Use of Smart Contracts to Facilitate Semi-Trustless Betting

Chris Cushman

chrisfcushman@gmail.com

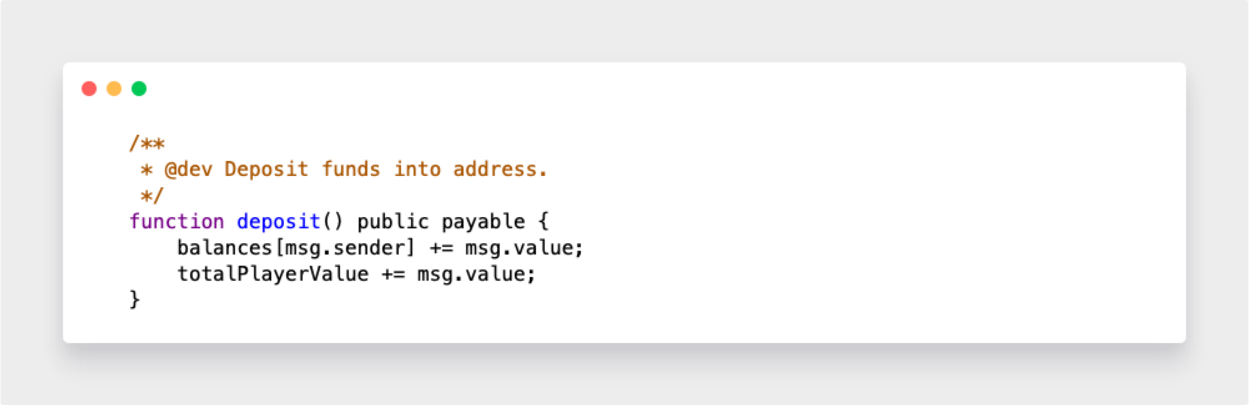
https://github.com/cmancushman/Money-Carlo-Smart-Contract

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

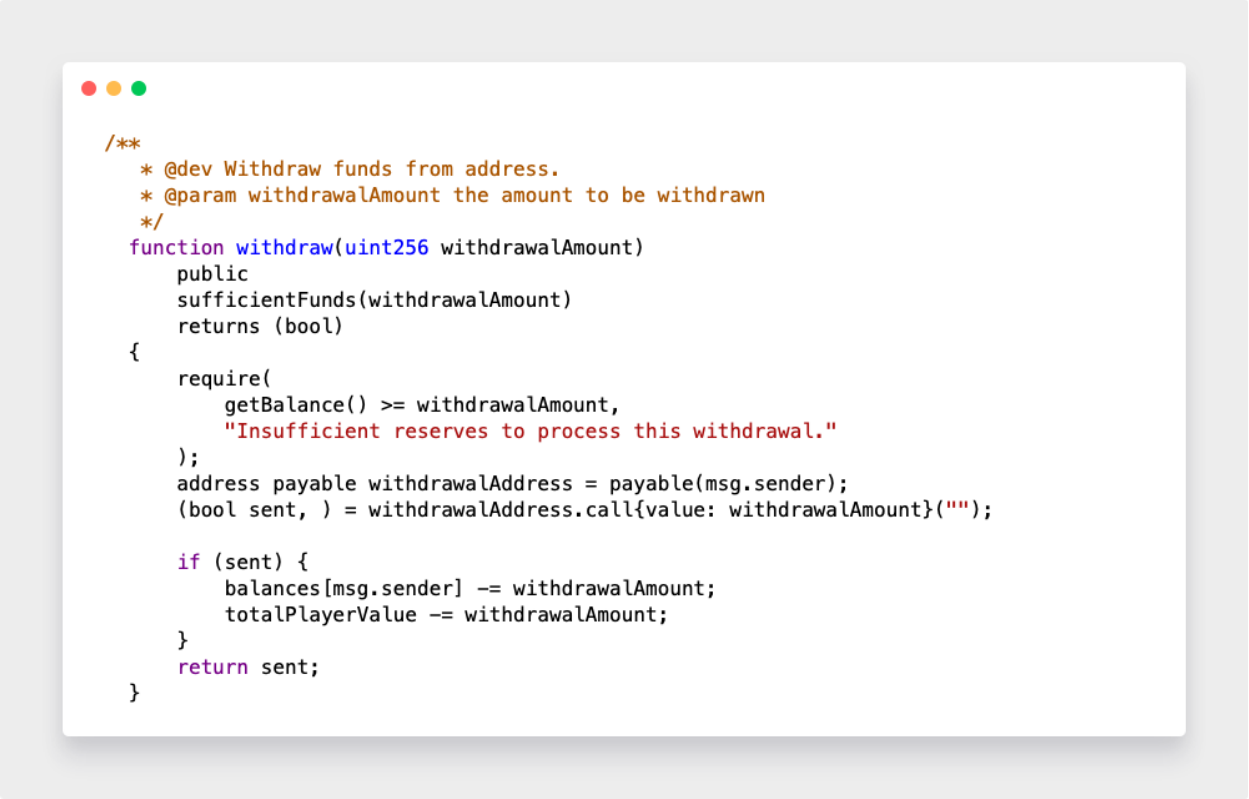
Users can easily send Ethereum to the smart contract address by calling the deposit function. This is a payable transaction that deposits funds into the smart contract and increments the user's on-chain balance.



```
/**
 * @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.

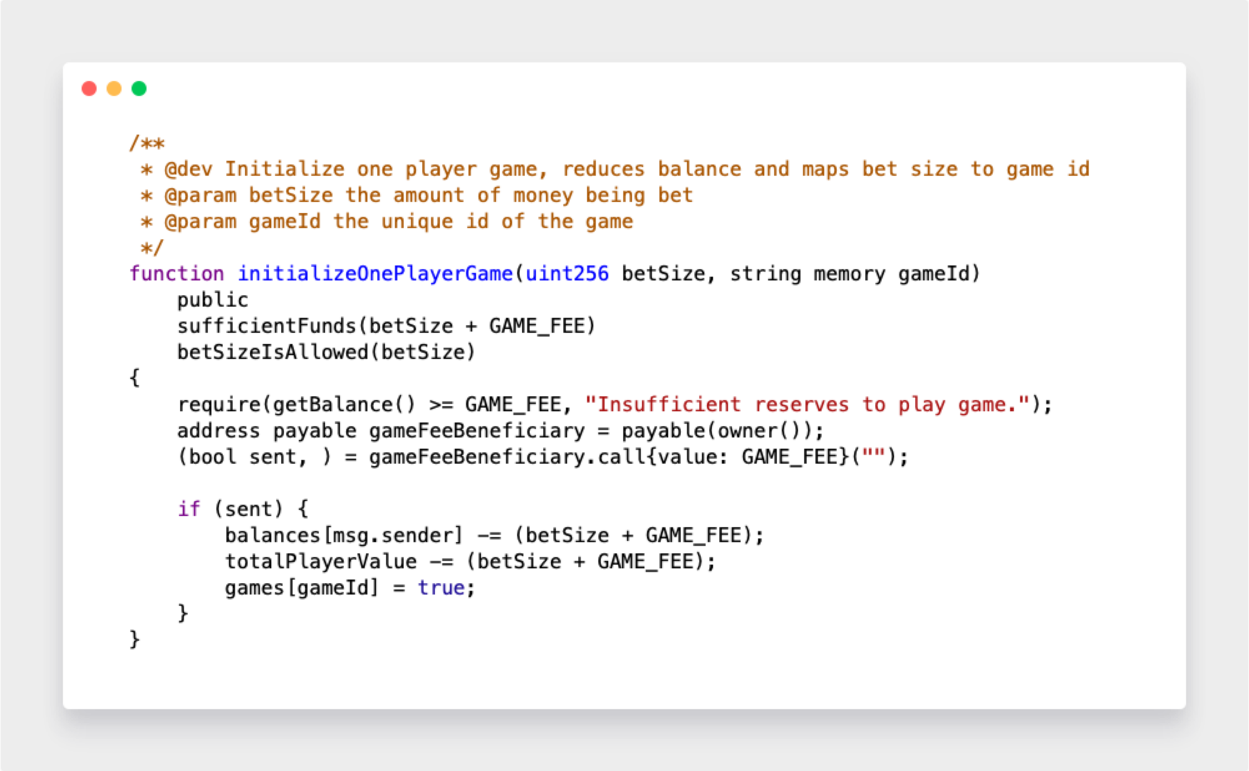


```
/**
   * @dev Withdraw funds from address.
   * @param withdrawalAmount the amount to be withdrawn
   */
  function withdraw(uint256 withdrawalAmount)
      public
      sufficientFunds(withdrawalAmount)
      returns (bool)
  {
      require(
          getBalance() >= withdrawalAmount,
          "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. 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.

