Security Audit Report

KILT Bonding Curve Pallet

v1.0

March 27, 2025

Table of Contents

Table of Contents **2**
License **3**
Disclaimer **4**
Introduction **5**
Purpose of This Report 5
Codebase Submitted for the Audit 5
Methodology 7
Functionality Overview 7
How to Read This Report **8**
Code Quality Criteria **9**
Summary of Findings **10**
Detailed Findings **12**
1. Potential loss of funds due to can_deposit error during the refund process 12
2. Configurations of PolynomialParameters can lead to overflow 12
3. The MaxConsumers constraint limits multi-asset pool usability 13
4. Incomplete asset cleanup in finish_destroy extrinsic may leave residual tokens 14
5. Centralization risks 15
6. Possibility of setting different management teams for assets within the same pool 16
7. Unused _owner parameter in reset_team function 16
8. Missing validations of currencies list during pool creation 17
9. Minimal error handling in try_from implementation 17
10. Redundant defensive assertion in refund_account function 18
11. Inconsistent pool lock state 18
12. Possible optimization in polynomial curve 19
13. Manager change allowed in destroying state pools 19
14. Contracts should implement a two-step ownership transfer 20
15. Unresolved TODO and FIXME comments in the codebase 21
16. Missing event emission for reset_team 21
17. Redundant manager and team updates are allowed 21
18. Dependencies are subject to publicly known vulnerabilities 22
Appendix: Test Cases **24**
1. Test case for “The MaxConsumers constraint limits multi-asset pool usability” 24
2. Test cases for “Configurations of PolynomialParameters can lead to overflow” 26

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

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 BOTLabs GmbH i. L. to perform a security audit of KILT Bonding Curve Substrate Pallet.

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/KILTprotocol/kilt-node
Scope	The scope is restricted to the changes applied in the following pull requests: • https://github.com/KILTprotocol/kilt-node/pull/764 reviewed at commit `118ae6f6065324702f006354e52eb602fb5d23bd`, base branch at `e5eb9160560f838c2b3e375686b409552990d858`. • https://github.com/KILTprotocol/kilt-node/pull/834 reviewed at commit `9d6bab4718832874b3cca2518e1cb77e7fab71b0`,

	base branch at `118ae6f6065324702f006354e52eb602fb5d23bd`.
Fixes verified at commit	`0360ad2101b9dd6a765f056306360c880b991b9f` 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

The KILT Bonding Curve Pallet implements a bonding curve mechanism to manage the issuance, exchange, and pricing of tokens within the KILT Protocol. It allows tokens to be minted and burned dynamically based on a predefined mathematical curve, ensuring that prices adjust algorithmically according to supply and demand.

The pallet supports different bonding curve formulas, making it adaptable to various economic models within the network.

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	High	The client provided detailed documentation outlining the specifications of the intended system behavior.
Test coverage	Medium-High	`cargo tarpaulin` reports a test coverage of `89.38%` for the `pallet-bonded-coins` package

Summary of Findings

No	Description	Severity	Status
1	Potential loss of funds due to `can_deposit` error during the refund process	Major	Resolved
2	Configurations of `PolynomialParameters` can lead to overflow	Minor	Acknowledged
3	The `MaxConsumers` constraint limits multi-asset pool usability	Minor	Resolved
4	Incomplete asset cleanup in `finish_destroy` extrinsic may leave residual tokens	Minor	Acknowledged
5	Centralization risks	Minor	Partially Resolved
6	Possibility of setting different management teams for assets within the same pool	Minor	Resolved
7	Unused `_owner` parameter in `reset_team` function	Informational	Resolved
8	Missing validations of currencies list during pool creation	Informational	Resolved
9	Minimal error handling in `try_from` implementation	Informational	Acknowledged
10	Redundant defensive assertion in `refund_account` function	Informational	Resolved
11	Inconsistent pool lock state	Informational	Resolved
12	Possible optimization in polynomial curve	Informational	Resolved
13	Manager change allowed in destroying state pools	Informational	Resolved
14	Contracts should implement a two-step ownership transfer	Informational	Acknowledged
15	Unresolved `TODO` and `FIXME` comments in the codebase	Informational	Resolved
16	Missing event emission for reset_team	Informational	Resolved
17	Redundant manager and team updates are allowed	Informational	Acknowledged

18	Dependencies are subject to publicly known vulnerabilities	**Informational**	**Acknowledged**

Detailed Findings

1. Potential loss of funds due to `can_deposit` error during the refund process

Severity: Major

In `pallets/pallet-bonded-coins/src/lib.rs:1007-1143`, the `refund_account` extrinsic allows users to burn their bonded tokens and claim collateral proportionally.

