Crypto Autos

SMART CONTRACTS REVIEW

zokyo

August 4th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passed a security audit.

ZOKYO AUDIT SCORING CRYPTOAUTOS

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 0 Medium issues: 0 points deducted
- 1 Low issue: 1 resolved = 0 points deducted
- 1 Informational issue: 1 resolved = 0 points deducted

Thus, the score is 100

TECHNICAL SUMMARY

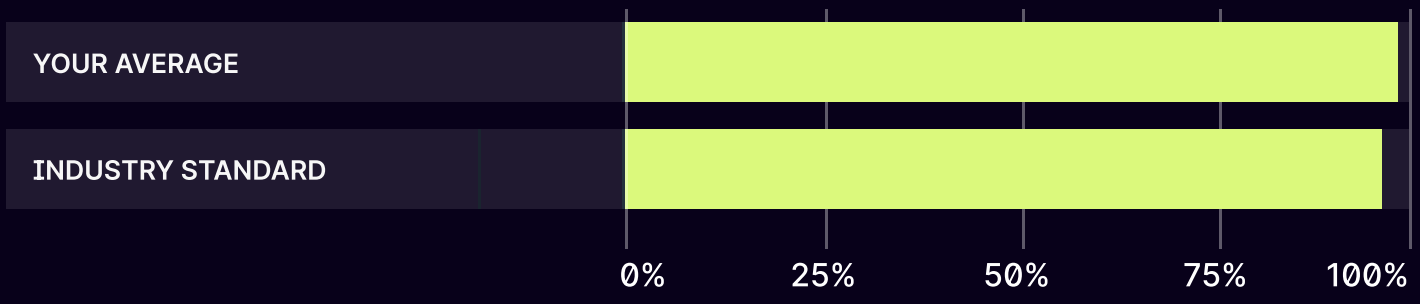
This document outlines the overall security of the CryptoAutos smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the CryptoAutos smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

Testable Code



100% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the CryptoAutos team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the CryptoAutos repository:
Repo: https://github.com/Baboons-dev/crypto-autos-staking-contract

Last commit - 1c2e5d1633acb27bc6f2e4647743aa11f5ca7fa4

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- Staking20Base.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of CryptoAutos smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The Staking20Base smart contract is an ERC-20 token staking system derived from Thirdweb's Staking20 contract, designed to allow users to stake a single ERC-20 token and earn rewards in either the same or a different ERC-20 token. It inherits access control through the Ownable contract and implements core staking mechanics via Staking20. The contract is initialized with staking and reward token addresses, reward ratio parameters, token decimals, and a designated admin. The reward token is stored in the contract and distributed to stakers based on staking time and the predefined reward ratio. The contract includes a global cooldown mechanism managed by the admin, which temporarily disables withdrawal and reward claiming for all users when active. Reward distribution is handled by the internal _mintRewards function, which transfers tokens to stakers and deducts the distributed amount from the internal rewardTokenBalance. Admin functions include setting the cooldown period, depositing reward tokens using transferFrom, and withdrawing excess reward tokens, all protected by nonReentrant and ownership checks. The actual amount of reward tokens received during deposit is calculated by measuring the token balance before and after transfer, which is useful for handling fee-on-transfer tokens. The contract ensures that after reward withdrawals, the staking token balance in the contract remains sufficient to cover user stakes. This design is suitable for simple staking systems where a single token is staked and rewards are distributed on a fixed ratio basis under full admin control.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the CryptoAutos team and the CryptoAutos team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Use of block.timestamp + durationInSeconds for Cooldown Timing can get displaced	Low	Resolved
2	Inconsistent Adherence to the Checks-Effects-Interactions (CEI) Pattern	Informational	Resolved

Use of block.timestamp + durationInSeconds for Cooldown Timing can get displaced

Description:
The setCooldown(uint256 durationInSeconds) function sets the cooldown expiration by adding the specified durationInSeconds to block.timestamp. While this approach may appear convenient, it introduces a potential vector for front-running. An attacker can observe a pending transaction with a predictable cooldown and submit their own transaction with a higher gas price, causing the original transaction to be mined later than intended. As a result, the actual cooldown end time would be later than expected by the original sender.

