SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: PrestigeClub
Prepared on: 15/11/2020
Revised on: 19/11/2020
Platform: Ethereum
Language: Solidity

Table of contents

Document

Name	Smart Contract Code Review and Security Analysis Report for Prestige Club
Platform	Ethereum / Solidity
MD5 hash	BFADA8FC61A2F695CFB07AFCFE558B4D
File name	PrestigeClub.sol
SHA256 hash	5BDD56C2DC0B6F226914A2FFBE91CC43C384B7AE02C564BCB0A153B79E6D776A
Date	15/11/2020

Project Scope

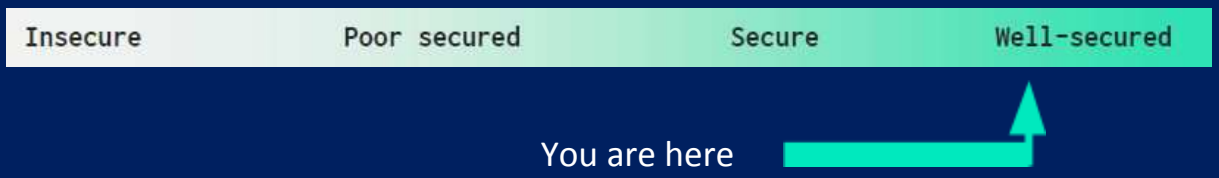
The scope of the project is Prestige Club smart contract.

We have scanned this smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that are considered (the full list includes them but is not limited to them):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment and after audit revisions, Customer`s solidity smart contract is: **well secured.**



Our team performed analysis of code functionality, manual audit and automated checks with smartDec, Mythril, Slither and remix IDE. All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in the AS-IS section and all found issues can be found in the Audit overview section.

We found 1 high, 2 medium and 0 low and some very low level issues.
We revised them again and they are resolved in the latest version.

Code Quality

Prestige Club protocol consists of modular smart contracts, which interact with each other . This modularization increases the efficiency of the code execution flow. This type of structure of smart contracts increases ease of code writing in the development process.

The Prestige Club team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way. Overall, the code is commented on necessary places to improve readability. Commenting can provide rich documentation for functions, return variables and more. Use of Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

As mentioned above, Commenting was done at necessary places. It's recommended to write comments in NatSpec format.

We were given a Prestige Club defi document, which is very helpful in understanding the overall architecture of the protocol. It also provided a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, library-like contracts are used in this smart contract infrastructure. Those were based on well known industry standard open source projects. We found no serious issues in that.

Smart contract has ownerOnly functions, which owner can execute to carry out admin level functionalities.

AS-IS overview

Prestige Club **contract overview**

Prestige Club is a smart contract that provides decentralized financial activities to the users. Its file format is described below:

Contract: Context
Inherit: null
About: This contract provides context in the form of msg.sender and msg.data.
Observation: Since this smart contract is not being used in GSN, or any externally called contract. so, this contract can be safely removed and can be used msg.serder and msg.data directly. This will save some gas. On another hand, the presence of this contract does not create any major issue.
Test Report: All passed including security check.
Score: **Passed**
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_msgSender	read	Passed	All Passed	No Issue	Passed
2	_msgData	read	Passed	All Passed	No Issue	Passed

Contract: Ownable
Inherit: context
About: This contract allows the owner to do admin based financial activities.
Observation: Transfer ownership is ACTIVE. If the owner sent ownership to invalid address by mistake (we have seen such scenarios in which the owner does this by mistake in hurry), then it will render the smart contract ownerless. It is recommended to implement acceptOwnership functions (described in Audit Finding section)
Test Report: All passed including security check.
Score: **Passed**
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	owner	read	Passed	All Passed	No Issue	Passed
2	Transferownership	write	Passed	All Passed	No Issue	Passed

Contract: Pausable

Inherit: Context

About: This contract allows the owner to pause/unpause some activities.

Observation: Passed

Test Report: All passed including security check.

Score: **Passed**

Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	Paused	read	Passed	All Passed	No Issue	Passed
2	_Pause	write	Passed	All Passed	No Issue	Passed
3	_unpause	write	passed	All Passed	No Issue	Passed

Contract: Prestigeclub

Inherit: ownable, Pausable

About: This contract provides users to do team based financial activities.

Observation: Use of SafeMath recommended.

Test Report: We found the possibility of re-entrency in the withdraw function (more details in the audit finding section below). Also, safemath should be implemented to prevent overflow/underflow possibility. So, this must be resolved before going to production.

