



Tn contracts

Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
RustyRabbit, Security Researcher

June 2, 2025

Contents

1 Introduction	2
1.1 About Cantina	2
1.2 Disclaimer	2
1.3 Risk assessment	2
1.3.1 Severity Classification	2
2 Security Review Summary	3
3 Findings	4
3.1 High Risk	4
3.1.1 Validator can bypass delegator for staking rewards	4
3.1.2 Missing validators in <code>_getValidators()</code> due to token ID gaps	4
3.1.3 Slashing penalties circumvented due to missing balance reset in <code>_consensusBurn()</code>	7
3.2 Medium Risk	8
3.2.1 Change of epoch Issuance takes effect in the current epoch potentially leading to higher rewards than expected	8
3.2.2 Missing BLS public key uniqueness check allows duplicate validator keys	9
3.3 Low Risk	11
3.3.1 Payable <code>permitWrap</code> function can silently lose user funds	11
3.3.2 <code>stake</code> does not validate all intended staking parameters	12
3.3.3 Invalid committee size check when ejecting validator	13
3.3.4 Missing validation in <code>upgradeStakeVersion()</code> function	14
3.3.5 Missing Transfer events in InterchainTEL's <code>_mint()</code> and <code>_burn()</code> functions	15
3.3.6 Incorrect eligible validators check during validator ejection	16
3.3.7 Rewards are payed out from <code>StakeManager</code> instead of <code>Issuance</code>	17
3.3.8 Missing zero committee size check in <code>_checkCommitteeSize()</code> function	18
3.3.9 Genesis validator stake allocation lacks explicit verification	19
3.3.10 Potential BLS key front-running in staking function	21
3.3.11 Validators may be unable to exit if continuously selected for committees	22
3.3.12 Permit front-running vulnerability in <code>permitWrap()</code> function	24
3.4 Gas Optimization	27
3.4.1 Code quality and optimization recommendations	27
3.5 Informational	30
3.5.1 Missing event emission for validator slashing	30
3.5.2 Incorrect balance check in <code>Issuance</code> 's <code>distributeStakeReward()</code>	31
3.5.3 <code>_updateEpochInfo</code> stores end block number of previous epoch instead of start block of the new epoch	32
3.5.4 Incorrect iTEL mint amount if <code>baseERC20</code> charges fees	32
3.5.5 <code>NewEpoch</code> event mixes information about different epochs	33
3.5.6 Consider separating the pauser role	33
3.5.7 Ineffective replay protection in validator delegation	33
3.5.8 Delegation risk and reward asymmetry in consensus system	35
3.5.9 Potential duplicate validators in committee without validation	36

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	<i>Must fix as soon as possible (if already deployed).</i>
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are rare combinations of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Telcoin creates low-cost, high-quality financial products for every mobile phone user in the world.

From May 8th to May 16th the Cantina team conducted a review of tn-contracts on commit hash a96328fa. The team identified a total of **27** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	3	0
Medium Risk	2	2	0
Low Risk	12	7	5
Gas Optimizations	1	1	0
Informational	9	6	3
Total	27	19	8

3 Findings

3.1 High Risk

3.1.1 Validator can bypass delegator for staking rewards

Severity: High Risk

Context: ConsensusRegistry.sol#L277-L297

Summary: There is a significant design issue in the ConsensusRegistry contract that allows a validator to claim rewards and unstake funds that should rightfully go to their delegator. This bypasses the delegation mechanism's intended purpose and could lead to the theft of rewards and staked funds from delegators.

Description: The ConsensusRegistry contract implements delegated staking to allow non Validators to stake on behalf of validators. However, the reward claiming and unstaking functions contain a logical flaw that allows validators to collect rewards and unstaked funds directly, bypassing the delegator who provided the stake. Both `claimStakeRewards()` and `unstake()` functions follow this pattern:

```
address recipient = validatorAddress;
if (msg.sender != validatorAddress) recipient = _checkKnownDelegation(validatorAddress, msg.sender);
```

This logic sets the `recipient` to the validator address by default and only checks for delegation if the caller is not the validator. This means that even if a validator has a delegator who provided the stake, the validator can directly call these functions and receive the rewards or unstaked funds for themselves.

Impact Explanation: The impact is high because this vulnerability allows validators to steal both:

1. Accumulated staking rewards that should go to their delegators.
2. The original staked funds provided by delegators during unstaking.

Likelihood Explanation: The likelihood is medium. While the vulnerability requires validators to act maliciously against their delegators, there is a clear financial incentive to do so, especially for validators with large delegated stakes or accumulated rewards. The code paths for both functions are readily accessible to validators without any technical barriers.

Recommendation: Modify the `claimStakeRewards()` and `unstake()` functions to always check for delegation regardless of who calls the function:

```
- address recipient = validatorAddress;
- if (msg.sender != validatorAddress) recipient = _checkKnownDelegation(validatorAddress, msg.sender);
+ address recipient = _getRecipient(validatorAddress);
+
+ // If caller is not the validator or delegator, revert
+ if (msg.sender != validatorAddress && msg.sender != recipient) {
+     revert NotValidatorOrDelegator(msg.sender);
+ }
```

Additionally, consider implementing a mechanism to split rewards between validators and delegators according to a predefined rate, rather than giving all rewards to one party.

Telcoin: This was a remnant of a previous spec where either validator or delegator could withdraw based on an assumption that delegation would only be arranged between validators and delegators with off-chain relationships. The current spec's invariant should be that the stake originator should always be the sole entity who receives all capital flows.

Resolved in commit [b3d8116](#).

Cantina Managed: The fix in commit [b3d8116](#) now allows either the validator or the recipient to unstake and claim rewards, however, the tokens will always be sent to the recipient regardless of the caller.

3.1.2 Missing validators in `_getValidators()` due to token ID gaps

Severity: High Risk

Context: ConsensusRegistry.sol#L576-L618

Summary: In the ConsensusRegistry contract, the `_getValidators()` function contains a flaw that can cause some validators to be missed when token IDs are not sequential. This occurs due to the function iterating based on `totalSupply` rather than the highest allocated validator ID.

Description: The `_getValidators()` function retrieves validator information based on their status. It loops through validator IDs from 1 to `totalSupply` to find matching validators:

```
function _getValidators(ValidatorStatus status) internal view returns (ValidatorInfo[] memory) {
    ValidatorInfo[] memory untrimmed = new ValidatorInfo[](totalSupply);
    uint256 numMatches;

    for (uint24 i = 1; i <= untrimmed.length; ++i) {
        ValidatorInfo storage current = validators[i];
        if (current.isRetired) continue;

        // Additional logic to check status...
    }

    // Trim and return result...
}
```

The issue arises when a validator's ConsensusNFT is burned, which decreases `totalSupply` but doesn't affect the ID sequence. For example:

1. Initial state: validators with IDs [1, 2, 3], `totalSupply` = 3.
2. Validator with ID 2 is burned: validators [1, 3], `totalSupply` = 2.
3. When `_getValidators()` runs, it only iterates up to `totalSupply` (2), missing validator with ID 3.

This problem affects all functions that rely on `_getValidators()`, including critical functions that manage the validator lifecycle and committee selection.

Impact Explanation: The impact of this issue is high as it can lead to:

1. Some active validators being excluded from committee selection.
2. Inaccurate validator counts for protocol operations.
3. Validators unable to claim rewards or participate in consensus despite being eligible.

The problem becomes more severe as the validator set grows and experiences more churn, creating larger gaps in the ID sequence.

Likelihood Explanation: The likelihood is medium. The issue only manifests when validator IDs become non-sequential due to burning tokens. In a stable network with low validator turnover, this might not occur frequently. However, it becomes increasingly likely as the network matures and validators join and leave over time.

Recommendation: Use ERC721Enumerable extension instead of the current approach. This would provide built-in enumeration capabilities and eliminate the need for custom tracking logic:

```
- abstract contract StakeManager is ERC721, EIP712, IStakeManager {
+ abstract contract StakeManager is ERC721Enumerable, EIP712, IStakeManager {
```

With `ERC721Enumerable`, you can use `tokenByIndex` to iterate through all existing tokens and rely on `totalSupply()` which correctly reflects the actual number of active tokens. The token IDs could directly correspond to the validator address making lookups more efficient.

Telcoin: This behavior was handled in the fuzz tests' by using a counter but that was not brought through the `_getValidators()` function as noted. Such a fix would be relatively simple without the use of `ERC721Enumerable`, like so:

```
function _getValidators(ValidatorStatus status) internal view returns (ValidatorInfo[] memory) {
    ValidatorInfo[] memory untrimmed = new ValidatorInfo[](totalSupply);
    uint256 numMatches;

    uint256 counter = untrimmed.length;
    for (uint24 i = 1; i <= counter; ++i) {
        ValidatorInfo storage current = validators[i];
        if (current.isRetired) { ++counter; continue; }
```

```

// queries for `Any` status include all unretired validators
bool matchFound = status == ValidatorStatus.Any;
if (!matchFound) {
    // mem cache to save SLOADs
    ValidatorStatus currentStatus = current.currentStatus;

    // include pending activation/exit due to committee service eligibility in next epoch
    if (status == ValidatorStatus.Active) {
        matchFound =
            currentStatus == ValidatorStatus.Active || currentStatus == ValidatorStatus.PendingExit
            || currentStatus == ValidatorStatus.PendingActivation
    };
} else {
    // all other queries return only exact matches
    matchFound = currentStatus == status;
}

if (matchFound) {
    untrimmed[numMatches++] = current;
}
}

// trim and return final array
ValidatorInfo[] memory validatorsMatched = new ValidatorInfo[](numMatches);
for (uint256 i; i < numMatches; ++i) {
    validatorsMatched[i] = untrimmed[i];
}

return validatorsMatched;
}

```

However, the general benefits of using an industry standard OZ implementation for enumerability and the improved mental model of making `tokenId == uint256(uint160validatorAddress)` are desirable and so the suggestion was implemented in [2e3343d](#).

Cantina Managed: [2e3343d](#) introduces the following changes:

1. StakeManager switched to `ERC721Enumerable`, removed the custom `totalSupply` counter, and uses the library's enumeration helpers everywhere (`totalSupply()`, `tokenByIndex`).
2. `_getValidators()` now iterates with:
 - `totalSupply()` from `ERC721Enumerable`.
 - `tokenByIndex(i) → _getAddress() → mapping validators[address]`.
 ⇒ every minted ConsensusNFT is checked, so gaps in token IDs no longer hide a validator.
3. All logic that previously relied on `uint24-indexed validators / stakeInfo` has been migrated to address-keyed `validators` and the new `balances` mapping.
4. Types expanded from `uint232 → uint256` for stake, rewards, issuance, slash amounts, header counts, etc...
5. Interface/struct and event order changes (`EpochInfo`, `Delegation`, `RewardInfo`, `Slash`) are reflected consistently in `ConsensusRegistry` and `StakeManager`.

Some considerations:

```

delegations[validatorAddress] =
    Delegation(blsPubkeyHash, msg.sender, validatorAddress, validatorVersion, nonce + 1);

```

We don't really need to store the `validatorAddress`, since that's known. It is required for the `DELEGATION_TYPEHASH`, however, that's just a string. We might not even need `validatorAddress` in `ValidatorInfo`, since it's indexed by the address already. However, it can be convenient when retrieving the information.

Telcoin: This is convenient for the protocol to retrieve information required for consensus.

Cantina Managed: Acknowledged.

3.1.3 Slashing penalties circumvented due to missing balance reset in `_consensusBurn()`

Severity: High Risk

Context: `ConsensusRegistry.sol#L483-L497`

Summary: In the ConsensusRegistry contract, the `_consensusBurn()` function fails to reset the validator's balance when ejecting a validator due to slashing or forced burning, leading to slashing penalties not being properly applied.

Description: The `_consensusBurn()` function is called from `applySlashes()` when a validator's balance would be reduced to zero after slashing, and also from the `burn()` function when a validator is forcefully removed. The function is responsible for ejecting the validator from committees, exiting, retiring, and unstaking them. However, it does not set the validator's balance to zero before unstaking. The issue arises from the condition in `applySlashes()` that calls `_consensusBurn()`:

```
if (info.balance > slash.amount) {
    info.balance -= slash.amount;
} else {
    // eject validators whose balance would reach 0
    _consensusBurn(tokenId, slash.validatorAddress);
}
```

Since `_consensusBurn()` doesn't set the balance to zero, the unstaking process in `_unstake()` will use the pre-slash balance (`bal`) which is inconsistent with the intent of the slashing mechanism:

```
function _unstake(
    address validatorAddress,
    address recipient,
    uint256 tokenId,
    uint8 validatorVersion
)
internal
virtual
returns (uint256)
{
    // wipe existing stakeInfo and burn the token
    StakeInfo storage info = stakeInfo[validatorAddress];
    uint232 bal = info.balance;
    info.balance = 0;
    info.tokenId = UNSTAKED;
    // ...

    // if slashed, consolidate remainder on the Issuance contract
    if (bal < stakeAmt) {
        (bool r,) = issuance.call{ value: stakeAmt - bal }("");
        r;
    }
}

return bal + rewards;
}
```

Impact Explanation: The impact is high. This issue results in slashing penalties not being properly applied when a validator is ejected from the protocol. The validator may receive funds they should have lost through slashing, which undermines a key security mechanism of the protocol. Additionally, this issue affects the forced burning of validators through the `burn()` function, which also calls `_consensusBurn()`.

Likelihood Explanation: The likelihood is medium. The issue will occur whenever a validator is slashed to the point where their balance would reach zero or when a validator is forcefully burned. While slashing may be a relatively rare event, it's a critical security feature that must work correctly when needed.

Recommendation: Add a line to set the validator's balance to zero in the `_consensusBurn()` function:

```

function _consensusBurn(uint24 tokenId, address validatorAddress) internal {
    // mark `validatorAddress` as spent using `UNSTAKED`
    stakeInfo[validatorAddress].tokenId = UNSTAKED;
+   stakeInfo[validatorAddress].balance = 0;

    // reverts if decremented committee size after ejection reaches 0, preventing network halt
    uint256 numEligible = _getValidators(ValidatorStatus.Active).length;
    _ejectFromCommittees(validatorAddress, numEligible);

    // exit, retire, and unstake + burn validator immediately
    ValidatorInfo storage validator = validators[tokenId];
    _exit(validator, currentEpoch);
    _retire(validator);
    address recipient = _getRecipient(validatorAddress);
    _unstake(validatorAddress, recipient, tokenId, validator.stakeVersion);
}

```

This ensures that a validator's balance is properly reset to zero during ejection, whether due to slashing or forced burning.

Telcoin: This was the intended behavior. The idea was to provide some lenience and make `burn()` usable as a force-unstake for validators that reach `EXIT` status but never bother to unstake/retire.

However this behavior (last slashes being converted to an unstake ejection without slashing) is not documented anywhere other than one word in the comment `// eject validators whose balance would reach 0 where "would" is used to imply differing behavior than otherwise would be normal (unstake ejection rather than slash)`.

I spoke to the team about this and we settled on changing the intended behavior to do as you both assumed, because.

- Force-unstaking a validator that exits but doesn't bother to unstake is extremely niche.
- Using `burn()` as a way to return validator stake doesn't fit well with the function name and main purpose.
- The differing behavior is not documented well, so assuming consistency for the last slash is intuitive.

The slashing spec's intended behavior should be amended and the finding should be addressed as valid.

As mentioned, the existing behavior while intentional was unintuitive and poorly documented. The recommendation was implemented in commit [41d4a58](#).

Cantina Managed: Fixed as recommended. The changes further include a check to allow unstaking if the validator has not begun the activation process yet.

3.2 Medium Risk

3.2.1 Change of epoch Issuance takes effect in the current epoch potentially leading to higher rewards than expected

Severity: Medium Risk

Context: `ConsensusRegistry.sol#L78`

Summary: One of the invariants of the `ConsensusRegistry` is that any change to the `StakeConfig` only takes effect in the next epoch, not the current. However when updating the `StakeConfig` the new issuance immediately takes effect, while the new duration only takes effect in the next epoch. This means the effective issuance per time unit of the epoch during which the `StakeConfig` is updated will be neither the old nor the new issuance schema. Depending on the values of both the issuance can be multiples of the intended amount.

Finding Description: When updating the `StakeConfig` the current version is increased and the new values are stored in the `versions` mapping.

```

function upgradeStakeVersion(StakeConfig calldata newConfig) external override onlyOwner whenNotPaused
→ returns (uint8)
{
    uint8 newVersion = ++stakeVersion;
    versions[newVersion] = newConfig;
    // ...
}

```

The duration of the current epoch was set when the previous epoch ended:

```

function _updateEpochInfo(address[] memory newCommittee) internal returns (uint32, uint32) {
    // ...
    uint32 newDuration = getCurrentStakeConfig().epochDuration;
    epochInfo[newEpochPointer] = EpochInfo(currentCommittee, uint64(block.number), newDuration);
    epochPointer = newEpochPointer;
    // ...
}

```

At the (correct) end of the current epoch the newly updated issuance is taken to calculate the validators' rewards:

```

function applyIncentives(RewardInfo[] calldata rewardInfos) public override onlySystemCall {
    // ...

    // derive and apply validator's weighted share of epoch issuance
    uint232 epochIssuance = getCurrentStakeConfig().epochIssuance;
    // ...
}

```

Impact Explanation: The impact is considered high as the issuance of the new StakeConfig combined with the old epochDuration could lead to a reward distribution several multiples of the intended amount.

Imagine a current reward of 100 TEL/day. When the StakeConfig would be updated to 700 TEL/7 days the resulting reward would be 700 TEL for 1 day in the epoch the change was executed.

Likelihood Explanation: The likelihood is considered medium as change of StakeConfig are expected to be infrequent.

Recommendation: Copy the epochIssuance from the current StakeConfig to the epochInfo at each epoch boundary in `_updateEpochInfo` as is done now for the epochDuration.

Telcoin: Good catch, this violates one of the contract's invariants which is that governance modification of stake config versions and should take effect in next epoch and not immediately, which includes the epoch issuance.

Resolved in commit [9f37cd1](#).

Cantina Managed: Fixed.

3.2.2 Missing BLS public key uniqueness check allows duplicate validator keys

Severity: Medium Risk

Context: `ConsensusRegistry.sol#L343-L369`

Summary: The `ConsensusRegistry` contract fails to validate the uniqueness of BLS public keys when staking, allowing multiple validators to use the same BLS key. This oversight could disrupt network consensus and reduce fault tolerance.

Description: In the `ConsensusRegistry` contract, the `_recordStaked()` function stores validator information without verifying that the BLS public key is unique:

```

function _recordStaked(
    bytes calldata blsPubkey,
    address validatorAddress,
    bool isDelegated,
    uint8 stakeVersion,
    uint24 tokenId,
    uint232 stakeAmt
)
internal
{
    ValidatorInfo memory newValidator = ValidatorInfo(
        blsPubkey,
        validatorAddress,
        PENDING_EPOCH,
        uint32(0),
        ValidatorStatus.Staked,
        false,
        isDelegated,
        stakeVersion
    );
    validators[tokenId] = newValidator;
    stakeInfo[validatorAddress].balance = stakeAmt;

    emit ValidatorStaked(newValidator);
}

```

The function accepts a BLS public key as input but doesn't check if it has been previously registered by another validator. A duplicate BLS key could cause several issues:

1. Validator discovery problems: If two validators use the same BLS key, there's a 50/50 chance of which record is returned during network queries.
2. Committee integrity issues: If two validators with the same BLS key are in the same committee, one's vote would be considered a double-vote, effectively reducing committee size.

As confirmed by the project team, the protocol requires unique BLS public keys, but this requirement is not enforced at the contract level.

Impact Explanation: The impact is medium to high. Duplicate BLS keys could:

1. Reduce effective committee size and network fault tolerance.
2. Disrupt peer discovery mechanisms.
3. Allow validators to unintentionally (or maliciously) interfere with each other.
4. Potentially lead to missed voting opportunities if votes are considered duplicates.

Likelihood Explanation: The likelihood is low to medium. While accidental duplication is unlikely, deliberate reuse is possible. A malicious actor could copy another validator's BLS public key, or a validator could reuse their own key across multiple validator identities.

Recommendation: Implement a check for BLS public key uniqueness in the staking functions:

```

+ mapping(bytes32 => bool) private usedBLSKeys;

function _recordStaked(
    bytes calldata blsPubkey,
    address validatorAddress,
    bool isDelegated,
    uint8 stakeVersion,
    uint24 tokenId,
    uint232 stakeAmt
)
internal
{
+    bytes32 blsPubkeyHash = keccak256(blsPubkey);
+    if (usedBLSKeys[blsPubkeyHash]) {
+        revert DuplicateBLSKey();
+    }
+    usedBLSKeys[blsPubkeyHash] = true;

    ValidatorInfo memory newValidator = ValidatorInfo(
        blsPubkey,
        validatorAddress,
        PENDING_EPOCH,
        uint32(0),
        ValidatorStatus.Staked,
        false,
        isDelegated,
        stakeVersion
    );
    validators[tokenId] = newValidator;
    stakeInfo[validatorAddress].balance = stakeAmt;

    emit ValidatorStaked(newValidator);
}

```

Also, add a new error:

```
error DuplicateBLSKey();
```

Telcoin: Great catch, resolved in commit [e666901](#).

Cantina Managed: Fixed as recommended.