Impact:
Using block.timestamp + durationInSeconds allows the cooldown end time to vary depending on transaction inclusion time, which can be manipulated via front-running. This undermines the determinism of the cooldown logic and can lead to unintended delays or timing mismatches in time-sensitive operations, especially in systems that rely on precise cooldown scheduling for security or economic correctness.

Recommendation:
Modify the function to accept an explicit cooldownEndsAt timestamp as an input parameter, rather than computing it based on the current block timestamp. This ensures that the cooldown period is determined entirely by the caller and is not subject to displacement by miner ordering or front-running. For example:

```
function setCooldown(uint256 cooldownEndsAt_) external virtual
nonReentrant {
    require(msg.sender == owner(), "Not authorized");
    require(cooldownEndsAt_ >= block.timestamp, "Cooldown must be in the
future");
    cooldownEndsAt = cooldownEndsAt_;
    emit CooldownSet(cooldownEndsAt);
}
```

This approach improves predictability and mitigates front-running risks associated with variable execution times.

Inconsistent Adherence to the Checks-Effects-Interactions (CEI) Pattern

Description:
The contract does not consistently follow the Checks-Effects-Interactions (CEI) pattern, particularly in functions involving ERC-20 token transfers, such as _mintRewards(), and _depositRewardsTokens(). In some instances, external interactions (i.e., token transfers) are performed before updating internal state variables like rewardTokenBalance.

Although non-reentrancy modifiers (e.g., nonReentrant) may mitigate immediate risks, adhering to the CEI pattern is a best practice in Solidity development. The pattern improves contract robustness by reducing the surface area for reentrancy and other state-dependent vulnerabilities.

Impact:
While not currently working with native tokens, where reentrancy introduces a direct and exploitable risk, deviating from the CEI pattern is not recommended neither when using ERC20 tokens

Recommendation:
Refactor the affected functions to strictly follow the CEI pattern by:

Checks – Performing all necessary validation and access control.

Effects – Updating all internal state variables (e.g., rewardTokenBalance) before any external calls.

Interactions – Executing external calls (e.g., token transfers) only after all internal state changes.

	Staking20Base.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting CryptoAutos in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the CryptoAutos contract requirements for details about issuance amounts and how the system handles these.

Ran 35 tests for test_zokyo/unit_test.t.sol:Staking20BaseUnitTest
[PASS] test_ClaimRewards_MultipleTimeUnits() (gas: 242261)
[PASS] test_ClaimRewards_RevertIf_CooldownActive() (gas: 211556)
[PASS] test_ClaimRewards_RevertIf_InsufficientRewards() (gas: 218225)
[PASS] test_ClaimRewards_Success() (gas: 241742)
[PASS] test_Constructor_BytecodeDeployment_EdgeCases() (gas: 223380)
[PASS] test_Constructor_RevertIf_RewardTokenZero() (gas: 53632)
[PASS] test_Constructor_RevertIf_StakingTokenZero() (gas: 48068)
[PASS] test_Constructor_Success() (gas: 39285)
[PASS] test_Constructor_SuccessPath_BothTokensValid() (gas: 3723211)
[PASS] test_Constructor_WrapperApproach_CoverBranches() (gas: 3826503)
[PASS] test_DepositRewardTokens_RevertIf_NotOwner() (gas: 19283)
[PASS] test_DepositRewardTokens_Success() (gas: 66304)
[PASS] test_DepositRewardTokens_TransferFromFailure() (gas: 3771557)
[PASS] test_DepositRewardTokens_WithDifferentAmounts() (gas: 100638)
[PASS] test_GetCooldownRemaining_WhenNotActive() (gas: 44597)
[PASS] test_MintRewards_TransferFailure() (gas: 3973912)
[PASS] test_MultipleUsersStakingAndRewards() (gas: 353684)
[PASS] test_SetCooldown_RevertIf_NotOwner() (gas: 19243)
[PASS] test_SetCooldown_Success() (gas: 43964)
[PASS] test_SetOwner_RevertIf_NotOwner() (gas: 15894)
[PASS] test_SetOwner_Success() (gas: 22385)
[PASS] test_SetRewardRatio_RevertIf_NotOwner() (gas: 14348)
[PASS] test_SetRewardRatio_Success() (gas: 103435)
[PASS] test_SetTimeUnit_RevertIf_NotOwner() (gas: 14005)
[PASS] test_SetTimeUnit_Success() (gas: 101250)
[PASS] test_Stake_Success() (gas: 196013)
[PASS] test_WithdrawRewardTokens_ExcessAmount() (gas: 40988)
[PASS] test_WithdrawRewardTokens_RevertIf_NotOwner() (gas: 19056)
[PASS] test_WithdrawRewardTokens_RevertIf_StakingTokenBalanceReduced() (gas: 6447374)
[PASS] test_WithdrawRewardTokens_StakingTokenBalanceCheck() (gas: 214051)

