

Use of Smart Contracts to Facilitate Semi-Trustless Betting (V2)

Chris Cushman

chris@snowape.io


<https://github.com/cmancushman/Snow-Ape-Smart-Contract>

Abstract. Gaming and user-driven websites not yet at-scale suffer from deposit/withdrawal delays and struggle to earn user trust. Use of funds is not apparent, nor is liquidity in a centralized custodial service such as Robinhood, DraftKings, Fanduel, etc. They are used and trusted because of their brand awareness and reputation, not because of their transparency or ability to handle funds. This situation that faces less prominent custodial entities can be improved by moving the custody of funds from internal reserves to an external smart contract.

Implementation

Deposits

Users can easily send Ethereum to the smart contract address by calling the `deposit` function. This is a payable transaction that deposits funds into the smart contract and increments the user's on-chain balance.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains Solidity code for a deposit function.

```
/**
 * @dev Deposit funds into address.
 */
function deposit() public payable {
    balances[msg.sender] += msg.value;
    totalPlayerValue += msg.value;
}
```

Withdrawals

Users can just as easily withdraw Ethereum from the smart contract address by calling the `withdraw` function. This is a transaction that allows the user to specify the withdrawal amount, checks if the user has a sufficient balance to make the withdrawal, and immediately transfers the requested balance to the user's address. Note, the balance of the smart contract itself also needs to match the withdrawal amount for the withdrawal to be successful. We touch on this in the "reserves" section.

```
/**
 * @dev Withdraw funds from address.
 * @param withdrawalAmount the amount to be withdrawn
 */
function withdraw(uint256 withdrawalAmount)
    public
    sufficientFunds(withdrawalAmount)
    returns (bool)
{
    require(
        getBalance() >= withdrawalAmount,
        "Insufficient reserves to process this withdrawal."
    );
    address payable withdrawalAddress = payable(msg.sender);
    (bool sent, ) = withdrawalAddress.call{value: withdrawalAmount}("");

    if (sent) {
        balances[msg.sender] -= withdrawalAmount;
        totalPlayerValue -= withdrawalAmount;
    }
    return sent;
}
```

Initializing Games

If users have a sufficient balance, they can initialize a game to be played off-chain by calling the function `initializeOnePlayerGame`. The user's balance is deducted by the `betSize` parameter, in addition to a fee to be paid for the gas spent by the owner of the contract to verify the result of the game. Then, an entry for the game id is stored on-chain, and the `bettingPoolBalance` is incremented by the bet amount. Once again, sufficient reserves on the side of the smart contract are required to initialize games, as the contract itself pays the gas fee to verify the outcome.

```
/**
 * @dev Initialize one player game, reduces balance, deposit into betting pool,
 * and adds an entry for the game id.
 * @param betSize the amount of money being bet
 * @param gameId the unique id of the game
 */
function initializeOnePlayerGame(uint256 betSize, string memory gameId)
    public
    sufficientFunds(betSize + GAME_FEE)
    betSizeIsAllowed(betSize)
{
    require(
        getBalance() >= GAME_FEE,
        "Insufficient reserves to play game."
    );
    address payable gameFeeBeneficiary = payable(owner());
    (bool sent, ) = gameFeeBeneficiary.call{value: GAME_FEE}("");

    if (sent) {
        balances[msg.sender] -= (betSize + GAME_FEE);
        totalPlayerValue -= (betSize + GAME_FEE);
        bettingPoolBalance += betSize;
        games[gameId] = true;
    }
}
```

Verifying Games

The owner of the contract then verifies if a game was won or lost by a user by calling the function `determineOutputOfOnePlayerGame` and adjusts the user's balance accordingly. If the game was won, increment the user's balance by the `betPayout` parameter, and decrease the `bettingPoolBalance` by the same amount. Otherwise, do nothing, as the player's balance was already deducted by the initial bet amount.

```
/**
 * @dev Validates the result of a bet and pays out a player if they won.
 * @param player the address of the player
 * @param gameId the unique id of the game
 * @param playerDidWin true if the winner did win, otherwise false
 * @param betPayout the amount to pay out
 */
function determineOutputOfOnePlayerGame(
    address player,
    string memory gameId,
    bool playerDidWin,
    uint256 betPayout
) public onlyOwner bettingPoolHasFunds(betPayout) {
    require(games[gameId] == true, "Game not valid.");

    if (playerDidWin == true) {
        balances[player] += betPayout;
        totalPlayerValue += betPayout;
        bettingPoolBalance -= betPayout;
    }

    games[gameId] = false;
}
```

Reserves

Three functions are used to calculate the contract reserves: `getBalance`, `getTotalPlayerValue`, and `getBettingPoolBalance`. The first function, `getBalance`, determines the actual balance on the smart contract, or the amount of money currently stored on the smart contract that is ready to be withdrawn. This number can also be calculated by the sum of the other two functions (`getTotalPlayerValue` and `getBettingPoolBalance`). The second function, `getTotalPlayerValue`, determines the sum of all player balances, or the total amount players are owed in the event of a unanimous withdrawal. This number will always be less than or equal to the total balance of the contract. The third and final function, `getBettingPoolBalance`, determines the amount of money currently locked in the betting pool. The betting pool is the balance into which users deposit when they initialize bets and is the liquidity available to pay out players that win those bets. The mandate of the contract owner is to fund the betting pool such that `getBettingPoolBalance` is high enough to fulfill player payouts. If the betting pool has insufficient liquidity, players will see delays in the time it takes for them to receive payouts. Both current and potential users can use these three functions to check on the total value locked (TVL) in the contract, as well as the liquidity of the betting pool, and to make an informed analysis of the platform before playing.

```
/**
 * @dev Gets the balance of the smart contract.
 * @return currentBalance
 */
function getBalance() public view returns (uint256) {
    uint256 currentBalance = address(this).balance;
    return currentBalance;
}

/**
 * @dev Gets the total sum of all player balances.
 * @return totalPlayerValue
 */
function getTotalPlayerValue() public view returns (uint256) {
    return totalPlayerValue;
}

/**
 * @dev Gets the total value locked in the betting pool.
 * @return bettingPoolBalance
 */
function getBettingPoolBalance() public view returns (uint256) {
    return bettingPoolBalance;
}
```

Checks & Balances

Because the validation of game outcomes is centralized (only the contract owner can verify the result of a game), this smart contract is not fully trustless. The users are placing trust in the entity verifying the games to do so correctly, and not to count games against them that they rightfully won. However, they do not have to place trust in the contract owner as far as custody of funds is concerned. The owner of the smart contract only has direct access to the betting pool, not to other users accounts, or to the liquidity of the contract beyond the betting pool size. This means is all of the users simply used the smart contract as a bank, and declined to make any bets, the contract owner would have no power over funds whatsoever, since all a bad-acting contract owner can do is siphon from the current betting pool. If a user only bets 1% or less of their account size at once, the amount of trust they need to give to the contract owner is very little. Conversely, the financial gain that the contract owner sees from sabotaging the betting pool would not likely be worth the reputational damage they receive from their disgruntled customers. Thus, responsible bettors that only place a small fraction of their portfolios into each bet can keep the contract owner in check and confidently custody their funds in the smart contract.

Conclusion

Although it is not a fully trustless contract, it provides transparency and limits third-party control of user funds, which can bolster user trust and consequently market acquisition for newer gaming entities lacking brand recognition.