In lines `1117-1126`, the implementation defines a guard that ensures that the `amount` is non-zero and that the collateral can be deposited in the `who` account. In case of failure, the extrinsic does not fail and returns `Ok`. The comment associated with this clause states:

"Funds are burnt but the collateral received is not sufficient to be deposited to the account. This is tolerated as otherwise, we could have edge cases where it's impossible to refund at least some accounts."

However, there are other potential failure scenarios for the `can_deposit` method beyond insufficient collateral. Specifically, if the collateral asset is marked as not sufficient and the caller has already reached the maximum number of allowed consumers (`MaxConsumers` limit), the `can_deposit` check will fail.

This scenario would result in the user irreversibly losing their funds, as the bonded tokens are burned, but the collateral cannot be deposited into their account.

Recommendation

We recommend implementing additional validation checks before burning bonded tokens to ensure that the `can_deposit` method will not fail due to `MaxConsumers` limitations.

Status: Resolved

2. Configurations of `PolynomialParameters` can lead to overflow

Severity: Minor

The `PolynomialParameters` struct, in `pallets/pallet-bonded-coins/src/curves/polynomial.rs:108-115`, defines parameters for the `CurveInput::Polynomial` curve, represented by the equation:

$$f(x) = \frac{m}{3}x^3 + \frac{n}{2}x^2 + ox$$

However, the implementation lacks validation to restrict values that could cause arithmetic overflow.

The `calculate_costs` function, defined in `pallets/pallet-bonded-coins/src/curves/polynomial.rs:138` responsible for computing costs for the polynomial curve, is susceptible to overflow due to the squared operations.

Consequently, depending on the values of `m`, `n`, `o`, and the number of currencies involved, the function may overflow during minting operations, potentially leading to a locked pool where no further tokens can be minted.

Test cases showcasing the issue are provided in Appendix.

Recommendation

We recommend enforcing boundary checks on `m`, `n`, and `o` to ensure their values remain within a safe range.

Status: Acknowledged

The client states that overflow only limits the maximum number of tokens that can be minted. They acknowledge that transparency regarding these implicit limits could be improved and have therefore added documentation in the Bonding Coin Specification. Additionally, they are exploring ways to make these limits more explicit and to clarify overflow conditions. However, they do not consider overflow a vulnerability at this time and assert that any fixed-precision system of this nature will inherently have technical limits that restrict the number of tokens that can be minted.

3. The `MaxConsumers` constraint limits multi-asset pool usability

Severity: Minor

In `pallets/pallet-bonded-coins/src/lib.rs:360-415`, the `create_pool` extrinsic enables users to create a new pool with support for up to `MAX_CURRENCIES` (`50`) different currencies. During this process, the function iterates through the specified currencies and invokes `Fungibles::create` with the `is_sufficient` parameter set to `false`. This configuration enforces the chain's existential deposit rules, requiring the system to verify if it can safely add a consumer reference to an account.

However, the `MaxConsumers` limit, set to `16`, restricts the number of non-sufficient assets an account can hold. This constraint prevents users from holding all fifty currencies in a pool and may lead to unexpected failures when minting tokens if the limit is exceeded.

As a result, pool designs that require a single user to manage more than eight currencies become impractical, significantly reducing the flexibility and usability of multi-asset pools.

A test case showcasing the issue is provided in Appendix.

Recommendation

We recommend revising the pool creation logic to allow certain assets to be marked as sufficient where appropriate.

Status: Resolved

4. Incomplete asset cleanup in `finish_destroy` extrinsic may leave residual tokens

Severity: Minor

In `pallets/pallet-bonded-coins/src/lib.rs:1255-1261`, the `finish_destroy` extrinsic allows permissionless completion of a pool's destruction, including asset removal and deposit refunds to the owner. The function calls `T::Fungibles::finish_destroy` to finalize asset destruction.

However, if the destruction process was initiated forcibly, there may still be tokens held by users.

