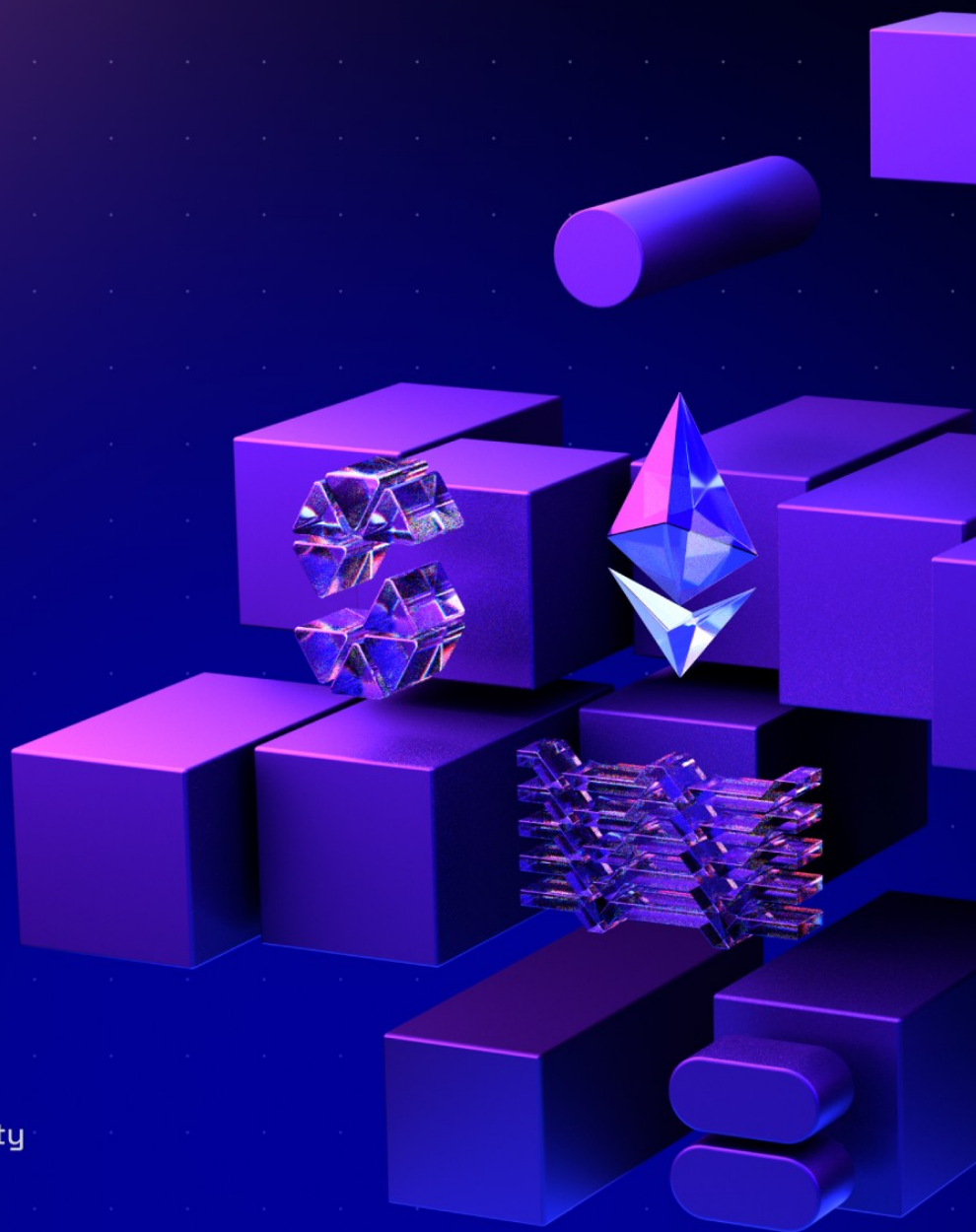
ackee
blockchain security

BitDCA

Staking contracts

18.9.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	03.07.2025
1.0	Final Report	16.07.2025
1.1	Fix Review	15.08.2025
2.0	Final Report	18.09.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.
2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.
3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.
4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.
5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Jan Kalivoda	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

BitDCA is a protocol that enables automatic micro-savings for card payments. Staking contracts is a subcomponent of BitDCA that allows users to stake BDCA tokens and earn rewards.

Revision 1.0

BitDCA engaged Ackee Blockchain Security to perform a security review of BitDCA Staking contracts with a total time donation of 6 engineering days in a period between June 23 and July 3, 2025, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit `c62d3dd`[1] in the private repository and the scope was the following:

- Staking.sol
- StakingNFT.sol

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we involved Wake testing framework. The Staking contract had an integration with an out-of-scope contract (Presale.sol) which was treated as a black box for the review purposes. During the review, we paid special attention to:

- ensuring the arithmetic of the system is correct;
- checking the fairness of the reward distribution;
- ensuring the staking process matches the expected behavior;
- detecting possible reentrancies in the code;
- ensuring access controls are not too relaxed or too strict; and
- looking for common issues such as data validation.

