



SMART CONTRACT AUDIT REPORT

for

AltixStaking



Prepared By: Xiaomi Huang

PeckShield
October 23, 2025

Document Properties

Client	ALTIX
Title	Smart Contract Audit Report
Target	AltixStaking
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 23, 2025	Xuxian Jiang	Final Release
1.0-rc	October 23, 2025	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About AltixStaking	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Improper totalRewardsDeposited Accounting in recoverUnreservedRewards()	11
3.2	Revisited claimAndWithdraw() Logic in AltixStaking	12
3.3	Trust Issue of Admin Keys	13
4	Conclusion	15
	References	16

1 | Introduction

Given the opportunity to review the design document and related source code of the `AltixStaking` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About AltixStaking

The `ALTIX` token ecosystem is composed of an `ERC-20` deflationary utility token and a fixed-term staking system that distributes rewards from a pre-funded pool (without minting). This audit covers the the fixed-term staking system, i.e., `AltixStaking`. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of `AltixStaking`

Item	Description
Name	ALTIX
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 23, 2025

In the following, we show the deployment address of the audited contract as well as the related checksum values of the given package for audit.

- `AltixStaking`: <https://amoy.polygonscan.com/address/0x90ce77c169322bcc533f9aaeccb6911524d66110>
- `ALTIX_Audit_Testnet_Package.zip`: 46aa3f549a1bb841255bc3711271e845 (MD5)

And here is the checksum value of the `AltixStaking` contract after all fixes for the issues found in the audit have been applied:

- AltixStaking: 815e9f6954bd0ca757a0205c0ccc535a (MD5)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `AltixStaking` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	
Low	1	
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key AltixStaking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper totalRewardsDeposited Accounting in recoverUnreservedRewards()	Business Logic	Resolved
PVE-002	Low	Revisited claimAndWithdraw() Logic in AltixStaking	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improper totalRewardsDeposited Accounting in recoverUnreservedRewards()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: AltixStaking
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned earlier, `AltixStaking` is a fixed-term staking system that rewards the staker with fixed APRs based on the staked duration. The contract also provides a privileged function to recover rewards that are not reserved yet. While examining its logic, we notice an issue that does not properly keep track of the total amount of deposited rewards, i.e., `totalRewardsDeposited`.

In the following, we show the implementation of the related function `recoverUnreservedRewards()`. As the name indicates, this function is used to recover any rewards that may not be reserved yet. However, it comes to our attention that the withdrawal of unreserved rewards needs to timely deduct `totalRewardsDeposited` by the withdrawal amount, i.e., `totalRewardsDeposited -= amount;`. Otherwise, the privileged account, if compromised, may abuse this function to drain all funds on the contract, including all user stakes.

```

78     function recoverUnreservedRewards(address to, uint256 amount) external onlyOwner {
79         require(to != address(0), "to=0");
80         uint256 available = rewardsAvailable();
81         require(amount <= available, "exceeds available");
82         token.safeTransfer(to, amount);
83         emit RewardsRecovered(to, amount);
84     }

```

Listing 3.1: `AltixStaking::recoverUnreservedRewards()`

Recommendation Revisit the `totalRewardsDeposited` accounting of in the above `recoverUnreservedRewards()` routine.

Status This issue has been resolved by deducting the recovered unreserved amount from `totalRewardsDeposited` in `recoverUnreservedRewards()`.

3.2 Revisited `claimAndWithdraw()` Logic in `AltixStaking`

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AltixStaking`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.1, the `AltixStaking` contract a fixed-term staking system that allows users to stake user funds. And the staked funds may be withdrawn after the expiry of chosen stake duration. Our analysis shows the withdrawal logic may be revisited as it can be completely paused. A decentralized design may pause the user deposits, but not user withdrawals.

In the following, we show the implementation of the related `claimAndWithdraw()` function, which is used to withdraw previous stakes after their expiry. We notice this function has a `whenNotPaused` modifier, which indicates it is subject to the protocol pause control. In other words, if the owner chooses to pause this contract, all user stakes may not be able to withdraw. As mentioned earlier, a better approach may allow to pause the deposit upon user entry, but not the withdrawal.

```

104     function claimAndWithdraw(uint256 stakeId) external whenNotPaused nonReentrant {
105         StakeInfo storage s = stakes[stakeId];
106         require(s.staker == msg.sender, "not staker");
107         require(!s.withdrawn, "already withdrawn");
108         require(block.timestamp >= s.endTs, "not matured");
109         s.withdrawn = true;
110         totalRewardsReserved -= s.reward;
111         totalRewardsPaid += s.reward;
112         token.safeTransfer(msg.sender, s.amount);
113         token.safeTransfer(msg.sender, s.reward);
114         emit Withdrawn(msg.sender, stakeId, s.amount, s.reward);
115     }

```

Listing 3.2: `AltixStaking::claimAndWithdraw()`

Recommendation Revise the above-mentioned routine to allow users to withdraw unconditionally.

Status This issue has been resolved by adding a new `emergencyWithdraw()` function to allow users to withdraw when the contract is paused.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AltixStaking`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

The `AltixStaking` protocol is designed with a privileged account, i.e., `owner`, that play a critical role in governing and regulating the system-wide operations (e.g., configure parameters, manage reward plans, and execute privileged operations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

58     function depositRewards(uint256 amount) external onlyOwner {
59         require(amount > 0, "amount=0");
60         token.safeTransferFrom(msg.sender, address(this), amount);
61         totalRewardsDeposited += amount;
62         emit RewardsDeposited(msg.sender, amount);
63     }

65     function setPlanActive(uint8 planId, bool isActive) external onlyOwner {
66         require(planId < plans.length, "planId");
67         plans[planId].active = isActive;
68         emit PlanStatusChanged(planId, isActive);
69     }

70     function setPlanAPR(uint8 planId, uint16 newApr) external onlyOwner {
71         require(planId < plans.length, "planId");
72         require(newApr <= 20, "apr too high");
73         plans[planId].apr = newApr;
74         emit PlanAPRChanged(planId, newApr);
75     }

76     function pause() external onlyOwner { _pause(); }
77     function unpause() external onlyOwner { _unpause(); }
78     function recoverUnreservedRewards(address to, uint256 amount) external onlyOwner {
79         require(to != address(0), "to=0");
80         uint256 available = rewardsAvailable();
81         require(amount <= available, "exceeds available");
82         token.safeTransfer(to, amount);
83         emit RewardsRecovered(to, amount);

```

Listing 3.3: Example Privileged Operations in AltixStaking

We understand the need of the privileged function for contract maintenance, but at the same time the extra power to the privileged account may also be a counter-party risk to the protocol users. It is worrisome if the privileged account is plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated with the use of a multi-sig for the privileged account.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `AltixStaking` protocol, which is part of the `ALTIX` token ecosystem. The ecosystem is composed of an `ERC-20` deflationary utility token and a fixed-term staking system that distributes rewards from a pre-funded pool (without minting). This audit covers the the fixed-term staking system, i.e., `AltixStaking`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [5] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.