Moonbird

smart contracts audit report

Prepared for:

Moonbird

Authors: HashEx audit team

May 2021

Contents

Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn’t say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report.

The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.

Introduction

HashEx was commissioned by the moonbird team to perform an audit of Moonbird's smart contracts. The audit was conducted between May 14 and May 20, 2021.

The audited code is deployed to testnet of Binance Smart Chain (BSC):
`0xCb017F5C69FC1cA370cC0d344919c6f961515BA4` MoonBird.sol,
`0xD2aDA8026a258D624b18a6A4185752C284b4279e` HalvingPool.sol.
No documentation was provided.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts.
- Formally check the logic behind given smart contracts.

Information in this report should be used to understand the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.
We found out that MoonBird token is based on Reflect.finance [1] custom token with an audit report available [2]. HalvingPool is a fork of ApeSwap's RewardApeV2 [3] which is a severely modified version of the MasterChef [4] contract by Sushiswap (audit [5,6]).

Update: Moonbird team has responded to this report. Individual responses to the high severity issues were added after each item in section.

The updated contracts are deployed to the Binance Smart Chain (BSC):
`0x3b23a0ffbc53198d86fa5927e8ee32f7ef699a14` (MBIRD Token),
`0xe536Ca4E42C5c674Cd7C159C49E68de6855f1D0c` (HalvingPool),
`0xE178052a4915fcB3189d2F8abe2b3776EabE1463` (HalvingPool)

Contracts overview

`MoonBird.sol`

Implementation of BEP20 token standard with the custom functionality of auto-yield by burning tokens and distributing the fees on transfers.

`HalvingPool`

The farming contract is similar to MasterChef from Sushiswap but uses a different reward model.

Found issues

ID	Title	Severity	Response
01	HalvingPool: stakes and rewards in the same token	Critical	Fixed
02	excludeAccount() abuse	High	P/fixed
03	for() loop in getCurrentSupply()	High	P/fixed
04	No safeguards on reward parameters	High	P/fixed
05	Inconsistent reward model	Medium	P/fixed
06	BEP20 token standard violation	Medium	Fixed
07	bonusEndBlock interferes with halving	Medium	Fixed
08	LP tokens accumulated to owner	Medium	P/fixed
09	emergencyRewardWithdraw() function	Medium	Fixed
10	Locked ether	Medium	Fixed
11	charity[] accounts model	Medium	Fixed
12	Excluded accounts are free from fees	Low	Informed
13	Excessive conditions in transfer()	Low	Informed
14	takeLiquidity() assumes excluded account	Low	P/fixed
15	sendToCharity() assumes excluded account	Low	P/fixed
16	poolInfo[] is not in use except 1st element	Low	Fixed
17	Variables naming	Low	Fixed
18	Code efficiency of MoonBird contract	Low	P/Fixed
19	General recommendations	Low	P/Fixed

`#01 HalvingPool: stakes and rewards in the same token` Critical

For the same `stakeToken` and `rewardToken` addresses in HalvingPool contract `rewardBalance()` and `totalStakeTokenBalance()` functions return the same amount of total staked tokens plus rewards. Taking into account issue #05 it's possible to withdraw all the funds anytime for the owner, and even an ordinary staker will receive all the rewards regardless of the real reward amount (see #06).

At the time of the audit, HalvingPool contract with the same addresses for stake and reward tokens is deployed to BSC testnet showing the developer's intentions. We recommend delaying product start until this issue is fixed. Otherwise, users should check if the addresses of the tokens differ before entering the staking.

Update: issue was fixed in the update.

`#02 excludeAccount() abuse` High

The owner of the MoonBird.sol contract can redistribute part of the tokens from users to a specific account. For this owner can exclude an account from the reward and include it back later. This will redistribute part of the tokens from holders in profit of the included account. The abuse mechanism can be seen in Appendix C. We suggest lock exclusion/inclusion methods by renouncing ownership.

Response from MoonBird team: MBIRD token ownership is behind a Timelock to prevent such abuse. Furthermore the ability to perform exclusions is key for cases like adding liquidity on a new LP.

Update: ownership of the token was transferred to a Timelock contract.

