Avoiding Common Attacks

Security measures were taken to prevent common attacks.

Visibility modifiers

It was ensured that the proper visibility modifier was applied to functions and variables. Functions which needed to be called externally were assigned the `public` modifier.

All variables have been assigned the `private` modifier to prevent (or better minimize the risk) the values to be tampered with externally. An important example is the `croupier` variable which defined the owner of the smart contract. Appropriate getters were created if the value of the private variables is required (source).

```
// Private variable
uint private tableStakeInWei;

// Public getter
function getTableStake() public view returns (uint) {
    return tableStakeInWei;
}
```

Access Control

Some functions should not be done by everyone so a function modifier called `ownerRequired` was created. This requires the sender's to match the owner's (croupier's) address (source).

```
// Function modifier which requires the caller of the function to be the owner of
the contract.
modifier ownerRequired {
    require(
        msg.sender == croupier,
        "Only owner can call this function."
    );
    _;
}

// Closes the table for betting.
// Only the croupier is allowed to close the table.
function closeBets() public ownerRequired {
    // We put a require so we do not spend unnecessary gas if the method is called
consecutively.
    require(isTableOpen,"The table is already closed for bets.");
    isTableOpen = false;
}
```

Force Ether

This smart contract is susceptible to a forced ether attack. Reason be it the `withdrawWinnings` functions uses the balance as guard.

An attacker can use the `selfdestruct` method of an other contract and putting the address of the roulette smart contract as the beneficiary. This way the roulette contract's fallback function is not triggered and the balance will be updated.

Although the balance is used as guard, several other guards were placed against the `withdrawWinnings` functions to mitigate the risk of attack or exploitation (source).

`Code snippet found under Reentrancy.`

Reentrancy

Reentrancy attacks were mitigated in two ways. In the `withdrawWinnings` function, the sender's balance is set to zero BEFORE the money is transferred. Secondly, for the function to be complete or successful, the `send` command is placed in a `require` (source).

```
function withdrawWinnings() public {
    require(!isTableOpen,"The table is not closed for betting.");
    require(winningNumber > -1, "No winning number is set.");
    require(playerWinnings[msg.sender] != 0, "There are no withdrawable
winnings.");
    require(address(this).balance >= playerWinnings[msg.sender], "There are not
enough funds to transfer.");
    uint transferAmount = playerWinnings[msg.sender];

    // Set the winnings mapping to 0 before the transfer.
    playerWinnings[msg.sender] = 0;
    require(msg.sender.send(transferAmount));
}
```