Our review resulted in 24 findings, ranging from Info to High severity. The most severe one is the H2 issue, which causes the reward distribution to be malformed. Several mechanisms in the code need to be completely reworked (such as the internal accounting, or NFT management), thus the codebase is not ready for deployment.

Ackee Blockchain Security recommends BitDCA:

- address all identified issues;
- create documentation for the codebase;
- create a comprehensive test suite; and
- perform another audit of the codebase.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

BitDCA engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

The review was performed between August 14 and August 15, 2025 on the commit `522ad96`[2]. The scope was fixes of the previous revision. All the findings were addressed and the codebase is significantly improved. However, the distribution function is still flawed and instead of Presale contract integration, there was introduced PancakeSwap V2 router for price calculation, which can also cause problems on distribution (see W7 price manipulation issue).

Ackee Blockchain Security recommends BitDCA:

- use oracles for price calculation during rewards distribution;
- define a specification for the distribution function and adjust the logic to match it; and

- perform an audit of the new changes before deployment.

See Report Revision 1.1 for more information about the revision.

Revision 2.0

BitDCA engaged Ackee Blockchain Security to perform a reaudit of BitDCA Staking contracts with a total time donation of 1 engineering days to address unresolved issues from the previous revision. The audit was performed on the commit `c05674c`[3].

The issues were addressed and resolved. It is important to note that the protocol functioning is heavily dependent on the behavior of the privileged roles. Namely, the distribution of rewards and the protocol parameters configuration.

Ackee Blockchain Security recommends BitDCA:

- write a comprehensive test suite; and
- simulate distribution transactions before executing them.

See Report Revision 2.0 for more information about the revision.

[1] full commit hash: `c62d3dda6db241dd4be5088d410971b23db43c5e`

[2] full commit hash: `522ad9645069128cd6fc55258f46478586a40262`

[3] full commit hash: `c05674ce62418572d1fb393efd7fcfe205c93259`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	3	2	6	7	7	25

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Inverted logic in NFT transfer hook	High	1.0	Fixed
H2: The `distributeRewards` function is flawed	High	1.0	Fixed
H3: The project is not compatible with smart accounts	High	1.0	Fixed
M1: Hardcoded Decimal Assumptions	Medium	1.0	Fixed
M2: Stake amount restriction can be bypassed	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
L1: Unsafe ERC20 Operations	Low	1.0	Fixed
L2: Inconsistent Access Control	Low	1.0	Fixed
L3: Max stake amount can be exceeded	Low	1.0	Fixed
L4: Missing Events for Critical State Changess	Low	1.0	Fixed
L5: Missing Pause Modifier on Reward Distribution	Low	1.0	Fixed
L6: Mint function is performing safe mint	Low	1.0	Fixed
W1: Affiliate program integration	Warning	1.0	Fixed
W2: Insufficient Data Validation	Warning	1.0	Fixed
W3: Potential lack of funds	Warning	1.0	Acknowledged
W4: Potential reentrancy due to NFT hook	Warning	1.0	Fixed
W5: Uninitialized variables and roles	Warning	1.0	Fixed
W6: Unknown swap conditions	Warning	1.0	Fixed
I1: Code duplication	Info	1.0	Fixed
I2: Division by Zero in Reward Calculation	Info	1.0	Fixed

Finding title	Severity	Reported	Status
I3: Ambiguous error messages	Info	1.0	Fixed
I4: Use of magic numbers	Info	1.0	Fixed
I5: Missing documentation	Info	1.0	Fixed
I6: Typos	Info	1.0	Fixed
I7: Unused variables	Info	1.0	Fixed
W7: Potential price manipulation on rewards distribution	Warning	1.1	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The protocol implements a staking system with NFT-based positions and tiered rewards. It allows users to lock their BDCA tokens for predefined periods in exchange for bonuses. There is also an option for additional bonus distribution during the staking period in USDT and BDCA.

Trust Model

The admin has excessive powers across all contracts, creating a potential single point of failure. The admin can change critical parameters, pause/unpause at will, modify tier parameters that affect user funds, and withdraw all tokens at any time via the `rescueToken` function. Also the contracts can be upgraded to different implementations.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

H1: Inverted logic in NFT transfer hook

High severity issue

Impact:	High	Likelihood:	Medium
Target:	StakingNFT.sol	Type:	Logic error

Description

In the `_beforeTokenTransfer` hook, the code checks if the receiver has code and sets it as the last wallet owner (`_lastWalletOwners`) if it does.

