Intuition TRUST Token, Bonding and Emissions

Date	September 2025
Auditors	Heiko Fisch, George Kobakhidze

1 Executive Summary

This report presents the results of our engagement with **Intuition** to review a set of smart contracts including the `TRUST` token upgrade, emissions, and bonding components.

The review was conducted in **September 2025** by **George Kobakhidze** and **Heiko Fisch**.

The audit focused on the system of contracts governing `TRUST` token emissions, reward calculation, and bonding. The central user-facing onchain component is `TrustBonding`, where users deposit (bond) their `TRUST` and later claim rewards based on protocol activity metrics and their share of bonded voting power. Other contracts in scope facilitate emission scheduling and cross-layer movement / instantiation of supply as the system operates across two blockchains.

This system spans the Base network (canonical ERC-20 `TRUST` supply) and the Intuition L3 (Caldera network) where `TRUST` functions as the native gas token. Canonical minting and burning occur on Base, while user reward interactions and protocol activity occur primarily on L3. Correct operation therefore depends on coordinated offchain and cross-chain infrastructure (bridging / MetaLayer components) and on parameter synchronization between emissions controllers on each layer. These operational dependencies were not in scope, though certain deployment and configuration observations were provided by the reviewers.

Many contracts act autonomously once configured, but the presence of upgradeable contracts and privileged roles means operational security and role hygiene are critical. Specifically, centralized control over minting and emission parameters must be managed responsibly.

Overall, the in-scope code is structured and internally consistent with its stated emission decay and reward allocation model. The design relies on timely bridging, parameter parity, and accurate external data (e.g. utilization inputs) that were outside scope. Most identified issues are minor or informational. Some findings require straightforward fixes and do not indicate systemic fragility. Users and operators should remain attentive to cross-layer synchronization, role assignment, and bridging liveness as these factors materially influence realized reward dynamics.

After delivery of the initial version of this report, the Intuition team has shared with us a set of PRs addressing findings from this report and an accompanying document that explains the fixes and/or provides comments on the individual issues. For each finding, we briefly describe in the "Resolution" box at its beginning how it has been fixed or share the corresponding acknowledgement note from the client.

2 Scope

This review focused on the 0xIntuition/intuition-contracts-v2 repository, revision `8b1a8f5a3dc6ad8365c73f4cec46fcd08c38bcca`

The following files were in scope:

File	SHA-1 hash
src/Trust.sol	`06b03cd94181a53d0434c02ce005a527d399c845`
src/protocol/emissions/BaseEmissionsController.sol	`02eb4d03979cc269d7a276df2686cbe2c936215c`
src/protocol/emissions/CoreEmissionsController.sol	`f5b8d961407294baad41ec6df5015580a0825905`
src/protocol/emissions/SatelliteEmissionsController.sol	`3f51849697c5ce03f4df2c7798bc3259a78fb324`
src/protocol/emissions/TrustBonding.sol	`fa65f2d65459706573a64da12c4052ae96c76e67`

The following files were out of scope and only viewed for context. Some issues may refer to them:

File	SHA-1 hash	Reason
src/WrappedTrust.sol	`fb5fd1dc03b4381052df6d27aa9eb216c520fa22`	Fork of the `WETH9` token, i.e. Wrapped ETH, for the `TRUST` token on Intuition L3 to conform with ERC-20 token standard.
src/external/curve/VotingEscrow.sol	`60ce29e3244a090866f221bde7c2f145244d80a2`	Adapted from Stargate DAO's Solidity port of the VotingEscrow Vyper library made by the Curve project.
src/legacy/TrustToken.sol	`1df190618973c275d03655d304b7c02f49d0e454`	Source code of the currently deployed TRUST token on Base.
src/protocol/emissions/MetaERC20Dispatcher.sol	`008e18a49716079ef40ec945b55940b397436f65`	External bridging interface component of the Caldera MetaLayer implementation that is subject to change.

Everything else in the repository was also out of scope.

Design aspects of the rewards system were largely out of scope too. While finding 5.2 discusses one particular way to game the rewards system (which has been fixed), there are other ways to be smart about the rewards that are not or not fully protocol-aligned. The Intuition team is aware of the fact that there are shortcomings and wants to give the current system a try, thereby also gathering information for future refinements or potentially even bigger changes. Our focus was on whether the description of the rewards system the client gave us is correctly implemented.

3 System Overview

The Intuition protocol consists of a complex web of contracts that build upon each other. In this audit, we review the `TRUST` token and its associated emissions and rewards contracts to incentivize user interactions with the protocol.

3.1 Cross-Layer Architecture

The system spans two blockchains. The canonical `TRUST` ERC-20 contract (upgraded existing deployment) resides on the Base network and anchors the total token supply. This is where the tokens get minted and burnt by the canonical contract. Intuition L3 (a Caldera network purpose-built for the protocol) treats `TRUST` as its native gas token. Native token availability on L3 therefore depends on correct, continuous operation of offchain and cross-chain bridging components. If the initial bridging stalls or misconfigures, the L3 could experience a functional outage (no gas for transactions) despite the Base-side supply being intact.

Two coordinating emissions controllers must remain in sync:

- `BaseEmissionsController` (origin side; mints canonical `TRUST`)
- `SatelliteEmissionsController` (L3 side; receives native-minted `TRUST` via Caldera MetaLayer machinery)

Both must share almost identical configuration (epoch length/start, cliff schedule, retention rate, target inflow metrics, emission ratio floor/ceiling, and any bridge identifiers). Divergence introduces inconsistent reward ceilings or orphaned accounting states. This creates an operational responsibility for Intuition and Caldera teams: configure, monitor, and reconcile parameters, and ensure bridging pathway integrity (MetaLayer dispatcher, hub, minting adapter). These are centralization points outside autonomous onchain guarantees.

3.2 Emissions Flow

Emissions are time-segmented by epochs and decay across discrete “cliffs.” They may be represented by the following formula:

Emissions = `E = B * r^c`

Where:

- `B` = base (initial maximum) emissions per epoch
- `r` = retention rate (0 < r ≤ 1)
- `c` = number of cliffs elapsed

This is subject to integer / rounding effects in implementation.

The entity with the `CONTROLLER` role on Base invokes `BaseEmissionsController`, which mints `TRUST` via the canonical token contract. The minted amount (bounded by the current cliff-adjusted limit) is forwarded to the Caldera MetaLayer hub and bridged. On Intuition L3 a Caldera contract performs a native mint directly to `SatelliteEmissionsController`. Tokens accumulate there until the `TrustBonding` contract calls the `SatelliteEmissionsController` to distribute the token to users as protocol rewards. Any bridge delay creates temporal skew between theoretical epoch emissions and availability for bonding rewards; sustained lag could compress later epoch payouts or require catch-up logic.

