

AUKI

SMART CONTRACTS REVIEW



September 26th 2024 | v. 1.0

Security Audit Score

PASS

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



ZOKYO AUDIT SCORING AUKI LABS

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

SCORING CALCULATION:

Let's assume each issue has a weight:

- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:

- 0 Critical issues: 0 points deducted
- 2 High issues: 1 resolved and 1 acknowledged = - 7 points deducted
- 3 Medium issues: 3 resolved = 0 points deducted
- 0 Low issues: 0 points deducted
- 2 Informational issues: 2 resolved = 0 points deducted

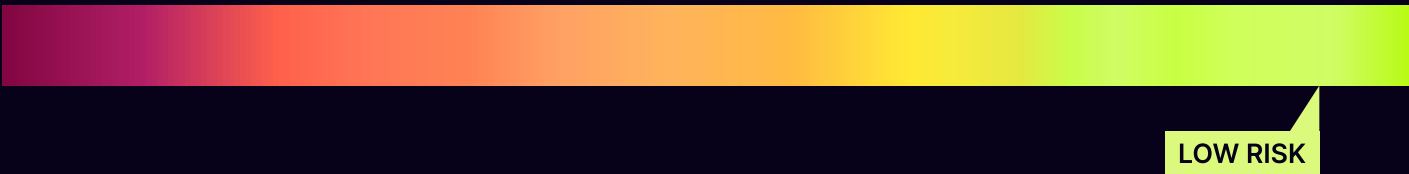
Thus, $100 - 7 = 93$

TECHNICAL SUMMARY

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

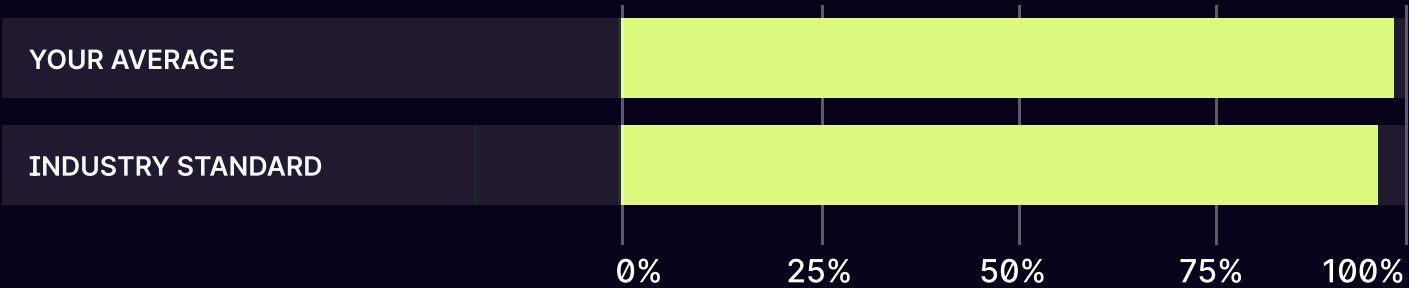
The scope of this audit was to analyze and document the Auki Labs smart contract/s codebase for quality, security, and correctness.

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 Auki Labs 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	5
Executive Summary	7
Structure and Organization of the Document	8
Complete Analysis	9
Code Coverage and Test Results for all files written by Zokyo Security	15

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Auki Labs repository:

Repo: <https://github.com/aukilabs/auki-contracts/pull/71>

Last commit -5540163bb5c7c395ecf9fd1aa39036afbb83e9c3

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

- StakingWithDelayedWithdrawalContract.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 Auki Labs smart contract/s. To do so, the code was reviewed line by line by our smart contract/s developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Truffle/Hardhat 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 **StakingWithDelayedWithdrawalContract** is a smart contract built for staking ERC20 tokens, using Solidity 0.8.19 and OpenZeppelin's upgradeable libraries. It allows users to stake a specific token, referred to as the `stakingToken`, with defined periods for thawing and withdrawal. The thawing period refers to the time staked tokens must remain locked before becoming available, while the withdrawal period refers to the time that must pass after a withdrawal request is made before tokens can be withdrawn.

