#HashEx
BLOCKCHAIN SECURITY

Swapsy

smart contracts preliminary audit report for internal use only

March 2023

hashex.org

contact@hashex.org

Contents

1. 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.

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. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Swapsy team to perform an audit of their smart contract. The audit was conducted between 05/03/2023 and 08/03/2023.

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 for understanding 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.

The code is available at 0x61bb3D293a90f1a0ece7B4A9ea35DC7c5cBe55bc in the Goerli testnet.

Update. The updated code was rechecked according to deployed contract at 0xbef991010724261dbe8f01692e8ecfa3aad6894d.

2.1 Summary

Project name	Swapsy
URL	https://swapsy.io/
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
Swapsy	0x61bb3D293a90f1a0ece7B4A9ea35DC7c5cBe55bc
Whitelist	
Imports and interfaces	

3. Found issues

High	3 (25%)
Medium	3 (25%)
Low	4 (33%)
Info	2 (17%)

C1. Swapsy

ID	Severity	Title	Status
C1-01	High	Error in revenue calculation	Resolved
C1-02	High	Wrong amount is checked upon creating sell order	Resolved
C1-03	High	Fees are not limited	Partially fixed
C1-04	Medium	Historical price is used for buyer's fee calculation	Acknowledged
C1-05	Medium	Mixed up commission calculation	Resolved
C1-06	Low	Gas optimizations	Partially fixed
C1-07	Low	Lack of events	Resolved
C1-08	Low	SafeERC20 library not used for token transfers	Resolved
C1-09	Low	Lack of checks for sent native currency amount in the sellToken() function	Resolved

C1-10	Info	Confusing error messages	Resolved
C1-11	Info	Typos in code documentation	Resolved

C2. Whitelist

ID	Severity	Title	Status
C2-01	Medium	Lack of re-use protection	Resolved

4. Contracts

C1. Swapsy

Overview

An order book contract for P2P swaps for both native currency and arbitrary ERC20 tokens. Fees are taken both from seller and buyer, fee amount is handled by external contract, which is out of the scope of this audit.

Issues

C1-01 Error in revenue calculation

High Resolved

The internal function `_buy()` calculates and adds a fee to the protocol's revenue even if it should have zero value.

The functions `buyWithToken()` and `buyWithETH()` calculate fees as follows:

```
function buyWithToken(uint256 id) public payable nonReentrant {

    uint256 buyerFee = ISwapsyManager(swapsyManager).getFeeForSeller();
    uint256 buyerFeeAmt = (_allSwaps[id].totalAmountOutEth * buyerFee) / 1000;


    if(ERC721(NFT).balanceOf(msg.sender) > 0){

        buyerFeeAmt = 0;

    } else {

        require(
        msg.value == buyerFeeAmt,
        "Swapsy: incorrect price");
    }
    ...
```

```
        _buy(id);
    }
```

If a user has an NFT, the buyer fee is not taken. But the function `_buy()` aways adds a fee to protocols revenue:

```
        function _buy(uint256 id) internal {
          ...
          uint256 buyerFee = ISwapsyManager(swapsyManager).getFeeForSeller();
          uint256 buyerFeeAmt = (_swaps.totalAmountInEth * buyerFee) / 1000;
          ...
        }
```

Also, in the `buyWithToken()` function the fee is calculated from `totalAmountOutEth` value, but in the `_buy()` function it's calculated from `totalAmountInEth`.

Recommendations

Pass the fee as a parameter to the `_buy()` function.

C1-02 Wrong amount is checked upon creating sell order

High Resolved

The function `sellETH()` checks the transferred amount sent to the contract as `_totalAmountInEth`.

```
    function sellETH(
        bytes32 amountIn,
        bytes32 amountOut,
        address tokenIn,
        address tokenOut,
        bytes32 totalAmountInEth,
        bytes32 totalAmountOutEth,
        bytes32 swapId,
        uint256 timeout,
        uint256 deadline,
        bytes memory _signature
```

```
    ) public payable nonReentrant isSenderWhitelisted(amountIn, amountOut, tokenIn,
tokenOut, totalAmountInEth, totalAmountOutEth, swapId, timeout, _signature)
    {
      ...
        if(ERC721(NFT).balanceOf(msg.sender) > 0){
            sellerFeeAmt = 0;
            require(_totalAmountInEth + sellerFeeAmt == msg.value, "Swapsy: Only ETH");
        } else {
            require(_totalAmountInEth + sellerFeeAmt == msg.value, "Swapsy: ETH + FEE");
        }
    }
```

The `amountIn` parameter should be checked against the `msg.value`. If the signer account is compromised, an attacker can create a sell order with a big `amountIn` and small `totalAmountInEth` values and then withdraw funds from the contract with the `cancel()` function which returns `amounIn` to the seller.

Recommendation

Fix the `sellETH()` to check the sent amount against `amountIn` parameter.

C1-03 Fees are not limited

High Partially fixed

The contract takes fees from the seller and the buyer, but the amount of fees is not checked for upper limits.

```
    function sellToken(
        bytes32 amountIn,
        bytes32 amountOut,
        address tokenIn,
        address tokenOut,
        bytes32 totalAmountInEth,
        bytes32 totalAmountOutEth,
        bytes32 swapId,
        uint256 timeout,
        uint256 deadline,
        bytes memory _signature
    ) public payable nonReentrant isSenderWhitelisted(amountIn, amountOut, tokenIn,
```

```
tokenOut, totalAmountInEth, totalAmountOutEth, swapId, timeout, _signature)
    {

        uint256 _amountIn = bytes32ToString(amountIn);
        uint256 _amountOut = bytes32ToString(amountOut);
        uint256 _totalAmountInEth = bytes32ToString(totalAmountInEth);
        uint256 _totalAmountOutEth = bytes32ToString(totalAmountOutEth);

        require(timeout > block.timestamp, "Swapsy: signature expired");
        require(deadline > block.timestamp, "Swapsy: past timestamp");
        require((_amountIn > 0) && (_amountOut > 0), "Swapsy: zero I/O amount");

        uint256 sellerFee = ISwapsyManager(swapsyManager).getFeeForSeller();
        uint256 sellerFeeAmt = (_totalAmountInEth * sellerFee) / 1000;
        ...
    }
```

The returned value of the `getFeeForSeller()` should be checked for max/min limits in the `sellETH()`, `sellToken()`, `buyWithETH()`, and `buyWithToken()` functions.

Recommendation

Set limits for the fees and check that they are not exceeded in the Swapsy contract. Wrap the checks in `try-catch`.

Update

The external non-view call to `swapsyManager` was moved to try-catch section, but actual fee amount is read with an additional secondary call to the same address. If the `swapsyManager` contract returns different values for second of consecutive calls, it would be used without checking for max value.

C1-04 Historical price is used for buyer's fee calculation

● Medium ⊘ Acknowledged

The assessment of the trade amount in ETH is saved in the `totalAmountOutEth` function during the creation of the sell order. At the time of buying this amount may be completely different due to the asset price change.

Recommendation

Do not save the `totalAmountOutEth` upon the sell creation but pass it in the buy transaction and validate it with a signature.

C1-05 Mixed up commission calculation

Medium Resolved

The functions `buyWithETH()` and `buyWithToken()` use the `ISwapsyManager(swapsyManager).getFeeForSeller()` function to calculate the buyer fees. The ISwapsyManager has an interface for the buyer fees and it should be used.

```
function buyWithETH(uint256 id) public payable nonReentrant {

    uint256 buyerFee = ISwapsyManager(swapsyManager).getFeeForSeller();
    ...
}

function buyWithToken(uint256 id) public payable nonReentrant {

    uint256 buyerFee = ISwapsyManager(swapsyManager).getFeeForSeller();
    ...
}
```

Recommendation

Use `getFeeForBuyer()` from the SwapsyManager to calculate the buyer fees.

C1-06 Gas optimizations

Low Partially fixed

1. `SWAPS` struct could be optimized by combining the address with enum in a single storage slot.

2. `totalAmountInEth`, `totalAmountOutEth`, and timeout parameters of the SWAPS structure are not used and should be removed.

3. Ineffective casting `bytes32` to `uint256` in the `sellETH()` and `sellToken()` functions.

`bytes32ToString()` and `stringToUint()` should be removed.

4. External call for fee amount should be moved inside if clause in the `sellETH()`, `sellToken()`, `buyWithETH()`, and `buyWithToken()` functions.

5. The ERC20 transfer requirement should be moved outside the if-else clause in the `sellToken()` functions to reduce deployed bytecode.

6. The deadline requirement is redundant in the `_sell()` function, as it is already checked both in the `sellETH()` and `sellToken()` functions.

7. Multiple reads from storage of the totalSwaps variable in the `_sell()` function. Use memoization for gas savings.

8. Double read from storage of the revenue variable in the `_buy()` function.

9. Unnecessary external call for swapsyManager in the `_buy()` function, as `buyerFeeAmt` is already calculated in the parent function.

10. Multiple reads from storage of `_swapsByUser[user].length` and `_swapsByUser[user][i]` in the `getSwapsByUser()` function.

11. Multiple reads from the storage of `_swapsBySwapId[swapId].length` and `_swapsBySwapId[swapId][i]` in the `getSwapsBySwapId()` function.

12. Double read from the storage of the revenue variable in the `withdraw()` function.

C1-07 Lack of events

Low Resolved

The functions `setSwapsyManager()` and `updateNft()` don't emit events, which complicates the tracking of important changes off-chain.

C1-08 SafeERC20 library not used for token transfers

Low Resolved

Token checks made upon token transfers may fail for certain types of tokens, e.g. USDT in Ethereum which does return a boolean on transfers.

```
            require(
                IERC20(_swaps.tokenIn).transfer(msg.sender, _swaps.amountIn),
                "Swapsy: cancellation failed"
            );
            (bool sent,) = msg.sender.call{value: _swaps.sellerFeeEth}("");
```

Recommendation

Use OpenZeppelin's SafeERC20 library to handle token transfers.

C1-09 Lack of checks for sent native currency amount in the sellToken() function

Low Resolved

The function `sellToken()` validates the amount of sent native currency only if the sender doesn't hold NFT for the fee discount.

```
    function sellToken(
        bytes32 amountIn,
        bytes32 amountOut,
        address tokenIn,
        address tokenOut,
        bytes32 totalAmountInEth,
        bytes32 totalAmountOutEth,
        bytes32 swapId,
        uint256 timeout,
        uint256 deadline,
        bytes memory _signature
    ) public payable nonReentrant isSenderWhitelisted(amountIn, amountOut, tokenIn,
tokenOut, totalAmountInEth, totalAmountOutEth, swapId, timeout, _signature)
    {
      ...
      if(ERC721(NFT).balanceOf(msg.sender) > 0){
```

```
            sellerFeeAmt = 0;
            require(IERC20(tokenIn).transferFrom(msg.sender, address(this), _amountIn),
"TOKEN Transfer issue");

        } else {

            require(IERC20(tokenIn).transferFrom(msg.sender, address(this), _amountIn),
"TOKEN Transfer issue");
            require(sellerFeeAmt == msg.value ,"Swapsy: ETH - Platform Fee wrong");
        }
        ...
    }
```

Recommendation

Add a `require` check for `msg.value == 0` in the first if clause.

C1-10 Confusing error messages

Info Resolved

If a wrong amount is sent in `buyWithEth()` and `buyWithToken()` functions, an error appears with a message that says "incorrect price". We recommend changing the message to "incorrect amount sent".

We also recommend changing the "only for token price" in the `buyWithToken()` function to "only for buying with a token"

C1-11 Typos in code documentation

Info Resolved

Typos reduce the code's readability. Typos in 'handelled', 'licence'.

C2. Whitelist

Overview

An authorization model that uses EIP712 signatures to grant user access to the creation of the order.

Issues

C2-01 Lack of re-use protection

Medium Resolved

A once-created signature can be reused until the timeout timestamp. The sender's address is not included in the signature's digest, meaning anyone can re-use generated signature once it's on-chain or in a failed transaction.

```
    modifier isSenderWhitelisted(
        bytes32 amountIn,
        bytes32 amountOut,
        address tokenIn,
        address tokenOut,
        bytes32 totalAmountInEth,
        bytes32 totalAmountOutEth,
        bytes32 swapId,
        uint256 timeout,
        bytes memory _signature
    ) {
        require(
            getSigner(amountIn, amountOut, tokenIn, tokenOut, totalAmountInEth,
totalAmountOutEth, swapId, timeout, _signature) ==
                whitelistSigner,
            "Whitelist: Invalid signature"
        );
        _;
    }

    function getSigner(
        bytes32 amountIn,
```

```
        bytes32 amountOut,
        address tokenIn,
        address tokenOut,
        bytes32 totalAmountInEth,
        bytes32 totalAmountOutEth,
        bytes32 swapId,
        uint256 timeout,
        bytes memory _signature
    ) public view returns (address) {
        bytes32 digest = _hashTypedDataV4(
            keccak256(
                abi.encode(WHITELIST_TYPEHASH, amountIn, amountOut, tokenIn, tokenOut,
totalAmountInEth, totalAmountOutEth, swapId, timeout)
            )
        );
        return ECDSA.recover(digest, _signature);
    }
```

Recommendation

Consider including `msg.sender` into the `_hashTypedDataV4()` and introducing signature reuse protection via nonces or mapping of used hashes.

C3. Imports and interfaces

Overview

Strings, ECDSA, EIP712, Context, Ownable, ReentrancyGuard, and IERC20 contracts are imported from the OpenZeppelin library.

ERC721 is an interface for the EIP721 token standard.

ISwapsyManager is a limited interface for the fee management contract, containing only 2 external view functions: getFeeForSeller() and getFeeForBuyer().

No issues were found.

5. Conclusion

3 high, 3 medium, 4 low severity issues were found during the audit. 2 high, 2 medium, 3 low issues were resolved in the update.

The reviewed 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.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

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

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY