ZEALYNX SECURITY

Web3 Security & Smart Contract Development

Ample Protocol

Security Assessment

May 19th, 2025

Prepared by Zealynx Security
contact@zealynx.io
Zealynx.io

Sergio	@Seecoalba
Bloqarl	@TheBlockChainer

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Cyfrin, Monadex, Lido, Inverter, Ribbon Protocol, and Paragon.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent security assessment of the AMPLE Staking Smart Contract was done by Zealynx Security, with a focus on the security aspects of the application's implementation.

We performed the security evaluation based on the agreed scope during 7 days, following our approach and methodology. Based on the scope and our performed activities, our security assessment revealed 1 High, 3 Medium, 4 Low and 11 Informational security issues.

3.2 About AMPLE Protocol

Website : ampleprotocol.xyz
Docs: ample-protocol

Ample is a decentralized protocol that reimagines how intellectual property (IP) is created, licensed, and monetized. By tokenizing IP assets and enabling on-chain licensing and royalty automation, Ample provides creators, brands, and IP owners with transparent, programmable tools to manage their rights without intermediaries. Built with chain abstraction and cross-chain compatibility, it ensures seamless interoperability across platforms and marketplaces.

The protocol includes modular infrastructure for IP issuance, compliance, governance, and funding, making it adaptable to various industries. Staking and rewards mechanisms, including NFT-based boosts, incentivize user participation while supporting ecosystem growth. These features complement a broader goal of enhancing liquidity and access to IP as a digital asset class. Ample aligns with the future of Web3 by democratizing ownership and enabling scalable, transparent IP markets.

3.3 About Codebase

The code under review is composed of single smart contract written in Solidity language and includes 125 nSLOC - normalized source lines of code.

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, **Formal Verification**, and **meticulous manual security review.**

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality**: We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices**: Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments**: We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over a 7-day engagement, the Zealynx Security team - including both senior auditors and interns - conducted a security audit of the StakingContract.sol implementation developed by the Ample team. The audit focused on identifying vulnerabilities, logic flaws, and implementation-level issues that could affect reward integrity, staking behavior, and the reliability of the protocol's incentive system.

A total of 21 issues were identified and categorized as follows:

- 1 High severity
- 4 Medium severity
- 5 Low severity
- 11 Informational

The initial commit hash that was audited:
`45f12c5d992d82c14166aba34a455fe9412045ba`

The key findings include:

- **Lack of reward token segregation:** The contract does not separate user-staked tokens from reward tokens, creating a risk of reward payments depleting user deposits. This could result in protocol insolvency under high withdrawal conditions.

- **Tier update mismatch:** Modifying staking tiers does not automatically update the lockPeriods array, leading to inconsistencies between configured parameters and what users see or select in the UI.

- **Missing NFT ownership check on withdrawal:** Boosted rewards are granted based on initial NFT ownership, but not revalidated at the time of withdrawal. This creates an opportunity for abuse through NFT renting or delegation.

- **No cap on reward funding:** The contract lacks enforcement of the documented 10% token supply limit for reward allocation. Without this check, excess tokens may be mistakenly or maliciously allocated to the staking pool.

While the contract achieves its core staking functionality, improvements in validation logic, upgradeability support, and accounting separation are recommended to strengthen protocol safety and future extensibility.

Summary of Findings :

Vulnerability	Severity
[H-01] Lack of reward token segregation leads to unreliable reward accounting and potential underpayment	High
[M-01] Tier updates not reflected in lockPeriods array leads to stale staking period information	Medium
[M-02] Lack of NFT ownership check on closePosition enables boost market exploitation	Medium
[M-03] Missing max reward allocation check enables exceeding 10% token supply limit	Medium
[M-04] Missing pause/unpause functions prevents emergency protocol shutdown	Medium
[L-01] Manual reward funding mechanism poses operational risk	Low
[L-02] Non-upgradeable contract may require complex migration for future changes	Low
[L-03] stakeTokens function accepts any lock period with non-zero tier value	Low
[L-04] Custom ownership implementation lacks secure ownership transfer mechanism	Low

Vulnerability	Severity
[L-05] Missing initializer protection enables potential initialization hijacking	Low
[I-01] Redundant allowance check	Informational
[I-02] Consider adding a totalStaked per address getter	Informational
[I-03] Missing position existence check leads to misleading errors	Informational
[I-04] Inconsistent terminology between rewards and interest	Informational
[I-05] Redundant initialization of uint to zero	Informational
[I-06] Unused IStakingContract interface	Informational
[I-07] Redundant positionId in Position struct	Informational
[I-08] Consider using named mappings for improved code readability	Informational
[I-09] Consider updating to latest Solidity version	Informational
[I-10] Use custom errors instead of string revert messages	Informational
[I-11] Inconsistent Use of ReentrancyGuard	Informational

7. Audit Findings

7.1 High Severity Findings

[H-01] Lack of reward token segregation leads to unreliable reward accounting and potential underpayment

Description

The StakingContract doesn't differentiate between staked tokens and reward tokens in its balance. When rewards are funded, they're simply added to the contract's balance without any internal tracking:

```
function fundRewards(uint256 amount) external onlyOwner {
    // Tokens are added to contract balance
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);
    // No internal tracking of rewards vs staked tokens
    emit RewardsFunded(msg.sender, amount);
}
```

When users stake or withdraw, the contract has no way to distinguish between:

- Tokens that were staked by users.
- Tokens that were funded as rewards.
- Tokens that are reserved for promised interest.

Impact

- The contract's inability to differentiate between staked tokens and reward tokens creates a dangerous situation where the protocol might be promising rewards it cannot fulfill.
- Since the contract balance combines both staked tokens and rewards, it becomes impossible to verify if there are sufficient rewards to cover all promised interest. This could lead to a scenario where the contract inadvertently uses staked tokens to pay rewards, effectively creating a fractional reserve system.
- In a high-withdrawal scenario, this could result in later users being unable to withdraw their funds, as the contract might have used their staked tokens to pay earlier users' rewards.

Proof of Concept

```
function test_POC_InsolvencyRisk() public {
    // Setup multiple users for the demonstration
    uint256 stakeAmount = 100 * 10**18; // 100 tokens with 18 decimals
    uint256 numUsers = 5;

    // Create staking positions for multiple users
    for (uint256 i = 0; i < numUsers; i++) {
        address userAddress = users[i].addr;

        // Fund the user with tokens
        vm.startPrank(address(testToken));
        testToken.mint(userAddress, stakeAmount);
        vm.stopPrank();

        // User stakes tokens
        vm.startPrank(userAddress);
        testToken.approve(address(stakingContract), stakeAmount);
        stakingContract.stakeTokens(30, stakeAmount);
        vm.stopPrank();
    }

    // Calculate total promised funds (principal + interest)
    uint256 totalPromisedFunds = 0;
    for (uint256 i = 0; i < stakingContract.currentPositionId(); i++) {
        StakingContract.Position memory position = stakingContract.getPositionById(i);
        if (position.open) {
            totalPromisedFunds += position.tokensStaked + position.tokensInterest;
        }
    }

    // Check current contract balance
    uint256 initialContractBalance = testToken.balanceOf(address(stakingContract));
    emit log_named_uint("Current contract balance", initialContractBalance);
    emit log_named_uint("Total promised to users", totalPromisedFunds);

    // Simulate fund drainage (accidental withdrawal by owner or hack)
    // Leave only half of the promised amount in the contract
    uint256 drainAmount = initialContractBalance - (totalPromisedFunds / 2);

    vm.startPrank(address(stakingContract));
    testToken.transfer(address(0xdead), drainAmount);
    vm.stopPrank();

    // Verify the new balance after drainage
    uint256 balanceAfterDrainage = testToken.balanceOf(address(stakingContract));
    emit log_named_uint("Balance after drainage", balanceAfterDrainage);
    assertTrue(balanceAfterDrainage < totalPromisedFunds, "Contract should have insufficient funds");

    // Advance time to allow position closures
    vm.warp(block.timestamp + 31 days);

    // First users can withdraw their funds (first-come, first-served)
    vm.startPrank(users[0].addr);
    stakingContract.closePosition(0);
    vm.stopPrank();

    vm.startPrank(users[1].addr);
    stakingContract.closePosition(1);
    vm.stopPrank();

    // Check remaining balance after first withdrawals
    uint256 balanceAfterFirstWithdrawals = testToken.balanceOf(address(stakingContract));
    emit log_named_uint("Balance after first withdrawals", balanceAfterFirstWithdrawals);

    // Last user attempts to withdraw but will not receive full amount
    address lastUser = users[4].addr;
    uint256 initialLastUserBalance = testToken.balanceOf(lastUser);

    vm.startPrank(lastUser);
    try stakingContract.closePosition(4) {
        // If it doesn't revert, check if user received less than promised
        StakingContract.Position memory lastPosition = stakingContract.getPositionById(4);
        uint256 promisedAmount = lastPosition.tokensStaked + lastPosition.tokensInterest;
        uint256 lastUserFinalBalance = testToken.balanceOf(lastUser);
        uint256 actuallyReceived = lastUserFinalBalance - initialLastUserBalance;

        emit log_named_uint("Promised to last user", promisedAmount);
        emit log_named_uint("Actually received by last user", actuallyReceived);

        if (actuallyReceived < promisedAmount) {
            emit log("Last user received less than promised - VULNERABILITY CONFIRMED");
        } else {
            emit log("ERROR: Last user should have received less than promised");
            assert(false);
        }
    } catch {
        // If transaction reverts, it's because the contract has insufficient funds
        emit log("Transaction failed due to insufficient funds - VULNERABILITY CONFIRMED");
    }
    vm.stopPrank();
}
```

Recommendation

1. Add separate accounting for rewards:

```
uint256 private _totalRewardPool;
uint256 private _totalPromisedRewards;

function fundRewards(uint256 amount) external onlyOwner {
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);
    _totalRewardPool += amount;
    emit RewardsFunded(msg.sender, amount);
}

function stakeTokens(uint numDays, uint amount) external nonReentrant {
    uint256 interest = calculateInterest(tiers[numDays], amount);
    require(_totalRewardPool ⩾ _totalPromisedRewards + interest,
            "Insufficient rewards available");
    _totalPromisedRewards += interest;
    // ... rest of function
}
```

2. Add reward tracking in position closure:

```
function closePosition(uint positionId) external nonReentrant {
    Position storage position = positions[positionId];
    // ... validation checks

    _totalPromisedRewards -= position.tokensInterest;
    _totalRewardPool -= position.tokensInterest;

    // Transfer principal + interest
    stakingToken.safeTransfer(msg.sender,
        position.tokensStaked + position.tokensInterest);
}
```

3. Add view functions for transparency:

```
function getRewardMetrics() external view returns (
    uint256 availableRewards,
    uint256 promisedRewards
) {
    return (_totalRewardPool, _totalPromisedRewards);
}
```

Ample Protocol:

Fixed

Zealynx:

Verified

7.2 Medium Severity Findings

[M-01] Tier updates not reflected in lockPeriods array leads to stale staking period information

Description

The StakingContract maintains staking period information in two separate data structures: tiers and lockPeriods. When updating tiers through the updateTier function, the corresponding lockPeriods array is not updated to reflect these changes:

```
function updateTier(uint numDays, uint basisPoints) external onlyOwner {
    require(basisPoints ⩽ 10000, "Invalid basis points");
    uint oldRate = tiers[numDays];
    tiers[numDays] = basisPoints;
    // lockPeriods is not updated here
    emit TierUpdated(numDays, oldRate, basisPoints);
}
```

This creates a misalignment between the actual available staking periods in tiers and what's exposed through the getLockPeriods() function:

```
function getLockPeriods() external view returns (uint[] memory) {
    return lockPeriods;  // Returns stale data that doesn't reflect tier updates
}
```

Impact

- Stale Data: getLockPeriods() returns outdated information that doesn't reflect the current state of available staking periods after tier updates.
- UI/Integration Inconsistency: Users and integrated protocols receive incorrect information about available staking periods, as getLockPeriods() doesn't reflect the updates made through updateTier.
- Protocol Confusion: Creates confusion about which staking periods are actually valid, as the two data structures maintain different states.

