Security audit of smart contracts

Addresses of smart contracts on mainnet:

Lottery1ETH: *0x865EE5df064bc1F4A39B95B75e612dD86011d35b*

Lottery10ETH: *0x150dBfC384bA5C13c304EfD2Efee73Cc57cC2C16*

Lottery100ETH: *0xb02Ae0bd0e1431337fCe668d76A6BA4b6eCADD84*

RefStorage: *0x978275D7652a35DC8Df9ce6B62822Aea6A97589D*

Audit result:

In Lottery project's code **was not discovered** critical vulnerabilities and backdoors. Contracts can be used in real funds turnover.

Limitations associated with obtaining random numbers in smart contracts **acceptable** due to relatively small prizes. Contracts can be used in real funds turnover.

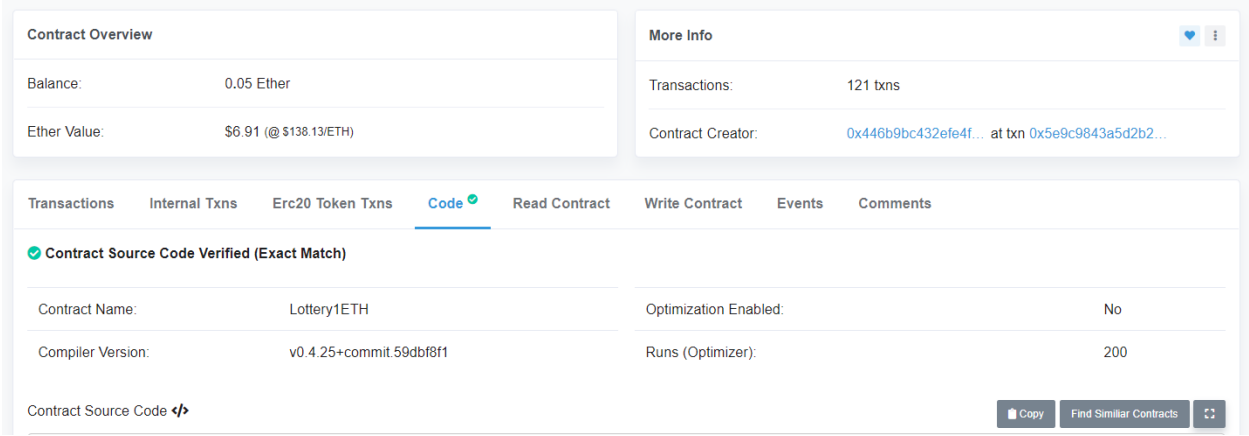
Contract owners **may not** suspend the current circulation of tickets, as well as may not prevent the distribution of prizes and change the key settings of circulation (except for the referral program).

Drawing can be initiated by any user (sending any number of ETH to the lottery address), after a few blocks after the purchase of the last ticket in the draw (see the description of the functional). Then (if the project is not paused), the sale of a new circulation automatically begins.

Note that one of the disadvantages of this code is the lack of commenting, which complicates the understanding of the functionality of contracts. We thank developers of contracts, who clarified some technical solutions.

Description

This audit reviewed contracts “Lottery”, as well as sub-contracts «RefStorage» and «Storage». The code of each smart contract is verified on EtherScan and opened for viewing.



All three contracts are used for 3 types of lotteries: 100 tickets, 1,000 tickets and 10,000 tickets. In all variations, a contract code is identical except for such parameters as the number of lottery tickets and winners, prize amounts and some numerical values in technical implementation of obtaining a pseudo source of entropy (see below for details).

> *Note: In this contract the library of safety computing SafeMath that prevent mistakes in smart contracts computing is NOT used. On the basis of computing's context, there are no possible cases of transfusion in a contract.*

> *Note: this contract may have had a potential backdoor in the prize mailout, but it uses "send" rather than "transfer" for ETH sending. This eliminates owner's possibility to resist funds mailout.*

Lottery's characteristics:

The ticket's price in all contracts is uniform and unchangeable: 0.01 ETH. (line 204)
In one transaction it is possible to buy only 1 ticket.

```
203
204     uint256 constant public PRICE = 0.01 ether;
205
```

Note: if the amount sent is less than the ticket's price, transaction will be rejected, if higher – change will be automatically returned (lines 261–263).

```
260
261         if (msg.value > PRICE) {
262             msg.sender.transfer(msg.value - PRICE);
263         }
264
```

Note: the lottery ticket in the project cannot be purchased from another smart contract, which excludes mass auto-purchase. A special modifier NotFromContract (line 227) is used for that.

```
226
227     modifier notFromContract() {
228         address addr = msg.sender;
229         uint256 size;
230         assembly { size := extcodesize(addr) }
231         require(size <= 0);
232         _;
233     }
234
```

Winners defining and prize drawing start after all tickets in the cycle are sold (100, 1000, 10000 tickets), then the cycle starts again (while first ticket is being purchased).

