect trade pricing Market manipulation, especialy with respect to incentive tokens	Tampering with liquidity incentives or misrepresenting pool rewards Tampering the rebasing token supply to manipulate pool Exploiting the liquidity mining reward mechanism by manipulating reward parameters to extract disproportionate benefits Manipulati	Claiming inability to remove liquidity due to technical errors through liquidity withdrawal Fear, Uncertainty, Doubt (FUD) attack	Cross-chain rebasing token arbitrage Collator transaction ordering manipulation Cross-parachain transaction timing attack Exploiting publicly available pricing discrepancies to execute risk-free profit trades by taking advantage of information asymmetry	Draining liquidity pools by forcing slippage through rapid trades Intentionally draining a protocol of liquidity to crash token prices Coordinating the movement of liquidity out of targeted pools to deliberately destabilize the pool's operational state	Bypassing liquidity restrictions to withdraw excessive amounts

	Spoofing	**Tampering**	**Repudiation**	**Information Disclosure**	**Denial of Service**	**Elevated Privileges**
		ng the timing of rebase events to alter token supply adjustments in a way that favors the attacker Leveraging the interaction between amplification parameters and rebase events to induce a compounded destabilization of pool balances Forcing an unintended acceleration of the rebase rate to trigger rapid and destabilizing changes in token supply		Disrupting the synchronization between rebase events and oracle updates to create erroneous pricing signals		
Externally owned accounts	Lost account	Pharming/ phishing/ social engineering	Compromised account	Private key leakage Doxxing/identity disclosure	DOS of infrastructure Blacklisted account evasion:	Compromised private key

	Spoofing	Tampering	Repudiation	Information Disclosure	Denial of Service	Elevated Privileges
					ensure no blacklisted accounts can operate within pools (such as providing liquidity, e.g., assets might be frozen)	
Deposit and withdrawal of funds from Parachain	Fake identities submitting deposit/withdrawal transactions	Modification of transaction data to redirect funds	Absence of an audit trail allowing users to deny submitted transactions	-	Sequencers censoring or delaying transactions	Unauthorized access to modify transaction parameters or bypass validations
Peg update and liquidity operations	Fake pool creation requests or injection of bogus liquidity events	Unauthorized modification of pool parameters (fees, amplification, peg values)	Lack of traceability of changes in pool parameters and liquidity operations	-	Overloading the pool with rapid liquidity operations to disrupt invariants	Unauthorized modifications due to bypassing permission checks

Mitigation Matrix

The following mitigation matrix describes each of the threat vectors identified in the STRIDE classification above, assigning an impact and likelihood and suggesting countermeasures and mitigation strategies. Countermeasures can be taken to identify and react to a threat, while mitigation strategies prevent a threat or reduce its impact or likelihood.

Governance Operations

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Governance proposal manipulation Attackers submit malicious proposals via social engineering, influencing voting outcomes unfairly.	Medium	Medium	Require minimum stake for governance proposals.	Audit governance activity and track proposal history.
Governance process tampering Attackers interfere with the governance process, bypassing checks to introduce harmful proposals.	High	Medium	Enforce time delays on governance actions.	Use decentralized review committees for proposals.
Governance action repudiation Attackers use multisig setups or anonymity to deny accountability for malicious governance actions.	Medium	Low	Log and publish all governance actions publicly.	Enforce KYC for privileged governance roles.
Governance denial of service (DoS) Flooding the governance system with spam proposals or Sybil attacks to prevent legitimate decision-making.	High	High	Implement proposal fee or minimum stake to submit proposals.	Monitor for spam activity and rate-limit proposals.
Amplification parameter	High	Low	Enhanced security	Implement

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Manipulation for rebasing pools Attackers influence governance to set inappropriate amplification parameters for pools containing rebasing tokens, creating pricing curves that can be exploited due to their misalignment with actual token behavior.			reviews for amplification parameter changes involving rebasing token pools.	parameter boundaries specifically designed for rebasing token pools that limit potential manipulation.

Privileged Role Exploits

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Privileged role spoofing Attackers impersonate privileged accounts (e.g., `AuthorityOrigin`) to manipulate governance or AMM parameters.	High	Medium	Enforce multi-sig authentication, role-based access control.	Monitor privileged accounts for anomalies, require transaction approvals.
Malicious smart contract deployment Attackers deploy smart contracts that mimic legitimate protocol functions to deceive users.	High	Low	Require multi-party approval for governance upgrades.	Enforce security audits before upgrades.
Fake token registration Attackers register counterfeit tokens in the pool to trick users or distort price calculations.	High	Medium	Require token verification before registration.	Whitelist known trusted assets, block unverified token additions.
Bypassing compliance	High	Medium	Use decentralized	Regular

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Controls Attackers alter compliance mechanisms to allow restricted or blacklisted assets.			identity verification for compliance enforcement.	compliance audits, enforce compliance at the protocol level.
Malicious code injection in governance upgrades Attackers introduce malicious pallet updates to alter protocol execution or governance parameters.	High	Low	Require multi-party approval for governance upgrades.	Enforce security audits before upgrades.
Mathematical model manipulation Attackers tamper with key economic variables (e.g., amplification factors) to exploit AMM mechanics.	High	Medium	Set parameter modification limits, enforce multi-sig approval.	Use automated validation checks before applying changes.
Doxxing of governance participants Governance participants' identities are leaked, leading to potential coercion or external attacks.	Medium	Low	Allow pseudonymous governance with encrypted verification.	Limit access to personally identifiable information (PII).
Protocol freeze attack Attackers gain access to privileged controls to pause trading or liquidity withdrawals, disrupting the protocol.	High	Medium	Limit emergency controls, require multi-party approval for pauses.	Monitor for abnormal administrative actions.
Regulatory bypass via privileged access Attackers change governance rules (e.g., modifying KYC or	High	Medium	Enforce role-based access control for policy changes.	Regularly audit regulatory actions.

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
tradability policies) for personal benefit.				
Unauthorized access to protocol controls Attackers escalate privileges to access restricted protocol functions, such as minting tokens or modifying fees.	High	Medium	Implement strict privilege separation.	Monitor and log all administrative actions.
Fee structure manipulation Attackers modify transaction or liquidity fees, affecting the protocol's economic balance.	High	Medium	Require governance approval for fee changes.	Monitor fee adjustments for unusual changes.
Governance parameter exploitation Attackers adjust governance settings (e.g., voting power, tradability rules, fees) to gain an unfair economic advantage.	High	Medium	Use time-locked parameter changes.	Enforce voting cooldown periods.
Compromising emergency controls Attackers exploit emergency control mechanisms to trigger forced liquidations, pauses, or shutdowns.	High	Medium	Restrict emergency controls to multi-party governance.	Log all emergency actions with real-time alerts.

Oracle Manipulation

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Oracle price distortion	High	High	Use multiple	Implement

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Attack Attackers manipulate oracles to inject fake price data, affecting AMM execution.			independent oracles.	circuit breakers on extreme price changes.
Oracle spoofing attack Attackers create fake oracles to inject incorrect pricing information into the system.	High	Medium	Whitelist trusted oracle providers.	Use cryptographic signatures for oracle data validation.
Malicious oracle data injection Attackers submit false updates to on-chain oracles, distorting trade execution and market behavior.	High	Medium	Use cross-validation across multiple data sources.	Limit update frequency for sensitive oracle inputs.
Oracle-internal price inconsistency attack Attackers exploit inconsistencies between external oracle prices and internal pool calculations for rebasing tokens, especially after liquidity changes, to extract value through arbitrage.	High	Medium	Implement synchronized price update mechanisms that ensure internal calculations and oracle data remain aligned, especially after liquidity operations.	Deploy monitoring systems that temporarily increase fees or restrict trades when price inconsistencies are detected.
Oracle denial of service (DoS) Attackers flood the oracle system with frequent updates to disrupt real price discovery.	High	Medium	Rate-limit oracle updates.	Monitor for excessive update activity.
Privileged oracle price manipulation	High	Medium	Use decentralized oracle governance.	Audit oracle price updates

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Attackers gain unauthorized control over oracle feeds, adjusting price updates to favor specific trades.				regularly.

Economic Attacks

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
AMM calculation manipulation Attackers modify AMM formulas or spoof trades to create artificial price shifts.	High	Medium	Use circuit breakers for extreme trade deviations.	Validate price calculations with multiple sources, monitor trade patterns for manipulation.
Liquidity incentive fraud Attackers misrepresent reward structures to lure liquidity providers under false terms.	Medium	Medium	Publish transparent reward distribution rules.	Audit reward calculations periodically.
Rebasing attack on pool liquidity Attackers exploit rebasing mechanics to burn, inflate, or misprice specific assets in the pool to profit from liquidity incentive distribution distortions.	High	Medium	Implement safeguards for rebasing token handling, especially in conjunction with liquidity incentive token distributions.	Monitor rebasing changes and ensure liquidity incentive program aligns with pool liquidity management.
Liquidity withdrawal Fear, Uncertainty, Doubt (FUD) attack Attackers spread false claims about liquidity	Medium	Medium	Provide clear liquidity status updates.	Use fact-checking mechanisms to counter misinformation

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
issues to induce panic selling or freezing of funds.				.
Cross-chain rebasing token arbitrage Attackers exploit timing differences in how rebasing tokens are valued across different parachains, using cross-chain transactions to profit from temporary price discrepancies.	Medium	Medium	Implement cross-chain price verification for rebasing tokens before executing trades.	Apply increased fees for cross-chain trades involving rebasing tokens when significant price movement has recently occurred.
Collator transaction ordering manipulation Collators manipulate transaction ordering within blocks to create favorable conditions for specific trades, particularly around rebasing token operations.	Medium	Low	Implement fair ordering mechanisms and transaction batching protocols.	Monitor collator behavior patterns and implement protocol-level safeguards against transaction reordering.
Cross-parachain transaction timing attack Attackers observe cross-chain transactions via XCM and position trades to benefit from temporary price impacts, particularly with rebasing tokens.	Medium	Low	Implement price impact limits for cross-chain operations.	Monitor for correlated trading patterns around XCM messages.
Slippage exploit attack Attackers execute rapid trades to force high slippage and drain liquidity.	High	Medium	Set slippage tolerance limits and implement circuit breakers.	Monitor trade patterns for excessive slippage activity.

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Liquidity crash attack Attackers remove large amounts of liquidity, causing token price collapses.	High	Medium	Require phased liquidity withdrawals for large transactions.	Monitor liquidity withdrawals in real-time.
Liquidity drain via bypass exploit Attackers circumvent withdrawal limits to extract excessive liquidity.	High	Medium	Restrict maximum withdrawal amounts within a timeframe.	Use smart contract monitoring for unauthorized bypass attempts.
Liquidity mining extraction attack Attackers deposit liquidity to earn token incentives, then strategically remove liquidity when the value of earned incentives exceeds the removal fee, even if it negatively impacts the pool.	High	High	Implement time-locked incentives that vest gradually and design removal fees that scale with the volatility impact on the pool.	Monitor for patterns of cyclic liquidity provision and withdrawal that correlate with incentive distributions.
Rebasing incentive arbitrage Attackers exploit misalignment between rebasing token growth and incentive distribution mechanisms, capturing both yield from rebasing tokens and protocol incentives before removal fees can balance the equation.	Medium	High	Design incentive structures that account for the underlying yield of rebasing tokens and reduce incentives for pools containing tokens with high natural yield.	Implement withdrawal fees that dynamically adjust based on recent rebasing events and incentive distributions.
Strategic liquidity migration Attackers monitor multiple pools and	Medium	Medium	Implement protocol-wide cooldown periods that apply across all pools and design	Track addresses that frequently migrate liquidity and

Threat Vector	**Impact**	**Likelihood**	**Mitigation**	**Countermeasures**
migrate liquidity to capture short-term incentives across different pools, creating liquidity instability without paying appropriate fees.			cross-pool incentive mechanisms that discourage rapid migration.	apply higher fees to these addresses.
Rebasing timing exploitation Attackers strategically time trades around rebasing events to profit from temporary value discrepancies before pool calculations update.	High	Medium	Implement synchronous rebasing updates across all pool operations.	Monitor for trading patterns correlated with rebasing schedules.
Amplification-rebase compound effect Attackers exploit the combined effect of amplification parameter changes and rebasing events, which create compounded pricing distortions.	High	Medium	Implement cooling periods when both amplification and significant rebasing occur simultaneously.	Automatically adjust fee parameters during combined events.
Rebase rate acceleration attack Attackers manipulate market conditions to accelerate rebase rates in Aave v3 (e.g., by influencing utilization rates), then exploit your protocol's rebasing update mechanism.	Medium	Low	Implement upper bounds on recognized rebase rates within a time period.	Monitor for unusual rebase acceleration patterns and temporarily increase fees.

Externally Owned Accounts

Threat Vector	**Impact**	**Likelihood**	**Mitigation**	**Countermeasures**
Lost account	Low	Low	Have a clear policy	Enforce policy

Threat Vector	**Impact**	**Likelihood**	**Mitigation**	**Countermeasures**
The attacker claims that they own an account and the access to the private key has been lost.			not to refund lost assets or restore privileges.	and strictness.
Pharming/ phishing/ social engineering The attacker may manipulate users' or development teams' wallets, lure them to malicious front-ends, manipulate DNS records, or use social engineering to trick users/teams into signing manipulated transactions transferring funds/permissions.	Medium	Medium	Educate users and team, protect DNS records, create awareness, offer blacklists with malicious sites, create activity on social channels to build reputable channels, deploy front-ends on IPFS or other decentralized infrastructure.	Monitor all systems, monitor communities and impersonations/malicious copies of official channels, communicate attempted pharming/phishing/social engineering, have processes in place to recover from DNS manipulation, attacks on front-ends quickly.
Compromised account The attacker claims they are a victim of scapegoating, denying responsibility for their attack.	Low	Low	Have a clear policy not to refund lost assets or restore privileges.	Enforce policy and strictness.
Private key leakage Private keys are accidentally shared or logged.	Medium	Medium	Educate users and team, ensure private keys are properly handled in wallet	Monitor all systems, have a policy in place to rotate

Threat Vector	**Impact**	**Likelihood**	**Mitigation**	**Countermeasures**
			software, use hardware wallets/air-gapped devices, security keys, multi-signatures.	keys.
Doxxing/identity disclosure Private data such as the off-chain identity of users disclosed.	Low	Medium	Educate users and team, no storage of identity/sensible data in databases that link identity to account addresses, follow privacy regulations and guidelines.	-
DOS of infrastructure DOS attack on a validator, relayer, an end user's device/network or on the blockchain node they interact with.	Low	Low	Educate users and team, use firewalls, sentry architecture, load balancers, VPNs.	Monitor infrastructure, and have processes in place to elastically provision and deploy additional resources.
Blacklisted account evasion Attackers attempt to bypass blacklisting mechanisms to operate within pools, such as providing liquidity, withdrawing funds, or executing trades using restricted assets. This can include circumventing compliance rules or using proxies to disguise blacklisted accounts.	Medium	Medium	Enforce blacklist checks before transactions.	Use monitoring systems to flag blacklisted accounts.
Compromised private key	Medium	Medium	Educate users and team.	Monitor all systems, have a policy in

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Private keys may be compromised.				place to rotate keys.

Deposit and withdrawal of funds from Parachain

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Fake identities Impersonation of legitimate users to submit unauthorized deposit/withdrawal transactions.	High	Medium	Enforce robust authentication by validating digital signatures on transaction submissions and verifying the origin via runtime primitives.	Require cryptographic signature validation and ensure origin verification through the runtime's built-in verification mechanisms.
Modification of transaction data Altering transaction parameters during execution to redirect funds.	High	Medium	Apply cryptographic checks (signatures, checksums) on transaction payloads and use atomic, transactional extrinsics to ensure complete state updates.	Utilize the `#[transactional]` attribute on sensitive functions and verify transaction integrity before state mutation.
Lack of auditability Absence of an auditable trail, allowing parties to deny submitted transactions.	Medium	Low	Emit comprehensive events for all state-changing operations to provide an immutable audit trail.	Maintain on-chain logs mapping each transaction to an authenticated account and use event logs for

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
				post-incident audits.
Transaction censorship Flooding the governance system with spam proposals or Sybil attacks to prevent legitimate decision-making.	High	Medium	Implement decentralized consensus mechanisms and monitor block inclusion and apply weight limits to throttle excessive transaction submissions.	Use fallback mechanisms to detect and recover from censored/delayed transactions and enforce weight limits via runtime configuration.
Unauthorized access escalation Unauthorized access enabling modification or bypass of transaction validations.	High	Low	Strictly enforce role-based access controls using predefined origins (e.g. `T::AuthorityOrigin`) for all critical operations.	Maintain on-chain logs mapping each transaction to an authenticated account and use event logs for post-incident audits.

Peg update and liquidity operations

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Fake pool creation Creating pools using not legitimate assets or by an unauthorized entity.	High	Medium	Validate pool creation and liquidity events by checking asset registration and enforcing pool creation via `AuthorityOrigin`.	Ensure that pool creation extrinsics require valid authorization and proper initial liquidity deposits.
Unauthorized parameter	Medium	Medium	Apply atomic,	Enforce the

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
modification Attackers modify the protocol's parameters influencing its behaviour.			transactional updates for liquidity operations and parameter changes.	use of `#[transactional]` macro on state-modifying extrinsics.
Repudiation Lack of auditable information leading to disputes challenging the protocol.	Medium	Low	Emit comprehensive events for every critical change enabling traceability of modifications.	Maintain an audit log with event data linked to authenticated origins and use those logs for any relevant dispute resolutions.
Denial of service Rapid liquidity changes influencing the health of the pool, including the invariants.	High	Medium	Enforce minimum liquidity thresholds and use robust invariant checks to prevent state disruption.	Implement weight limits on rapid liquidity operations. Throttling excessive extrinsics to protect state consistency.
Elevated privileges Unauthorized access to critical and privileged operations.	High	Low	Enforce strict role-based access for operations that modify critical pool parameters using appropriate Origins.	Conduct regular security reviews and audits. Require multi-signature or additional authentication for elevated operations.