HashEx

Sombra Token

smart contract audit report

Prepared for:

sombra.app

Authors: HashEx audit team

August 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 fail of addLiquidity() 13

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 Sombra team to perform an audit of their smart contracts. The audit was conducted between August 08 and August 12, 2021.

The audited code was provided in the .sol file without any documentation.

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 Sombra token uses a Safemoon-like architecture with the audit available [1,2] and Reflect.finance model (audit report is also available [3]).

Update: Sombra team has responded to this report. Individual responses were added after each item in the section. The updated code is located in the @SombraNFT/SMBR GitHub repository after the 28aba61 commit. The same updated contracts are deployed to the Binance Smart chain (BSC): `0x44Df6C4Bfa313c3E53E89eE7d4aaa1b8A606BB36` Sombra.

Contracts overview

`Sombra`

Implementation of ERC20 token standard [4] with the custom functionality of auto-yield by burning tokens and distributing the fees on transfers.

`Context, Ownable, Address, SafeMath, and various interfaces`

There are modified versions of OpenZeppelin's contracts (pre 0.4.0 version) and interfaces for dex interacting.

Found issues

ID	Title	Severity	Response
01	updateUniswapRouter() abuse	High	Fixed
02	Add liquidity from user	Medium	Fixed
03	Assert using	Medium	Fixed
04	Discontinuous function for sellCap	Medium	Fixed
05	Owner sellCap abuse	Medium	Fixed
06	Commission changing	Medium	Fixed
07	addLiquidity() recipient	Medium	Responded
08	Old version of OpenZeppelin’s contracts	Low	Fixed
09	Code style	Low	Fixed
10	Insufficient amount of events	Low	Fixed
11	Uninformative error messages	Low	Fixed
12	Excessive variables and computations	Low	Fixed
13	Multiple read from storage	Low	Fixed
14	Wrong visibility	Low	Fixed
15	Excessive external calls	Low	Fixed
16	SellCap vulnerability	Low	Fixed

#01 updateUniswapRouter() abuse — High

The owner of the contract can put the wrong address of the dex router in function `updateUniswapRouter()` L895. This can lead to the failure of transfer functions.

Recommendation: We recommend checking the dex router contract by calling some read functions (like `WETH()` or `factory()`).

Update: the issue was fixed, router update is callable only by Timelock contract with the minimum delay of 2 days.

Commentary on the update: Timelock contract is a modified version of Compound Finance (audit available [6]), but the first admin change doesn't need any delay, see L78.

#02 Add liquidity from user — Medium

A transaction, in which the user adds liquidity to the Sombra/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 Sombra into the liquidity
2) The user's WBNB are sent to the dex pair
3) Transfer of Sombra 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 will 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 C (token code was modified to avoid hardcoded Router address).

Update: the issue was fixed.

#03 Assert using — Medium

In the `recieve()` function L914 there is an `assert` function. The problem is that unlike `require` or `revert` functions, `assert` is burning all remaining gas. For example, If a user tries to send BNB to the contract via Metamask, it will set a maximum amount of gas limit for the transaction (in BSC it is over 80M gas). When said transaction fails, it burns all gas. For BSC with the gas price of 5 gwei this will cost over 0.64 BNB.

Update: the issue was fixed.

#04 Discontinuous function for sellCap — Medium

The formula for calculating the maximum amount of tokens that a user can sell to dex is not continuous. For example, if a user has a balance with `sellCapMinimum` number of tokens then they can sell the whole amount, but if a user has a balance with `sellCapMinimum+1` number of tokens then they can only sell `sellCapPercent` (default value is 10%) of his balance.

Update: the issue was fixed, sellCap functionality was removed.

#05 Owner sellCap abuse — Medium

The owner and the users that don't have commissions on buy/sell (they are determined by the owner) have no restrictions on the amount to sell. They are not affected by SellCap.

Update: the issue was fixed, sellCap functionality was removed.

#06 Commission changing — Medium

There are two types of commissions (percentage to distribute between users and percentage to increase liquidity). The owner can change them by using the `setFees()` function (L859), but they cannot change only one type of commission.

Update: the issue was fixed.

#07 addLiquidity() recipient Low

`addLiquidity()` function in SafeMoon L1146 calls for `uniswapV2Router.addLiquidityETH()` function with the parameter of lp tokens recipient set to owner address. With time the owner address might 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.

Sombra team response: we don't see this as an issue as initial liquidity will be locked, and any liquidity received from the token will be locked at intervals as well. We want to maintain the ability to remove liquidity in the event of upgrading to PancakeSwap V3 when it releases, or migrating to a new exchange.

#08 Old version of OpenZeppelin's contracts Low

This contract uses an old version of OpenZeppelin's contracts (pre 0.4.0). This is not recommended for solidity over 0.8.0.

Update: the issue was fixed.

#09 Code style Low

1) According to the solidity style guide, all constants should be named in UPPER_CASE_WITH_UNDERSCORES style. Such variables as `name` (L669), `symbol` (L670), `decimals` (L672), and `_tTotal` (L681).
2) In calculating the variable `numTokensSellToAddToLiquidity` (L710) there is constant 9, which corresponds to variable `decimals` (L671). But for preventing accidental mistakes this constant should be replaced with variable `decimals`.

Update: the issues were fixed.

#10 Insufficient amount of events Low

For functions `excludeFromReward` (L827), `includeInReward` (L838), `excludeFromFee` (L851) and `includeInFee` (L855) there should be appropriate events.

Update: the issue were fixed.

#11 Uninformative error messages — Low

In functions `_transferStandard` (L1053), `_transferBothExcluded` (L1067), `_transferToExcluded` (L1082) and `_transferFromExcluded` (L1097) there should be `require` statements for checking the balance of the `sender` account with appropriate error messages. It is needed for the user's understanding of why the transaction has failed.

Update: the issue was fixed.

#12 Excessive variables and computations — Low

1) Variables `MAX` (L680) and `_rTotal` (L682) are excessive.
2) In function `_tokenTransfer` (L1039) last else is unnecessary.
3) In functions `_transferStandard` (L1053), `_transferBothExcluded` (L1067), `_transferToExcluded` (L1082) and `_transferFromExcluded` (L1097) there are calls of functions `_takeLiquidity` (L960) and `_reflectFee` (L955) even if the variables `tLiquidity` and `tFee` equal zero.
4) In function `_takeLiquidity` (L960) there is a call of `_getRate` (L944) function. But it is a double call of this function in the transfer transaction. The first one is in the `_getValues` (L919) function. It can be fixed by returning `rLiquidity` in `_getValues` function and passing it in the `_takeLiquidity`.
5) In function `reflectionFromToken` if statement is unnecessary. It could be done without if.

Update: the issues were fixed.

#13 Multiple read from storage — Low

1) In function `_transfer` (L994) state variables `marketPair[from]`, `marketPair[to]`, `_isExcludedFromFee[from]` and `numTokensSellToAddToLiquidity` are read twice.
2) In function `_tokenTransfer` (L1039) state variables `_isExcluded[sender]` and `_isExcluded[recipient]` are read more than once.
3) In function `excludeFromReward` (L827) state variable `_rOwned[account]` is read multiple times and state variable `_tOwned[account]` first written then read.
4) In function `includeInReward` (L838) state variable `_tOwned[account]` is read twice.
5) In function `_getCurrentSupply` (L949) state variable `_tSupply` could be read twice.
6) In functions `swapTokensForEth` (L1129) and `addLiquidity` (1146) state variable uniswapV2Router is read multiple times.

Update: the issues were fixed.

#14 Wrong visibility — Low

1) Function `tokenFromReflection` (L821) can be made private.
2) Function `isExcludedFromFee` (L982) can be made external.
3) Function `_updateMarketPair` can be made private.

Update: the issues were fixed.

#15 Excessive external calls — Low

1) Approve in functions `swapTokensForEth` (L1129) and `addLiquidity` (1146) can be more gas efficient. Approve should be made only when allowance is lower than transfer amount and made for `type(uint256).max` value.
2) In function `swapTokensForEth` (L1129) receiving the WETH address can be more gas efficient. There should be a state variable that stores the WETH address. From this variable the contract should receive the WETH address. This variable is initialized in the constructor and changed in the function `updateUniswapRouter` (L895).

Update: the issues were fixed.

#16 SellCap vulnerability — Low

SellCap implementation has a vulnerability. This limitation could be bypassed by a contract that cells its token balance in cycle in one transaction.

Update: the issue was fixed, sellCap functionality was removed.

Conclusion

1 high severity issue was found. The contracts are highly dependent on the owner's account. Users using the project have to trust the owner and that the owner's account is properly secured.

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

Update: Sombra team has responded to this report. All issues were fixed except one with medium severity (#7 addLiquidity() recipient). The Sombra team responded that the liquidity will be manually locked at intervals. In such a case there will be no serious risks for investors, but users should check that the owner account has not accumulated a significant amount of liquidity. Updated contracts are located in the @SombraNFT/SMBR GitHub repository after the 28aba61 commit. The same updated contracts are deployed to the Binance Smart chain (BSC): `0x44Df6C4Bfa313c3E53E89eE7d4aaa1b8A606BB36` Sombra.

References

1. SafeMoon audit by CertiK
2. SafeMoon audit by HashEx
3. Audit report for Reflect.finance
4. ERC-20 standard
5. BEP-20 standard
6. Timelock audit by OpenZeppelin

Appendix A. Issues' severity classification

We consider an issue to be critical if it may cause unlimited losses or break 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 fail of addLiquidity()

```
describe("Sombra token", function () {
    it("should fail when autoliquify happens during 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 Sombra = await ethers.getContractFactory("Sombra");
        const token = await Sombra.deploy(router.address);

        let totalSupply = await token.totalSupply();
        let addLiquidityAmountToken = totalSupply.div(5);
        let numOfTokensToAdd = 100000000000000;
        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
        );
        await token.transfer(token.address, numOfTokensToAdd);
        await owner.sendTransaction({ to: weth.address, value: initialBalance });
        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 });
        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");
    });
});
```

Hardhat framework test output

```
  Sombra 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 happens during adding liquidity (1650ms)
```