Listing 1. Excerpt from StakingNFT

```
141 {
142     uint32 size;
143
144     assembly {
145         size := extcodesize(to)
146     }
147
148     if(size > 0) {
149         _lastWalletOwners[firstTokenId] = to;
150     }
151
152     super._beforeTokenTransfer(from, to, firstTokenId, batchSize);
153 }
```

This variable is used in the `currentRealOwner` function. The function checks if the NFT owner has code and returns the last wallet owner if it does.

Listing 2. Excerpt from StakingNFT

```
92 function currentRealOwner(uint256 _tokenId) public virtual view returns
   (address) {
93     uint32 size;
94     address addr = ownerOf(_tokenId);
95
96     assembly {
97         size := extcodesize(addr)
```

```
98      }
99
100     if(size > 0) {
101         return _lastWalletOwners[_tokenId];
102     }
103
104     return addr;
105 }
```

The intended logic tracks EOA holders in case the NFT is temporarily held by a contract. However, the current implementation causes rewards to be distributed to the holding contract instead of the EOA owner. As a result, the `currentRealOwner` function always returns the same value as the `ownerOf` function, which violates the intended logic.

Exploit scenario

Alice, a user, decides to list the NFT representing her staking position on a marketplace while continuing to receive rewards in the meantime. However, due to the inverted hook logic, the rewards are distributed to the marketplace contract instead of Alice's EOA address.

Recommendation

If the intended logic should be preserved, the hook must set the last wallet owner only when the receiver does not have code. However, this logic is also problematic because users with smart accounts will not be eligible for rewards (see H3). If this solution is intended only for EOA holders, it must be clearly communicated to users.

Fix 1.1

The relevant code is removed.

Go back to Findings Summary

H2: The `distributeRewards` function is flawed

High severity issue

Impact:	Medium	Likelihood:	High
Target:	Staking.sol	Type:	Logic error

Description

The `distributeRewards` function contains several logic flaws that can lead to:

- unfair distribution of rewards;
- loss of rewards; and
- denial of service.

First, rewards can only be distributed within specific ranges. These ranges could be very large, either because of the number of participants in staking or because malicious stakers want to prevent rewards from being distributed. Large ranges can lead to denial of service, and rewards must then be distributed in smaller chunks. Depending on the conditions, this can be very costly for the distributor, or some staking participants will not receive any rewards.

Second, another problem arises when the `distributeRewards` function is called multiple times over the same token holders, since there are no updates to the accounting. Before iterating over the token holders, the `usdtTotalShareAmount` is calculated as the balance of USDT in the contract. Then bonuses are calculated based on the available USDT amount and staking ratio. BDCA and USDT are transferred during the iteration without updates to the accounting, and thus the USDT or BDCA amounts can potentially be emptied before distribution completes.

Last but not least, the total staked amount includes expired NFT locks. Once

someone stakes after their lock expires, they will not receive any more rewards but generally cause fewer rewards to be distributed because they own a percentage of the share.

The function needs to be completely reworked.

Listing 3. Excerpt from Staking

```
409 uint256 usdtTotalShareAmount = usdtToken.balanceOf(address(this));
410
411 for(uint256 i = _fromNftId; i <= _tillNftId; i++) {
412     if(_eligibleNftId(i)) {
413         address tokenOwner = stakingNft.currentRealOwner(i);
414         uint256 stakedAmount = nftLockedAmount[i];
415         // calculate both BDCA and USDT bonuses
416         (bdcaBonus, usdtBonus) = calculateBonuses(
417             usdtTotalShareAmount,
418             totalStaked,
419             stakedAmount,
420             nftTier[i]
421         );
422
423         // Distribute BDCA and USDT bonuses to NFT owner
424         if(bdcaBonus > 0) {
425             bdcaToken.safeTransfer(tokenOwner, bdcaBonus);
426         }
427
428         if(usdtBonus > 0) {
429             usdtToken.safeTransfer(tokenOwner, usdtBonus);
430         }
```

Exploit scenario

Bob, the distributor, calls the `distributeRewards` function over a large range of token IDs. The function runs out of gas and reverts, forcing him to call it again with smaller ranges. However, only the first call results in the intended distribution according to the deposited amount. All subsequent calls distribute fewer rewards as the contract's USDT balance decreases.

Recommendation

Rewrite the `distributeRewards` function to use a pull-based design pattern. It will not directly send tokens to all recipients, but instead, it will perform changes in the contract's accounting and users will be able to claim their rewards on their own. This approach will prevent huge gas requirements and potential denial of service. Then it is important to specify the business requirements for the distribution of rewards. According to that, state variables representing the inner state should be created that will allow changing the accounting fairly and efficiently. There are multiple options for making rewards available to users without expired locks. Rewards can be streamed continuously, in a time-based manner, made available for redemption in epochs, or distributed through other mechanisms.

Update 1.1

The `distributeRewards` function has been updated; however, it still contains some flaws. The function continues to loop over the stakeholders, which is inefficient. Additionally, new logic involving rounds has been introduced, but its purpose is unclear. This logic does not prevent double distributions across multiple calls and also allows the total distributed amount in a round to be exceeded through separate distribution amounts.

Fix 2.0

The function is reworked to operate according to the clarified specification and is functioning as intended. However, correct parameter settings by the distributor are now essential to achieve the desired distribution.

Go back to Findings Summary

H3: The project is not compatible with smart accounts

High severity issue

Impact:	High	Likelihood:	Medium
Target:	StakingNFT.sol, Staking.sol	Type:	Logic error

Description

The function for reward distribution uses the `currentRealOwner` function to get the owner of the NFT.

Listing 4. Excerpt from Staking

```
411 for(uint256 i = _fromNftId; i <= _tillNftId; i++) {
412     if(_eligibleNftId(i)) {
413         address tokenOwner = stakingNft.currentRealOwner(i);
```

Listing 5. Excerpt from StakingNFT

```
92 function currentRealOwner(uint256 _tokenId) public virtual view returns
   (address) {
93     uint32 size;
94     address addr = ownerOf(_tokenId);
95
96     assembly {
97         size := extcodesize(addr)
98     }
99
100     if(size > 0) {
101         return _lastWalletOwners[_tokenId];
102     }
103
104     return addr;
105 }
```

The `currentRealOwner` function uses the `_lastWalletOwners` variable to get the

EOA owner of the NFT. However, if the `stake` function is called with an account that has code from the beginning, it can result in the `currentRealOwner` returning the zero address. In that case, once `distributeRewards` is called, it sends rewards to the zero address.

Exploit scenario

Alice, a user, uses a Safe account to stake BDCA. She holds the stake position in the Safe account for years, while not receiving any rewards, because they are sent to the zero address.

Recommendation

Change the way distribution works (see H3) to solve the issue. Otherwise, avoid recognizing EOAs and smart contracts, since after the latest hardfork (EIP-7702), it does not make sense to build logic upon that.

Fix 1.1

The relevant code is removed.

Go back to Findings Summary

M1: Hardcoded Decimal Assumptions

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Staking.sol	Type:	Arithmetics

Description

The `Staking.sol` contract uses hardcoded decimal assumptions for the BDCA and USDT tokens (both 18 decimals). While the BDCA token is custom and deployment is intended only for the BSC network, where USDT also has 18 decimals, this approach is safe. However, these tokens must always have 18 decimals, or they must be correctly handled with different decimals. On other chains, such as Ethereum, USDT commonly has 6 decimals.

Listing 6. Excerpt from Staking

```
408 uint256 usdtBonus;
409 uint256 usdtTotalShareAmount = usdtToken.balanceOf(address(this));
410
411 for(uint256 i = _fromNftId; i <= _tillNftId; i++) {
412     if(_eligibleNftId(i)) {
413         address tokenOwner = stakingNft.currentRealOwner(i);
414         uint256 stakedAmount = nftLockedAmount[i];
415         // calculate both BDCA and USDT bonuses
416         (bdcaBonus, usdtBonus) = calculateBonuses(
417             usdtTotalShareAmount,
418             totalStaked,
419             stakedAmount,
420             nftTier[i]
421         );
```

Exploit scenario

Alice, the contract admin, decides to deploy the Staking contract on the Ethereum network. Since USDT has 6 decimals on Ethereum, the conversion

rate becomes malformed and leads to a loss of funds.

Recommendation

Ensure the tokens always have 18 decimals, or implement correct handling for tokens with different decimals.

Fix 1.1

The USDT token has been replaced with a custom token that can be set arbitrarily; however, the number of decimals for this token is now explicitly specified and handled in the contract during value conversion between the BDCA token and the custom token in the `calculateBonuses` function.

Go back to Findings Summary

M2: Stake amount restriction can be bypassed

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Staking.sol	Type:	Logic error

Description

The `stakeBonusAmount` function does not have any restrictions on the stake amount like the other `stake` functions have.

Listing 7. Excerpt from Staking

```
287 function stakeBonusAmount(address _staker, uint256 _amount, uint8 _tier)
288     external
289     virtual
290     notPaused
291     onlyRole(BONUS_PROVIDER_ROLE)
292 {
293     LockDuration storage currentTierLockDuration = tierLockDuration[_tier];
294
295     require(currentTierLockDuration._years > 0
296         || currentTierLockDuration._months > 0
297         || currentTierLockDuration._days > 0, "Tier does not exist");
298
299     bdcaToken.safeTransferFrom(msg.sender, address(this), _amount);
```

Additionally, it also allows staking within private tiers.

Exploit scenario

Alice, a user, encounters a situation where the maximum stake amount is reached. Alice calls the `stakeBonusAmount` function (through CCIP relayer) and exceeds the maximum amount.

Alternatively, Bob, another user, sets the minimum amount. Bob calls the `stakeBonusAmount` function (through CCIP relayer) with a stake amount equal

to 1 wei. As a result, NFT emission is spammed to prevent the distribution of rewards.