`#03 for() loop in getCurrentSupply()` High

The mechanism of removing addresses from auto-yielding in MoonBird.sol implies a loop over excluded addresses for every transfer operation or balance inquiry. This may lead to extreme gas costs up to the block gas limit and may be avoided only by the owner restricting the number of excluded addresses. In an extreme situation with a large number of excluded addresses transaction gas may exceed maximum block gas size and all transfers will be effectively blocked. Moreover, `includeAccount()` function relies on the same `for()` loop which may lead to irreversible contract malfunction.

Response from MoonBird team: Sadly most if not all reflect tokens rely on such approach. We plan to put the MoonBird contract ownership behind a timelock.

Update: ownership of the token was transferred to a Timelock contract.

#04 No safeguards on reward parameters High

There are no restrictions on `rewardPerBlock` in HalvingPool.sol L256. The owner of the contract can set the very big reward per block and take the full balance of the contract with the front-run transaction.

Response from MoonBird team: The HalvingPool rewards will never be updated without the communities approval. These are behind a timelock.

Update: ownership of the token was transferred to a Timelock contract.

#05 Inconsistent reward model in HalvingPool.sol Medium

The HalvingPool.sol contract uses the same reward model as the original SushiSwap's MasterChef (`reward ∝ rewardPerBlock * (delta of block.number)`). At the same time the contract's balance is refilled externally and there are no guarantees that it's balance will be equal or greater than the generated reward value.

This may lead to a dangerous situation of inconsistency when only first users withdrawing their rewards will get their expected reward (value from the `pendingReward() function`), while others will get zero rewards.

Update: deployed pools were topped up with MBird tokens to support 7 years. Pools are put behind a timelock, so emission rate can be changed after a minimum delay of 1 day.

#06 BEP20 token standard violation Medium

Implementation of `transfer()` function in MoonBird.sol L251 does not allow to input zero amount as it's demanded in the ERC-20 [6] and BEP-20 [7] standards. This issue may break the interaction with smart contracts that rely on full ERC20 support.

Update: issue was fixed in the update.

#07 bonusEndBlock interferes with halving Medium

HalvingPool.sol reward model includes halving the `rewardPerBlock` every fixed period of blocks. At the same time, `getMultiplier()` function sets all the rewards to zero after the block number of `bonusEndBlock`. We recommend testing the model and set the halving period accurately, without needing the `bonusEndBlock` reward's cancellation.

Update: issue was fixed in the update.

#08 LP tokens accumulated to owner Medium

`addLiquidity()` function in MoonBird.sol L342 calls for `addLiquidityETH()` function of the PancakeSwap Router with the parameter of lp tokens recipient set to owner's address. With time the owner address may accumulate a significant amount of LP tokens which may be dangerous for token economics if an owner acts maliciously or its account gets compromised.

This issue can be fixed by changing the recipient address to the MoonBird contract or by renouncing ownership which will effectively lock the generated LP tokens.

Owner can withdraw all tokens on the contract generated for liquidity by setting a malicious router contract with `updateSwapRouter()` function. This can be secured only by renouncing ownership (router upgrades won't be available) or by implementing governance for setting a new dex router address.

Update: ownership of the token was transferred to a Timelock contract.

#09 emergencyRewardWithdraw() conditions Medium

`emergencyRewardWithdraw()` function in HalvingPool.sol L279 allows the owner to collect the contract's reward pool anytime. Line 280 contains the check of `stakeToken == rewardToken` which always returns false in case of the same token for staking and rewards.

We recommend renouncing ownership as soon as possible.

Update: issue was fixed in the update.

#10 Locked ether Medium

The payable `receive()` function in MoonBird L83 makes it possible for the contract to receive ether/bnb. Moreover, `addLiquidityETH()` from Uniswap or PancakeSwap Router returns the ETH/BNB leftovers back to the sender. There's no implemented mechanism for handling this contract's ETH/BNB balance.

Update: issue was fixed in the update.

#11 charity[] accounts model Low

`setAsCharityAccount()` function in MoonBird contract L222 adds an account to the charity addresses list, but that doesn't mean it'll receive any part of charity fees. However, charity accounts are excluded from paying fees on transfers and there's no way to revert this exclusion. If some account is mistakenly set as a charity account there is no way to set it back. In absence of documentation, we can't ensure if this is the correct behavior or not but exclusion.

Update: issue was fixed in the update.

#12 Excluded accounts are free from fees Low

Unlike the other RFI tokens, excluded from rewards accounts in MoonBird.sol contract L277 are also excluded from paying fees on transfers. In absence of documentation, we can't ensure if this is the correct behavior or not.

Response from MoonBird team: Intended. This allows the halving pools to not charge fees on deposits. Just withdrawals.

`#13  Excessive conditions in transfer()  Low`

`_transer()` function in MoonBird contract L248 contains several excessive code parts, e.g., condition in L259, remove and restore fees functions, 5 cases for 4 different types of transfers.

Update: code was refactored and some excessive code parts were removed.

`#14  takeLiquidity() assumes excluded account  Low`

`_takeLiquidity()` function in MoonBird contract L444 updates `_tOwned[]` balance of the contract itself as if it's excluded from rewards which is not the default state after deployment. Also, it calls `_getRate()` instead of taking its value from parameters.

Update: account was excluded after deployment.

`#15  sendToCharity() assumes excluded account  Low`

`_sendToCharity()` function in MoonBird contract L501 updates `_tOwned[]` balance of the charity address as if it's excluded from rewards which is not the default state after deployment. Also, it calls `_getRate()` instead of taking it's value from parameters.

Update: account was excluded after deployment.

`#16  poolInfo[] is not in use except 1st element  Low`

HalvingPool contract contains `poolInfo[]` array and uses only its first element. Public function `updatePool()` will fail on any `pid` parameter except zero, and `massUpdatePools()` function is useless.

Update: issue was fixed in the update.

`#17  Variables naming  Low`

Variables `stakeToken`, `rewardToken`, `startBlock` of HalvingPool contract should be declared immutable and named in UPPERCASE.

Update: issue was fixed in the update.

`#18  Code efficiency of MoonBird contract  Low`

`getRTransferAmount()` function in MoonBird L479 performs excessive multiplications.

`_charity[]` array in MoonBird contract is not in use.

`removeAllFee()` and `restoreAllFee()` functions of MoonBird contract write 12 state variables for 1 transfer without fees.

`_excluded[]` array in MoonBird contract is used only to calculate the sum of its elements.

Update: some of these issues were resolved in the update.

#19 General recommendations Low

MoonBird contract contains `_getTaxFee()` and `_getMaxTxAmount()` which are not in use. The default value of `numTokensSellToAddToLiquidity` is extremely small compared to max transaction size or total token supply. This will significantly increase the average gas cost of transfers. Pragma version is not fixed, moreover, the used version makes the SafeMath library useless.

Conclusion

Reviewed contracts are deployed to BSC testnet at:

`0xCb017F5C69FC1cA370cC0d344919c6f961515BA4`,

`0xD2aDA8026a258D624b18a6A4185752C284b4279e`,
`0x6C07af861C372Df904AD4c7D096b2A40C3D8F953`.

The audited token contract is a fork of SafeMoon token and based on the Reflect.finance model. Farming contract is based on MasterChef from SushiSwap with a different reward model.

1 critical and 5 high severity issues were found. We recommend fixing it before deployment to mainnet.

Issues number #03, #05, #08, #09 can be mitigated if the owner of the contracts is trusted and uses the contracts with extra care.

The token contract is very dependent on the owner's account as the owner gets all the liquidity generated from the token. Also if the owner account is compromised the attacker can block all token transfers.

Audit includes recommendations on the code improving and preventing potential attacks.

Update: The updated contracts are deployed to the Binance Smart Chain (BSC):
`0x3b23a0ffbc53198d86fa5927e8ee32f7ef699a14` (MBIRD Token),

`0xe536Ca4E42C5c674Cd7C159C49E68de6855f1D0c` (HalvingPool),
`0xE178052a4915fcB3189d2F8abe2b3776EabE1463` (HalvingPool)

Ownership of the token and pools was transferred to a Timelock contract (out of scope of the current audit).

References

1. Reflect.finace github repo
2. Audit report for Reflect.finance
3. ApeSwap BEP20RewardApeV2 staking
4. SushuSwap's MasterChef contract
5. SushiSwap audit by PeckShield
6. SushiSwap audit by Quantstamp
7. ERC-20 standard
8. BEP-20 standard

Appendix A. Issues' severity classification

We consider an issue critical, if it may cause unlimited losses or breaks the workflow of the contract and could be easily triggered.

High severity issues may lead to limited losses or break interaction with users or other contracts under very specific conditions.

Medium severity issues do not cause the full loss of functionality but break the contract logic.

Low severity issues are typically nonoptimal code, unused variables, errors in messages. Usually, these issues do not need immediate reactions.

Appendix B. List of examined issue types

Business logic overview

Functionality checks

Following best practices

Access control and authorization

Reentrancy attacks

Front-run attacks

DoS with (unexpected) revert

DoS with block gas limit

Transaction-ordering dependence

ERC/BEP and other standards violation

Unchecked math

Implicit visibility levels

Excessive gas usage

Timestamp dependence

Forcibly sending ether to a contract

Weak sources of randomness

Shadowing state variables

Usage of deprecated code

Appendix C. Hardhat framework test for possible abuse of excludeAccount()

```
const {expect} = require("chai");
const {formatUnits, parseEther } = ethers.utils;
describe("MoonBird token", function () {
    it("should run exclude include attack", async function () {
        const [owner, alice, bob, charity] = await ethers.getSigners()
        const PancakeFactory = await ethers.getContractFactory("PancakeFactory");
        const factory = await PancakeFactory.deploy(owner.address);
        const initialBalance = parseEther('1000');
        const WETH = await ethers.getContractFactory("WETH9");
        const weth = await WETH.deploy();
        await owner.sendTransaction({ to: weth.address, value: initialBalance})
        const PancakeRouter = await ethers.getContractFactory("PancakeRouter")
        const router = await PancakeRouter.deploy(factory.address, weth.address)
        const MoonBird = await ethers.getContractFactory("MoonBird");
        const token = await MoonBird.deploy(router.address, charity.address);
        const addLiquidityAmount = parseEther('1000');
        await token.approve(router.address, addLiquidityAmount)
        await weth.approve(router.address, addLiquidityAmount)
        await router.addLiquidity(token.address, weth.address, addLiquidityAmount,
addLiquidityAmount, 0, 0,
            owner.address, 1000000000000)
        const decimals = await token.decimals();
        const formatAmount = (amount) => formatUnits(amount, decimals)
        console.log('excluding owner from reward')
        await token.excludeAccount(owner.address)
        let totalSupply = await token.totalSupply();
        await token.transfer(alice.address, totalSupply.div(2))
        console.log(`total supply: ${formatAmount(totalSupply)}`)
        let balance = await token.balanceOf(owner.address)
        console.log(`owner balance is: ${formatAmount(balance)}`)
        const txCount = 200
        console.log(`\nsending ${txCount} maxTxAmount transactions between users`);
        const maxTxAmount = parseEther('420000');
        for(let i = 0; i < txCount; i++) {
            await token.connect(alice).transfer(bob.address, maxTxAmount)
            let bobBalance = await token.balanceOf(bob.address);
            await token.connect(bob).transfer(alice.address, bobBalance);
        }
        balance = await token.balanceOf(owner.address)
        console.log(`owner balance is: ${formatAmount(balance)}`)
        let aliceBalance = await token.balanceOf(alice.address)
        console.log(`alice balance is: ${formatAmount(aliceBalance)}`)
        console.log('\nincluding address back to reward')
        await token.includeAccount(owner.address)
        const newOwnerBalance = await token.balanceOf(owner.address)
        console.log(`owner balance is: ${formatAmount(newOwnerBalance)}`)
        let newAliceBalance = await token.balanceOf(alice.address)
        const aliceLoss = aliceBalance.sub(newAliceBalance)
        console.log(`alice balance is: ${formatAmount(newAliceBalance)}`)
        console.log(`alice loss is: ${aliceLoss.mul(100).div(aliceBalance)}% or
${formatAmount(aliceLoss)} tokens`)
        const ownerProfit = newOwnerBalance.sub(balance)
        console.log(`owner profit is: ${ownerProfit.mul(100).div(balance)}% or
${formatAmount(ownerProfit)} tokens`)
    })
 });
```

Hardhat framework test output

```
  MoonBird token
 excluding owner from reward
 total supply: 21000000.0
 owner balance is: 10499000.0

 sending 200 maxTxAmount transactions between users
 owner balance is: 10499000.0
 alice balance is: 3003107.719426852883946122

 including address back to reward
 owner balance is: 11357176.405871084166432562
 alice balance is: 2708512.089852819139964218
 alice loss is: 9% or 294595.629574033743981904 tokens
 owner profit is: 8% or 858176.405871084166432562 tokens
```