Smart Contract Audit Report

Sharp AI Staking Contract underwent a comprehensive audit on December 7, 2024

Smart Contract	staking.sol
Type Of Utility	Staking
Platform	ETH, Ethereum Virtual Machine
Language	Solidity
Method	Manual and Statics Analysis
Address	0x7900E2F5eC37dA2098881729Fda6784e31e02Aa8

AVERAGE Security Score

The score is determined by analyzing the lines of code and assigning weights to issues based on their severity and confidence levels. To enhance your score, review the detailed results and apply the recommended remediation strategies.

Vulnerability Summary

- **2** Critical
- **0** High
- **0** Medium
- **1** Low
- **8** Information

Content

Classification and Severity

Critical

This vulnerability could lead to significant consequences, such as the loss or mismanagement of funds, or other severe financial impacts.

High

High-severity vulnerabilities represent a major risk to the Smart Contract and the organization. They could result in user fund losses under certain conditions and are difficult to exploit.

Medium

This issue affects the functionality of the contract but does not cause substantial disruption to its overall operations.

Low

This issue has a minor impact on the contract's functionality and does not significantly affect its operation.

Information

This issue does not interfere with the contract's functionality but addressing it would follow best practices.

Audit Scope

This Audit Report mainly focuses on the overall security of the **Sharp AI** token Staking Smart Contract. This audit was conducted with rigorous attention to the general implementation of the contract and by examining the overall architectural layout of the software implementation. The reliability and correctness of this smart contract's codebase are being assessed.

The auditing process pays special attention to the following considerations:

- Identifies security related issues within each contract and the system of contract.
- A full assessment of the code quality and general software architecture patterns and best practices used.

Audit Method

Rigorous testing of the project has been performed. Detailed code base analysis was conducted, reviewing the smart contract architecture to ensure it is structured and safe.

A detailed, line by line inspection of the codebase was conducted to find any potential security vulnerabilities such as denial of service attacks, race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

Automated and manual testing was employed that included:

- Analysis of on-chain data security
- Analysis of the code in-depth and detailed, manual review of the code, line-by-line.
- Deployment of the code on an in-house testnet blockchain and running live tests
- Determining failure preparations and if worst-case scenario protocols are in place
- Analysis of any third-party code use and verifying the overall security of this

Findings

This report has been developed to identify issues and vulnerabilities in Sharp AI Staking Smart Contract. During the audit, we uncovered 11 issues of varying severity levels. We employed Manual Review and Static Analysis alongside thorough manual code reviews to identify the following findings.

ID	Title	Severity	Status
C001	Incorrect Access Control	Critical	Acknowledged
L001	Use Of Floating Pragma	Low	Acknowledged
I001	Hard-Coded Address Detected	Information	Acknowledged
I002	Block Values As A Proxy For Time	Information	Acknowledged
I003	Name Mapping Parameters	Information	Acknowledged

C001.1 - Incorrect Access Control (Emergency Withdraw)

Title	Severity	Status
Incorrect Access Control (Emergency Withdraw)	Critical	Acknowledged

Description

The `emergencyWithdraw` function is restricted to `ADMIN_ROLE`. However, there is no mechanism to prevent unauthorized addresses from being assigned this role. If an attacker gains control over the `ADMIN_ROLE`, they could drain all tokens from the contract, leading to significant financial loss for the project and its users.

```
154     function emergencyWithdraw(uint256 _amount) external onlyRole(ADMIN_ROLE) {
155         require(_amount > 0, "Amount must be greater than 0");
156         require(stakingToken.balanceOf(address(this)) >= _amount, "Insufficient contract balance");
157 
158         stakingToken.safeTransfer(adminWallet, _amount);
159         emit EmergencyWithdrawal(msg.sender, _amount);
160     }
161 
```

Recommendation

We recommend implementing stricter controls on role assignment. Only the `DEFAULT_ADMIN_ROLE` should be allowed to grant or revoke the `ADMIN_ROLE`. Additionally, consider using a multi-signature wallet to manage sensitive roles.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

C001.2 - Incorrect Access Control (Default Role Management)

Title	Severity	Status
Incorrect Access Control (Default Role Management)	Critical	Acknowledged

Description

The `DEFAULT_ADMIN_ROLE` has unrestricted authority over role management and is initially assigned to the deployer(`msg.sender`) without further safeguards. If the deployer's private key is compromised, all access control could be overridden, leading to unauthorized execution of critical functions.

```
60        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
61        _grantRole(ADMIN_ROLE, _adminWallet);
```

Recommendation

We recommend transferring the `DEFAULT_ADMIN_ROLE` to a multi-signature wallet to minimize the risk of compromise. Additionally, introducing a time-delay mechanism for role changes can provide an opportunity for review.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

L001.1 - Use Of Floating Pragma

Title	Severity	Status
Use Of Floating Pragma	Low	Acknowledged

Description

The contract specifies the pragma version as `^0.8.27` , which is a floating pragma. Floating pragmas allow the contract to compile with newer versions of Solidity within the specified range. While this can make it easier to adopt bug fixes and new features, it also increases the risk of introducing unexpected behavior or vulnerabilities due to changes in the compiler.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.27;
3
```

Recommendation

It is recommended to use a fixed pragma version instead of a floating version to ensure the contract compiles with the intended version of the Solidity compiler. Update the pragma to: `^0.8.27`

This guarantees consistent behavior during compilation and reduces the risk of introducing compiler-specific issues.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I001.1 - Hard-Coded Address Detected

Title	Severity	Status
Hard-Coded Address Detected	Information	Acknowledged

Description

The contract contains a hard-coded address for the `stakingToken`. Hard-coding addresses can lead to inflexibility and potential issues if the address needs to be changed in the future. It also makes the contract less reusable for different deployments or networks.

```
53        address tokenAddress = 0xAddB6dC7E2F7caEA67621DD3Ca2e8321ade33286;
54        stakingToken = IERC20(tokenAddress);
```

Recommendation

Replace the hard-coded address with a constructor parameter or an initialization function to allow dynamic assignment of the token address at deployment. This approach improves flexibility and makes the contract more adaptable to various environments.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I002.1 - Block Values As A Proxy For Time (Lock Time Setting)

Title	Severity	Status
Block Values As A Proxy For Time (Lock Time Setting)	Information	Acknowledged

Description

This line uses `block.timestamp` to calculate and set the lock end time for a staking tier. While this is a standard approach, miners can manipulate `block.timestamp` within a range of 15 seconds, which might result in slight inaccuracies for lock durations.

```
76          userBalances[msg.sender][_tier] += _amount;
77          userLocks[msg.sender][_tier] = block.timestamp + tierConfig.lockTime;
78
79          totalStaked += _amount;
80          totalLocked += _amount;
81
82          stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
83
84          emit Staked(msg.sender, _amount, _tier, userLocks[msg.sender][_tier]);
85      }
```

Recommendation

If precision is essential, consider using an external time oracle like Chainlink for tamper-resistant timestamps. If `block.timestamp` is retained, clearly document the acceptable tolerance for such inaccuracies.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I002.2 - Block Values As A Proxy For Time (Early Unstaking Check)

Title	Severity	Status
Block Values As A Proxy For Time (Early Unstaking Check)	Information	Acknowledged

Description

This line compares `block.timestamp` with the user's lock period to check for early unstaking. Miner manipulation `block.timestamp` could allow users to unstake slightly earlier than the intended lock period.

```
91
92        if (block.timestamp < userLocks[msg.sender][_tier]) {
93            require(allowEarlyUnstake, "Early unstaking not allowed");
```

Recommendation

Add a small buffer to the lock time or use an external oracle to validate timestamps. For less critical applications, ensure that the risk of minor timestamp manipulation is documented and acceptable.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I002.3 - Block Values As A Proxy For Time (Penalty Exemption)

Title	Severity	Status
Block Values As A Proxy For Time (Penalty Exemption)	Information	Acknowledged

Description

`block.timestamp` is compared to the lock end time to determine if a penalty should apply. A minor deviation in `block.timestamp` could allow users to avoid penalties slightly before their actual lock time expires.

```
127         if (block.timestamp >= _lockEndTime) {
128             return 0;
129         }
```

Recommendation

Consider padding the lock end time slightly to reduce the risk of manipulation or use an oracle for precise time data. If `block.timestamp` is kept, ensure that minor discrepancies are considered in the penalty calculations.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I002.4 - Block Values As A Proxy For Time (Penalty Calculation)

Title	Severity	Status
Block Values As A Proxy For Time (Penalty Calculation)	Information	Acknowledged

```
/// @dev Calculates the penalty for early unstaking
function _calculatePenalty(uint256 _amount, uint256 _penaltyRate, uint256 _lockEndTime) private view returns (uint256) {
    if (block.timestamp >= _lockEndTime) {
        return 0;
    }
    return (_amount * _penaltyRate) / 100;
}
```

Description

This line indirectly relies on `block.timestamp` through `_calculatePenalty` to calculate the penalty amount for early unstaking. Slight inaccuracies in the timestamp could lead to unintended results in penalty deductions.

Recommendation

To ensure penalty calculations are accurate, consider validating `block.timestamp` using an external time oracle or implementing error tolerance within the calculation logic. For example, add a margin to account for potential manipulation.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I003.1 - Name Mapping Parameters (User Balances)

Title	Severity	Status
Name Mapping Parameters (User Balances)	Information	Acknowledged

```
/// @notice Unstake tokens from a specific tier
function unstake(Tier _tier) external nonReentrant whenNotPaused {
    uint256 amount = userBalances[msg.sender][_tier];
    require(amount > 0, "No staked balance in this tier");

    if (block.timestamp < userLocks[msg.sender][_tier]) {
        require(allowEarlyUnstake, "Early unstaking not allowed");
    }

    (uint256 penaltyAmount, uint256 withdrawalAmount) = _processUnstake(amount, _tier);

    userBalances[msg.sender][_tier] = 0;

    if (block.timestamp >= userLocks[msg.sender][_tier]) {
        totalUnlocked += amount;
    }
```

Description

This mapping stores the balance of tokens staked by a user in each tier. While descriptive, using `userBalances` as a name may not clearly communicate the tier-specific context of the balances at first glance.

Recommendation

Consider renaming the mapping to `tieredUserBalances` or a similar name to make it more explicit that this mapping relates to balances per staking tier. Clearer naming improves readability and reduces ambiguity during audits or debugging.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

I003.2 - Name Mapping Parameters (Lock Expiry Times)

Title	Severity	Status
Name Mapping Parameters (Lock Expiry Times)	Information	Acknowledged

```
/// @dev Processes the unstake logic, calculates penalties and updates totals
function _processUnstake(uint256 _amount, Tier _tier) private returns (uint256, uint256) {
    uint256 penaltyAmount = _calculatePenalty(_amount, tiers[_tier].penalty, userLocks[msg.sender][_tier]);
    uint256 withdrawalAmount = _amount - penaltyAmount;

    totalStaked -= _amount;
    totalLocked -= _amount;

    if (penaltyAmount > 0) {
        stakingToken.safeTransfer(adminWallet, penaltyAmount);
        emit PenaltyApplied(msg.sender, penaltyAmount, _tier);
```

Description

This mapping tracks the lock expiration time for each user in a specific staking tier. The name `userLocks` is functional but could be more descriptive to indicate it relates to lock expiration.

Recommendation

Update the mapping name to something more descriptive, like `userLockExpiryTimes`, to explicitly reflect its purpose. This adjustment enhances the clarity of the contract's purpose and improves maintainability.

Alleviation

[`Staking Contract`]: Issue acknowledged.

I003.3 - Name Mapping Parameters (Tier Settings)

Title	Severity	Status
Name Mapping Parameters (Tier Settings)	Information	Acknowledged

StakingFinal.sol

```
contract Staking is AccessControl, ReentrancyGuard, Pausable {
    struct TierConfiguration {
        uint256 rewardRate;
        uint256 penalty;
    }

    mapping(Tier => TierConfiguration) public tiers;
    mapping(address => mapping(Tier => uint256)) public userBalances;
    mapping(address => mapping(Tier => uint256)) public userLocks;

    event Staked(address indexed user, uint256 amount, Tier indexed tier, uint256 lockEndTime);
    event Unstaked(address indexed user, uint256 amount, Tier indexed tier, uint256 penaltyAmount);
    event PenaltyApplied(address indexed user, uint256 penaltyAmount, Tier indexed tier);
    event TierConfigurationUpdated(Tier indexed tier, uint256 lockTime, uint256 rewardRate, uint256 penalty);
    event EmergencyWithdrawal(address indexed admin, uint256 amount);
    event ContractPaused(address indexed admin);
    event ContractUnpaused(address indexed admin);
```

Description

This mapping stores configuration details for each staking tier. The name `tiers` is concise but lacks detail to indicate it specifically pertains to tier configurations.

Recommendation

Rename the mapping to something more self-explanatory, such `tierConfigurations`. This change helps future developers and auditors immediately understand its purpose without delving into the contract's code.

Alleviation

[`Staking Contract` **]**: Issue acknowledged.

Conclusion

The **Staking** contract is a well-designed and secure implementation of a tiered staking system, offering a strong foundation for decentralized finance applications. It incorporates robust features such as tier-based staking, configurable lock times, and penalty mechanisms, ensuring reliability and user safety.

However, while the contract is secure and safe to use in its current form, it can be further enhanced by addressing the following areas:

1. **Prevent Abuse:** Add mechanisms to prevent rapid restaking abuse after early withdrawals, ensuring fair use of the system.
2. **Admin Controls:** Strengthen administrative actions by introducing multi-signature approvals for sensitive operations like tier updates and emergency withdrawals, reducing risks of misuse.
3. **Penalty Calculation:** Refine penalty logic to address edge cases and ensure fair and consistent application across all scenarios.

These improvements will elevate the contract to align with industry-leading best practices, enhancing both user trust and system resilience.

Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. To get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HedgePay and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HedgePay) owe no duty of care towards you or any other person, nor does HedgePay make any warranty or representation to any person on the accuracy or completeness of the report.

The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HedgePay hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HedgePay hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HedgePay, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.