Ether Authority

BETBEETLE BETTING SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	BetBeetle Team (https://betbeetle.com)
Prepared on:	28/04/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Standard
audit@etherauthority.io	

.

Table of contents

Project file

Name	Smart Contract Code Review and Security Analysis Report for BetBeetle Betting Contract
Platform	Binance Smart Chain / Solidity
File 1	Bet.sol
File 1 MD5 Hash	E1474EBB81C5A13D3710DF5288530A29
BscScan Contract Code URL	https://testnet.bscscan.com/address/0xe19deef1af8784e11b1e59eca55469990bc6e625#code
Audit Date	28/04/2021
Revised Contract Code	https://bscscan.com/address/0x10169324b0434345b0ffed5ff00964e8842dfa67#code
Revision Date	10/06/2021

Introduction

We were contracted by the BetBeetle team to perform the Security audit of the Betting smart contracts code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on 28/04/2021.

The Audit type was Standard Audit. Which means this audit is concluded based on Standard audit scope, which is one security engineer performing an audit procedure for 2 days. This document outlines all the findings as well as an AS-IS overview of the smart contract codes.

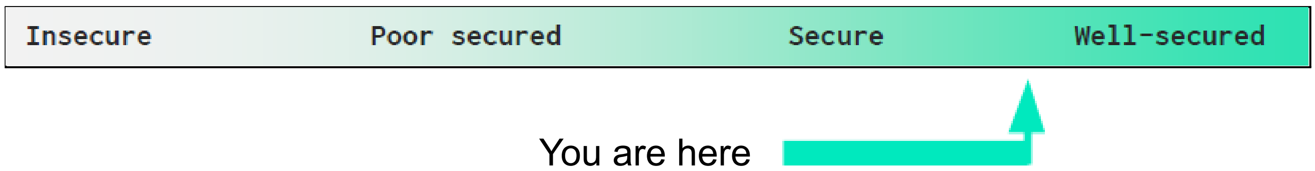
Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	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	Rectified
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Rectified
	DoS possibility	Rectified
	Other programming issues	Passed
Code Specification	Function 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
	High consumption 'for/while' loop	Moderated
	High consumption 'storage' storage	Passed
	"Out of Gas" Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	Tokenomics issues	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: PASSED

Executive Summary

According to the **standard** audit assessment and after revision, Customer`s solidity smart contract is close to **Well Secured**.



We used various tools like MythX, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all identified issues can be found in the Audit overview section.

We found 2 Critical, 2 high, 0 medium, 2 low and some very low level issues. These issues are fixed/acknowledged in the revised version of the code.

Code Quality

BetBeetle betting smart contract consists of 1 core solidity file. This smart contract also contains Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the BetBeetle protocol are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in the BetBeetle protocol.

The BetBeetle team has **not** provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Overall, code parts are well commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

We were given BetBeetle Betting smart contracts code in the form of a file. The hash of that file and its web link are mentioned above in the table.

As mentioned above, most code parts are well commented. so anyone can quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Another source of information was its official website betbeetle.com, which provided rich information about the project architecture and tokenomics.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well known industry standard open source projects. And their core code blocks are written well.

Apart from libraries, BetBeetle Betting smart contract interacts with another token contract as external smart contract calls.

AS-IS overview

BetBeetle is a Betting smart contract. It has features as users can create bets, participate in other existing bets, etc. Following are the main components of core smart contracts.

Bet.sol

(1) Usages

(a) using SafeMath for uint256;
(b) using Address for address;
(c) using SafeERC20 for BasicToken;

(2) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	constructor	write	Passed	No Issue	Passed
2	addAdmin	write	Passed	No Issue	Passed
3	setMinBet	write	Passed	No Issue	Passed
4	createBet	write	Passed	No Issue	Passed
5	updateBetsStates	write	Infinite Loop Possibility	Keep array length limited	Passed with consent
6	sendBet	write	Passed	No Issue	Passed
7	getBets	read	Infinite Loop Possibility	Keep array length limited	Passed with consent
8	getPositivesBets	read	Infinite Loop Possibility	Keep array length limited	Passed with consent
9	getBet	read	Passed	No Issue	Passed
11	checkAdmin	read	Passed	No Issue	Passed
12	getBetsCount	read	Passed	No Issue	Passed
13	getBetTotalOptions0	read	Passed	No Issue	Passed
14	getBetTotalOptions1	read	Passed	No Issue	Passed
15	getBetters	read	Passed	No Issue	Passed
16	withdrawERC20	write	Passed	No Issue	Passed
17	withdrawETH	write	multiple withdraw	see audit finding	Rectified

18	depositERC20	write	logical issue	see audit finding section	Rectified
19	cancelBet	write	DoS possibility	see audit finding section	Rectified
20	declareWinner	write	Infinite Loop Possibility	Keep array length limited	Passed with consent

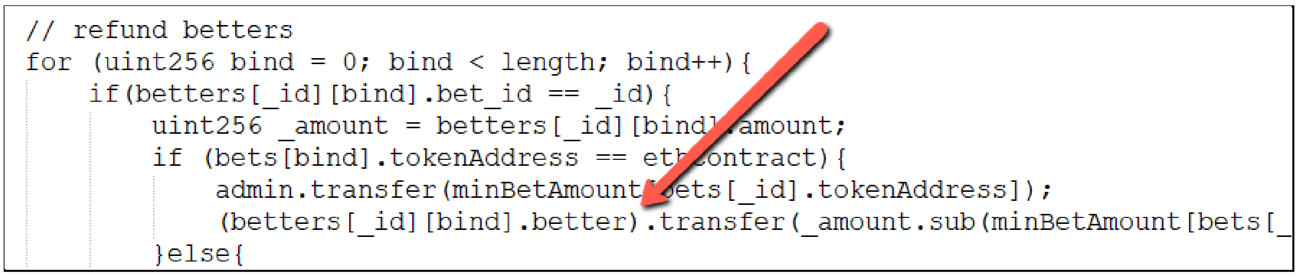
Severity Definitions

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

(1) Denial of Service (DoS) possibility



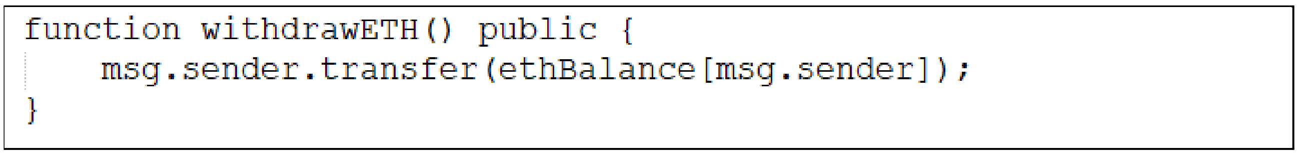
```
// refund betters
for (uint256 bind = 0; bind < length; bind++){
    if(betters[_id][bind].bet_id == _id){
        uint256 _amount = betters[_id][bind].amount;
        if (bets[bind].tokenAddress == ethContract){
            admin.transfer(minBetAmount[bets[_id].tokenAddress]);
            (betters[_id][bind].better).transfer(_amount.sub(minBetAmount[bets[_
        }else{
```

In the cancelBet function, If an attacker has used a contract address as better, which does not have fallback function, then it reverts. This makes the entire function call to revert and it will never let admin to refund the users for that particular bet ID.

Solution: Best approach is to check if the better wallet is a smart contract or normal wallet. And prevent smart contract wallets from placing the bets.

Update: This issue is fixed in revised smart contract code.

(2) Users can withdraw multiple times



```
function withdrawETH() public {
    msg.sender.transfer(ethBalance[msg.sender]);
}
```

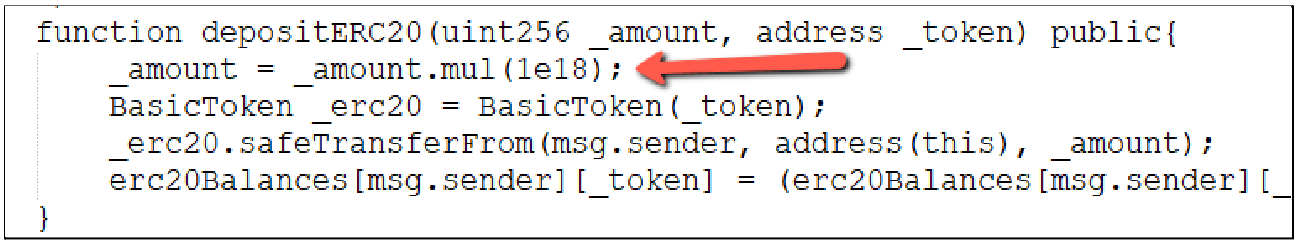
If a user has positive balance in ethBalance mapping, then he can keep withdrawing until the entire fund is drained from the smart contract.

Solution: Update the ethBalance mapping with zero, before sending the fun to the user. Please use Checks-Effects-Interactions pattern to prevent reentrancy attacks.

Update: This issue is fixed in revised smart contract code.

High

(1) No fraction value possible



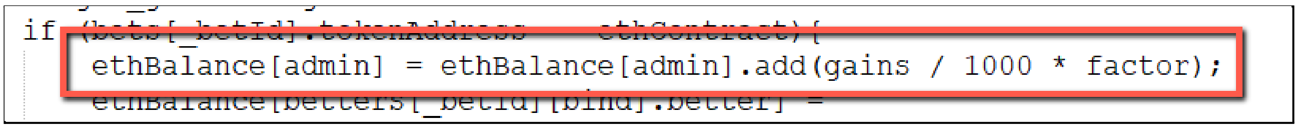
```
function depositERC20(uint256 _amount, address _token) public{
    _amount = _amount.mul(1e18);
    BasicToken _erc20 = BasicToken(_token);
    _erc20.safeTransferFrom(msg.sender, address(this), _amount);
    erc20Balances[msg.sender][_token] = (erc20Balances[msg.sender][_
}
```

_amount is being multiplied with 1e18 inside the function. Current scenario will prevent users from sending fractional amounts such as 15.50, etc.

