**Introduction**

In this Smart Contract audit we’ll cover the following topics:

1. Disclaimer
2. Overview of the audit and nice features
3. Attack made to the contract
4. Critical vulnerabilities found in the contract
5. Medium vulnerabilities found in the contract
6. Low severity vulnerabilities found
7. Summary of the audit

**1. Disclaimer**

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

**2. Overview**

The project has 5 files, the BetData.sol, GameData.sol, Coordinator.sol, SafeMath.sol, Migrations.sol file which contains around 1000 lines of Solidity code.

The solidity code should facilitate peer to peer gambling transactions. First, the site pulls data about sports games and displays this to the user, then this data is also sent to an instance of the contract. There are two types of users on the site - bet makers and bet takers. Bet makers act like a book keeper, they click on the game they want to bet on, and set the line they want to offer, for example -110 or 1.91(in decimal odds). They then designate an Ethereum volume and send the eth to the smart contract via metamask. These offers are then displayed to the bet takers, who can begin to “fill” the bet makers orders. Ex: bet maker offers Yankees winning at 1 eth at -110, 10 bet makers betting 0.1 eth each “take” the bet at -110. They send their eth to the contract as well. At the end of the game, the results are updated, and sent to the contract. The contract should pay out to the users metamask wallets individually. Additionally, the bet maker should be allowed to cancel his offer, and any volume remaining in the contract which is Unfilled (meaning no taker has filled it) will Be returned at the end of the game. Betting shuts off at game time.

***Nice Features:***

1. The contract provides a good suite of functionality that will be useful for the entire contract: It uses [SafeMath](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol) library to check for overflows and underflows which is a pretty good practice.
2. Modifiers to check the overflows and underflows.
3. Good amount of check for verifying that the bet function is used properly.

**3. Attacks made to the contract**

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

* **Over and under flows** An overflow happens when the limit of the type variable uint256, 2 \*\* 256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

For instance, if I want to assign a value to a uint bigger than 2 \*\* 256 it will simple go to 0 — this is dangerous.

On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0. For example, if you subtract 0 - 1 the result will be = 2 \*\* 256 instead of -1.

This is quite dangerous. However This contract checks for overflows and underflows in **[OpenZeppelin's](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol)***[SafeMath](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol)* and there is no instance of direct arithmetic operations.

* ***Short address attack*** This attack affects ERC20 tokens, was discovered by the Golem team and consists of the following:

A user creates an Ethereum wallet with a trailing 0, which is not hard because it’s only a digit. For instance: 0xiofa8d97756as7df5sd8f75g8675ds8gsdg0

Then he buys tokens by removing the last zero:  
Buy 1000 tokens from account 0xiofa8d97756as7df5sd8f75g8675ds8gsdg

If the token contract has enough amount of tokens and the buy function doesn’t check the length of the address of the sender, the Ethereum’s virtual machine will just add zeros to the transaction until the address is complete.

The virtual machine will return 256000 for each 1000 tokens bought. This is a bug of the virtual machine that’s yet not fixed so whenever you want to buy tokens make sure to check the length of the address.

This contract isn’t vulnerable to this attack since it’s not an ERC20 token.

* ***Reentrancy*** No Issues
* ***Costly Loop*** No Issue. There is one loop in GameData.sol file at line 113. This will not create problems when the number of teams are not much. Since Ethereum is very resource-constrained environment, prices per computational step are orders of magnitude higher than the centralized providers. Moreover, Ethereum miners impose a limit on the total number of gas consumed in a block. If **array.length** is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed. This becomes a security issue, if an external actor influences **array.length**.
* ***DoS by external function call in require*** No Issues

**4. Critical vulnerabilities found in the contract**

There aren’t any critical issues in the smart contracts audited.

**5. Medium vulnerabilities found in the contract**

* In the file GameData.sol file, in addSport(), should not allow to enter the same sport id again. Should have a check to restrict user to enter the sport id which already exists.
* You’re specifying a pragma version with the caret symbol (^) up front which tells the compiler to use any version of solidity bigger than 0.4.15 .

This is not a good practice since there could be major changes between versions that would make your code unstable. That’s why I recommend to set a fixed version without the caret like 0.4.15.

* I would recommend you to use latest versions of solidity compiler instead of older versions as latest versions contains many critical bug fixes. Using compiler version 0.4.24 is recommended.
* In SafeMath.sol file, the version is 0.4.18. The compiler version should be same in all the solidity files.

**6. Low severity vulnerabilities found**

* In GameData.sol file, Line 55, 59, 158, 121, 137, 153, 170, 162, 112: The function is declared as **constant.** Currently, for functions the **constant** modifier is a synonym for **view** (which is the preferred option). Consider using **view** for functions and **constant** for state variables.
* Use view or pure instead of constant- This will increase the readability of the functions and will more clearly specify the purpose of the function. view functions means that they will not make any changes to the storage but will make reads from the storage.

Pure functions mean they will neither make any changes to the storage nor will read from the storage.

Exchange function mul() of SafeMath to

function mul(uint256 a, uint256 b) internal pure returns (uint256) {

uint256 c = a \* b;

require(a == 0 || c / a == b);

return c;

}

* Assign all functions the visibility modifier like public, internal or external. By default the modifier is assumed to be public by the solidity compiler.
* The code is not well commented and this is usually not a good industry practice. Each and every function should be well commented and all the parameters passed should be well defined. For example: in BetData.sol file, function makeBetOnTeam() is not commented. All the parameters passed inside the function should have the clear comments describing their role in the function.

**7. Summary of the audit**

* Overall the logic implemented seems quite good. My final recommendation would be to pay more attention to the visibility of the functions since it’s quite important to define who’s supposed to executed the functions and to follow best practices regarding the use of assert, require etc. (which you are doing ;).
* The mechanism to bet and distribute rewards is quite simple so it shouldn’t bring major issues.
* This is a secure contract that will store safely the funds while it’s working.