Recommendation

Modify updateTier to maintain synchronization between tiers and lockPeriods:

```
function updateTier(uint numDays, uint basisPoints) external onlyOwner {
    require(basisPoints ⩽ 10000, "Invalid basis points");

    uint oldRate = tiers[numDays];

    // If this is a new period, add it to lockPeriods
    if (tiers[numDays] == 0) {
        lockPeriods.push(numDays);
    }

    // Update the tier
    tiers[numDays] = basisPoints;

    emit TierUpdated(numDays, oldRate, basisPoints);
}
```

Ample Protocol:

Fixed

Zealynx:

Verified

[M-02] Lack of NFT ownership check on closePosition enables boost market exploitation

Description

The StakingContract only checks NFT ownership at stake time and permanently stores the boosted rewards in the position struct. This allows for the creation of NFT rental markets specifically designed to exploit the boost mechanism:

```
// NFT check only happens once at stake time
if (isNftHolder(msg.sender)) {
    boostedInterest = (baseInterest * rewardBoostMultiplier) / 100;
}

// Boosted rewards are permanently stored
positions[currentPositionId] = Position({
    tokensInterest: boostedInterest,  // Locked at stake time
    // ...
});

// No NFT verification at withdrawal
function closePosition(uint positionId) external nonReentrant {
    uint totalTokens = position.tokensStaked + position.tokensInterest;
    stakingToken.safeTransfer(msg.sender, totalTokens);
}
```

This design enables the creation of centralized "boost-as-a-service" markets where:

- NFT holders can rent out their NFTs for brief moments
- Stakers can rent NFTs just for the staking transaction
- Multiple stakers can use the same NFT to get boosted rewards

Impact

The vulnerability undermines the entire NFT boost mechanism:

1. Protocol loses ability to reward genuine long-term NFT holders
2. Single NFT can be used to boost multiple positions
3. Creates a secondary market that commoditizes the boost feature
4. Economic impact equals (boostedInterest - baseInterest) for each exploited position

Recommendation

During mitigation period, we identified an improved approach to handle NFT boosts. We recommend storing both the base and boosted reward values in the Position struct, along with tracking the initial NFT ownership status.

1. Update the Position struct to track both rewards and NFT status

```
struct Position {
    address walletAddress;
    uint createdDate;
    uint unlockDate;
    uint lockedDays;
    uint baseReward;      // Store base reward
    uint finalReward;     // Store boosted reward if applicable
    bool initialNftHolder; // Track if user had NFT when staking
    uint tokensStaked;
    bool open;
}
```

2. In stakeTokens(), capture NFT status and both reward values

```
function stakeTokens(uint numDays, uint amount) external nonReentrant whenNotPaused {
    // ... existing validation ...

    stakingToken.safeTransferFrom(msg.sender, address(this), amount);

    // Capture NFT status at staking time
    bool userHasNft = isNftHolder(msg.sender);

    // Calculate both reward values
    uint baseReward = _calculateReward(tiers[numDays], amount);
    uint finalReward = userHasNft
        ? (baseReward * rewardBoostMultiplier) / 100
        : baseReward;

    // Track promised rewards
    _totalPromisedRewards += finalReward;

    // Store all values in the position
    positions[currentPositionId] = Position({
        walletAddress: msg.sender,
        createdDate: block.timestamp,
        unlockDate: block.timestamp + (numDays * 1 days),
        lockedDays: numDays,
        baseReward: baseReward,
        finalReward: finalReward,
        initialNftHolder: userHasNft,
        tokensStaked: amount,
        open: true
    });

    // ... rest of function ...
}
```

3. In closePosition(), use stored values with conditional logic

```
function closePosition(uint positionId) external nonReentrant {
    // ... existing validation ...

    Position storage pos = positions[positionId];

    // Determine final reward based on initial and current NFT status
    uint finalReward = (isNftHolder(msg.sender) && pos.initialNftHolder)
        ? pos.finalReward
        : pos.baseReward;

    _totalPromisedRewards -= finalReward;
    pos.open = false;

    uint payout = pos.tokensStaked + finalReward;
    stakingToken.safeTransfer(msg.sender, payout);

    // ... rest of function ...
}
```

This implementation ensures:

1. Users who had NFTs when staking and still have them at withdrawal get the boosted reward
2. Users who had NFTs when staking but no longer have them at withdrawal get the base reward
3. Users who didn't have NFTs when staking but acquired them later don't get retroactively boosted

Ample Protocol:

Fixed

Zealynx:

The team implemented our recommendation to track NFT ownership, but we identified an additional optimization during verification.
Recommendation has been updated.

Update: Suggestion implemented and verified.

[M-03] Missing max reward allocation check enables exceeding 10% token supply limit

Description

The StakingContract allows the owner to fund rewards without any limit through the fundRewards function:

```
function fundRewards(uint256 amount) external onlyOwner {
    require(amount > 0, "Amount must be greater than zero");
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);
    emit RewardsFunded(msg.sender, amount);
}
```

According to the protocol's tokenomics (shown in README.md), only 10% of the total supply should be allocated for rewards. However, there is no on-chain enforcement of this limit, which could lead to overfunding beyond the intended allocation.

Impact

If the protocol becomes successful and active for a long period:

- Owner could accidentally fund more than 10% of total supply.
- This breaks the tokenomics promises made to users and investors.
- Excessive rewards dilute token value.
- Could affect other token allocations (team, partnerships, etc.).

Recommendation

1. Import and inherit from OpenZeppelin's Pausable contract to manage staking state:

```
import "@openzeppelin/contracts/security/Pausable.sol";
```

2. Add state variables to track rewards:

```
uint256 public constant MAX_REWARD_ALLOCATION = 10; // 10%
uint256 public totalRewardsFunded;
```

3. Expose pause/unpause functions (restricted to owner):

```
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
```

4. Modify the fundRewards function to enforce the 10% limit:

```
function fundRewards(uint256 amount) external onlyOwner {
    require(amount > 0, "Amount must be greater than zero");

    uint256 maxRewards = (stakingToken.totalSupply() * MAX_REWARD_ALLOCATION) / 100;
    require(totalRewardsFunded + amount ⩽ maxRewards,
            "Exceeds max reward allocation");

    totalRewardsFunded += amount;
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);

    if (totalRewardsFunded ⩾ maxRewards) {
        _pause();
    }

    emit RewardsFunded(msg.sender, amount);
}
```

5. Add whenNotPaused modifier to staking function:

```
function stakeTokens(uint numDays, uint amount) external whenNotPaused {
    // ... rest of function
}
```

This ensures that:

- The protocol never exceeds its promised 10% reward allocation
- Staking is automatically paused if the maximum allocation is reached
- Owner has explicit control over pausing/unpausing through dedicated functions
- Maintains transparency with events for rewards funded and contract state changes

Ample Protocol:

Fixed

Zealynx:

A new issue has been raised [M-04] since the mitigation was not followed exactly as proposed.

Update: Fixed and verified.

[M-04] Missing pause/unpause functions prevents emergency protocol shutdown

Description

This issue has been found during mitigation period. It was part of the recommended mitigation for the max reward allocation issue [M-03].

The contract inherits from PausableUpgradeable but doesn't implement the necessary functions to utilize this functionality.

The contract initializes the pausable functionality with __Pausable_init() but doesn't expose any functions to actually pause or unpause the contract.

Impact

If a critical vulnerability is discovered or if the contract needs to be temporarily halted for any reason:

1. The team cannot pause the contract to prevent further exploitation
2. Users can continue interacting with a potentially vulnerable contract
3. The emergency mitigation mechanism is completely non-functional

This is particularly concerning as the pause functionality was meant to be a safety mechanism for emergency situations.

Recommendation

Add pause and unpause functions:

```
/**
 * @notice Pauses the contract, preventing staking and unstaking
 * @dev Only callable by owner
 */
function pause() public onlyOwner {
    _pause();
}

/**
 * @notice Unpauses the contract, allowing staking and unstaking
 * @dev Only callable by owner
 */
function unpause() public onlyOwner {
    _unpause();
}
```

Ample Protocol:

Fixed

Zealynx:

Verified

7.3 Low Severity Findings

[L-01] Manual reward funding mechanism poses operational risk

Description

The contract relies entirely on manual funding through the fundRewards() function:

```
function fundRewards(uint256 amount) external onlyOwner {
    require(amount > 0, "Amount must be greater than zero");
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);
    emit RewardsFunded(msg.sender, amount);
}
```

While it's expected that the team will fund rewards, relying solely on manual funding introduces operational risks:

1. Human error in timing of funding
2. Key person risk if funding coordinator is unavailable
3. No automated top-up mechanism
4. No warning system for low balance

Recommendation

Consider implementing safety mechanisms around manual funding:

1. Add monitoring events:

```
event LowBalance(uint256 currentBalance, uint256 threshold);
```

2. Add a minimum balance threshold:

```
uint256 public constant MIN_BALANCE_THRESHOLD = 1000e18;

function checkBalance() public {
    if (stakingToken.balanceOf(address(this)) < MIN_BALANCE_THRESHOLD) {
        emit LowBalance(stakingToken.balanceOf(address(this)), MIN_BALANCE_THRESHOLD);
    }
}
```

Ample Protocol:

Fixed

Zealynx:

Verified. Both event for monitoring added and also the checkBalance function.

[L-02] Non-upgradeable contract may require complex migration for future changes

Description

The contract is currently implemented as a non-upgradeable contract:

```
contract StakingContract is ReentrancyGuard {
    // No upgrade mechanism
}
```

However, the team has indicated plans for future modifications. Without upgradeability:

1. New features will require new contract deployment
2. Users will need to migrate positions manually
3. Complex coordination for rewards and NFT boosts
4. Risk of user confusion during migration
5. Potential for parallel contracts running simultaneously

Recommendation

Consider implementing an upgradeability pattern of your choice.

Ample Protocol:

Fixed - Implemented TransparentUpdableProxy

Zealynx:

A new issue has been found and reported [L-05]
Update: fixed and verified

[L-03] stakeTokens accepts any lock period with non-zero tier value

Description

The contract's validation for staking periods only checks if the tier value is greater than zero:

```
function stakeTokens(uint numDays, uint amount) external nonReentrant {
    require(tiers[numDays] > 0, "Invalid lock period");
    // ... rest of function
}
```

This means:

- Any period that has a non-zero tier value is accepted, even if not in lockPeriods array
- Users interacting directly with the contract could stake for unintended periods
- Frontend dropdown could be changes to a numerical field

Recommendation

Use a mapping for verification if period is allowed:

```
mapping(uint period => bool allowed) public isAllowedPeriod;

constructor(
    address _stakingToken,
    address _nftContract,
    uint _rewardBoostMultiplier
) {
    // ... other initialization

    // Initialize allowed periods
    isAllowedPeriod[30] = true;
    isAllowedPeriod[60] = true;
    isAllowedPeriod[90] = true;

    // Set tiers as before...

}

function stakeTokens(uint numDays, uint amount) external nonReentrant {
    require(isAllowedPeriod[numDays], "Lock period not in allowed list");
    require(tiers[numDays] > 0, "Invalid lock period");
    // ... rest of function
}
```

Ample Protocol:

Fixed

Zealynx:

Verified. isAllowedPeriod introduced

[L-04] Custom ownership implementation lacks secure ownership transfer mechanism

Description

The contract implements a basic ownership mechanism:

```
address public owner;

modifier onlyOwner() {
    require(msg.sender == owner, "Only owner can call this function");
    _;
}
```

However, this implementation:

1. Lacks ability to transfer ownership (no transferOwnership function)
2. Has no two-step ownership transfer process
3. Could lead to permanent loss of admin functions if owner address is compromised

This is particularly concerning given the protocol's roadmap includes transitioning to DAO governance, which will require a secure ownership transfer mechanism.

It's recommended to replace the custom ownership implementation with OpenZeppelin's Ownable2Step.

Ample Protocol:

Fixed

Zealynx:

Verified

[L-05] Missing initializer protection enables potential initialization hijacking

Description

This issue has been raised during the mitigation period while reviewing [L-02]

The upgradeable StakingContract implementation is missing the _disableInitializers() call in its constructor.

When using the transparent proxy pattern with OpenZeppelin's upgradeable contracts, the implementation contract should have its initializers disabled to prevent attackers from directly initializing the implementation contract (not the proxy).

Without this protection, an attacker could potentially:

1. Call initialize() directly on the implementation contract
2. Set themselves as the owner of the implementation
3. Potentially cause confusion or interfere with proxy operations

Add a constructor that disables initializers:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Ample Protocol:

Fixed

Zealynx:

Verified

7.4 Informational Findings

[I-01] Redundant allowance check

Description

The stakeTokens function performs an explicit allowance check before calling safeTransferFrom:

```
uint allowed = stakingToken.allowance(msg.sender, address(this));
require(allowed ⩾ amount, "Allowance too low. Please approve more tokens.");
stakingToken.safeTransferFrom(msg.sender, address(this), amount);
```

This check is redundant since safeTransferFrom already includes an allowance verification. The duplicate check results in unnecessary gas consumption from an extra external call.

[I-02] Consider adding a totalStaked per address getter

Description

The contract provides getPositionIdsForAddress to retrieve all position IDs for a wallet, but lacks a convenient way to get the total amount staked across all positions for a specific user. Currently, to get this information, a client would need to:

1. Call getPositionIdsForAddress
2. Iterate through each ID
3. Call getPositionById for each position
4. Sum up the tokensStaked values

It might be useful to add a getTotalStakedForAddress() function.

[I-03] Missing position existence check leads to misleading errors

Description

In the closePosition function, there's no validation that the positionId exists before accessing it:

```
function closePosition(uint positionId) external nonReentrant {
    Position storage position = positions[positionId];

    require(position.walletAddress == msg.sender, "Only position creator may modify position");
    require(position.open == true, "Position is already closed");
    // ...
}
```

If a user provides a positionId >= currentPositionId, the position will be empty (all fields zero-initialized). This leads to misleading error message:

- If msg.sender is rightfully trying to close his position but entered the wrong positionId: "Only position creator may modify position"

Add an explicit existence check at the start of the function:

require(positionId < currentPositionId, "Position does not exist");

[I-04] Inconsistent terminology between rewards and interest

Description

The contract uses both "rewards" and "interest" terms interchangeably, which could lead to confusion. For example:

```
// Uses "rewards" terminology
function fundRewards(uint256 amount) external onlyOwner {
    // ...
}
event RewardsFunded(address indexed funder, uint256 amount);

// But uses "interest" terminology
struct Position {
    uint256 tokensInterest;  // Should be tokensReward
    uint256 percentInterest; // Should be percentReward
}

function calculateInterest(uint basisPoints, uint tokenAmount) private pure {
    // Should be calculateReward
}
```

The protocol documentation and UI refer to these as "rewards", so the contract should maintain consistency with this terminology.

We recommend to align all variable and function names to use "reward" terminology.

[I-05] Redundant initialization of uint to zero

Description

In the constructor, currentPositionId is explicitly initialized to 0:

```
constructor(
    address _stakingToken,
    address _nftContract,
    uint _rewardBoostMultiplier
) {
    currentPositionId = 0;  // Redundant initialization
    // ...
}
```

In Solidity, all state variables are automatically initialized to their default values. For uint, this default value is 0, making this initialization redundant and consuming unnecessary gas.

[I-06] Unused IStakingContract interface

Description

The codebase contains an IStakingContract interface file that is not being used by the StakingContract. This creates confusion as:

1. The interface exists but is not imported
2. The contract doesn't implement it
3. There are discrepancies between interface and implementation

```
// Current contract:
contract StakingContract is ReentrancyGuard {
    // No interface implementation
}
```

Either remove the interface if it's not needed or properly implement it.

[I-07] Redundant positionId in Position struct

Description

The Position struct includes a positionId field that duplicates the key used in the positions mapping:

```
struct Position {
    uint positionId;      // Redundant with mapping key
    address walletAddress;
    // ... other fields
}

mapping(uint ⇒ Position) public positions; // positionId is already the key
```

While this might seem convenient for frontend integration or event emission, it:

1. Wastes gas by storing the same data twice
2. Requires maintaining consistency between mapping key and struct field
3. Increases contract deployment cost

It is recommended to remove the redundant positionId from the struct and use the mapping key when needed.

[I-08] Consider using named mappings for improved code readability

Description

The contract uses traditional mapping syntax instead of the more readable named mapping syntax introduced in Solidity 0.8.18:

```
// Current implementation
mapping(uint ⇒ Position) public positions;
mapping(address ⇒ uint[]) public positionIdsByAddress;
mapping(uint ⇒ uint) public tiers;
```

Named mappings make the code more self-documenting by explicitly stating what the keys and values represent.

Update to latest stable Solidity version and use named mappings:

```
// More readable implementation
mapping(uint positionId ⇒ Position position) public positions;
mapping(address user ⇒ uint[] positionIds) public positionIdsByAddress;
mapping(uint daysLocked ⇒ uint basisPoints) public tiers;
```

This improves:

- Code readability
- Self-documentation
- Maintenance and auditing
- Function parameter consistency (names can match mapping key/value names)

[I-09] Consider updating to latest Solidity version

Description

The contract uses an older Solidity version:

```
pragma solidity ^0.8.0;
```

Recent Solidity versions (e.g., 0.8.23) include important features and optimizations such as:

- Named mappings for better readability
- Push0 opcode support for gas optimization
- Custom errors with error.selector
- Using for with free functions
- Improved overflow checks

Update to the latest stable version of Solidity.

[I-10] Use custom errors instead of string revert messages

Description

The contract uses string messages in require statements throughout the code:

```
require(_stakingToken != address(0), "Staking token address cannot be zero");
require(_nftContract != address(0), "NFT contract address cannot be zero");
require(msg.sender == owner, "Only owner can call this function");
// ... more require statements with strings
```

Custom errors, introduced in Solidity 0.8.4, are more gas-efficient and provide better error handling capabilities:

- Save deployment gas (no need to store error strings)
- Save runtime gas (cheaper than require with strings)
- Allow passing parameters for better error details
- Provide clearer error handling in the frontend

Replace require statements with custom errors:

```
error ZeroAddress(string parameter);
error Unauthorized(address caller, string role);
error InvalidAmount(uint amount, string reason);
```

[I-11] Inconsistent Use of ReentrancyGuard

Description

The fundRewards function lacks the nonReentrant modifier while other token-interacting functions (stakeTokens, closePosition) implement it. This inconsistency appears to be an oversight during the addition of this new function:

```
/**
 * ✅ New Function: Fund rewards manually
 * Allows owner to send extra tokens to the contract so users can claim their interest.
 */
function fundRewards(uint256 amount) external onlyOwner {
    // Missing nonReentrant modifier
}
```

While there's no immediate security risk due to trusted tokens, this inconsistency in security patterns could cause issues if the contract is upgraded or integrated with different tokens in the future.

Add the nonReentrant modifier to fundRewards for consistency with other token-interacting functions.