MSHASH

Smart Contract Review

Deliverable: Smart Contract Audit Report

Security Report Aug 2022

Disclaimer

The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the Company. The content, conclusions and recommendations set out in this publication are elaborated in the specific for only project.

HireCA does not guarantee the accuracy of the data included in this study. All representations, warranties, undertakings and guarantees relating to the report are excluded, particularly concerning – but not limited to – the qualities of the assessed projects and products. Neither the Company nor any personating on the Company's behalf may be held responsible for the use that may be made of the information contained herein. HireCA retains the right to display audit reports and other content elements as examples of their work in their portfolio and as content features in other projects with protecting all security purpose of customer. The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities fixed - upon a decision of the Customer.

Overview

Background

MSHASH requested that HireCA perform an Extensive Smart Contract audit of their Smart Contract.

Project Dates

The following is the project schedule for this review and report:

Aug 26 : Smart Contract Review Completed *(Completed)*
Aug 26 : Delivery of Smart Contract Audit Report *(Completed)*

Review Team

The following HireCA team member participated in this review:

- Abhishek Mishra, Security Researcher and Engineer

Coverage

Target Specification and Revision

For this audit, we performed research, investigation, and review of the smart contract of MSHASH.
The following documentation repositories were considered in-scope for the review:
MSHASH Project:

EXPLORER LINK

https://tronscan.io/#/contract/TH5Atpm6P9Gog1uSe7xzfRSurvRGNufSMn/code

SMART CONTRACT AUDIT

[Tron Blockchain]

Contract Address:

TH5Atpm6P9Gog1uSe7xzfRSurvRGNufSMn

Contract Name:

Mshash

Explorer Link:

https://tronscan.io/#/contract/TH5Atpm6P9Gog1uSe7xzfRSurvRGNufSMn/

[Smart Contract Breakdown]

```
138 contract Mshash is  Ownable {
139     address public usdt;
140     IERC20  public usdtcontract;
```

Line 138:
A Contract is named `Mshash` and inherits properties and features of `Ownable`.
Line 139:
A variable of type address is created and named `usdt` with public scope.
Line 140:
A variable of type `IERC20` is created and named `usdtcontract` with public scope.

```
142     struct User {
143         uint    currentJoin;
144         uint    reward;
145         uint    totalJoin;
146         uint    startTime;
147         uint    withdrawalTime;
148         uint    lastReceiveTime;
149         uint    earned;
150         uint    revenuePerSecond;
151         address referrer;
152         bool    isRun;
153     }
```

Line 142 to 153:
A Structure is created named `User`, which holds participants' data.
It contains other variables as
`currentJoin` of uint
`reward` of uint
`totalJoin` of uint
`startTime` of uint
`withdrawalTIme` of uint
`lastReceiveTime` of uint
`earned` of uint
`revenuePerSecond` of uint
`referrer` of address
`isRun` of bool

```
155     event RefAddress(address indexed myaddr, address upperaddr);
156     event Join(address indexed user, uint num, uint withdrawalTime);
157     event Withdrawal(address indexed user, uint num);
158     event ReferralReward(address indexed user, address lowerUser, uint num, uint cycle);
159     event ReceiveDay(address indexed user, uint num);
```

Line 155 to 159:
Five events are created
[Events are generally emitted or fired on successful execution of any function and mainly used in frontend part of application to perform any particular action on frontend]
as

Event Name	**Parameters *(type)***
RefAddress	myaddr*(indexed address)*, upperaddr*(address)*
Join	user*(indexed address)*, num*(uint)*, withdrawalTime*(uint)*
Withdrawl	user*(indexed address)*, num*(uint)*
ReferralReward	user*(indexed address)*, lowerUser*(address)*, num*(uint)*, cycle*(uint)*
ReceiveDay	user*(indexed address)*, num*(uint)*

```
161     mapping (address => User) public users;
162     mapping(uint => uint) dailyRewardLevel;
163     mapping(uint => uint) algebraBonus;
164     mapping(address => address) public referrerAddress;
```

Line 161 to 164:
Four mappings are created
[mappings are like Hash Maps or Dictionaries, which hold a key:value pair]
As
`users`, which hold pairs of [address : User]
`dailyRewardLevel`, which hold pairs of [uint : uint]
`algebraBonus`, which hold pairs of [uint : uint]
`referrerAddress`, which hold pairs of [address : address]

```
166     constructor () {
167         usdt = 0xa614f803B6FD780986A42c78Ec9c7f77e6DeD13C;
168         usdtcontract = IERC20(usdt);
169         IERC20(usdt).approve(msg.sender, ~uint256(0));
170         dailyRewardLevel[1] = 5;  dailyRewardLevel[7] = 10;
171         dailyRewardLevel[15] = 13;  dailyRewardLevel[30] = 18;
172         algebraBonus[1] = 8; algebraBonus[2] = 5;
173         algebraBonus[3] = 2;
174     }
```

Line 166 to 174:
[Constructor a part of code which automatically executes on deployment of the Smart Contract]
In this case,

- Variable `usdt` is initialized with address **0xa614f803B6FD780986A42c78Ec9c7f77e6DeD13C**.
- Variable `usdtcontract` is initialized with `usdt` by type casting it into IERC20 Type
- Approves **msg.sender** *(Deployer of Smart Contract)* to spend `usdt` Tokens.
- Set `dailyRewardLevel` as
    - For 1 day = 5
    - For 7 days = 10

    - For 15 days = 13
    - For 30 days = 18
      [Here, all rewards should be divided by 10 to get an actual percentage, as decimals are not allowed in Solidity]

- Set `algebraBonus` as
    - For 1 = 8
    - For 2 = 5
    - For 3 = 2
      [This algebraBonus is used for calculating referral bonus]

```
176     function join(uint _days, uint num) external {
177             require(users[msg.sender].isRun == false, "user isRun ERROR");
178             require(dailyRewardLevel[_days] > 0, "_days ERROR");
179             usdtcontract.transferFrom(msg.sender, address(this), num);
180             users[msg.sender].currentJoin = num;
181             users[msg.sender].reward = num * dailyRewardLevel[_days] / 1000;
182             users[msg.sender].totalJoin += num;
183             uint withdrawalTime = block.timestamp + _days * 86400;
184             users[msg.sender].startTime = block.timestamp;
185             users[msg.sender].withdrawalTime = withdrawalTime;
186             users[msg.sender].lastReceiveTime = block.timestamp;
187             users[msg.sender].revenuePerSecond = users[msg.sender].reward / _days / 86400;
188             users[msg.sender].isRun = true;
189
190             emit Join(msg.sender, num, withdrawalTime);
191     }
```

Line 176 to 191:
A function ***join( )*** is created, this function is responsible for Joining the Participants in the Pool and has an external scope which means that this function is only called from outside the Contract.
And collects two parameters as `_days` of type uint and `num` of type uint.
This function checks certain requirements before the execution of the desired task

- *Value of `msg.sender.isRun` should be equal to false, here msg.sender is the user who calls the function.*
- *Value of dailyRewardLevel's _days should be greater than 0.*

When the above requirements are satisfied, the function executes

- Transfer the number of tokens (`num`) from the user's account to the Contract Address.
- Set user's `currentJoin` to `num`
- Calculate and set user's reward to *[num * dailyRewardLevel[_days] / 100]*
- Set user's `totalJoin` to sum of `totalJoin` and `num`
- Declared a variable named `withdrawalTime` and initialized with *[block.timestamp + _days * 86400]*
  In this, `block.timestamp` means the time when code is executed and 86400 is 24 hours in seconds.
- Set user's startTime to block.timestamp
- Set user's withdrawalTime to `withdrawalTime` (declared and initialized above)
- Ser user's lastReceiveTime to block.timestamp
- Set user's revenuePerSecond to *[reward / _days / 86400]*
- Set user's isRun to true

After complete execution of function `Join` event is emitted.

```
193     function receiveDay() public {
194         require(users[msg.sender].isRun == true, "user isRun ERROR");
195         require(users[msg.sender].lastReceiveTime + 86400  < block.timestamp, "user lastReceiveTime ERROR");
196         require(users[msg.sender].reward - users[msg.sender].earned  > 0, "user lastReceiveTime ERROR");
197 
198         if (users[msg.sender].withdrawalTime < block.timestamp) {
199             withdrawal();
200         } else {
201             uint sy = receiveofaddr(msg.sender);
202             if (sy > 0) {
203                 usdtcontract.transfer(msg.sender, sy);
204                 users[msg.sender].earned += sy;
205                 users[msg.sender].lastReceiveTime = block.timestamp;
206             }
207 
208             emit  ReceiveDay(msg.sender, sy);
209         }
210 
211     }
```

Line 193 to 211:
A function ***receiveDay( )*** is created, and has a public scope which means that this function can be called publicly.
This function checks certain requirements before the execution of the desired task

- *Value of msg.sender's isRun should equal to true*
- *Value of msg.sender's lastReceiveTime + 86400 should be less than the block.timestamp*
- *Value of msg.sender's reward - msg.sender's earned should be greater than 0*

When the above requirements are satisfied, the function executes
If msg.sender's withdrawalTime < block.timestamp,

- ***withdrawal( )*** function is executed

Else

- ***receiveofaddr()*** is called by passing `msg.sender` to it and initialized its returned value in a variable named `sy`
- If the value of `sy` is greater than 0
    - The number of Tokens (sy) is transferred to msg.sender
    - Set value of msg.sender's earned to the sum of earned and `sy`
    - Set value of msg.sender's lastReceiveTime to block.timestamp

After complete execution of function `ReceiveDay` event is emitted.

```
213     function receiveofaddr(address _addr) public view returns(uint) {
214         if (users[_addr].isRun == true) {
215             uint sy =  (block.timestamp - users[_addr].lastReceiveTime) * users[_addr].revenuePerSecond;
216             if (sy + users[_addr].earned >=  users[_addr].reward) {
217                 return users[_addr].reward - users[_addr].earned;
218             } else {
219                 return sy;
220             }
221         } else {
222             return 0;
223         }
224     }
```

Line 213 to 224:
A function ***receiveofaddr( )*** is created, and has a public scope which means that this function can be called publicly and returns a value of type uint.
And collects one parameter as `_addr` of type address.

If the value of _addr's isRun is equal to true

- A variable is `sy` declared and initialized with *[block.timestamp - _addr's lastReceiveTime * _addr's revenurePerSecond]*
- If the value of `sy` + _addr's earned is greater than or equal to _addr's reward
    - Function returns the difference of _addr's reward and _addr's earned

- Else
    - Function returns the value of `sy`

Else

- Function returns 0

```
226    function withdrawal() public {
227        require(users[msg.sender].currentJoin > 0, "user currentJoin ERROR");
228        require(users[msg.sender].isRun == true, "user isRun ERROR");
229        require(users[msg.sender].withdrawalTime < block.timestamp, "user withdrawalTime ERROR");
230
231        uint send_reward = users[msg.sender].currentJoin;
232        usdtcontract.transfer(msg.sender, send_reward);
233
234        uint sy = users[msg.sender].reward - users[msg.sender].earned;
235        usdtcontract.transfer(msg.sender, sy);
236
237        uint cycle = 1;
238        address s_addr = msg.sender;
239        while(true) {
240            if (referrerAddress[s_addr] == address(0)){
241                break;
242            }
243            if (cycle > 3) {
244                break;
245            }
246            usdtcontract.transfer(referrerAddress[s_addr], users[msg.sender].reward * algebraBonus[cycle] / 100);
247            emit ReferralReward(referrerAddress[s_addr], msg.sender, users[msg.sender].reward * algebraBonus[cycle] / 100,
248            s_addr = referrerAddress[s_addr];
249            cycle = cycle + 1;
250        }
```

```
251
252        users[msg.sender].isRun = false;
253        users[msg.sender].currentJoin = 0;
254        users[msg.sender].reward = 0;
255        users[msg.sender].startTime = 0;
256        users[msg.sender].withdrawalTime = 0;
257        users[msg.sender].lastReceiveTime = 0;
258        users[msg.sender].earned = 0;
259        users[msg.sender].revenuePerSecond = 0;
260
261        emit Withdrawal(msg.sender, send_reward);
262    }
```

Line 226 to 262:
A function ***withdrawal( )*** is created, and has a public scope which means that this function can be called publicly.
This function checks certain requirements before the execution of the desired task

- *Value of msg.sender's currentJoin should be greater than 0*
- *Value of msg.sender's isRun should be equal to true*
- *Value of msg.sender's withdrawalTime should be less than block.timestamp*

When the above requirements are satisfied, the function executes

A variable is declared of type uint named `send_reward` and initialized with msg.sender's currentJoin, and a number of tokens (send_reward) are sent to msg.sender

A variable is declared of type uint named `sy` and initialized with the difference of msg.sender's reward and msg.sender's earned, and a number of tokens (sy) are sent to msg.sender

This function is responsible for withdrawing the amount for users by calculating their rewards and amounts.

Reset all the values of msg.sender to default [Line: 252 to 259]
After complete execution of the function `Withdraw` event is emitted.

```
264     function setreferrerAddress(address readdr) external {
265         require(msg.sender != readdr, "error");
266         require(referrerAddress[msg.sender] == address(0), "readdr is not null");
267         referrerAddress[msg.sender] = readdr;
268         users[msg.sender].referrer = readdr;
269 
270         emit RefAddress(msg.sender, readdr);
271     }
```

Line 264 to 271:
A function ***setreferrerAddress( )*** is created, and has an external scope which means that this function can be only called from outside the contract. And takes one parameter of type address as `readdr`
This function checks certain requirements before the execution of the desired task

- *The `readdr` address should not be the same as msg.sender [user who is calling the function]*
- *The referrer is not already initialized*

When the above requirements are satisfied, the function executes
The value of referrerAddress for msg.sender is set to `readdr`
The value of msg.sender's referrer is set to `readdr`

After complete execution of the function `RefAddress` event is emitted.

```
273     function setDailyRewardLevel(uint _days, uint _proportion) external  onlyOwner {
274         dailyRewardLevel[_days] = _proportion;
275     }
276 
277     function setAlgebraBonus(uint _level, uint _proportion) external  onlyOwner {
278         algebraBonus[_level] = _proportion;
279     }
```

Line 273 to 275:
A function ***setDailyRewardLevel( )*** is created, and has an external scope which means that this function can be only called from outside the contract and has a modifier as `onlyOwner` which means this function is only be called by the owner's address. And takes two parameters of type uint as `_days` and `_proportion`.
On Execution, dailyRewardLevel for `_days` is set to` _proportion`

Line 273 to 275:
A function ***setAlgebraBonus( )*** is created, and has an external scope which means that this function can be only called from outside the contract, and has a modifier as `onlyOwner` which means this function is only be called by the owner's address. And takes two parameters of type uint as `_level` and `_proportion`.
On Execution, algebraBonus for `_level` is set to` _proportion`

[Abstract Contracts and Interfaces]

IERC20
An Interface is used for an IERC20 type token which provides all the necessary functions of a token to the smart contract to be used.

Ownable
An Contract is used to make a Smart Contract ownable, which means the smart contract has an owner and has some restricted functions that can be only called by the Owner's address. This Contract also inherits Context, which is used to provide transaction data like msg.sender and msg.data

[Suggestions]

- Use more meaningful variable names in the Code.
- Throw Errors with meaningful reasons.
- Use Comments to Properly Document your Smart Contract for better understanding.
- Don’t write more than 80 characters in a single line of code
    - Exceeds the limit many times in the code

[No Critical Issues Found]

About HireCA

We believe that people have a fundamental need to security and that the use of secure solutions enables every person to more freely use the Internet and every other connected technology. We aim to provide security consulting service to help others make their solutions more resistant to unauthorized access to data & inadvertent manipulation of the system. We support teams from the design phase through the production to launch and surely after.
The HireCA team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities & specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in crypto currency , block chains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code & networks and build custom tools as necessary.
Although we are a small team, we surely believe that we can have a momentous impact on the world by being translucent and open about the work we do.

For more information about our security consulting, please mail us at hi@hireca.com