HashEx

SeaChain token

smart contract audit report

Prepared for:

seachaintoken.com

Authors: HashEx audit team

September 2021

Contents

Disclaimer 3

Introduction 4

Contracts overview 4

Found issues 5

Conclusion 10

References 11

Appendix A. Issues' severity classification 12

Appendix B. List of examined issue types 12

Appendix C. Hardhat framework test for possible abuse of excludeContract() 13

Appendix D. Hardhat framework test for issue #05 (Add liquidity from user) 16

Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice 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 SeaChain team to perform an audit of SeaChain smart contracts. The audit was conducted between September 08 and September 10, 2021.

The audited code is deployed at `0xb2fdc382ce3032af0e9dabb8e66f39466c6a1b8b` in Binance Smart Chain (BSC). The very limited documentation was available on SeaChain 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 the SeaChainToken token is based on the SafeMoon token which is a fork of Reflect.finance [1] with an audit report available [2]. There are also audits of the Safemoon.sol contract itself [3, 4].

Update: the SeaChain team has responded to this report. Individual responses were added after each item in the section. Updated contracts are deployed to the BSC:
`0x36b24B2F78725495e858AF9e72f7Df69DaDE3dca` SeaChain token.

Contracts overview

`SeaChain`

Implementation of ERC20 token standard with the custom functionality of auto-yield by burning tokens and distributing the fees on transfers. Fork of SafeMoon token. Fees are distributed between users (5%), automatic addition to liquidity(5%). From the tokens for liquidity fees are taken for charity (20%), marketing (10%), and governance (10%) addresses.

`Ownable`

A heavily modified version of OpenZeppelin's contract.

Found issues

ID	Title	Severity	Response
01	excludeFromReward() abuse	High	Responded
02	excluded[] length problem	High	Responded
03	Missing transferOwnership functionality	High	Fixed
04	ERC20 standard violation	Medium	P/Fixed
05	addLiquidity() recipient	Medium	Responded
06	Add liquidity from user	Medium	Responded
07	Hardcoded addresses	Medium	Fixed
08	Account cannot be unset from devWallet	Medium	Fixed
09	Values do not conform to documentation	Medium	Responded
10	Max tx amount functionality does not work	Medium	Fixed
11	No slippage checks on swaps and adding liquidity	Medium	Responded
12	Wrong calculations for adding liquidity	Low	Fixed
13	inSwapAndLiquify visibility	Low	Fixed
14	Incorrect error message	Low	Fixed
15	Missing events	Low	Fixed
16	Using transfer() function to transfer BNB	Low	Responded
17	Unsafe calculations	Low	Fixed
18	swapAndLiquify may fail on adding zero amounts	Low	Fixed
19	General recommendations	Low	P/fixed

#01 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 22% 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.

Recommendation: create an operator role that can call the `includeInReward()` function and put this role behind a timelock with minimum 3 days delay.

Team response: we will exclude our large locked wallets and never include them again to prevent stolen reflections

#02 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 can 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.

Recommendation: refactor contract code so for() loop won't be used in balance calculations.

Team response: we will manage the # of excluded addresses and keep it low to avoid issues in the future

#03 Missing transferOwnership functionality — High

The Ownable contract misses transfer ownership functionality. If the owner account gets compromised the only way is to renounce ownership.

Recommendation: stick to the original Ownable contract from OpenZeppelin and implement additional functionality through inheritance.

Update: the issue was fixed.

#04 ERC20 standard violation — Medium

Implementation of `transfer()` function (L1068) does not allow to input zero amount as it's demanded in ERC20 [5] and BEP20 [6] 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.
Recommendation: allow zero token transfers.

Update: the issue was partially fixed in terms of zero transfers.

#05 addLiquidity() recipient — Medium

`addLiquidity()` function in L1128 calls for `uniswapV2Router.addLiquidityETH()` function with the parameter of lp tokens recipient set to the `_lockedLiquidity` address. With time the `_lockedLiquidity` address may accumulate a significant amount of LP tokens which may be dangerous for token economics if the owner acts maliciously or their account gets compromised. The owner can change the `_lockedLiquidity` address. Investors should check if the liquidity is actually locked.
Team response: revised the tokenomics blurb, and will provide evidence that liquidity is locked and the lockedLiquidity address points to that token locker

#06 Add liquidity from user — Medium

A transaction, in which a user adds liquidity to the SeaChain/WBNB pair that is connected with the dex router and where auto liquify proceeds can fail. This can happen because the user's WBNB can be used for auto liquify.
This can be reproduced:

1) A user wants to add WBNB and SeaChain into the liquidity
2) User's WBNB are sent to the dex pair
3) Transfer of SeaChain token is called (to send the tokens to the dex pair)
4) Before changing the user's token balance and the dex pair (the actual transfer action) auto liquify on the token contract is called
5) At this point, the user's WBNB are already in the dex pair. Because of that, the token sale and the liquidity addition work incorrectly.
6) The token contract changes the user's balance and the dex pair (actual transfer action)
7) During step 5, the user's WBNB was used in the operations of auto liquify. Because of that, the minting action of LP tokens for the user will work incorrectly and the transaction will be reverted.

Hardhat framework test for that issue can be seen in Appendix D (token code was modified to avoid hardcoded Router address).
Team response: situation seems fairly unlikely, as we will be airdropping liquidity providers LP tokens at the start, so shouldn't be many folks minting LP during the initial period of high volume.

#07 Hardcoded addresses Medium

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

Update: the issue was fixed.

#08 Account cannot be unset from devWallet Medium

Function setAsDevWallet can only set an account as a devWallet account. There is no way to set account as not devWallet if it was added by mistake.

```
function setAsDevWallet(address account) external onlyOwner {
    _isDevWallet[account] = true;
}
```

Recommendation: allow setting account as not dev wallet:

```
function setAsDevWallet(address account, bool isDev) external onlyOwner {
    _isDevWallet[account] = isDev;
}
```

Update: the issue was fixed. devWallet was removed completely.

#09 Values do not conform to documentation Medium

The website states that 40% of tokens are burned and 10% will be burned. At the time of this audit, no tokens were burnt and there is no functionality for automatic burning.
The documentation on the project website and the documentation in the contract differ from the actual values in the code. The documentation in the contract (L6-L10) constitutes the following fees:

```
#SeaChain features:
1% fee auto add to the liquidity providers
1% to community governance
3% fee auto add to charity wallet for Ocean Cleanup
5% fee auto distribute to all holder
```

The actual values are:

- The liquidity fee is 5% on transfers.
- The tax fee is 5% on transfers.
- The charity fee is taken by sending to the charity address 40% of BNB amount to be added as liquidity.
- The marketing fee is taken by sending to the charity address 20% of BNB amount to be added as liquidity.
- The governance fee is taken by sending to the charity address 20% of BNB amount to be added as liquidity.

Update: the percentages were updated but they're still not matching the documentation.

Team response: is an internal issue, we will revise documentation

#10 Max tx amount functionality does not work — Medium

The contract checks for max transaction amount for antiwhale protection. But the maximum amount of transfer is hardcoded to the total supply of the token.

Update: the issue was fixed. Antiwhale protection was removed completely.

#11 No slippage checks on swaps and adding liquidity — Medium

The functions swapTokensForEth() and `addLiquidity()` do not perform slippage checks. The transactions may be frontrunned. This is an architectural decision, but the owner of the token should be aware that if the `numTokensSellToAddToLiquidity` parameter is set to a big value it creates incentives on frontrun attacks.
Team response: We will monitor and manage swapAndLiquify threshold, shouldn't be necessary to do slippage checks

#12 Wrong calculations for adding liquidity — Low

The `swapAndLiquify()` function splits tokens for two parts to be added as liquidity. The second part is then converted to BNB. But then a commission of 80% is taken on the BNB part before the liquidity is added.

Recommendation: take the commission first and then split the rest of the tokens for adding as liquidity.

Update: the issue was fixed.

#13 inSwapAndLiquify visibility — Low

`inSwapAndLiquify` variable in L759 has no explicit visibility.

Update: the issue was fixed.

#14 Incorrect error message — Low

Incorrect error message in L917: must be "Account is already included".

Update: the issue was fixed.

#15 Missing events — Low

Functions deliver(), excludeFromReward(), includeInReward(), excludeFromFee(), includeInFee(), updateNumTokensSellToAddToLiquidity() are missing events for important values change.

Recommendation: we recommend emitting events on changing important values in the contract.

Update: the issue was fixed

#16 Using transfer() function to transfer BNB Low

Use of `address.trasfer()` function for sending BNB is discouraged as it forwards only 2300 gas to the callee which may not be enough if the callee is a contract and it performs some actions in `fallback()` function. L1111, L1140-L1142.

Recommendation: use `address.call()` and check for reentrancy.

Team response: mitigated completely by sending BNB only to team controlled EOAs

#17 Unsafe calculations Low

In L1145 unsafe addition is used.

Recommendation: use `SafeMath` library for all calculations.

Update: the issue was fixed.

#18 swapAndLiquify may fail on adding zero amounts Low

Apeswap router will fail if called with zero balances (call in functions swapTokensForEth() and swapAndLiquify()).

Recommendation: add `if(tokenAmount > 10000)` in swapTokensForEth and `if (ethAmount > 10000)` in the addLiquidity.

Update: the issue was fixed.

#19 General recommendations Low

The 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 the Ethereum mainnet.

We recommend fixing the pragma version to avoid discrepancies in contract behavior compiled with different Solidity versions.

We discourage changing well-tested libraries contracts such as the Ownable contract from OpenZeppelin. Additional functionality should be implemented via inheritance.

Code contains useless condition L1207 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 BNB and to add liquidity to Apeswap.

3 high severity and 6 medium severity issues were found. Parameters set in the contract differ from the website and token documentation.

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.

Update: the SeaChain team has responded to this report. 2 high severity issues remain in the updated code. Individual responses were added after each item in the section. Updated contracts are deployed to the BSC:
`0x36b24B2F78725495e858AF9e72f7Df69DaDE3dca` SeaChain token.

References

1. Reflect.finace github repo
2. Audit report for Reflect.finance
3. SafeMoon audit by CertiK
4. SafeMoon audit by HashEx
5. ERC-20 standard
6. 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()

Test code

Contract was modified to enable setting the router address in the constructor for testing purposes.

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

 describe("SeaChain 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 SeaChain = await ethers.getContractFactory("SeaChain");
        const token = await SeaChain.deploy(router.address);

        const decimals = await token.decimals();
        const formatAmount = (amount) => formatUnits(amount, decimals);

        let totalSupply = await token.totalSupply();
```

```
        const addLiquidityAmountEth = parseEther("1000");
        const addLiquidityAmountToken = totalSupply.div(10);
        await token.approve(router.address, addLiquidityAmountToken);
        await weth.approve(router.address, addLiquidityAmountEth);
        await router.addLiquidity(
            token.address,
            weth.address,
            addLiquidityAmountToken,
            addLiquidityAmountEth,
            0,
            0,
            owner.address,
            1000000000000
        );

        await token.includeInFee(owner.address);
        console.log("excluding owner from reward");
        await token.excludeFromReward(owner.address);

        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 = 20;
        console.log(`\nsending ${txCount} sellCapMinimum transactions between
users`);
        const maxTxAmount = (await token._maxTxAmount()).div(10);
        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`
        );
    });
 });
```

Test output

```
     SeaChain token
  excluding owner from reward
  total supply: 1000000000000.0
  owner balance is: 400000000000.0

  sending 20 sellCapMinimum transactions between users
  owner balance is: 400000000000.0
  alice balance is: 194901987074.610794306
```

```
including address back to reward
owner balance is: 489075463361.795150784
alice balance is: 165967012393.268184557
alice loss is: 14% or 28934974681.342609749 tokens
owner profit is: 22% or 89075463361.795150784 tokens
```

Appendix D. Hardhat framework test for issue #05 (Add liquidity from user)

Test code

Contract was modified to enable setting the router address in the constructor for testing purposes.

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

describe("SeaChain token", function () {
   it("should fail when autoliquify in adding liquidity", async function () {
       const [owner] = 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 Token = await ethers.getContractFactory("SeaChain");
       const token = await Token.deploy(router.address);
```

```
        let totalSupply = await token.totalSupply();

        let addLiquidityAmountToken = totalSupply.div(5);
        await token.approve(router.address, addLiquidityAmountToken);
        await weth.approve(router.address, initialBalance);
        await router.addLiquidity(
            token.address,
            weth.address,
            addLiquidityAmountToken,
            initialBalance,
            0,
            0,
            owner.address,
            1000000000000
        );

        let numOfTokensToAdd = await token.numTokensSellToAddToLiquidity();
        await token.transfer(token.address, numOfTokensToAdd);
        await owner.sendTransaction({ to: weth.address, value: initialBalance
});
        addLiquidityAmountToken = totalSupply.div(5);
        await token.approve(router.address, addLiquidityAmountToken);
        await weth.approve(router.address, initialBalance);

        console.log("Trying to add liquidity where weth token is first...");

        await expect(
            router.addLiquidity(
                weth.address,
                token.address,
                initialBalance,
                addLiquidityAmountToken,
                0,
                0,
                owner.address,
                1000000000000
```

```
            )
        ).to.be.revertedWith("Pancake: INSUFFICIENT_LIQUIDITY_MINTED");

        console.log("Adding liquidity where weth is the first token failed");

        await owner.sendTransaction({ to: weth.address, value: initialBalance
});
        addLiquidityAmountToken = totalSupply.div(5);
        await token.approve(router.address, addLiquidityAmountToken);
        await weth.approve(router.address, initialBalance);

        console.log("Trying to add liquidity where WETH is the second
token...");

        await router.addLiquidity(
            token.address,
            weth.address,
            addLiquidityAmountToken,
            initialBalance,
            0,
            0,
            owner.address,
            1000000000000
        );

        console.log("Adding liquidity where WETH is the second token was
successful");
    });
});
```

Test output

```
  SeaChain token
Trying to add liquidity where weth token is first...
Adding liquidity where weth is the first token failed
Trying to add liquidity where WETH is the second token...
Adding liquidity where WETH is the second token was successful
    ✓ should fail when autoliquify in adding liquidity (1829ms)
```