In such cases, `finish_destroy` alone is insufficient, as the total supply has not been burned and it does not account for remaining balances in user accounts.

Recommendation

We recommend calling `T::Fungibles::destroyAccounts` before executing `T::Fungibles::finish_destroy`. This ensures that all remaining tokens are properly removed before finalizing the asset destruction.

Status: Acknowledged

The client states that calling `(force_)start_destroy` transitions all linked assets to a destroying state, allowing the assets pallet's permissionless `destroy_accounts` transaction to remove any residual accounts. They chose not to re-expose or wrap this functionality within their pallet to prevent unbounded transaction sizes that could exceed block limits, particularly for pools with numerous bonded currencies. As a result, pool destruction is a multi-step process that may involve multiple extrinsics.

To enhance clarity, they have added a "Pool Life Cycle" section to the Bonding Coin Specification, detailing the necessary steps.

5. Centralization risks

Severity: Minor

During the audit, multiple centralization concerns were identified within the bonding curve pallets.

While certain privileged roles, such as managers, are necessary for managing critical configurations, excessive control can undermine the system's trustlessness.

Key centralization risks include:

- **Excessive control by the pool manager**
    - The pool manager can initiate the refund process. Additionally, since the `refund_account` extrinsic applies for refunds proportionally across all bonded currencies, assuming equal value, the refund process can be abused to perform market manipulations.
    - The pool manager can trigger pool destruction and obtain any funds deposited into it by using `force_start_destroy` and skipping the refunding process.
    - The pool manager can impose a `Lock` on the pool.
    - The pool manager can assign or modify the currency management team.
    - The pool manager can freeze assets.
- **Root privileges**
    - The `ForceOrigin` requires `EnsureRoot`, meaning all administrative actions require root access. Relying on root centralizes power within a single entity.
    - The `ForceOrigin` can trigger pool destruction and obtain any funds deposited into it by using `force_start_destroy` and skipping the refunding process.

Recommendation

We recommend evaluating and documenting the centralization risks of the protocol.

Status: Partially Resolved

The client implemented a flag to enable a more granular configuration of manager privileges and added a section outlining centralization risks to enhance transparency and risk awareness.

6. Possibility of setting different management teams for assets within the same pool

Severity: Minor

The `reset_team` function implemented in `pallets/pallet-bonded-coins/src/lib.rs:474` is intended for setting the management team for a currency issued by a given pool.

However, since pools can contain multiple currencies, this function only updates the management team for a single currency at a time.

This creates a scenario where different entities can manage different currencies within the same pool, leading to inconsistent control.

Such a situation may result in unexpected behavior, including disruptions if one of the currencies is suddenly frozen, impacting overall pool operations.

Recommendation

We recommend modifying the implementation to ensure that a single entity is responsible for managing all currencies within a given pool.

Status: Resolved

7. Unused `_owner` parameter in `reset_team` function

Severity: Informational

In the `reset_team` implementation within the traits module in `pallets/pallet-bonded-coins/src/trait.rs:90`, the `_owner` parameter is never used in the function logic. The function simply ignores it and instead retrieves the owner from `AssetsPallet::<T, I>::owner(id.clone())`.

In `pallets/pallet-bonded-coins/src/lib.rs`, this function is called with `pool_id_account` as the owner parameter. There's no guarantee that `pool_id_account` will always equal the value retrieved from `owner(id)` call, causing incorrect assumptions in the code.

Recommendation

We recommend either utilizing the provided `_owner` parameter or removing it from the function definition to improve code clarity and prevent potential mismatches.

Status: Resolved

8. Missing validations of currencies list during pool creation

Severity: Informational

In `pallets/pallet-bonded-coins/src/lib.rs:360`, the `create_pool` extrinsic is responsible for creating a pool by taking a vector of currencies and their metadata and then generating the corresponding fungible assets.

However, while `NextAssetId` ensures unique asset IDs, there is no validation to prevent multiple currencies with identical metadata (names and symbols) from being created within the same pool. This could lead to user confusion when interacting with the pool, as multiple assets with identical identifiers may be indistinguishable.

Additionally, the extrinsic does not enforce a non-empty pool constraint, allowing the creation of empty pools that serve no functional purpose.

