Pashov Audit Group

HYBUX
Security Review

November 11th 2025 - November 12th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. About HYBUX 4
5. Executive Summary 4
6. Findings 5
Critical findings 6
[C-01] Incorrect reward calculation 6
High findings 8
[H-01] Cross-contract signature replay allows users to inflate rewards 8
Medium findings 9
[M-01] Function signature mismatch causes DoS 9
[M-02] Tier Removal in `setRarityWeights()` Can Invalidate Active Stakes 10
Low findings 11
[L-01] Missing deadline and nonce in signature 11
[L-02] NFTs can be stuck for smart contract users without ERC721 receiver support 11
[L-03] Missing storage gap in `NFTStaking` could cause storage corruption in upgrades 12
[L-04] Global reward pool design creates race condition 12
[L-05] Stale data in `tokenToRarityIndex` mapping 12
[L-06] Lack of Event Emissions in Critical NFTStaking Functions 13
[L-07] Unstaking fails if contract lacks sufficient HYBUX balance, causing stuck NFTs 13
[L-08] Stakers can actively abuse rate changes to maximize rewards unfairly 14

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

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

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About HYBUX

Hybux is a staking and reward distribution system designed to manage NFT-based staking, pooled emissions, and tokenized reward flows.

5. Executive Summary

A time-boxed security review of the **hytopiagg/hybux-staking-audit** repository was done by Pashov Audit Group, during which **h2134**, **0xl33**, **0xbepresent**, **imsrybr0** engaged to review **HYBUX**. A total of **12** issues were uncovered.

Protocol Summary	
Project Name	HYBUX
Protocol Type	Token Staking
Timeline	November 11th 2025 - November 12th 2025

Review commit hash:

- 735e5a99d4797482b4a85dc355dc8817ea2184ca (hytopiagg/hybux-staking-audit)

Fixes review commit hash:

- 3834eff5b7ca36c1af577d9470d06ee65731b6bc (hytopiagg/hybux-staking-audit)

Scope

`HYBUX.sol` `IHYBUX.sol` `NFTStaking.sol` `NFTStakingRouter.sol` `NodeKeyPool.sol`

6. Findings

Findings count

Severity	Amount
Critical	1
High	1
Medium	2
Low	8
Total findings	**12**

Summary of findings

ID	Title	Severity	Status
[C-01]	Incorrect reward calculation	Critical	**Resolved**
[H-01]	Cross-contract signature replay allows users to inflate rewards	High	**Resolved**
[M-01]	Function signature mismatch causes DoS	Medium	**Resolved**
[M-02]	Tier Removal in `setRarityWeights()` Can Invalidate Active Stakes	Medium	**Resolved**
[L-01]	Missing deadline and nonce in signature	Low	**Resolved**
[L-02]	NFTs can be stuck for smart contract users without ERC721 receiver support	Low	Acknowledged
[L-03]	Missing storage gap in `NFTStaking` could cause storage corruption in upgrades	Low	**Resolved**
[L-04]	Global reward pool design creates race condition	Low	Acknowledged
[L-05]	Stale data in `tokenToRarityIndex` mapping	Low	Acknowledged
[L-06]	Lack of Event Emissions in Critical NFTStaking Functions	Low	**Resolved**
[L-07]	Unstaking fails if contract lacks sufficient HYBUX balance, causing stuck NFTs	Low	**Resolved**
[L-08]	Stakers can actively abuse rate changes to maximize rewards unfairly	Low	Acknowledged

Critical findings

[C-01] Incorrect reward calculation

Severity

Impact: High

Likelihood: High

Description

The `_unstakeNFTs` internal function calls `claimRewards()` instead of `_claimRewards(_sender)` when processing unstaking operations. While this works correctly for direct unstaking (where `msg.sender` is the user), it fails when unstaking through the router because `msg.sender` becomes the router contract address.

```
File: NFTStaking.sol
    function unstakeNFTsRouter(address _sender, uint256[] calldata _tokenIds) external {
        require(msg.sender == stakingRouterAddress, "Only router");

@>      _unstakeNFTs(_sender, _tokenIds);
    }
...
    function claimRewards() public {
@>      _claimRewards(msg.sender);
    }
...
    function _unstakeNFTs(address _sender, uint256[] calldata _tokenIds) internal {
@>    claimRewards();
```

When a user unstakes through `unstakeNFTsRouter`, the function correctly passes the user's address as `_sender` to `_unstakeNFTs`. However, `_unstakeNFTs` then calls `claimRewards()`, which internally calls `_claimRewards(msg.sender)`. Since `msg.sender` is the router contract, rewards are calculated and transferred to the router instead of the user.

The staking implementation correctly anticipated router usage and consistently used `_sender` for all storage operations. Consider the following scenario:

1. User Alice has staked NFTs for 7 days, accumulating 1000 HYBUX in rewards
2. Alice calls `StakingRouter.unstakeNFTs()` with her NFT token IDs
3. Router calls `NFTStaking.unstakeNFTsRouter(aliceAddress, tokenIds)`
4. `unstakeNFTsRouter` validates caller is router, then calls `_unstakeNFTs(aliceAddress, tokenIds)`
5. `_unstakeNFTs` calls `claimRewards()` - but `msg.sender` is the router contract
6. `_claimRewards(routerAddress)` calculates rewards for router instead of Alice

7. Alice loses her 1000 HYBUX rewards

Recommendations

`NFTStaking._unstakeNFTs();` change to `_claimRewards(_sender);` to ensure rewards are always calculated for the correct user address regardless of whether unstaking occurs directly or through the router.

High findings

[H-01] Cross-contract signature replay allows users to inflate rewards

Severity

Impact: High

Likelihood: Medium

Description

The signature verification in `NFTStaking._stakeNFTs()` does not include the contract address in the hash, allowing signatures to be replayed across the three different NFTStaking contracts (WORLDS_STAKING, GRAYBOYS_STAKING, AVATARS_STAKING).

```
bytes32 hash = keccak256(abi.encode(_sender, _tokenIds, _rarityWeightIndexes));
```

An attacker who owns the same token ID across different NFT collections can obtain a signature for a high-rarity NFT in one collection, then replay that signature when staking a low-rarity NFT with the same token ID in another collection. This results in the low-rarity NFT receiving rewards as if it were high-rarity.

Attack Example: 1. User owns token #100 in both Worlds (e.g. legendary, weight 100) and Grayboys (e.g. common, weight 1) collections 2. User obtains valid signature and stakes in Worlds contract (legitimate) 3. User replays the same signature on Grayboys contract with the common NFT 4. Common NFT now earns rewards with legendary multiplier

Recommendations

Include the contract address in the signature hash to prevent cross-contract replay:

```
bytes32 hash = keccak256(abi.encode(_sender, _tokenIds, _rarityWeightIndexes, address(this)));
```

Medium findings

[M-01] Function signature mismatch causes DoS

Severity

Impact: Low

Likelihood: High

Description

The `INFTStaking` interface in `NFTStakingRouter` defines `calculateRewards()` with a single parameter, but the actual implementation in `NFTStaking` is different:

Interface:

```
function calculateRewards(address user) external returns (uint256)
```

Implementation:

```
function calculateRewards(address _user, bool _includePendingRewards) public view returns
(uint256)
```

When `NFTStakingRouter.claimRewardsRouter()` calls `calculateRewards()`, the function selector doesn't match, causing the call to revert. This results in a complete denial of service for users attempting to claim rewards through the router.

Recommendations

Update the interface and router calls to match the implementation:

```
// In INFTStaking interface:
function calculateRewards(address user, bool includePendingRewards) public view returns
(uint256);

// In NFTStakingRouter.claimRewardsRouter():
uint256 amount = INFTStaking(stakingAddr).calculateRewards(msg.sender, true);
```

Ensure `_includePendingRewards` is set to `true` to match the actual amount claimed (relevant for the emitted event).

[M-02] Tier Removal in `setRarityWeights()` Can Invalidate Active Stakes

Severity

Impact: High

Likelihood: Low

Description

The `setRarityWeights()` function allows the contract owner to update the `rarityWeights` array. However, the function contains a critical flaw: it does not check whether the new, shorter array is removing rarity tiers that currently have active NFT stakes.

NFTStaking.sol#L100-L104:

```
function setRarityWeights(
    uint256[] calldata _rarityWeights
) public onlyOwner {
    rarityWeights = _rarityWeights;
}
```

If the owner submits a transaction to shorten the `rarityWeights` array, any NFTs staked at the now-removed indices will become effectively frozen between the time the transaction is mined and the time they are unstaked. The `getRarityWeightMultiplier()` function, which iterates only over the length of the new, shorter array, will exclude these NFTs from reward calculations.

This results in a permanent loss of rewards for the affected users for as long as their NFTs remain staked under the invalid index. Even if the owner checks the state before broadcasting and confirms no NFTs are staked in the tiers to be removed, a user could submit a completely normal and legitimate staking transaction after the owner's transaction is broadcast but before it is mined. When the owner's transaction executes, it will remove the tiers, immediately invalidating the newly staked NFTs.

Recommendations

Add an on-chain check in `setRarityWeights()` to prevent reducing the `rarityWeights` array length if any NFTs are staked in the tiers to be removed.

Low findings

[L-01] Missing deadline and nonce in signature

The signature hash in `NFTStaking._stakeNFTs()` does not include a deadline or nonce parameter. This creates several issues:

- If an NFT's rarity is downgraded (offchain logic changes, where same nft corresponds to different rarity weight index), users can continue using old signatures with higher rarity indexes. For example, if NFT #100 changes from Legendary (index 3) to Common (index 0) due to metadata updates or rarity rebalancing, a user holding an old signature can still stake with index 3 and earn rewards at the legendary multiplier indefinitely.
- Signatures remain valid forever. If the signature authority's private key is compromised or intentionally rotated to a new key, all previously issued signatures continue to work. The owner has no way to invalidate old signatures without upgrading the contract.
- Without nonce tracking, signatures can be reused after unstaking. While this may be intentional design, it prevents the signature authority from correcting mistakenly issued signatures with incorrect rarity values.

Recommendations: Include a deadline timestamp and nonce in the signature hash to enable expiry and one-time use. Implement nonce tracking per user to prevent reuse, and add deadline validation to ensure signatures expire after a reasonable timeframe.

[L-02] NFTs can be stuck for smart contract users without ERC721 receiver support

The `_unstakeNFTs()` function in `NFTStaking` uses `safeTransferFrom()` to return NFTs to users. When the user is a smart contract that does not implement the `onERC721Received()` function, the transfer will revert due to the `checkOnERC721Received()` safety check in `ERC721Utils`. This causes the unstaking transaction to fail, permanently locking the NFT in the staking contract unless the contract is upgraded.

Recommendations: Use regular transfer instead of safe version when returning NFTs to users during unstaking. Alternatively, implement a check during staking that ensures smart contract users (`code.length > 0`) support the ERC721 receiver interface before allowing them to stake.

[L-03] Missing storage gap in `NFTStaking` could cause storage corruption in upgrades

The `getStakedNFTs()` function in `NFTStaking` is marked as `virtual`, indicating the contract may be designed for inheritance. However, the contract lacks a storage gap (`uint256[50] private __gap`). If `NFTStaking` is inherited by another contract and later upgraded to add new state variables, those variables could overwrite storage slots used by the child contract, causing storage corruption.

Recommendations: Either add a storage gap variable (`uint256[50] private __gap`) after the existing state variables, or remove the `virtual` keyword from `getStakedNFTs()` if inheritance is not intended.

[L-04] Global reward pool design creates race condition

`NodeKeyPool` uses a single global `lastClaimTimestamp` and distributes all accumulated rewards to the first user who calls `burnAndClaim()` . This creates a winner-takes-all race condition where:

- Rewards accumulate globally over time in a single pool
- The first caller receives all accumulated rewards
- Subsequent callers receive only rewards accumulated since the last claim
- No proportional distribution based on user contribution or ownership

Example scenario: - `secondsPassed` = 100,000 - `rate` = 1e18 per second - `accumulated` = 100,000 * 1e18 = 100,000e18 tokens - 2 users each own 1 NodeKey NFT

User A calls `burnAndClaim()` : - `calculateAccumulated()` = 100,000e18 tokens - Gets 100,000e18 tokens - `lastClaimTimestamp` = `block.timestamp`

User B calls `burnAndClaim()` 10 seconds later: - `calculateAccumulated()` = 10 * 1e18 = 10e18 tokens - Gets only 10e18 tokens

This design creates unfair distribution where timing determines rewards rather than any merit-based system.

Recommendations: Redesign the reward mechanism to use per-user tracking or proportional distribution. Consider tracking individual user contributions or implementing a fair distribution mechanism that doesn't favor the first claimer.

[L-05] Stale data in `tokenToRarityIndex` mapping

The `tokenToRarityIndex` mapping is not cleared when NFTs are unstaked, leading to stale data in this public mapping that can be misleading for external observers.

The mapping tracks rarity index assignments for staked tokens, but when tokens are unstaked, the mapping retains the old values even though the tokens are no longer staked.

```
File: src/staking/NFTStaking.sol
    mapping(uint256 => uint256) public tokenToRarityIndex;
```

Public mapping contains stale data after unstaking, state inconsistency between `nftStaked` and `tokenToRarityIndex` mappings.

Clear the `tokenToRarityIndex` mapping when unstaking NFTs.

[L-06] Lack of Event Emissions in Critical NFTStaking Functions

Several critical functions in the NFTStaking contract lack event emissions, significantly reducing the contract's transparency and off-chain observability. Key state-changing operations such as `_stakeNFTs()` , `_unstakeNFTs()` , `_claimRewards()` , `setRarityWeights()` , and `setRate()` are executed silently without emitting corresponding events. This omission makes it difficult for external systems (like dApp frontends, indexers, or monitoring services) to reliably track contract activity and state changes.

The absence of events forces off-chain systems to rely solely on inefficient and potentially unreliable methods like polling or transaction receipt parsing to detect user actions and administrative changes. This degrades the user experience by delaying updates and increases the complexity of building reliable monitoring tools. For administrative functions like `setRarityWeights()` and `setRate()` , the lack of events makes it challenging for the community to transparently audit governance actions, potentially reducing trust in the protocol's management.

Recommendation: Emit meaningful events for all significant state-changing operations.

[L-07] Unstaking fails if contract lacks sufficient HYBUX balance, causing stuck NFTs

The `_unstakeNFTs` function forces reward claiming before NFT withdrawal, which can cause NFTs to become permanently stuck if the contract doesn't have enough HYBUX tokens to pay accumulated rewards.

```
File: src/staking/NFTStaking.sol
    function _unstakeNFTs(address _sender, uint256[] calldata _tokenIds) internal {
@>      claimRewards();  // Forces reward claiming before unstaking
```

The `claimRewards()` function calls `_claimRewards()` , which attempts to transfer HYBUX tokens:

```
File: src/staking/NFTStaking.sol
    function _claimRewards(
        address _sender
```

```
     ) internal {
         uint256 rewards = calculateRewards(_sender, true);
         if (rewards > 0) {
             pendingRewards[_sender] = 0;
             lastUpdateTimestamp[_sender] = block.timestamp;
@>          hybux.safeTransfer(_sender, rewards);  // Reverts if insufficient balance
         }
     }
```

Impact:

- Users cannot unstake their NFTs if the contract becomes insolvent
- NFTs become permanently stuck in the contract
- Users lose access to their staked assets during low-liquidity periods

Recommendations

Decouple unstaking from reward claiming to ensure users can always recover their NFTs.

[L-08] Stakers can actively abuse rate changes to maximize rewards unfairly

The global `rate` variable can be changed by the owner, but stakers can actively abuse this by timing their reward claims strategically around rate changes. When calculating rewards, the current rate applies to the entire staking period since the user's last update, creating opportunities for manipulation.

```
File: NFTStaking.sol
@>   uint256 public rate;
...
     function setRate(
         uint256 _rate
     ) public onlyOwner {
@>       rate = _rate;
     }
...
     function calculateRewards(address _user, bool _includePendingRewards) public view
returns (uint256) {
...
@>       uint256 timePassed = block.timestamp - lastUpdateTimestamp[_user];
         uint256 totalWeight = getRarityWeightMultiplier(_user);
@>       uint256 accumulated = timePassed * rate * totalWeight;
...
```

Consider the following scenarios:

Rate Decrease Exploitation: 1. Owner announces rate will decrease from 100 to 50 tokens/day 2. Sophisticated stakers immediately claim all rewards at 100/day rate 3. Rate decreases to 50/day 4. Other stakers claiming after the change get reduced rewards 5. Abusers who timed their claims correctly get maximum rewards

Rate Increase Exploitation: 1. Rate increases from 50 to 100 tokens/day 2. Stakers who have been staking during the low-rate period wait to claim 3. When claiming after the increase, they receive 100/day rewards for their entire low-rate staking period 4. This gives them double the rewards they were promised during staking

Recommendations

Implement rate-per-token storage to ensure each NFT earns rewards based on the rate active when it was staked. This prevents strategic timing of claims and ensures fair reward distribution based on rates active during actual staking periods.