DeHacker

Code Security Assessment

Project Chosen

June 17th, 2022

Contents

Summary

DeHacker's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow/underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service/logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Issue Categories

Every issue in this report was assigned a severity level from the following:

Critical severity issues

A vulnerability that can disrupt the contract functioning in a number of scenarios or creates a risk that the contract may be broken.

Major severity issues

A vulnerability that affects the desired outcome when using a contract or provides the opportunity to use a contract in an unintended way.

Medium severity issues

A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.

Minor severity issues

A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.

Informational

A vulnerability that has informational character but is not affecting any of the code.

Overview

Project Summary

Project Name	Project Chosen
Platform	BSC
website	https://www.chosen.zone/
Type	Defi
Language	Solidity

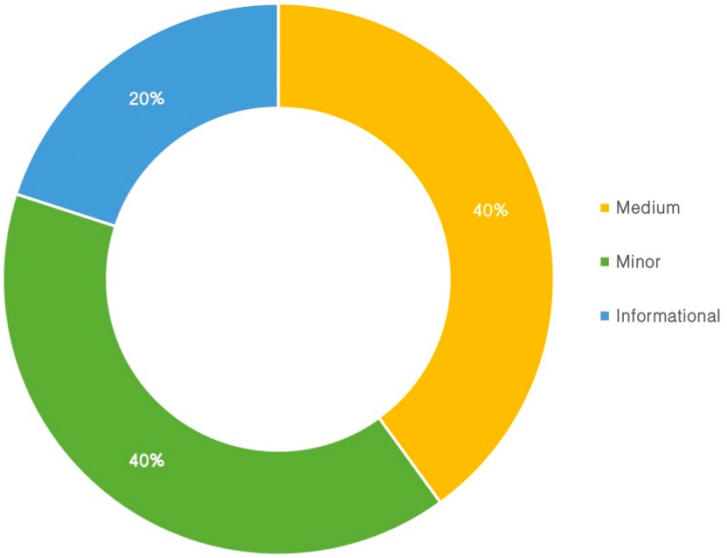
Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
Critical	0	0	0	0	0	0
Major	0	0	0	0	0	0
Medium	2	0	0	0	0	2
Minor	2	0	0	0	0	2
Informational	1	0	0	0	0	1
Discussion	0	0	0	0	0	0

Audit scope

ID	File	SHA256 Checksum
IOP	IgoOrePool.sol	0b61214aba29bbdea54ef12b679d9788f15429e8ec17ca37c2ce440104030ebe
VCN	VoucherNft.sol	53c6a4c9865815b3e3daaa54951b0432b1bae8f4a72c00b3ee384446952ca8d4
TKP	TokenTransferProxy.sol	44c4a8acbb13381721f5cdcaa363986efae3fc4a3d78256ce61d14e9505efddf

Findings



ID	Title	Category	Severity	Status
CSI-01	Improved Logic in VoucherNft: queryUserAllNft()	Business Logic	Minor	Resolved
CSI-02	Revisited Logic in IgoOrePool: adminSafeTransfer()	Business Logic	Medium	Resolved
CSI-03	Potential Lock of User Stakes	Numeric Errors	Medium	Resolved
CSI-04	Simplified Logic in receiveRewards()	Business Logic	Informational	Resolved
CSI-05	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Minor	Resolved

Medium

CSI-01 | Revisited Logic in IgoOrePool: adminSafeTransfer()

Category	Severity	Location	Status
Business Logic	Medium	IgoOrePool.sol:50-77	Resolved

Description

The Project Chosen protocol has a built-in incentive mechanism that encourages participating users to stake the configured _mortgageLp for rewards. The incentive mechanism is mainly implemented in the IgoOrePool contract, and our analysis shows the contract also contains a privileged function that needs to be revisited.

To elaborate, we show below this privileged function adminSafeTransfer(). As the name indicates, this function facilitates the administrative transfer of the funds in the current incentive pool. However, it allows the current operator to withdraw the staked funds from protocol users. This design needs to be revisited so that the staked funds will not be jeopardized. In other words, there is a need to ensure the given _token cannot be the staked token _mortgageLp with the following requirement: require(_token != _mortgageLp).

```
50      function adminSafeTransfer(
51          address _token,
52          address _to,
53          uint256 _amount
54      ) public onlyOperator {
55          _upgradeSafeTransfer(_token, _to, _amount);
56      }

58      function _upgradeSafeTransfer(
59          address _token,
60          address _to,
61          uint256 _amount
62      ) internal {
63          require(_token != address(0), "Token can not be 0x0!");
64          require(_amount > 0, "Transfer limit cannot be 0!");

66          uint256 balance = IERC20(_token).balanceOf(address(this));

68          if (_amount > balance) {
69              require(
70                  _amount.sub(balance) < uint256(1).mul(1e18),
71                  "Insufficient contract balance!"
72              );
73              IERC20(_token).safeTransfer(_to, balance);
74          } else {
75              IERC20(_token).safeTransfer(_to, _amount);
76          }
77      }
```

Recommendation

Revise the above adminSafeTransfer() function so that the user stakes cannot be administratively transferred without their permission.

Alleviation

The Project Chosen team has properly fixed the issue.

CSI-03 | Potential Lock of User Stakes

Category	Severity	Location	Status
Numeric Errors	Medium	IgoOrePool:104-120; 292-304	Resolved

Description

As mentioned earlier, the Project Chosen protocol shares an incentivizer mechanism that is inspired from Synthetix. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token. And it is part of the `updateReward()` modifier that would be invoked up-front for almost every public function in `IgoOrePool` to update and use the latest reward rate.

The reason is due to the known potential underflow pitfall when the `endTime` parameter is inappropriately configured. In particular, as the `rewardPerToken()` routine involves the multiplication of three uint256 integers and the first integer depends on the `lastTimeRewardApplicable()`. `sub(lastUpdateTime)` (lines 114 – 115). While the `endTime` parameter is configured to be smaller than `lastUpdateTime`, it may result in an undesirable underflow, which effectively reverts the `rewardPerToken()` execution!

```
104     function lastTimeRewardApplicable() public view returns (uint256) {
105         return Math.min(block.timestamp, endTime);
106     }
107
108     function rewardPerToken() public view returns (uint256) {
109         if (totalPower == 0) {
110             return intervalReward;
111         }
112         return
113             intervalReward.add(
114                 lastTimeRewardApplicable()
115                     .sub(lastUpdateTime)
116                     .mul(miningOutput)
117                     .mul(1e18)
118                     .div(totalPower)
119             );
120     }
```

```
292     function updateStartTime(uint256 _time) public onlyOperator {
293         startTime = _time;
294     }
295
296     function updateEndTime(uint256 _time) public onlyOperator {
297         endTime = _time;
298     }
299
300     function updateMiningOutput(uint256 _yield) public onlyOperator {
301         intervalReward = rewardPerToken();
302         lastUpdateTime = lastTimeRewardApplicable();
303         miningOutput = _yield;
304     }
```

The underflow may in essence lock all deposited funds! Note that an authentication check on the caller of `onlyOperator()` greatly alleviates such concern. Currently, only the operator address is able to call `updateEndTime()` and this address can be set when the contract is deployed. Apparently, if the operator is a normal address; it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the `endTime` parameter will not be configured to underflow and lock users' funds.

Recommendation

Mitigate the potential underflow risk in the `IgoOrePool` pool.

Alleviation

The Project Chosen team has properly fixed the issue.

Minor

CSI-01 | Improved Logic in VoucherNft: queryUserAllNft()

Category	Severity	Location	Status
Business Logic	Minor	VoucherNft.sol:89-98	Resolved

Description

The Project Chosen protocol will generate a winner medal NFT to the user who successfully predicts the token price movement after the token's TGE. While examining this medal NFT logic, we notice the current implementation can be improved.

To elaborate, we show below the related queryUserAllNft() function. It is a getter routine that is designed to return the information about the given list of NFTs. However, the current implementation does not properly initialize the return array and fails to return the queried information.

```
89      function queryUserAllNft(uint256[] memory _ids)
90          public
91          view
92          returns (NftInfo[] memory nftArray)
93      {
94          for (uint256 i = 0; i < _ids.length; i++) {
95              nftArray[nftArray.length] = nftInfo[_ids[i]];
96          }
97          return nftArray;
98      }
```

Recommendation

Improve the above queryUserAllNft() routine to properly return the queried NFT information.

Alleviation

This issue has been resolved as the team clarifiies that nftArray does not need to be specially initialized: If `nftArray` is not fifilled with matching data, it will just return empty

CSI-05 | Accommodation of Non-ERC20-Compliant Tokens

Category	Severity	Location	Status
Business Logic	Minor	TokenTransferProxy.sol:126-139; IgoOrePool:179-217	Resolved

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

We use the popular stable coin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the IERC20 interface has defined the `transfer()` interface with a bool return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of USDT's `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()`/`transferFrom()` as well, i.e., `safeApprove()`/`safeTransferFrom()`. In current implementation, if we examine the `IgoOrePool::withdraw()` routine that is designed to transfer the funds back to the staking user. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (lines 210 and 213).

Recommendation

Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom().

Alleviation

The Project Chosen team has properly fixed the issue.

Informational

CSI-04 | Simplified Logic in receiveRewards()

Category	Severity	Location	Status
Business Logic	Informational	IgoOrePool:88-96,218-227	Resolved

Description

As mentioned earlier, the Project Chosen protocol shares an incentivizer mechanism inspired from Synthetix, which has the receiveRewards() routine to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the receiveRewards() routine has a modifier, i.e., updateReward(msg.sender)), which timely updates the given user's (earned) rewards in rewards[_account] (line 219).

```
88    modifier updateReward(address _account) {
89        intervalReward = rewardPerToken();
90        lastUpdateTime = lastTimeRewardApplicable();
91        if (_account != address(0)) {
92            rewards[_account] = earned(_account);
93            userLastIntervalReward[_account] = intervalReward;
94        }
95        _;
96    }
```

```
218    function receiveRewards() public updateReward(msg.sender) {
219        uint256 reward = earned(msg.sender);
220        if (reward > 0) {
221            rewards[msg.sender] = 0;
222            IERC20(_profitLp).transfer(msg.sender, reward);
223        } else {
224            reward = 0;
225        }
226        emit ReceiveRewards(msg.sender, reward);
227    }
```

Having the modifiifier updateReward(), there is no need to re-calculate the earned reward for the given user. In other words, we can simply re-use the calculated rewards[msg.sender] and assign it to the reward variable (line 219).

Recommendation

Avoid the duplicated calculation of the caller's reward in receiveRewards(), which also leads to (small) benefificial reduction of associated gas cost.

Alleviation

The Project Chosen team has properly fixed the issue.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Coding Style

Coding Style findings usually do not affect the generated bytecode but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block. timestamp works.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux “sha256sum" command against the target file.

About

DeHacker is a team of auditors and white hat hackers who perform security audits and assessments. With decades of experience in security and distributed systems, our experts focus on the ins and outs of system security. Our services follow clear and prudent industry standards. Whether it's reviewing the smallest modifications or a new platform, we'll provide an in-depth security survey at every stage of your company's project. We provide comprehensive vulnerability reports and identify structural inefficiencies in smart contract code, combining high-end security research with a real-world attacker mindset to reduce risk and harden code.

BLOCKCHAIINS

Ethereum
Cosmos
Eos
Substrate

TECH STACK

Python
Solidity
Rust
c++

CONTACTS

https://dehacker.io
https://twitter.com/dehackerio
https://github.com/dehacker/audits_public
https://t.me/dehackerio
https://blog.dehacker.io/

June 2022