

CPRO Locking Audit Report

Document version: **1.0.1**

Auditors:

Srdjan Delić

Tatjana Radovanović

Audit date:

27.10.2025

Contents:

Contents:	1
Overview	2
About CPRO Locking	2
Disclaimer	2
Risk Classification	3
Tools	3
Protocol Summary	3
Description	3
Documentation	3
Actors and Roles	3
Actors	3
Key Components	4
Audit Scope	4
Result Summary	4
Findings	5
Security Controls Review	9
Gas & Performance Observations	10

Overview

Repository / commit hash / tag: _____

Files / folders included in the audit: CPROLocking.sol

Language / compiler version: Solidity ^0.8.28

Testing networks: Remix, Sepolia, ChatGPT

Audit start date: 2025-10-27

Audit end date: 2025-10-27

About CPRO Locking

CPRO (CryptoProcessor) Locking is an [ownable](#) smart contract developed by the CPRO Team for the Ethereum chain. The main purpose of the locking smart contract is to enable users to lock ERC20 tokens (CPROToken in this case) for a specified duration.

Disclaimer

The audit team will implement extensive and exhaustive internal tooling and methodologies into providing a complete token audit process. The audit team **cannot** guarantee that all potential vulnerabilities or risks will be identified and as such holds **no responsibility** if issues are later identified in any further step of the development or deployment process for the token smart contract(s) being audited.

The audit is focused on [Solidity](#) contracts security and quality of code with the consideration for the implementation and adherence of the best, industry-wide practices implemented in the said smart contracts.

In case of future smart contract revisions or edits, all previous audits will be deemed as unverified/deprecated and **invalidated**. For even minor edits to the audited contracts, the same audit methodologies should be applied again in order to achieve the same level of confidence and security.

Risk Classification

Severity chart:

- **Critical** - High impact and high likelihood of happening. Requires immediate attention and fixing
- **High** - Medium impact and high likelihood of happening. Requires attention and fixing
- **Medium** - Medium impacts and medium likelihood of happening. No urgent attention required, but recommended addressing before next deployment/testing cycle
- **Low** - Low impact and low likelihood of happening. Can be postponed to the next development cycle
- **Informative** - Audit team's own suggestions for code, structure and security improvements and potential vulnerabilities that could manifest in the near future, but weren't part of the required token specifications

Tools

[Slither](#) - Hardhat/Foundry projects

[Aderyn](#) - Hardhat/Foundry projects (v. 0.6.4)

AI Tools - [ChatGPT](#), [Claude](#), [RemixAI Assistant](#)

Protocol Summary

Description

Documentation

The CPRO project documentation is under an NDA and is not available for public view.

Actors and Roles

Actors

- **Token_Team**: Token smart contract developers and core team members.
- **Users**: Receivers of the CPRO token via scheduling contracts.

Key Components

CPROLocking.sol is the locking smart contract developed by the CPRO Team and it manages the locking functionality of the ERC20 CPRO tokens.

This contract implements a token locking system where users can lock or unlock their tokens, preventing any withdrawal attempts until a specified time.

Some of the key features of the locking contract:

- Users can lock tokens and specify an unlock time in the future
- Locked tokens cannot be withdrawn before the unlock time
- The owner can lock tokens with the authority of other users
- Locks can be extended to a later time
- Users can batch unlock multiple locks at once
- Only the token owner can claim their specific lock
- Gas limit protection on batch operations (max 50 locks per batch)
- [Reentrancy](#) protection

Audit Scope

- CPROLocking.sol

Result Summary

There are security concerns and some critical issues found during the audit with a handful of high/medium issues around ERC-20 safety and overflow protection in *view* functions.

NO scamming behavior found during the audit.

The smart contract has security features:

- Uses ReentrancyGuard for claimTokens()
- Uses Ownable for access control
- Prevents reentrancy, overflows (Solidity ^0.8.x)

The following findings are a consolidated set of issues and observations from various tools and audit engineers.

Findings

1. **Changing state after an external call can lead to re-entrancy attacks. Use the checks-effects-interactions pattern to avoid this issue**

Severity: **CRITICAL**

Location: CPROLocking.sol (line 87), function *lockTokens*(uint256 amount,uint256 unlockTime, string calldata lockType) external returns (uint256 lockId)

Description: Uses raw transfer/transferFrom, which can fail silently on some tokens (e.g., USDT, BNB)

Recommendation:

Replace with OpenZeppelin's SafeERC20 for all token transfers

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

2. **Changing state after an external call can lead to re-entrancy attacks. Use the checks-effects-interactions pattern to avoid this issue**

Severity: **CRITICAL**

Location: CPROLocking.sol (line 129), function *lockTokensFor*(address user,uint256 amount,uint256 unlockTime,string calldata lockType) external onlyOwner returns (uint256 lockId)

Description: Uses raw transfer/transferFrom, which can fail silently on some tokens (e.g., USDT, BNB)

Recommendation:

Replace with OpenZeppelin's SafeERC20 for all token transfers

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

3. **No Slippage Protection**

Severity: **CRITICAL**

Location: CPROLocking.sol, function *batchUnlock*

Description: If one lock in the batch fails (e.g., already claimed), the entire batch reverts, but gas is still consumed

Recommendation:

Consider a "partial unlock" pattern or require explicit confirmation of revert-on-failure

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

4. Front-Running

Severity: **CRITICAL**

Location: CPROLocking.sol, function *lockTokensFor*

Description: The owner can lock tokens for any user without their consent (e.g., force-locking a victim's funds)

Recommendation:

Add a whitelist or require user approval (e.g., EIP-712 signature)

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

5. No Check for lockType Validity

Severity: **CRITICAL**

Location: CPROLocking.sol, functions *lockTokensFor*, *lockTokens*

Description: Arbitrary strings (e.g., "scam") can be passed as lockType

Recommendation:

Restrict to predefined types (e.g., enum LockType { Team, Liquidity, Marketing })

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

6. Integer Overflow

Severity: **CRITICAL**

Location: CPROLocking.sol

Description: Unbounded loops over userLocks could hit gas limits or overflow unlockableAmount

Recommendation:

Add a loop limit (e.g., require(lockIds.length <= 100, "Too many locks"))

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

7. No Protection Against Reused lockId

Severity: **CRITICAL**

Location: CPROLocking.sol, functions *lockTokensFor*, *lockTokens*

Description: If nextLockId somehow resets (e.g., via malicious selfdestruct + redeploy), old locks could be overwritten

Recommendation:

Use a mapping to track used IDs or make nextLockId immutable

CPRO Team: acknowledged

Audit Team: Waiting for fix implementation

8. Lack of Events for Critical Admin Actions

Severity: CRITICAL

Location: CPROLocking.sol, function *emergencyWithdraw*

Description: *emergencyWithdraw* emits no event, hiding owner activity

Recommendation:

Add an EmergencyWithdraw event

CPRO Team: acknowledged

Audit Team: Waiting for fix implementation

9. Inefficient Storage

Severity: CRITICAL

Location: CPROLocking.sol

Description: *userLocks* mapping stores an unbounded array per user (gas-costly for large users)

Recommendation:

Use a EnumerableSet (OpenZeppelin) or paginate results

CPRO Team: acknowledged

Audit Team: Waiting for fix implementation

10. No Batch Locking

Severity: MEDIUM

Location: CPROLocking.sol

Description: *Users must call lockTokens repeatedly for multiple locks*

Recommendation:

Add a batchLock function

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

11. Unlicensed Code

Severity: **LOW**

Location: CPROLocking.sol

Description: *The UNLICENSED SPDX identifier is not recommended for production*

Recommendation:

Add MIT/GPL/Apache license at top of the contract

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

12. Missing function specifications

Severity: **LOW**

Location: CPROLocking.sol

Description: *Some functions (e.g., emergencyWithdraw) do not have function specifications (e.g. @param/@return docs)*

Recommendation:

Add missing function specifications everywhere

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

13. PUSH0 Opcode

Severity: **LOW**

Location: CPROLocking.sol

Description: *Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes*

Recommendation:

Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

14. Costly operations inside loop

Severity: **HIGH**

Location: CPROLocking.sol, function *batchUnlock*, line 302

Description: *Invoking SSTORE operations in loops may waste gas:*

```
for (uint256 i = 0; i < lockIds.length; i++) {}
```

Recommendation:

Use a local variable to hold the loop computation result.

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

15. Not checking return value

Severity: **LOW**

Location: CPROLocking.sol, line 342

Description: *Function returns a value but it is ignored*

```
IERC20(_token).transfer(owner(), amount)
```

Recommendation:

Consider checking the return value.

CPRO Team: **acknowledged**

Audit Team: Waiting for fix implementation

Security Controls Review

Control	Status	Notes
Access Control	✓	<code>onlyOwner</code> used for owner functions (<code>lockTokensFor</code> , <code>emergencyWithdraw</code>)
Pause Mechanism	✓	Properly implemented with modifiers
Input validation:	✓	<code>amount > 0</code> , <code>unlockTime > block.timestamp</code> , <code>user != address(0)</code> checks present — good
External calls	✓	Uses <code>IERC20.transfer/transferFrom</code>

Reentrancy	✗	<code>nonReentrant</code> applied on <code>unlockTokens</code> and <code>batchUnlock</code> — good.
Time-dependence	✓	Uses <code>block.timestamp</code> — acceptable for timelocks (expected)
Overflow / Underflow	✓	Safe under Solidity ≥ 0.8
Randomness sources	✗	None - ok

Gas & Performance Observations

- Per-lock storage with string is costly. If many locks are expected, prefer `bytes32` or enumerated types.
- `getUserTotalLocked` and `getUserUnlockableAmount` are view loops over `userLocks`. Views are fine but heavy loops could be expensive if called on-chain (they are viewed so only off-chain nodes pay execution cost; avoid calling from contracts).
- `batchUnlock` limit of 50 is a reasonable mitigation against gas exhaustion for user-side batched unlocks.