Recommendation

We recommend implementing validation to:

- Ensure currency metadata (names and symbols) are unique within a given pool.
- Reject empty pools by enforcing a minimum number of currencies required for pool creation.

Status: Resolved

9. Minimal error handling in `try_from` implementation

Severity: Informational

In `pallets/pallet-bonded-coins/src/curves/square-root.rs:116`, the `try_from` implementation uses an overly simplistic error type, returning an empty `()` value upon failure.

This approach provides no meaningful context about the nature of the conversion failure, making debugging and error handling more challenging for both developers and users.

Recommendation

We recommend replacing the empty unit error type with a custom error enum that provides more specific information about conversion failures, such as whether the problem occurred with parameter `m` or `n`, or the specific reason for the conversion failure.

Status: Acknowledged

The client states that error reporting for parameter validation can be improved, and they are considering the addition of more informative error messages and logging in the future.

However, they note that propagating these errors to callers is challenging without introducing a larger set of broad error cases, as pallet errors are implemented as simple enums that do not support carrying detailed error messages.

10. Redundant defensive assertion in `refund_account` function

Severity: Informational

In `pallets/pallet-bonded-coins/src/lib.rs:1078`, a `devensive_assert` check validating if `sum_of_issuances` is bigger or equal to `burnt` amount is performed.

However, this assertion is meaningless since `sum_of_issuances` has already been guaranteed to be at least equal to `burnt` in the code directly above it. The `burnt` value is explicitly added to `sum_of_issuances`, making it impossible to be bigger than the mentioned sum.

Recommendation

We recommend removing this redundant assertion to simplify the code.

Status: Resolved

11. Inconsistent pool lock state

Severity: Informational

In `pallets/pallet-bonded-coins/src/lib.rs:571-587`, the `set_lock` extrinsic allows a pool manager to lock a pool using a `Locks` object that determines whether minting and burning operations are disabled.

However, the function does not enforce validation to ensure that both `allow_mint` and `allow_burn` are set to `false`. As a result, a pool can be incorrectly marked as `PoolStatus::Locked` even when minting and burning are still allowed.

This leads to an inconsistent state where a pool appears locked while operations remain enabled. The expected behavior in such cases is for the status to remain `PoolStatus::Active`.

Recommendation

We recommend adding a validation check within the `set_lock` function to ensure that a pool is only transitioned to `PoolStatus::Locked` when both `allow_mint` and `allow_burn` are explicitly disabled.

Status: Resolved

12. Possible optimization in polynomial curve

Severity: Informational

The `calculate_costs` function in `pallets/pallet-bonded-coins/src/curves/polynomial.rs` can be optimized to reduce unnecessary computations.

The function calculates helper variables before using them to compute `term1` and `term2`, which rely on the `m` and `n` coefficients as multipliers.

However, if `m` or `n` is equal to zero, the corresponding term (`term1` or `term2`) will also evaluate to zero.

In such cases, performing these calculations is redundant and results in unnecessary computational overhead.

Recommendation

We recommend introducing pre-checks for `m` and `n` coefficients to bypass unnecessary calculations when their values are zero.

Status: Resolved

13. Manager change allowed in destroying state pools

Severity: Informational

In `pallets/pallet-bonded-coins/src/lib.rs:530`, the `reset_manager` function allows changing the pool manager. However, a manager change should not be permitted when the pool is in a non-live (destroying) state. Allowing this could lead to inconsistencies or unintended control transfers in pools that are being decommissioned.

Recommendation

We recommend introducing a check to prevent manager changes when the pool is not in a live state.

Status: Resolved

14. Contracts should implement a two-step ownership transfer

Severity: Informational

In `pallets/pallet-bonded-coins/src/lib.rs:525-547`, the `reset_manager` extrinsic allows the current manager to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to an incorrect address. More than that, transferring managership to a non-existing address or resigning manager duties, when the pool is locked, prevents the pool from ever being unlocked.

A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

1. The current manager proposes a new manager address that is validated.
2. The new manager account claims ownership, which applies the configuration changes.

At the same time, making the pool permissionless can be allowed as a single step by ensuring that it is unlocked at the same time.

Status: Acknowledged

The client states that they have decided against implementing two-step ownership changes for two primary reasons:

1. Assigning a `None` manager is a standard part of the lifecycle for unpermissioned pools, which is incompatible with a two-step transfer process.
2. Two-step ownership transfers are not common practice in similar pallets. The client believes that errors in ownership transfers are rare, better prevented at the application level, and can be addressed through chain governance mechanisms.

15. Unresolved `TODO` and `FIXME` comments in the codebase

Severity: Informational

The following instances of TODO comments were identified within the given scope of this audit (excluding tests):

- `pallets/pallet-bonded-coins/src/curves/mod.rs:169`
- `pallets/pallet-bonded-coins/src/lib:327`
- `pallets/pallet-bonded-coins/src/lib:1418`
- `pallets/pallet-bonded-coins/src/lib:1491`
- `runtimes/peregrine/src/weights/pallet_bonded_assets:46`

Totally, there are more than 40 `TODO` and `FIXME` comments throughout the codebase.

Recommendation

We recommend resolving or removing the given `TODO` and `FIXME` comments.

Status: Resolved

16. Missing event emission for `reset_team`

Severity: Informational

In `pallets/pallet-bonded-coins/src/lib.rs:474`, the `reset_team` extrinsic updates the `admin` and `freezer` roles.

However, the update is not announced to potential on-chain listeners, as no event is emitted upon a successful execution. This lack of notification can lead to inconsistencies in off-chain tracking and governance monitoring.

Recommendation

We recommend declaring and emitting an event whenever an administrative team update occurs.

Status: Resolved

17. Redundant manager and team updates are allowed

Severity: Informational

The `reset_team` and `reset_manager` extrinsics defined respectively in `pallets/pallet-bonded-coins/src/lib.rs:474-503` and `pallets/pallet-bonded-coins/src/lib.rs:527-547`, allow updates to administrative roles but do not validate whether the new values differ from the existing ones.

- `reset_team`: Updates the `admin` and `freezer` roles using the `PoolManagingTeam` structure.
- `reset_manager`: Allows setting a new manager via an optional `new_manager` account ID.

Since no validation is performed, redundant updates can be executed unnecessarily, consuming resources without changing the system state.

Recommendation

We recommend implementing a validation check to reject redundant updates when the new values match the existing ones. In the case of `reset_manager`, the current manager must be the signer, making this validation straightforward.

Status: Acknowledged

The client states that resource allocation is governed by economic principles, such as transaction fees, and that resources are consumed regardless of whether a transaction results in an error or a state change. Also, they state that rejecting a transaction simply because the system is already in the desired state is suboptimal from a user experience perspective.

18. Dependencies are subject to publicly known vulnerabilities

Severity: Informational

The project dependencies are not up-to-date and contain publicly known Rust vulnerabilities:

1. Infinite loop based on network input (rustls)
2. Timing variability (curve25519-dalek)
3. Punycode labels that do not produce any non-ASCII when decoded (idna)

Additionally:

- Dependencies `parity-util-mem` and `parity-wasm` are deprecated
- The version of `parity-scale-codec` in use is `3.1.5` and was released in June 2022. The latest version is `3.7.4`.

Recommendation

We recommend updating the aforementioned dependencies and regularly performing automated dependency checks using the `cargo audit` command.

Status: Acknowledged

The client states that dependencies are shared across all pallets within the repository. While they ensure that known vulnerabilities are addressed prior to each runtime release, they are unable to update or manage dependencies on a per-pallet basis, including for the specific pallet under review.

