PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For ApeSwap (NFA Staking)

17 November 2021

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

1 Overview

This report has been prepared for ApeSwap's NFA Staking contract on the Binance Smart Chain (BSC). Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	ApeSwap (NFA Staking)
URL	https://apeswap.finance
Platform	Binance Smart Chain
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match

`NFAStaking (instances)`	`0x0aAd4a7509617c9254B20e565bFcf58B917Fac2c`
	`0xeb6D978c6cA8BF3c2ffaC5536e13021920E4e3E6`
	`0xBabFEce5c014ec5cE5A7Dd36625F942CA0f7E313`
	`0xed66A25F551555e0345635651062806O166f81F9`
	`0x9CAe4d6Fd1Ad948C810D6F1b4Fdbd69716BA50a0`
		MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2	2	-	-
Medium	1	0	-	1
Low	3	2	-	1
Informational	5	4	-	1
Total	**11**	**8**	**-**	**3**

Classification of Issues

Severity	Description
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 NFAStaking

ID	Severity	Summary	Status
01	HIGH	Contract can be drained of all rewards due to a miscalculation of the reward mechanism within the emergency withdraw function	RESOLVED
02	HIGH	`emergencyWithdrawAll` reverts when you have more than 1 NFT in the contract	RESOLVED
03	MEDIUM	Governance can transfer any token balance in the contract to themselves	ACKNOWLEDGED
04	LOW	Governance can adjust rewards in hindsight by setting `setRewardPerBlock`	RESOLVED
05	LOW	Gas use can become unbounded in any function that has to iterate over a user's NFTs	RESOLVED
06	LOW	`setRewardPerBlock` has no maximum safeguard	ACKNOWLEDGED
07	INFO	TIER should be uint256	RESOLVED
08	INFO	Unnecessary computation in critical sections, leading to higher than needed gas usage	RESOLVED
09	INFO	Typographical errors	RESOLVED
10	INFO	`NFT_CONTRACT`, `NFA_TIERS`, `REWARD_TOKEN`, `_rewardPerBlock`, `startBlock` and `bonusEndBlock` are never verified on initialization, leading to an unusable pool with operations that revert	ACKNOWLEDGED
11	INFO	`msg.sender` is unnecessarily cast to `address(msg.sender)`	RESOLVED

2 Findings

2.1 NFAStaking

The NFAStaking contract is a MasterChef contract with only a single pool that allows its users to stake NFTs. Each NFT rarity tier will have a separate NFAStaking contract as each deployment of the contract only allows a specific NFT rarity to be staked. Checking of the NFT's rarity tier is done via another contract which is outside the scope of this audit. The NFAStaking contract still supports the traditional safety features such as emergency withdrawal.

Similar to a Masterchef, pool emissions will be rewarded to the stakers based on the amount of NFTs they have staked. If a user stakes half of the total NFTs within the pool, they will earn half of the emissions.

One notable aspect of the NFAStaking contract is that there is no deposit fee because NFTs are being deposited.

2.1.1 Privileged Roles and Actions

The following functions can be called by the owner of the contract:

- `transferOwnership`
- `renounceOwnership`
- `setBonusEndBlock`
- `setRewardPerBlock`
- `emergencyRewardWithdraw`
- `sweepToken`

2.1.2 Issues & Recommendations

Issue #01	**Contract can be drained of all rewards due to a miscalculation of the reward mechanism within the emergency withdraw function**
Severity	HIGH SEVERITY
Location	Lines 259-270

```
function emergencyWithdraw(uint256[] calldata _ids) external
nonReentrant {
    UserInfo storage user = userInfo[msg.sender];
    for (uint256 i=0; i<_ids.length; i++) {
        require(userHoldings[msg.sender].contains(_ids[i]),
"Unauthorizd id");
        userHoldings[msg.sender].remove(_ids[i]);
        user.amount--;
        totalStaked--;
        NFT_CONTRACT.transferFrom(address(this),
address(msg.sender), _ids[i]);
    }
    user.rewardDebt = 0;
    emit EmergencyWithdraw(msg.sender, user.amount);
}
```

Description

There is an egregious exploit that is possible that results in the draining of all rewards from the contract.

This exploit involves the use of `emergencyWithdraw` and `withdraw` or `deposit` and the manipulation of their inputs. This Masterchef/pool contract is unique in that it lets you stake NFTs instead of regular tokens. The consequence of this is that when depositing and withdrawing, the amount of the token is not specified but instead the IDs of the NFTs have to be specified for either action.

The way we encode the NFTs we want for depositing or withdrawing is in a data structure called an array, which is a contiguous list of elements, in this case the IDs of the NFTs. One unfortunate consequence of this data structure is it is possible to encode an "empty" array – one with no elements.

`emergencyWithdraw` takes this array of NFT IDs to withdraw as the user may just want to withdraw a subset of the NFTs they have staked inside this contract.

