ZEALYNX

Web3 Security & Smart Contract Development

Nexalo

Smart Contract Audit

December 22, 2025

Zealynx
contact@zealynx.io

Carlos (Bloqarl)	@TheBlockChainer
Stephen	@derastephh
Springfield Yonga	@0xspryon
Strapontin	@0xStrapontin

Contents

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Badger DAO, Ample protocol, Lido, Inverter, Matchain, and Golden Grid.

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. 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 Solidity smart contract Audit of the Nexalo protocol was conducted by Zealynx, with a focus on the potential security vulnerabilities.

We performed the security evaluation based on the agreed scope during 2 weeks, following our systematic approach and methodology. Based on the scope and our performed activities, our security assessment revealed 3 Critical, 5 High, 10 Medium and 5 Low.

3.2 About Nexalo

Nexalo is an autonomous on-chain raffle protocol built on blockchain technology that combines decentralized lottery mechanics with tokenized participation rights. The system features six distinct raffle products with varying price points and prize pools that users can participate in by purchasing numbered tickets. Each ticket purchase is mathematically converted into a proportional NXL token allocation using predetermined ratios across three product tiers.

The protocol utilizes Chainlink VRF (Verifiable Random Function) to ensure provably fair lottery draws. Winners are determined by calculating overlaps between randomly drawn numbers and player-owned tickets using efficient bitwise operations. The system implements sophisticated fund distribution mechanisms for multi-stakeholder reward allocation and includes advanced features like linear vesting schedules, proportional staking rewards, and an annual buyback-and-burn mechanism to create deflationary pressure.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes ~1,150 nSLOC - normalized source lines of code. The codebase consists of six Solidity smart contracts: NexumManager implementing raffle mechanics and Chainlink VRF callbacks, ReferralNetwork and AmbassadorRegistry managing multi-level commission distributions, NXLToken handling vesting schedules and reward allocations, NexaloStaking calculating proportional staking rewards, and TreasuryBTC managing BTC-backed treasury operations with precision arithmetic for fund allocations across six distinct ecosystem channels.

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**, 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 4-day engagement, Zealynx team conducted a security review of the Nexalo Smart Contract implementation.

The audit focused on identifying vulnerabilities, logic flaws, and implementation-level issues that could affect the lottery protocol's random number generation, pixel-to-bitmap conversions, financial distribution mechanisms, and overall system fairness.

A total of 22 issues were identified and categorized as follows:
- 3 Critical severity
- 5 High severity
- 10 Medium severity
- 4 Low severity

The initial commit hash audited is:
- da7c925bec3b3927aa3c88738a1f1b7375d1c887

The key findings include:

- **Product deactivation causing fund lockup**: The protocol accepts payment and assigns tickets before attempting NXL distribution. When NXL rewards are exhausted, the product deactivates mid-transaction, leaving rounds incomplete with insufficient tickets to trigger winner selection. All payments become permanently trapped with no recovery mechanism.
- **Multiple reward claims with same tokens**: The claimRewards mechanism uses live NXL balances instead of snapshot balances. Users can transfer tokens between addresses and claim the same snapshot rewards multiple times, draining the TreasuryBTC contract.
- **Ticket ownership overwrite vulnerability**: The buyTickets function assigns tickets sequentially without verifying existing ownership. Attackers can overwrite tickets previously purchased via buySpecificTickets, stealing ownership and claiming prizes intended for legitimate buyers.
- **Undistributed referral commissions locked**: When buyers lack complete referral chains, corresponding commission percentages remain locked in the ReferralNetwork contract. These undistributed funds accumulate with no recovery mechanism.

Summary of Findings :

Vulnerability	Severity	Status
[C-01] Private key in plain sight in .env	Critical	Fixed
[C-02] NXL distribution failure during ticket purchase leads to permanent fund lockup and broken lottery rounds	Critical	Fixed
[C-03] TreasuryBTC::claimRewards can be manipulated due to balanceOf()	Critical	Not Fixed
[H-01] Ticket ownership can be overwritten due to incorrect logic in NexumManager::buyTickets	High	Fixed
[H-02] Flawed reward distribution mechanism causes pool depletion and DOS in NexaloStaking	High	Fixed
[H-03] Raffles may not distribute all funds, resulting in locked stablecoins during distribution	High	Fixed
[H-04] Idle NXL in the token contracts result in diluted rewards for users in TreasuryBTC	High	Not Fixed
[H-05] Broken ticket index continuity leads to invalid winner selection and product.maxTickets invariant break	High	Partially Fixed
[M-01] Incorrect access control in TreasuryBTC::withdrawForStaking	Medium	Not Fixed
[M-02] Denial of service in AmbassadorRegistry::distributeFunds due to usdc/usdt blacklist behaviour	Medium	Fixed

Vulnerability	Severity	Status
[M-03] fulfillRandomWords may revert, causing loss of funds and DoS in rewards distribution	Medium	Fixed
[M-04] Funds available for distribution are burned if a user paid for more rewards, resulting in a loss of rewards.	Medium	Fixed
[M-05] Incomplete accounting in receiveFunds() leads to permanent denial of service after reward claims	Medium	Fixed
[M-06] Missing VRF Failure Handling Leads to Permanent Round Lock and Fund Freezing	Medium	Fixed
[M-07] Immutable audit funds address and missing approval mechanism leads to unreachable cleanup logic and mixed fund accounting	Medium	Fixed
[M-08] Broken token burn logic prevents undistributed reward cleanup	Medium	Not Fixed
[M-09] Insufficient available rewards will result in locked staked funds for users	Medium	Fixed
[M-10] Distribution of NXL failing during round settlement will not deactivate the product, resulting in a new raffle with new rewards available	Medium	Fixed
[L-01] Unsafe ERC20 transfer operations allow silent failures and incompatibility with non-standard tokens	Low	Fixed
[L-02] Incorrect Emergency Withdrawal Implementation Allows Extraction of Active Round Funds	Low	Not Fixed
[L-03] Uninitialized lastWithdrawalTime allows immediate first withdrawal bypassing 30-day timelock	Low	Not Fixed
[L-04] Raffle winner can also win instant rewards, which will give them more than 50% of the expected allocation	Low	Fixed

7. Audit Findings

7.1 Critical Severity Findings

[C-01] Private key in plain sight in .env

Affected files

.env#L2

Description

Sensitive information, including API keys and private keys, was found stored directly in environment variables. Such files (e.g., .env) must never be committed to version control as this exposes critical secrets to potential compromise.

Impact

A malicious actor could retrieve the exposed private keys and use them to steal associated funds or access protected services.

Recommendations

All keys (both private and API) should be rotated immediately to invalidate any potentially exposed credentials. Additionally, include the .env file in .gitignore to prevent it from being pushed to the repository in the future.actor

Nexalo:

Fixed

Zealynx:

Private keys removed

[C-02] NXL distribution failure during ticket purchase leads to permanent fund lockup and broken lottery rounds

Affected files

NexumManager.sol#L245-L290

NexumManager.sol#L292-L323

Description

The buyTickets() and buySpecificTickets() functions accept payment and assign tickets before attempting to distribute NXL rewards. When NXL rewards are exhausted, the _distributeNXL() function catches the failure and calls _handleNXLExhaustion(), which immediately deactivates the product. This creates a critical state where:

1. User payment has already been collected
2. Tickets have already been assigned
3. Product becomes deactivated mid-transaction
4. Round remains incomplete with insufficient tickets to trigger winner selection

Vulnerable Scenario:

```
// Current flow in buyTickets()
function buyTickets(...) external validProduct(productId) {
    // Step 1: Product is active, validation passes ✅

    // Step 2: Payment collected ✅
    stablecoin.transferFrom(msg.sender, address(this), totalPrice);

    // Step 3: Tickets assigned ✅
    ticketOwner[productId][roundId][ticketNumber] = msg.sender;
    round.ticketsSold++;

    // Step 4: NXL distribution fails ❌
    _distributeNXL(msg.sender, nxlAmount, productId);
        // -> catch block triggers _handleNXLExhaustion(productId)
        // -> products[productId].active = false;  💀

    // Product now deactivated but transaction continues
}
```

1. NXL rewards pool becomes exhausted or nearly exhausted
2. First buyer after exhaustion calls buyTickets() for FLASH product (needs 1000 tickets)
3. User pays 1 USDT, receives ticket #0
4. NXL distribution fails → product deactivated
5. Round now shows: 1 ticket sold / 1000 needed
6. All subsequent buyers revert due to validProduct modifier: require(products[productId].active, "Product inactive")
7. Round can never reach round.ticketsSold >= product.maxTickets
8. VRF randomness is never requested
9. Winner is never selected
10. All funds collected in this round are permanently locked in the contract

Impact

This vulnerability results in permanent fund lockup and complete system failure once NXL rewards are depleted. When the reward pool is exhausted, the next ticket purchase triggers a critical sequence: the contract accepts payment and assigns tickets, then fails during NXL distribution and immediately deactivates the product mid-transaction. The round is left in an irrecoverable state with tickets sold but insufficient capacity to ever trigger winner selection through VRF.

