expelee

Building the Futuristic Blockchain Ecosystem

SECURITY AUDIT REPORT

SecureNFTStaking

expelee

RISK FINDINGS

Severity	Found
High	2
Medium	3
Low	1
Informational	1

Findings Summary

Severity	Issue	Status
Critical	None	✅
High	Centralization risk due to `adminRecover`(owner can withdraw any NFT), Locking Period	⚠️
Medium	Direct NFT transfers not tracked (admin can recover), State Update After External Call, Index Corruption,	⚠️
Low	Unbounded Array Return	⚠️
Informational	Floating pragma (^0.8.20)	⚠️

TABLE OF CONTENTS

OVERVIEW

The Expelee team has performed a line-by-line manual analysis and automated review of the smart contract. The smart contract was analysed mainly for common smart contract vulnerabilities, exploits, and manipulation hacks. According to the smart contract audit:

Audit Result	**High Risk Detected**
Audit Date	**2 June 2025**

CONTRACT DETAILS

Summary:

The SecureNFTStaking contract allows users to stake and unstake ERC721 NFTs, supports batch operations, admin recovery, and provides human-readable duration queries. The contract uses OpenZeppelin libraries for security and best practices.

Contract Name: SecureNFTStaking

AUDIT METHODOLOGY

Audit Details

Our comprehensive audit report provides a full overview of the audited system's architecture, smart contract codebase, and details on any vulnerabilities found within the system.

Audit Goals

The audit goal is to ensure that the project is built to protect investors and users, preventing potentially catastrophic vulnerabilities after launch , that lead to scams and rugpulls.

Code Quality

Our analysis includes both automatic tests and manual code analysis for the following aspects:

- Exploits
- Back-doors
- Vulnerability
- Accuracy
- Readability

Tools

- Manual Review: The code has undergone a line-by-line review by the Ace team.
- BSC Test Network: All tests were conducted on the BSC Test network, and each test has a corresponding transaction attached to it. These tests can be found in the "Functional Tests" section of the report.
- Slither: The code has undergone static analysis using Slither.

VULNERABILITY CHECKS

Design Logic	Passed
Compiler warnings	Passed
Private user data leaks	Passed
Timestamps dependence	Passed
Integer overflow and underflow	Passed
Race conditions & reentrancy. Cross-function race conditions	Passed
Possible delays in data delivery	Passed
Oracle calls	Passed
Front Running	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Economy model	Passed
Impact of the exchange rate on the logic	Passed
Malicious event log	Passed
Scoping and declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Cross-function race conditions	Passed
Safe Zepplin module	Passed

RISK CLASSIFICATION

When performing smart contract audits, our specialists look for known vulnerabilities as well as logical and acces control issues within the code. The exploitation of these issues by malicious actors may cause serious financial damage to projects that failed to get an audit in time. We categorize these vulnerabilities by the following levels:

High Risk

Issues on this level are critical to the smart contract's performance/functionality and should be fixed before moving to a live environment.

Medium Risk

Issues on this level are critical to the smart contract's performance/functionality and should be fixed before moving to a live environment.

Low Risk

Issues on this level are minor details and warning that can remain unfixed.

Informational

Issues on this level are minor details and warning that can remain unfixed.

OWNER PRIVILEGES

The contract uses OpenZeppelin's Ownable module. The owner (set at deployment) has the following exclusive privileges:

- Pause/Unpause the contract:
  - pause() and unpause() can only be called by the owner to halt or resume staking/unstaking operations.
- Admin Recovery:
  - adminRecover(uint256 tokenId, address to) allows the owner to recover any NFT held by the contract (whether staked or sent directly) and transfer it to a specified address.
- No other privileged functions:
  - All staking and unstaking operations are user-driven and not accessible to the owner unless the owner is also a user.

Security Note:

The owner has significant control, including the ability to pause all user operations and recover any NFT in the contract. Ownership should be secured with a multisig or hardware wallet.

MANUAL REVIEW

Severity Criteria

Expelee assesses the severity of disclosed vulnerabilities according to methodology based on OWASP standarts.

Vulnerabilities are dividend into three primary risk categroies:
High
Medium
Low

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious input handling
- Escalation of privileges
- Arithmetic
- Gas use

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

HIGH RISK FINDING

Centralization – Centralization risk due to adminRecover(owner can withdraw any NFT)

Severity: HIGH

Description:

The adminRecover function allows the contract owner to transfer any NFT held by the contract (including user-staked NFTs or NFTs sent directly to the contract) to any address. This is intended for recovery but gives the owner full control over all NFTs in the contract.

```
function adminRecover(uint256 tokenId, address to) external onlyOwner {
 require(to != address(0), "Invalid address");
 require(
 nftCollection.ownerOf(tokenId) == address(this),
  "Token not in contract"
);

 if (stakedBy[tokenId] != address(0)) {
 _removeStake(stakedBy[tokenId], tokenId);
 }

 try nftCollection.safeTransferFrom(address(this), to, tokenId) {
 emit Recovered(to, tokenId);
 } catch {
 revert("Transfer failed during admin recovery");
}
}
```

Impact: User assets could be at risk of unauthorized withdrawal if the owner account is compromised or acts maliciously.

Recommendation:

- Make users aware of this centralization risk.
- Use a multisig wallet for contract ownership to reduce risk.
- Optionally, restrict adminRecover to only non-staked NFTs or add a time-lock for transparency.

Summary:

The adminRecover function is a necessary tool for asset recovery, but it introduces a medium-severity centralization risk. Users should be made aware, and ownership should be secured appropriately.

HIGH RISK FINDING

Centralization – The owner can Lock the token.

Severity: HIGH

Description:
The owner can pause the contract indefinitely, preventing any user from unstaking.
This can be used to block user withdrawals maliciously or unintentionally (e.g., if the owner is compromised or loses access).

```
function pause() external onlyOwner whenNotPaused {
 isPaused = true;
emit Paused();
}

function unpause() external onlyOwner whenPaused {
isPaused = false;
emit Unpaused();
}
```

Impact:
Pausable controls are legitimate, but full admin control without fallback or timeout carries abuse risk.

Recommendation: It is recommended that there should be a locking period

MEDIUM RISK FINDING

Centralization – Index Corruption in _removeStake()
Severity: Medium

Description:

- This assumption relies on _tokenIndex being accurate.
- If somehow the mapping _tokenIndex[user][tokenId] is wrong (due to storage manipulation or a subtle bug elsewhere), the contract will revert forever on unstake or adminRecover.
- There's no fallback or corrective logic.

```
require(userTokens[index] == tokenId, "Index corruption");
```

Impact:
Medium — A single storage corruption (even rare) could brick unstaking for a user.

Recommendation:
- Add a sanity check and allow recovery logic to bypass or fix index corruption cases.
- Emit an event for admin intervention.

MEDIUM RISK FINDING

Centralization – State Update After External Call

Severity: Medium

Description:

- In the stake and batchStake functions, the contract calls nftCollection.safeTransferFrom (an external call) before updating internal staking state with _addStake. If the external ERC721 contract is malicious, it could attempt reentrancy before the state is updated.
- function stake(uint256 tokenId) external nonReentrant whenNotPaused {

```
address user = msg.sender;

require(nftCollection.ownerOf(tokenId) == user, "Not token owner");
require(
nftCollection.getApproved(tokenId) == address(this) ||
nftCollection.isApprovedForAll(user, address(this)),
"Not approved"
 );
require(stakedBy[tokenId] == address(0), "Already staked");

nftCollection.safeTransferFrom(user, address(this), tokenId);
_addStake(user, tokenId);
}
function batchStake(uint256[] calldata tokenIds)
 external
 nonReentrant
 whenNotPaused
 returns (bool[] memory successList)
 {
require(tokenIds.length <= MAX_BATCH_LIMIT, "Batch too large");
address user = msg.sender;
successList = new bool[](tokenIds.length);
uint256 stakedCount = 0;

for (uint256 i = 0; i < tokenIds.length; i++) {
uint256 tokenId = tokenIds[i];
```

MEDIUM RISK FINDING

```
if (nftCollection.ownerOf(tokenId) != user) {
emit StakeFailed(user, tokenId, "Not token owner");
successList[i] = false;
continue;
}

if (
 nftCollection.getApproved(tokenId) != address(this) &&
!nftCollection.isApprovedForAll(user, address(this))
) {
emit StakeFailed(user, tokenId, "Not approved");
successList[i] = false;
continue;
}

if (stakedBy[tokenId] != address(0)) {
emit StakeFailed(user, tokenId, "Already staked");
successList[i] = false;
continue;
}

try nftCollection.safeTransferFrom(user, address(this), tokenId) {
_addStake(user, tokenId);
successList[i] = true;
stakedCount++;
} catch {
emit StakeFailed(user, tokenId, "Transfer failed");
successList[i] = false;
}
}

 emit StakedBatch(user, stakedCount);
}
```

Impact: This is mitigated by the use of the nonReentrant modifier, but updating state before external calls is considered best practice (checks-effects-interactions pattern).

Recommendation: Consider updating staking state before making external calls, or document why the current approach is safe due to nonReentrant.

MEDIUM RISK FINDING

Centralization – Direct NFT Transfers Not Tracked

Severity: Medium

Description:

If a user sends an NFT directly to the contract (not via stake), it is not tracked in staking mappings and can only be recovered by the admin.

```
function onERC721Received(address, address, uint256, bytes calldata)
    external
    pure
    override
    returns (bytes4)
  {
    return this.onERC721Received.selector;
  }
```

Impact:

- NFT may appear "stuck" to users.

Recommendation:

- Document this behavior for users, or revert in onERC721Received unless called from stake.

LOW RISK FINDING

Centralization – Unbounded Array Return

Severity: Low

Description:

getUserStakedTokens returns the full array of staked tokens for a user.

```
    function getUserStakedTokens(address user)
external
view
returns (uint256[] memory)
{
return _userStakedTokens[user];
}
```

Impact:
If a user has a very large number of staked tokens, the call may run out of gas or exceed RPC limits.

Recommendation:
Consider adding pagination if you expect users to stake many tokens.

INFORMATIONAL FINDING

Centralization – Floating Pragma
Severity: Information

Description:

The contract uses pragma solidity ^0.8.20;.
pragma solidity ^0.8.20;

Impact:
May introduce unexpected behavior if a newer compiler version is used.

Recommendation:
Pin to a specific compiler version (e.g., pragma solidity 0.8.20;).

FUNCTIONAL TESTING

Functional Testing: adminRecover

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../src/SecureNFTStaking.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol";

contract MockNFT is ERC721 {
 constructor() ERC721("MockNFT", "MNFT") {}

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

contract SecureNFTStakingTest is Test {
 SecureNFTStaking public staking;
 MockNFT public nft;
 address public user = address(0xABCD);
 address public recipient = address(0xDCBA);
 address public owner = address(this); // Inherit Test contract is owner

 function setUp() public {
  nft = new MockNFT();
  staking = new SecureNFTStaking(address(nft));

  // Mint and approve token to staking contract
  nft.mint(user, 1);
  vm.prank(user);
  nft.approve(address(staking), 1);
```

FUNCTIONAL TESTING

```
    // Stake the token
    vm.prank(user);
    staking.stake(1);
  }

  function testAdminRecoverAsOwner() public {
    // Admin recovers token from staked user to a recipient
    staking.adminRecover(1, recipient);

    assertEq(nft.ownerOf(1), recipient, "Token not recovered to recipient");
    assertEq(staking.stakedBy(1), address(0), "Token should be unstaked in record");
  }

  function testAdminRecoverFailsForNonOwner() public {
    // Expect revert when non-owner calls adminRecover
    vm.prank(user);
    vm.expectRevert("Ownable: caller is not the owner");
    staking.adminRecover(1, recipient);
  }

  function testAdminRecoverCentralizationRisk() public {
    // Owner forcefully transfers user's staked token
    address attacker = address(0xDEAD);
    staking.adminRecover(1, attacker);

    assertEq(nft.ownerOf(1), attacker, "Admin can force-transfer tokens");
    assertEq(staking.stakedBy(1), address(0), "Token should be removed from staking
records");
  }
}
```

FUNCTIONAL TESTING

Here is a Foundry test for the adminRecover function in your SecureNFTStaking contract, specifically designed to demonstrate and validate the centralization risk — that is, the owner can recover (force-unstake) NFTs from users at will.

This test checks:

1. Only the owner can call adminRecover.
2. adminRecover transfers a staked token to a different address.
3. It bypasses normal unstaking authorization logic.

```
Encountered a total of 1 failing tests, 2 tests succeeded
abhaypatel@MSI-Abhay:/mnt/d/Expelee/Expelee contract/Staking$
```

```
Ran 3 tests for test/testAdminRecover.t.sol:SecureNFTStakingTest
[PASS] testAdminRecoverAsOwner() (gas: 86822)
Traces:
  [88658] SecureNFTStakingTest::testAdminRecoverAsOwner()
    ├─ [95025] SecureNFTStaking::adminRecover(1, 0x0000000000000000000000000000000000000DCbA)
    │   ├─ [2576] MockNFT::ownerOf(1) [staticcall]
    │   │   └─ ← [Return] SecureNFTStaking: [0x2e234DAe75C793f67A35089C9d99245E1C58470b]
    │   ├─ [38792] MockNFT::safeTransferFrom(SecureNFTStaking: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], 0x0000000000000000000000000000000000000DCbA, 1)
    │   │   ├─ emit Transfer(from: SecureNFTStaking: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], to: 0x0000000000000000000000000000000000000DCbA, tokenId: 1)
    │   │   └─ ← [Stop]
    │   ├─ emit Recovered(to: 0x0000000000000000000000000000000000000DCbA, tokenId: 1)
    │   └─ ← [Stop]
    ├─ [576] MockNFT::ownerOf(1) [staticcall]
    │   └─ ← [Return] 0x0000000000000000000000000000000000000DCbA
    ├─ [0] VM::assertEq(0x0000000000000000000000000000000000000DCbA, 0x0000000000000000000000000000000000000DCbA, "Token not recovered to recipient") [staticcall]
    │   └─ ← [Return]
    ├─ [500] SecureNFTStaking::stakedBy(1) [staticcall]
    │   └─ ← [Return] 0x0000000000000000000000000000000000000000
    ├─ [0] VM::assertEq(0x0000000000000000000000000000000000000000, 0x0000000000000000000000000000000000000000, "Token should be unstaked in record") [staticcal
l]
    │   └─ ← [Return]
    └─ ← [Stop]

[PASS] testAdminRecoverCentralizationRisk() (gas: 84480)
Traces:
  [86784] SecureNFTStakingTest::testAdminRecoverCentralizationRisk()
    ├─ [95025] SecureNFTStaking::adminRecover(1, 0x0000000000000000000000000000000000000dEaD)
    │   ├─ [2576] MockNFT::ownerOf(1) [staticcall]
    │   │   └─ ← [Return] SecureNFTStaking: [0x2e234DAe75C793f67A35089C9d99245E1C58470b]
    │   ├─ [38792] MockNFT::safeTransferFrom(SecureNFTStaking: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], 0x0000000000000000000000000000000000000dEaD, 1)
    │   │   ├─ emit Transfer(from: SecureNFTStaking: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], to: 0x0000000000000000000000000000000000000dEaD, tokenId: 1)
    │   │   └─ ← [Stop]
    │   ├─ emit Recovered(to: 0x0000000000000000000000000000000000000dEaD, tokenId: 1)
    │   └─ ← [Stop]
    ├─ [576] MockNFT::ownerOf(1) [staticcall]
    │   └─ ← [Return] 0x0000000000000000000000000000000000000dEaD
    ├─ [0] VM::assertEq(0x0000000000000000000000000000000000000dEaD, 0x0000000000000000000000000000000000000dEaD, "Admin can force-transfer tokens") [staticcall]
    │   └─ ← [Return]
    ├─ [500] SecureNFTStaking::stakedBy(1) [staticcall]
    │   └─ ← [Return] 0x0000000000000000000000000000000000000000
    ├─ [0] VM::assertEq(0x0000000000000000000000000000000000000000, 0x0000000000000000000000000000000000000000, "Token should be removed from staking records")
[staticcall]
    │   └─ ← [Return]
    └─ ← [Stop]
```

CONCLUSION

The SecureNFTStaking contract is generally well-structured and adheres to Solidity best practices, including the use of ReentrancyGuard, Ownable, and safe token transfer mechanisms. However, the contract exhibits several important security considerations:

- The most critical issue is a centralization risk posed by the adminRecover() function, which allows the contract owner to forcibly transfer user-owned NFTs without consent. While useful for recovery scenarios, it grants unilateral power to the owner, which could be abused or misused.
-
- The pause() mechanism is fully controlled by the owner and could be weaponized to indefinitely lock user assets.
- Several medium-risk issues have been identified:
  - Potential index corruption in staking state could lead to users being unable to unstake tokens.
  - The contract allows direct NFT transfers, which are accepted without validation and may result in lost tokens.
  - Poor error transparency in batch operations may cause confusion and hinder issue resolution.

No high-severity bugs such as reentrancy, arithmetic overflow/underflow, or unauthorized access were found beyond owner-controlled functions.

RECOMMENDATİONS

To improve the contract's robustness and decentralization:

- Implement safeguards on admin functions, such as a time delay, multisig, or DAO governance.
- Harden state management logic to defend against index corruption.
- Reject unauthorized direct NFT transfers in onERC721Received().
- Improve error propagation and transparency in batch actions.
- Consider user-level escape mechanisms in case of extended contract pauses.

ABOUT EXPELEE

Expelee is a product-based aspirational Web3 start-up. Coping up with numerous solutions for blockchain security and constructing a Web3 ecosystem from deal making platform to developer hosting open platform, while also developing our own commercial and sustainable blockchain.

www.expelee.com

expeleeofficial

expelee

Expelee

expelee

expelee_official

expelee-co

expelee

Building the Futuristic Blockchain Ecosystem

DISCLAIMER

All the content provided in this document is for general information only and should not be used as financial advice or a reason to buy any investment. Team provides no guarantess against the sale of team tokens or the removal of liquidity by the project audited in this document.

Always do your own research and project yourselves from being scammed. The Expelee team has audited this project for general information and only expresses their opinion based on similar projects and checks from popular diagnostic tools.

Under no circumstances did Expelee receive a payment to manipulate those results or change the awarding badge that we will be adding in our website. Alway do your own research and protect yourselves from scams.

This document should not be presented as a reason to buy or not buy any particular token. The Expelee team disclaims any liability for the resulting losses.