
# Frank castle

## Lido Audit

December 2025

# Lido_Report

## Content



## About Frank Castle 🦀

Frank Castle is a profissional smart contract security researcher with a focused expertise in auditing Rust-based contracts and decentralized infrastructure across leading blockchain ecosystems, including Solana , Polkadot , and Cosmos (CosmWasm).🦀

Frank Castle has audited Lido, GMX ,Pump.fun, LayerZero, Synthetix , Hydration ,DUB Social and several multi-million protocols.

with more than ~25 Rust Audit , ~15 Solana Audits , and +100 criticals/highs found , All the reports can be found here

For private audit or consulting requests please reach out to me via Telegram @castle_chain , **Twitter** (@0xfrank_auditor) or **Discord** (castle_chain).

## Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

## Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

## Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-01] | Underconstraint on reference_slot Allows Arbitrary Future Slot Proofs | High | Resolved |

## Findings

## 1. High Findings

## [H-01] Underconstraint on `reference_slot` Allows Arbitrary Future Slot Proofs

### Severity

**Impact:** High

**Likelihood:** High

### Description

The contract and circuit logic imposes only minimal constraints on the `reference_slot` parameter when submitting a new report via `submitReportData()`. Specifically, if `reference_slot` differs from `bc_slot`, the logic merely checks that the `reference_slot` itself does not have a block and that all slots between it and `bc_slot` do not have blocks. Beyond that, there is no upper bound or stricter limit on `reference_slot`.

1. **Exploitability for Future Slots**
   An attacker can set `bc_slot` to the last valid **past** slot that is known to have a block. Then, for an arbitrarily large **future** `reference_slot`, the contract's `_verify_reference_and_bc_slot()` will loop backward, decrementing `reference_slot` until it reaches `bc_slot`. Because future slots do not exist yet (and thus the beacon roots precompile will revert internally for those timestamps),

`_blockExists()` will always return `false` for those slots. Hence, these future slots satisfy the condition "if `reference_slot != bc_slot`, then the reference slot must not have a block."

- This attack can be performed repeatedly, letting the attacker claim "valid" data for any future `reference_slot`, as long as they keep `bc_slot` pointed to the same past slot that actually contains a block.
- The main practical constraint is how large `reference_slot` can be before the gas limit is reached (due to the backward iteration in `_verify_reference_and_bc_slot()`).

2. **Behavior of `_blockExists()` With Future Slots**
   - The function `_blockExists(slot)` calls `_getBeaconBlockHashForTimestamp(_slotToTimestamp(slot))`.
   - For a future slot (one that has not actually occurred yet), the beacon roots precompile provides no data and reverts internally. Consequently, `_getBeaconBlockHashForTimestamp()` returns `(false, 0x0)`, so `_blockExists(slot)` evaluates to `false && (0 != 0) → false`.
   - This makes every future slot effectively "empty," letting `_verify_reference_and_bc_slot()` pass for any future `reference_slot`.
3. **Inability to Override Attackers' Reports**
   - After an attacker successfully sets a report for a particular `reference_slot`, the function checks `report_at_slot.reference_slot == 0` before accepting any new report for that same slot.
   - Consequently, once the attacker has submitted a "valid" proof referencing a large future slot, **no one else** can override or update that slot. Any subsequent attempts to submit a new report for the same `reference_slot` will fail with "Report was already accepted."
4. **Impact of Projecting a Single bc_slot onto All Future Slots**
   - By using the same `bc_slot` (from a valid past block) and inflating `reference_slot` to any future value, an attacker can project the **same** (outdated) state onto multiple future slots.
   - Because this oracle is used as a "second opinion," valid future reports from the main oracle might be reverted, leading to a **Denial-of-Service (DoS)** scenario for legitimate updates that require a second opinion. In other words, the outdated report from the attacker could cause valid new reports to fail verification if they conflict with the outdated report.
5. **Exploitability for Previous Slots That Do Not Have Blocks**
   - Similarly, an attacker can target **older** `reference_slot` values that never had blocks. By using the same `bc_slot` from a past slot with a block, they can set a report for an old reference slot that is recorded as "empty" and thus pass the check since the loop won't execute.
   - While outdated slots are not frequently used, and thus this scenario may be less critical than future-slot exploitation, it remains a potential avenue for abuse or confusion if older data is ever referenced or needed.

## Recommendations

Impose an upper bound on `reference_slot` to ensure it cannot reference slots beyond the current on-chain time (or a suitably safe margin). It is also recommended to add a check that require `reference_slot` greater than `bc_slot`.