The emergency withdrawal logic is as follows: for every NFT ID specified, transfer it back to the sender, and when done, set the `user.rewardDebt` to 0. This logic does not hold when the user is able to choose what gets withdrawn, even specifying "nothing" to withdraw. If the user withdraws "nothing", the `user.rewardDebt` still gets set to 0, even if their NFTs are still staked within the contract.

Now let us turn to the deposit logic: first update the pool, calculate the user's pending rewards and if there are rewards pending, transfer that balance to the user. Next, for each NFT specified, transfer it into the contract. For reference the calculation for the rewards is as follows:

```
uint256 pending = user.amount * poolInfo.accRewardTokenPerShare /
1e30 - user.rewardDebt;
```

Notice the use of `user.rewardDebt` to make sure the user only gets what they are owed. So we know that in `emergencyWithdraw`, the `user.rewardDebt` is set to 0, and we know that `user.rewardDebt` is subtracted from the pending rewards to make sure the user doesn't get more than they are entitled to.

We can now explore the exploit:

- Step1: Deposit any amount of NFTs into the contract, as the amount of NFTs staked must be greater than 0 or rewards will always be 0.
- Step 2: `emergencyWithdraw` passing an empty array to signify withdrawing no NFTs. This will set your `user.rewardDebt` to 0, but leave your NFTs in the contract.
- Step 3: `deposit` specifying an empty array for what you want to deposit. Your rewards will be calculated but `user.rewardDebt` is 0, meaning you will have an amount of rewards greater than 0.
- Step 4: Repeat 2 and 3 until the contract is drained of the reward token balance.

Recommendation

Consider removing `emergencyWithdraw` completely. `emergencyWithdrawAll` should be used instead with a reasonable limit on the amount of tokens a user can deposit. Consider also renaming this all function to `emergencyWithdraw` for third-party tool compliance.

Resolution

RESOLVED

The partial `withdraw` method has been removed in favor of an emergency withdrawal method which withdraws all tokens.

Issue #02	**`emergencyWithdrawAll` reverts when you have more than 1 NFT in the contract**
Severity	HIGH SEVERITY
Location	Lines 273-285

```
function emergencyWithdrawAll() external nonReentrant {
    UserInfo storage user = userInfo[msg.sender];
    uint256 len = userHoldings[msg.sender].length();
    for (uint256 i = 0; i < len; i++) {
        uint256 currentId = userHoldings[msg.sender].at(i);
        userHoldings[msg.sender].remove(currentId);
        user.amount--;
        totalStaked--;
        NFT_CONTRACT.transferFrom(address(this),
address(msg.sender), currentId);
    }
    user.rewardDebt = 0;
    emit EmergencyWithdraw(msg.sender, user.amount);
}
```

Description

Presently, the `emergencyWithdrawAll` function will always revert due to a misimplementation of the iteration logic to remove all the individual NFTs from the user's stored balance. To understand this revert we must understand some of the implementation details of `emergencyWithdrawAll`. Under the hood, the NFTs staked are ultimately stored in an array, a contiguous list of NFT IDs.

The logic to remove these NFTs involves manipulating this array, which is error prone. One common pitfall is to remove things from the array while iterating (going from element to element, start to end).

When removing the NFTs we first see how many we have, the length of the array, then start at the first element and work our way to the last element, getting the element using an array operation called indexing this means specifying a number from 0 to the length of the array minus one (because we start at 0, we cannot go all the way to the length of the array as we would have one extra number, so we subtract 1). If we happen to specify a number that is not in those bounds, the contract will revert.

	Therefore, whenever we visit an element (NFT ID), we remove it from the array via its index, and then we transfer that ID from the contract back to the owner. Now, however, the array is one less in length, and we are still going to iterate like the array hasn't changed, as we computed the length beforehand. This means we will inevitably index the array where no element exists, and this will always revert.
Recommendation	Consider iterating backwards so that removing from the array does not result in indexing out of bounds.
Resolution	RESOLVED The function has been renamed to `emergencyWithdraw` and now iterates from the last index to the first. It should be noted that OpenZeppelin might not have this as a behavioral constraint on the behavior of an enumerable set, so the client must be careful when upgrading to newer or different EnumerableSet versions.

Issue #03	**Governance can transfer any token balance in the contract to themselves**
Severity	MEDIUM SEVERITY
Location	`Lines 288, 298` `emergencyRewardWithdraw and sweepToken`
Description	The contract contains functions to withdraw tokens from the contract. The combination of these functions allows governance to withdraw any token balance from the contract. This could be considered a risk if the governance is untrusted as they could withdraw the reward token and potentially steal or dump it. Furthermore, if governance keys are ever compromised, a malicious third party could do this.
Recommendation	Consider adding a timelock to make this transparent to users, and removing `emergencyRewardWithdraw`.
Resolution	ACKNOWLEDGED

Issue #04	**Governance can adjust rewards in hindsight by setting `setRewardPerBlock`**
Severity	LOW SEVERITY

Location	Lines 251-254
	`function setRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {`
	`    rewardPerBlock = _rewardPerBlock;`
	`    emit LogUpdatePool(bonusEndBlock, rewardPerBlock);`
	`}`

Description	`setRewardPerBlock` does not update the pool before changing the `rewardBlock`. This means that the next update will erroneously use the new value instead of the legacy one. This gives governance the opportunity to adjust `setRewardPerBlock` then update the pool, lowering or increasing the amount of rewards owed retrospectively.
Recommendation	Consider updating the pool prior to setting the `rewardPerBlock` in `setRewardPerBlock` by calling `updatePool()` as the first line of the function. Consider furthermore capping the maximum `rewardPerBlock`.
Resolution	RESOLVED `updatePool` is now called at the start of the `setRewardPerBlock` function.

Issue #05	Gas use can become unbounded in any function that has to iterate over a user's NFTs
Severity	LOW SEVERITY
Description	This contract is novel in that instead of staking an amount of some token, it stakes individual NFTs. To compensate for this, the contract uses a lot of iteration. Gas usage scales poorly with iteration meaning if a user possesses a large number of NFTs the loop can run out of gas locking these NFTs within the contract indefinitely.
Recommendation	Consider adding a limit on the number of NFTs that a user can have staked that results in gas costs that are under the gas limit.
Resolution	RESOLVED Governance can now enforce a maximum number of NFTs to prevent this issue. This limit is initially set to 50.

Issue #06	`setRewardPerBlock` has no maximum safeguard
Severity	LOW SEVERITY

Location	Lines 251-254
	`function setRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {`
	`  rewardPerBlock = _rewardPerBlock;`
	`  emit LogUpdatePool(bonusEndBlock, rewardPerBlock);`
	`}`

Description	Projects sometimes accidentally update their emission rate to a severely high number either by accident or with malicious intent to allow governance to increase it to a large emission rate and then potentially dump these immediately after.
Recommendation	Consider adding a `MAX_EMISSION_RATE` variable and setting it to a reasonable value. `require(_rewardPerBlock <= MAX_EMISSION_RATE,"Too high");`
Resolution	ACKNOWLEDGED

Issue #07	**TIER should be uint256**
Severity	INFORMATIONAL
Location	Line 46 `uint8 public TIER;`
Description	`TIER` is restricted to `uint8` but is passed to `isNfaOfRarityTier` which expects a `uint256`. `uint8` also consumes the same gas and storage on blockchain which makes this more than unnecessary. If the client were to implement gas usage tests, they would notice that moving to `uint256` will even slightly decrease gas usage as it requires less conversion and validation logic.
Recommendation	Consider making `TIER` `uint256`.
Resolution	RESOLVED

Issue #08	**Unnecessary computation in critical sections, leading to higher than needed gas usage**
Severity	INFORMATIONAL
Description	In `deposit` and `withdraw`, the user amount and total staked are computed on every iteration, but they can be derived from the input arrays lengths.
Recommendation	Consider computing `user.amount` and `totalStaked` outside the loop using the lengths of the array.
Resolution	RESOLVED

Issue #09	Typographical errors
Severity	INFORMATIONAL

Location	Line 64
	`// Total allocation poitns. Must be the sum of all allocation points in all pools.`
	
	Lines 209 and 263
	`require(userHoldings[msg.sender].contains(_ids[i]), "Unauthorizd id");`

Description	Some of these typographical errors will also appear in logs on the blockchain.
Recommendation	Consider fixing these errors.
Resolution	RESOLVED

Issue #10	**`NFT_CONTRACT`, `NFA_TIERS`, `REWARD_TOKEN`, `_rewardPerBlock`, `startBlock` and `bonusEndBlock` are never verified on initialization, leading to an unusable pool with operations that revert**
Severity	INFORMATIONAL
Description	When initializing the NFAStaking contract, the addresses for `NFT_CONTRACT`, `NFA_TIERS`, and `REWARD_TOKEN` are not checked to ensure they support their respective interfaces. This could lead to an unusable pool. When initializing the NFAStaking contract, `rewardPerBlock` could start at a high number causing functions of the contract to revert. `startBlock` and `bonusEndBlock` are not validated in the initializer, meaning any erroneous combination of blocks can render the contracts reward system unusable.
Recommendation	Consider validating the above addresses during initialization. Consider validating the `rewardPerBlock` on initialization. Consider validating that `startBlock` is in the future and `bonusEndBlock` is later. Furthermore, on the adjustment function of `bonusEndBlock` it might be valuable to add validation logic as well.
Resolution	ACKNOWLEDGED

Issue #11	**`msg.sender` is unnecessarily cast to `address(msg.sender)`**
Severity	INFORMATIONAL
Description	`msg.sender` is cast to `address(msg.sender)` throughout the contract when used with `safeTransferRewardInternal` and other functions. This is unnecessary.
Recommendation	Consider replacing all occurrences of `address(msg.sender)` with `msg.sender`.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY