#HashEx
BLOCKCHAIN SECURITY

Chainspot

smart contracts final audit report

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). HashEx has exclusive rights to publish the results of this audit on company's web and social sites.

2. Overview

HashEx was commissioned by the **Chainspot** team to perform an audit of their smart contract. The audit was conducted between **2023-03-15** and **2023-03-19**.

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 @Chainspot-router/contracts GitHub repository and was audited after the commit eb91ead.

Update. Recheck was made after the commit 17fb73.

2.1 Summary

Project name	Chainspot
URL	https://chainspot.io/
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
ChainspotProxy	
ProxyFee	
ProxyWithdrawal	

3. Found issues

Critical	1 (8%)
High	1 (8%)
Medium	2 (17%)
Low	5 (42%)
Info	3 (25%)

C1. ChainspotProxy

ID	Severity	Title	Status
C1-01	Critical	An attacker can steal users' tokens approved on the contract	Resolved
C1-02	Medium	Lack of msg.value check in proxyTokens function	Resolved
C1-03	Medium	Lack of reentrancy checks	Resolved
C1-04	Low	Excessive usage of low-level calls	Resolved
C1-05	Low	Gas optimizations	Partially fixed
C1-06	Low	Possible unspent allowance	Resolved
C1-07	Low	Result of ERC20 token transfer is not checked	Resolved
C1-08	Info	Lack of in-code documentation	Resolved
C1-09	Info	Typos	Partially fixed

C1-10	Info	Unclear purpose of the receive() function	Acknowledged

C2. ProxyFee

ID	Severity	Title	Status
C2-01	High	No fees constraints	Resolved
C2-02	Low	Lack of events	Resolved

4. Contracts

C1. ChainspotProxy

Overview

The main contract sends arbitrary messages on its behalf.

Issues

C1-01 An attacker can steal users' tokens approved on the contract

Critical Resolved

The contract allows sending arbitrary messages on its behalf with the `metaProxy()` function.

```
function metaProxy(address tokenAddress, address approveTo, address callDataTo, bytes
memory data) external payable {
    ...
    proxyCoins(callDataTo, data);
    ...
}

function proxyCoins(address to, bytes memory data) internal {
        ...
        (success, ) = to.call{value: resultAmount}(data);
        require(success, "Proxy: transfer not sended");

        emit ProxyCoinsEvent(to, amount, resultAmount, feeAmount);
    }
```

The function `proxyTokens()` assumes that users approve their ERC20 tokens for the ChainspotProxy contract to interact with it.

```
    function proxyTokens(address tokenAddress, address approveTo, address callDataTo,
bytes memory data) internal {
        address selfAddress = address(this);
```

```
        address fromAddress = msg.sender;

        (bool success, bytes memory result) = tokenAddress.call(
            abi.encodeWithSignature("allowance(address,address)", fromAddress,
selfAddress)
        );
        require(success, "Proxy: allowance request failed");
        uint amount = abi.decode(result, (uint));
        require(amount > 0, "Proxy: amount is to small");
        ...
  }
```

In case some tokens are approved, an attacker can transfer all approved tokens to his wallet. To do this an attacker calls the `metaProxy()` function and passes the address of the token in the `callDataTo` parameter and forms the `data` parameter to call the token's `transferFrom()` function to transfer tokens from the user to the attacker.

Recommendation

Create a whitelist of the contracts that can be called. Check if these contracts don't have any privileged access for the router contract.

C1-02 Lack of msg.value check in proxyTokens function

Medium Resolved

The function `proxyTokens()` doesn't check if the native currency value sent with the transaction is zero. The value accidentally sent with the call native currency will remain on the contract's balance. Only the contract owner can withdraw the native currency from the contract.

C1-03 Lack of reentrancy checks

Medium Resolved

The contract calls external addresses and changes state after calls.

```
    function proxyCoins(address to, bytes memory data) internal {
        ...
```

```
        (success, ) = to.call{value: resultAmount}(data);
        require(success, "Proxy: transfer not sended");

        emit ProxyCoinsEvent(to, amount, resultAmount, feeAmount);
    }
```

```
    function proxyTokens(address tokenAddress, address approveTo, address callDataTo,
bytes memory data) internal {
      ...
        (success, ) = callDataTo.call(data);
        require(success, "Proxy: call data request failed");

        emit ProxyTokensEvent(tokenAddress, amount, routerAmount, feeAmount, approveTo,
callDataTo);
    }
```

Recommendation

Use ReentrancyGuard from OpenZeppelin's library to avoid potential reentrancy attacks.

C1-04 Excessive usage of low-level calls

Low Resolved

The contract uses low-level calls to interact with tokens.

```
    function proxyTokens(address tokenAddress, address approveTo, address callDataTo,
bytes memory data) internal {
        ...

        (bool success, bytes memory result) = tokenAddress.call(
            abi.encodeWithSignature("allowance(address,address)", fromAddress,
selfAddress)
        );
        require(success, "Proxy: allowance request failed");
        uint amount = abi.decode(result, (uint));
        require(amount > 0, "Proxy: amount is to small");
        ...
    }
```

An ERC20 interface could be used to simplify interactions with tokens.

Recommendation

Use the IERC20 interface to interact with ERC20 tokens.

C1-05 Gas optimizations

Low Partially fixed

1. Use `calldata` instead of `memory` for function parameters where possible.

2. Fee is calculated twice in the `calcAmount` and `calFee` functions.

3. No need to check for token allowance in the `proxyTokens()` function.

4. Fees are transferred to the owner on every call. Pull payment can be used to save gas.

C1-06 Possible unspent allowance

Low Resolved

The function `proxyTokens()` approves tokens for the `approveTo` address. If this allowance isn't spent in the transaction, the tokens remain on the contract, and the owner of the `approveTo` account will be able to transfer tokens to himself. This situation is possible in case wrong parameters are passed to the function, but it's always good to be fail-proof.

Recommendation

Check if the allowance given in the transaction is spent.

C1-07 Result of ERC20 token transfer is not checked

Low Resolved

The ERC20 standard states that the transfer function should return true after a successful transfer. The code does not check if the returned value is false.

Recommendation

User OpenZeppelin's SafeERC20 library to handle token transfers.

Update

require statements were added to check if the token returned false on transfer. However, it's important to note that such checks will fail on successful transfers on tokens that do not fully comply with the ERC20 standard like USDT in Ethereum, and don't return any value.

C1-08 Lack of in-code documentation

Info Resolved

The contract has no Natspec documentation. We recommend adding documentation to at least all `public` and `external` functions.

C1-09 Typos

Info Partially fixed

Typos reduce the code's readability.1) 'amount is to small' should be replaced with 'amount is too small'2) 'not sended' should be replaced with 'not sent'

Update

New typos were introduces with the update: 'refert' in a require message.

C1-10 Unclear purpose of the receive() function

Info Acknowledged

The usage of the `receive()` function is unclear. It opens the possibility to send native currency to the contract. Only the owner of the contract can withdraw the contract's balance. The function has no documentation and its purpose is unclear.

C2. ProxyFee

Overview

A contract with fee calculation functions. Inherited by the ChainspotProxy contract.

Issues

C2-01 No fees constraints

High Resolved

There are no constraints on the amount of the fees. The contract owner can set arbitrary bit fees.

```
    function setFeeBase(uint _feeBase) public onlyOwner {
        require(_feeBase > 0, "Fee: feeBase must be valid");

        feeBase = _feeBase;
    }

    function setFeeMul(uint _feeMul) public onlyOwner {
        require(_feeMul > 0, "Fee: feeMul must be valid");

        feeMul = _feeMul;
    }
```

Recommendation

Set a maximum percent for the fees, e.g. 10%, and do not allow setting fees that exceed the threshold.

C2-02 Lack of events

Low Resolved

The function `setFeeBase()` , `setFeeMul()` don't emit events, which complicates the tracking of important changes off-chain.

C3. ProxyWithdrawal

Overview

A helper contract with interfaces for allowing the contract owner to withdraw native currency and ERC20 tokens to specified addresses. Inherited by the ChainspotProxy contract.

5. Conclusion

1 critical, 1 high, 2 medium, 5 low severity issues were found during the audit. 1 critical, 1 high, 2 medium, 4 low issues were resolved in the update.

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.
- **Info.** Issues that do not impact the contract operation. Usually, info severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. 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