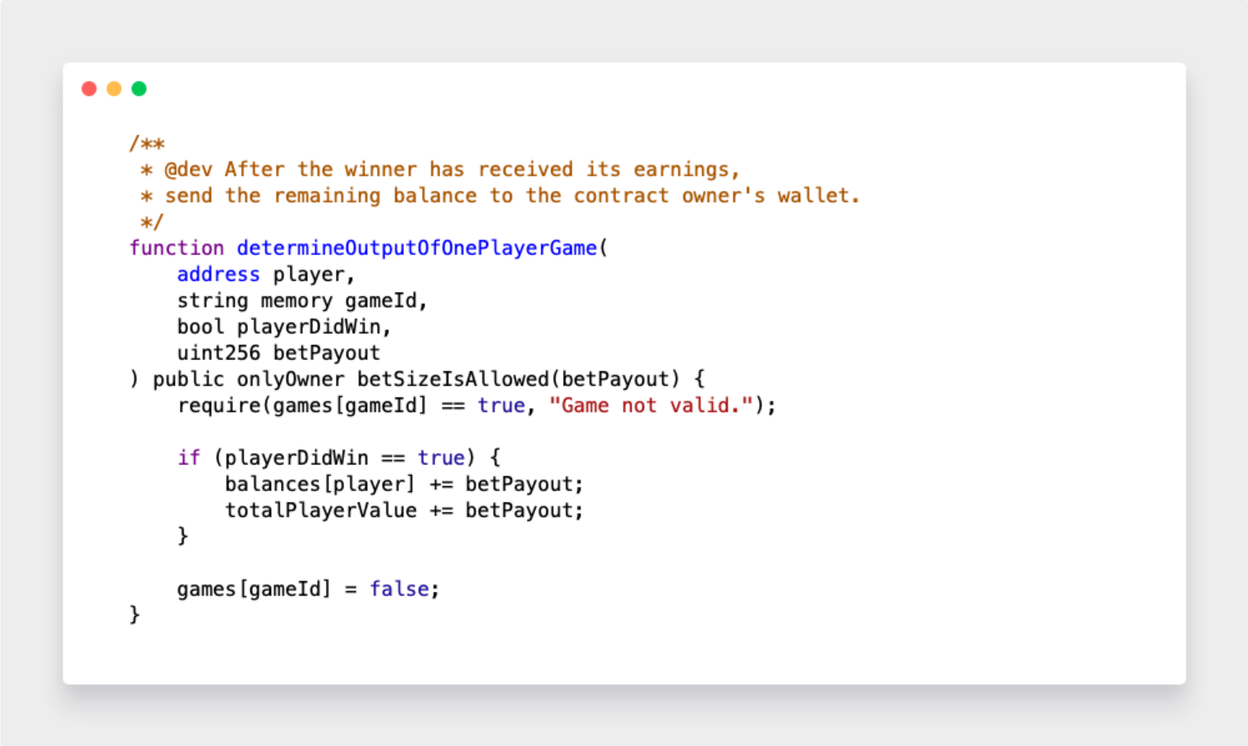


```
/**
 * @dev Initialize one player game, reduces balance and maps bet size to game id
 * @param betSize the amount of money being bet
 * @param gameId the unique id of the game
 */
function initializeOnePlayerGame(uint256 betSize, string memory gameId)
    public
    sufficientFunds(betSize + GAME_FEE)
    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);
        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. Otherwise, do nothing, as the player’s balance was already deducted by the initial bet amount.



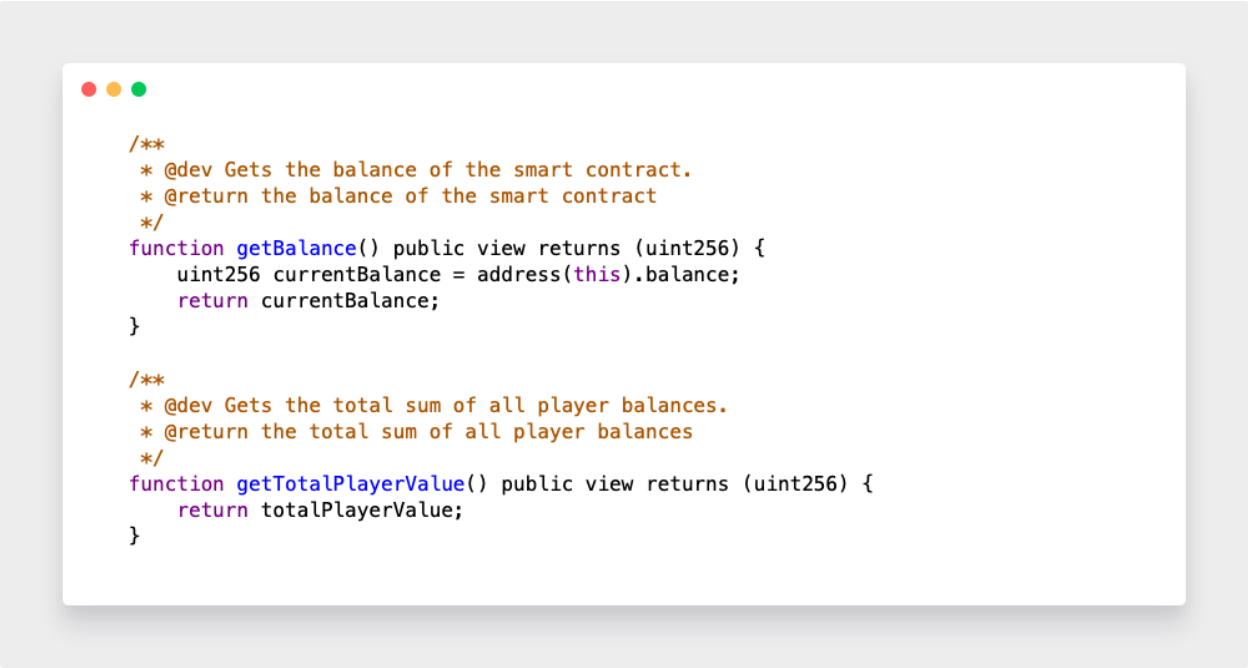
```
/**
 * @dev After the winner has received its earnings,
 * send the remaining balance to the contract owner's wallet.
 */
function determineOutputOfOnePlayerGame(
    address player,
    string memory gameId,
    bool playerDidWin,
    uint256 betPayout
) public onlyOwner betSizeIsAllowed(betPayout) {
    require(games[gameId] == true, "Game not valid.");

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

    games[gameId] = false;
}
```

Reserves

Two functions are used to calculate the contract reserves: getBalance and getTotalPlayerValue. The first function 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. The second function determines the sum of all player balances, in other words the total amount players are owed in the event of a unanimous withdrawal. The mandate of the contract owner is to fund the smart contract such that getBalance is suitably high in comparison to getTotalPlayerValue. For some, this may mean that the contract balance is always greater than or equal to total player value, so that users can feel comfortable withdrawing any time they need to. This is likely the case for newer projects with less liquidity, as they will want to ensure user trust. More established entities may be comfortable running on lower reserves, as perhaps they have plenty of outside liquidity to fund the contract and more brand trust. Typical banks run at reserve requirements of 10%, for reference. Both current and potential users can use these functions to check on the total value locked (TVL) in the contract, as well as the reserve levels of the contract, to make an informed analysis of the platform before playing.



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

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

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. They are also placing trust in the entity not to initialize and verify winning games for themselves, as to siphon funds out of the smart contract. If this mechanism were not limited, there would be a direct backdoor that would allow the contract owner to liquidate all funds at once by doing the following: initialize a bet from their own wallet, set the payout to the size of the contract balance, "verify" that they won the game, and withdraw their balance. However, in the contract the functionality of game verification is limited by MAX_BET_SIZE. This limit only allows a user to win a payout that is less than or equal to the immutable maximum bet size. Thus, the most a bad actor could siphon from the smart contract in one transaction is less than or equal to the maximum bet size. This creates a bottleneck for a bad actor and buys users time to withdraw funds in the case of an exploit. The smaller MAX_BET_SIZE is set to, the harder it is for a bad actor to siphon funds. While this is not a perfect check, it adds a layer of protection to users, and helps establish trust for the contract owner.

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.