



SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

For

**Match & StakingPortal
(Order #14092020)**

Prepared By: Chandan Kumar
Prepared For: Staking Portal
Prepared on: 12/09/2020
audit@etherauthority.io

Table of Content

1. Disclaimer
2. Overview of the audit
3. Attacks made to the contract
4. Good things in smart contract
5. Critical vulnerabilities found in the contract
6. Medium vulnerabilities found in the contract
7. Low severity vulnerabilities found in the contract
8. Very low severity vulnerabilities found in the contract
9. Gas Optimization Discussion
10. Summary of the audit

THIS DOCUMENT MAY CONTAIN CONFIDENTIAL INFORMATION ABOUT IT SYSTEMS AND INTELLECTUAL PROPERTY OF THE CUSTOMER AS WELL AS INFORMATION ABOUT POTENTIAL VULNERABILITIES AND METHODS OF THEIR EXPLOITATION. THE REPORT CONTAINING CONFIDENTIAL INFORMATION CAN BE USED INTERNALLY BY THE CUSTOMER OR IT CAN BE DISCLOSED PUBLICLY AFTER ALL VULNERABILITIES ARE FIXED - UPON DECISION OF CUSTOMER.

2. Overview of the audit

The project has one smart contract file and two other dependent contracts which are already in production:

`StakePortalV3.sol`

533 lines

It contains approx 533 lines of Solidity code. All functions and state variables are properly commented, the latest stable version (0.5.0) with perfectly implemented code blocks, with properly assigned visibility of functions and consistent proper variable tracking flow makes this contract absolutely perfect. But apart from this, **there is some logical syntax error and potential damage which makes this contract not to fit for production, but those can be altered/changes/corrected easily to make it fit for production.** Else code is also robust and fully protected towards many popular attack possibilities which is a good sign of code approach.

Apart from errors (must solve before deploy), and warnings these contract are

- Contract compiled successfully up to version 0.7.0
- Perfect administrative control
- Optimized for GAS.
- Good use of safe math.
- Staking triggered by an external contract well written.
- Unstaking and release with lock period defined well
- Average reward method used , good for one call withdrawal but little deviations may be felt over time, but used by many in production.

The audit was performed by two senior solidity auditors at EtherAuthority. The team has extensive work experience in developing and auditing the smart contracts.

This audit procedure also included the use of automated software to further scan of the code to identify potential issues:

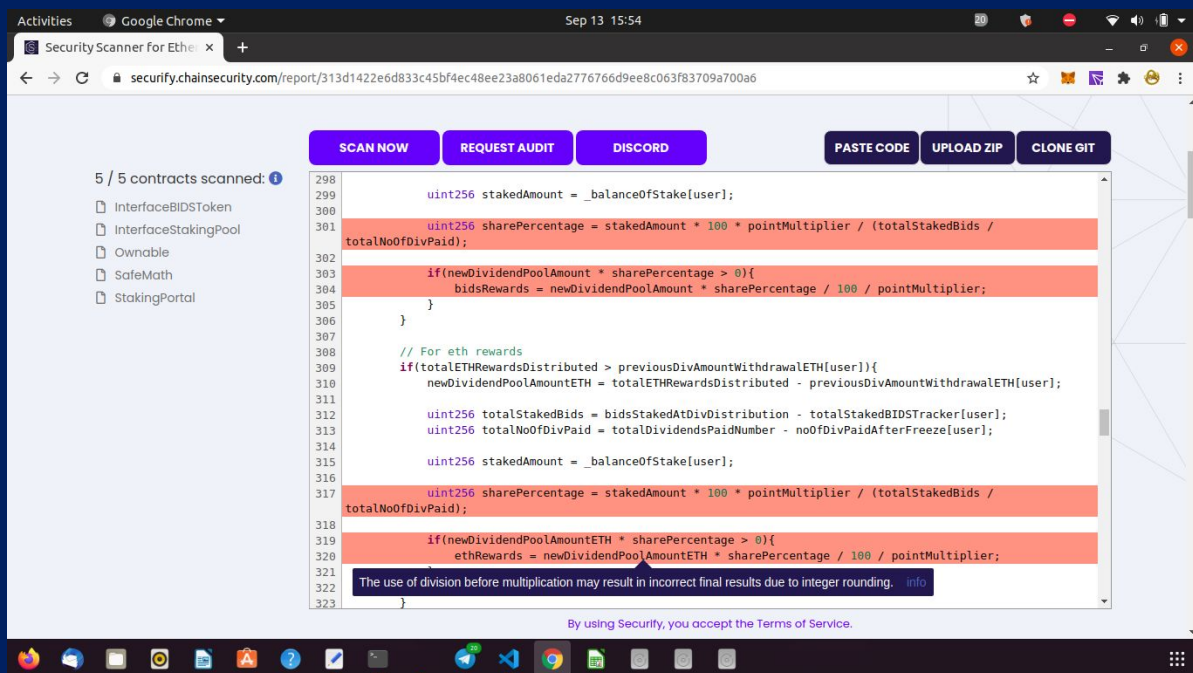
To Do With :

- Error:** Must be corrected before deploy
- Warning:** Should be checked with surrounding logics, if OK then good to go
- Ignore:** No need to pay attention

For example:

<https://securify.chainsecurity.com/report/313d1422e6d833c45bf4ec48ee23a8061eda2776766d9ee8c063f83709a700a6>

Here all mentioned reports of tools are either mentioned already or not such serious to pay attention to.



And on <https://tool.smartdec.net/scan/b7bcc64364ef4f02b384a9fe3c7dc843> also no such serious error found in test

Implicit visibility level <https://mythx.io> tool provided as remix.ethereum.org plugin

Above are the only few points raised by the automated tools and taken into consideration, and these are not such a problem actually for ex. Loops are limited by iteration, safe math protects some attacks, and address zero is checked to move into the processing part of the function so All are OK.

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Moderated
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Moderated
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
Burn	Lower limit for Burn	N/A

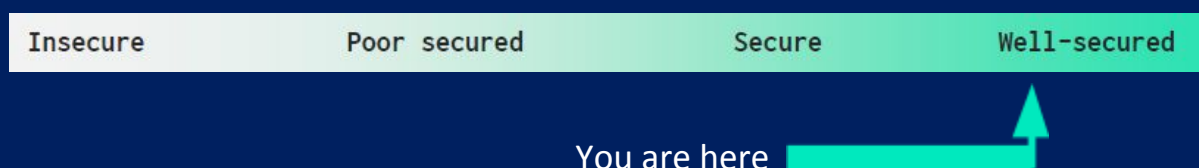
EtherAuthority Limited (www.EtherAuthority.io)

	High consumption 'for/while' loop	Passed
	High consumption 'storage' storage	Passed
	"Out of Gas" Attack	Moderated
Business Risk	The maximum limit for mintage not set	N/A
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: **Passed** (after rectifying observed issues)

Point of Marks:

According to the initial and revised assessment, Customer`s smart contract is **Well-secured**.



Main Details of findings are following:

Modifier "onlyTokenContract" defined, but no where used, instead local require defined. Either this modifier should be removed or local required related to this should be replaced by this modifier.

Address(0) check needed in "tokenCallback" function. To avoid mis-lead of dependent contracts (there also address(0) not checked) which is in production.

“_unstakeAmount” parameter should be checked for “0” in “Unstake” function.

While relocating funds (where transfer is used) variable reset should be before transfer to avoid any re-entrancy attack.

For ex “_investorPendingUnstake[msg.sender] = 0;” should be placed before “InterfaceBIDSToken(bidsTokenContractAddress).transfer(msg.sender, withdrawableAmount);” for security reason not to hack any token or double spend.

“require(user!=address(0))” in “availableRewards” function missing.

And also the same address(0) check needed as:

require(_investor!=address(0),"Invalid address"); in “updateTracker” function.

The ChangeSigner function updates new users to “true” but not disabling the old one to “false” which may cause compromised security to some extent.

Fallback function can accept payment but there is no way to withdraw any unclaimed amount , those currencies may be locked in the system, and because the averaging system is being used as dividend distribution so for this point of view also there should be admin withdrawal for unclaimed amounts.

Changing the globalHalt return value is always false.

Special requested calculation check for dividend distribution : As per request, we checked the dividend distribution methodology for potential value difference in outcomes. As per our observation, the distribution logic is based on “averaging” methods to track payout and pending payout amount across all the users. As averaging itself by nature, will never be accurate. So little deviation is naturally with this approach as many other houses are using the same method of distribution. So over large scale the -ve deviation will also be averaged by +ve deviation so discrepancy will less over time, and hence this method can be used for dividend distribution.

Note : If accurate distribution is needed , then we recommend to use separated distribution records for each distribution with respect to each user’s deposit on that instance of time. This method has only drawbacks if an user does not withdraw for long, his loop of calculation will be too deep. Hence there will be difficulty in withdrawing in one call. But this can be trically handled in programming with limited loop iteration.

3. Attacks tested on the contract

In order to check for the security of the contract, we tested several attacks on the code. Some of those are as below:

3.1: Over and under flows

SafeMath library is used in the contract, which prevents the possibility of overflow and underflow attacks. Most contract parts worked well.

3.2: Short address attack

Although this contract is **not vulnerable** to this attack because it is good that functions are called after checking the validity of the address from the outside client.

3.3: Visibility & Delegate call

Delegate call is not used in the contract thus it does not have this vulnerability.

3.4: Reentrancy / The DAO /hack or double spend

Use of “require” function used which is good and Checks-Effects-Interactions pattern in this smart contract mitigated this vulnerability, and also some calls rooted internally is good and safe.

But the dividend distribution part in contract needs to shift value update before transfer, because it is a public access type function so double-spend/reentrancy is possible.

3.5: Forcing ether to a contract

Here, the Smart Contract’s balance has never been used as guard, which mitigated this vulnerability

3.6: Denial Of Service (DoS)

There is **no** any process consuming loops (if loops then limited) in the contracts which could be used for DoS attacks. and thus this contract is safe to DoS.

4. Good things in the smart contract

4.1 Checks-Effects-Interactions pattern

While transferring tokens, this contract does all the process first and then transfers them. The same while doing other processes too. This is very good practice which prevents malicious possibility.

4.2 Functions input parameters passed

The functions in this contract verify the validity of the input parameters, and these validations cannot be by-passed in anyway.

4.3 Conditions validations

The validation of input parameters is done to prevent overflow and underflow of integers. Although the SafeMath library used is also a good programming flow.
<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol>

5. Critical vulnerabilities found in the contract

Critical issues that could damage heavily the integrity of the contract. Some bug that would allow attackers to steal ether is a critical issue.

Multiple signer possibility : while changing signer pls disable the old signer.

Transfer should be initiated after resetting the variable to avoid any re-entrancy attack.

Above issue was rectified

6. Medium vulnerabilities found in the contract

Those vulnerabilities that could damage the contract but with some kind of limitations. Like a bug allowing people to modify a random variable. These are discussed above apart from that

**** No such medium vulnerabilities found in contract.**

7. Low severity vulnerabilities found

Those do not damage the contract, but better to resolve and make code clean.

7.1: Compiler version can be fixed for higher one.

**** No other low severity vulnerabilities found in contract.**

8. Very low severity vulnerabilities found

The presence of these things does not make any negative effect. But just to clean up the code.

**** No such vulnerabilities found in contract.**

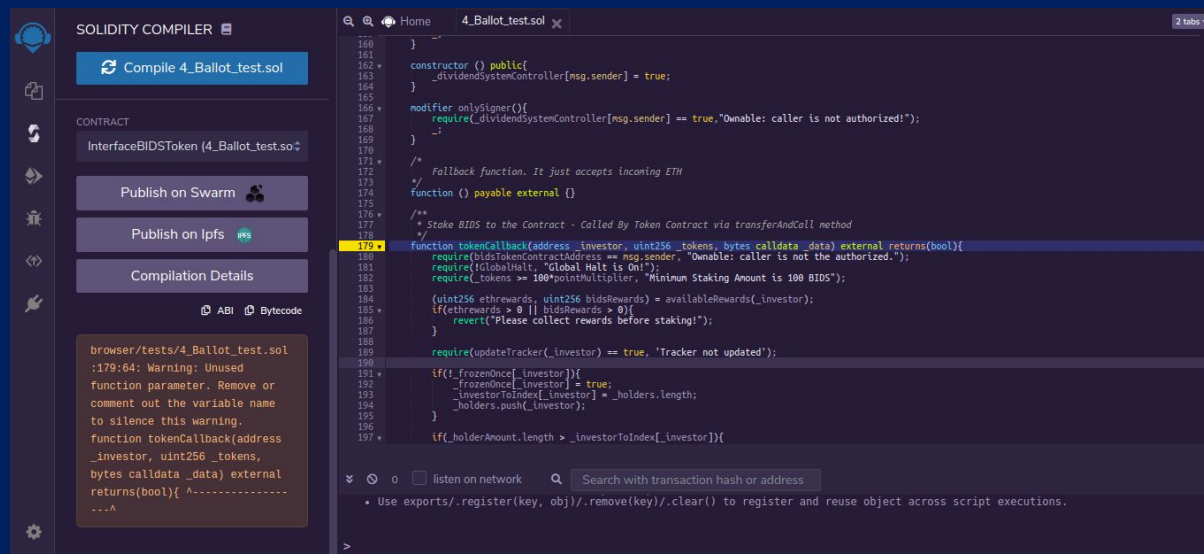
9. Gas Optimization Discussion

=> The Contract quite good to gas (has low for the gas cost). Little more can be improved by using more optimized storage by packing multiple variables under uint256 size limit.

10. Summary of the Audit

Overall, after modification of all which are discussed above the compiled output of code for token/dividend implementation as below.

Compiler showed couple of warnings, as below: this warning is for un-used variable that can be removed to make code clean



The screenshot shows the Solidity Compiler interface with the following components:

- SOLIDITY COMPILER** header with a "Compile 4_Ballot_test.sol" button.
- CONTRACT** section showing "InterfaceBIDSToken (4_Ballot_test.sol)".
- Buttons for "Publish on Swarm", "Publish on Ipfs", and "Compilation Details".
- ABI** and **Bytecode** tabs.
- Warning Panel** (bottom left):

```
browser/tests/4_Ballot_test.sol
:179:64: Warning: Unused
function parameter. Remove or
comment out the variable name
to silence this warning.
function tokenCallback(address
_investor, uint256 _tokens,
bytes calldata _data) external
returns(bool){ ^-----
---^
```
- Code Editor** (right):

```
160 }
161
162 constructor () public{
163     _dividendSystemController[msg.sender] = true;
164 }
165
166 modifier onlySigner(){
167     require(_dividendSystemController[msg.sender] == true, "Ownable: caller is not authorized!");
168     _;
169 }
170
171 /*
172  * Fallback function. It just accepts incoming ETH
173  */
174 function () payable external {}
175
176 /**
177  * Stake BIDS to the Contract - Called By Token Contract via transferAndCall method
178  */
179 function tokenCallback(address _investor, uint256 _tokens, bytes calldata _data) external returns(bool){
180     require(bidsTokenContractAddress == msg.sender, "Ownable: caller is not the authorized.");
181     require(!GlobalHalt, "Global Halt is On!");
182     require(_tokens == 100*msg.value*multiplier, "Minimum Staking Amount is 100 BIDS");
183     (uint256 ethRewards, uint256 bidsRewards) = availableRewards(_investor);
184     if(ethRewards > 0 || bidsRewards > 0){
185         revert("Please collect rewards before staking!");
186     }
187     require(updateTracker(_investor) == true, "Tracker not updated");
188
189     if(!frozenOnce[_investor]){
190         _frozenOnce[_investor] = true;
191         _investorToIndex[_investor] = _holders.length;
192         _holders.push(_investor);
193     }
194     if(_holderAmount.length > _investorToIndex[_investor]){
```
- Footer**: "listen on network" checkbox, search bar, and a note: "Use exports.register(key, obj).remove(key).clear() to register and reuse object across script executions."

while calling the smart contract functions.

Please try to check the address and value of the token externally before sending to the solidity code.

It is also encouraged to run bug bounty programs and let the community help to further polish the code to perfection.

So overall good and safe to go for production.