Recommendation

Implement proper stake amount restrictions in the `stakeBonusAmount` function to match the validation applied to other staking functions. Also, if staking to private tiers is not desired, there should be additional restrictions on that.

Fix 1.1

The `_validateStakeParameters` function is implemented and is now consistently called in all staking operations.

Go back to Findings Summary

L1: Unsafe ERC20 Operations

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Staking.sol	Type:	Data validation

Description

The contract uses direct `transferFrom` and `transfer` methods instead of using SafeERC20 library methods consistently. Some ERC20 tokens don't return boolean values or return false on failure, which could lead to silent failures. While SafeERC20 library is imported, it's not used consistently throughout the contract, creating potential for silent transfer failures with non-standard ERC20 tokens.

Listing 8. Excerpt from Staking

```
358 bdcaToken.transferFrom(vaultAddress, address(this), parentBonus);
```

Listing 9. Excerpt from Staking

```
451 require(
452     IERC20Metadata(_token).transfer(msg.sender, balance),
453     "Token transfer failed"
454 );
```

Exploit scenario

Alice, the contract admin, attempts to rescue tokens from the contract using the `rescueToken` function with an ERC20 token that does not return boolean values (like USDT). The direct `transfer` call fails but does not revert the transaction. The contract continues execution believing the transfer succeeded.

Recommendation

Use SafeERC20 library methods consistently throughout the contract instead of direct `transferFrom` and `transfer` calls.

Fix 1.1

The `SafeERC20` library is now used consistently throughout the contract.

Go back to Findings Summary

L2: Inconsistent Access Control

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	StakingNFT.sol	Type:	Access control

Description

The StakingNFT contract implements a custom `owner()` and `transferOwnership()` function alongside AccessControl, creating two parallel permission systems. This inconsistency creates confusion about which permission system to use and potential security gaps.

Listing 10. Excerpt from StakingNFT

```
83 function owner() public view returns(address) {
84     return _owner;
85 }
86
87 function transferOwnership(address newOwner) public virtual
   onlyRole(DEFAULT_ADMIN_ROLE) {
88     require(newOwner != address(0), "Ownable: new owner is the zero
   address");
89     _owner = newOwner;
```

The contract uses both OpenZeppelin's AccessControl (with DEFAULT_ADMIN_ROLE) and a custom ownership pattern, which creates ambiguity about which system controls what functionality and could lead to privilege escalation or access control bypass.

Exploit scenario

Alice, the current owner of the StakingNFT contract, decides to transfer ownership to Bob, a new protocol administrator. She calls `transferOwnership(bob)`, which updates the `_owner` variable to Bob's address.

However, this function does not transfer the `DEFAULT_ADMIN_ROLE` in the AccessControl system, which remains with Alice.

Recommendation

Remove the custom ownership pattern and use only the AccessControl contract.

Fix 1.1

The owner functionality is now explicitly documented as not being used for access control.

Go back to Findings Summary

L3: Max stake amount can be exceeded

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	Staking.sol	Type:	Logic error

Description

The `maxStake` amount is a maximum amount of stake that can be reached. However, the `totalStaked` variable is tracking the total amount of deposited tokens, plus the bonus amount. During deposits, the `maxStake` amount is validated against the new amount passed to the staking contract and not the amount, plus bonuses. As a result, it is possible to exceed the `maxStake` amount. According to current setting of staking tiers it can be up 30%, but it may be more with different tiers.

Listing 11. Excerpt from Staking

```
314 function stake(uint256 _amount, uint8 _tier)
315     external
316     virtual
317     notPaused
318     nonReentrant
319 {
320     require(_amount >= minStake, "Amount is below the minimum stake");
321     uint256 stakingAmountWithBonus = calculateStakeBonus(_amount, _tier);
322
323     require(totalStaked + _amount <= maxStake, "Exceeds max total staked");
```

Listing 12. Excerpt from Staking

```
375     totalStaked += stakingAmountWithBonus;
376
377     emit Staked(msg.sender, stakingAmountWithBonus, lockDuration, tokenId);
378 }
```

Exploit scenario

The `totalStaked` amount is equal to 900 and the max stake amount is 1000. Alice stakes 99 tokens, but instead of setting totalStaked to 990, it is set to 1017 (including bonus). As a result, the `maxStake` amount is exceeded.

Recommendation

Ensure the `maxStake` amount can not be exceeded. Or document intentions of the current behavior.

Fix 1.1

The amount is now properly validated on all entrypoints.

Go back to Findings Summary

L4: Missing Events for Critical State Changess

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	Staking.sol, StakingNFT.sol	Type:	Logging

Description

Many admin functions don not emit events for critical state changes, reducing transparency and making it harder for off-chain monitoring systems to track important protocol changes. Missing events in Staking.sol are in functions such as:

- pause
- unpause
- setMinStake
- setVaultAddress
- updatePresaleAddress
- updateStakingNftAddress
- updateBDCATokenAddress
- updateUSDTTokenAddress
- updateTier
- updateRevenueLanchingStatus
- rescueToken

Then in StakingNFT.sol those are:

- updateStakingAddress
- updateBaseURI

- transferOwnership

Exploit scenario

Alice, the contract admin, calls the `pause` function in Staking.sol. The function does not emit an event for this critical state change.

Recommendation

Add events for critical state changes.

Fix 1.1

The events are now present.

Go back to Findings Summary

L5: Missing Pause Modifier on Reward Distribution

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	Staking.sol	Type:	Logic error

Description

The `distributeRewards` function is not protected by the `notPaused` modifier, allowing it to be called even when the protocol is paused. This allows the flow of funds in case of a pause.

Exploit scenario

Alice, the contract admin, calls the `pause` function in Staking.sol. The protocol is paused paused but a off-chain bot that is responsible for triggering the reward distribution is not aware of the pause and continues to distribute rewards.

Recommendation

Add `notPaused` modifier to the `distributeRewards` function.

Fix 1.1

The `notPaused` modifier is added to the `distributeRewards` function.

Go back to Findings Summary

L6: Mint function is performing safe mint

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	StakingNFT.sol	Type:	Standards violation

Description

The `mint` function in the StakingNFT contract has the same behavior as expected from a `safeMint` function. While it is not defined in the ERC-721 standard, it is expected that the `mint` function skips calling the `onERC721Received` function. In this case, this function is called.

Exploit scenario

Alice, the minting authority, calls the `mint` function where the receiver is a contract that does not have the `onERC721Received` function implemented. As a result, the transaction reverts.

Recommendation

Make a distinction between `mint` and `safeMint` functions.

Fix 1.1

Both functions are now implemented.

Go back to Findings Summary

W1: Affiliate program integration

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol	Type:	Logic error

Description

The `stake` function attempts to retrieve staking bonuses based on the affiliate program in the Presale contract for all deposits above 1000 BDCA.

Since the Presale contract is out of scope, its behavior cannot be exactly verified; however, from the perspective of the Staking contract, it allows bonuses to be obtained for each stake.

This part of the function introduces non-trivial complexity due to the integration with the Presale contract.

Listing 13. Excerpt from Staking

```
339 if(_amount >= 1000 ether) {
340     uint256 affiliatedId = presale.addressToAffiliateId(msg.sender);
341     bool isFounder = presale.isFounder(msg.sender);
342 
343     if(affiliatedId > 0 && !isFounder) {
344         address parentAddress = presale.affiliateIdToAddress(
345 
    presale.affiliateIdToParentAffiliateId(affiliatedId)
346                                 );
347         uint256 parentNftbalance = stakingNft.balanceOf(parentAddress);
348 
349         if(parentNftbalance > 0) {
350             for(uint256 i; i < parentNftbalance; i++) {
351                 uint256 parentTokenId =
    stakingNft.tokenOfOwnerByIndex(parentAddress, i);
352 
353                 if(_eligibleNftId(parentTokenId)) {
354                     uint256 parentBonus =
    tierAffiliateBonusPersentages[_tier] * _amount / TIER_DENOMINATOR;
355                     if(parentBonus > 0) {
356                         nftLockedAmount[parentTokenId] += parentBonus;
```

```
357                         totalStaked += parentBonus;
358                         bdcaToken.transferFrom(vaultAddress, address(this),
    parentBonus);
359                     }
360                     break;
361                 }
362             }
363         }
364     }
365 }
```

Any revert from the Presale contract will revert the staking call and potentially block staking.

Recommendation

Perform an audit of the Presale contract integration or remove the affiliate logic from the Staking contract.

Fix 1.1

The Presale contract is set but is not used anywhere; therefore, this finding is no longer relevant.

Go back to Findings Summary

W2: Insufficient Data Validation

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol	Type:	Data validation

Description

The `Staking.sol` contract does not sufficiently validate the data provided to the admin setter functions. Thorough data validation is important to prevent configuration issues and strengthen trust assumptions toward the protocol.

Exploit scenario

Recommendation

Define reasonable bounds for protocol parameters and apply checks in the setter functions to satisfy them.

Fix 1.1

The data validation is addedd.

Go back to Findings Summary

W3: Potential lack of funds

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol	Type:	Logic error

Description

During deposits, the bonus amount is pulled from the vault. Continuous staking and unstaking is covered until the contract receives USDT and distribution starts. Then, if there is no mechanism for covering BDCA tokens, it can cause a lack of funds for users who want to unstake.

Recommendation

Ensure that after rewards distribution, the contract is liquid enough to cover all unstake operations.

Go back to Findings Summary

W4: Potential reentrancy due to NFT hook

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol	Type:	Reentrancy

Description

The staking contract performs safe minting of StakingNFT during calls to the stake function. This triggers the `onERC721Received` function on recipients and can potentially be exploited to perform reentrancy attacks. The publicly-accessible `stake` function has a `nonReentrant` modifier, so it is not vulnerable in this case, but other stake functions lack this protection, or it can become a problem in future development.

Exploit scenario

Alice, a malicious user, deploys a contract that implements the `onERC721Received` function. When Alice calls a stake function that lacks the `nonReentrant` modifier, the contract mints an NFT to Alice's contract. During the minting process, the `onERC721Received` function is called on Alice's contract, which can then reenter the staking contract and potentially manipulate state before the original transaction completes.

Recommendation

Implement the `nonReentrant` modifier on all stake functions and be aware of this potential attack vector during NFT minting operations.

Fix 1.1

The `nonReentrant` modifier is now used on all stake functions.

Go back to Findings Summary

W5: Uninitialized variables and roles

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol, StakingNFT.sol	Type:	Logic error

Description

Several variables and roles are not initialized in the protocol during initialization. This can lead to a partially functioning protocol, unexpected behavior, or even loss of funds.

The `MINTER` and `BURNER` roles are not initialized in the `StakingNFT` contract. If the `MINTER` role is set but the `BURNER` role is not, users can only stake but not unstake. Additionally, the `PAUSER` role is not explicitly set. In the `Staking` contract, the Vault address is not explicitly set, which can cause the `stake` function to revert.

Exploit scenario

Alice, a user, attempts to interact with the protocol after deployment. Bob, the protocol administrator, has not properly initialized all required variables and roles.

Alice calls the `stake` function in the `Staking` contract and some time passes. At some moment she is able to unstake her tokens. She calls the `unstake` function but it fails. She is not able to unstake.

Recommendation

Ensure that all variables are initialized and all roles are set before users begin interacting with the protocol to guarantee full functionality.

Fix 1.1

The values are now properly initialized.

Go back to Findings Summary

W6: Unknown swap conditions

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol	Type:	Logic error

Description

The `ustdToBDCA` function relies on the Presale (out-of-scope component) contract's price. The swap conditions cannot be verified for correctness and lack proper slippage control. Since the minimum amount expected on output is not passed as a parameter, it is impossible to guarantee from the Staking contract that the swap will be fair.

Listing 14. Excerpt from Staking

```
498 function ustdToBDCA(uint256 _amount) public view returns (uint256) {
499     return presale.paidWithTokenToPresaleToken(_amount);
```

Recommendation

Be aware of this dependency or implement a proper slippage control from the staking contract to guarantee the fair swap.

Fix 1.1

The Presale contract is set but is not used anywhere; therefore, this finding is no longer relevant.

Go back to Findings Summary

I1: Code duplication

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol	Type:	Code quality

Description

The stake functions could share logic in helper functions to prevent code duplication, improve code readability, and avoid issues connected to that.

These lines of code could be extracted into a `_stake` function called by the other functions at the end of execution:

Listing 15. Excerpt from Staking

```
367 uint256 tokenId = _mintNFT(msg.sender);
368 uint256 lockDuration = uint256(block.timestamp)
369                             .addYears(currentTierLockDuration._years)
370                             .addMonths(currentTierLockDuration._months)
371                             .addDays(currentTierLockDuration._days);
372 nftLockExpire[tokenId] = lockDuration;
373 nftLockedAmount[tokenId] = stakingAmountWithBonus;
374 nftTier[tokenId] = _tier;
375 totalStaked += stakingAmountWithBonus;
376
377 emit Staked(msg.sender, stakingAmountWithBonus, lockDuration, tokenId);
```

Additionally, the validation at the beginning of the function can be separated and called in all functions.

Recommendation

Extract the duplicated code into helper functions.

Fix 1.1

The duplicated code is significantly reduced.

Go back to Findings Summary

I2: Division by Zero in Reward Calculation

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol	Type:	Arithmetics

Description

If `_totalStakedBalance` is 0 in the reward calculation, division will revert with no explanatory error message.

Listing 16. Excerpt from Staking

```
485 usdtBonus = _usdtTotalShareAmount * _stakedAmount / _totalStakedBalance;
```

The function doesn't validate that `_totalStakedBalance` is non-zero before performing division.

Recommendation

Add explicit data validation for the variable with an explanatory error message in case it is zero.

Fix 1.1

The `calculateBonuses` function now returns zero for each return value if `_totalStakedBalance` is zero.

Go back to Findings Summary

I3: Ambiguous error messages

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol	Type:	Logging

Description

The error messages in the modifiers for pause and unpause functions reference the presale contract, but the pausing functionality applies to the staking contract.

Listing 17. Excerpt from Staking

```
71 /**
72  * @dev Throws is the presale is paused
73  */
74 modifier notPaused()
75 {
76     require(!paused, "Presale is paused");
77     _;
78 }
79
80 /**
81  * @dev Throws is presale is NOT paused
82  */
83 modifier isPaused()
84 {
85     require(paused, "Presale is not paused");
86     _;
87 }
```

Recommendation

Update the error messages to accurately reflect the staking contract context.

Fix 1.1

The error message is updated.

Go back to Findings Summary

I4: Use of magic numbers

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol	Type:	Code quality

Description

The contracts contain hard-coded values (magic numbers) such as `1000 ether` for the minimum amount that leads to affiliate bonus. This declaration can be misleading since it represents a specific token amount including decimals. A declared constant, such as `MIN_AFFILIATE_AMOUNT` with an explanation, would be clearer in the code.

Listing 18. Excerpt from Staking

```
338 if(_amount >= 1000 ether) {
339     uint256 affiliatedId = presale.addressToAffiliateId(msg.sender);
```

Recommendation

Replace the hard-coded numbers with constant declarations.

Fix 1.1

The magic numbers are replaced with constant declarations.

Go back to Findings Summary

I5: Missing documentation

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol, StakingNFT.sol	Type:	Code quality

Description

The codebase lacks documentation and NatSpec comments for the functions in the contracts.

Recommendation

Add comprehensive documentation of the protocol functionality and NatSpec comments throughout the codebase.

Fix 1.1

The NatSpec comments are added.

Go back to Findings Summary

I6: Typos

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol	Type:	Code quality

Description

The codebase contains typos. While these typos are used consistently, they do not affect the functionality of the contract, but they can cause issues in future development.

Listing 19. Excerpt from Staking

```
498 function ustdToBDCA(uint256 _amount) public view returns (uint256) {
499     return presale.paidWithTokenToPresaleToken(_amount);
500 }
```

Listing 20. Excerpt from Staking

```
54 mapping(uint8 => uint256) public tierBonusPersentages;
55 // Tiers affiliated bonuses
56 mapping(uint8 => uint256) public tierAffiliateBonusPersentages;
```

Recommendation

Fix the typos.

Fix 1.1

The typos are fixed.

Go back to Findings Summary

I7: Unused variables

Impact:	Info	Likelihood:	N/A
Target:	StakingNFT.sol	Type:	Code quality

Description

The `tier` and `lockExpire` variables are retrieved but never used in the `tokenURI` function, causing gas wastage and code clarity issues. The function fetches these values from the staking contract but does not incorporate them into the returned URI.

Listing 21. Excerpt from StakingNFT

```
111 uint256 tier;
112 uint256 lockExpire;
113
114 if(address(staking) != address(0)) {
115     tier = staking.nftTier(tokenId);
116     lockExpire = staking.nftLockExpire(tokenId);
117 }
```

Recommendation

Remove the unused variables.

Fix 1.1

The unused variables are removed.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Revision team is the same as in Report Revision 1.0.

System Overview

The changes in the system are focused on fixes from the previous revision. Most of the codebase was simplified and documented. A PancakeSwap V2 router was introduced for price calculation between a custom token and the BDCA token during rewards distribution. Due to hardcoded values, the codebase is now deployable only on the BSC network. Also, this newly introduced change is out of scope of this revision similarly to anything else without a direct connection to the issues found in the previous revision.

Trust Model

The trust model remained unchanged since the previous revision.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

W7: Potential price manipulation on rewards distribution

Impact:	Warning	Likelihood:	N/A
Target:	Staking.sol	Type:	Logic error

Description

The PancakeSwap V2 Router is introduced to the contract for price calculation between custom tokens and BDCA. The price is used for value conversion to determine how much BDCA rewards should be paid during rewards distribution to stakers.

However, the value conversion is based on the pool price, which can be manipulated depending on pool liquidity. A malicious actor can spot when rewards are being distributed to them and manipulate the pool price to receive more rewards.

Recommendation

Implement reliable price oracles such as TWAP (Time-Weighted Average Price) or Chainlink price feeds instead of relying solely on pool prices. Add price validation mechanisms to detect and prevent price manipulation attempts during reward calculations.

Fix 2.0

The issue is fixed by removing the integration.

Go back to Findings Summary

Report Revision 2.0

Revision Team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The codebase is simplified and resolves issues from the previous revision. However, the simplification places greater requirements on the privileged roles to act responsibly and correctly.

Trust Model

The distributor role is now responsible for selecting the price conversion ratio between the BDCA token and the custom token in the codebase (`_bdcaToCustomTokenRate`). Additionally, the distributor must choose the correct subset of the staked amounts (`eligibleNFTHoldersAmount`) which will be used for calculations in the distribution and distribute to stakers accordingly. If all these parameters are not set together correctly, the distribution will be flawed.

Appendix A: How to cite

Please cite this document as:

> Ackee Blockchain Security, BitDCA: Staking contracts, 18.9.2025.

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz