

Lynxcap - Audit

Security Assessment

CertiK Assessed on Sept 10th, 2026





CertiK Assessed on Sept 10th, 2026

Lynxcap - Audit

The security assessment was prepared by CertiK.

Executive Summary

TYPES

Staking

ECOSYSTEM

EVM Compatible

METHODS

Formal Verification, Manual Review, Static Analysis

LANGUAGE

Solidity

TIMELINE

Preliminary comments published on 04/28/2026

Final report published on 09/10/2026

Vulnerability Summary



16

Total Findings

11

Resolved

0

Partially Resolved

5

Acknowledged

0

Declined

2 Centralization

2 Acknowledged



Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.

0 Critical

Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.

4 Major

3 Resolved, 1 Acknowledged



Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.

4 Medium

4 Resolved



Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.

6 Minor

4 Resolved, 2 Acknowledged



Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.

0 Informational

Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | LYNXCAP - AUDIT

Audit Summary

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

Findings

[LYA-01 : Centralization Risks In LynxStakeV2.Sol](#)

[LYA-02 : Centralized Control Of Contract Upgrade](#)

[LYA-03 : Admin Can Fabricate Unbacked Positions And Extract Real User Deposits](#)

[LYA-04 : Inactive Strike Double-Pays Previously Claimed APR Interest](#)

[LYA-05 : Insufficient Funds For Withdrawals Due To Unfunded Position Creation](#)

[LYA-06 : `claimFull` Returns Full `stakeAmount` Regardless Of `mintedLiquidityAmount`, Enabling Potential Double-Claim](#)

[LYA-07 : Compound Positions Stop Compounding After Interest Checkpoints](#)

[LYA-08 : User Set Beneficiary Is Not Honored](#)

[LYA-09 : Legacy V1 Positions Use Current Global `termSeconds`, Enabling Retroactive Maturity Changes](#)

[LYA-15 : Interest Continues Accruing After Fixed-Term Position Maturity](#)

[LYA-10 : Missing Zero Address Validation](#)

[LYA-11 : `openReferralRewardPosition` Bypasses KYC Stake Threshold Enforcement](#)

[LYA-12 : Inconsistent State Update In `strikeInactive` Inflates User's Staked Amount](#)

[LYA-13 : `isMature` View Function Reverts Instead Of Returning False](#)

[LYA-14 : Unbounded Loop In `claimBatch` Causes DoS](#)

[LYA-16 : `transferPosition` Copies `mintedLiquidityAmount` Without Updating The Withdrawal Library](#)

Appendix

Disclaimer

CODEBASE | LYNXCAP - AUDIT

Repository

Initial commit:

<https://github.com/LynxCapIT/smartcontract/tree/2d09d3fbb02d21931360b1673f0eaa0daa9a924a>

Remediation commit: <https://github.com/LynxCapIT/smartcontract/tree/8ab736b7708cff324dc5fd7f7dfe7d36c787da88>

AUDIT SCOPE | LYNXCAP - AUDIT

LynxCapIT/smartcontract

 src/V2/LynxStakeV2.sol

APPROACH & METHODS | LYNXCAP - AUDIT

This audit was conducted for Lynxcap to evaluate the security and correctness of the smart contracts associated with the Lynxcap - Audit project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Formal Verification, Manual Review, and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

FINDINGS | LYNXCAP - AUDIT



16

Total Findings

0

Critical

2

Centralization

4

Major

4

Medium

6

Minor

0

Informational

This report has been prepared for Lynxcap to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 16 issues were identified. Leveraging a combination of Formal Verification, Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
LYA-01	Centralization Risks In LynxStakeV2.Sol	Centralization	Centralization	● Acknowledged
LYA-02	Centralized Control Of Contract Upgrade	Centralization	Centralization	● Acknowledged
LYA-03	Admin Can Fabricate Unbacked Positions And Extract Real User Deposits	Logical Issue	Major	● Resolved
LYA-04	Inactive Strike Double-Pays Previously Claimed APR Interest	Logical Issue	Major	● Resolved
LYA-05	Insufficient Funds For Withdrawals Due To Unfunded Position Creation	Design Issue	Major	● Acknowledged
LYA-06	<code>claimFull</code> Returns Full <code>stakeAmount</code> Regardless Of <code>mintedLiquidityAmount</code> , Enabling Potential Double-Claim	Logical Issue	Major	● Resolved
LYA-07	Compound Positions Stop Compounding After Interest Checkpoints	Incorrect Calculation	Medium	● Resolved
LYA-08	User Set Beneficiary Is Not Honored	Inconsistency	Medium	● Resolved
LYA-09	Legacy V1 Positions Use Current Global <code>termSeconds</code> , Enabling Retroactive Maturity Changes	Logical Issue	Medium	● Resolved
LYA-15	Interest Continues Accruing After Fixed-Term Position Maturity	Inconsistency	Medium	● Resolved

ID	Title	Category	Severity	Status
LYA-10	Missing Zero Address Validation	Volatile Code	Minor	● Resolved
LYA-11	<code>openReferralRewardPosition</code> Bypasses KYC Stake Threshold Enforcement	Logical Issue	Minor	● Acknowledged
LYA-12	Inconsistent State Update In <code>strikeInactive</code> Inflates User's Staked Amount	Logical Issue	Minor	● Resolved
LYA-13	<code>isMature</code> View Function Reverts Instead Of Returning False	Inconsistency	Minor	● Resolved
LYA-14	Unbounded Loop In <code>claimBatch</code> Causes DoS	Denial of Service	Minor	● Resolved
LYA-16	<code>transferPosition</code> Copies <code>mintedLiquidityAmount</code> Without Updating The Withdrawal Library	Logical Issue	Minor	● Acknowledged

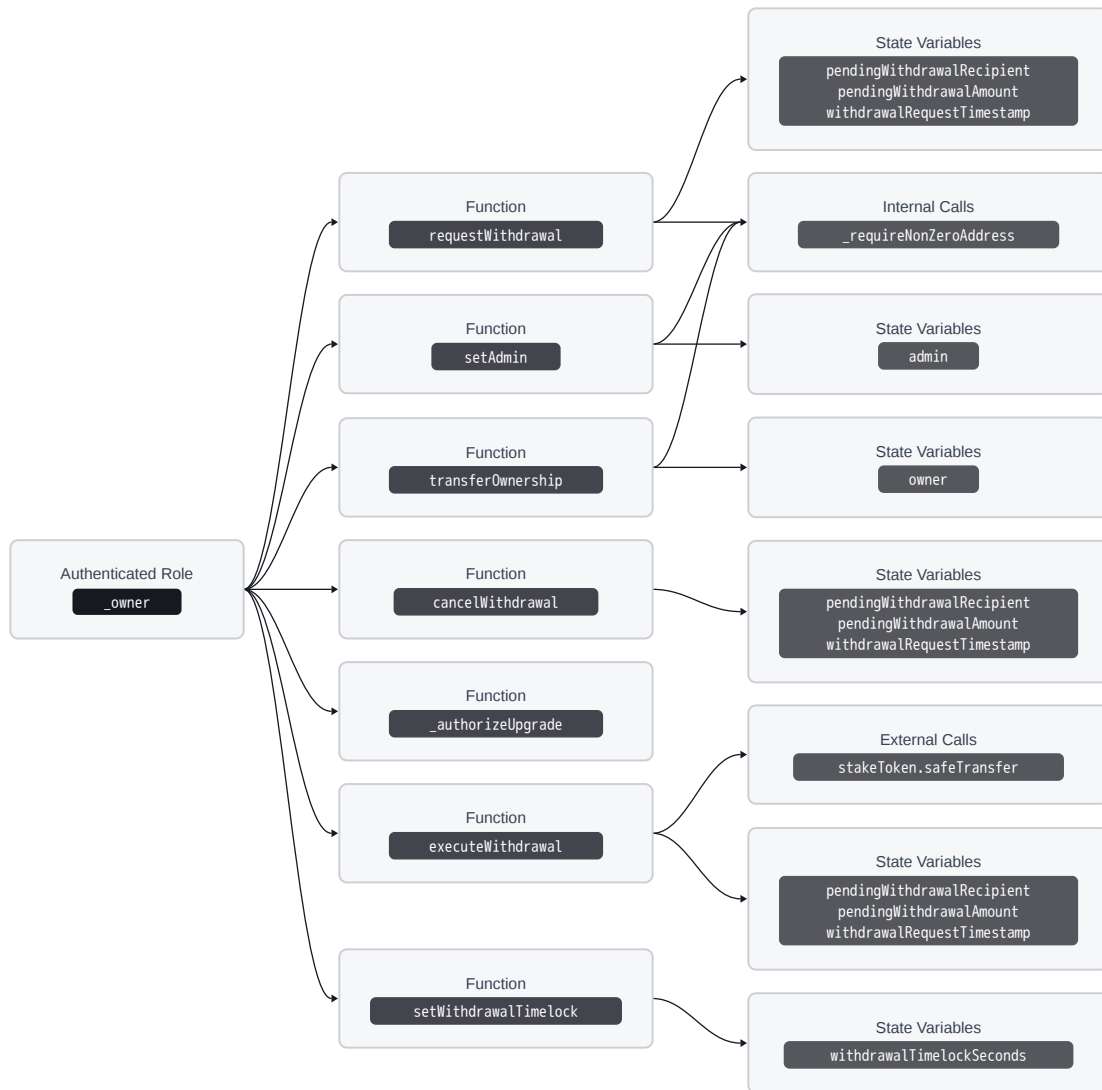
LYA-01 | Centralization Risks In LynxStakeV2.Sol

Category	Severity	Location	Status
Centralization	● Centralization		● Acknowledged

Description

In the contract `LynxStakeV2` the role `owner` has authority over the functions shown in the diagram below. Any compromise to the `owner` account may allow the hacker to take advantage of this authority and

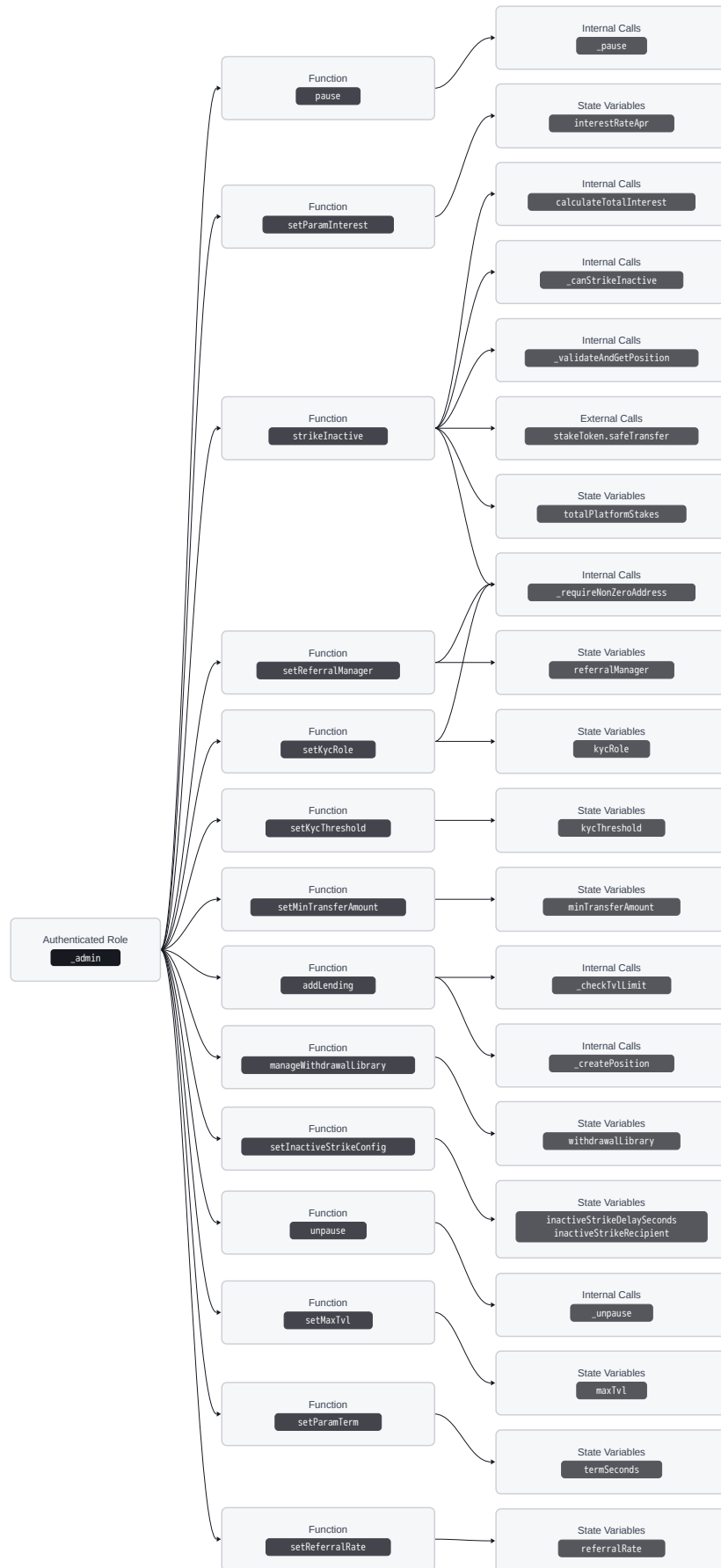
- `transferOwnership(address)`: Can transfer ownership to any address.
- `requestWithdrawal(uint256, address)`: Can initiate withdrawal of contract funds to any recipient address.
- `executeWithdrawal()`: Can execute pending withdrawals after timelock, draining contract tokens.
- `cancelWithdrawal()`: Can cancel pending withdrawals.
- `setWithdrawalTimelock(uint256)`: Can modify the withdrawal timelock duration (minimum 12 hours enforced).
- `_authorizeUpgrade(address)`: Can authorize contract upgrades via UUPS pattern, enabling arbitrary logic changes.



In the contract `LynxStakeV2` the role `admin` has authority over the functions shown in the diagram below. Any compromise to the `admin` account may allow the hacker to take advantage of this authority and

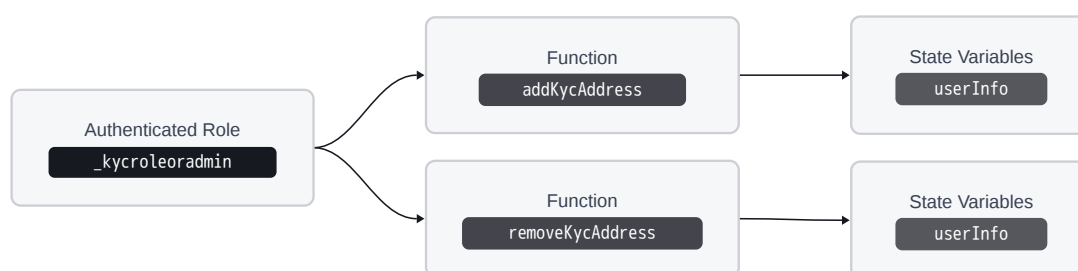
- `setKycRole(address)` : Can assign KYC role to any address.
- `setReferralManager(address)` : Can assign referral manager role to any address.
- `addKycAddress(address)` : Can KYC-verify any user, bypassing normal verification processes.
- `removeKycAddress(address)` : Can revoke KYC status from any user, blocking their ability to stake above threshold.
- `addLending(address, uint256, bool)` : Can create stake positions for any user using contract funds, potentially diluting legitimate stakers or creating artificial positions.
- `strikeInactive(address, uint256)` : Can force-close mature positions and transfer principal + interest to `inactiveStrikeRecipient`, potentially seizing user funds after delay period.
- `manageWithdrawalLibrary(address)` : Can set or disable the withdrawal library contract, affecting liquidity token minting functionality.
- `setParamInterest(uint256)` : Can modify the interest rate for new positions, impacting user returns.

- `setParamTerm(uint32)` : Can modify the term duration for new positions.
- `setKycThreshold(uint256)` : Can modify KYC threshold, potentially blocking or enabling staking for users.
- `setInactiveStrikeConfig(address, uint256)` : Can change the inactive strike recipient and delay, redirecting seized funds to any address.
- `pause()` : Can pause all state-changing operations, blocking staking, withdrawals, and claims.
- `unpause()` : Can unpause contract operations.
- `setReferralRate(uint256)` : Can modify referral rate, affecting reward distribution.
- `setMaxTvl(uint256)` : Can modify TVL limit, potentially blocking new stakes or allowing unlimited deposits.
- `setMinTransferAmount(uint256)` : Can modify minimum transfer amount for position transfers.



In the contract `LynxStakeV2` the role `kycRole` has authority over the functions shown in the diagram below. Any compromise to the `kycRole` account may allow the hacker to take advantage of this authority and

- `addKycAddress(address)` : Can KYC-verify any user, bypassing normal verification processes.
- `removeKycAddress(address)` : Can revoke KYC status from any user, blocking their ability to stake above threshold.



In the contract `LynxStakeV2` the role `referralmanager` has authority over the functions shown in the diagram below. Any compromise to the `referralmanager` account may allow the hacker to take advantage of this authority and

- `registerReferrer(address, address)` : Can register referrers for users, potentially manipulating referral reward distribution.



Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (2/3, 3/5) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;

AND

- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;

AND

- Introduction of a DAO/governance/voting module to increase transparency and user involvement.

AND

- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.

OR

- Remove the risky functionality.

I Alleviation

[Lynxcap, 04/30/2026]: The team acknowledged the issue and provided the following statement:

"LynxCap operates in the traditional finance space. Yields offered through our crypto platform are therefore generated off-chain (through our investment into RWA - notably Real Estate backed debt in central and eastern Europ). Therefore this platform cannot be fully decentralized. As the nature of our business requires us to off-ramp crypto funds into FIAT currency to be invested"

[CertiK, 09/08/2026]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

LYA-02 | Centralized Control Of Contract Upgrade

Category	Severity	Location	Status
Centralization	● Centralization	src/V2/LynxStakeV2.sol: 13	● Acknowledged

Description

In the contract `LynxStakeV2`, the role `owner` has the authority to update the implementation contract behind the proxy.

Any compromise to the `owner` account may allow a hacker to take advantage of this authority and change the implementation contract which is pointed by proxy and therefore execute potential malicious functionality in the implementation contract.

Recommendation

We recommend that the team make efforts to restrict access to the admin of the proxy contract. A strategy of combining a time-lock and a multi-signature (2/3, 3/5) wallet can be used to prevent a single point of failure due to a private key compromise. In addition, the team should be transparent and notify the community in advance whenever they plan to migrate to a new implementation contract.

Here are some feasible short-term and long-term suggestions that would mitigate the potential risk to a different level and suggestions that would permanently fully resolve the risk.

Short Term:

A combination of a time-lock and a multi signature (2/3, 3/5) wallet mitigate the risk by delaying the sensitive operation and avoiding a single point of key management failure.

- A time-lock with reasonable latency, such as 48 hours, for awareness of privileged operations;
AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to a private key compromised;
AND
- A medium/blog link for sharing the time-lock contract and multi-signers addresses information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Long Term:

A combination of a time-lock on the contract upgrade operation and a DAO for controlling the upgrade operation mitigate the contract upgrade risk by applying transparency and decentralization.

- A time-lock with reasonable latency, such as 48 hours, for community awareness of privileged operations;
AND
- Introduction of a DAO, governance, or voting module to increase decentralization, transparency, and user involvement;
AND
- A medium/blog link for sharing the time-lock contract, multi-signers addresses, and DAO information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Permanent:

Renouncing ownership of the `admin` account or removing the upgrade functionality can *fully* resolve the risk.

- Renounce the ownership and never claim back the privileged role;
OR
- Remove the risky functionality.

Note: we recommend the project team consider the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

I Alleviation

[Lynxcap, 04/30/2026]: We implemented a time lock feature for the withdrawal of funds from the smart contract.

[CertiK, 09/08/2026]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

LYA-03 | Admin Can Fabricate Unbacked Positions And Extract Real User Deposits

Category	Severity	Location	Status
Logical Issue	● Major	src/V2/LynxStakeV2.sol: 426–443	● Resolved

Description

The admin and owner together hold sufficient privileges to fabricate unbacked token obligations and settle them using legitimate user deposits. The mechanism is built from three capabilities that are each legitimate individually but create a circular extraction loop when combined:

Capability 1 — Admin creates positions without depositing tokens:

```
// addLending: transferTokens_ = false (no tokens enter the contract)
uint256 positionId = _createPosition(userAddress_, amount_, isCompound_, false);
```

The admin can call `addLending` to create arbitrarily large stake positions for any address. These positions are indistinguishable from real user stakes. They increment `totalPlatformStakes` (L1373), accrue interest (L1217–1223), and are fully claimable via `claimFull` (L676) and withdrawable via `withdraw` (L641). No tokens are transferred into the contract. There is no on-chain record distinguishing lending positions from user-deposited positions.

Capability 2 — Rewards are denominated in the principal token:

All interest is paid in the same `stakeToken` used for deposits. When a lending position matures and the beneficiary calls `claimFull` → `withdraw`, real tokens are transferred out of the contract (L1395):

```
stakeToken.safeTransfer(user_, amount_);
```

These tokens can only come from one source: deposits made by other users via `stake()`.

Capability 3 — Owner can extract contract reserves directly:

```
function requestWithdrawal(uint256 amount_, address recipient_) external onlyOwner
...
function executeWithdrawal() external onlyOwner nonReentrant {
    stakeToken.safeTransfer(recipient, amount);
}
```

The owner can withdraw arbitrary amounts from the contract after a 12-hour minimum timelock. Combined with `setWithdrawalTimelock` (L345), the timelock can be reduced to the minimum at any time.

Scenario

1. Legitimate users call `stake()`, depositing real tokens into the contract.
2. Admin calls `addLending(insider, largeAmount, false)`: creates a position for an insider address with no deposit.
3. The insider's position accrues real interest against the contract's reserve.
4. After maturity, the insider calls `claimFull()` → `withdraw()`: receiving principal + interest in real tokens funded by other users' deposits.
5. Alternatively, the owner calls `requestWithdrawal` → `executeWithdrawal` to drain reserves directly.
6. Late legitimate users who attempt to `withdraw()` find the contract insolvent. Their `safeTransfer` reverts due to insufficient funds.

Recommendation

This is a structural centralization risk inherent to the admin/owner privilege model. Mitigations include:

1. Require token backing for `addLending`:

Use `transferTokens_ = true` or enforce a solvency check:

```
function addLending(address userAddress_, uint256 amount_, bool isCompound_)
external onlyAdmin ... {
    require(
        stakeToken.balanceOf(address(this)) >= totalPlatformStakes + amount_ +
        totalAccruedInterest(),
        "Insufficient reserve backing"
    );
    _createPosition(userAddress_, amount_, isCompound_, false);
}
```

2. Multi-sig or governance for admin/owner:

Require multiple signers for privileged roles that have access to `addLending()`, `setParamInterest()`, and `requestWithdrawal()` to prevent unilateral action. Detailed Multi-sig/Timelock mitigation recommendations can be found in finding LYA-01.

Alleviation

[Lynxcap, 04/30/2026]:

"`addLending()` is a deprecated feature that was not and will not be used. This can be removed from the contract"

[Certik, 09/10/2026]: The team heeded the advice and resolved the issue in commit

[30ec7db37ae5626beb9108c90178574eeb2c4aaf](#).

LYA-04 | Inactive Strike Double-Pays Previously Claimed APR Interest

Category	Severity	Location	Status
Logical Issue	Major	src/V2/LynxStakeV2.sol: 450	Resolved

Description

`LynxStakeV2.sol` has an accounting bug in `strikeInactive()`. When closing an inactive position, it pays `position.stakeAmount + totalInterest`, where `totalInterest` comes from `calculateTotalInterest(...)` and represents all accrued interest, not just the unpaid remainder. For APR positions that previously used `claimInterest()` or `claimBatch()`, this causes already claimed interest to be paid again. The result is overpayment to `inactiveStrikeRecipient`, reserve loss, and possible failure of the inactive cleanup flow if the contract cannot fund the duplicate payment.

The main issue is in `strikeInactive()`. It calculates:

- `uint256 totalInterest = calculateTotalInterest(userAddress_, positionId_);`
- `uint256 totalAmount = position.stakeAmount + totalInterest;`

`calculateTotalInterest()` returns cumulative accrued interest as `position.lastUpdatedInterest + pending`, where `pending` is based on `_calculateAccruedInterest(position)`. It does not subtract `position.claimedInterest`. Because of that, its return value includes interest that may already have been paid earlier.

It then transfers `totalAmount` to `inactiveStrikeRecipient`. This uses gross lifetime interest for settlement.

As a result, `strikeInactive(...)` can pay interest that was already claimed earlier through `claimInterest()` or `claimBatch()`. The excess payment equals the amount already stored in `position.claimedInterest`.

Recommendation

Update `strikeInactive(...)` to settle only the unpaid interest, not the gross lifetime accrued interest. Before computing the payout, derive the claimable remainder by subtracting `position.claimedInterest` from the total accrued APR interest, and transfer only `position.stakeAmount` plus that unpaid amount to `inactiveStrikeRecipient`.

Ensure the inactive-strike path uses the same accounting model as `claimInterest()` / `claimBatch()`, so previously settled interest cannot be paid again.

Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [6f55079e3f4b0eb882bd8fed3b304825f2d21da1](#).

LYA-05 | Insufficient Funds For Withdrawals Due To Unfunded Position Creation

Category	Severity	Location	Status
Design Issue	● Major	src/V2/LynxStakeV2.sol: 426, 621, 676	● Acknowledged

Description

The `LynxStakeV2` contract's design allows for the creation of token liabilities without the corresponding deposit of assets, leading to a state of insolvency where the contract cannot honor all withdrawal obligations.

The primary token inflow occurs when users `stake()` and transfer tokens into the contract. However, there are multiple outflows and liability-creating functions that are not backed by new assets.

First, the `addLending()` function permits the admin to create new staking positions for any user. This function calls the internal `_createPosition` with the `transferTokens_` parameter set to `false`, meaning a new liability is created against the contract's existing token pool without any new tokens being deposited.

Second, the `openReferralRewardPosition()` function allows users to convert referral points into new staking positions. This function also calls `_createPosition` with `transferTokens_` set to `false`. The principal for these reward positions is sourced from the common staking pool, not a separate rewards reserve.

Finally, the interest paid to all stakers represents a liability that grows over time. The contract does not have an internal mechanism to generate yield. The interest is paid from the common pool of staked tokens, meaning that earlier stakers' rewards are paid using the principal of later stakers.

These three mechanisms ensure that the contract's total liabilities will eventually exceed its token holdings, causing withdrawal failures for later users.

Proof of Concept

Add below function to `test/V2/LynxStakeV2.t.sol`.

```
function test_PoC_UnfundedLendingDrainsContract() public {
    // This test demonstrates the insolvency vulnerability.
    // 1. A legitimate user (user1) stakes 1000 tokens.
    // 2. The admin creates an unfunded position for user2 for 1000 tokens using
    addLending.
    // 3. The contract now has 2000 tokens in liabilities but only 1000 tokens
    in assets.
    // 4. After maturation, the privileged user (user2) withdraws their 1000
    tokens first.
    // 5. The original staker (user1) is now unable to withdraw their funds as
    the contract is empty.

    uint256 stakeAmount = 1000 * 10 ** 18;

    // 1. user1 stakes 1000 tokens.
    vm.startPrank(user1);
    stakeToken.approve(address(lynxStakeV2), stakeAmount);
    lynxStakeV2.stake(stakeAmount, false);
    vm.stopPrank();

    assertEq(stakeToken.balanceOf(address(lynxStakeV2)), stakeAmount);
    assertEq(lynxStakeV2.getTotalPlatformStakes(), stakeAmount);

    // 2. Admin creates an unfunded position for user2.
    // user2 is KYC'd in setUp, so they are a valid recipient for lending and
    withdrawal.
    vm.prank(admin);
    lynxStakeV2.addLending(user2, stakeAmount, false);

    // Assert the mismatch: total stakes are 2000, but balance is only 1000.
    assertEq(stakeToken.balanceOf(address(lynxStakeV2)), stakeAmount);
    assertEq(lynxStakeV2.getTotalPlatformStakes(), stakeAmount * 2);

    // 3. Advance time past maturity.
    vm.warp(block.timestamp + termSecondsConfig + 1 days);

    // 4. The privileged user (user2) withdraws first.
    uint256 interestUser2 = lynxStakeV2.calculateTotalInterest(user2, 0);

    // user2 claims their principal + interest and withdraws it.
    vm.startPrank(user2);
    uint256 claimedBy2 = lynxStakeV2.claimFull(0);
    assertEq(claimedBy2, stakeAmount + interestUser2);
    lynxStakeV2.withdraw(stakeToken.balanceOf(address(lynxStakeV2)));
    vm.stopPrank();

    // The contract is now drained of the initial principal (from user1) plus
    the interest we minted for user2.
    // The balance should be zero.
```

```
assertEq(stakeToken.balanceOf(address(lynxStakeV2)), 0);

// 5. The original staker (user1) is now unable to withdraw.
// First, user1 claims their position. This will succeed internally.
uint256 interestUser1 = lynxStakeV2.calculateTotalInterest(user1, 0);
vm.startPrank(user1);
uint256 claimedBy1 = lynxStakeV2.claimFull(0);
assertEq(claimedBy1, stakeAmount + interestUser1);

// Now, user1 attempts to withdraw their unlocked balance.
// This is the crucial step. The withdraw call will fail because the
contract has no more tokens.
// The underlying ERC20 mock will revert with an insufficient balance error.
vm.expectRevert();
lynxStakeV2.withdrawAll();
vm.stopPrank();
}
```

```
> forge test --mt test_PoC_UnfundedLendingDrainsContract
...
Ran 1 test for test/V2/LynxStakeV2.t.sol:LynxStakeV2Test
[PASS] test_PoC_UnfundedLendingDrainsContract() (gas: 534262)
```

Recommendation

The protocol must ensure that all created liabilities are fully backed by assets. The `addLending` function, in its current form, allows the admin to print unbacked liabilities and should be removed or redesigned to use a separate, protocol-owned treasury. Similarly, referral rewards paid via `openReferralRewardPosition` must be funded from a dedicated rewards pool, not the user-staking pool. Interest payments should be funded by a sustainable, external revenue source that deposits funds into the contract to cover the accrued interest.

Alleviation

[Lynxcap, 04/29/2026]:

The `addLending()` function was initially created for an internal initial transfer of funds (without the need for our users to physically deposit funds)

But has since been decided not to use. Therefore this function is deprecated and can be removed from the smart contract.

The `openReferralRewardPosition()` is a function we intend to use and is a mechanism for us to reward people that promote our product.

The nature of our business means that we can never be truly "decentralized" and therefore will always need to actively manage the treasury.

"Interest paid to all stakers represents a liability that grows over time"

Again, as above, we identify this and recognise it. But will be actively managing the treasury from real world asset sales.

[Certik, 09/10/2026]: The team removed the `addLending()` function and updated the README and code comments to clarify that interest and referral rewards are funded through off-chain RWA activities. However, sufficient reserve backing is not enforced on-chain.

LYA-06 | `claimFull` Returns Full `stakeAmount` Regardless Of `mintedLiquidityAmount`, Enabling Potential Double-Claim

Category	Severity	Location	Status
Logical Issue	● Major	src/V2/LynxStakeV2.sol: 685	● Resolved

Description

When a user mints liquidity tokens via `mintLiquidityTokens` (L757), the `mintedLiquidityAmount` field is incremented (L1327), and this amount reduces the `effectivePrincipal` used for interest calculations (L1218):

```
// L1218:  
uint256 effectivePrincipal = position.stakeAmount - position.mintedLiquidityAmount;
```

However, when `claimFull` is called, the full `stakeAmount` is returned to the user regardless of how much liquidity has been minted against the position:

```
// L685:  
uint256 totalAmount = position.stakeAmount + interestClaimed;
```

The staking contract itself does not invoke any callback on the `IWithdrawalLibrary` during `claimFull` to burn, invalidate, or otherwise reconcile the previously minted liquidity tokens.

Impact:

Depending on the `IWithdrawalLibrary` implementation (which is external and out of scope), this creates a potential double-claim scenario. The user receives the full principal back via `claimFull` while simultaneously holding liquidity tokens that represent a claim against that same principal. If those liquidity tokens are redeemable for underlying value, the user extracts more value than they deposited. Even if the withdrawal library implementation prevents this, the staking contract has a fundamental accounting gap: It treats the full `stakeAmount` as returnable even when a portion has been withdrawn/minted as liquidity.

Scenario

1. User stakes 1000 tokens. `position.stakeAmount = 1000`.
2. User calls `mintLiquidityTokens(positionId, 1000)`: receives 1000 liquidity tokens.
`position.mintedLiquidityAmount = 1000`.
3. Interest accrues on `effectivePrincipal = 0` (1000 - 1000), so no additional interest.
4. Position matures. User calls `claimFull(positionId)`.
5. `totalAmount = 1000 + 0 = 1000`. User's `unlockedBalance` increases by 1000.
6. User withdraws 1000 tokens from unlocked balance.

7. User still holds 1000 liquidity tokens from step 2. If liquidity tokens are redeemable, user has extracted 2000 tokens of value from a 1000-token stake.

I Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test, console} from "forge-std/Test.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {LynxStakeV2} from "../src/V2/LynxStakeV2.sol";
import {IWithdrawalLibrary} from "../src/interfaces/IWithdrawalLibrary.sol";

// ===== Mock Contracts =====

contract MockERC20 is ERC20 {
    constructor() ERC20("Mock Token", "MTK") {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

contract MockWithdrawalLibrary is IWithdrawalLibrary {
    function mintLiquidityTokens(uint256, uint256, address) external pure returns
(bool) {
        return true;
    }
}

// ===== PoC Test =====

contract ClaimFullIgnoresMintedLiquidity is Test {
    LynxStakeV2 public staking;
    MockERC20 public token;
    MockWithdrawalLibrary public withdrawalLib;

    address public owner = makeAddr("owner");
    address public admin = makeAddr("admin");
    address public user = makeAddr("user");
    address public inactiveRecipient = makeAddr("inactiveRecipient");

    uint256 constant STAKE_AMOUNT = 1000e18;
    uint256 constant INTEREST_RATE = 500; // 5% APR in basis points
    uint32 constant TERM_YEARS_SCALED = 100; // 1 year
    uint256 constant REFERRAL_RATE = 100; // 1%
    uint256 constant MAX_TVL = 1_000_000e18;

    function setUp() public {
        // Deploy mocks
        token = new MockERC20();
        withdrawalLib = new MockWithdrawalLibrary();
    }
}
```

```
// Deploy LynxStakeV2 behind an ERC1967 proxy
vm.startPrank(owner);

LynxStakeV2 implementation = new LynxStakeV2();
bytes memory initData = abi.encodeCall(
    LynxStakeV2.initialize,
    (
        address(token),
        INTEREST_RATE,
        TERM_YEARS_SCALED,
        admin,
        inactiveRecipient,
        100, // inactiveStrikeDelayYearsScaled = 1 year
        REFERRAL_RATE,
        MAX_TVL
    )
);
ERC1967Proxy proxy = new ERC1967Proxy(address(implementation), initData);
staking = LynxStakeV2(address(proxy));

vm.stopPrank();

// Admin sets the withdrawal library
vm.prank(admin);
staking.manageWithdrawalLibrary(address(withdrawalLib));

// Admin KYC-verifies the user
vm.prank(admin);
staking.addKycAddress(user);

// Mint tokens to user and to the staking contract (for interest payouts)
token.mint(user, STAKE_AMOUNT);
token.mint(address(staking), 10_000e18); // extra for interest
}

function claimFullReturnsFullPrincipalDespiteMintedLiquidity() public {
    // --- Step 1: User stakes 1000 tokens (simple APR, non-compound) ---
    vm.startPrank(user);
    token.approve(address(staking), STAKE_AMOUNT);
    staking.stake(STAKE_AMOUNT, false);
    vm.stopPrank();

    uint256 positionId = 0;

    // Verify position created correctly
    (uint256 stakeAmount, bool isActive,) = staking.getStakePosition(user,
positionId);
    assertEq(stakeAmount, STAKE_AMOUNT, "Stake amount mismatch");
    assertTrue(isActive, "Position should be active");
}
```

```
// --- Step 2: User mints 100% of stake as liquidity tokens ---
vm.prank(user);
staking.mintLiquidityTokens(positionId, STAKE_AMOUNT);

// Verify mintedLiquidityAmount == stakeAmount
uint256 mintedLiquidity = staking.getMintedLiquidityAmount(user,
positionId);
assertEq(mintedLiquidity, STAKE_AMOUNT, "Minted liquidity should equal full
stake");

// --- Step 3: Verify effective principal is 0 ---
// After minting 100% liquidity, effectivePrincipal = stakeAmount -
mintedLiquidityAmount = 0
// This means interest should accrue on 0 principal
console.log("--- Position State After Minting Liquidity ---");
console.log("stakeAmount:          ", stakeAmount);
console.log("mintedLiquidityAmount: ", mintedLiquidity);
console.log("effectivePrincipal:      ", stakeAmount - mintedLiquidity);

// --- Step 4: Warp to maturity (1 year + 1 second) ---
uint256 termSeconds = uint256(TERM_YEARS_SCALED) * 365 * 24 * 3600 / 100;
vm.warp(block.timestamp + termSeconds + 1);

// --- Step 5: User calls claimFull ---
uint256 userBalanceBefore = token.balanceOf(user);

vm.prank(user);
uint256 totalClaimed = staking.claimFull(positionId);

uint256 userBalanceAfter = token.balanceOf(user);

console.log("");
console.log("--- Claim Results ---");
console.log("totalClaimed (returned by claimFull):", totalClaimed);
console.log("User balance before withdraw:      ", userBalanceBefore);

// The user's unlocked balance now contains the full principal
uint256 unlockedBalance = staking.getUnlockedBalance(user);
console.log("unlockedBalance after claimFull:    ", unlockedBalance);

// Withdraw to realize the tokens
vm.prank(user);
staking.withdrawAll();

uint256 userBalanceFinal = token.balanceOf(user);
console.log("User balance after withdrawAll:     ", userBalanceFinal);

// The critical assertion: user received back >= full stakeAmount
```

```

    // despite having already minted 100% as liquidity tokens
    assertGe(totalClaimed, STAKE_AMOUNT, "BUG CONFIRMED: claimFull returned full
principal despite 100% liquidity minted");

    console.log("");
    console.log("User staked:           ", STAKE_AMOUNT);
    console.log("User minted as liquidity: ", STAKE_AMOUNT);
    console.log("User claimed back:         ", totalClaimed);
    console.log("Double value extraction:   User extracted value TWICE: once via
liquidity mint, once via claimFull");
  }
}

```

PoC output:

```

[PASS] claimFullReturnsFullPrincipalDespiteMintedLiquidity() (gas: 313008)
  Logs:
    --- Position State After Minting Liquidity ---
    stakeAmount:           100000000000000000000
    mintedLiquidityAmount: 100000000000000000000
    effectivePrincipal:     0

    --- Claim Results ---
    totalClaimed (returned by claimFull): 100000000000000000000
    User balance before withdraw:         0
    unlockedBalance after claimFull:       100000000000000000000
    User balance after withdrawAll:        100000000000000000000

    User staked:           100000000000000000000
    User minted as liquidity: 100000000000000000000
    User claimed back:      100000000000000000000
    Double value extraction: User extracted value TWICE: once via liquidity
mint, once via claimFull

    Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.49ms (431.30µs
CPU time)

    Ran 1 test suite in 12.96ms (1.49ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)

```

Recommendation

It's recommended that the team add a callback to the `IWithdrawalLibrary` in `claimFull` that burns or invalidates the minted liquidity tokens before returning the full principal:

```

if (position.mintedLiquidityAmount > 0) {
  IWithdrawalLibrary(withdrawalLibrary).onPositionClosed(msg.sender, positionId_);
}

```

I Alleviation

[Lynxcap, 04/29/2026]:

The minting of liquidity tokens is not a feature that is available to users at the moment (at least in the UI). This is a placeholder for future development.

For now users only have the possibility to "stake..... wait the full term to maturity... then redeem principle in full".

If this can be misused, then this can simply be removed from the contract for now and implemented correctly in the next phase of development.

[Lynxcap, 09/10/2026]: The team heeded the advice and removed the related minting functions in commit [30ec7db37ae5626beb9108c90178574eeb2c4aaf](#).

LYA-07 | Compound Positions Stop Compounding After Interest Checkpoints

Category	Severity	Location	Status
Incorrect Calculation	● Medium	src/V2/LynxStakeV2.sol: 1217	● Resolved

Description

In `LynxStakeV2.sol`, compound positions (`StakePosition.isCompound == true`) do not keep a consistent compounding base after an interest checkpoint. `_updatePositionInterest()` stores accrued interest in `StakePosition.lastUpdatedInterest` and resets `StakePosition.lastUpdatedTimestamp`, but later `_calculateAccruedInterest()` still computes APY from `position.lastUpdatedTimestamp`. As a result, previously accrued interest stops earning interest after the first checkpoint. This causes compound positions to be underpaid in functions such as `claimFull()`, `calculateTotalInterest()`, and `strikeInactive()`.

For compound positions, `calculateApy(effectivePrincipal, position.interestRate, block.timestamp, position.lastUpdatedTimestamp)`. This means future APY is always calculated from the raw principal like base `position.stakeAmount - position.mintedLiquidityAmount`. It does not include previously checkpointed interest stored in `StakePosition.lastUpdatedInterest`:

```
function _calculateAccruedInterest(StakePosition storage position) internal view
returns (uint256) {
    uint256 effectivePrincipal = position.stakeAmount -
    position.mintedLiquidityAmount;

    return position.isCompound
        ? calculateApy(effectivePrincipal, position.interestRate, block.timestamp,
            position.lastUpdatedTimestamp)
        : calculateApr(effectivePrincipal, position.interestRate, block.timestamp,
            position.lastUpdatedTimestamp);
}
```

Both `transferPosition()` and `mintLiquidityTokens()` trigger `_updatePositionInterest()` during their execution. This step resets the `StakePosition.lastUpdatedTimestamp` to the current block time. This behavior conflicts with the intended economics for `StakePosition.isCompound == true`. The contract explicitly blocks `claimInterest` on compound positions through `CannotClaimInterestOnCompoundPositions()`, which suggests interest should remain inside the position to preserve compounding. However, any path that calls `_updatePositionInterest(...)` before maturity can still break that compounding chain.

As a result, compound positions that are touched before maturity are systematically underpaid compared to true APY compounding. After the first checkpoint, stored interest in `StakePosition.lastUpdatedInterest` no longer earns interest.

Recommendation

For positions with `StakePosition.isCompound == true`, ensure each interest checkpoint preserves the compounding base.

Align the accounting so that previously accrued interest in `StakePosition.lastUpdatedInterest` either becomes part of the principal used for future APY calculations or is otherwise included in the effective base for all later accrual periods. Apply this consistently across every path that checkpoints interest, including transfers, liquidity minting, claims, inactivity handling, and view functions.

■ Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [6f55079e3f4b0eb882bd8fed3b304825f2d21da1](#).

LYA-08 | User Set Beneficiary Is Not Honored

Category	Severity	Location	Status
Inconsistency	● Medium	src/V2/LynxStakeV2.sol: 744	● Resolved

Description

The `setBeneficiaryAddress` function allows a user to specify a `userBeneficiary` address, creating the expectation that this address will receive funds from claims and withdrawals. However, the implementation of the `withdraw`, `withdrawAll`, `claimInterest`, `claimFull`, and `claimBatch` functions disregard this setting. All these functions result in funds being transferred to the `msg.sender` rather than the designated `userBeneficiary`.

As a result, a user who sets a beneficiary address for their account will find that it is never effective, and funds are instead returned directly to their own address. This defeats the purpose of the beneficiary feature and can lead to user confusion and funds not being routed as intended.

Recommendation

To resolve this issue, the contract's withdrawal and claiming logic should be modified to prioritize the `userBeneficiary` address if one has been set. Specifically, in the `_executeWithdrawal` function, instead of transferring tokens to the `user_`, the logic should first check if `userInfo[user_].userBeneficiary` is a non-zero address. If it is, the transfer should be directed to the beneficiary address. Otherwise, it can default to the `user_` address. This ensures that the user-specified beneficiary receives the funds as expected, making the feature functional.

Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [6f55079e3f4b0eb882bd8fed3b304825f2d21da1](#).

LYA-09 | Legacy V1 Positions Use Current Global `termSeconds` , Enabling Retroactive Maturity Changes

Category	Severity	Location	Status
Logical Issue	● Medium	src/V2/LynxStakeV2.sol: 1414~1419	● Resolved

Description

Positions created before the term snapshot feature was introduced have `termSnapshotInitialized == false` . The `_getTermSecondsForPosition` helper falls back to the current global `termSeconds` for these positions:

```
function _getTermSecondsForPosition(StakePosition storage position) internal view
returns (uint256) {
    if (!position.termSnapshotInitialized) {
        return termSeconds; // L1416: Returns current global value, not the value at
position creation
    }
    return position.termSecondsSnapshot;
}
```

This means any admin change to `termSeconds` via `setParamTerm` (L502) retroactively affects the maturity of all V1 positions.

If the admin reduces `termSeconds` , V1 positions may become prematurely mature, allowing early withdrawal of locked funds or making them eligible for `strikeInactive` . If the admin increases `termSeconds` , V1 users' lockup periods are extended beyond what they agreed to when staking. V1 users have no certainty about when their positions will mature.

Recommendation

Consider implementing a one-time migration function that snapshots the current `termSeconds` for all V1 positions:

```
function migrateV1PositionTerm(address user_, uint256 positionId_) external
onlyAdmin {
    StakePosition storage position = userInfo[user_].userPositions[positionId_];
    require(!position.termSnapshotInitialized, "Already migrated");
    position.termSnapshotInitialized = true;
    position.termSecondsSnapshot = termSeconds;
}
```

Alternatively, document this as a known limitation and ensure `termSeconds` is not changed while V1 positions remain active.

Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [fa7a1070b36d887bed9d015d3e4e6eeb7f40dd54](#).

LYA-15 | Interest Continues Accruing After Fixed-Term Position Maturity

Category	Severity	Location	Status
Inconsistency	Medium	src/V2/LynxStakeV2.sol: 1217	Resolved

Description

The function `_calculateAccruedInterest()` calculates pending interest from `position.lastUpdatedTimestamp` to `block.timestamp` on `position.stakeAmount - position.mintedLiquidityAmount`.

The accrual window is never capped at the position's maturity. Maturity is derived from `position.initiationTimestamp + _getTermSecondsForPosition(position)`, which for normal snapshot-based positions is effectively `position.initiationTimestamp + position.termSecondsSnapshot`. However, neither `_calculateAccruedInterest(...)` nor `calculateApr(...)` bounds the end timestamp to that maturity. `calculateApr(...)` explicitly uses `currentTimestamp - lastUpdatedTimestamp` as `secondsPassed`, so any time after maturity is still counted.

The contract already has maturity checks, but they are used only as eligibility gates. In particular:

- `_isMature(StakePosition storage position)` checks whether `block.timestamp >= position.initiationTimestamp + _getTermSecondsForPosition(position)`
- `claimFull(uint256 positionId_)` only checks maturity and reverts with `PositionIsNotMature()` if the position is not mature
- `_canStrikeInactive(StakePosition storage position)` uses maturity plus `inactiveStrikeDelaySeconds` only to decide when `strikeInactive(...)` is allowed

These checks do not stop interest accrual after maturity.

As a result, fixed-term positions can receive more interest than the agreed term allows. The longer a mature position remains unsettled, the larger the excess payout becomes.

Recommendation

It's recommended that interest accrual for fixed-term positions is capped at the maturity timestamp, so that no additional interest accumulates beyond the agreed-upon term. Functions responsible for calculating accrued interest, such as `_calculateAccruedInterest()` and `calculateApr()`, should use the lesser of `block.timestamp` and the position's maturity timestamp when determining the interest accrual window. This ensures that payouts remain consistent with the expected terms and prevents users from earning unintended extra interest after their positions mature.

Alleviation

[Lynxcap, 05/04/2026]:

This is not the intended reward logic. Positions should stop accruing interest at the maturity date and nothing past that.

[CertiK, 09/10/2026]:

The team introduced `_accrualEndTimeStamp()` to cap interest accrual at the position's maturity timestamp in [commit 30ec7db37ae5626beb9108c90178574eeb2c4aaf](#). The shared interest calculation now uses this capped timestamp, preventing additional interest from accruing after maturity.

Legacy term mutability remains tracked under LYA-09, while the treatment of interest accrued before the upgrade requires confirmation as part of the migration process.

LYA-10 | Missing Zero Address Validation

Category	Severity	Location	Status
Volatile Code	Minor	src/V2/LynxStakeV2.sol: 529	Resolved

Description

The address input for `setInactiveStrikeConfig` is missing a check that it is not `address(0)`.

Recommendation

We recommend adding a check the passed-in address is not `address(0)` to prevent unexpected errors.

Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [6f55079e3f4b0eb882bd8fed3b304825f2d21da1](#).

LYA-11 | `openReferralRewardPosition` Bypasses KYC Stake Threshold Enforcement

Category	Severity	Location	Status
Logical Issue	Minor	src/V2/LynxStakeV2.sol: 621	Acknowledged

Description

The `openReferralRewardPosition()` function creates a stake position for the caller without applying the KYC threshold check enforced on all other user-initiated staking paths. A non-KYC-verified user can accumulate referral points and convert them into positions that push their `totalAmountStaked` above `kycThreshold`, circumventing the contract's admission control.

The intended KYC gate exists on the normal deposit path. In `stake()`, the contract uses `requiresKyc()`, which flows into `_requiresKyc()` and then `canStakeWithoutKyc()`. That logic allows staking only when `userInfo[user_].totalAmountStaked + amount_ <= kycThreshold` or `userInfo[user_].kycVerified` is true. Otherwise it reverts with `AmountExceedsKycThreshold()`.

The bypass becomes effective because `openReferralRewardPosition()` calls `_createPosition()`. The helper `_createPosition()` unconditionally:

- creates an active `StakePosition`,
- increments `userInfo[user_].totalAmountStaked` by `amount_`,
- increments `totalPlatformStakes` by `amount_`.

This means `openReferralRewardPosition()` is a user-accessible alternate staking path that increases normal stake exposure without passing through the contract's only threshold-based KYC admission check.

As a result, a non-KYC user can create and hold stake exposure that `stake()` should have rejected under `kycThreshold`.

Recommendation

Apply the same per-user KYC enforcement to `openReferralRewardPosition()` as is applied to `stake()`. Before converting `referralPoints[msg.sender]` into an active position, validate the reward amount against the user's current `totalAmountStaked` and `kycVerified` status using the existing KYC admission logic, and revert when the resulting exposure would exceed `kycThreshold`.

Alleviation

[Lynxcap, 09/08/2026]:

The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

LYA-12 | Inconsistent State Update In `strikeInactive` Inflates User's Staked Amount

Category	Severity	Location	Status
Logical Issue	Minor	src/V2/LynxStakeV2.sol: 450	Resolved

Description

The `strikeInactive` function, which is used by an administrator to close a user's expired staking position, fails to correctly update the user's total staked amount. While it does decrease the `totalPlatformStakes`, it does not decrease the `userInfo[userAddress_].totalAmountStaked`. This is in contrast to the `claimFull` function, which correctly reduces both values when a user closes a position.

Because `totalAmountStaked` is used in the `canStakeWithoutKyc` function to determine if a user can stake more without KYC, this inconsistency can cause a user's staked amount to be permanently inflated after a position is struck.

As a result, users who have had positions struck inactive may be prevented from making new stakes even if their actual staked amount is below the KYC threshold.

Recommendation

The `strikeInactive` function should be updated to decrease the user's `totalAmountStaked` by the `position.stakeAmount`, just as the `claimFull` function does. Adding the line `userInfo[userAddress_].totalAmountStaked -= position.stakeAmount;` to the `strikeInactive` function will ensure that the user's total staked amount is accurately reflected after a position is closed by an admin. This will align the behavior of `strikeInactive` with `claimFull` and prevent the user's `totalAmountStaked` from being permanently inflated, ensuring the `canStakeWithoutKyc` check functions as intended.

Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [6f55079e3f4b0eb882bd8fed3b304825f2d21da1](#).

LYA-13 | `isMature` View Function Reverts Instead Of Returning False

Category	Severity	Location	Status
Inconsistency	Minor	src/V2/LynxStakeV2.sol: 936	Resolved

Description

The `isMature()` function calls `_validateAndGetPosition()` which reverts if the position is inactive, instead of returning `false`:

```
function isMature(uint256 positionId_) external view returns (bool) {
    StakePosition storage position = _validateAndGetPosition(msg.sender,
positionId_);
    return _isMature(position);
}

function _validateAndGetPosition(...) internal view returns (StakePosition storage
position) {
    position = _requirePositionExists(user_, positionId_);
    if (!position.isActive) revert PositionIsNotActive(); // Reverts instead of
returning false
    return position;
}
```

Recommendation

Return false for inactive positions instead of reverting:

```
function isMature(uint256 positionId_) external view returns (bool) {
    StakePosition storage position = _requirePositionExists(msg.sender,
positionId_);
    if (!position.isActive) return false;
    return _isMature(position);
}
```

Alleviation

[Lynxcap, 09/08/2026]:

The team heeded the advice and resolved the issue in commit [6f55079e3f4b0eb882bd8fed3b304825f2d21da1](#).

LYA-14 | Unbounded Loop In `claimBatch` Causes DoS

Category	Severity	Location	Status
Denial of Service	● Minor	src/V2/LynxStakeV2.sol: 709	● Resolved

Description

The `claimBatch()` function iterates through all user positions without pagination or limits. A user with many positions will hit block gas limits, making these functions unusable.

```
function claimBatch() external nonReentrant whenNotPaused returns (uint256) {
    UserInfo storage user = userInfo[msg.sender];
    uint256 positionCount = user.positionCount; // Could be very large
    // ...
    for (uint256 i = 0; i < positionCount;) { // Unbounded loop
        StakePosition storage position = user.userPositions[i];
        // ... processing each position with storage reads/writes
    }
}
```

Recommendation

Consider implementing pagination for batch operations:

```
function claimBatch(uint256 startIndex, uint256 count) external nonReentrant
whenNotPaused returns (uint256) {
    UserInfo storage user = userInfo[msg.sender];
    uint256 endIndex = Math.min(startIndex + count, user.positionCount);
    // Process positions from startIndex to endIndex
}
```

Alleviation

[Lynxcap, 06/28/2026]: The team heeded the advice and resolved the issue by implementing paginated `claimBatch` in commit [12c5caea9685](#)

LYA-16 | `transferPosition` Copies `mintedLiquidityAmount` Without Updating The Withdrawal Library

Category	Severity	Location	Status
Logical Issue	● Minor	src/V2/LynxStakeV2.sol: 776-812	● Acknowledged

Description

When a position is transferred, `mintedLiquidityAmount` is copied to the recipient's new position:

```
// L795: Liquidity amount copied to new position
newPos.mintedLiquidityAmount = position.mintedLiquidityAmount;
```

However, the `IWithdrawalLibrary` is never notified of the ownership change. The withdrawal library is out of this audit scope. It likely tracks liquidity tokens by the original user address and position ID. After the transfer, the sender's position is deactivated (L805), but the library may still reference it. The recipient inherits a reduced effective principal for interest calculation but has no ownership relationship with the liquidity tokens tracked by the external library.

Recommendation

We recommend the team clarify whether positions with outstanding minted liquidity (`mintedLiquidityAmount > 0`) are intended to be transferable. If the `IWithdrawalLibrary` tracks liquidity ownership by user address and position ID, then transferring such positions creates a state desynchronization that the staking contract cannot reconcile on its own.

Alleviation

[Lynxcap, 09/08/2026]: The team acknowledged the issue and decided not to make changes in the current engagement.

APPENDIX | LYNXCAP - AUDIT

Finding Categories

Categories	Description
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.

Lynxcap - Audit Security Assessment | CertiK Assessed on Sept 10th, 2026 | Copyright © CertiK