Appendix: Test Cases

1. Test case for "The `MaxConsumers` constraint limits multi-asset pool usability"

The following test case can be executed in `pallets/pallet-bonded-coins/src/tests/transactions/mint_into.rs`.

```
#[test]
fn mint_with_50_currencies_maxconsumers_limitation() {
    // 1. Create 50 different bonded currency IDs
    let mut currencies = Vec::with_capacity(50);
    for i in 0..50 {
        currencies.push(1 + i);
    }

    // 2. Create a pool_id deterministically from these 50 currencies.
    let pool_id: AccountIdOf<Test> = calculate_pool_id(&currencies);

    // 3. Use a polynomial curve with a large cubic coefficient to grow
quickly.
    //    If you want an even more explosive cost, try increasing 'm'.
    let curve = Curve::Polynomial(PolynomialParameters {
        m: Float::from_num(1),
        n: Float::from_num(2),
        o: Float::from_num(3),
    });

    // 4. Provide a huge initial collateral so we don't fail from
insufficient funds.
    //    We want to see if the cost arithmetic itself will overflow.
    let initial_collateral = u128::MAX / 10;


    // Build the pool with all 50 currencies, each initially at zero
supply.
    ExtBuilder::default()
      .with_native_balances(vec![(ACCOUNT_00, ONE_HUNDRED_KILT)])  //
for fees
      .with_collaterals(vec![DEFAULT_COLLATERAL_CURRENCY_ID])
      .with_bonded_balance(vec![
            (DEFAULT_COLLATERAL_CURRENCY_ID, ACCOUNT_00,
initial_collateral),
      ])
```

```
        .with_pools(vec![(
            pool_id.clone(),
            generate_pool_details(
                currencies.clone(),
                curve,
                true,                                   // transferable
                None,                                   // no forced state
                None,                                   // manager
                Some(DEFAULT_COLLATERAL_CURRENCY_ID),
                None,
                None,
            ),
        )])
        .build_and_execute_with_sanity_tests(|| {


            // ---------------------------------------------------
            // 5. MINT INTO EACH CURRENCY TO BUILD LARGE SUPPLY
            // ---------------------------------------------------
            // By minting a large amount in each currency, we
significantly raise
            // its supply. This will become part of the
"accumulated_passive_issuance"
            // for the *other* currencies in subsequent calls.
            let mint_each = 1u128;
            let max_cost = u128::MAX; // effectively "no slippage" limit

                  let origin:RuntimeOrigin =
frame_system::RawOrigin::Signed(ACCOUNT_00).into();
                  for (idx, _currency_id) in
currencies.iter().enumerate() {

                let result = BondingPallet::mint_into(
                    origin.clone(),
                    pool_id.clone(),
                    idx as u32,       // currency index in the bonded
list
                    ACCOUNT_00,       // beneficiary
                    mint_each,    // amount to mint
                    max_cost,
                    50,                // curve type or slip page param
                );

                        match result {
                              Ok(_) => {
```

```
                                    // If it succeeded, it means you
haven't overflowed yet with these numbers
                                    println!("Test minted amount
correctly");
                                }
                                Err(e) => {
                                    println!("Test error => {:?}", e);
                                }
                            }
                        }
                    });
}
```

2. **Test cases for "Configurations of `PolynomialParameters` can lead to overflow"**

