**Security measures to secure smart contracts**

security measures employed so contracts are not susceptible to the common kinds of attacks include:

**Restricting Access**

Using modifiers to restrict access restricting access to withdraw methods ensuring only players can make withdrawals:

modifier onlyPlayer(){

require(player[msg.sender].isPlayer == true);

\_;

}

**Safeguarding against Re-entrancy**

When withdrawAll() is executed, player[msg.sender].balance and player[msg.sender].placedBets values are set to ‘0’ before transferring the availableAmount, which is the sum of both values. External calls to this function are broken because the condition require(availableAmount >= 0) is not true.

A similar method is applied to withdrawBalance().

player[msg.sender].balance -= \_withdrawalAmount leaves us with a new balance. However there is also a reEntrancyMutex boolean variable which acts as a toggle to indicate when the point of withdrawal has been met. The reEntrancyMutex variable is required to be false before continuing to execute the rest of the code. Just before a transfer is made, the variable is then set to true. This ensures that on re-entrancy execution is broken by require(!reEntrancyMutex). The variable is released once execution continues on from the transfer line. Although the require(availableAmount >= \_withdrawalAmount) would prevent the method from running if the statement was not true, it would not stop a bad actor from draining the players balance while it is true. So the re-entrancy mutex acts to control the flow of the transfer.

**Safeguarding against Denial of service attacks**

Employing the pull over push pattern confines withdrawal tasks to the withdraw method and requires an address to request funds rather than automatically pushing funds to an address. Management of funds within the contract (i.e movement of funds from player[msg.sender].balance to player[msg.sender].placedBets) is handled separately within the placeBet(), undoBet() and playGame() methods.