3.3 Low Risk

3.3.1 Payable permitWrap function can silently lose user funds

Severity: Low Risk

Context: [InterchainTEL.sol#L98-L121](#)

Description: The `permitWrap()` function in the `InterchainTEL` contract is marked as `payable` but does not use the sent native funds, causing any ETH sent to the function to be permanently lost.

```

function permitWrap(
    address owner,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
)
    external
    payable
    virtual
{
    if (amount == 0) revert MintFailed(owner, amount);

    WETH wTEL = WETH(payable(address(baseERC20)));
    wTEL.permit(owner, address(this), amount, deadline, v, r, s);

    bool success = wTEL.transferFrom(owner, address(this), amount);
    if (!success) revert PermitWrapFailed(owner, amount);

    _mint(owner, amount);
    emit Wrap(owner, amount);
}

```

This is concerning because:

1. The contract also has a similar function `doubleWrap()` that is both marked as `payable` and actually uses the native funds sent to it.
2. Users might confuse these similar functions and expect `permitWrap()` to work with ETH like `doubleWrap()`.
3. There is no warning or revert when users mistakenly send ETH to `permitWrap()`.

Recommendation: Remove the `payable` keyword from the `permitWrap()` function:

```

function permitWrap(
    address owner,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
)
    external
    - payable
    virtual
{
    // function body...
}

```

Telcoin: Good find, a user error that accidentally includes native TEL when calling would result in permanently burned funds. To prevent the possibility of such an error, function's payability is removed in commit [2776cdd](#).

Cantina Managed: Fixed as recommended.

3.3.2 stake does not validate all intended staking parameters

Severity: Low Risk

Context: [ConsensusRegistry.sol#L177-L189](#)

Summary: When a user stakes they assume the rules they do so under is conform to the current version of `stakeConfig`. The `stakeAmount` however is the only thing that is verified.

When a user happens to transmit their transaction after a `upgradeStakeVersion` that uses the same `stakeAmount` but changes other parameters the user will unknowingly stake under the rules of the new '`stakeConfig`'. This could be a longer duration, a lower issuance or a higher minimal withdrawal amount.

Finding Description: When governance changes the parameters of the `StakeConfig` via `upgradeStakeVersion` it specifies the `takeAmount`, `minWithdrawAmount`, `epochIssuance` and `epochDuration`:

```

struct StakeConfig {
    uint232 stakeAmount;
    uint232 minWithdrawAmount;
    uint232 epochIssuance;
    uint32 epochDuration;
}

```

When a validator stakes they implicitly do so under the current `stakeConfig`.

```

function stake(bytes calldata blsPubkey) external payable override whenNotPaused {
    // ...
    uint8 validatorVersion = stakeVersion;
    uint232 stakeAmt = _checkStakeValue(msg.value, validatorVersion);
    // ...
    _recordStaked(blsPubkey, msg.sender, false, validatorVersion, tokenId, stakeAmt);
}

```

If however some time before the stake transaction is submitted a `stakeConfig` update has happened that keeps the same `stakeAmount` the check on the `msg.value` will pass but the other staking parameters that apply will be those of the newer version.

Impact Explanation: The impact is medium as the user is unaware of the modified rules their stake has been processed under and they will only discover this at the end of the next epoch. If there are insufficient active validators the user also may be unable unstake for longer time than expected.

Likelihood Explanation: The likelihood is low as `stakeConfig` changes will not be frequent.

Recommendation (optional): Additionally to the `msg.value` check also checks that the version the user intended to stake under corresponds to the current version.

Telcoin: Stake versions are assigned at stake time and not whitelist time- this means that when governance sets a new `StakeConfig` version, stake versions should take effect for all validators/stakers at the same time, *including those who have already been whitelisted*. This is why the `structHash` used by `delegationDigest()` uses the current stake version to craft the digest that validators sign, which is an explicit cryptographic endorsement by the validator for the current version.

In that sense, the behavior described is correct: the only relevant check is that the `msg.value` provided to `stake()` or `delegateStake()` matches that of the current version.

However, new stake versions should take effect only *after the current epoch in which it was set*, ie after rolling over into a new epoch. This is outlined explicitly by the invariant describing migration to new stake versions.

The current behavior does not comply with that invariant: stake actions made after governance has made a configuration change but before advancing to the next epoch will assign the validator to the version which should not yet have taken effect. Accordingly, a fix is implemented in commit [ccf4ccd](#).

Cantina Managed: Fixed.

3.3.3 Invalid committee size check when ejecting validator

Severity: Low Risk

Context: [ConsensusRegistry.sol#L452-L481](#)

Description: In the `ConsensusRegistry` contract, the `_ejectFromCommittees` function contains a logical error when checking the resulting committee size after ejecting a validator. The function assumes that the validator being ejected is always a member of the committee, which may not be true. The problematic code is in the `_ejectFromCommittees` function:

```

address[] storage currentCommittee = _getRecentEpochInfo(current, current, currentEpochPointer).committee;
_checkCommitteeSize(numEligible, currentCommittee.length - 1);
_eject(currentCommittee, validatorAddress);

```

The issue is that the function calls `_checkCommitteeSize(numEligible, currentCommittee.length - 1)` before actually ejecting the validator, assuming that the post-ejection size will be `currentCommittee.length - 1`. However, if the validator is not actually in the committee, the actual post-ejection size would still be `currentCommittee.length`, making the check incorrect.

The `_eject` function does not return any value indicating whether an ejection occurred:

```
function _eject(address[] storage committee, address validatorAddress) internal {
    uint256 len = committee.length;
    for (uint256 i; i < len; ++i) {
        if (committee[i] == validatorAddress) {
            committee[i] = committee[len - 1];
            committee.pop();
            break;
        }
    }
}
```

If the validator is not found in the committee, the function simply completes without modifying the committee size.

Recommendation: Modify the `_eject` function to return a boolean indicating whether an ejection occurred, and update the committee size check accordingly:

```
- function _eject(address[] storage committee, address validatorAddress) internal {
+ function _eject(address[] storage committee, address validatorAddress) internal returns (bool) {
    uint256 len = committee.length;
    for (uint256 i; i < len; ++i) {
        if (committee[i] == validatorAddress) {
            committee[i] = committee[len - 1];
            committee.pop();
-            break;
+            return true;
        }
    }
+    return false;
}
```

Then update the call sites to check the actual post-ejection committee size:

```
address[] storage currentCommittee = _getRecentEpochInfo(current, current, currentEpochPointer).committee;
- _checkCommitteeSize(numEligible, currentCommittee.length - 1);
- _eject(currentCommittee, validatorAddress);
+ bool ejected = _eject(currentCommittee, validatorAddress);
+ uint256 committeeSize = ejected ? currentCommittee.length - 1 : currentCommittee.length;
+ _checkCommitteeSize(numEligible, committeeSize);
```

This ensures that the committee size check is accurate regardless of whether the validator was actually in the committee or not.

Telcoin: Thank you, ejection results are now incorporated during consensus burn to ensure correct committee size checks as of commit [2ed52ff](#).

Cantina Managed: Fixed as recommended.

3.3.4 Missing validation in `upgradeStakeVersion()` function

Severity: Low Risk

Context: [ConsensusRegistry.sol#L717-L729](#)

Description: The `upgradeStakeVersion()` function in the `ConsensusRegistry` contract lacks proper validation of the input parameters, potentially allowing configuration values that could break protocol functionality.

```
function upgradeStakeVersion(StakeConfig calldata newConfig)
    external
    override
    onlyOwner
    whenNotPaused
    returns (uint8)
{
    uint8 newVersion = ++stakeVersion;
    versions[newVersion] = newConfig;

    return newVersion;
}
```

This function accepts a new StakeConfig struct containing four key protocol parameters:

- `stakeAmount`: Required amount for staking.
- `minWithdrawAmount`: Minimum rewards required for withdrawal.
- `epochIssuance`: Rewards distributed per epoch.
- `epochDuration`: Length of an epoch.

However, the function doesn't perform any validation on these values, which could lead to several issues:

1. If `minWithdrawAmount` is set too high, rewards might become unclaimable.
2. If `stakeAmount` is set too low or too high, it could undermine the economic security model.
3. If `epochDuration` is set to an unreasonable value, it could disrupt the consensus process.
4. If `epochIssuance` is set incorrectly, it could lead to inflation issues or insufficient incentives.

Recommendation: Add validation checks for all configuration parameters:

```
function upgradeStakeVersion(StakeConfig calldata newConfig)
    external
    override
    onlyOwner
    whenNotPaused
    returns (uint8)
{
    // Stake amount should be within reasonable bounds
    if (newConfig.stakeAmount < MIN_STAKE_AMOUNT || newConfig.stakeAmount > MAX_STAKE_AMOUNT) {
        revert InvalidStakeAmount(newConfig.stakeAmount);
    }

    // Minimum withdraw amount should be a fraction of stake amount
    if (newConfig.minWithdrawAmount > newConfig.stakeAmount / 10) {
        revert InvalidMinWithdrawAmount(newConfig.minWithdrawAmount);
    }

    // Epoch duration should be reasonable
    if (newConfig.epochDuration < MIN_EPOCH_DURATION || newConfig.epochDuration > MAX_EPOCH_DURATION) {
        revert InvalidEpochDuration(newConfig.epochDuration);
    }

    // Epoch issuance shouldn't exceed maximum inflation rate
    if (newConfig.epochIssuance > MAX_EPOCH_ISSUANCE) {
        revert InvalidEpochIssuance(newConfig.epochIssuance);
    }

    uint8 newVersion = ++stakeVersion;
    versions[newVersion] = newConfig;

    return newVersion;
}
```

The specific bounds (`MIN_STAKE_AMOUNT`, `MAX_STAKE_AMOUNT`, etc.) should be determined based on the protocol's economic model and requirements. Consider implementing these as constants at the contract level.

Telcoin: After speaking with the team, explicit validation of governance-supplied config parameters is seen as desirable but too early to strictly implement ahead of MNO testnet pilot. Also worth noting is that these config parameters will be voted on by the community. For flexibility during the MNO testnet pilot phase, only validation of epoch duration was added with the intent to add further and stricter validations to sanity-check governance as we move toward mainnet. See commit [b6408f7](#).

Cantina Managed: Fixed as recommended.

3.3.5 Missing Transfer events in InterchainTEL's `_mint()` and `_burn()` functions

Severity: Low Risk

Context: [InterchainTEL.sol#L123-L141](#)

Description: The InterchainTEL contract overrides the `_mint()` and `_burn()` functions but does not emit the standard ERC20 Transfer events, making the contract non-compliant with the ERC20 standard. This can cause issues with third-party applications, tools, and interfaces that rely on these events for tracking token movements.

In the ERC20 standard, Transfer events must be emitted for all token transfers, including minting (transfer from address(0)) and burning (transfer to address(0)). The InterchainTEL contract overrides these functions without ensuring these events are properly emitted:

```
function _mint(address account, uint256 amount) internal virtual override whenNotPaused {
    if (account == address(0)) revert ZeroAddressNotAllowed();
    _clean(account);

    // 10e12 TEL supply can never overflow w/out inflating 27 orders of magnitude
    uint128 bytes16Amount = SafeCastLib.toUint128(amount);
    _unsettledRecords[account].enqueue(bytes16Amount, block.timestamp + recoverableWindow);

    _totalSupply += amount;
    _accountState[account].nonce++;
    _accountState[account].balance += bytes16Amount;
    // Missing Transfer event
}

function _burn(address account, uint256 amount) internal virtual override whenNotPaused {
    super._burn(account, amount);
    // Transfer event is not emitted by super._burn() in RecoverableWrapper
}
```

The parent implementation (`RecoverableWrapper`) does not emit the required `Transfer` events, and neither does the `InterchainTEL` contract. This makes the contract non-compliant with the ERC20 standard and can cause issues with tracking token minting and burning operations.

Recommendation: Add the missing `Transfer` events to both functions:

```
function _mint(address account, uint256 amount) internal virtual override whenNotPaused {
    if (account == address(0)) revert ZeroAddressNotAllowed();
    _clean(account);

    // 10e12 TEL supply can never overflow w/out inflating 27 orders of magnitude
    uint128 bytes16Amount = SafeCastLib.toUint128(amount);
    _unsettledRecords[account].enqueue(bytes16Amount, block.timestamp + recoverableWindow);

    _totalSupply += amount;
    _accountState[account].nonce++;
    _accountState[account].balance += bytes16Amount;
+     emit Transfer(address(0), account, amount);
}

function _burn(address account, uint256 amount) internal virtual override whenNotPaused {
    super._burn(account, amount);
+     emit Transfer(account, address(0), amount);
}
```

This ensures compatibility with the ERC20 standard and allows external applications to correctly track token minting and burning operations. Without these events, block explorers, analytics platforms, and other tools will not be able to properly track the token supply and transfers.

Telcoin: Added in commit [7e0d37b](#).

Cantina Managed: Fixed as recommended.

3.3.6 Incorrect eligible validators check during validator ejection

Severity: Low Risk

Context: [ConsensusRegistry.sol#L483-L497](#)

Summary: The `ConsensusRegistry` contract incorrectly calculates committee sizes when ejecting validators, potentially leading to invalid committee composition. The validation logic doesn't account for the validator being ejected possibly being part of the active set, causing the post-ejection committee size check to use an inflated validator count.

Description: In the `_consensusBurn()` function of the `ConsensusRegistry` contract, there is a potential issue with the way the number of eligible validators is calculated before ejecting a validator from committees.

The issue is that `numEligible` is calculated before the validator has been ejected from the active set. If the validator being ejected is part of the active validators (which is often the case), then the actual number of eligible validators after ejection will be `numEligible - 1`.

This could lead to incorrect committee size validation in the `_ejectFromCommittees()` function, which uses `numEligible` to check if the committee would still have enough validators after ejection:

```
function _ejectFromCommittees(address validatorAddress, uint256 numEligible) internal {
    // ...
    address[] storage currentCommittee = _getRecentEpochInfo(current, current, currentEpochPointer).committee;
    _checkCommitteeSize(numEligible, currentCommittee.length - 1);
    // ...
}
```

This validation ensures that committees always maintain a minimum size to keep the network operational.

Impact Explanation: The impact is medium. This issue could allow committees to be reduced below their intended minimum size, potentially affecting network stability and consensus.

Likelihood Explanation: The likelihood is low. This issue only manifests when ejecting a validator that is part of the active set, and validator ejection is typically a governance action that happens infrequently. Multiple validators would need to be ejected in quick succession to cause significant problems.

Recommendation: Modify the code to account for the validator being removed from the active set by subtracting 1 from `numEligible` if the validator being burned is currently active. Also cache the validator status for efficiency:

```
function _consensusBurn(uint24 tokenId, address validatorAddress) internal {
    // mark `validatorAddress` as spent using `UNSTAKED`
    stakeInfo[validatorAddress].tokenId = UNSTAKED;

    // Get validator info and status once for efficiency
    ValidatorInfo storage validator = validators[tokenId];
    ValidatorStatus status = validator.currentStatus;

    // reverts if decremented committee size after ejection reaches 0, preventing network halt
    uint256 numEligible = _getValidators(ValidatorStatus.Active).length;
    // If the validator being ejected is active, subtract 1 from numEligible
    if (status == ValidatorStatus.Active ||
        status == ValidatorStatus.PendingActivation ||
        status == ValidatorStatus.PendingExit) {
        numEligible = numEligible - 1;
    }
    _ejectFromCommittees(validatorAddress, numEligible);

    // exit, retire, and unstake + burn validator immediately
    ValidatorInfo storage validator = validators[tokenId];
    _exit(validator, currentEpoch);
    _retire(validator);
    address recipient = _getRecipient(validatorAddress);
    _unstake(validatorAddress, recipient, tokenId, validator.stakeVersion);
}
```

Telcoin: Confirming this is valid:

- In certain cases, the governance owner invoking `burn()` can cause an incorrect `numEligible` value to be fed into `_ejectFromCommittees() => _checkCommitteeSize()`.

Resolved in commit [523ba74](#).

Cantina Managed: Fixed as recommended.

3.3.7 Rewards are payed out from StakeManager instead of Issuance

Severity: Low Risk

Context: StakeManager.sol#L173-L203

Summary: Because of a recent refactoring a discrepancy emerged in the handling of funds between the StakeManager and the Issuance contract. The net effect is that the StakeManager is forwarding the rewards to the Issuance contract but also incorrectly indicates no rewards to be payed out from the Issuance balance.

Finding Description: This finding is a combination of two inconsistencies where the StakeManager forwards the total balance of a validator which includes the accumulated rewards to the Issuance contract.

Then it calculates the rewards as the surplus above the initial stake to indicate to the Issuance contract the amount to be added from the balance of the Issuance contract. However before this calculation is performed the balance is set to 0 which results in an indicated reward of 0.

The net result is that the StakeManager is supplying the rewards from what is supposed to be reserved for staked funds.

Impact Explanation: As the net result is actually the correct behavior regarding the reward payout, it is considered a Low. However the rewards are taken from the stake of other validators which can lead to other validator not being able to unstake unless the incorrectly removed staking funds are re-supplied by governance.

Likelihood Explanation: The likelihood is high as it will happen every time a user unstakes.

Recommendation: Calculate the rewards before setting the balance in storage to 0 and limit the forwarded amount from the Stakemanager to the initial stake.

Telcoin: Thank you, this is an odd one and can indeed reach an invalid state under certain conditions where the reward payouts exceed the total funds staked on the registry. Resolved in commit [1c3b0b5](#).

Cantina Managed: Fixed.

3.3.8 Missing zero committee size check in `_checkCommitteeSize()` function

Severity: Low Risk

Context: ConsensusRegistry.sol#L550-L556

Description: In the ConsensusRegistry contract, the `_checkCommitteeSize()` function is responsible for ensuring that committees maintain a valid size. However, it does not explicitly check if `committeeSize` is zero, which could potentially lead to an empty committee.

```
function _checkCommitteeSize(uint256 activeOrPending, uint256 committeeSize) internal pure {
    if (activeOrPending == 0 || committeeSize > activeOrPending) {
        revert InvalidCommitteeSize(activeOrPending, committeeSize);
    }
}
```

The function checks if:

1. `activeOrPending` is zero - which reverts if there are no active validators.
2. `committeeSize` is greater than `activeOrPending` - which enforces that committees don't exceed available validators.

However, it doesn't explicitly check if `committeeSize` is zero. An empty committee would be technically valid according to this function, but would prevent the network from reaching consensus, effectively causing a network halt.

Recommendation: Add an explicit check for zero committee size:

```
function _checkCommitteeSize(uint256 activeOrPending, uint256 committeeSize) internal pure {
-    if (activeOrPending == 0 || committeeSize > activeOrPending) {
+    if (activeOrPending == 0 || committeeSize == 0 || committeeSize > activeOrPending) {
        revert InvalidCommitteeSize(activeOrPending, committeeSize);
    }
}
```

Telcoin: This is technically true, however it's worth noting that committees cannot reach zero because the protocol derives committees and feeds them into `concludeEpoch()` by reading from the registry's `getValidators(Active)` state. This means the only way `committeeSize` could reach zero is if `activeOrPending` validators first reaches zero (which the contract blocks in this fn), after which the protocol would

read `getValidators(Active)` to assemble an empty committee and then feed it into the registry via `concludeEpoch()`.

However, including the check as recommended provides explicitness and clearly demonstrates the invariant that committees should never be empty, without needing to consult the protocol's rust codebase. Since the invalid state is not possible given the protocol implementation (which is admittedly out of scope), we believe this finding should be downgraded to informational and have implemented the suggestion for better clarity in commit [962e1e7](#).

Cantina Managed:

This means the only way `committeeSize` could reach zero is if `activeOrPending` validators first reaches zero (which the contract blocks in this fn), after which the protocol would read `getValidators(Active)` to assemble an empty committee and then feed it into the registry via `concludeEpoch()`.

We'd still argue that it would be important to include this check as this is an out of scope component which we cannot verify. And this might be true for `concludeEpoch()`, however, I don't think the same applies to `applySlashes()` (yet, this might also be checked by out of scope components) and the admin-only `burn()` function. For the `burn()` function alone, this finding should be valid.

Telcoin: Understandable- in that case we accept the low severity label, especially considering we made the change as recommended in commit [962e1e7](#).

Cantina Managed: Fixed as recommended.

3.3.9 Genesis validator stake allocation lacks explicit verification

Severity: Low Risk

Context: `ConsensusRegistry.sol#L667-L714`

Description: In the `ConsensusRegistry` contract, genesis validators are assigned stake without a clear mechanism for capturing or verifying the corresponding funds. During initialization, the contract assigns stake to validators:

```
validators[tokenId] = currentValidator;
stakeInfo[currentValidator.validatorAddress].tokenId = tokenId;
stakeInfo[currentValidator.validatorAddress].balance = genesisConfig_.stakeAmount;
totalSupply++;
_mint(currentValidator.validatorAddress, tokenId);
```

However, there's no explicit mechanism in the constructor to:

1. Collect the required stake from validators.
2. Verify the protocol has reserved sufficient funds.
3. Ensure the accounting of stake matches actual available TEL.

According to a project comment, genesis validator stake "*is done from the protocol side and decremented directly from the InterchainTEL contract during genesis*". However, the `InterchainTEL` contract doesn't contain a visible function for this purpose.

This creates a potential gap between the accounting of stake in the `ConsensusRegistry` contract and the actual TEL available in the system, especially since there's no clear enforcement mechanism to ensure the protocol correctly allocates sufficient funds for genesis validators.

Recommendation: Consider strengthening the genesis stake allocation mechanism through one of these approaches:

1. Make the `ConsensusRegistry` constructor payable and require it to receive funds equal to the total stake amount needed for all genesis validators.
2. Implement a clear integration mechanism between `ConsensusRegistry` and `InterchainTEL` that verifies and records the stake allocation during genesis.
3. At minimum, document the genesis stake allocation process thoroughly, including how the protocol ensures consistency between the recorded stake and actual available TEL.

This would improve transparency and security by ensuring the protocol's stake accounting is backed by real funds from the beginning.

Telcoin: Acknowledged - Opt for suggestion 3: "*At minimum, document the genesis stake allocation process thoroughly, including how the protocol ensures consistency between the recorded stake and actual available TEL*".

- Genesis Stake Allocation Process Documentation:

- Overview: Telcoin Network implements a secure mechanism for allocating stake to genesis validators through a pre-genesis execution process in the protocol layer. This document outlines how the protocol ensures consistency between the recorded stake in the ConsensusRegistry contract and the actual available TEL tokens.

- Process Flow:

1. Pre-Genesis TEL Bridging:

- * The native TEL tokens exist as ERC-20 on Ethereum.
- * TEL tokens are prepared for bridging via Axelar before the network launches.
- * The bridging transaction is queued but not executed until genesis.

2. Genesis Committee Creation:

- * The `CreateCommitteeArgs::execute()` function is called during network initialization.
- * Initial stake configuration is defined with parameters for stake amount, withdrawal limits, and rewards.
- * Total required stake is calculated as `initial_stake * number_of_validators`.

3. Pre-Genesis Contract Execution:

- * The protocol executes the `ConsensusRegistry` contract constructor in a temporary environment.
- * This generates the correct storage layout and bytecode that will be included in genesis.
- * Validators are registered with their stake amounts in this pre-execution phase.

4. InterchainTEL Balance Adjustment:

- * The total TEL supply is decremented by the amount required for genesis validators.
- * `genesis_stake = initial_stake * validators.len()`.
- * `itel_balance = total_supply - genesis_stake`.
- * This ensures the `InterchainTEL` contract has the correct reduced balance at genesis.

5. Genesis Account Creation:

- * `ConsensusRegistry` is deployed at a predetermined address with:
- * Total stake balance equal to all validators' combined stake.
- * Runtime bytecode and storage state from pre-execution.
- * `InterchainTEL` contract is deployed with the adjusted balance.

6. Genesis Finalization:

- * The modified genesis state is written to the genesis file.
- * All validators start with the same genesis state, ensuring consensus.

- Fund Consistency Verification: The protocol ensures consistency between recorded stake and actual available TEL through:

1. Explicit Balance Deduction:

```

let genesis_stake = self
    .initial_stake
    .checked_mul(U232::from(validators.len()))
    .expect("initial validators' stake");
let itel_balance = U256::from(
    clap_u232_parser("100_000_000_000")? - genesis_stake
);

```

2. Atomic Genesis Creation:

- * The stake allocation and corresponding token balance reduction happen in a single atomic genesis creation process.
- * This prevents any inconsistency between allocated stake and available tokens.

3. Pre-Genesis Validation:

- * Before finalizing genesis, validator information is validated.
- * This ensures all validators have proper credentials before stake is allocated.

4. Blockchain Verification:

- * Once the network launches, the bridged transaction from Ethereum is executed.
- * This completes the flow of TEL tokens from Ethereum to the Telcoin Network.
- * The token balances in the system will match the pre-allocated stake amounts.

- Security Guarantees:

- * No Double Counting: The process explicitly deducts the total validator stake from the Inter-chainTEL contract's balance, preventing inflation.
- * Mathematical Verification: Checks include `checked_mul` operations to prevent overflow and ensure accurate accounting.
- * Deterministic Addresses: The ConsensusRegistry is deployed at a predetermined address, ensuring all validators agree on its location.
- * Pre-Execution Validation: The contract's behavior is verified in a test environment before being included in genesis.

Cantina Managed: This is definitely useful for future audits and integrators.

- Re. "*TEL tokens are prepared for bridging via Axelar before the network launches*", are these TEL tokens provided by the Telcoin team or also users?
- `clap_u232_parser("100_000_000_000")? - genesis_stake`: Is it guaranteed that 100_000_000_000 tokens will be bridged in total to back up the `iTEL` balance (or is this simply a placeholder?).
- Will the `genesis_stake` have a guaranteed backing by Telcoin tokens (and not users)?

Telcoin:

- The TEL tokens for genesis are provided by governance and will be handled by the protocol team only.
- The `iTEL` balance is simply a placeholder. The tokens that aren't bridged over will be locked in `iTEL`.
- Yes, these are the tokens that governance will transfer through the bridge.

Cantina Managed: Acknowledged.

3.3.10 Potential BLS key front-running in staking function

Severity: Low Risk

Context: `ConsensusRegistry.sol#L176-L189`

Description: The `stake()` function in the `ConsensusRegistry` contract accepts a BLS public key without verifying its uniqueness, which could allow front-running of BLS keys:

```

function stake(bytes calldata blsPubkey) external payable override whenNotPaused {
    if (blsPubkey.length != 96) revert InvalidBLSPubkey();

    // require caller is known & whitelisted, having been issued a ConsensusNFT by governance
    uint8 validatorVersion = stakeVersion;
    uint232 stakeAmt = _checkStakeValue(msg.value, validatorVersion);
    uint24 tokenId = _checkConsensusNFTOwner(msg.sender);
    // require validator has not yet staked
    _checkValidatorStatus(tokenId, ValidatorStatus.Undefined);

    // enter validator in activation queue
    _recordStaked(blsPubkey, msg.sender, false, validatorVersion, tokenId, stakeAmt);
}

```

While unlikely, a malicious validator could potentially front-run another validator's transaction and register with the same BLS public key. Though the front-runner would still be subject to staking requirements and slashing risks, this could lead to:

1. Confusion in the protocol's node identification.
2. Potential issues with attributing consensus participation correctly.
3. Complications in reward distribution and slashing.

The same issue exists in the `delegateStake()` function where a BLS key is also provided without uniqueness verification.

Recommendation: Add a uniqueness check for BLS public keys to prevent the same key from being used by multiple validators to both `stake()` and `delegateStake()` functions. For a more robust solution, consider implementing a commit-reveal scheme for BLS keys to prevent front-running entirely, though this would add complexity to the staking process.

Telcoin: Acknowledged: The protocol enforces a proof of possession of BLS public keys on the rust side, so a malicious validator intending to frontrun not only needs to provide stake but will not pass said proof of possession check. Assuming they paid the stake to perform the frontrunning, governance is then able to step in and slash their stake via consensus burn and resolve the grieved state. Thus the attack is not an economically attractive.

Cantina Managed: Acknowledged.

3.3.11 Validators may be unable to exit if continuously selected for committees

Severity: Low Risk

Context: `ConsensusRegistry.sol#L41-L52`

Description: The `ConsensusRegistry` contract has a potential issue where validators who wish to exit the network might be prevented from doing so if they are continuously selected for future committees. This issue stems from the exit mechanism in the `_updateValidatorQueue()` function:

```

function _updateValidatorQueue(address[] calldata futureCommittee, uint32 current) internal {
    // ... activation logic ...

    ValidatorInfo[] memory pendingExit = _getValidators(ValidatorStatus.PendingExit);
    for (uint256 i; i < pendingExit.length; ++i) {
        // skip if validator is in current or either future committee
        uint8 currentEpochPointer = epochPointer;
        uint8 nextEpochPointer = (currentEpochPointer + 1) % 4;
        address[] memory currentCommittee = epochInfo[currentEpochPointer].committee;
        address[] memory nextCommittee = futureEpochInfo[nextEpochPointer].committee;
        address validatorAddress = pendingExit[i].validatorAddress;
        if (
            _isCommitteeMember(validatorAddress, currentCommittee)
            || _isCommitteeMember(validatorAddress, nextCommittee)
            || _isCommitteeMember(validatorAddress, futureCommittee)
        ) continue;

        uint24 tokenId = _getTokenId(validatorAddress);
        ValidatorInfo storage exitValidator = validators[tokenId];
        _exit(exitValidator, current);
    }
}

```

If a validator calls `beginExit()` to initiate the exit process, they enter the `PendingExit` status. However, they are only allowed to fully exit when they are no longer needed for committee service across three epochs (current, next, and the one after that).

The issue arises when the committee selection algorithm continuously selects the validator for future committees. If a validator is consistently chosen for committee service, they could be stuck in `PendingExit` status indefinitely with no guaranteed way to complete their exit from the network.

Recommendation: Implement a maximum time limit for validators in `PendingExit` status to ensure they can eventually exit regardless of committee selection. This could be done by adding an "exit requested" timestamp when `beginExit()` is called and forcing exit after a certain number of epochs:

```

function _updateValidatorQueue(address[] calldata futureCommittee, uint32 current) internal {
    // ... activation logic ...

    ValidatorInfo[] memory pendingExit = _getValidators(ValidatorStatus.PendingExit);
    for (uint256 i; i < pendingExit.length; ++i) {
        address validatorAddress = pendingExit[i].validatorAddress;
        uint24 tokenId = _getTokenId(validatorAddress);
        ValidatorInfo storage exitValidator = validators[tokenId];

        // Force exit after MAX_EXIT_WAIT epochs (e.g., 10 epochs)
        if (exitValidator.exitRequestEpoch + MAX_EXIT_WAIT <= current) {
            _exit(exitValidator, current);
            continue;
        }

        // Regular exit logic
        // skip if validator is in current or either future committee
        uint8 currentEpochPointer = epochPointer;
        uint8 nextEpochPointer = (currentEpochPointer + 1) % 4;
        address[] memory currentCommittee = epochInfo[currentEpochPointer].committee;
        address[] memory nextCommittee = futureEpochInfo[nextEpochPointer].committee;

        if (
            _isCommitteeMember(validatorAddress, currentCommittee)
            || _isCommitteeMember(validatorAddress, nextCommittee)
            || _isCommitteeMember(validatorAddress, futureCommittee)
        ) continue;

        _exit(exitValidator, current);
    }
}

```

This change would require adding an `exitRequestEpoch` field to the `ValidatorInfo` struct and updating the `beginExit()` function to record the current epoch when exit is requested.

Telcoin: Acknowledged. Telcoin Network prioritizes maintaining Byzantine Fault Tolerance (BFT) in validator committees, and this takes precedence over individual validator exit requests.

When a validator requests to exit but is continuously selected for committee service, this indicates the network needs their participation to maintain BFT. Rather than implementing an automatic time-based exit that could compromise network security, we've designed our governance system to handle this scenario:

1. If a validator wants to exit but cannot due to continuous committee selection, this signals to governance that the protocol is at risk of losing BFT.
2. Governance is then expected to proactively spin up its own validator node to replace the one that wishes to exit.
3. Once the governance node is fully synced and ready, it can replace the validator in PendingExit status.

The timing of this process is intentionally not hardcoded because it depends on variables like node sync time and epoch duration at that moment. Forcing an automatic exit after a predetermined number of epochs could result in network failure if a replacement isn't ready.

We believe this approach prioritizes network security while providing a governance-managed exit path for validators. That said, we recognize this is an edge case that should be clearly documented, and we'll update our documentation to make this design decision more explicit.

Cantina Managed: Acknowledged.

3.3.12 Permit front-running vulnerability in `permitWrap()` function

Severity: Low Risk

Context: [InterchainTEL.sol#L98-L121](#)

Description: The `permitWrap()` function in the `InterchainTEL` contract is vulnerable to front-running attacks that could cause it to fail unexpectedly:

```
function permitWrap(
    address owner,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
)
    external
    payable
    virtual
{
    if (amount == 0) revert MintFailed(owner, amount);

    WETH wTEL = WETH(payable(address(baseERC20)));
    wTEL.permit(owner, address(this), amount, deadline, v, r, s);

    bool success = wTEL.transferFrom(owner, address(this), amount);
    if (!success) revert PermitWrapFailed(owner, amount);

    _mint(owner, amount);
    emit Wrap(owner, amount);
}
```

An attacker who observes a `permitWrap()` transaction in the mempool could front-run it by executing the same `permit()` operation directly against the `wTEL` contract using the same parameters. This would cause the `permitWrap()` function to fail, as EIP-2612 permit signatures can only be used once. This allows a denial-of-service attack against users trying to utilize the `permit` functionality for wrapping.

Recommendation: Consider one of these approaches to address the vulnerability:

1. Implement a try/catch pattern around the `permit` call to handle potential failures gracefully:

```

function permitWrap(
    address owner,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
)
external
payable
virtual
{
    if (amount == 0) revert MintFailed(owner, amount);

    WETH wTEL = WETH(payable(address(baseERC20)));

    // Try the permit but don't revert if it fails (might have been front-run)
    try wTEL.permit(owner, address(this), amount, deadline, v, r, s) {
        // Permit succeeded
    } catch {
        // Permit failed - could have been front-run, or already approved
    }

    bool success = wTEL.transferFrom(owner, address(this), amount);
    if (!success) revert PermitWrapFailed(owner, amount);

    _mint(owner, amount);
    emit Wrap(owner, amount);
}

```

This approach has a trade-off: it could potentially allow dangling allowances to be used even if not intended by the user.

2. Restrict the function to only be callable by the token owner themselves:

```

function permitWrap(
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
)
external
payable
virtual
{
    address owner = msg.sender;
    if (amount == 0) revert MintFailed(owner, amount);

    WETH wTEL = WETH(payable(address(baseERC20)));
    wTEL.permit(owner, address(this), amount, deadline, v, r, s);

    bool success = wTEL.transferFrom(owner, address(this), amount);
    if (!success) revert PermitWrapFailed(owner, amount);

    _mint(owner, amount);
    emit Wrap(owner, amount);
}

```

This would prevent third-party permit submissions and ensure that only the actual token owner can execute this flow, reducing the risk of front-running.

3. A more secure but complex approach would be to implement nonces managed by the InterchainTEL contract itself rather than relying solely on the EIP-2612 permit mechanism.

Telcoin: Acknowledged: The potential for griefing via frontrunning is always present when EIP-2612 is invoked within a larger code execution context which does not limit the caller strictly to the token owner. As an example, Uniswap has provided functions addressing exactly this nuance.

As noted by this finding, in iTEL's case a frontrunner can cause a single Denial of Service revert by:

- Monitoring the mempool for calls to `permitWrap()`.

- Running an MEV frontrunner which extracts a user's ERC-2612 signature from their transaction *after it submitted to the mempool but before it is included in a block*.
- Rapidly submits a frontrun transaction providing the ERC-2612 permit directly to the wTEL contract, *before the settlement of the above transaction*.
- The original user call to `permitWrap()` reverts because the user's permit has been consumed.

The result is that the frontrunner pays for and performs the user's desired ERC-2612 permit approval on their behalf. There is no recurring DoS opportunity since the signature can only be used once, and the user now only needs to complete the second half of the iTEL wrap action by calling `wrap()`.

In short, this griefing action is economically unattractive because the frontrunner must pay gas and the end result is only a slightly inconvenient UX requiring a second action from the user. Any frontrunner deciding to repeatedly take this action would not be economically motivated but instead emotionally motivated, and eventually will run out of economic/emotional capital. While such a period would be marked by degraded UX, it would be temporary until the frontrunner runs out of resources or decides to allocate them more productively elsewhere.

Options considered but deemed undesirable:

- Implementing a try/catch pattern around the permit call to handle potential failures gracefully introduces potential for abuse of dangling approvals. Any caller would be able to wrap on behalf of users who have outstanding approvals to the iTEL contract. This is undesirable because it could happen without the user's knowledge and consent.
- Restricting the function to only be callable by the token owner themselves resolves the potential for frontrunning but also eliminates one of the main use cases for ERC2612 permits, which is that user-signed approvals can be executed by a third party on the user's behalf.
- To the best of my knowledge, implementing nonces managed by the InterchainTEL contract itself rather than relying solely on the EIP-2612 permit mechanism would not prevent frontrunning. This is the case so long as use of EIP-2612 permits by 3rd-parties on behalf of the user is supported and the mempool is public. Regardless of the iTEL's nonce management implementation, use of wTEL's EIP-2612 permit mechanism must be invoked for the user at some point during execution. This means the user's ERC2612 signature must be fed into the `permitWrap()` function call and thus would always be extractable & usable directly on the wTEL contract by a frontrunner.

Cantina Managed:

To the best of my knowledge, implementing nonces managed by the InterchainTEL contract itself rather than relying solely on the EIP-2612 permit mechanism would not prevent frontrunning...

You're right.

Restricting the function to only be callable by the token owner themselves resolves the potential for frontrunning but also eliminates one of the main use cases for ERC2612 permits, which is that user-signed approvals can be executed by a third party on the user's behalf.

I'd imagine that the main use case is a batched transaction which includes permit + wrap. This could be called by the user themselves and I wouldn't expect too many use cases where a user gives explicit permission to a third party to wrap their tokens for them.

There is no recurring DoS opportunity since the signature can only be used once, and the user now only needs to complete the second half of the iTEL wrap action by calling `wrap()`.

This assumes that the user is aware of what is happening. If the user repeatedly calls `permitWrap()`, then multiple transactions could fail. I'd still recommend the owner call restriction. However, this is more of a theoretical attack, since there is zero to minimal incentive. So, acknowledging it is fine.

Acknowledged.

3.4 Gas Optimization

3.4.1 Code quality and optimization recommendations

Severity: Gas Optimization

Context: (No context files were provided by the reviewer)

Description and Recommendations:

1. Remove TODO comments or implement the mentioned functionality:

```
/// @notice Read-only mechanism, not yet live
function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
    _requireOwned(tokenId);
    return _baseURI();
}

function _baseURI() internal view virtual override returns (string memory) {
    return ""; // TEL svg
}
```

Either implement the commented functionality or remove the comment to avoid confusion.

2. Simplify reward calculation to avoid unnecessary operations:

```
- uint232 weight = PRECISION_FACTOR * weights[i] / totalWeight;
- uint232 rewardAmount = (epochIssuance * weight) / PRECISION_FACTOR;
+ uint232 rewardAmount = epochIssuance * weights[i] / totalWeight;
```

The current approach unnecessarily introduces a large factor and then divides it out later.

3. Move totalWeight check outside the loop:

```
+ if (totalWeight == 0) return;
    for (uint256 i; i < rewardInfos.length; ++i) {
- if (totalWeight == 0) break;
    // ...
}
```

This avoids unnecessary loop iterations and improves code clarity.

4. Import constants from library rather than hardcoding:

```
- bytes32 private constant CREATE_DEPLOY_BYTECODE_HASH =
-     0xdb4bab1640a2602c9f66f33765d12be4af115acccf74b24515702961e82a71327;
+ import { CREATE_DEPLOY_BYTECODE_HASH } from
↳   "@axelarnetwork/interchain-token-service/contracts/utils/Create3AddressFixed.sol";
```

Importing constants from their source library improves maintainability and reduces potential for errors.

5. Use cached tokenManager value instead of recalculating:

```
function isMinter(address addr) external view virtual returns (bool) {
- if (addr == tokenManagerAddress()) return true;
+ if (addr == tokenManager) return true;
    return false;
}
```

Reuses the cached immutable value rather than recalculating the address.

6. Optimize array memory usage with assembly:

```

// trim and return final array
ValidatorInfo[] memory validatorsMatched = new ValidatorInfo[](numMatches);
- for (uint256 i; i < numMatches; ++i) {
-     validatorsMatched[i] = untrimmed[i];
- }
-
- return validatorsMatched;
+ // Could use assembly to set array length directly in memory
+ // without allocating new memory.
+ assembly {
+     mstore(untrimmed, numMatches)
+ }
- return untrimmed;

```

This optimization would avoid copying memory unnecessarily.

7. Optimize genesis committee initialization:

```

// first three epochs use initial validators as committee
+ for (uint256 j; j <= 2; ++j) {
+     epochInfo[j].epochDuration = genesisConfig_.epochDuration;
+     futureEpochInfo[j].epochDuration = genesisConfig_.epochDuration;
+ }

for (uint256 i; i < initialValidators_.length; ++i) {
    // ... validator initialization

    // Add validator to each epoch committee
    for (uint256 j; j <= 2; ++j) {
        EpochInfo storage epochZero = epochInfo[j];
        epochZero.committee.push(currentValidator.validatorAddress);
-        epochZero.epochDuration = genesisConfig_.epochDuration;
-        futureEpochInfo[j].committee.push(currentValidator.validatorAddress);
    }
}

```

Avoids redundant setting of the same value multiple times.

8. Correct comment about reward distribution path:

```

- // check rewards are claimable and send via the InterchainTEL contract
+ // check rewards are claimable and send via the Issuance contract
    uint232 rewards = _checkRewards(validatorAddress, validatorVersion);
    stakeInfo[validatorAddress].balance -= rewards;
    Issuance(issuance).distributeStakeReward(recipient, rewards);

```

Fixes an incorrect comment that references InterchainTEL instead of the Issuance contract.

9. Consider blocking `setApprovalForAll` for consistency:

```

function setApprovalForAll(address operator, bool approved) public virtual override {
    revert NotTransferable();
}

```

Although the function can't be effectively used since transfers are blocked, overriding it would maintain consistency with other transfer-related functions.

10. Optimize validator queue processing:

```

function _updateValidatorQueue(address[] calldata futureCommittee, uint32 current) internal {
    ValidatorInfo[] memory pendingActivation = _getValidators(ValidatorStatus.PendingActivation);
    for (uint256 i; i < pendingActivation.length; ++i) {
        uint24 tokenId = _getTokenId(pendingActivation[i].validatorAddress);
        ValidatorInfo storage activateValidator = validators[tokenId];

        _activate(activateValidator);
    }

    ValidatorInfo[] memory pendingExit = _getValidators(ValidatorStatus.PendingExit);
+   uint8 currentEpochPointer = epochPointer;
+   uint8 nextEpochPointer = (currentEpochPointer + 1) % 4;
+   address[] memory currentCommittee = epochInfo[currentEpochPointer].committee;
+   address[] memory nextCommittee = futureEpochInfo[nextEpochPointer].committee;

    for (uint256 i; i < pendingExit.length; ++i) {
        // skip if validator is in current or either future committee
-       uint8 currentEpochPointer = epochPointer;
-       uint8 nextEpochPointer = (currentEpochPointer + 1) % 4;
-       address[] memory currentCommittee = epochInfo[currentEpochPointer].committee;
-       address[] memory nextCommittee = futureEpochInfo[nextEpochPointer].committee;
-       address validatorAddress = pendingExit[i].validatorAddress;
        if (
            _isCommitteeMember(validatorAddress, currentCommittee)
            || _isCommitteeMember(validatorAddress, nextCommittee)
            || _isCommitteeMember(validatorAddress, futureCommittee)
        ) continue;

        // ... rest of the function
    }
}

```

This avoids repeatedly loading large arrays from storage into memory for each validator, which is costly due to memory expansion.

11. Consider overloading `_isCommitteeMember()` to accept calldata arrays:

```

function _isCommitteeMemberCalldata(address validatorAddress, address[] calldata committee) internal
    ← pure returns (bool) {
    for (uint256 i; i < committee.length; ++i) {
        if (committee[i] == validatorAddress) return true;
    }
    return false;
}

```

This would avoid copying the `futureCommittee` array to memory when checking for committee membership.

12. Consider optimizing committee membership checks with data structures:

For frequent committee membership lookups, consider using a more efficient data structure, such as mappings or enumerable sets:

```

// Using a mapping for O(1) lookups
mapping(address => bool) committeeMembership;

// Populate at committee creation
function _populateCommitteeMembership(address[] memory committee) internal {
    for (uint256 i; i < committee.length; ++i) {
        committeeMembership[committee[i]] = true;
    }
}

// Check membership in O(1) time
function _isCommitteeMember(address validatorAddress) internal view returns (bool) {
    return committeeMembership[validatorAddress];
}

```

This approach would be especially beneficial if committees are large or if membership checks are frequent.

Telcoin: Thank you, these are very helpful. Suggestions 1-10 were implemented in commit [c48fe3b](#).

- 11 is acknowledged but declined to favor code minimalism over the optimization benefit, especially considering the protocol uses a maximum committee size of 32 validators.
- 12 is likewise acknowledged and declined given the max committee size enforced by the protocol is 32 validators for network bandwidth and latency reasons. Some further reasoning:

With max committee length being 32, the loops will not grow beyond reasonable bounds. Introducing a hypothetical `committeeMembership` storage mapping to provide the $\mathcal{O}(1)$ benefits also introduces a need to update said mapping during each call to `concludeEpoch()`.

Since committees are stored and rolled over on a per epoch basis using the ring buffers, said mapping update must correctly handle deletion and addition of entries for the previous, current, next, and future epochs since those are the epochs that determine exit eligibility. All this would need to occur in `_updateEpochInfo()` when epochs are rolled over, but the mapping state would be consumed within `_updateValidatorQueue()` during `_isCommitteeMember()` checks, which happens right after `_updateEpochInfo()` within the top-level `concludeEpoch()` execution context.

In the case where those four epochs use entirely different committees, the suggestion's tradeoff involves iterating over all the committee arrays anyway, and can require up to 128 storage slot updates within the same system call. This makes the tradeoff benefit of $\mathcal{O}(1)$ complexity much less clear cut. In many imaginable scenarios, the current approach of iterating through the committee arrays multiple times for comparison against the validator pending exit involves equal or less complexity.

In cases where there are many validators pending in the exit queue and committees are large, the tradeoff would be worthwhile. However we believe the existing implementation is acceptable since it doesn't require the complex logic described above or introduce potential for extensive state updates which would only be consumed once (later on during the same system call).

Cantina Managed: Fixed as recommended. @markus-telcoin The optimization re. 6 is partially negated when the line `ValidatorInfo[] memory validatorsMatched = new ValidatorInfo[](numMatches);` is included, as this already allocates and expands memory regardless of whether it is written to. I have updated the description and recommendation. You could also simply declare it via `ValidatorInfo[] memory validatorsMatched` without the memory allocation.

Telcoin: Thank you. For completeness I went ahead and added the suggested improvement for efficient memory handling in commit [e4b77d9](#).

3.5 Informational

3.5.1 Missing event emission for validator slashing

Severity: Informational

Context: [ConsensusRegistry.sol#L89-L106](#)

Description: The `applySlashes()` function in the `ConsensusRegistry` contract applies penalties to validators but does not emit any events to record these actions. This makes it difficult to track slashing occurrences off-chain.

While the function changes validator state by reducing their balance or ejecting them entirely, it fails to emit events that would allow external systems to monitor these critical security actions. This contrasts with other state-changing operations in the contract, such as validator activation, exit, and rewards claiming, all of which emit appropriate events.

Recommendation: Add event emission for slashing actions in the `applySlashes()` function:

```

function applySlashes(Slash[] calldata slashes) external override onlySystemCall {
    for (uint256 i; i < slashes.length; ++i) {
        Slash calldata slash = slashes[i];
        // signed consensus header means validator is whitelisted, staked, & active
        uint24 tokenId = _gettokenId(slash.validatorAddress);
        // unless validator was forcibly retired & unstaked via burn: skip
        if (tokenId == UNSTAKED) continue;

        StakeInfo storage info = stakeInfo[slash.validatorAddress];
        if (info.balance > slash.amount) {
            info.balance -= slash.amount;
+           emit ValidatorSlashed(slash.validatorAddress, slash.amount, false);
        } else {
            // eject validators whose balance would reach 0
            _consensusBurn(tokenId, slash.validatorAddress);
+           emit ValidatorSlashed(slash.validatorAddress, slash.amount, true);
        }
    }
}

```

Telcoin: Added in commit [0f39021](#).

Cantina Managed: Fixed as recommended.

3.5.2 Incorrect balance check in Issuance's distributeStakeReward()

Severity: Informational

Context: [Issuance.sol#L29-L40](#)

Description: In the Issuance contract, the `distributeStakeReward()` function contains an incorrect balance check that only verifies if the contract has enough balance for the reward amount, but doesn't account for any additional value sent with the function call.

```

function distributeStakeReward(address recipient, uint256 rewardAmount) external payable virtual
→ onlyStakeManager {
    uint256 bal = address(this).balance;
    if (bal < rewardAmount) {
        revert InsufficientBalance(bal, rewardAmount);
    }

    uint256 totalAmount = rewardAmount + msg.value;
    (bool res,) = recipient.call{ value: totalAmount }("");
    if (!res) revert RewardDistributionFailure(recipient);
}

```

The issue is in the balance check. The function attempts to send `totalAmount` (which is `rewardAmount + msg.value`), but only checks if the contract's balance is greater than `rewardAmount`. This means that if:

- Contract balance is slightly larger than `rewardAmount`.
- `msg.value` is greater than zero.
- `rewardAmount + msg.value` exceeds the contract's balance.

Then the function will revert during the transfer attempt rather than providing a clear error message through the `InsufficientBalance` check.

Recommendation: Update the balance check to account for the total amount being sent:

```

function distributeStakeReward(address recipient, uint256 rewardAmount) external payable virtual
{
    → onlyStakeManager {
        uint256 bal = address(this).balance;
    -     if (bal < rewardAmount) {
    -         revert InsufficientBalance(bal, rewardAmount);
    -     }

        uint256 totalAmount = rewardAmount + msg.value;
    +     if (bal < totalAmount) {
    +         revert InsufficientBalance(bal, totalAmount);
    +     }

        (bool res,) = recipient.call{ value: totalAmount }("");
        if (!res) revert RewardDistributionFailure(recipient);
    }
}

```

Telcoin: Implemented in commit 93d442c.

Cantina Managed: Fixed as recommended.

3.5.3 `_updateEpochInfo` stores end block number of previous epoch instead of start block of the new epoch

Severity: Informational

Context: ConsensusRegistry.sol#L508

Description: At the end of each epoch `_updateEpochInfo` is called to update certain parameters of the current and next epoch. The `EpochInfo` is supposed to indicate the start of the epoch but the as this is called by `concludeEpoch` the `block.number` is actually the last block of the previous epoch.

Recommendation: Modify the `_updateEpochInfo` to use the `block.number + 1`:

```

- epochInfo[newEpochPointer] = EpochInfo(currentCommittee, uint64(block.number), newDuration);
+ epochInfo[newEpochPointer] = EpochInfo(currentCommittee, uint64(block.number)+1, newDuration);

```

Telcoin: Updated in commit 30a7e4a.

Cantina Managed: Fixed.

3.5.4 Incorrect iTEL mint amount if baseERC20 charges fees

Severity: Informational

Context: InterchainTEL.sol#L91-L94

Description: When calling `doubleWrap` the same number of iTEL tokens as native tokens deposited are minted. However if the underlying baseERC20 would charge a fee or have an exchange rate different from 1:1 the amount of iTEL tokens minted would differ from the amount of WTEL tokens received.

This is highly unlikely but as the `baseERC20` is a parameter of the constructor the used implementation is not guaranteed to be that of the current WTEL/WETH which does return the same amount.

Recommendation: Consider minting the exact amount of WTEL received by checking the balance before and after the WTEL deposit.

Telcoin: Acknowledged: This is true but only relevant if the WTEL implementation were to be changed to incorporate a fee at the contract level, which is not on the table because:

- TEL already has fees associated with it at the native gas layer, so fees at WTEL's ERC20 level is redundant.
- Adding fees to WTEL would break assumptions about the InterchainTEL's ITS integration.
- Adding fees to WTEL would break many existing applications built at the contract level like DeFi and NFT platforms (they are designed around WETH which does not use a fee). These applications would not be portable to TN as a result.

Cantina Managed: Acknowledged.

3.5.5 NewEpoch event mixes information about different epochs

Severity: Informational

Context: ConsensusRegistry.sol#L51

Description: When concluding an epoch the `NewEpoch` event is emitted indicating the `newCommittee` block-number +1 as the start block and duration. However `newCommittee` relates to `newEpoch + 2`, the start block number relates to `newEpoch` and the duration can still change even before the end of `newEpoch`.

Recommendation: Although the event is only intended for the intent of making data for epoch updates more readily available for validator dashboards consider clarifying that all three topics are not related to the same epoch and can still change before they go into effect.

Telcoin: Agreed that this behavior is confusing and unintuitive, commit [c2eae6e](#) uses nomenclature consistent with the rest of the contract for "new", "next", and "future" committees/epochs.

The committee passed into `concludeEpoch()` by the protocol refers to the committee for `newEpoch + 2` and is now called "futureCommittee", whereas the `newEpoch`'s committee is now pulled from `_updateEpochInfo()` so it can be emitted within the `NewEpoch` event. This way all parameters within `NewEpoch` refer to the same epoch.

Cantina Managed: Fixed.

3.5.6 Consider separating the pauser role

Severity: Informational

Context: ConsensusRegistry.sol#L628

Description: Pausing is currently only allowed by the contract owner. It is best practice to have a separate role for pausing and unpausing for separation of duty and so that the pauser role can be easily automated.

Recommendation: Consider using a separate pauser role or address.

Telcoin: Acknowledged. Pausability could be separated from the owner into an automatable and slightly more lightweight role, but the team ultimately concluded that adding another permissioned entity for such a critical consensus component was deemed undesirable. The reasoning is that in the worst case scenario, ie an attacker manages to compromise the registry's multisig owner, the attacker's privilege escalation is relatively limited and resolution requires a hard fork which would not be resolved by a separate pauser anyway.

Cantina Managed: Acknowledged.

3.5.7 Ineffective replay protection in validator delegation

Severity: Informational

Context: ConsensusRegistry.sol#L210-L225

Description: The `ConsensusRegistry` contract contains a flaw in its nonce handling for validator delegations, which renders the intended replay protection ineffective. In the `delegateStake()` function, a nonce is used as part of the signature verification process to prevent replay attacks. However, due to a logic error, the incremented nonce value is not properly stored, effectively nullifying this protection.

```

function delegateStake(
    bytes calldata blsPubkey,
    address validatorAddress,
    bytes calldata validatorSig
)
external
payable
override
whenNotPaused
{
    // ... other validation code ...

    uint64 nonce = delegations[validatorAddress].nonce++;
    bytes32 blsPubkeyHash = keccak256(blsPubkey);

    // governance may utilize white-glove onboarding or offchain agreements
    if (msg.sender != owner()) {
        bytes32 structHash =
            keccak256(abi.encode(DELEGATION_TYPEHASH, blsPubkeyHash, msg.sender, tokenId, validatorVersion,
                → nonce));
        bytes32 digest = _hashTypedData(structHash);
        if (!SignatureCheckerLib.isValidSignatureNowCalldata(validatorAddress, digest, validatorSig)) {
            revert NotValidator(validatorAddress);
        }
    }

    delegations[validatorAddress] = Delegation(blsPubkeyHash, msg.sender, tokenId, validatorVersion, nonce);
    // ... rest of function ...
}

```

The issue is that:

1. The current nonce is read and incremented (`nonce++`).
2. This incremented value is stored in the delegations mapping.
3. However, when creating the new Delegation struct, the function uses the original, non-incremented nonce value.
4. This effectively means that the nonce never increases, making signature replay possible.

Note that the actual impact is limited because validators can only have one NFT minted to them, and each delegation is tied to a specific validator address, BLS public key, and token ID.

Recommendation: Fix the nonce handling to ensure the incremented value is properly stored:

```

function delegateStake(
    bytes calldata blsPubkey,
    address validatorAddress,
    bytes calldata validatorSig
)
external
payable
override
whenNotPaused
{
    // ... other validation code ...

-    uint64 nonce = delegations[validatorAddress].nonce++;
+    uint64 nonce = delegations[validatorAddress].nonce;
    bytes32 blsPubkeyHash = keccak256(blsPubkey);

    // governance may utilize white-glove onboarding or offchain agreements
    if (msg.sender != owner()) {
        bytes32 structHash =
            keccak256(abi.encode(DELEGATION_TYPEHASH, blsPubkeyHash, msg.sender, tokenId, validatorVersion,
→         nonce));
        bytes32 digest = _hashTypedData(structHash);
        if (!SignatureCheckerLib.isValidSignatureNowCalldata(validatorAddress, digest, validatorSig)) {
            revert NotValidator(validatorAddress);
        }
    }

-    delegations[validatorAddress] = Delegation(blsPubkeyHash, msg.sender, tokenId, validatorVersion, nonce);
+    delegations[validatorAddress] = Delegation(blsPubkeyHash, msg.sender, tokenId, validatorVersion, nonce +
→ 1);
    // ... rest of function ...
}

```

Alternatively:

```

- uint64 nonce = delegations[validatorAddress].nonce++;
+ uint64 nonce = ++delegations[validatorAddress].nonce;

```

Telcoin: Good eye, it is true that the incremented nonce is being overwritten with the unincremented value cached in memory and should be amended for correctness. However, signature replay is not possible because a successful delegation transaction results in a state update of the validator's status from `Undefined` => `Staked` within `_recordStaked()`. Thus subsequent signature replay attempts will be blocked when reaching `_checkValidatorStatus(Undefined)` because its state is now `Staked`, effectively serving as stateful replay protection.

Seeing as signature replay cannot be achieved even though the nonce logic is incorrect, we believe this finding should be downgraded from low to informational. The nonce logic is fixed in commit [9f456c6](#).

Cantina Managed: Fixed as recommended. Changed severity from Low to Informational, since this was not exploitable at the time.

3.5.8 Delegation risk and reward asymmetry in consensus system

Severity: Informational

Context: `ConsensusRegistry.sol#L191-L222`

Description: In the current delegation model implemented in `ConsensusRegistry`, there's a significant imbalance in the risk/reward structure between validators and delegators:

1. Delegators provide the entire stake amount required for validation.
2. Delegators bear the full financial risk of slashing if the validator misbehaves.
3. Delegators receive 100% of the protocol rewards.
4. However, delegators have no control over validator behavior or operational security.

While the intended behavior is for all staking rewards to flow to the stake originator (the delegator), this creates an unusual economic arrangement where validators perform ongoing node operation duties with

no direct protocol rewards, and delegators receive all rewards despite not actively participating in validation.

This structure creates two opposing asymmetries:

- Risk Asymmetry: Delegators bear full financial risk of slashing for validator misbehavior without control over validator operations.
- Reward Asymmetry: Validators receive no direct protocol rewards for their ongoing operational work and infrastructure costs, while delegators receive 100% of rewards without operational responsibilities.

This dual asymmetry could lead to several issues:

- Validators might seek off-chain compensation from delegators, creating unregulated arrangements.
- Validators have reduced incentives for proper operation since they don't receive a share of the protocol rewards.
- The unstable economic arrangement may result in fewer delegations or validators seeking ways to circumvent the intended flow of rewards.

Recommendation: Consider implementing a more balanced risk/reward structure that aligns incentives between validators and delegators. This could include mechanisms for sharing both risks and rewards in proportion to responsibilities, or establishing a formal fee structure that provides validators with economic incentives while maintaining appropriate risk exposure. At minimum, clearly document these asymmetries to ensure both validators and delegators fully understand the economic model they're participating in.

Telcoin: Acknowledged, with some context: It is true that for delegation mechanics in this audit's scope, the dynamic of economic incentives between delegators and validators do not overlap. This is however intentional and not the full picture. The reasoning is twofold:

1. The protocol enforces only the simplest and most agnostic delegation invariant possible: delegators put money in (stake), so they receive money out (stake and rewards).

As noted by this finding, the above means the protocol's base layer for delegation separates technical duties from capital flows by sending all capital solely to the delegator address. The intent is to offload the economic structure for risk and reward entirely to the delegator address, in order to enable modular reward schemas in the form of delegator smart contracts whose implementations are customizable. The concept is similar on a high level to Rocketpool minipools: delegator contracts implement revenue share schemas. The registry specifically uses Solady's `SignatureCheckerLib::isValidSignatureNowCalldata()` which includes EIP-1271 compliance to open the door for reward schema contracts located at the delegator address.

2. Telcoin-Network validators are highly trusted and must perform an offchain signature handshake with delegators (explicitly accept each delegation/delegator via signature).

This means delegator contract reward schemas can be specific to the validator & delegator because the delegation is agreed upon and signed by both validator and delegator. In this sense they are comparable to a contractual agreement between both parties.

Given the two points above, the use of delegator contracts rejoins the sandboxed incentives for delegators and validators, and aligns their interests so that:

- Delegators would not receive 100% of rewards; depending on delegator contract implementation, some portion would be allocated to validators.
- Delegators will not bear full financial risk for validators being slashed; when using a delegator contract, slashes applied to the contract's full balance effectively slash both delegator and validator balances.

Cantina Managed: Acknowledged.

3.5.9 Potential duplicate validators in committee without validation

Severity: Informational

Context: ConsensusRegistry.sol#L41-L52

Description: The `concludeEpoch()` function in the `ConsensusRegistry` contract accepts a new committee array without validating it for duplicate validator addresses:

```
function concludeEpoch(address[] calldata newCommittee) external override onlySystemCall {
    // update epoch ring buffer info, validator queue
    (uint32 newEpoch, uint32 duration) = _updateEpochInfo(newCommittee);
    _updateValidatorQueue(newCommittee, newEpoch);

    // assert new epoch committee is valid against total now eligible
    ValidatorInfo[] memory newActive = _getValidators(ValidatorStatus.Active);
    _checkCommitteeSize(newActive.length, newCommittee.length);

    emit NewEpoch(EpochInfo(newCommittee, uint64(block.number + 1), duration));
}
```

This function is called by the system at epoch boundaries and assigns voting rights to validators in the committee. However, the function does not:

1. Check for duplicate validator addresses in the committee array.
2. Enforce any sorting or ordering requirement.

This could potentially lead to:

- A validator appearing multiple times in the committee, effectively giving them multiple votes.
- Issues with consensus logic that might assume unique committee members.

Recommendation: Add validation to ensure the committee contains only unique validator addresses. This could be done by requiring the committee array to be sorted, making it easy to check for duplicates.

Telcoin: Implemented in [92e41c5](#).

Cantina Managed: Fixed as recommended. `concludeEpoch()` now validates that the new committee is in descending order enforcing uniqueness. If the new committee is of size 0, it will fail inside `_enforceSorting()` due to an underflow instead of failing in `_checkCommitteeSize()`. For better error messages, you could:

1. Run `_checkCommitteeSize()`.
2. Check `_enforceSorting()`.
3. Call `_updateValidatorQueue()`. This change would then first perform the checks and then create the effects.