Score: **Passed after revision**

Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	receive	write	Passed	All Passed	No Issue	Passed
3	UpdateUpline	read	Passed	All Passed	No Issue	Passed
4	UpdatePayout	write	Passed	All Passed	No Issue	Passed
5	getInterestPayout	read	Passed	All Passed	No Issue	Passed
6	getPoolPayout	read	Passed	All Passed	No Issue	Passed
7	getDownlinePayout	read	Passed	All Passed	No Issue	Passed
8	getDirectsPayout	read	Passed	All Passed	No Issue	Passed
9	PushPoolState	write	Passed	All Passed	No Issue	Passed
10	UpdateUserPool	write	Passed	All Passed	No Issue	Passed
11	UpdateDownlineBonusStage	write	Passed	All Passed	No Issue	Passed
12	CalculateDirects	read	Passed	All Passed	No Issue	Passed
13	CalculateDirects	read	Passed	All Passed	No Issue	Passed
14	Withdraw	write	Passed	All Passed	No Issue	Passed

15	_SetReferral	write	Passed	All Passed	No Issue	Passed
16	totalDeposits	read	Passed	All Passed	No Issue	Passed
17	invest	write	Passed	All Passed	No Issue	Passed
18	reinvest	write	Passed	All Passed	No Issue	Passed
19	SetMinDeposit	write	Passed	All Passed	No Issue	Passed
20	SetMinWithdraw	write	Passed	All Passed	No Issue	Passed
21	Pause	write	Passed	All Passed	No Issue	Passed
22	Unpause	write	Passed	All Passed	No Issue	Passed
23	getUserData	read	Passed	All Passed	No Issue	Passed
24	getUserList	write	Passed	All Passed	No Issue	Passed
25	getusers	write	Passed	All Passed	No Issue	Passed
26	triggerCalculation	write	Passed	All Passed	No Issue	Passed

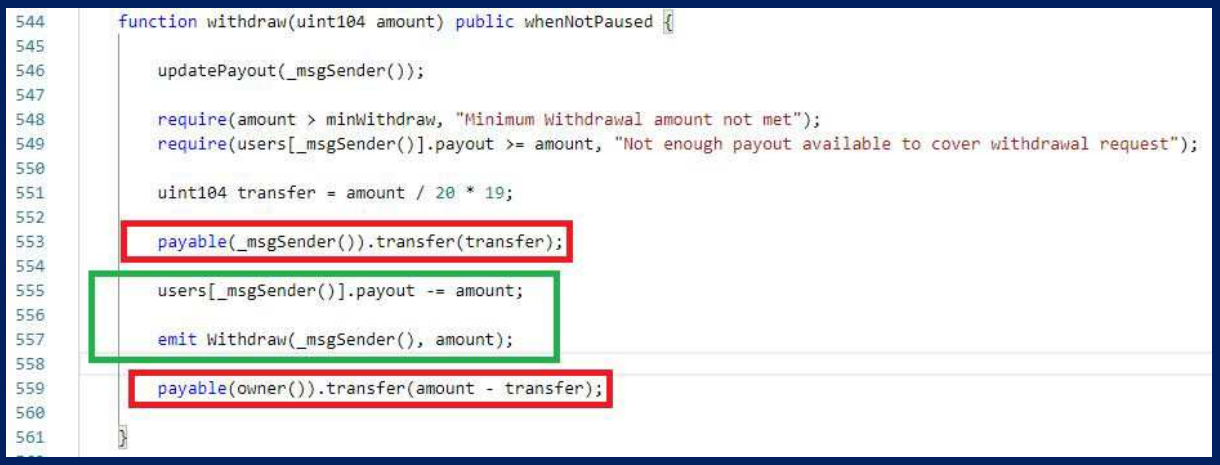
Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Vulnerabilities

(1) Double Spend (reentrancy) might be attacked. In withdraw function, variable update should be before transfer initiation. (as in the image below, the green box should be above the red box)



```
544    function withdraw(uint104 amount) public whenNotPaused {
545
546        updatePayout(_msgSender());
547
548        require(amount > minWithdraw, "Minimum Withdrawal amount not met");
549        require(users[_msgSender()].payout >= amount, "Not enough payout available to cover withdrawal request");
550
551        uint104 transfer = amount / 20 * 19;
552
553        payable(_msgSender()).transfer(transfer);
554
555        users[_msgSender()].payout -= amount;
556
557        emit Withdraw(_msgSender(), amount);
558
559        payable(owner()).transfer(amount - transfer);
560
561    }
```

Revision: This issue was fixed by Prestige club team

High Severity Vulnerabilities

No high severity vulnerabilities were found.

Medium Severity Vulnerabilities

(1) SafeMath is not used. Some test cases may trigger overflow / underflow. Although it costs very little extra gas, it gives protection against such attacks. There are unlimited potential use case scenarios where it might go wrong. So, it's better to eliminate that entirely using SafeMath. Please implement it from open zepelin: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

Revision: This issue was fixed by Prestige club team

(2) Multiplication after division, this may lead to 0 return values in some lower input values. It is a good practice to do all multiplications first and then divide it last. So, the equation would be:

deposit * quote / 10000

```
355         }else{
356             quote = 10;
357         }
358
359         return deposit / 10000 * quote;
360     }
361
362     //Pool Payout does not get calculated per day but fo
363     function getPoolPayout(address adr, uint40 dayz) pub
364
```

Revision: This issue was fixed by Prestige club team

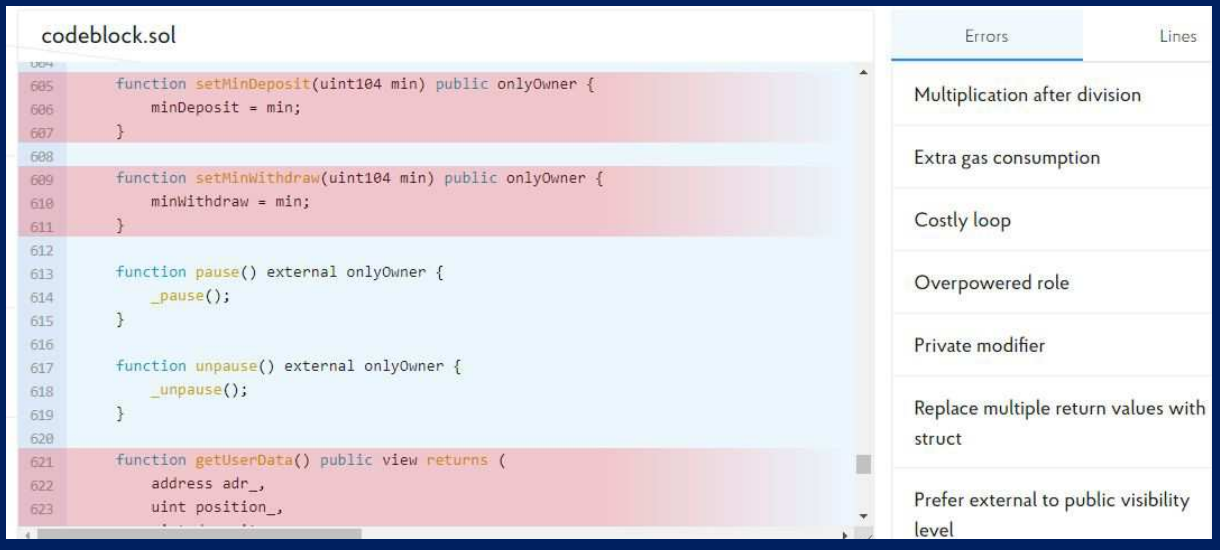
Low Severity Vulnerabilities

No low severity vulnerabilities were found.

Very Low Severity Vulnerabilities (Resolved and acknowledged)

(1) Expansive loop, loop is limited by array length in line 379 and 465, If it is limited by plan then OK else needs to change to avoid function call fail or High Gas consumption

(2) Other errors indicated by static tool analysis can be ignored for production.



(3) Please make the ownership transfer function PASSIVE. In other words, sending ownership to any wallet directly to invalid wallet by mistake can create problems (we have seen such scenarios happen to contract owners). In passive transfer, the new owner must accept ownership for the actual ownership to be transferred. Please use following structure:

```
function transferOwnership(address _newOwner) public onlyOwner {
    newOwner = _newOwner;
}

//this flow is to prevent transferring ownership to wrong wallet by mistake
function acceptOwnership() public {
    require(msg.sender == newOwner);
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
    newOwner = address(0);
}
```

Conclusion

We were given a contract file. And we have used all possible tests based on given objects as files. The contract is mostly commented. We recommend using the NatScap standard.

Since possible test cases can be unlimited for such extensive smart contract protocol, so we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

Smart contracts within the scope, were manually reviewed and analyzed with static analysis tools. Smart Contract's high level description of functionality was presented in As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of contracts after audit revision is, **Well Secured**. This contract is good to go for the production.

Disclaimers

The smart contracts given for audit have been analyzed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases are unlimited, so the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee explicit security of the audited smart contracts.