# Proof of Capital

**Smart Contract Security Audit**

Date: February 10, 2025

Version: 1.4

# Contents

# 1 Executive Summary

The security review of the Proof of Capital protocol revealed a generally sound implementation with no critical vulnerabilities that could lead to immediate financial losses. However, several areas require attention before deployment:

- Complete absence of unit tests
- Insufficient inline documentation, user guides and architectural documentation
- Missing deployment procedures and configuration documentation

The identified issues primarily fall into the categories of code quality, documentation gaps, and preventive security measures. While some findings are marked as "High" or "Medium", these mainly represent opportunities for improvement in error handling and input validation rather than immediate security threats.

| **Category** | **Count** | **Acknowledged** | **Fixed** |
|---|---|---|---|
| Critical | 0 | 0 | 0 |
| High | 1 | 0 | 1 |
| Medium | 3 | 0 | 3 |
| Low | 4 | 1 | 3 |
| Informational | 9 | 2 | 7 |

Table 1.1: Summary of Findings by Category

# 2 Overview

## 2.1 Project Details

Proof of Capital is a stepped liquidity pool with lockup mechanics for the TON blockchain. The source code is written in Tact 1.5 and is not publically available at the date of audit.

## 2.2 Methodology

**Audit Duration:** 2025-01-09 – 2025-01-22

**Audit Methodology:** Architecture Review, Unit Testing, Functional Testing, Static Program Analysis, Manual Review.

**Errors checked:**

- Accounting errors
- Inconsistent state updates
- Atomicity violations
- Access control
- Centralization of power
- Number rounding errors
- Incorrect standard token implementation
- Timestamp dependence
- Transaction-ordering dependence

**Auditor:** Georgiy Komarov

**Audited version:** `915fe243863f9332895e3ed3b619ae82294b226e`

**Fixed version:** `472828696892a8825b7c1af68c199ee98b954b9a`

## 2.3 Scope of the Audit

The following components were included in this security review:

- Smart contract source code (Tact)
- Deployment scripts and configuration

The following aspects were explicitly excluded from this audit:

- Frontend implementation
- Economic viability and tokenomics beyond basic contract design
- Off-chain infrastructure and external integrations

## 2.4 Audit Deliverables

This security audit includes the following deliverables:

- This audit report detailing all findings and recommendations
- Architectural assessment and suggestions
- An example implementaion of unit tests contributed to the repository
- The suggested methodology for testing

# 3 Findings

## 3.1 Severity Levels

Smart contract vulnerabilities are categorized by severity using the following criteria:

**Critical:** Immediate, direct threat to contract funds or control. Trivially exploitable vulnerabilities that allow theft, permanent locking of funds, or complete contract takeover through arbitrary code execution. Must be fixed before deployment.

**High:** Significant vulnerabilities affecting contract's core functionality or economic model. Includes practical attack vectors that could lead to fund manipulation, broken core logic, or severe economic exploits. Should block deployment until resolved.

**Medium:** Notable technical or economic weaknesses requiring specific conditions to exploit. Covers state corruption, balance accounting errors, and economic edge cases that could disadvantage users but don't directly threaten funds. Should be fixed but doesn't block deployment.

**Low:** Minor implementation issues with limited practical impact. Includes state inconsistencies, gas inefficiencies, and edge case failures that don't fundamentally threaten contract operation. Can be addressed in future updates.

**Informational:** Code quality issues and best practice violations without security impact. Covers gas optimizations, missing documentation, poor naming conventions, and other maintenance concerns. Fix at convenience.

## 3.2 High

### H1: Unfair token acceptance in `JettonTransferNotification`

**Status:** Fixed

While this behavior was lately documented in the protocol specifications ("Features of Selling Jetton Remainders"), the current implementation of `JettonTransferNotification` presents a minor economic risk that requires attention. The contract accepts excess jettons beyond its buyback capacity, resulting in users receiving fewer TON than expected for their tokens.

```
if (tokenForBuyback >= tokensAvailableForBuyback) {
    self.contractJettonBalance = self.contractJettonBalance + (
   tokenForBuyback - tokensAvailableForBuyback);
    tokenForBuyback = tokensAvailableForBuyback;
}
```

This implementation silently accepts excess tokens while only paying for the maximum available amount (`tokensAvailableForBuyback`). This creates an unfair situation where users receive less compensation than the true value of their tokens.

Impact

Despite being a documented feature, this implementation could lead to:

- Users losing value due to incomplete understanding of the mechanism
- Accumulated excess jettons in the contract with no clear redemption path
- Potential manipulation of the contract's economic parameters

Recommendation

While users are advised to "study the output of the jetton_available getter function before sending jettons," we recommend implementing additional safeguards:

1. Add a check to reject transactions that exceed the available buyback capacity
2. Implement a revert mechanism for excess tokens
3. Consider adding explicit warning messages in transaction responses

## 3.3 Medium

### M1: Lack of bounds in `lockEndTime`

**Status: Fixed**

The `lockEndTime` field determines the locking period and is set in `receive(msg: ExtendLock)`. It should have a reasonable upper limit. Otherwise, a malfunction in the scripts or frontend components that set this field could result in indefinite token locking.

**Recommendation**: Add limit to `receive(msg: ExtendLock)`:

```
let newLockEndTime = self.lockEndTime + msg.additionalTime;
require(newLockEndTime - now() < TWO_YEARS, "Lock end time cannot exceed
   two years");
self.lockEndTime = newLockEndTime;
```

## M2: Incorrect value validation in `receiver()`

**Status:** Fixed

In `receiver()`, there's a flawed value check:

```
let value = context().value - self.commission * 2;
// ...
// require(value > self.commission, "Amount must be greater than 0.15");
```

This validation is logically incorrect because `value` is already defined as `context().value - self.commission * 2`. Comparing this reduced value against `self.commission` makes no sense, as we've already subtracted twice the commission amount. What we actually need is a check that the resulting `value` is non-negative to prevent underflow conditions.

**Recommendation**: Replace with a proper non-negative check:

```
require(value >= 0, "Insufficient funds to cover commission");
```

## M3: Silent failure in token redemption logic

**Status:** Fixed

In `receive(msg:  JettonTransferNotification)`, there's a dangerous silent failure when handling insufficient contract balance:

```
// Dangerous silent failure:
if (self.contractTonBalance - tonToPay) < 0 {
    self.contractTonBalance = 0;
} else {
    self.contractTonBalance = self.contractTonBalance - tonToPay;
}
```

This implementation silently fails when the contract has insufficient TON balance to pay for token redemption. Instead of reverting the transaction or notifying the user, it simply sets the balance to zero and continues execution. This creates a situation where users may not receive full compensation for their tokens.

**Impact**:

- No error notification to users about failed redemptions
- Creates dangerous discrepancy between expected and actual contract balance
- Return wallet owner loses visibility into failed operations

**Recommendation**: Replace the silent failure with an explicit requirement check:

```
require(self.contractTonBalance >= tonToPay,
        "Insufficient contract balance for redemption");
self.contractTonBalance = self.contractTonBalance - tonToPay;
```

**Note**: While this could be bypassed by the return wallet or user address requesting fewer tokens, the owner's ability to withdraw tokens provides a safety mechanism. However, explicit failure is still preferable to silent failure for proper error handling and user experience.

## 3.4 Low

### L1: Insufficient input address validation

**Status: Fixed**

There are some addresses that are hardcoded to zero in the contract. If initialization is incorrect (due to frontend/script bugs or improper usage), tokens may become permanently locked.

**Recommendations**:

- Replace hardcoded `newAddress(0, 0)` in initialization of `self.contractJettonWalletAddress` with `owner`. If for some reasons its value was not set in `receive(msg: TakeWalletAddress)`, `transferTokensTo` will send tokens to the controlled address, thus they won't be lost.
- Implement input validation in `receive(msg: TakeWalletAddress)` to ensure that the new address is not zero.
- Implement the same validation and initialization scheme for `additionalJettonWalletAddress` and `additionalJettonMasterAddress`.

### L2: Inconsistent commission multiplier in `WithdrawUnusedTons`

**Status: Fixed**

The contract uses different commission multipliers when calculating unused TON balance. While `unused_tons()` getter deducts a single commission:

```
get fun unused_tons(): Int {
    return myBalance() - self.contractTonBalance - self.commission;
}
```

The `WithdrawUnusedTons` message handler uses an arbitrary 10x multiplier:

```
let unusedTonToWithdraw: Int = myBalance() -
    self.contractTonBalance - (self.commission * 10);
```

This inconsistency may cause confusion for integrators and slightly different behavior between viewing and withdrawing unused tons.

**Recommendation**: Standardize the commission multiplier across all balance calculations:

```
const COMMISSION_MULTIPLIER: Int = 10;
let unusedTonToWithdraw: Int = myBalance() -
    self.contractTonBalance - (self.commission * COMMISSION_MULTIPLIER);
```

**Impact**:

- Minor inconsistency between balance reporting and withdrawal
- Impossible to withdraw funds due to incorrect calculations in corner cases
- No risk of fund loss as calculations are conservative
- Potential confusion for contract integrators

### L3: Misleading variable naming for step reduction multiplier

**Status: Fixed**

There is the following pattern present in `calculateTonForTokenAmountEarned` and `calculateJettonsToGiveForTonAmount`:

```
if (localCurrentStep > self.trendChangeStep) {
  jettonsPerLevel = (jettonsPerLevel *
    (1000 - self.levelIncreaseMultiplierafterTrend)) / 1000;
} else {
  jettonsPerLevel = (jettonsPerLevel *
    (1000 + self.levelIncreaseMultiplier)) / 1000;
}
```

The implementation is mathematically correct - it properly reduces the jettons per level after the trend change. However, the variable name `levelIncreaseMultiplierAfterTrend` is misleading since it's used as a reduction factor, not an increase multiplier.

**Recommendations**:

1. Rename the variable to better reflect its purpose:

   ```
   levelDecreaseMultiplierAfterTrend
   ```

2. Document the reduction behavior in code comments
3. Create unit tests to verify the reduction logic behaves as expected

### L4: Missing emergency pause functionality

**Status: Acknowledged**

The contract lacks pause/unpause functionality which could be crucial for handling critical security events or necessary protocol upgrades. While the contract includes owner privileges for various administrative functions, it cannot be temporarily suspended in case of emergencies.

**Recommendation**: Implement pause mechanism with the following features:

- Add pause/unpause functions accessible only by owner
- Automatically extend lock times by the duration of the pause
- Block all user operations while paused

- Allow read operations to continue functioning
- Emit events for pause/unpause actions

**Impact**:

- Ability to respond to critical security events

## 3.5 Informational

### I1: State modifications before validation

**Status: Fixed**

In `receive(msg: ExtendLock)`, mutable state changes occur before input validation:

```
self.lockEndTime = self.lockEndTime + msg.additionalTime;
```

While this doesn't currently cause issues, it could lead to problems when additional logic is implemented. State modifications should occur after all validation checks.

### I2: Inconsistent use of constants

**Status: Fixed**

Several values should be defined as constants for gas optimization:

- Convert `self.tonBalanceRedemption` to a constant
- Define `self.commission` as a constant:

```
require(value > self.commission, "Amount must be greater than 0.05");
require(value > self.commission, "Amount must be greater than 0.15");
```

### I3: Unused fields

**Status: Acknowledged**

The following fields are only used in getters and not involved in core logic:

- `additionalJettonMasterAddress`
- `additionalJettonWalletAddress`
- `anotherTokenAmount`
- `tonBalanceRedemption`

Consider implementing code generation for these getter-only fields.

### I4: Unnecessary computation in `receiver()`

**Status: Fixed**

In the `receiver()` function, there's an unnecessarily complex calculation:

```
let value = context().value - self.commission * 2;
if (sender() == self.owner){
    self.contractTonBalance = self.contractTonBalance + value + self.
   commission * 2;
}
```

This can be simplified to use `context.value()` directly, avoiding redundant arithmetic operations:

```
let value = context().value - self.commission * 2;
if (sender() == self.owner){
    self.contractTonBalance = self.contractTonBalance + context.value();
}
```

**Impact**: Gas optimization through reduction of unnecessary arithmetic operations.

### I5: Redundant variable initialization

**Status: Acknowledged**

In `calculateJettonsToGiveForTonAmount`, there's a misleading variable initialization:

```
let jettonsPerLevel: Int = self.quantityJettonsPerLevel;
```

This initialization is redundant as the variable is reassigned before its first use. This creates confusion about the actual data flow and wastes gas on an unused operation.

**Recommendation**: Remove the initialization and declare the variable at its first meaningful assignment:

**Impact**: Code clarity improvement and minor gas optimization.

### I6: Magic numbers should be constants

**Status: Fixed**

The contract uses several magic numbers directly in the code. While these numbers are valid, using raw numbers makes the code error-prone and harder to maintain. Tact's constant system evaluates these at compile time with zero runtime cost.

Example of current implementation:

```
if (lockTime > 5_184_000) {
    // ...
}
```

**Recommendation**:

Define constants at the global, contract or trait level:

```
const SIXTY_DAYS: Int = 5_184_000;  // 60 * 24 * 60 * 60
```

Then use these named constants throughout the code:

```
if (lockTime > SIXTY_DAYS) {
    // ...
}
```

**Impact**:

- Improved code readability
- Reduced chance of errors when dealing with time calculations
- Zero gas cost due to compile-time evaluation
- Easier maintenance and updates of time-based parameters

## I7: Inefficient conditional logic in `JettonTransferNotification`

**Status: Fixed**

The `receive(msg: JettonTransferNotification)` function contains redundant code that could be simplified using the standard library's `min()` function:

```
let tonToPay: Int = 0;
if (amount >= tokensAvailableForReturnBuyback) {
    tonToPay = self.calculateTonForTokenAmountEarned(
   tokensAvailableForReturnBuyback);
    self.jettonsEarned = self.jettonsEarned +
   tokensAvailableForReturnBuyback;
}
else {
    tonToPay = self.calculateTonForTokenAmountEarned(amount);
    self.jettonsEarned = self.jettonsEarned + amount;
}
```

**Recommendation**: Use the `min()` function to simplify the logic:

```
let effectiveAmount: Int = min(amount, tokensAvailableForReturnBuyback);
let tonToPay: Int = self.calculateTonForTokenAmountEarned(effectiveAmount);
self.jettonsEarned = self.jettonsEarned + effectiveAmount;
```

**Impact**:

- Reduced code duplication
- Improved maintainability
- Better readability
- Reduced chance of logic errors in future modifications

## I8: Repetitive patterns should be inline functions

**Status: Fixed**

Several code patterns are repeated throughout the contract. While the compiler is likely smart enough to optimize these, using inline functions would improve code clarity and reduce the likelihood of copy-paste errors during development.

Example patterns that should be extracted:

```
// Pattern 1: Profit percentage calculation
if (localCurrentStep > self.trendChangeStep) {
    return self.profitPercentage;
} else {
    return self.profitPercentage * 2;
}

// Pattern 2: Jettons per level calculation
if (localCurrentStep > self.trendChangeStep) {
    return (jettonsPerLevel *
        (1000 - self.levelIncreaseMultiplierafterTrend)) / 1000;
} else {
    return (jettonsPerLevel *
        (1000 + self.levelIncreaseMultiplier)) / 1000;
}
```

**Recommendation**: Extract these and similar patterns into inline functions.

**Impact**:

- Improved code maintainability
- Reduced chance of copy-paste errors
- Better readability and self-documentation
- Zero runtime overhead due to inlining
- Easier testing of isolated business logic

## I9: Centralization risks in the owner role

**Status: Acknowledged**

The contract grants significant privileges to both the owner addresses, creating potential single points of failure. The owner can:

- Extend lock period
- Modify contract's critical parameters

While this centralization is common in similar DeFi protocols, it represents a significant trust assumption that should be explicitly documented.

**Recommendation**:

- Implement multi-signature wallet requirements for privileged operations
- Document owner's priviligies
- Consider time-locks for critical parameter changes
- Implement proper key management security measures

# 4 Recommendations

1. Implement comprehensive unit testing based on the examples provided
2. Implement the CI pipline that includes unit testing and static analysis via Misti
3. Use Blueprint to deploy and interact with contracts, as it is the most commonly adopted tool in the ecosystem
4. Perform rigorous testing when developing the frontend
5. Improve documentation:
    - Provide a clear description of the protocol and its rationale, including:
        - Step system documentation: number of steps, price coefficients per step, token allocation per step
        - Trend change mechanism: trigger conditions and resulting modifications
        - All default parameters from contract initialization (commission, multipliers, thresholds)
        - Include a schematic representation of the stepped liquidity mechanism
    - Add comments to fields and constants
    - Add docstrings to functions that include information about how they modify the contract's mutable state
6. Publish the project source on GitHub and verifier.ton.org
    - This is important for building trust
7. Adopt common security practices:
    - Create a bug bounty program
    - Add an emergency contract to the GitHub repository