3.3 Rewards Model

Per-user epoch rewards are multiplicative across four factors:

```
reward = E * Us * Tu * Uu
```

Where:

- `E`: epoch emission cap as above.
- `Us` (system utilization): floor + (system_inflow_delta / target) * (1 - floor).
- `Tu` (user voting share): user bonded voting power / total bonded voting power from `VotingEscrow`.
- `Uu` (user utilization): floor + (user_inflow_delta / target) * (1 - floor).

The “floor” (lower bound emission ratio) guarantees a minimum participation-based share even under low activity. Inflow deltas measure change in deposited amounts between epochs relative to prior claimed totals, rewarding growth rather than static positions.

3.4 Source of Utilization Data (Out of Scope Component)

Intuition `MultiVault` contracts (out of audit scope) are the primary application layer where users deposit `TRUST` to express protocol-level signals. These contracts surface:

- Per-user epoch inflows
- Aggregate system inflows

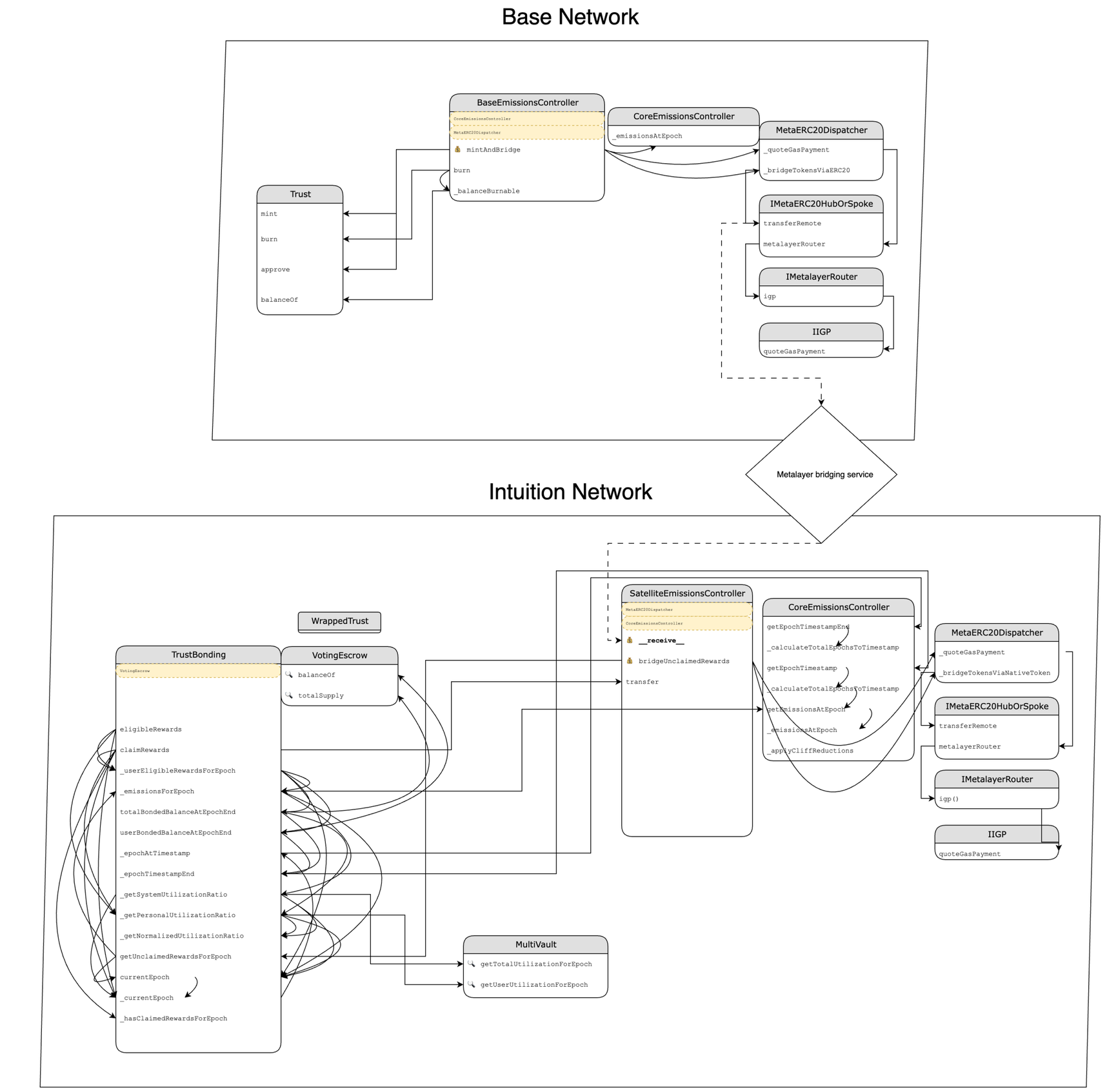
`TrustBonding` consumes these values to derive user and system utilization (deltas vs prior epoch baselines), then queries `VotingEscrow` for bonded balances (time-weighted or supply-weighted voting power, depending on the external library semantics). Although `MultiVault` logic is not assessed here, its correctness directly impacts reward fairness and emission pacing.

3.5 Bonding and Distribution

Workflow:

1. User locks `TRUST` (or maintains an existing lock) contributing to voting power.
2. User activity in MultiVaults generates inflow deltas recorded per epoch.
3. At claim, `TrustBonding`:
    - Determines current epoch & cliff-derived E from the satellite controller.
    - Computes Us and Uu from recorded inflows and targets.
    - Computes Tu from `VotingEscrow` data.
    - Multiplies to obtain reward and instructs `SatelliteEmissionsController` to transfer.
4. Reward is transferred as native `TRUST` on L3.

The diagram below showcases how intertwined the system contracts are with each other to retrieve relevant data and perform asset flows. Specifically, the `eligibleRewards` and `claimRewards` functions are considered for the `TrustBonding` contract, while other contracts have most of their intra- and cross-contract calls represented.



4 Security Specification

4.1 Actors & Trust model

The relevant actors are listed below with their respective abilities and trust placed into them, if any:

MultiVault Users

These users deposit and withdraw `TRUST` in out-of-scope MultiVault contracts, creating per-user and aggregate inflow deltas that `TrustBonding` later consumes. They are not trusted for correctness; strategic timing can influence utilization factors but they don't have any privileged access to any functions.

TrustBonding Users

These participants lock ("bond") `TRUST` to obtain voting power and later claim rewards computed from emissions, utilization, and voting share. They cannot directly tune parameters at this time, though their voting power can be used later for that.

Caldera Team

This team operates L3 infrastructure and the bridging pathway (MetaLayer dispatcher / hub / mint adapter) that enables native `TRUST` issuance on Intuition L3. High operational trust is placed here: outages may delay emissions and bridging unclaimed rewards back to Base. Compromised access can potentially strand or misdirect bridged value.

Proxy Admin Owners (Upgrade Authorities)

This entity can upgrade proxied contracts (`TRUST`, emission controllers, `TrustBonding`) and thus alter code or storage arbitrarily. It carries the highest trust burden; a compromise allows unlimited minting, logic substitution, or seizure until a decentralized / time-locked governance process replaces or constrains it.

`Trust` Token Roles

`CONTROLLER_ROLE` may invoke `mint()` and `burn()` (intended only for `BaseEmissionsController`); misuse or misassignment can lead to arbitrary minting (incl. unbounded inflation) and/or burning from others' addresses. `DEFAULT_ADMIN_ROLE` governs role grants and revocations, indirectly controlling mint/burn authorization and expanding or contracting privilege surfaces. The one-time `INITIAL_ADMIN` handles migration / reinitialization of the token contract and is thus trusted to provide correct addresses for privileged roles.

`BaseEmissionsController` Roles

`CONTROLLER_ROLE` calls `mintAndBridge()`, initiating epoch mints subject to schedule constraints and triggering bridging; incorrect usage can mistime emissions, delaying reward and thus breaking user accounting for claims since they won't get them in the right epoch. `DEFAULT_ADMIN_ROLE` adjusts bridging parameters (`setMessageGasCost`, `setFinalityState`, `setMetaERC20SpokeOrHub`, `setRecipientDomain`), may call `burn()`, and manages roles; parameter errors or compromised access to the `DEFAULT_ADMIN_ROLE` address can strand or halt emissions.

`SatelliteEmissionsController` Roles

`CONTROLLER_ROLE` transfers native `TRUST` to recipients. The intended address for this is the `TrustBonding` contract, though assignment to an unintended address would enable direct draining of emission balances. `DEFAULT_ADMIN_ROLE` manages the same bridging parameters as it would for the `BaseEmissionsController` contract plus it is able to call `bridgeUnclaimedRewards(epoch)` and perform role administration. A compromised `DEFAULT_ADMIN_ROLE` address would be able to assign the `CONTROLLER_ROLE` to their owned address and transfer out any emissions sent to this contract.

`TrustBonding` Roles

`PAUSER_ROLE` can halt core claim logic via `pause()`, providing emergency response but also a censorship / liveness lever if abused. `DEFAULT_ADMIN_ROLE` restores activity with `unpause()` and manages role assignments, becoming central for recovery. `TIMELOCK_ROLE` sets integration addresses (`setMultiVault`, `updateSatelliteEmissionsController`) and adjusts utilization lower bounds. By changing those addresses, namely the `setMultiVault`, the `TIMELOCK_ROLE` would be able to change how many tokens are retrieved from the `SatelliteEmissionsController` emissions. As before, the `DEFAULT_ADMIN_ROLE` can indirectly perform all role activities via assigning them to the addresses it controls.

Below is a User Flow diagram to help facilitate understanding of the system:

4.2 Other Security-Related Aspects

Parameter Synchronization.

Divergence between Base and L3 emission controller configs (epochs, cliffs, retention, utilization targets) undermines reward correctness and can produce stuck or excess balances.

Bridging Integrity.

Any mismatch in domain IDs, hub addresses, or finality state settings can stall or orphan emissions. In general, offchain Caldera-ran components are critical for system health.

Role Concentration.

Consolidation of multiple admin roles in a few EOA increases the number of single points of failure.

Upgrade Surface.

Absence of time delays or multi-sig requirements amplifies privileged risk.

5 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

5.1 `eligibleRewards` Function Has Incorrect Epoch Calculation Major ✓ Fixed

Resolution
Fixed in PR 51 (reviewed commit bbc309a79af3398d5205cc517a385c3f74c48059) by performing the correct calculation. The function was reformatted and renamed in later PRs, such as in PR 83 (reviewed commit 89dc71e9aef6cc5d332bf1e8daab5a1b9a009eff) where a more straightforward previous epoch calculation was used.

Description

To calculate the rewards for users, the `TrustBonding` contract uses two primary functions - `eligibleRewards` and `claimRewards`. The former is a `view` function that is meant to return the currently eligible rewards for the user, while `claimRewards` actually claims

those rewards and performs contract state updates.

An important part of those calculations is assessing the current epoch and calculating the previous epoch. While this is simple, there is an edge if the current epoch is 0, as the previous epoch wouldn't exist. The `eligibleRewards` has an issue in this calculation in that it always returns 0:

src/protocol/emissions/TrustBonding.sol:L216

```
uint256 prevEpoch = currentEpochLocal == 1 ? currentEpochLocal - 1 : 0;
```

In turn, this always returns the rewards that are eligible for the user for the epoch 0, no matter the current epoch. It is important to note that the `claimRewards` function performs the calculation correctly and does not have this issue. However, the impact of this incorrect view function could be quite significant if it is called by other integrating contracts. That said, the damage is contained if it is only used by offchain components, like frontends.

It is also noteworthy that the `eligibleRewards()` function does not call `_hasClaimedRewardsForEpoch()` like `claimRewards()` does. Therefore, it would always show what the user could claim in the current epoch, no matter if they had already claimed it or not.

Recommendation

Address the epoch calculation in the `eligibleRewards()` function. Consider adding the call to `_hasClaimedRewardsForEpoch()` in the `eligibleRewards()` function as well, as appropriate for the use case.

5.2 Users That Skip Claims Automatically Get Maximum Personal Utilization Ratio Medium ✓ Fixed

Resolution

Fixed in PR 66 (reviewed commit 35a380665dc3793e79ba7103dfe074c8083c8fab) by returning the minimum personal utilization ratio for users who skip rewards claims. Also accommodates an edge case for users who had no claimable rewards in the previous epoch at all, such as first time users or those that have entirely exited from the system at one point for a duration of at least one epoch.

Description

Personal utilization ratio in the `TrustBonding` contract depends on two user actions - their utilization delta between two epochs, and how many tokens they have claimed in the previous epoch. Users can exploit this calculation to claim disproportionate rewards by strategically skipping epochs and making minimal `MultiVault` deposits.

This strategy hinges on getting to this return statement which returns maximum possible personal utilization ratio number:

src/protocol/emissions/TrustBonding.sol:L514-L516

```
if (userUtilizationTarget == 0 || userUtilizationDelta >= userUtilizationTarget) {
    return BASIS_POINTS_DIVISOR;
}
```

The first check allows users who never had claims before to participate in the system unpunished, i.e. if their previous rewards (target) is 0, then get full rewards. The second check is simply the intended mechanism of the calculation - if the user has more utilization than they have been rewarded, provide the maximum ratio.