All stablecoin payments collected in the affected round become permanently trapped in the NexumManager contract. The contract provides no refund mechanism for incomplete rounds and no administrative function exists to manually resolve or complete stuck rounds, making fund recovery impossible.

The first victim of NXL exhaustion pays full price for their tickets but receives no NXL rewards, with no clear error message indicating the failure. All subsequent purchase attempts revert immediately at the validProduct modifier, completely blocking further participation in that lottery product.

Recommendations

Replace the current try/catch logic in _distributeNXL():

```
// Current broken implementation
function _distributeNXL(address recipient, uint256 amount, uint256 productId) private {
    if (amount == 0) return;
    try nxlToken.distributeReward(recipient, amount) {
    } catch {
        _handleNXLExhaustion(productId);  // Wrong assumption
    }
}
```

With explicit state validation:

```
// Fixed implementation
function _distributeNXL(address recipient, uint256 amount, uint256 productId) private {
    if (amount == 0) return;

    // Explicitly check NXL availability before distribution
    uint256 available = nxlToken.getAvailableRewards();

    if (available < amount) {
        // Actually exhausted - handle it appropriately
        _handleNXLExhaustion(productId);
        return;
    }

    // Safe to distribute - any failure here is a real error that should revert
    nxlToken.distributeReward(recipient, amount);
}
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[C-03] TreasuryBTC::claimRewards can be manipulated due to balanceOf()

Affected files

TreasuryBTC#L148

Description

Reward claim in TreasuryBTC::claimRewards and TreasuryBTC::claimMultipleRewards is calculated using live balances at claim time instead of using the nxl token balance at the snapshot creation time. So the same nxl tokens can be transferred across different addresses and used to claim rewards multiple times for the a snapshot, draining protocol funds.

Vulnerable Scenario:
The following steps help understand the issue:

- Address A with nfl token balance calls TreasuryBTC::claimRewards and successfully claims tokens
- Address A sends his nfl balance to another address B. Address B calls TreasuryBTC::claimRewards. this will also work because the function uses live balances of an address.
- An attacker can repeat this multiple times to drain contract stableCoin balance.

Impact

Drain of protocol funds

Recommendations

Make use of user nxl token balance at the snapshot creation time using ERC20Snapshot:

balanceOfAt(user, snapshotId)
totalSupplyAt(snapshotId)

Nexalo:

Claim rewards system removed.

Zealynx:

We recommend to follow recommendation instead of removing chore feature.

7.2 High Severity Findings

[H-01] Ticket ownership can be overwritten due to incorrect logic in NexumManager::buyTickets

Affected files

NexumManager.sol#L265-L272

Description

The contract supports two ticket purchase flows, NexumManager::buySpecificTickets enforces a user selects ticket number system and NexumManager::buyTickets enforces a sequential assignment process. Because NexumManager::buyTickets assigns tickets based on round.ticketsSold without checking if ticketOwner[productId][roundId][ticketNumber] is already set, a ticket previously bought at a high index via NexumManager::buySpecificTickets can later be overwritten by sequential purchases. This allows an attacker to steal ticket ownership and potentially win jackpots or instant rewards tied to that ticket.

Vulnerable Scenario:

The following steps help understand the issue:

- Victim buys a high index ticket using `NexumManager::buySpecificTickets`, ticket #50.
- `ticketOwner[productId][roundId][50]` is set to the victim.
- `round.ticketsSold` is still low (only 40 sold) because `buySpecificTickets` purchases do not depend on index order.
- Ticket purchases continues as normal with calls to `NexumManager::buyTickets`, which assigns tickets sequentially:

```
uint256 ticketNumber = round.ticketsSold;
ticketOwner[productId][roundId][ticketNumber] = msg.sender;
round.ticketsSold++;
```

When round.ticketsSold reaches 49, An attacker can call NexumManager::buyTickets and overwrite:

```
ticketOwner[productId][roundId][50]
```

If ticket #50 is selected as a winner, the attacker receives the payout instead of the victim.

Impact

- Silent theft of ticket ownership.
- Attacker can gets rewards meant for another user.
- Loss of trust.

Recommendations

Prevent overwriting of tickets in `NexumManager::buyTickets`.

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[H-02] Flawed reward distribution mechanism causes pool depletion and DOS in NexaloStaking

Affected files

NexaloStaking.sol#L182-L187

Description

The NexaloStaking contract calculates WBTC rewards by converting individual users' NXL stake amounts to USD equivalents using hardcoded prices (1 NXL = $0.05, 1 WBTC = $50,000), then converting to WBTC. This approach has a fundamental design flaw: rewards are calculated independently per user based on USD values, not proportionally to the available WBTC reward pool.

```
function calculatePendingRewards(address user) public view returns (uint256) {
    uint256 timeStaked = block.timestamp - userStake.lastClaimTime;

    // Calculate 4% APY in NXL
    uint256 rewardsInNXL = (userStake.amount * APY_RATE * timeStaked) / (SECONDS_PER_YEAR * 10000);

    // Convert to WBTC using hardcoded ratio: 1 WBTC = 1,000,000 NXL
    uint256 rewardsInWBTC = (rewardsInNXL * 1e8) / (1_000_000 * 1e18);

    return rewardsInWBTC;
}
```

Vulnerable Scenario:

1. Protocol stakes externally and earns 1 WBTC ($87,000 at current prices) in rewards
2. Alice stakes 1M NXL → After 1 year, calculates 40k NXL reward (4% APY)
3. Using hardcoded prices: 40k NXL * $0.05 = $2,000 → $2,000 / $50k = 0.04 WBTC
4. Bob stakes 1M NXL → Same calculation → 0.04 WBTC reward
5. Total claims: 0.08 WBTC, but pool only has 0.0115 WBTC (1 WBTC / 87k * 1000 per user)
6. Alice claims successfully, Bob's claim reverts with "Insufficient WBTC"
7. DOS: All subsequent stakes and unstakes fail because they call _claimRewards

Impact

Each time (which will be most of the time) the prices of these assets are not the exact hardcoded values, the staking contract will deliver more or less rewards than is expected.
Given the protocol distribute staking rewards each time a user tries to stake, if these rewards in absolute btc number (calculated with an undervalued btc) is more than the actual wbtc balance of the pool (calculated with the current market rate of btc), users won't be able to stake nor unstake.

Recommendations

- Redesign the reward mechanism to use proportional distribution based on stake ratios rather than USD conversions:

Nexalo:

Fixed.

Zealynx:

Verified. Fixed

[H-03] Raffles may not distribute all funds, resulting in locked stablecoins during distribution

Affected files

NexumManager.sol#L338
ReferralNetwork.sol#L65-L97

Description

When a raffle round ends and funds are distributed, the ReferralNetwork::distributeCommissions() function fails to distribute all allocated referral commissions. When a buyer doesn't have all three referral levels populated, the corresponding commission percentages (5%, 3%, or 2%) are not distributed and remain stuck in the ReferralNetwork contract.

Vulnerable Scenario:
The following steps help understand the issue:

1. A raffle round completes with 1000 tickets sold at 1 USDT each, collecting 1000 USDT total.

2. The _distributeRound() function allocates 10% (100 USDT) for multilevel referral commissions and transfers it to the ReferralNetwork contract, then calls distributeCommissions().

3. If the winner only has 1 referral level (no level 2 or level 3 referrers):

- Level 1 receives: (100 * 5000) / 10000 = 5 USDT
- Level 2 would receive: (100 * 3000) / 10000 = 3 USDT → Not distributed (no referrer)
- Level 3 would receive: (100 * 2000) / 10000 = 2 USDT → Not distributed (no referrer)
- Result: 5 USDT stuck in ReferralNetwork contract

4. Over hundreds of rounds, these undistributed commissions accumulate into significant locked funds that cannot be recovered or redistributed.

Impact

Funds will progressively accumulate in the ReferralNetwork contract with no mechanism to recover or redistribute them. The referral commission system fails to achieve its intended 10% distribution when buyers lack complete referral chains, reducing the effectiveness of the referral incentive program. Over time, the cumulative undistributed commissions can result in substantial locked value.

Recommendations

Modify distributeCommissions() to track distributed amounts and return undistributed funds to the caller (NexumManager):

```
 function distributeCommissions(address buyer, uint256 totalAmount) external onlyManager nonReentrant
+ returns (uint256 amountUndistributed)
 {
     require(totalAmount > 0, "Invalid amount");

+    uint256 leftToDistribute = totalAmount;

     address level1 = referrer[buyer];
     if (level1 != address(0)) {
         uint256 amount1 = (totalAmount * LEVEL1_PCT) / 10000;
         if (amount1 > 0) {
             require(stablecoin.transfer(level1, amount1), "Transfer failed");
             totalEarned[level1] += amount1;
+            leftToDistribute -= amount1;
             emit CommissionPaid(level1, buyer, 1, amount1);
         }
     }

     address level2 = level1 != address(0) ? referrer[level1] : address(0);
     if (level2 != address(0)) {
         uint256 amount2 = (totalAmount * LEVEL2_PCT) / 10000;
         if (amount2 > 0) {
             require(stablecoin.transfer(level2, amount2), "Transfer failed");
             totalEarned[level2] += amount2;
+            leftToDistribute -= amount2;
             emit CommissionPaid(level2, buyer, 2, amount2);
         }
     }

     address level3 = level2 != address(0) ? referrer[level2] : address(0);
     if (level3 != address(0)) {
         uint256 amount3 = (totalAmount * LEVEL3_PCT) / 10000;
         if (amount3 > 0) {
             require(stablecoin.transfer(level3, amount3), "Transfer failed");
             totalEarned[level3] += amount3;
+            leftToDistribute -= amount3;
             emit CommissionPaid(level3, buyer, 3, amount3);
         }
     }

+    // Return undistributed funds to NexumManager
+    if (leftToDistribute > 0) {
+        require(stablecoin.transfer(msg.sender, leftToDistribute), "Return transfer failed");
+    }

+    // Returns the value of undistributed funds, to be handled in NexumManager
+    return leftToDistribute;
 }
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[H-04] Idle NXL in the token contracts result in diluted rewards for users in TreasuryBTC

Affected files

TreasuryBTC.sol
NXLToken.sol

Description

The TreasuryBTC contract distributes monthly BTC staking rewards to NXL token holders proportionally based on their holdings. However, the depositMonthlyRewards() function uses nxlToken.totalSupply() to calculate reward shares, which includes all minted NXL tokens which including the 96 million tokens sitting idle in the NXLToken contract reserved for future raffle rewards, as well as unvested founder and partner tokens.

This significantly dilutes rewards for actual NXL holders, as the reward calculation treats non-circulating tokens as if they were actively held by users.

Vulnerable Scenario:
The NXLToken contract has a total supply of 100 million NXL:

96 million NXL sitting in the contract for future raffle rewards. 3 million NXL for founder (vesting over 2 years). 1 million NXL for partner (vesting over 1 year). Alice participates in raffles and earns 10,000 NXL tokens, which she holds in her wallet.

The founder deposits 1,000 USDT as monthly BTC staking rewards via `depositMonthlyRewards()`.

The function creates a snapshot with: `totalRewards = 1,000 USDT` and `totalNXLSupply = 100,000,000 NXL` (entire total supply)

When Alice claims her rewards, the calculation is:
userReward = (1,000 USDT * 10,000 NXL) / 100,000,000 NXL = 0.1 USDT
Alice receives only 0.1 USDT despite being the only actual NXL holder who earned tokens through participation.

If the calculation used only circulating supply (10,000 NXL), Alice would receive:
userReward = (1,000 USDT * 10,000 NXL) / 10,000 NXL = 1,000 USDT
Alice loses 99.99% of her rightful rewards because idle and unvested tokens are counted in the total supply.

Impact

Users who earn NXL tokens through raffle participation receive drastically reduced rewards due to the inclusion of non-circulating tokens in the reward calculation. This creates severe unfairness:

- Actual NXL holders receive only a tiny fraction of rewards they should earn (potentially 99%+ reduction)
- Idle tokens in the NXLToken contract dilute all user rewards despite not being distributed
- Unvested founder and partner tokens further dilute rewards even though they haven't been claimed
- The vast majority of deposited rewards become unclaimable, remaining locked in the TreasuryBTC contract
- Users have no economic incentive to hold NXL tokens if rewards are negligible

Recommendations

Modify depositMonthlyRewards() to use circulating supply instead of total supply. The circulating supply should exclude tokens held by the NXLToken contract and unvested tokens:

```
function depositMonthlyRewards(uint256 rewardAmount) external onlyOwner {
    require(rewardAmount > 0, "Amount must be > 0");

    require(
        stablecoin.transferFrom(msg.sender, address(this), rewardAmount),
        "Transfer failed"
    );

    uint256 snapshotId = snapshotCount;
-   uint256 totalSupply = nxlToken.totalSupply();

+   // Calculate circulating supply by excluding non-circulating tokens
+   uint256 totalSupply = nxlToken.totalSupply();
+   uint256 contractBalance = nxlToken.balanceOf(address(nxlToken));
+   uint256 circulatingSupply = totalSupply - contractBalance;

+   require(circulatingSupply > 0, "No circulating supply");

    rewardSnapshots[snapshotId] = RewardSnapshot({
        timestamp: block.timestamp,
        totalRewards: rewardAmount,
-       totalNXLSupply: totalSupply,
+       totalNXLSupply: circulatingSupply,
        distributed: false
    });

    snapshotCount++;
    emit RewardsDeposited(snapshotId, rewardAmount);
}
```

This ensures rewards are distributed only among actual NXL holders who earned their tokens through participation, rather than being diluted by idle contract reserves.

Nexalo:

Deleted the monthly WBTC rewards for NXL holders

Zealynx:

This is a fundamental protocol change, not a fix. We recommend to follow our recommendation.

[H-05] Broken ticket index continuity leads to invalid winner selection and product.maxTickets invariant break

Affected files

NexumManager::buySpecificTickets

Description

The contract assumes that ticket indices from 0 to round.ticketsSold - 1 are continuously owned. However, NexumManager::buySpecificTickets allows users to purchase any ticket numbers while only increasing round.ticketsSold by the number of tickets bought, not by the highest index used. This creates gaps in ticket ownership that violate the round internal accounting assumptions and also breaks the maxTickets check.

Vulnerable Scenario:

- A round starts with round.ticketsSold = 0 and product.maxTickets = 1000.
- A user buys tickets through buySpecificTickets, choosing not consecutive ticket numbers such as {10, 11, 12, 13, 14}.
- At this point, round.ticketsSold == 5, but ticket indices like {0, 1, 2, 3, 4} have no owner.
- Other users continue buying tickets through buyTickets, which increases round.ticketsSold sequentially, until round.ticketsSold reaches 1000.
- When the protocol selects winners using randomness, it may select an index that has no owner.
- If that index is chosen, the reward is sent to address(0) causing loss of funds.

Impact

- Loss of rewards if a ticket with no owner becomes the winner.
- Higher odds to win since the maxTickets check can be bypassed.
- Loss of trust.

Recommendations

- Apply the maxTickets checks present in buyTickets into buySpecificTickets.
- Maintain an counter of valid ticket indices and use that structure for winner selection instead of assuming index continuity.

Nexalo:

Fixed. Applied the maxTickets checks present in buyTickets into buySpecificTickets

Zealynx:

Partially fixed. We recommend to also, maintain an counter of valid ticket indices and use that structure for winner selection instead of assuming index continuity.

7.3 Medium Severity Findings

[M-01] Incorrect access control in `TreasuryBTC::withdrawForStaking`

Affected files

TreasuryBTC.sol#L100-L101

Description

The function TreasuryBTC::withdrawForStaking has a conflicting access control design, it checks msg.sender == founder || msg.sender == owner(), but the onlyOwner modifier already restricts access to the owner. So, the founder can never call the function if they are not the owner.

```
function withdrawForStaking(uint256 amount) external onlyOwner {
        require(msg.sender == founder || msg.sender == owner(), "Only founder");
------------------------
}
```

Recommendations

Remove the onlyOwner modifier.

Nexalo:

`withdrawForStaking` functionality has been completely removed

Zealynx:

If the founder was supposed to have withdrawal access independent of the owner, that's now lost.

[M-02] Denial of service in AmbassadorRegistry::distributeFunds due to usdc/usdt blacklist behaviour

Affected files

AmbassadorRegistry.sol#L65

Description

The AmbassadorRegistry::distributeFunds function loops over all active ambassadors and transfers tokens using stablecoin.transfer:

```
require(stablecoin.transfer(ambassador, amountPerAmbassador), "Transfer failed");
```

If any ambassador address is blacklisted by usdt/usdc, the transfer will revert. This will cause the whole function to fail, preventing all other ambassadors from receiving their rewards.

Impact

Denial of service for reward distribution as it will affect all ambassadors that were supposed to receive rewards.

Recommendations

Rewrite the function to handle failed transfers separately so it does not revert the whole function:

```
bool success = stablecoin.transfer(ambassador, amountPerAmbassador);
if (success) {
    ambassadors[ambassador].totalEarned += amountPerAmbassador;
} else {
    emit TransferFailed(ambassador, amountPerAmbassador);
}
```

Use a pull based system instead of a push system.

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[M-03] `fulfillRandomWords` may revert, causing loss of funds and DoS in rewards distribution

Affected files

NexumManager#L365

Description

The function `fulfillRandomWords` is only callable by chainlink VRF to select a raffle winner when all tickets of a product have been bought. Because it can't be called manually, and because it handles distribution of funds, the function execution must never revert, or the contract will remain in a state where the product's round can't change, and funds will never be distributed.
Here are the possible reasons why the function may revert:
- The contract has less funds than the total to distribute (unlikely, as funds are drawn when users are paying for tickets)
- Any address to which the tokens are supposed to be sent is blacklisted (low likelyhood, this rarely happens)

This link refers to Chainlink's official documentation regarding reverts using VRF:
https://docs.chain.link/vrf/v2-5/security#fulfillrandomwords-must-not-revert

Vulnerable Scenario:
The following steps help understand the issue:
1. Users buy all tickets for the raffle
2. The contract calls Chainlink VRF to select a winner
3. One of the user who should receive stablecoins from the raffle is blacklisted
4. When `NexumManager` receives the VRF response, it immediatly executes the rewards calculations and transfers
5. Because a receiver is blacklisted, `fulfillRandomWords` reverts
6. `fulfillRandomWords` cannot be called with the same requestId again
7. The raffle product's round cannot increase, so we can't keep on using it for the raffles, and users can't receive rewards

Impact

- In the worst case scenario where the randomness can not be fulfilled, the raffle for this product will stop working and the rewards will remain stuck in the contract `NexumManager`.

- It is important that `fulfillRandomWords` and every functions it calls must never revert, as the contract will enter a state that cannot be fixed then: the distribution will not occur, resulting in loss of funds, and the raffle for this product will permanently be deactivated (DoS).

Recommendations

- Ensure the function can not revert under any condition (remove all `require` in `fulfillRandomWords` and called functions, and ensure external calls cannot revert).

- Consider implementing a pull-base functionnality for reward distributions, so users will pull the funds themselves. For example:

```
function _newRewards(address user, uint256 amount) private {
    pendingRewards[user] += amount;
}

function claimRewards(address user, uint256 amount) external {
    pendingRewards[user] -= amount;
    _safeTransfer(user, amount);
}
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[M-04] Funds available for distribution are burned if a user paid for more rewards, resulting in a loss of rewards.

Affected files

NexumManager
NXLToken.sol

Description

When users purchase raffle tickets in NexumManager, they are entitled to receive NXL token rewards proportional to the number of tickets purchased. However, the current implementation follows an "all-or-nothing" approach: if the NXL contract doesn't have enough tokens to fulfill the entire reward amount, the user receives nothing at all, despite having paid for their tickets.

The issue occurs in the `_distributeNXL()` function which uses a try-catch block. When `nxlToken.distributeReward()` reverts due to insufficient rewards, the catch block calls `_handleNXLExhaustion()` which deactivates the product and attempts to burn remaining tokens, but the user who just purchased tickets receives zero NXL tokens.

Vulnerable Scenario:
1. The NXL token contract has 5 NXL tokens remaining in its rewards pool (after accounting for vesting reserves).
2. Alice purchases 10 tickets for the BLACKBLOK product, which should reward her with `10 tickets * 1 NXL per ticket = 10 NXL` total.
3. The `_distributeNXL()` function is called with `amount = 10 NXL`.
4. The `distributeReward()` function in NXLToken checks available rewards and finds only 5 NXL available.
5. The function reverts with "Insufficient rewards available" because it requires exactly 10 NXL but only 5 are available.
6. The catch block in `_distributeNXL()` triggers `_handleNXLExhaustion()`, which:
   - Deactivates the BLACKBLOK product
   - Attempts to burn remaining tokens
   - Alice receives 0 NXL despite paying full price for 10 tickets
   - If she had paid the price for 5 tickets, she would have gotten 5 NXL

7. Alice has paid for 10 tickets but received no NXL rewards, while the contract still holds 5 NXL that could have been partially distributed to her.

Impact

Users who purchase tickets when the NXL rewards pool is running low receive zero NXL tokens despite paying full price. This creates severe unfairness as users pay full stablecoin price but receive 0% of expected NXL rewards when partial rewards are available.

For example, with 5 NXL remaining, a user expecting 10 NXL receives nothing instead of the available 5 NXL (100% loss vs 50% distribution).

Recommendations

Modify the reward distribution logic to support partial distributions when full rewards are unavailable:

Update `NXLToken::distributeReward()` to accept partial distributions:

```
function distributeReward(address recipient, uint256 amount) external {
    require(msg.sender == nexumManager, "Only NexumManager");
    require(recipient != address(0), "Invalid recipient");
    require(amount > 0, "Amount must be > 0");

    uint256 availableRewards = getAvailableRewards();
-   require(availableRewards >= amount, "Insufficient rewards available");

+   // Distribute available amount (full or partial)
+   uint256 amountToDistribute = availableRewards >= amount ? amount : availableRewards;
+   require(amountToDistribute > 0, "No rewards available");

-   rewardsDistributed += amount;
+   rewardsDistributed += amountToDistribute;

-   _transfer(address(this), recipient, amount);
+   _transfer(address(this), recipient, amountToDistribute);

-   emit RewardDistributed(recipient, amount);
+   emit RewardDistributed(recipient, amountToDistribute);
}
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[M-05] Incomplete accounting in receiveFunds() leads to permanent denial of service after reward claims

Affected files

TreasuryBTC.sol#L79-L86

Description

The `receiveFunds()` function calculates new deposits by comparing the current stablecoin balance against expected balance based on `totalDeposited` and `totalWithdrawnForStaking`. However, this accounting formula fails to track two critical balance changes: monthly reward deposits and reward claims, causing arithmetic underflow and permanent function failure.

```
function receiveFunds() external {
    uint256 balance = stablecoin.balanceOf(address(this));
    uint256 newDeposit = balance - (totalDeposited - totalWithdrawnForStaking);

    if (newDeposit > 0) {
        totalDeposited += newDeposit;
        emit FundsReceived(msg.sender, newDeposit);
    }
}
```

The formula assumes: `expectedBalance = totalDeposited - totalWithdrawnForStaking`

However, the actual balance is affected by:

- Monthly reward deposits (line 119-138 in `depositMonthlyRewards`) which increase balance without updating `totalDeposited`
- Reward claims (lines 158, 191 in `claimRewards` and `claimMultipleRewards)` which decrease balance and update `totalRewardsDistributed`, but this variable is never used in the `receiveFunds()` calculation

Vulnerable Scenario:

The following steps demonstrate the issue:

1. NexumManager deposits 10,000 USDC via `depositFunds()`:

    - `balance = 10,000`
    - `totalDeposited = 10,000`
    - `totalWithdrawnForStaking = 0`

2. Owner deposits 5,000 USDC as monthly BTC staking rewards via depositMonthlyRewards(5000e6):
    - `balance = 15,000`
    - `totalDeposited = 10,000` (unchanged - this is the problem)
    - Monthly rewards are NOT tracked in `totalDeposited`

3. Users claim 2,000 USDC in rewards:
    - `balance = 13,000`
    - `totalDeposited = 10,000` (unchanged)
    - totalRewardsDistributed = 2,000` (tracked but not used in `receiveFunds()`)

4. Someone calls receiveFunds():
    - `expectedBalance = 10,000 - 0 = 10,000`
    - `actualBalance = 13,000`
    - `newDeposit = 13,000 - 10,000 = 3,000`
    - Function succeeds but incorrectly attributes 3,000 as new deposit
    - `totalDeposited = 13,000` (now includes some of the reward pool)

5. Users claim another 4,000 USDC in rewards:
    - `balance = 9,000`
    - `totalDeposited = 13,000`
    - `totalWithdrawnForStaking = 0`

6. Anyone calls `receiveFunds()`:
    - `expectedBalance = 13,000 - 0 = 13,000`
    - `actualBalance = 9,000`
    - `newDeposit = 9,000 - 13,000 = -4,000`
    - Arithmetic underflow in Solidity 0.8+ → Transaction reverts

7. All subsequent calls to `receiveFunds()` permanently revert because the actual balance is now below the expected balance.

The root cause: Monthly reward deposits temporarily inflate the balance above expected levels, masking the accounting issue. Once total reward claims exceed the untracked monthly reward deposits, the actual balance falls below the calculated expected balance, causing permanent underflow.

Impact

- The `receiveFunds()` function becomes permanently unusable after normal protocol operations involving monthly reward deposits and user claims. This creates a denial of service for the permissionless accounting mechanism that NexumManager relies on to deposit treasury funds.

- While depositFunds()` remains functional as an alternative, the broken `receiveFunds()` function prevents automatic balance reconciliation and causes integration issues between NexumManager and TreasuryBTC. The function failure also prevents proper tracking of all funds entering the contract.

Recommendations

- Redesign the accounting system to track all balance-changing operations:

```
// Add state variable
uint256 public totalRewardDeposits;

function depositMonthlyRewards(uint256 rewardAmount) external onlyOwner {
    require(rewardAmount > 0, "Amount must be > 0");

    require(
        stablecoin.transferFrom(msg.sender, address(this), rewardAmount),
        "Transfer failed"
    );

    totalRewardDeposits += rewardAmount; // Track reward deposits

    uint256 snapshotId = snapshotCount;
    uint256 totalSupply = nxlToken.totalSupply();

    rewardSnapshots[snapshotId] = RewardSnapshot({
        timestamp: block.timestamp,
        totalRewards: rewardAmount,
        totalNXLSupply: totalSupply,
        distributed: false
    });

    snapshotCount++;
    emit RewardsDeposited(snapshotId, rewardAmount);
}
```

```
function receiveFunds() external {
    uint256 balance = stablecoin.balanceOf(address(this));

    // Calculate expected balance including all tracked operations
    uint256 expectedBalance = totalDeposited
        + totalRewardDeposits
        - totalWithdrawnForStaking
        - totalRewardsDistributed;

    if (balance > expectedBalance) {
        uint256 newDeposit = balance - expectedBalance;
        totalDeposited += newDeposit;
        emit FundsReceived(msg.sender, newDeposit);
    }
}
```

This ensures the expected balance calculation accounts for all funds entering and leaving the contract, preventing underflow and maintaining accurate accounting.

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[M-06] Missing VRF Failure Handling Leads to Permanent Round Lock and Fund Freezing

Affected files

NexumManager.sol#L325-L343

Description

When a lottery round reaches maxTickets, the contract requests randomness from Chainlink VRF to select a winner. However, the implementation lacks any timeout mechanism or manual resolution path if the VRF callback never arrives.

```
function _requestRandomWinner(uint256 productId, uint256 roundId) private {
    Round storage round = rounds[productId][roundId];
    require(!round.vrfRequested, "VRF already requested");

    uint256 requestId = vrfCoordinator.requestRandomWords(...);

    vrfRequestToProduct[requestId] = productId;
    vrfRequestToRound[requestId] = roundId;
    round.vrfRequested = true;
    round.vrfRequestId = requestId;

    emit VRFRequested(requestId, productId, roundId);
}
```

Once round.vrfRequested is set to true, there is no way to:
- Re-request randomness if VRF fails
- Manually complete the round
- Refund users if the round cannot complete
- Start a new round for that product

Vulnerable Scenario:

- A PREMIUM product round reaches 1000 tickets sold (maxTickets), triggering VRF request.
- The Chainlink VRF coordinator experiences downtime, or the subscription runs out of LINK tokens.
- The VRF callback never executes, leaving the round permanently stuck with `round.vrfRequested = true and `round.completed = false`.
- Line 327 prevents any re-request: `require(!round.vrfRequested, "VRF already requested")`.
- All funds from that round (1000 tickets × 20 USDC = 20,000 USDC) are locked in the contract.

- Users cannot participate in new rounds for that product since `currentRound[productId]` never increments.
- No admin function exists to resolve this stuck state.

This is not a theoretical concern—Chainlink VRF has experienced outages, and subscription management issues are common operational risks.

Impact

VRF callback failures result in permanent freezing of round funds and complete inability to continue lottery operations for the affected product. Users lose their ticket purchases with no recovery mechanism, and the product becomes permanently inoperable.

Recommendations

Implement a timeout-based permissionless fallback mechanism to handle VRF failures without relying on owner privileges:

1. Track when VRF requests are made:

```
mapping(uint256 => mapping(uint256 => uint256)) public roundVRFRequestTime;
function _requestRandomWinner(uint256 productId, uint256 roundId) private {
    // ... existing code ...
    roundVRFRequestTime[productId][roundId] = block.timestamp;
}
```

2. Add a permissionless emergency resolution function:

```
function resolveStuckRound(
    uint256 productId,
    uint256 roundId
) external {
    Round storage round = rounds[productId][roundId];
    require(round.vrfRequested && !round.completed, "Round not stuck");
    require(
        block.timestamp > roundVRFRequestTime[productId][roundId] + 7 days,
        "VRF timeout not reached"
    );

    // Use block-based fallback randomness
    uint256 fallbackRandom = uint256(
        keccak256(abi.encodePacked(
            block.timestamp,
            block.prevrandao,
            block.number,
            productId,
            roundId
        ))
    );
```

```
    uint256 winningTicket = fallbackRandom % round.ticketsSold;
    address winner = ticketOwner[productId][roundId][winningTicket];

    round.vrfRandomWord = fallbackRandom;
    round.winner = winner;
    round.completed = true;

    _distributeRound(productId, roundId, winner, fallbackRandom);
    emit RoundCompleted(productId, roundId, winner,
                        products[productId].jackpotUSD, winningTicket);
    _startNewRound(productId);
}
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[M-07] Immutable audit funds address and missing approval mechanism leads to unreachable cleanup logic and mixed fund accounting

Description

NexumManager.sol#L147
NexumManager.sol#L315-L322
NexumManager.sol#L409-L410

Description

The auditFunds address is initialized to the same value as founder in the constructor and cannot be changed due to missing setter functions. This creates multiple issues with audit fund management and cleanup logic.

```
// NexumManager.sol:L145-L147 (constructor)
founder = _founder;
partner = _partner;
auditFunds = _founder;  // Same as founder, permanently
```

During prize distribution, the contract sends separate payments to both addresses:

```
// NexumManager.sol:L406-L416
uint256 founderAmount = (operationsTotal * FOUNDER_SHARE) / 1000;
_safeTransfer(founder, founderAmount);

uint256 auditAmount = (operationsTotal * AUDIT_SHARE) / 1000;
_safeTransfer(auditFunds, auditAmount);  // Goes to founder since auditFunds == founder

uint256 feesAmount = (operationsTotal * FEES_SHARE) / 1000;
_safeTransfer(founder, feesAmount);  // Also to founder
```

The cleanup logic in _handleNXLExhaustion() attempts to transfer audit funds when all products become inactive:

```
// NexumManager.sol:L315-L322
if (allInactive && auditFunds != address(0) && auditFunds != founder) {
    uint256 auditBalance = stablecoin.balanceOf(auditFunds);
    if (auditBalance > 0) {
        try stablecoin.transferFrom(auditFunds, founder, auditBalance) {
            emit AuditFundsTransferred(founder, auditBalance);
        } catch {}
    }
}
```

Vulnerable Scenario:

1. Contract is deployed with founder and auditFunds both pointing to the same address
2. During lottery operations, the contract distributes funds:
   a. Founder receives founderAmount + feesAmount directly
   b. auditFunds receives auditAmount (but goes to same address as founder)
3. All three payment streams mix in the founder's address with no way to distinguish audit funds
4. When all products become inactive, the cleanup logic attempts to execute
5. The condition auditFunds != founder evaluates to false since they're the same address
6. The audit fund transfer block never executes - dead code
7. Even if auditFunds could be changed, the transferFrom would fail because:
   a. It attempts to transfer from auditFunds address to founder
   b. auditFunds address has never approved NexumManager to spend tokens
   c. No mechanism exists for granting this approval
8. No setter functions exist to update founder, partner, or auditFunds addresses post-deployment

Impact

- The audit fund management is fundamentally broken with multiple consequences. The cleanup logic at protocol end-of-life is completely unreachable dead code since auditFunds != founder is hardcoded to false forever with no way to change either address post-deployment.
- During normal operations, audit funds, founder operational funds, and protocol fees all mix in the same address with no accounting separation, making it impossible to track or manage audit funds independently. The design confusion is evident: the code sends audit payments TO auditFunds during distribution but then attempts to retrieve funds FROM auditFunds at exhaustion using transferFrom without any approval mechanism. This entire feature serves no purpose in its current implementation.

Recommendations

Implement a pull base system allowing users to pull rewards themselves.

Recommendations

Use pull pattern for audit funds

Instead of pushing funds at exhaustion, allow authorized address to withdraw audit allocation:

```
mapping(address => uint256) public auditFundBalance;

function _distributePrizes(...) private {
    // ... existing logic ...

    uint256 auditAmount = (operationsTotal * AUDIT_SHARE) / 1000;
    auditFundBalance[auditFunds] += auditAmount;  // Track separately

    // ... rest of distribution ...
}

function withdrawAuditFunds() external {
    require(msg.sender == auditFunds, "Not authorized");
    uint256 amount = auditFundBalance[msg.sender];
    require(amount > 0, "No funds");

    auditFundBalance[msg.sender] = 0;
    require(stablecoin.transfer(msg.sender, amount), "Transfer failed");
}
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[M-08] Broken token burn logic prevents undistributed reward cleanup

Affected Files

- NexumManager.sol#L303
- NXLToken.sol#L231-L241

Description

The _handleNXLExhaustion() function attempts to burn undistributed NXL rewards when a product's reward pool is exhausted. However, it incorrectly passes the entire NXL token contract balance to the burnUndistributed() function instead of only the available rewards amount.

```
// NexumManager.sol:L303
try nxlToken.burnUndistributed(nxlToken.balanceOf(address(nxlToken))) {
    emit NXLRewardsExhausted(productId);
} catch {}
```

The burnUndistributed() function in NXLToken.sol protects vested tokens by checking against getAvailableRewards():

```
// NXLToken.sol:L231-241
function burnUndistributed(uint256 amount) external {
    require(msg.sender == nexumManager, "Only NexumManager");
    require(amount > 0, "Amount must be > 0");

    uint256 availableRewards = getAvailableRewards();
    require(availableRewards >= amount, "Insufficient balance");

    _burn(address(this), amount);
    emit TokensBurned(amount);
}
```

Vulnerable Scenario:

- NXL reward pool becomes exhausted during a ticket purchase
- _handleNXLExhaustion() is called and attempts to burn tokens
- Contract passes balanceOf(address(nxlToken)) which includes both available rewards and reserved vesting tokens (e.g., 50M available + 4M vesting = 54M total).
- burnUndistributed() receives 54M but getAvailableRewards() returns only 50M

- The require(availableRewards >= amount) check fails: 50M < 54M
- Transaction reverts but the empty catch {} block silently swallows the error
- No tokens are actually burned
- The NXLRewardsExhausted event is never emitted
- System believes cleanup occurred but nothing happened

Impact

The token cleanup mechanism fails silently every time it's triggered. When NXL rewards are exhausted, any remaining undistributed reward tokens (dust amounts or leftovers) remain in the contract instead of being burned as intended. The NXLRewardsExhausted event is never emitted, making it impossible to track when products have actually exhausted their reward pools through event monitoring. While vested tokens are protected from destruction by the getAvailableRewards() check, the intended cleanup behavior never executes.

Recommendations

Check available rewards explicitly before burning and always emit the exhaustion event. Remove try/catch which masks real errors:

```
function _handleNXLExhaustion(uint256 productId) private {
    products[productId].active = false;

    // Query available rewards first
    uint256 availableRewards = nxlToken.getAvailableRewards();

    // Burn only if there are tokens to burn
    if (availableRewards > 0) {
        nxlToken.burnUndistributed(availableRewards);
    }

    // Always emit the exhaustion event
    emit NXLRewardsExhausted(productId);

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

This approach:
- Explicitly checks state before taking action
- Only calls burn when there are actually tokens to burn
- Doesn't hide unexpected errors (if burn fails, the transaction reverts as it should)
- Always emits the exhaustion event for proper monitoring
- Makes the code logic clear and predictable

Nexalo:

Token Burn Mechanism removed

Zealynx:

Bug avoided by removing the feature. This is a significant protocol change, not a fix.

[M-09] Insufficient available rewards will result in locked staked funds for users

Affected files

TreasuryBTC.sol#L79-L86

Description

The receiveFunds() function calculates new deposits by comparing the current stablecoin balance against expected balance based on totalDeposited and totalWithdrawnForStaking. However, this accounting formula fails to track two critical balance changes: monthly reward deposits and reward claims, causing arithmetic underflow and permanent function failure.

```
function receiveFunds() external {
    uint256 balance = stablecoin.balanceOf(address(this));
    uint256 newDeposit = balance - (totalDeposited - totalWithdrawnForStaking);

    if (newDeposit > 0) {
        totalDeposited += newDeposit;
        emit FundsReceived(msg.sender, newDeposit);
    }
}
```

The formula assumes: expectedBalance = totalDeposited - totalWithdrawnForStaking

However, the actual balance is affected by:

- Monthly reward deposits (line 119-138 in depositMonthlyRewards) which increase balance without updating totalDeposited
- Reward claims (lines 158, 191 in claimRewards and claimMultipleRewards) which decrease balance and update totalRewardsDistributed, but this variable is never used in the receiveFunds() calculation

Vulnerable Scenario:

The following steps demonstrate the issue:

1. NexumManager deposits 10,000 USDC via depositFunds():
   a. balance = 10,000
   b. totalDeposited = 10,000
   c. totalWithdrawnForStaking = 0
2. Owner deposits 5,000 USDC as monthly BTC staking rewards via depositMonthlyRewards(5000e6):
   a. balance = 15,000
   b. totalDeposited = 10,000 (unchanged - this is the problem)
   c. Monthly rewards are NOT tracked in totalDeposited
3. Users claim 2,000 USDC in rewards:
   a. balance = 13,000
   b. totalDeposited = 10,000 (unchanged)
   c. totalRewardsDistributed = 2,000 (tracked but not used in receiveFunds())
4. Someone calls receiveFunds():
   a. expectedBalance = 10,000 - 0 = 10,000
   b. actualBalance = 13,000
   c. newDeposit = 13,000 - 10,000 = 3,000
   d. Function succeeds but incorrectly attributes 3,000 as new deposit
   e. totalDeposited = 13,000 (now includes some of the reward pool)
5. Users claim another 4,000 USDC in rewards:
   a. balance = 9,000
   b. totalDeposited = 13,000
   c. totalWithdrawnForStaking = 0
6. Anyone calls receiveFunds():
   a. expectedBalance = 13,000 - 0 = 13,000
   b. actualBalance = 9,000
   c. newDeposit = 9,000 - 13,000 = -4,000
   d. Arithmetic underflow in Solidity 0.8+ → Transaction reverts
7. All subsequent calls to receiveFunds() permanently revert because the actual balance is now below the expected balance.

The root cause: Monthly reward deposits temporarily inflate the balance above expected levels, masking the accounting issue. Once total reward claims exceed the untracked monthly reward deposits, the actual balance falls below the calculated expected balance, causing permanent underflow.

Impact

The receiveFunds() function becomes permanently unusable after normal protocol operations involving monthly reward deposits and user claims. This creates a denial of service for the permissionless accounting mechanism that NexumManager relies on to deposit treasury funds.

While depositFunds() remains functional as an alternative, the broken receiveFunds() function prevents automatic balance reconciliation and causes integration issues between NexumManager and TreasuryBTC. The function failure also prevents proper tracking of all funds entering the contract.

Recommendations

Implement a pull base system allowing users to pull rewards themselves.

```
// Add state variable
uint256 public totalRewardDeposits;

function depositMonthlyRewards(uint256 rewardAmount) external onlyOwner {
    require(rewardAmount > 0, "Amount must be > 0");

    require(
        stablecoin.transferFrom(msg.sender, address(this), rewardAmount),
        "Transfer failed"
    );

    totalRewardDeposits += rewardAmount; // Track reward deposits

    uint256 snapshotId = snapshotCount;
    uint256 totalSupply = nxlToken.totalSupply();

    rewardSnapshots[snapshotId] = RewardSnapshot({
        timestamp: block.timestamp,
        totalRewards: rewardAmount,
        totalNXLSupply: totalSupply,
        distributed: false
    });

    snapshotCount++;
    emit RewardsDeposited(snapshotId, rewardAmount);
}

function receiveFunds() external {
    uint256 balance = stablecoin.balanceOf(address(this));

    // Calculate expected balance including all tracked operations
    uint256 expectedBalance = totalDeposited
        + totalRewardDeposits
        - totalWithdrawnForStaking
        - totalRewardsDistributed;

    if (balance > expectedBalance) {
        uint256 newDeposit = balance - expectedBalance;
        totalDeposited += newDeposit;
        emit FundsReceived(msg.sender, newDeposit);
    }
}
```

This ensures the expected balance calculation accounts for all funds entering and leaving the contract, preventing underflow and maintaining accurate accounting.

Nexalo:

Fixed

Zealynx:

The DOS vulnerability remains. Users can still be locked out of staking/unstaking when WBTC reserves are insufficient to pay their pending rewards.

[M-10] Distribution of NXL failing during round settlement will not deactivate the product, resulting in a new raffle with new rewards available

Affected files

TreasuryBTC.sol#L79-L86

Description

The _distributeRound function distributes the NXL winner bonus by calling nxlToken.distributeReward(winner, product.nxlWinnerBonus) directly instead of using the _distributeNXL helper, which encapsulates the exhaustion handling logic for the NXL token supply.

When distributeReward fails because the NXL reward supply is exhausted, the try/catch in _distributeRound simply ignores the failure and does not trigger _handleNXLExhaustion, so products[productId].active remains true and the product is not deactivated.

Vulnerable Scenario:
The following steps help understand the issue:

1. The last ticket of a raffle is bought and rewards distribution occurs normally
2. The winner is selected, but there isn't enough NXL rewards for them
3. nxlToken.distributeReward fails silently, and the product remains active
4. A new rounds is started, without NXL available anymore

Recommendations

Replace the direct try nxlToken.distributeReward(winner, product.nxlWinnerBonus) { } catch { } call in _distributeRound with a call to _distributeNXL(winner, product.nxlWinnerBonus, productId) so that exhaustion of NXL rewards always triggers _handleNXLExhaustion and deactivates the product as intended.

At the end of NexumManager::_distributeRound:

```
-    try nxlToken.distributeReward(winner, product.nxlWinnerBonus) {} catch {}
+    _distributeNXL(winner, product.nxlWinnerBonus, productId);
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

7.4 Low Severity Findings

[L-01] Unsafe ERC20 transfer operations allow silent failures and incompatibility with non-standard tokens

Affected Files

TreasuryBTC.sol#L92, L115, L123, L158, L191, L270, L277
NexumManager.sol#L218, L263, L318, L464
ReferralNetwork.sol#L72, L82, L92
AmbassadorRegistry.sol#L65
DonationVault.sol#L66, L99, L266
NexaloStaking.sol#L87, L129, L160, L207, L276, L286

Description

The protocol uses two unsafe patterns for ERC20 token transfers throughout the codebase:

Pattern 1: Unchecked transfers

```
// TreasuryBTC.sol:277
function recoverERC20(address tokenAddress, uint256 amount) external onlyOwner {
    require(tokenAddress != address(stablecoin), "Cannot recover stablecoin");
    require(amount > 0, "Amount must be > 0");

    IERC20(tokenAddress).transfer(owner(), amount); // No return value check
}
```

This pattern appears in:
- `TreasuryBTC.sol:277` - `recoverERC20()`
- `DonationVault.sol:266` - `recoverTokens()`
- `NexaloStaking.sol:286` - `recoverToken()`

These transfers have **no return value validation**. If the token returns `false` on failure (instead of reverting), the function continues execution as if the transfer succeeded, causing state updates without actual token movement.

Pattern 2: Require-wrapped transfers (compatibility issue)

```
// TreasuryBTC.sol:158
require(stablecoin.transfer(msg.sender, userReward), "Transfer failed");
```

This pattern assumes all ERC20 tokens return a boolean value. However, some tokens (like certain implementations) may:
- Return nothing (void) instead of bool
- Not fully comply with ERC20 standard return values
- Cause unexpected behavior with strict ABI decoding

Additionally, the protocol doesn't handle tokens that return `false` on failure instead of reverting. While `require()` catches explicit reverts, it doesn't protect against tokens that return `false` to signal failure.

Recommendations:

Adopt OpenZeppelin's `SafeERC20` library for all ERC20 token interactions:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract TreasuryBTC is Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;

    IERC20 public immutable stablecoin;
    IERC20 public immutable nxlToken;

    // Replace all transfers:
    // OLD: require(stablecoin.transfer(msg.sender, amount), "Transfer failed");
    // NEW: stablecoin.safeTransfer(msg.sender, amount);

    // OLD: require(stablecoin.transferFrom(msg.sender, address(this), amount), "Transfer failed");
    // NEW: stablecoin.safeTransferFrom(msg.sender, address(this), amount);

    // OLD: IERC20(token).transfer(owner(), amount);
    // NEW: IERC20(token).safeTransfer(owner(), amount);
}
```

Apply this pattern to all contracts.

Nexalo:

Fixed

Zealynx:

Verified. Fixed.

[L-02] Incorrect Emergency Withdrawal Implementation Allows Extraction of Active Round Funds

Description

The `emergencyWithdraw()` function is designed to recover tokens that are mistakenly sent to the contract. However, its current implementation withdraws the entire stablecoin balance without distinguishing between funds belonging to active rounds and unaccounted tokens.

```
function emergencyWithdraw() external onlyOwner {
    uint256 balance = stablecoin.balanceOf(address(this));
    require(balance > 0, "No balance");
    _safeTransfer(owner(), balance);
}
```

The contract does not maintain internal accounting to track which funds belong to active lottery rounds. Since the protocol operates with 6 different products running simultaneous rounds, the contract's balance includes:
- Funds from ongoing rounds waiting to reach `maxTickets`
- Funds from completed rounds waiting for VRF callback
- Any mistakenly sent tokens

The emergency withdrawal function cannot safely fulfill its intended purpose. Attempting to recover mistakenly sent funds will inadvertently extract funds belonging to active lottery rounds, breaking the prize distribution mechanism and causing loss of user funds. This makes the emergency function unusable in its current form.

Recommendations:

Implement internal accounting to differentiate between active round funds and unaccounted tokens:

1. Add a state variable to track funds in active rounds:

```
uint256 public totalActiveRoundFunds;
```

2. Update when tickets are purchased:

```
round.totalCollected += totalPrice;
totalActiveRoundFunds += totalPrice;
```

3. Decrease when rounds are distributed:

```
function _distributeRound(...) private {
    uint256 total = round.totalCollected;
    totalActiveRoundFunds -= total;
    // ... rest of distribution
}
```

Implement internal accounting to differentiate between active round funds and unaccounted tokens:
4. Modify `emergencyWithdraw()` to only extract unaccounted funds:

```
function emergencyWithdraw() external onlyOwner {
    uint256 balance = stablecoin.balanceOf(address(this));
    uint256 unaccountedFunds = balance - totalActiveRoundFunds;
    require(unaccountedFunds > 0, "No unaccounted funds");
    _safeTransfer(owner(), unaccountedFunds);
}
```

Nexalo:
Removed `emergencyWithdraw()` function

Zealynx:
Removing emergencyWithdraw() eliminates the immediate risk but creates a permanent stuck fund problem.

[L-03] Uninitialized lastWithdrawalTime allows immediate first withdrawal bypassing 30-day timelock

Description

The `withdrawForStaking` function enforces a 30-day minimum interval between withdrawals to protect user funds from rapid extraction. However, the `lastWithdrawalTime` state variable is never initialized and defaults to `0`, allowing the first withdrawal to bypass the timelock completely.

```
function withdrawForStaking(uint256 amount) external onlyOwner {
    require(msg.sender == founder || msg.sender == owner(), "Only founder");
    require(stakingActive, "Staking not active");
    require(amount > 0, "Amount must be > 0");
    require(
        block.timestamp >= lastWithdrawalTime + MIN_WITHDRAWAL_INTERVAL,
        "Must wait 30 days"
    );

    uint256 availableBalance = stablecoin.balanceOf(address(this));
    require(availableBalance >= amount, "Insufficient balance");

    lastWithdrawalTime = block.timestamp;
    totalWithdrawnForStaking += amount;

    require(stablecoin.transfer(founder, amount), "Transfer failed");
    emit FundsWithdrawnForStaking(founder, amount);
}
```

The timelock check at line 104-105 compares `block.timestamp` against `lastWithdrawalTime + MIN_WITHDRAWAL_INTERVAL`. When `lastWithdrawalTime = 0`, this becomes `block.timestamp >= 0 + 30 days`, which is always true since current Unix timestamps are far greater than 30 days (current time is ~1.7 billion seconds since epoch).

The 30-day withdrawal timelock is bypassed for the first withdrawal, allowing immediate extraction of all accumulated treasury funds. This defeats the purpose of the timelock protection, which is meant to prevent rapid fund extraction and provide transparency/security for users who contribute to the treasury through lottery participation. While the timelock functions correctly after the first withdrawal, the initial bypass creates a window where user funds are not adequately protected.

Recommendations

Adopt OpenZeppelin's SafeERC20 library for all ERC20 token interactions:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract TreasuryBTC is Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;

    IERC20 public immutable stablecoin;
    IERC20 public immutable nxlToken;

    // Replace all transfers:
    // OLD: require(stablecoin.transfer(msg.sender, amount), "Transfer failed");
    // NEW: stablecoin.safeTransfer(msg.sender, amount);

    // OLD: require(stablecoin.transferFrom(msg.sender, address(this), amount), "Transfer failed");
    // NEW: stablecoin.safeTransferFrom(msg.sender, address(this), amount);

    // OLD: IERC20(token).transfer(owner(), amount);
    // NEW: IERC20(token).safeTransfer(owner(), amount);
}
```

Nexalo:

Removed `withdrawForStaking function

Zealynx:

With this design choice, the protocol no longer supports direct withdrawal of treasury funds for staking purposes.

[L-04] Raffle winner can also win instant rewards, which will give them more than 50% of the expected allocation

Affected files

- NexumManager.sol#L448

Description

During the reward distribution process in _distributeRound(), the main raffle winner can also be selected as one of the instant reward winners. This allows a single user to win both the main prize (50% of the pot) and one or more instant rewards (up to 10% of the pot), which contradicts the intended fairness of distributing rewards across multiple participants.

The issue occurs because _distributeInstantRewardsVRF() selects instant winners using a deterministic algorithm based on the same VRF random word used to select the main winner, but it does not exclude the main winner's ticket from the instant winner selection.

Vulnerable Scenario:
The following steps help understand the issue:

1. A raffle round completes with 100 tickets sold, collecting 100 USDT total.
2. The VRF callback in fulfillRandomWords() selects ticket #42 as the winning ticket, owned by Alice.
3. Alice is set as the main winner and receives the main prize: (100 * 5000) / 10000 = 50 USDT.
4. The _distributeInstantRewardsVRF() function is called to distribute instant rewards worth (100 * 1000) / 10000 = 10 USDT among up to 10 winners.
5. The function calculates instant winner tickets using:
   a. seed = keccak256(randomWord, totalCollected, productId, roundId)
   b. offset = seed % 100
   c. step = 100 / 10 = 10
   d. Instant winners at tickets: offset, offset+10, offset+20, ..., offset+90 (all mod 100)
6. If offset = 42 or any of the calculated positions equals 42, Alice's ticket #42 is selected again as an instant winner.
7. Alice receives an additional 10 USDT / 10 = 1 USDT instant reward on top of her 50 USDT main prize.

The main raffle winner can receive additional instant rewards, concentrating more winnings in a single participant rather than distributing them fairly across multiple users. This undermines the purpose of instant rewards, which should provide smaller prizes to additional participants to increase engagement and perceived fairness. Users who could have won instant rewards lose their opportunity when the main winner occupies one of the instant winner slots.

Recommendations

Modify _distributeInstantRewardsVRF() to exclude the main winner from instant reward selection:

```
function _distributeInstantRewardsVRF(
    uint256 productId,
    uint256 roundId,
    uint256 totalAmount,
    uint256 randomWord
) private {
    Round storage round = rounds[productId][roundId];

    if (round.ticketsSold == 0 || totalAmount == 0) return;

    uint256 winnersCount = round.ticketsSold < 10 ? round.ticketsSold : 10;
    uint256 prizePerWinner = totalAmount / winnersCount;

    uint256 seed = uint256(keccak256(abi.encodePacked(
        randomWord,
        round.totalCollected,
        productId,
        roundId
    )));

    uint256 mainWinningTicket = randomWord % round.ticketsSold;
    uint256 offset = seed % round.ticketsSold;
    uint256 step = round.ticketsSold / winnersCount;

    for (uint256 i = 0; i < winnersCount; i++) {
        uint256 ticketIndex = (offset + (i * step)) % round.ticketsSold;

        // Skip main winner's ticket
        if (ticketIndex == mainWinningTicket) continue;

        address instantWinner = ticketOwner[productId][roundId][ticketIndex];

        if (instantWinner != address(0)) {
            _safeTransfer(instantWinner, prizePerWinner);
        }
    }
}
```

Nexalo:

Fixed

Zealynx:

Verified. Fixed.