Solution: Ideally, that _amount must be inputed after multiplying from the client side.

Update: This issue is fixed in revised smart contract code.

(2) Multiplication after division


```
if (bets[_betId].tokenAddress == ethContract){
    ethBalance[admin] = ethBalance[admin].add(gains / 1000 * factor);
    ethBalance[betters[_betId][bind].better] =
```

Line number #675, #680, #681, #683, #684 has a formula which has multiplication after division. Solidity being a primitive language, it may raise a scenario when the value of that formula becomes 0 or unexpected values.

Solution: Please put all the multiplications first and then put any division values.

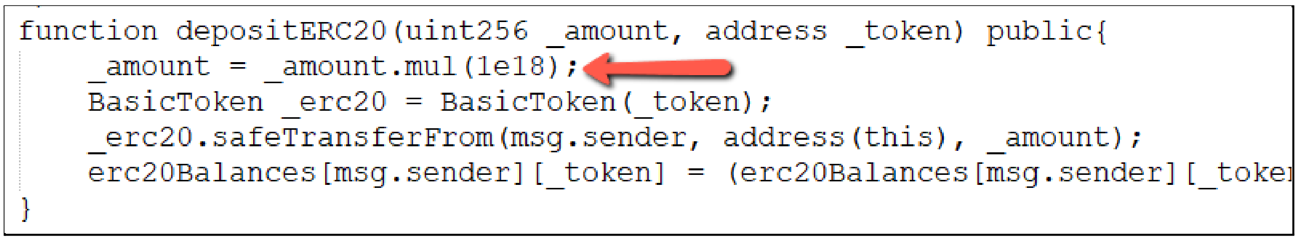
Update: This issue is fixed in revised smart contract code.

Medium

No medium severity vulnerabilities were found.

Low

(1) Tokens having decimals other than 18 will not work


```
function depositERC20(uint256 _amount, address _token) public{
    _amount = _amount.mul(1e18);
    BasicToken _erc20 = BasicToken(_token);
    _erc20.safeTransferFrom(msg.sender, address(this), _amount);
    erc20Balances[msg.sender][_token] = (erc20Balances[msg.sender][_toke
}
```

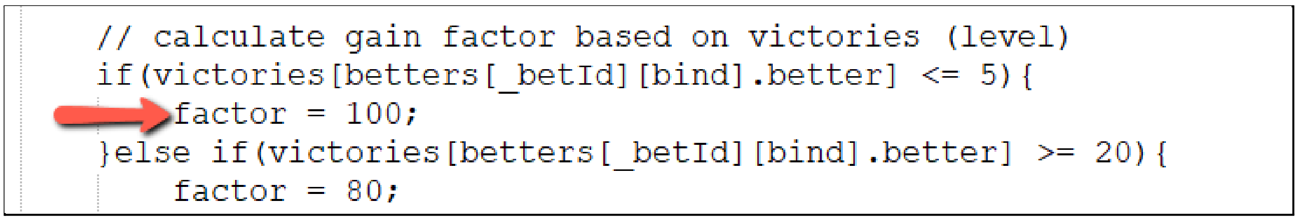
Since here, it is hard coded multiplying with 1e18, so any tokens having decimals other than 18 will not work.

Solution: Never multiply for the decimals inside the function. Instead, just provide the token amount which is already having decimal multiplied.

Update: This issue is fixed in revised smart contract code.

Very Low / Best practices

(1) State variable used instead of local variable



```
    // calculate gain factor based on victories (level)
    if(victories[betters[_betId][bind].better] <= 5){
        factor = 100;
    }else if(victories[betters[_betId][bind].better] >= 20){
        factor = 80;
```

In the declareWinner function, factor variable was used which is a state variable. So everything reading or updating that variable will consume more gas.

<u>Solution</u>: Declare a local variable and use it inside the function. It will save gas cost.

<u>Update</u>: This issue is fixed in revised smart contract code.

(2) Since this is a Betting contract, then there is no need for the token contract code. If you need to interact with any token contrac, then just a simple interface is enough. If that code block is not needed, then better to remove it to make the code clean.

<u>Update</u>: This issue is fixed in revised smart contract code.

(3) Important functions lack events

- setMinBet
- createBet
- sendBet
- withdrawERC20
- withdrawETH
- depositERC20
- cancelBet
- declareWinner

<u>Update</u>: This issue is fixed in revised smart contract code.

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. We found some major issues, which were rectified in the revised version of the code. **So it is good to go for the production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, 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 the reviewed contract, based on extensive audit procedure scope is "**Well Secured**".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:
In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:
Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract 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).

Due to the fact that 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 also suggest to conduct a bug bounty program to confirm the high level of security of this smart contract.

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.

EtherAuthority