However, a malicious user can pretend to be a first time claimer by forcing their previous target to be 0. Specifically, when a user has no claimed rewards in the previous epoch their utilization target becomes 0:

src/protocol/emissions/TrustBonding.sol:L509

```
uint256 userUtilizationTarget = userClaimedRewardsForEpoch[_account][_epoch - 1];
```

And in order to get to this point, all the user needs to do is make sure their utilization delta is positive:

src/protocol/emissions/TrustBonding.sol:L493-L503

```
// Fetch the personal utilization before and after the epoch
int256 userUtilizationBefore = IMultiVault(multiVault).getUserUtilizationForEpoch(_account, _epoch - 1);
int256 userUtilizationAfter = IMultiVault(multiVault).getUserUtilizationForEpoch(_account, _epoch);

// Since rawUtilizationDelta is signed, we only do a sign check, as the explicit underflow check is not needed
int256 rawUtilizationDelta = userUtilizationAfter - userUtilizationBefore;

// If the utilizationDelta is negative or zero, we return the minimum personal utilization ratio
if (rawUtilizationDelta <= 0) {
    return personalUtilizationLowerBound;
}
```

This can be done by simply increasing their deposited amounts in the `MultiVault` contract by the smallest possible deposit. The exact number is based on the `MultiVault` configurations and individual vaults to be deposited into, but the point is that it would allow the deposit amount to be independent of the actual size of the rewards the user has incoming.

The impact is that users can avoid doing almost any deposit activity in the `MultiVault` contracts and achieve an average of half of maximum personal utilization ratio, and thus half of maximum rewards share based on user activity, every epoch. That is because they will skip their claim one epoch and get maximum rewards based on personal utilization ratio the next epoch.

The exact impact would depend on the rewards configuration. For example, if the lower bound of personal utilization ratio is 80%, then it would not make sense to perform this scheme. If it is something like 10%, then skipping a claim to achieve full utilization ratio the next epoch starts appearing very attractive.

Recommendation

Assess the intended behavior of the user and evaluate if skipping a claim warrants disincentives.

5.3 Lack of Atomicity for Deployment and Initialization as Well as for Upgrade and Reinitialization

Medium ✓ Fixed

Resolution

Addressed in PR 64 (reviewed commit b1e960888da25dec915a16a265290b551cc1693f) by including initialization data along with the proxy deployment. Additionally, contracts such as `BaseEmissionsController` and `SatelliteEmissionsController` were modified to not require each other's (and `TrustBonding`'s) contract addresses upon initialization, which would happen at deployment time due to the fix for this issue. Therefore, contracts can be deployed and initialized individually and after that, the necessary functions can be called by administrators to set up the remaining addresses, such as `setSatelliteEmissionsController`. In PR 64 and PR 83 (reviewed commit 89dc71e9aef6cc5d332bf1e8daab5a1b9a009eff) appropriate zero-address checks were introduced to ensure the contracts were fully configured prior to usage, such as in `bridgeUnclaimedEmissions`.

Regarding the token contract upgrade, the restriction that `reinitialize` can only be called by `INITIAL_ADMIN` has been removed. Hence, it is now essential that upgrade and reinitialization will happen atomically. This has been noted in the "Mainnet Deployment Steps" in the description of PR 64, so this part of the finding has been acknowledged and the client intends to follow this recommendation.

Description

Most contracts in the system are upgradeable. It is crucial that proxies are deployed and initialized atomically; otherwise, an attacker can front-run the initialization call and, possibly, cause serious damage.

While the deployment scripts are not in scope for this review, we took a short look at them anyway and noticed that, currently, deployment and initialization are not atomic. The client has assured us they're aware of this issue and have tolerated it for testing but will implement atomic deployment and initialization before going live.

There will likely be a few challenges with this. For example, the `TrustBonding` contract's `initialize` function requires the address of the `SatelliteEmissionsController` and vice versa. So addresses have to be pre-computed before deployment or (permissioned) code to change an address after initialization has to be added – which might cause more complications, such as preventing system/contract use before everything is set up correctly.

In principle, what we discussed above for deployment and initialization also applies to upgrade and reinitialization: They should be atomic too. The only contract that is to be upgraded this time is the token contract on Base. Its new implementation contract, `Trust`, has a `reinitialize` function that can only be called by an address that is hard-coded in the contract source:

src/Trust.sol:L21-L22

```
/// @notice Address of the initial admin, which is allowed to perform the contract reinitialization
address public constant INITIAL_ADMIN = 0xa28d4AAcA48bE54824dA53a19b05121DE71Ef480;
```

src/Trust.sol:L59-L62

```
function reinitialize(address _admin, address _controller) external reinitializer(2) {
    if (msg.sender != INITIAL_ADMIN) {
        revert Trust_OnlyInitialAdmin();
    }
```

In other words, the `Trust` contract has an ad hoc mechanism that prevents reinitialization by an unauthorized party. Nevertheless, the non-atomicity means that other functions can be called between upgrade and reintialization. While – in this particular case – we don't see an immediate problem with that, we still believe it would be cleaner to rely on the established pattern of atomic upgrade and reinitialization.

Recommendation

We recommend implementing atomic deployment and initialization for all new deployments and atomic upgrade and reinitialization for the token contract upgrade.

5.4 `WrappedTrust` Uses Solidity's `transfer` Function Medium ✓ Fixed

Resolution

Fixed in PR 52 (reviewed commit 249d7f52bc21263ba3abc65ba6600ab3b3dc9d81) by using the OZ `Address` library's `sendValue` function instead of `transfer`.

Description

The `WrappedTrust` contract is an ERC-20 wrapper around the native asset of the chain; according to the Intuition team, it is forked from WETH9. The `withdraw` function reduces the sender's token balance by the given amount and sends the same amount of the

chain's native asset back. The latter is achieved with Solidity's `transfer` function, which equips the call to the recipient with a fixed amount of 2300 gas. In other words, if the `msg.sender` of the `withdraw` call is a contract C that – upon receipt of a transfer of the native asset without calldata (typically, the `receive` function is called in such a case) – consumes more than 2300 gas, then the withdrawal will revert, and the funds will be stuck.

In many cases, there will be a workaround for such a situation: For example, it might be possible to upgrade C (which will often, but not necessarily be a wallet), or C might have the capability to transfer the wrapped Trust to a different address, which doesn't exhibit the same problem for withdrawals. Nevertheless, there is no guarantee that the situation can be resolved, and in the worst case, the funds would be stuck.

Nowadays, it is frequently recommended to not use Solidity's `transfer` anymore for this reason, and instead utilize OpenZeppelin's `Address.sendValue`, which essentially makes a low-level call with empty calldata and the amount to be transferred but without limiting the gas forwarded.

An important point to keep in mind is that this opens the door to reentrancy – which wasn't possible with only 2300 gas sent along. In the case of `WrappedTrust`, this is not a big problem, but we recommend moving the emission of the `Withdrawal` event in front of the actual transfer of the native asset. Otherwise, events could be emitted in the wrong order via reentrancy, possibly causing problems for event-monitoring code.

Recommendation

We recommend importing OpenZeppelin's `Address` library and replacing the following two lines

src/WrappedTrust.sol:L43-L44

```
payable(msg.sender).transfer(amount);
emit Withdrawal(msg.sender, amount);
```

with

```
emit Withdrawal(msg.sender, amount);
Address.sendValue(payable(msg.sender), amount);
```

5.5 Points Related to Access Control in General and to Minting/Burning of Tokens Medium ✓ Fixed

Resolution

Improved in PR 53 (reviewed commit 2eb9c8d90af9dbec97b7ceaaac60a2c7a38260c2) by:

- modifying the timelock address to not be a role managed by the `DEFAULT_ADMIN_ROLE` but instead be a state variable with the associated `onlyTimelock` modifier, which can only be changed by itself;
- changing the `burn` function such that it can be called by anyone but only burns `msg.sender`'s tokens;
- restricting minting to a single address stored in `baseEmissionsController`, instead of a role that could be assigned to several different addresses. (It should be noted that `baseEmissionsController`'s value can be changed by any address with the `DEFAULT_ADMIN_ROLE`, so addresses with that role are still, indirectly, in control of minting.)

Description

The contracts make heavy use of OpenZeppelin's Access Control framework, in which roles can be defined and granted to certain addresses. A typical usage pattern is to have privileged contract operations that can only be executed by addresses which have a certain role. Role management itself is also handled via roles: Each role has a manager role that can grant/revoke the role to/from addresses. The same role can be granted to several addresses. In the following, we discuss two particular points related to access control, but the conclusion is more general.

A. The `TrustBonding` contract defines a role `TIMELOCK_ROLE`:

src/protocol/emissions/TrustBonding.sol:L73-L74

```
/// @notice Role used for the timelocked operations
bytes32 public constant TIMELOCK_ROLE = keccak256("TIMELOCK_ROLE");
```

Changing certain system parameters (like the personal/system utilization lower bounds) or changing the addresses of contracts `TrustBonding` interacts with can only be done by an address that has the `TIMELOCK_ROLE` role. The name of the role suggests that these actions can't be executed immediately, so no "surprising changes" should occur. However, even if the role is initially assigned to a timelock contract, the role's administrator (`DEFAULT_ADMIN_ROLE`) can still grant the role to an additional address that is not a timelock contract, so changes would be possible immediately – unless role management is behind a timelock too. And even if that is the case, the contract could be upgraded immediately to an entirely different logic – unless contract upgrades are also behind a timelock.

B. In the `Trust` token contract, only an address that has the `CONTROLLER_ROLE` can mint or burn:

src/Trust.sol:L98-L104

```
function mint(address to, uint256 amount) public override onlyRole(CONTROLLER_ROLE) {
    _mint(to, amount);
}

function burn(address from, uint256 amount) external onlyRole(CONTROLLER_ROLE) {
    _burn(from, amount);
}
```

Since the `CONTROLLER_ROLE` role should be granted to the `BaseEmissionsController` contract, at a quick glance, it might seem that minting and burning are "safe," in the sense that it is contract-controlled. However, the administrator of `CONTROLLER_ROLE` (`DEFAULT_ADMIN` , unless changed) could simply grant the role to another address, and then this address could mint and burn tokens at will. Notably, even the token of others can be burnt, so this address would be in complete control of the token's balances. The same effect – full control over the token balances – could be achieved by upgrading the `BaseEmissionsController` or the `Trust` token itself.

Recommendation

It is important to note that the Intuition team is – mostly through contract upgrades and role administration – in full control of the system, even the token balances. If stronger guarantees are to be given to users, the considerations discussed above have to be taken into account. We recommend clarifying and documenting what guarantees you want to give (and have the code checked whether it holds up to these guarantees) and where users have to trust you.

Specifically, for the two points above:

A. If role management and contract upgrades are not behind a timelock too, the role `TIMELOCK_ROLE` should be renamed to avoid misleading readers of the code.

B. Since `BaseEmissionsController` only mints for itself and only burns from itself, the `mint` and `burn` functions in `Trust` should be changed to only take an `amount` parameter, but not `to` / `from` anymore. The address minted to / burned from should then be `msg.sender` . Of course, the calls of these functions in `BaseEmissionsController` have to be adjusted too.
This change in the `burn` function would at least remove the power to burn someone else's token from the current contract logic and therefore take it out of the hands of role management and whoever can upgrade `BaseEmissionsController` . Of course, an upgrade of the token contract itself would still be possible and put the balances in control of the upgraded contract.
Note: Removing the `to` argument from the `mint` function is only recommended because it's not needed and for consistency with the suggested change in `burn` , but it has no tangible advantage other than these. So `mint` could also be kept as it is. However, *if* the `to` argument is removed, a version of `mint` *with* the `to` argument should still be present and revert unconditionally, as a function with the same signature is inherited from the legacy token contract `TrustToken` and would – if not overridden – still be callable after the upgrade.

5.6 Setters Don't Check Values Consistently Minor ✓ Fixed

Resolution

Fixed in PR 54 (reviewed commit a474e6569243693605e3a104a42516d298e75019) by adding zero-address checks to initializers and setters.

Description

Across the system of contracts in scope, there are many admin setter functions to define special addresses, bounds for token emissions, and so on. Some of those setters employ value checks on provided parameters, mostly in the `TrustBonding` contract:

src/protocol/emissions/TrustBonding.sol:L372-L375

```
function setMultiVault(address _multiVault) external onlyRole(TIMELOCK_ROLE) {
    if (_multiVault == address(0)) {
        revert TrustBonding_ZeroAddress();
    }
```

src/protocol/emissions/TrustBonding.sol:L383-L389

```
function updateSatelliteEmissionsController(address _satelliteEmissionsController)
    external
    onlyRole(TIMELOCK_ROLE)
{
    if (_satelliteEmissionsController == address(0)) {
        revert TrustBonding_ZeroAddress();
    }
```

While others, such as those in `SatelliteEmissionsController` and `BaseEmissionsController` don't have any checks:

src/protocol/emissions/SatelliteEmissionsController.sol:L132-L134

```
function setMetaERC20SpokeOrHub(address newMetaERC20SpokeOrHub) external onlyRole(DEFAULT_ADMIN_ROLE) {
    _setMetaERC20SpokeOrHub(newMetaERC20SpokeOrHub);
}
```

src/protocol/emissions/BaseEmissionsController.sol:L186-L188

```
function setMetaERC20SpokeOrHub(address newMetaERC20SpokeOrHub) external onlyRole(DEFAULT_ADMIN_ROLE) {
    _setMetaERC20SpokeOrHub(newMetaERC20SpokeOrHub);
}
```

src/protocol/emissions/MetaERC20Dispatcher.sol:L91-L94

```
function _setMetaERC20SpokeOrHub(address newMetaERC20SpokeOrHub) internal {
    _metaERC20SpokeOrHub = newMetaERC20SpokeOrHub;
    emit MetaERC20SpokeOrHubUpdated(newMetaERC20SpokeOrHub);
}
```

Similarly, the `initialize` function in `TrustBonding` performs the appropriate checks, while the `initialize` functions in the emissions controllers don't. This is particularly important for admin roles, such as `DEFAULT_ADMIN`.

Recommendation

Apply validation checks across all parameter setting and intialization functions in `SatelliteEmissionsController` and `BaseEmissionsController` consistently, such as in `TrustBonding`.

5.7 Missing Getter Functions in `SatelliteEmissionsController` Minor ✓ Fixed

Resolution

Fixed in PR 55 (reviewed commit 185c520ee6d593feee29fd0ffcb5f47a1ecc3e75) by creating getters in the `SatelliteEmissionsController` contract.

Description

The state variables in both emissions controllers are `internal`. For example, these are the state variables in `SatelliteEmissionsController`:

src/protocol/emissions/SatelliteEmissionsController.sol:L37-L48

```
/* ================================================== */
/*                   INTERNAL STATE                   */
/* ================================================== */

/// @notice Address of the TrustBonding contract
address internal _TRUST_BONDING;

/// @notice Address of the BaseEmissionsController contract
address internal _BASE_EMISSIONS_CONTROLLER;

/// @notice Mapping of bridged rewards for each epoch
mapping(uint256 epoch => uint256 amount) internal _bridgedRewards;
```

Since they are internal, no getter functions will be automatically generated by the compiler. While `BaseEmissionsController` has self-written getter functions for all state variables, these are missing in `SatelliteEmissionsController`.

Recommendation

Analogously to `BaseEmissionsController`, implement getter functions for `SatelliteEmissionsController`'s state variables and add them to the interface.

Alternatively, make the state variables in both emissions controllers `public`, which will cause Solidity to auto-generate getter functions for them. (The name of these will be different from the current names of the self-written getters, though.) Then remove the self-implemented getters, and adjust both interfaces accordingly.

5.8 Missing Events Minor ✓ Fixed

Resolution

Fixed in PR 56 (reviewed commit bf19c19669f9ada5f961e69823d54c1ac2edc709) by emitting more events in `BaseEmissionsController` and `SatelliteEmissionsController`, as well as calling more internal functions in `TrustBonding` that emit events instead of setting values directly.

Description

Several state-changing functions don't emit events:

A. In `SatelliteEmissionsController`, both `transfer` and `bridgeUnclaimedRewards` should emit an event with the relevant information.

B. In `BaseEmissionsController`, the `burn` function should, ideally, emit an event too. While a corresponding event is emitted on the token contract, it might still be nice to have one on the emissions controller, too.

C. Several initialization functions don't emit an event for the values they set. Specifically, this is true for `BaseEmissionsController`, `SatelliteEmissionsController`, and `TrustBonding`. (Note that the `_grantRole` does emit an event, so this is already covered.) Generally, it is an elegant pattern to have internal `_set*` functions for the values that are set in an initialization function, which do (1) the necessary validations, (2) set or update the value, and (3) emit the corresponding event. The initialization function then just has to call these internal function and doesn't have to concern itself with validations or events. (This pattern has essentially been followed in `MetaERC20Dispatcher`, for instance, although there are no validations currently.) Moreover, access-restricted external functions to change these values can be easily added too, as they just have to be an access-controlled wrapper around the internal setter. Writing the intialization function and external setter functions each in an ad hoc manner easily leads to forgotten events or repeated validation checks (e.g., in `TrustBonding`).

Recommendation

We recommend adding at least the missing events. Additionally, consider some refactorizations as discussed in C.

5.9 Gap Variables Missing From Some Contracts Minor ✓ Fixed

Resolution

Fixed in PR 57 (reviewed commit d55bb15da6f36f2cec1e189b33d5ddf09f87f7c4) by adding gap variables to `VotingEscrow`, `CoreEmissionsController`, `MetaERC20Dispatcher`, and `SatelliteEmissionsController`.

Description

Most contracts in the system are upgradeable. They use the traditional storage layout (as opposed to namespaced storage), and while namespaced storage might be generally preferable, the fact that `TrustBonding` inherits from `VotingEscrow` might cause some complications here for a simple and complete switch to namespaced storage: `VotingEscrow` is not in scope for this review, but it is adapted from a non-upgradeable version from Stargate DAO and uses the traditional storage layout. Hence, significant changes – and the accompanying risk of introducing mistakes – would be required to modify this contract to utilize namespaced storage. On the other hand, keeping the traditional storage layout for `VotingEscrow` and have all other contracts use namespaced storage would be inconsistent too. Consequently, we think using a traditional storage layout throughout the codebase (with the exception of the OpenZeppelin Upgradeable Contracts – which use namespaced storage, but that should be fairly "hidden") is a defendable choice.

One pain point of the traditional storage layout is the need for gap variables – basically, reserved space that can be utilized later if an upgrade requires additional storage. Some, but not all contracts in scope have such a gap variable. It is present in `Trust`, `BaseEmissionsController`, and `TrustBonding`, but `CoreEmissionsController`, `MetaERC20Dispatcher`, and `SatelliteEmissionsController` don't have it. Crucially, `CoreEmissionsController` and `MetaERC20Dispatcher` are contracts that are inherited from, while `BaseEmissionsController`, `SatelliteEmissionsController`, and `TrustBonding` are most-derived contracts that currently no other contract inherits from.

State variables are laid out in the C3-linearized order, from most basic to most derived. (In more detail, this is explained here.) Therefore, it is important that contracts that are inherited from have a gap variable, so we can add state variables (and reduce the gap accordingly) later without moving the state variables of contracts that are more derived in the C3 linearization. On the other hand, "end contracts", i.e., contracts that are currently not inherited from, don't need a gap variable right now – but it doesn't hurt either.

Recommendation

Add a gap variable to `CoreEmissionsController` and `MetaERC20Dispatcher`, so if you want/have to change these contracts for an upgrade at a later point, it won't be a hassle.

There's no disadvantage to keeping the gap in `Trust`, `BaseEmissionsController`, and `TrustBonding`; in fact, should you ever decide to inherit from these contracts and the gap is already there, it can't be forgotten to add it, so we actually recommend keeping it. (Some people think it's ugly, though, to have gap variables in most derived contracts.) However, if you do keep the gap variable in these contracts, you should also add one to `SatelliteEmissionsController` for consistency.

Finally, a word regarding the `VotingEscrow` contract, which is out of scope for this review. As mentioned above, it is adapted from Stargate DAO and assumed – for the purposes of this review – to work as intended. Some changes have been made to this contract, compared to the original. For instance, it now uses the upgradeable OZ contracts, while the Stargate version uses the non-upgradeable OZ library. Moreover, `Ownable` has been replaced with `AccessControlUpgradeable`. In short, several changes have been made already, and while, in principle, changes should be avoided or at least kept to a minimum, regarding the amount of changes already made, it might make sense to add a gap variable too.

5.10 Libraries Used in Versions With Known Issues or at Unreleased Commits Minor ✓ Fixed

Resolution

Fixed in PR 58 (reviewed commit 657d226f7ac28ecff92edf2283808d86b9e446ad) by updating OZ and Solady libraries to versions `v5.4.0` and `v0.1.26`, respectively.

Description

The codebase utilizes libraries which have been imported in versions with known issues or at commits that do not correspond to a released version.

Examples

1. The Trust token contract, which has originally been developed with OpenZeppelin's v4 upgradeable contracts, will receive an upgrade. Since v5 switched to namespaced storage, even the upgraded token code will use v4, in order to preserve the original storage layout. However, the version actually used in v4.9.2, which has known issues. Instead, the latest bugfix release from the v4.9 series should be used, which – at the time of writing – is v4.9.6.
2. The rest of the codebase uses OpenZeppelin libraries in version 5. However, the versions used – openzeppelin-contracts @ efdc7cd and openzeppelin-contracts-upgradeable @ 5bab080 – do not correspond to a released version and are, in fact, ahead of any release.
3. The solady library is used in the version at commit hash `[solady @ 33b4b98](https://github.com/vectorized/solady/tree/33b4b98e350bbcba6aa85642957c313e98b5f911)`, which does not correspond to a released version – and is behind the latest release v0.1.26.

Recommendation

As a general rule, we recommend using libraries in released versions that have no known issues, since known issues might affect your codebase, and unreleased versions are likely in active development and might contain experimental or unfinished code that is quite possibly less reviewed and tested.

5.11 Different Version Pragmas Minor ✓ Fixed

Resolution

Fixed in PR 59 (reviewed commit 564e25c7de75c7429fac401d34ad475b5140ec36) by standardizing to `pragma solidity 0.8.29` across the codebase.

Description

The contracts across the repository use different `pragma solidity`. For example:

- `Trust.sol` has `pragma solidity ^0.8.19;`.
- `TrustBonding.sol` and `WrappedTrust.sol` have `pragma solidity ^0.8.27;`.
- Emissions controllers and `MetaERC20Dispatcher.sol` have `pragma solidity ^0.8.29;`.

Standardizing the same Solidity version across contracts in the codebase is best for maintainability of the repository. Also, the pragma just states the minimum version that this file compiles with. The actual compiler version specified in `foundry.toml` is v0.8.29, so the pragmas could use `^0.8.29`.

Recommendation

Standardize the version pragmas. Specifically, change it to `pragma solidity ^0.8.29;` in `Trust.sol`, `TrustBonding.sol`, and `WrappedTrust.sol`.

5.12 Changed Token Name Might Not Be Universally Picked Up Minor Acknowledged

Resolution

Acknowledged with the following note:

> We tested on Base mainnet & Sepolia: explorer token tracker pages do not auto-reflect name changes (shows only via “Read as Proxy → `name()`”). We accept this trade-off and will proactively contact Basescan ahead of mainnet upgrade to ensure correct display.

Description

The Trust ERC-20 token contract currently deployed on Base has the name and symbol “TRUST”. The planned upgrade of this contract intends to change the token’s name to “Intuition” by overriding the `name` function as follows:

src/Trust.sol:L80-L87

```
/**
 * @notice Returns the name of the token
 * @dev Overrides the `name` function in ERC20Upgradeable
 * @return Name of the token
 */
function name() public view virtual override returns (string memory) {
    return "Intuition";
}
```

From a contracts-only perspective, there is nothing wrong with that. However, it seems to be fairly uncommon to change the name of a token after its deployment. In fact, in a standard OpenZeppelin ERC-20 contract it can’t be changed:

```
    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * Both values are immutable: they can only be set once during construction.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }
```

Hence, it is conceivable that tools like block explorers wouldn’t pick up the change, i.e., they may just read the name and symbol once and then assume it won’t change. And even if some tools do, there might still be others that don’t, which might lead to confusion or inconsistent information.

Recommendation

With the concerns raised above, reconsider whether the token’s name should indeed be changed. Maybe run some tests on the tools you consider most important and/or contact website operators to investigate if/how this will be handled or how a name change can be initiated.

5.13 Wrapped Token Name Slightly Incorrect ✓ Fixed

Resolution

Fixed in PR 60 (reviewed commit a3b4eb454c590dce31405dfb4711954a873391f7) by changing the token name to “Wrapped TRUST”.

Description

As most ERC-20 tokens do, `WrappedTrust` has a `name` value. In this case, it is defined as `Wrapped Trust`:

src/WrappedTrust.sol:L19

```
string public name = "Wrapped Trust";
```

However, the original token that gets wrapped - `Trust` - actually has the name `Intuition` and symbol `TRUST`. As such, the name for the wrapped version should either be changed to `Wrapped Intuition` or `Wrapped TRUST`.

Recommendation

Adjust the `WrappedTrust` token name as appropriate.

5.14 Different Inheritance Orders in `BaseEmissionsController` and `SatelliteEmissionsController` ✓ Fixed

Resolution

Fixed in PR 61 (reviewed commit 3c940c60f284196c5d87c99499d0481128be0154) by aligning the order of inheritance in `SatelliteEmissionsController` and `BaseEmissionsController`.

Description

With the exception of their respective interface, both `BaseEmissionsController` and `SatelliteEmissionsController` inherit from the same contracts:

src/protocol/emissions/BaseEmissionsController.sol:L22-L27

```
contract BaseEmissionsController is
    IBaseEmissionsController,
    AccessControlUpgradeable,
    ReentrancyGuardUpgradeable,
    CoreEmissionsController,
    MetaERC20Dispatcher
```

src/protocol/emissions/SatelliteEmissionsController.sol:L22-L27

```
contract SatelliteEmissionsController is
    ISatelliteEmissionsController,
    AccessControlUpgradeable,
    ReentrancyGuardUpgradeable,
    MetaERC20Dispatcher,
    CoreEmissionsController
```

However, the order in which `CoreEmissionsController` and `MetaERC20Dispatcher` are inherited is reversed in these two contracts. While these two emissions controllers are technically independent of each other – and even deployed on different chains – they are intentionally similar, and having needless differences in the linearization order is at least inelegant.

Moreover, as initialization functions are supposed to be called from most basic to most derived (to mimic constructor behavior), the calling order of `__CoreEmissionsController_init` and `__MetaERC20Dispatcher_init` in `SatelliteEmissionsController` is, strictly speaking, wrong.

Recommendation

In the list of `SatelliteEmissionsController`’s inherited contracts, reverse the order of `MetaERC20Dispatcher` and `CoreEmissionsController`.

5.15 Notable Points Regarding the Calculation of Emissions for a Particular Epoch Acknowledged

Resolution

Acknowledged with the following note:

> Added tests demonstrating emissions would not round to zero until ~400 years post-start with our intended parameters - which means so no practical risk. No code change was required.

The associated PR 63 only adds tests and didn’t affect files in scope, so it wasn’t reviewed.

Description

The “base amount” of emissions for a particular epoch is calculated in `CoreEmissionsController._emissionsAtEpoch`. The idea is that a certain start value is repeatedly decreased by a certain percentage after a certain amount of epochs have passed. The formula for the base amount is:

```
_EMISSIONS_PER_EPOCH` * (`_EMISSIONS_RETENTION_FACTOR` / `BASIS_POINTS_DIVISOR`) ^ (floor(epoch / `_EMISSIONS_REDUCTION_CLIFF`)
```

Note that this is a mathematical formula, not Solidity code, i.e., the division `/` is meant to be an exact ratio. The symbol `^` denotes exponentiation, and is realized in the code with solady’s `FixedPointMathLib.rpow` function. This function implements exponentation by the well-known repeated-squaring method; notably, `rpow` doesn’t round intermediate results *down* but to the next representable number in the given precision – which can be up or down, depending on the number. Normally, the rounding direction should favor the protocol, but in this case we don’t expect any problems, since the Intuition team is free to choose any “base amount” for emissions they deem appropriate, and even a constant or growing amount wouldn’t be wrong per se.

A second aspect to keep in mind is the precision of the `rpow` calculation. For example, assume the following: `_EMISSIONS_PER_EPOCH = 10**27` (18 decimals), `_EMISSIONS_REDUCTION_CLIFF = 1`, and `_EMISSIONS_RETENTION_FACTOR = 1000`, which means a 90% reduction per epoch. The natural assumption might be that, after n epochs, the emitted amount is 10^(27-n), for n <= 27, and that is also what we’d receive if we divided by 10 in every round. However, while the emitted amount is indeed 10^9 after epoch 18, it drops immediately to 0 after epoch 19. The reason is that `rpow` is used with a precision of 18 decimals, so 10^-19 is not exactly representable anymore and rounded to 0, which makes the result of the multiplication 0 too.

Recommendation

To guard against such “quirks” and unwelcome surprises, we recommend the following: As soon as the actual values to be used have been determined by the team, run a test with these values and manually inspect the results for a sufficient number of epochs to make sure the actual numbers you’ll get are acceptable.

5.16 Miscellaneous Informational Points ✓ Fixed

Resolution

Fixed in PR 62 (reviewed commit 0751d8c9ae818b5904539821c0f8b48e9591f918) by addressing each of the points, namely:

- A. Removed unused imports.
- B. Removed the call to `__AccessControl_init()` in `TrustBonding`.
- C. Removed unused state variables `maxAnnualEmission` and `maxClaimableProtocolFeesForEpoch` from `TrustBonding`.
- D. Removed `recipientAddress` from the `MetaERC20DispatchInit` struct.
- E. Adjusted `_epochsPerYear()` visibility to `internal`.
- F. Adjusted the `TrustMintedAndBridged` event to include `_SATELLITE_EMISSIONS_CONTROLLER` instead of `address(this)`.
- G. Acknowledged the `immutable` variable recommendation but decided against to simplify future upgrades.
- H. Removed the `utilizationTarget == 0` check.
- I. Implemented more usage of helper functions, specifically `_calculateTotalEpochsToTimestamp()`.

Descriptions and Recommendations

A. Unused imports. There are several unused imports throughout the codebase. For example:

- In `TrustBonding.sol`, importing `Address`, `IERC20` and `SafeERC20` is not necessary. The `using SafeERC20 for IERC20;` can be removed as well. Importing `AccessControlUpgradeable` isn’t necessary either, as this is already imported by and inherited from `VotingEscrow`.
- In `SatelliteEmissionsController`, importing `Address`, `IERC20` and `SafeERC20` is not necessary. The `using SafeERC20 for IERC20;` can be removed as well.
- In `BaseEmissionsController`, importing `IERC20` and `Math` isn’t necessary.

B. `TrustBonding` doesn’t inherit *directly* from `AccessControlUpgradeable` but calls `__AccessControl_init`. As `TrustBonding` inherits from `VotingEscrow` – which in turn inherits from `AccessControlUpgradeable` and calls `__AccessControl_init` in its own initialization function `__VotingEscrow_init` – the `__AccessControl_init` call in `TrustBonding.initialize` should be removed:

src/protocol/emissions/TrustBonding.sol:L152

```
__AccessControl_init();
```

C. In `TrustBonding`, the state variables `maxAnnualEmission` and `maxClaimableProtocolFeesForEpoch` are not used. Unless some logic has been forgotten to implement, these can be removed.

D. The member `recipientAddress` in the struct `MetaERC20DispatchInit` is never used and can be removed.

E. In `TrustBonding`, the `_epochsPerYear` function is public but listed under “INTERNAL FUNCTIONS”. Note that there is already a public `epochsPerYear` function (without underscore), so `_epochsPerYear` (with underscore) is under the right heading but should be `internal`.

F. The `TrustMintedAndBridged` event is only emitted in one place:

src/protocol/emissions/BaseEmissionsController.sol:L168

```
emit TrustMintedAndBridged(address(this), amount, epoch);
```

Always emitting this event with `address(this)` as first parameter doesn’t make much sense, as the value is always the address we’re currently observing for events.

G. State variables that are only set once in a contract's initialization function and can't be changed later could, alternatively, be immutables. In that case, they'd have to be set in the constructor, not in the initialization function.
However, note that this has tie-ins with issue 5.3 – which should be solved first. Moreover, with immutables it won't be possible to deploy another proxy that points to the same implementation but uses different values; in such a case, a new implementation contract would have to be deployed too.

H. Consider the following check in `TrustBonding._getSystemUtilizationRatio`:

src/protocol/emissions/TrustBonding.sol:L556-L558

```
if (utilizationTarget == 0 || utilizationDelta >= utilizationTarget) {
    return BASIS_POINTS_DIVISOR;
}
```

If the first condition is fulfilled, the second is automatically too, since `utilizationDelta` is a `uint256`. So the part `utilizationTarget == 0 ||` could be removed. The same situation arises in `TrustBonding._getPersonalUtilizationRatio`.

I. Consider using more helper functions. In `CoreEmissionsController`, the `_calculateEpochEmissionsAt()` function could use the `_calculateTotalEpochsToTimestamp()` defined below to perform the calculation of `currentEpochNumber`:

src/protocol/emissions/CoreEmissionsController.sol:L184-L190

```
function _calculateEpochEmissionsAt(uint256 timestamp) internal view returns (uint256) {
    if (timestamp < _START_TIMESTAMP) {
        return 0;
    }

    // Calculate current epoch number
    uint256 currentEpochNumber = (timestamp - _START_TIMESTAMP) / _EPOCH_LENGTH;
```

src/protocol/emissions/CoreEmissionsController.sol:L199-L205

```
function _calculateTotalEpochsToTimestamp(uint256 timestamp) internal view returns (uint256) {
    if (timestamp < _START_TIMESTAMP) {
        return 0;
    }

    return (timestamp - _START_TIMESTAMP) / _EPOCH_LENGTH;
}
```

Appendix 1 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.1.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.1.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.1.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.