The following test case can be executed in `pallets/pallet-bonded-coins/src/tests/transactions/mint_into.rs`.

```
#[test]
fn single_currency_mint_until_overflow() {
    // 1. We only have one currency in the bonded list
    let currency_id = DEFAULT_BONDED_CURRENCY_ID;
    let pool_id: AccountIdOf<Test> = calculate_pool_id(&[currency_id]);

    // 2. Use a polynomial curve that grows quickly
    //    so that repeated small mints eventually reach overflow
territory.
    let curve = Curve::Polynomial(PolynomialParameters {
        m: Float::from_num(10u128.pow(15)),
        n: Float::from_num(10000),
        o: Float::from_num(1),
    });

    // 3. Provide enough collateral so we never fail due to insufficient
funds
    //    (we want to see an arithmetic overflow, not a FundsUnavailable
error).
    let initial_collateral = u128::MAX / 10;

    // 4. Build the pool with a single currency, starting at zero
supply.
    ExtBuilder::default()
        .with_native_balances(vec![(ACCOUNT_00, ONE_HUNDRED_KILT)])  //
```

```
for fees
        .with_collaterals(vec![DEFAULT_COLLATERAL_CURRENCY_ID])
        .with_bonded_balance(vec![
            (DEFAULT_COLLATERAL_CURRENCY_ID, ACCOUNT_00,
initial_collateral),
        ])
        .with_pools(vec![(
            pool_id.clone(),
            generate_pool_details(
                vec![currency_id],
                curve,
                true,                                   // transferable
                None,                                   // no forced state
                None,                                   // manager
                Some(DEFAULT_COLLATERAL_CURRENCY_ID),
                None,
                None,
            ),
        )])
        .build_and_execute_with_sanity_tests(|| {
            let origin:RuntimeOrigin =
frame_system::RawOrigin::Signed(ACCOUNT_00).into();
            let max_cost = u128::MAX; // no practical slippage limit

            let mut total_minted = 0u128;
            loop {
                // Mint exactly 1 token each time
                let result = BondingPallet::mint_into(
                    origin.clone(),
                    pool_id.clone(),
                    0,          // index of the single currency
                    ACCOUNT_00,  // beneficiary
                    10u128.pow(9),
                    max_cost,    // slippage guard
                    1,
                );

                match result {
                    Ok(_) => {
                        total_minted = total_minted.saturating_add(1);
                        // If it succeeds, keep going until we overflow.
                    }
                    Err(e) => {
                        println!(
                            "Minting overflowed (or failed) after
```

```
iteration {}. Error: {:?}",
                                total_minted, e
                            );
                            // If this is ArithmeticError::Overflow, we've
triggered an overflow as intended.
                            break;
                        }
                    }
                }
            });
}
```

```
#[test]
fn mint_with_50_currencies_accumulate_passive() {
    // 1. Create 50 different bonded currency IDs
    let mut currencies = Vec::with_capacity(50);
    for i in 0..50 {
        currencies.push(1 + i);
    }

    // 2. Create a pool_id deterministically from these 50 currencies.
    let pool_id: AccountIdOf<Test> = calculate_pool_id(&currencies);

    // 3. Use a polynomial curve with a large cubic coefficient to grow
quickly.
    //    If you want an even more explosive cost, try increasing 'm'.
    let curve = Curve::Polynomial(PolynomialParameters {
        m: Float::from_num(1),
        n: Float::from_num(2),
        o: Float::from_num(3),
    });

    // 4. Provide a huge initial collateral so we don't fail from
insufficient funds.
    //    We want to see if the cost arithmetic itself will overflow.
      let mut funded_accounts = Vec::new();
      let mut native_accounts = Vec::new();

    let initial_collateral_each = u128::MAX / 10 / 50 ; // arbitrary
large share
    for i in 0..50 {
        let acct = AccountId::new([i as u8; 32]);
        funded_accounts.push((DEFAULT_COLLATERAL_CURRENCY_ID,
acct.clone(), initial_collateral_each));
            native_accounts.push((acct, ONE_HUNDRED_KILT))
```

```
    }

    // Build the pool with all 50 currencies, each initially at zero
supply.
    ExtBuilder::default()
        .with_native_balances(native_accounts) // for fees
        .with_collaterals(vec![DEFAULT_COLLATERAL_CURRENCY_ID])
        .with_bonded_balance(funded_accounts)
        .with_pools(vec![(
            pool_id.clone(),
            generate_pool_details(
                currencies.clone(),
                curve,
                true,                                  // transferable
                None,                                  // no forced state
                None,                                  // manager
                Some(DEFAULT_COLLATERAL_CURRENCY_ID),
                None,
                None,
            ),
        )])
        .build_and_execute_with_sanity_tests(|| {


            // ---------------------------------------------------
            // 5. MINT INTO EACH CURRENCY TO BUILD LARGE SUPPLY
            // ---------------------------------------------------
            // By minting a large amount in each currency, we
significantly raise
            // its supply. This will become part of the
"accumulated_passive_issuance"
            // for the *other* currencies in subsequent calls.
            let big_mint_each = 2u128.saturating_mul(10u128.pow(16)); //
try bigger if you need more extreme
            let max_cost = u128::MAX; // effectively "no slippage" limit

            for (idx, _currency_id) in currencies.iter().enumerate() {
                        let account =
AccountId::new([idx.try_into().unwrap(); 32]);
                        let origin:RuntimeOrigin =
frame_system::RawOrigin::Signed(account.clone()).into();

                let result = BondingPallet::mint_into(
                    origin.clone(),
                    pool_id.clone(),
```

```
                    idx as u32,       // currency index in the bonded
list
                    account,       // beneficiary
                    big_mint_each,    // amount to mint
                    max_cost,
                    50,                // curve type or slip page param
                );

                    match result {
                        Ok(_) => {
                            // If it succeeded, it means you
haven't overflowed yet with these numbers
                            println!("Test minted amount
*without* overflow!");
                        }
                        Err(e) => {
                            println!("Test error => {:?}", e);
                            // If you see
ArithmeticError::Overflow, you've successfully triggered an overflow
                        }
                    }
            }
        });
}
```