There are 3 types of winners: Silver, Gold, Brilliant.

Number of winners / winning amount (lines 215-218):

	100 tickets	1000 tickets	10000 tickets
Silver	10 / 0.02 ETH	20 / 0.1 ETH	40 / 0.5 ETH
Gold	2 / 0.05 ETH	5 / 0.2 ETH	10 / 1.0 ETH
Brilliant	1 / 0.50 ETH	1 / 5.0 ETH	1 / 50.0 ETH

```
215
216     uint256[] silver    = [10, 0.02 ether];
217     uint256[] gold      = [2,  0.05 ether];
218     uint256[] brilliant = [1,  0.50 ether];
219
```

> *Note: each draw is held independently among all lottery tickets, therefore, the same ticket can theoretically be the winner of several prizes.*

Referral programm:

There is a unified referral system in the project, which implemented by a separate RefStorage contract. The system is placed in a separate contract to ensure that the specified referrer is attached in all types of project's lotteries.

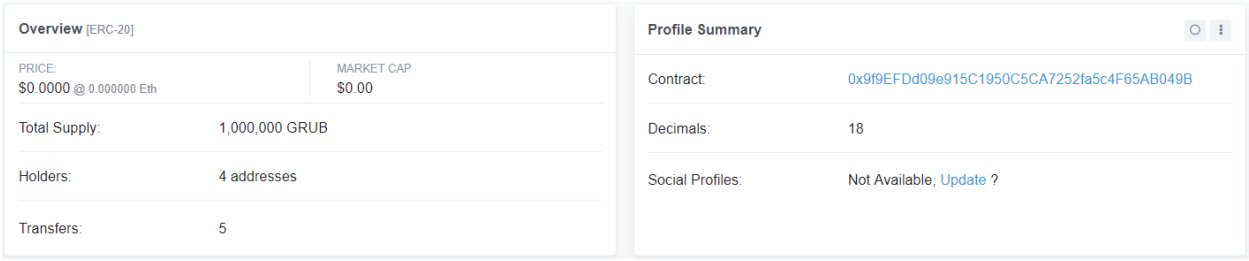
The referrer is indicated in "Data" box when buying a ticket. For data processing from "Data" box, a code contains standard method bytesToAddress (lines 357-362).

> *Note: referral program's key parameters in a contract can be changed by the owner. (lines 142-152)*

```
119
120     uint256 public prize = 0.00005 ether;
121     uint256 public interval = 100;
122
```

Bonus to a referrer and a lottery participant:
"Prize" variable (line 120): the default value is 0.00005 project tokens (50000000000000 excluding decimal places). (the actual information can be found in "read contract" tab in RefStorage contract: *0x978275D7652a35DC8Df9ce6B62822Aea6A97589D*)

> *Note: in the code (line 120), the prize is specified in the form of 0.00005 ether which is the form of writing the number 50000000000000, since the prefix "ether" in Solidity adds 18 zeros after the decimal point (that is how many decimal places the project token has).*



> *Note: GOLD RUBLE (GRUB) bonus token contract is 0x9f9EFDd09e915C1950C5CA7252fa5c4f65ab049b (line 139).*

```
137
138     constructor() public {
139         token = IERC20(address(0x9f9EFDd09e915C1950C5CA7252fa5c4F65AB049B));
140     }
141
```

This amount is issued once in a certain interval according to the number of purchased tickets, after specifying a referrer, and also issued to each winner of "Gold" type.

"Interval" variable (line 121): the default value is 100 tickets. (the actual information can be found in "read contract" tab in RefStorage contract: *0x978275D7652a35DC8Df9ce6B62822Aea6A97589D*)

Also, "interval" variable sets the minimum threshold of purchased tickets, allowing a participant to become a project referrer.

There are 3 key methods implemented in "Storage" contract code (all of them are available for calling only a limited list of addresses, according to the code's logic–lotteries contracts (lines 133–136):

NewTicket (154–164) – a mark on a new ticket purchase, if 100 tickets have been purchased since the referrer's indication, both a user and a referrer receive a bonus. The bonus is sent every 100 tickets if there are enough tokens on the contract balance.

```
153
154     function newTicket() external restricted {
155         players[tx.origin].tickets++;
156         if (players[tx.origin].referrer != address(0) && (players[tx.origin].tickets - players[tx.origin].checkpoint) % interval == 0) {
157             if (token.balanceOf(address(this)) >= prize * 2) {
158                 token.transfer(tx.origin, prize);
159                 emit BonusSent(tx.origin, prize);
160                 token.transfer(players[tx.origin].referrer, prize);
161                 emit BonusSent(players[tx.origin].referrer, prize);
162             }
163         }
164     }
165
```

AddReferrer (166–173) – pinning a referrer.

```
165
166     function addReferrer(address referrer) external restricted {
167         if (players[tx.origin].referrer == address(0) && players[referrer].tickets >= interval && referrer != tx.origin) {
168             players[tx.origin].referrer = referrer;
169             players[tx.origin].checkpoint = players[tx.origin].tickets;
170
171             emit ReferrerAdded(tx.origin, referrer);
172         }
173     }
174
```

Note: a referrer is indicated once and cannot be changed in the future.

SendBonus (175–181) – send a bonus to a user according to the code logic – to a winner Gold type.

```
174
175     function sendBonus(address winner) external restricted {
176         if (token.balanceOf(address(this)) >= prize) {
177             token.transfer(winner, prize);
178
179             emit BonusSent(winner, prize);
180         }
181     }
182
```

Storage contract:

For the system of Gold type winners selection's functioning, there is an identical contract "Storage" in each lottery type (lines 58-112). This contract is a temporary storage of data on the current leaders in the number of purchased tickets. This contract redeploys during each cycle of drawing (line 331). This technical solution is implemented in a contract because it is less expensive to create a new data warehouse than to clear the previous one.

Two methods are implemented in this contract:
Purchase (70-91) – a method available for calling only for the main lottery contract. The logic of this method saves the number of purchased tickets in this cycle for a user, and if necessary, erases previous records.

```
69
70      function purchase(address addr) public {
71          require(msg.sender == game);
72
73          amount[addr]++;
74          if (amount[addr] > 1) {
75              level[amount[addr]].push(addr);
76              if (amount[addr] > 2) {
77                  for (uint256 i = 0; i < level[amount[addr] - 1].length; i++) {
78                      if (level[amount[addr] - 1][i] == addr) {
79                          delete level[amount[addr] - 1][i];
80                          break;
81                      }
82                  }
83              } else if (amount[addr] == 2) {
84                  count++;
85              }
86              if (amount[addr] > maximum) {
87                  maximum = amount[addr];
88              }
89          }
90
91      }
92
```

Draw (93-110) - info function transmitting the leaders in purchased tickets in the current cycle. This method is used by the project's main contract during the prize mailout.

```
92
93      function draw(uint256 goldenWinners) public view returns(address[] addresses) {
94
95          addresses = new address[](goldenWinners);
96          uint256 winnersCount;
97
98          for (uint256 i = maximum; i >= 2; i--) {
99              for (uint256 j = 0; j < level[i].length; j++) {
100                 if (level[i][j] != address(0)) {
101                     addresses[winnersCount] = level[i][j];
102                     winnersCount++;
103                     if (winnersCount == goldenWinners) {
104                         return;
105                     }
106                 }
107             }
108         }
109
110     }
111
112 }
113
```

The main contract:

The constructor function of the contract (235-241) is called once-only at deployment contract to the network, the code sets variables to store addresses of Storage contracts, RefStorage, pseudococaine LotteryTicket and WinnerTicket, and updates the “gameCount”.

> *Note: in the Read Contract tab on EtherScan, you can always see how many ticket draws have already been played by looking at the gameCount counter.*

```
234
235     constructor(address RS_Addr) public {
236         x = new Storage();
237         LT = new LotteryTicket();
238         WT = new WinnerTicket();
239         RS = RefStorage(RS_Addr);
240         gameCount++;
241     }
242
```

Ticket purchase is carried out using the Fallback function, which is automatically called when sending to the ETH contract address (lines 243-279). For correct functioning in the code spelled out a lot of checks and conditions for the drawing, installation referrer, return delivery, ticket purchase.

```
242
243     function() public payable notFromContract {
244
245         if (players.length == 0 && paused) {
246             revert();
247         }
248
249         if (players.length == limit) {
250             drawing();
251
252             if (players.length == 0 && paused || msg.value < PRICE) {
253                 msg.sender.transfer(msg.value);
254                 return;
255             }
256
257         }
258
259         require(msg.value >= PRICE);
260
261         if (msg.value > PRICE) {
262             msg.sender.transfer(msg.value - PRICE);
263         }
264
265         if (msg.data.length != 0) {
266             RS.addReferrer(bytesToAddress(bytes(msg.data)));
267         }
268
269         players.push(msg.sender);
270         x.purchase(msg.sender);
271         RS.newTicket();
272         LT.emitEvent(msg.sender);
273         emit NewPlayer(msg.sender, gameCount);
274
275         if (players.length == limit) {
276             drawing();
277         }
278
279     }
280
```

The drawing's principle:

The drawing mechanism operates in two transactions: installation of a future reference block and the drawing itself.

All the key functionality is in the internal Drawing function (lines 281–342), which is automatically called when any ETH is sent (even 0), when all tickets in the draw are redeemed.

```
280
281     function drawing() internal {
282
```

In other words, to initiate the drawing of prizes after the redemption of the entire circulation, it is necessary to send any number of ETH to the address of the contract (even 0). If more than 0.01 ETH is sent, the first ticket in the next cycle will be automatically purchased.

During the last ticket's purchase in the cycle (100, 1000 or 10000), a certain future Ethereum reference block is set: +10 blocks for the first contract, +20 for the second and +40 for the third (line 286).
It is the chain of block hashes (10, 20 or 40) that independently defines the winners.

```
284
285         if (block.number >= futureblock + 240) {
286             futureblock = block.number + 10;
287             return;
288         }
289
```

The next transaction must pass in the period from the reference block to 250 blocks from the previous transaction. All transactions for the conduct of the distribution of prizes or to purchase tickets to the waiting period of the reference block will be reverted (line 283).

```
282
283         require(block.number > futureblock, "Awaiting for a future block");
284
```

Note: In the Ethereum network, it is possible to get data only about the last 256 blocks, for all the rest requested hash will be known in advance - 0. This limitation accounted for in the lottery code, and in excess of 250 blocks of the reference block to be installed again (lines 288-290).

Then the winners are defined and prizes are sent out in this order:

1) A simple principle of defining is used for Silver type winners: one block hash defines one winner (lines 292-297).
The logic doesn't require complications because prize for this position is from 0.02 to 0.5 ETH, which is a small amount, therefore, it is economically unprofitable to manipulate block hashes using large mining capacities.

```
291
292         for (uint256 i = 0; i < silver[0]; i++) {
293             address winner = players[uint((blockhash(futureblock - 1 - i))) % players.length];
294             winner.send(silver[1]);
295             WT.emitEvent(winner);
296             emit SilverWinner(winner, silver[1], gameCount);
297         }
298
```

2) Gold type winners are defined in advance: they are those users who have purchased the largest number of tickets in this cycle. (lines 299-313)
A query about the winners is done in "Storage" sub-contract (line 306).

Note: among two users who purchased the same number of tickets, the one whose the last ticket's purchase transaction was earlier in Ethereum network takes the highest place.

```
298
299         uint256 goldenWinners = gold[0];
300         uint256 goldenPrize = gold[1];
301         if (x.count() < gold[0]) {
302             goldenWinners = x.count();
303             goldenPrize = gold[0] * gold[1] / x.count();
304         }
305         if (goldenWinners != 0) {
306             address[] memory addresses = x.draw(goldenWinners);
307             for (uint256 k = 0; k < addresses.length; k++) {
308                 addresses[k].send(goldenPrize);
309                 RS.sendBonus(addresses[k]);
310                 WT.emitEvent(addresses[k]);
311                 emit GoldenWinner(addresses[k], goldenPrize, gameCount);
312             }
313         }
314
```

3) The complicated logic of defining is used for Brilliant type winners: draw's outcome is influenced by a chain of independent blocks (lines 315-327)
For a circulation of 100 tickets 7 blocks are used 7, for 1000 tickets - 10 blocks, for 10,000 tickets - 14 blocks (line 315).
The process of winner selection is as follows: the selection of lottery tickets is reduced by 2 times with each Ethereum block involved (lines 319-321).

```
314
315         uint256 laps = 7;
316         uint256 winnerIdx;
317         uint256 indexes = players.length * 1e18;
318         for (uint256 j = 0; j < laps; j++) {
319             uint256 change = (indexes) / (2 ** (j+1));
320             if (uint(blockhash(futureblock - j)) % 2 == 0) {
321                 winnerIdx += change;
322             }
323         }
324         winnerIdx = winnerIdx / 1e18;
325         players[winnerIdx].send(brilliant[1]);
326         WT.emitEvent(players[winnerIdx]);
327         emit BrilliantWinner(players[winnerIdx], brilliant[1], gameCount);
328
```

Example of the lottery's first type, stepwise winner defining using a chain of 7 blocks:
100 - 50 - 25 - 12.5 - 6.25 - 3.125 - 1.5625 - 1.

Each lottery ticket corresponds to a unique combination of hash blocks. The exact definition of the winning ticket is due to the use in calculations of magnitude 1e18 (lines 317, 324).

Each lottery ticket corresponds to a unique combination of hash blocks. The exact definition of the winning ticket is due to the use of the multiplier 1e18 (lines 317, 324).

Therefore, in order to win a certain ticket in the draw of 10,000 tickets, it is necessary that 14 consecutive blocks of hash correspond to the required to win this ticket, if at least one hash is changed, then the winner will be another ticket.

Next, the cycle is updated (lines 329-332).

```
328
329         players.length = 0;
330         futureblock = 0;
331         x = new Storage();
332         gameCount++;
333
```

Also, the cost of winner defining and sending prizes to a transaction sender (lines 290, 334-336) are replenished. In other words, all the costs of sending the transaction of buying the first ticket are paid from the balance of the smart contract, and the user also buys a ticket.

```
333
334         uint256 txCost = tx.gasprice * (gas - gasleft());
335         msg.sender.send(txCost);
336         emit txCostRefunded(msg.sender, txCost);
337
```

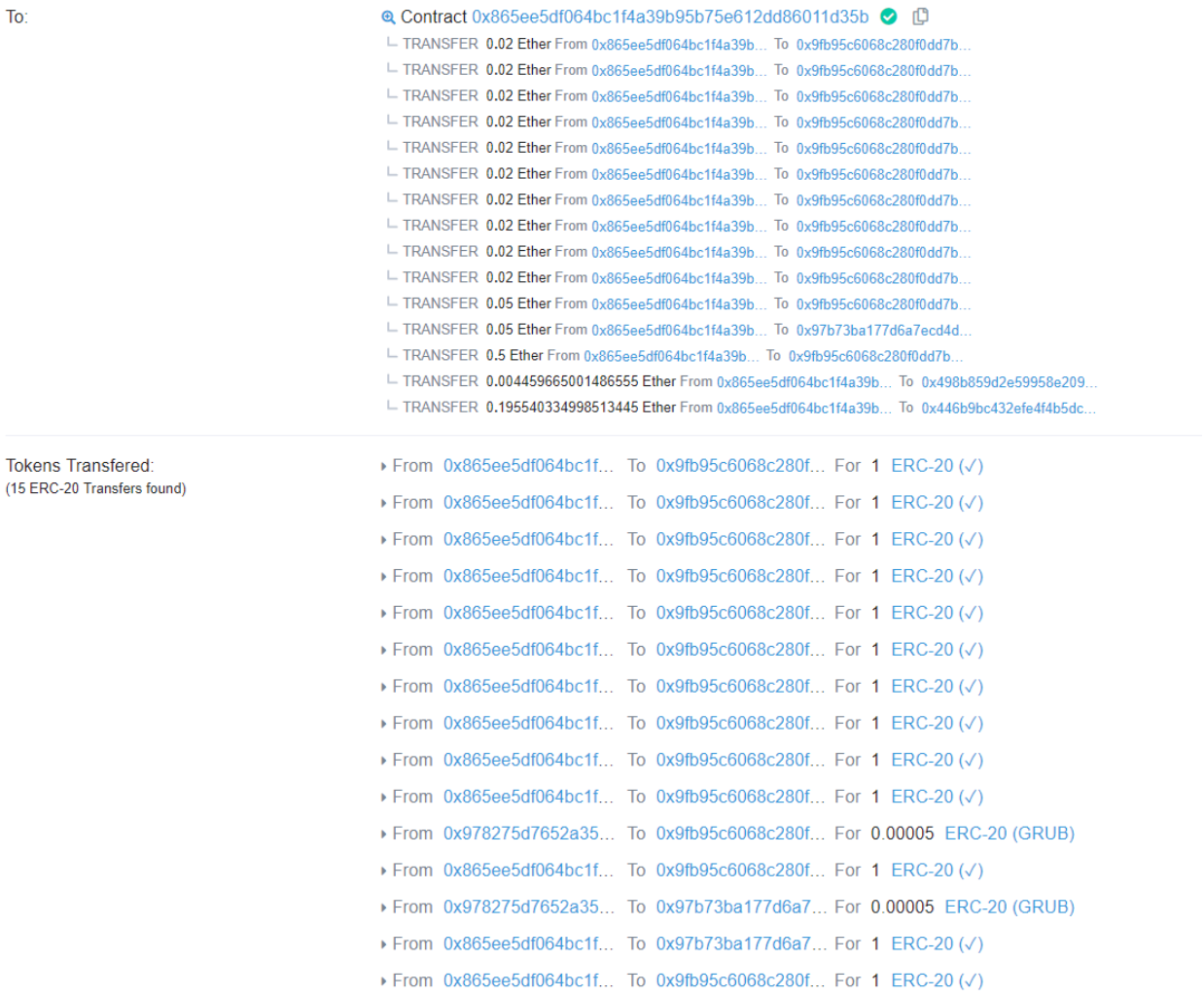
Remaining balance is sent to a project owner.

```
337
338         uint256 fee = address(this).balance - msg.value;
339         owner.send(fee);
340         emit FeePayed(owner, fee);
341
```

After successful performing of the "drawing" method, a standard purchase of the first ticket in the new cycle occurs.

An example of a transaction:

An example of a test transaction of drawing prizes in the 100 ticket draw can be seen here:
https://etherscan.io/tx/0x3be1c7d6a475125927e643ec61bdd7a7bec1948cb5f47295ec3772bbe43b3bbf



As can be seen from the transaction: the first was sent prizes such as Silver (10 times in 0.02 ETH), then the 2nd prize at the 0.05 ETH for the winners such as Gold, 0.5 for the Brilliant ETH, and was further restored transaction cost for the sender ~ 0.00446 ETH, and the remainder is sent to the developer's wallet ~ 0.1955 ETH.

> *Note: in this draw, almost all tickets were purchased from one address, and only the second gold prize was issued to another address.*

In addition:

A "pause" function is implemented in a contract (lines 214, 245-247, 251, 344-350). In case the lottery being paused, a contract is terminated only after the current draw's end, i.e. the pause prevents only the first ticket purchase and cannot interfere with the current cycle.

The code has a function of ERC20 tokens withdrawal from a contract (apparently to clear a contract from bounty and advertising) (lines 352-355).

```
351
352     function withdrawERC20(address ERC20Token, address recipient) external onlyOwner {
353         uint256 amount = IERC20(ERC20Token).balanceOf(address(this));
354         IERC20(ERC20Token).transfer(recipient, amount);
355     }
356
```

Info functions are available in "read on etherscan" tab:
AmountOfPlayers - the number of sold tickets in the current cycle.
ReferrerOf - user referrer (if any).
TicketsOf - the total number of ever purchased tickets by a user.

Also there is a standard Ownable contract in the code (36-56) for implementation of contract ownership right, a short interface of ERC20 standard (3-6).

The contracts implemented a system of events (220-225) to transmit information about events in the blockchain to the external environment.

```
219
220     event NewPlayer(address indexed addr, uint256 indexed gameCount);
221     event SilverWinner(address indexed addr, uint256 prize, uint256 indexed gameCount);
222     event GoldenWinner(address indexed addr, uint256 prize, uint256 indexed gameCount);
223     event BrilliantWinner(address indexed addr, uint256 prize, uint256 indexed gameCount);
224     event txCostRefunded(address indexed addr, uint256 amount);
225     event FeePayed(address indexed owner, uint256 amount);
226
```

In the code, the event is called using the emit token.

Implemented two pseudo tokens for calling special events (8-34) Lottery Ticket and Winner Ticket to lottery ticket buyers and winners respectively. The call

data events displayed on EtherScan as a reference token RC20 with the appropriate name. These pseudo tokens have no real value and do not affect the functionality of contracts.