The contract implements role-based access control, assigning roles such as `PAUSER_ROLE`, `SLASHER_ROLE`, `CONFIG_ROLE`, and `UPGRADER_ROLE` to manage critical functions. This ensures that only authorized users can perform sensitive actions, such as pausing the contract in emergencies.

Users can stake tokens by granting the contract allowance or by using EIP-2612 permits, which allow third-party staking. The withdrawal process is carefully managed, requiring users to request withdrawals, which can be revoked if necessary. This ensures that funds are only accessible once they have fully thawed.

The contract tracks total staked tokens and individual staker information using a `Stake` struct, which includes the staker's address, the staked amount, and relevant timestamps for thawing and availability. Additionally, the contract emits events for key actions like staking and withdrawal requests, improving transparency and facilitating easier monitoring.

Overall, the **StakingWithDelayedWithdrawalContract** provides a clear and efficient method for managing token staking in decentralized finance applications.

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 Auki Labs team and the Auki Labs 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	Front-running in slash Function	High	Resolved
2	Unregulated Slashing without Rules and Governance	High	Acknowledged
3	Risk of Arbitrary Withdrawal Period Changes	Medium	Resolved
4	Unchecked transfers return value	Medium	Resolved
5	The wrong value is emitted	Medium	Resolved
6	Could use delete keyword	Informational	Resolved
7	Unnecessary requirement	Informational	Resolved

Front-running in `slash` Function

The `slash` function can only be called when the contract is not paused, which leaves the protocol vulnerable to front-running attacks. A malicious staker who is about to be slashed and already requested withdrawal could monitor the network for slashing transactions and front-run them by withdrawing their staked tokens before the `slash` function is executed. This would allow them to avoid the slashing penalty entirely.

Recommendation:

Once a wallet requested withdrawal and the withdrawal period passes, we should consider their status withdrawn and make their server read/delete-only, and don't receive rewards.

Unregulated Slashing without Rules and Governance

The `slash` function allows any account with the `SLASHER_ROLE` to slash stakers without any clearly defined rules or governance mechanism. This unregulated slashing power could be abused, leading to unjustified slashing of stakers, resulting in a loss of staked tokens without valid cause or due process.

Recommendation:

Implement slashing rules and governance controls to ensure that slashing decisions are justified and transparent. Governance mechanisms should include:

- 1. Slashing Conditions:** Define clear criteria for when a staker can be slashed. For example, slashing could be limited to specific offenses, such as protocol violations or malicious behavior.
- 2. Governance Voting:** Require a governance vote or multi-signature approval process to initiate slashing actions, ensuring community oversight and reducing the likelihood of abuse.
- 3. Timelock:** Introduce a timelock mechanism before slashing takes effect, giving stakers the opportunity to appeal the slashing decision or withdraw under certain circumstances.

Risk of Arbitrary Withdrawal Period Changes

The `setWithdrawalPeriod` function allows an account with the `CONFIG_ROLE` to update the withdrawal period at any time. This introduces a critical risk where the withdrawal period could be extended to an excessively long duration after users have already staked their funds, effectively locking them out from withdrawing their assets for an indefinite period.

This kind of manipulation could trap user funds, leading to loss of trust in the protocol.

Recommendation:

To prevent this issue, store the withdrawal period at the time the user stakes their funds, ensuring that any subsequent changes to the global withdrawal period do not affect already staked funds. This ensures that the withdrawal conditions for each staker remain consistent throughout their staking duration.

Unchecked transfers return value

ERC20 token transfers are performed using the `transfer` and `transferFrom` methods in the contract. These methods return a boolean value indicating the success of the operation, as per the ERC20 standard. However, the contract does not check these return values, which can lead to scenarios where token transfers fail silently. This can result in inconsistent contract states and potential loss of funds.

Recommendation:

Use `SafeERC20` library from `OpenZeppelin`, which handles these inconsistencies and ensures compatibility.

The wrong value is emitted

The `_requestWithdrawal()` function emits `WithdrawalRequested()` event and the first parameter of the event is designed to be stakers addresses.

But the function emits `msg.sender` for the first parameter of the event but if the owner calls the function, then the owner address is emitted instead of the staker address.

Recommendation:

Replace `msg.sender` with `_staker`.

Could use delete keyword

In the `_withdraw()` function, each element (`_s.amountStaked`, `_s.owner`, `_s.thawingTimestamp`, `_s.availableTimestamp`) of `stakes[_staker]` is updated to `0` even if there is a “delete” keyword to delete the entire struct storage.

The same for the `slash()` function.

Recommendation:

Use “delete `stakes[_staker]`” to remove stakers data.

Unnecessary requirement

In the `_withdraw()` function, `_thawingTimestamp` is compared with `block.timestamp` and then `_availableTimestamp` is compared with `0` and `block.timestamp`. However, if `block.timestamp` is greater than `_availableTimestamp`, it means the current time passed the `_thawingTimestamp`.

Recommendation:

Remove a comparison between `_thawingTimestamp` and `block.timestamp`.

StakingWithDelayedWithdrawalContract.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 Auki Labs in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Truffle/Hardhat testing framework.

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

```
[PASS] testAuthorizeUpgrade() (gas: 2917785)
[PASS] testConstructorAndInitializer() (gas: 3090109)
[PASS] testNonUpgraderCannotUpgrade() (gas: 2938304)
[PASS] testPauseWithoutPermission() (gas: 46918)
[PASS] testRequestWithdrawal() (gas: 205501)
[PASS] testRequestWithdrawalByOwner() (gas: 255252)
[PASS] testRequestWithdrawalFailedWhenPaused() (gas: 215594)
[PASS] testRequestWithdrawalTwice() (gas: 197989)
[PASS] testRequestWithdrawalWithoutFullyThawed() (gas: 169736)
[PASS] testRequestWithdrawalWithoutPermission() (gas: 175470)
[PASS] testRequestWithdrawalWithoutStake() (gas: 26592)
[PASS] testSetStakeAmount() (gas: 60109)
[PASS] testSetStakeAmountToZero() (gas: 15783)
[PASS] testSetWithdrawalPeriod() (gas: 60222)
[PASS] testSetWithdrawalPeriodToZero() (gas: 15849)
[PASS] testSlash() (gas: 167033)
[PASS] testSlashAfterStakeWithPermit() (gas: 206244)
[PASS] testSlashFailedWhenPaused() (gas: 191751)
[PASS] testSlashFromNonAdmin() (gas: 203293)
[PASS] testSlashingZeroToken() (gas: 22387)
[PASS] testStake() (gas: 274981)
[PASS] testStakeEvent() (gas: 166880)
[PASS] testStakeFailedWhenPaused() (gas: 184386)
[PASS] testStakeTwice() (gas: 189391)
[PASS] testStakeWithPermit() (gas: 229640)
[PASS] testStakeWithPermitAndStake() (gas: 257667)
[PASS] testStakeWithPermitBelowStakeAmount() (gas: 34167)
[PASS] testStakeWithPermitWithWrongContract() (gas: 31904)
[PASS] testUnPauseWithoutPermission() (gas: 73762)
[PASS] testUpdateThawingPeriod() (gas: 62684)
```

```

[PASS] testUpdateThawingPeriodToZero() (gas: 15798)
[PASS] testWithdraw() (gas: 186619)
[PASS] testWithdrawAsStakerAfterStakeWithPermit() (gas:
220071)
[PASS] testWithdrawBeforePeriod() (gas: 197987)
[PASS] testWithdrawByOwner() (gas: 225181)
[PASS] testWithdrawWithoutFullyThawed() (gas: 169747)
[PASS] testWithdrawWithoutPermission() (gas: 203712)
[PASS] testWithdrawWithoutRequest() (gas: 172222)
[PASS] testWithdrawalFailedWhenPaused() (gas: 235497)

```

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	%UNCOVERED LINES
StakingWithDelayedWithdrawalContract.sol	100	100	100	100	
All Files	100	100	100	100	

We are grateful for the opportunity to work with the Auki Labs 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 Auki Labs team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