[PASS] test_WithdrawRewardTokens_Success() (gas: 54647)
[PASS] test_WithdrawRewardTokens_TernaryBranches() (gas: 68038)
[PASS] test_WithdrawRewardTokens_TransferFailure() (gas: 3803005)
[PASS] test_Withdraw_RevertIf_CooldownActive() (gas: 211339)
[PASS] test_Withdraw_Success() (gas: 220410)
Suite result: ok. 35 passed; 0 failed; 0 skipped; finished in 21.86ms (56.42ms CPU time)

Ran 12 tests for test_zokyo/fuzz_test.t.sol:Staking20BaseFuzzTest
[PASS] testFuzz_Constructor_ValidInputs(uint80,uint256,uint256,uint8,uint8) (runs: 1007, μ: 6049334, ~: 6049590)
[PASS] testFuzz_Cooldown_ValidDurations(uint256) (runs: 1007, μ: 48803, ~: 49472)
[PASS] testFuzz_DepositRewardTokens_ValidAmounts(uint256) (runs: 1007, μ: 69859, ~: 69365)
[PASS] testFuzz_ExcessWithdrawRewardTokens_EdgeCase(uint256) (runs: 1007, μ: 58307, ~: 58348)
[PASS] testFuzz_InvariantStakingTokenBalance(uint256,uint256) (runs: 1007, μ: 250342, ~: 250807)
[PASS] testFuzz_MultipleStakeWithdraw_Sequences(uint256,uint256,uint256) (runs: 1007, μ: 299634, ~: 299424)
[PASS] testFuzz_RewardCalculation_TimeUnits(uint256) (runs: 1007, μ: 261578, ~: 261827)
[PASS] testFuzz_RewardRatio_Updates(uint256,uint256) (runs: 1007, μ: 109553, ~: 108628)
[PASS] testFuzz_Stake_ValidAmounts(uint256) (runs: 1007, μ: 214931, ~: 214286)
[PASS] testFuzz_TimeUnit_Updates(uint80) (runs: 1007, μ: 107362, ~: 107336)
[PASS] testFuzz_WithdrawRewardTokens_ValidAmounts(uint256) (runs: 1007, μ: 60389, ~: 59841)
[PASS] testFuzz_Withdraw_ValidAmounts(uint256,uint256) (runs: 1007, μ: 256727, ~: 256384)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 1.47s (10.09s CPU time)

Ran 2 test suites in 1.47s (1.49s CPU time): 47 tests passed, 0 failed, 0 skipped (47 total tests)

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
Staking20Base.sol	100	91.67%	100	100
All Files	**100**	**91.67%**	**100**	**100**

We are grateful for the opportunity to work with the CryptoAutos team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the CryptoAutos team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo