ethereum

How can I securely generate a random number in my smart contract?

Asked 7 years, 6 months ago Modified 1 month ago Viewed 97k times

231

If I am writing a smart contract for some kind of (gambling) game, how can I generate a random number securely? What are the best practises and security trade-offs to be aware of?

contract-design security gambling

Share Improve this question Follow edited Mar 15, 2016 at 7:59 asked Jan 21, 2016 at 6:56

J-B
8,771 16 45 76

5 This is a very interesting question that may have a lot of applications other than a gambling game. The best application I can think of is a truly random number based cryptocurrency mining scheme - simply give the reward to a random miner/user. A network with this scheme would attract new miners while it would need very little electricity to run in comparison to Bitcoin's proof of work which is nothing else but energy waste. – Kozuch May 6, 2016 at 19:58

@Kozuch: although reliable random numbers would be a big step, there would still be a number of problems with that approach, e.g. Sybil attacks, etc. – D.F.F Mar 22, 2021 at 8:25

20 Answers

Sorted by: Highest score (default)

182

There are a few trade-offs and key points to keep in mind in this area.

1. **Any** decision that a user makes which affects the outcome gives that user an unfair advantage. Examples include:
   - Using a blockhash, timestamp, or other miner-defined value. Keep in mind that the miner has a choice of whether to publish a block or not, so they could conceivably have one chance at the prize *per block they mine.*
   - Any user-submitted random number. Even if the user pre-commits to a number, they have a choice in whether or not to reveal the number.
2. Everything that the contract sees, the public sees.

- This means that the number should not be generated until after entry into the lottery has been closed.

3. The EVM will not outrace a physical computer.
    - Any number that the contract generates may be known before the end of that block. Leave time between the generation of the number and its use.

Now for the technique:

Perfectly Decentralized Lottery-Style Non-Malleable Commitment:

1. The Casino sets aside a reward for a random number
2. Each user generates their own 256-bit secret random number `N`.[0]
3. Users create their commitment by hashing their `N` with their address: `bytes32 hash = sha3(N,msg.sender)` [1]
    - note: steps 2 & 3 should be performed locally, in secret
4. Users send their hash to the contract, along with ether greater than or equal in value to the value of the random number.[2]
5. Submissions continue for some number of blocks, or until sufficient participants have joined.

Once the submission round has ended, the reveal round begins.

6. Each user submits their random number `N` to the contract
7. The contract verifies that the `sha3(N,msg.sender)` matches the original submission

If the user does not submit a valid `N` in time, his deposit is forfeit.

8. The `N`s are all `XOR`'d together to generate the resulting random number.
9. This number is used to determine which of the participants receives the reward. (`N % numUsers`)[43]

Notes and alternatives:

[0.] It is critical to the security of this scheme that `N` be chosen randomly from a high-entropy distribution - otherwise an adversary could "guess-and-check" values of `N` to determine the committed value. A previous version of this answer failed to explicitly mention this consideration.

Alternatively, users may select a uniformly random 256-bit value `K` and compute the commitment as `sha3(msg.sender, K, N)`. To reveal, both `K` and `N` are revealed but only `N` is used in further steps.

[1.] The users must concatenate their address to their `N` before hashing. Otherwise, another user could simply submit an identical hash, then wait for `N` to be revealed, then submit that. `N XOR N` equals 0, so this could be used to cancel out all submissions except the attacker's.

2. This is where the trade-offs come in. The last person to reveal their `N` has a choice whether to reveal or to not reveal. This essentially gives them a double chance at winning. Enter enough times, and you get a new choice for each entry. *Hint: Miners chose the order of transactions in a block*. In order to discourage this, users must put up a large security deposit, equal to the value they would gain by manipulating the random number. This could be a problem for many users, especially for large jackpots, even with game-theoretic optimizations.

- A commonly used alternative is a semi-centralized system. This requires the "house" to submit the first hash and last reveal. If the house does not fulfill their duty, everyone's ether is returned. This has issues, such as the house choosing to flake if a jackpot payout is imminent. The idea is that the house's reputation is at stake.
- Note that this essentially centralizes the whole system. One simply needs to take down the house for the whole operation to be shut down. This risk can be reduced by hiring multiple trusted non-players to be the first/last commiters.
- Another trick is to use hired professional third parties to "mine" randomness for you, so that the players need not be bothered with the process.

3 As long as the random numbers `N` are 256 bits, this modular reduction will not introduce noticeable modulo bias. If a uniform sample from a set of more than `2^128` members is needed, you must use alternative techniques

4. A reward is necessary in order to foster competition among participants. It causes a classic tragedy-of-the-commons/prisoner's dilemma situation. Collusion between participants would allow them to win a large pot and split it among themselves, but if everyone knows what everyone else will submit, they know what they themselves should submit to win the reward. Thus, if the reward is larger than the value of the random number divided by the number of players, then everyone is incentivised to keep their own number a secret in order to have a better shot at the reward. Note that only one of the participants needs to submit a good random number, and the result will be unpredictable.

Examples: Chainlink VRF, RANDAO, The thread where I first thought through this

Share Improve this answer Follow edited Jun 24 at 15:42 answered Jan 21, 2016 at 8:12

Tjaden Hess
36.4k 10 90 118

20 Thinking out loud here: what about using one (or several) oracle to get the SHA-256 of a particular Bitcoin block number (not found yet) as a seed. Would this work for Ethereum games where the payout is inferior to the reward of mining a BTC block? Because the cost of finding a BTC block is very high, deciding "not to publish" the block and forfeiting 12.5 BTC reward plus tx fees would not make sense it the payout of the Ethereum game is less than, say, 12.5 BTC. – Cedric Martin Feb 3, 2016 at 16:57

7 This is actually a very interesting idea. You would need to wait for a few confirmations, but it would definitely increase the cost to manipulate the results. I think I'll make a contract to do this... – Tjaden Hess

Feb 3, 2016 at 17:25

4 @TjadenHess Can we just construct the random number sequentially from future ETH block hashes bit-by-bit? The 1st bit of the random number is the rightmost bit of the future block #1 hash, the 2nd bit is the rightmost bit of the future block #2 hash etc until some future block #L. If we get more 1s than 0s in the end, then it is "Heads", otherwise it is "Tails". No single block can significantly influence the result (not even the last one). By increasing length L, you can make it progressively more expensive for any miner to cheat continuously. – akhmed May 21, 2016 at 4:16

2 @akhmed That's an interesting idea, but as L grows, any single miner's influence will *grow*, not decrease, since on average all of the honest miners are putting out 1s and 0s with equal probability, so every block the malicious miner makes shifts the total 1s vs 0s more unequal – Tjaden Hess Jan 25, 2017 at 18:09

2 The exact function used to calculate the final result doesn't matter much, you could pairwise hash as well. The point is that the Ns are committed beforehand so there's no way of knowing what the contribution will be. It's sort of similar to the resoning behind a one-time pad; each bit has a 50/50 chance of being 1, so there's no way to pick your N to manipulate it. – Tjaden Hess Feb 21, 2018 at 15:39

31

Note that with linagee's suggestion, you're not just trusting random.org, you're also trusting Oraclize. Oraclize publish a TLS notary proof to show that the data they're giving you really came from random.org, but that's not enough for this case: We need to know that this was the *only* data they got from random.org. Otherwise they could keep trying and throwing away random numbers from random.org until they got one which won their bet, and you would have no way of detecting this.

Share Improve this answer Follow

answered Feb 29, 2016 at 0:42
Edmund Edgar
16.7k 1 28 58

7 Not to mention, the proof isn't verified by the contract, it's verified off-chain. Thus, they could run of with the money whenever they want – Tjaden Hess Feb 29, 2016 at 3:16

Right. These dice games are actually the one case I can think of where these proofs might actually be useful, because they typically turn over very fast, so if you're able to detect fraud in an automated way and blow the whistle you may be able to keep the damage quite small. Without the proofs you'd still find out about the fraud (the people who got screwed would cry blue murder and people would cross-check the original data sources) but it might be a bit slower. But it's only mitigation not prevention, and like I say you need a deterministic data source not a random number. – Edmund Edgar Feb 29, 2016 at 7:14

17

You could use https://api.random.org/json-rpc/1/ which gives you a random source of data through JSON and Oraclize which allows you to make use of the feed inside an Ethereum contract and optionally have it strongly authenticated as having come from random.org. (Along with existing methods of using the hash of the block, timestamp, and such.)

You would be "trusting" random.org to feed you random data. You might mitigate the risk by using multiple sources of randomness.

Share Improve this answer Follow edited Jan 21, 2016 at 8:56 answered Jan 21, 2016 at 8:38

linagee
6,078 25 32

6 Additional warnings about this approach here ethereum.stackexchange.com/a/1655/42 (in case the answers get separated) – eth ♦ Feb 29, 2016 at 12:55

11

To do it in practice, use a Chainlink VRF.

Conceptually, use this answer.

To make the random number, we need a few variables:

`keyhash` : The public key hash of the oracle(s). This is to make SURE they are making a random number based on the seed we give it). We are going to use the ropsten Chainlink VRF oracle.

`userProvidedSeed` : A seed of our choice, this is another piece we are going to use to prove they are making us a random number. Every time we call rollDice, we should use a different seed. See choosing a seed for more information. This is steps #2 "Each user generates their own secret random number N" and #3 "Users create their commitment by hashing their N with their address: bytes32 hash = sha3(N,msg.sender)1" in this answer.

`_vrfcoordinator` and `_link` : These are the contract address of the vrfcoordinator and Chainlink token. They are hard coded for Ropsten in this contract.

A full example is given below.

To deploy the contract, use the _vrfCoordinator and _link addresses supplied in the comments. Then use your hashed number as the `userProvidedSeed` .

```
pragma solidity 0.6.2;

import
"https://raw.githubusercontent.com/smartcontractkit/chainlink/develop/contracts/src/v0.6/VRFCo

contract Verifiable6SidedDiceRoll is VRFConsumerBase {
    using SafeMath for uint;

    bytes32 internal keyHash;
    uint256 internal fee;

    event RequestRandomness(
        bytes32 indexed requestId,
        bytes32 keyHash,
        uint256 seed
    );

    event RequestRandomnessFulfilled(
        bytes32 indexed requestId,
        uint256 randomness
    );

    /**
     * @notice Constructor inherits VRFConsumerBase
     * @dev Ropsten deployment params:
     * @dev   _vrfCoordinator: 0xf720CF1B963e0e7bE9F58fd471EFa67e7bF00cfb
     * @dev   _link:           0x20fE562d797A42Dcb3399062AE9546cd06f63280
     */
    constructor(address _vrfCoordinator, address _link)
        VRFConsumerBase(_vrfCoordinator, _link) public
    {
        vrfCoordinator = _vrfCoordinator;
        LINK = LinkTokenInterface(_link);
        keyHash = 0xced103054e349b8dfb51352f0f8fa9b5d20dde3d06f9f43cb2b85bc64b238205; //
hard-coded for Ropsten
        fee = 10 ** 18; // 1 LINK hard-coded for Ropsten
```

Note: At the current moment, the Chainlink VRF doesn't have the decentralized `XOR` ing feature, this is still in development. See the Chainlink blog for more details.

Answer from Chainlink Labs developer advocate

Share Improve this answer Follow   edited Dec 18, 2022 at 20:26   answered May 22, 2020 at 6:41
Patrick Collins
9,913 4 41 87

10

Since different contracts are **securing different amounts of value**, at the **opposite end of the spectrum to RANDAO is the simple BLOCKHASH**.

If BLOCKHASH may suit your purpose, it's highly recommended to review the question (below is only a snippet):

When can BLOCKHASH be safely used for a random number? When would it be unsafe?

> As a general rule, BLOCKHASH can only be safely used for a random number if the total amount of value resting on the quality of that randomness is lower than what a miner earns by mining a single block.

Share Improve this answer Follow   edited Apr 13, 2017 at 13:01   answered Feb 22, 2016 at 19:39
Community Bot
1
eth ♦
84k 53 280 404

5 I think it is a bad idea to have *any* value resting on a blockhash. Keep in mind that if the **total** value, between all contracts on the network, resting on that blockhash is more than 5 Ether, there is an incentive for miners to cheat. – Tjaden Hess Feb 23, 2016 at 3:57

1 Upvoted. To clarify for players of small lotteries that do use blockhash, you can get "attacked" even though a lottery may have a limit of say 1 Finney. If there are over 5000 other players gambling 1 Finney, that exceeds the 5 Ether mining reward and there is incentive for a miner to cheat. – eth ♦ Feb 23, 2016 at 4:24

And note that this applies to timestamp too, and any combination of the two. Although you should never ever use timestamp. – Tjaden Hess Feb 23, 2016 at 4:48

2 Relevant question about timestamp and how it's more vulnerable to attack than blockhash: ethereum.stackexchange.com/q/413/42 – eth ♦ Feb 29, 2016 at 9:46

1 Under proof of stake, does this means the BLOCKHASH is almost never safe? (unless fees are very high, as a staker only loses a fraction of compound interest by not pushing a block) – fair glu May 12, 2016 at 9:47

7

The short answer is you can't. RANDAO works, but it's slow, and if your game is popular, there will be a strong incentive for people to game applying the last number.

I suggest using an oracle to provide randomness. (as mentioned in Tjaden Hess' Notes and Alternatives 2c.) The two major benefits are you can strongly assert that your number is independent of any other wager, which is vital to pricing the risk of using that number. Secondly, there is effectively no limit on entropy throughput. If there is a marketplace of oracles, then you can leverage the irrevocable history of the blockchain to model the likelihood that a given oracle is colluding with players or not. Of course, the players, the house, and the oracle all provide their own entropy. Other features such as whitelisting, blacklisting, bonds asserting non-collusion, etc etc can be added as desired.

Share Improve this answer Follow

answered Feb 5, 2016 at 17:58

A. Frederick Dudley
442 2 6

7

It seems to me the various suggestions presented here all have in common a model in which a smart contract is acting as the server in what is essentially a regular ol' single-server/multi-client architecture, and much of the concern is that a miner will game the system - which is sorta the problem that has always existed when there is a single trusted server.

Why not just turn the thing on its side and simply not involve the contract at all in the random number generation process?

A lottery-like game might work this way:

- Each gambler submits a transaction to the lottery contract containing the fee, a desired "picked" number and an encrypted random seed value. If the same gambler wants to pick another number only the fee and pick are required - the seed is per-gambler, not per bet.
- After betting closes, the gambler sends another transaction - this time containing nothing but the key to decrypt his seed.

- After everyone has provided decryption keys (or a sufficient amount of time elapses) each gambler fetches all of the keys and encrypted seeds and computes the "real" random number - probably just a modulo-sum of the decrytped seeds. If it is one of the gambler's picks, a transaction is sent to the contract that causes the contract to verify that the gambler provided a seed, verify that the winning number supplied in the transaction is in fact correct and was picked by the gambler, and then award the winnings.

Lots of hand-waving and left-out details here, but I believe the basic idea is sound.

Share Improve this answer Follow

edited Jul 26, 2017 at 23:50
r_alex_hall
103 4

answered May 12, 2016 at 3:56
jimkberry
1,296 11 11

1 Security objections to this, anyone? This seems sound to me, and it seems to eliminate the trust/oracle problems pointed out in other answers. The revealed secret keys absolutely verify the encrypted seed values were prior created, and what they were. @jimkberry, what do you mean by "If the gambler wants to pick another number..?" Another number when? In another round of gambling (when all other users also pick a number; when the smart contract is employed to create a new payout)? After they've already picked a number (overriding their previous pick)? – r_alex_hall Jul 27, 2017 at 0:03

2 @r_alex_hall : Wow, I don't even remember answering this question, it's been so long... Since I was hand-waving in terms of a lottery I'm pretty sure that by "picking another number" I meant the equivalent of "buying another ticket" in a traditional lottery. – jimkberry Aug 7, 2017 at 21:09

1 @jimkberry I think the critic to this solution was the last-revealer had the chance to not reveal. so he would get double chance. (he can calculate if he wins the lottery by not revealing) and mind that you can't really know if the not-revealing is the winning person, because it's easy to have multiple identities. so only solution is a deposit for each person that's more costly than having a "2nd chance", which is then not sooo good for user experience? – EralpB Nov 9, 2017 at 8:28

1 @jimkberry imagine 1,2,3,4 and 5 are playing the game 3 and 5 are the same "person", 1 2 3 4 reveals their decrypt keys and then 5 can choose not to reveal if 3 is already winning. – EralpB Nov 9, 2017 at 8:29

7

Why not just use the official lottery figures from the television? For instance, the German Lotto Zahlen. Then you build in a voting mechanism where the players can vote on the winning numbers after the lottery numbers have been drawn and published on television. To incentivize also losers to vote, you could offer players to reimburse say 5% of their stake if they decide to vote and their vote turns out to find consensus among all players

Share Improve this answer Follow

answered Feb 16, 2019 at 19:29
Walodja1987
141 1 6

1 And if losers outnumber winners, why losers will not vote against official truth? – rlib Oct 20, 2021 at 10:52

You need an Oracle for this solution. Though its safe, it Costs. – Cyberience Jan 5, 2022 at 8:29

5

I've published **eth-random**, which is a simplified guide about how we implemented RNG on CryptoKitties.

There are some caveats about it, but to this point I haven't seen a single founded hack attempt on it.

Update 1: For a practical example of the implementation see CK contract source code - functions `_breedWith` (act as commit) and `giveBirth` + `geneScience.mixGenes` (act as reveal).

Share Improve this answer Follow edited Oct 14, 2019 at 3:35 answered Oct 9, 2019 at 19:17

Fabiano Soriani
355 1 4 13

Nice! Spare on the details though... there doesn't seem to be a bare-bones description of what the RNG actually is. – Chan-Ho Suh Oct 13, 2019 at 1:04

Hey @Chan-HoSuh it's indeed more of a technique description than one particular implementation, because currently it's impossible to create just one Math.random() equivalent on Ethereum.
– Fabiano Soriani Oct 14, 2019 at 3:39

Fabiano, thanks for posting the update! – Chan-Ho Suh Oct 14, 2019 at 4:18

3

Lets think that you are using a block hash to decide a lottery winner and miner A participates to that lottery and buys one ticket. Then he mines many valid blocks and throws them away, if they are not suited to his ticket.

In practice, every valid block that a miner throws away costs him 5 ethers. If your lottery's odds are so, that miner A has to eg create 1 000 000 valid blocks till he finds a proper block for him to win, he wil throw away 5 000 000 ethers.

If your lottery's main win is 1 000 000 ethers it makes no sense for a miner to throw away 5 000 000 eth to win 1 000 000.

Share Improve this answer Follow answered Mar 15, 2016 at 8:22

Matias
1,099 1 10 16

2 The issue with this analysis is that you're assuming that the miner is only buying one ticket. What if he's buying half the tickets, or in the worst case all but one ticket? Then the odds are much higher.
– Tjaden Hess Apr 11, 2016 at 4:13

I have an idea that builds on Tjaden's protocol that I outline here:

Is this approach to an ethereum lottery sound and/or novel?

3

It solves the problem of having to put up a large security deposit in lotteries. Would appreciate some feedback.

Share Improve this answer Follow

edited Apr 13, 2017 at 13:01
Community Bot
1

answered Mar 28, 2017 at 7:03
EthJ
173 7

3

How about this flow:

1) Each user generates a string "I am a part of the lottery and my number is 8272143" (Secretly) where 8272143 is the self chosen number.

2) Each user encrypts its own string with a self chosen password (secret)

3) Each user publishes the encrypted string, so now everyone can see all the encrypted strings

4) When it is decided to not allow any more participants, all users publish their self chosen passwords and all strings are decrypted. Users who do not publish their passwords are excluded from winning.

5) Everyone can confirm, that each player speaks the truth about their password, as the first part of the string must be "I am a part of the lottery and my number is " so there is no way to fake another number at this point.

6) All the strings are now concatenated, and a hash is generated. **This hash is the final random number!** In case of a lottery, you could pick the number closest to the hash, but this is just a case specific detail.

This way, there will be no waiting for blocks to complete. Only brute forcing decryption of the strings will allow for an advantage. This will be avoided by using strong enough encryption.

(I am a total rookie in this field, so bare with me if this is just nonsense.)

UPDATE:

I realise, that if 2 clients work together, the last client can choose to wait publishing his password and to be excluded in case his friend will win this way. A solution can be, that all users have an amount of ETH on the line which they will have to pay of they do not publish their password. All in all, this solution is not very sexy, I admit.

Share Improve this answer Follow

edited Jun 1, 2017 at 9:13

answered Jun 1, 2017 at 7:32
Stephan Møller
131 3

1 It seems to me that this and the answer by @jimkberry are duplicate? jimkberry's was first ;) – r_alex_hall Jul 26, 2017 at 23:56

3

I don't know the best secure method, but please see bellow a **list of Vulnerable implementations** (so finally **do not use**):

- PRNGs using block variables as a source of entropy
- PRNGs based on a blockhash of some past block
- PRNGs based on a blockhash of a past block combined with a seed deemed private
- PRNGs prone to front-running

You may take a look on Predicting Random Numbers in Ethereum Smart Contracts, which explain all security issues relative to different kinds of pseudo-random number generator (PRNG) implemented in (a lot of existing) Smart Contract.

Share Improve this answer Follow

answered Oct 30, 2018 at 11:23

A. STEFANI
151 4

3

Just a update to this question, but this time it use a technique call verifiable delay function (vdf). It was built based upon time lock puzzle invented by Ronald L Rivest et al. The core idea behind is that if you can't calculate the result in a set amount of time, then the result is pseudo random. So with that in mind, vdf is a function which take in an input and it take maybe 1 hour to compute (You can change the time in the setup phase of this function), then it output the result and a evidence that this result is indeed calculate using that input in no time.

So the idea is just like Perfectly Decentralized Lottery-Style Non-Malleable Commitment that was proposed in this question answer but we just need to add vdf into it in a step before

> 8. The Ns are all XOR'd together to generate the resulting random number.

So our step is:

9. With that result we let it run through our vdf function (The time to calculate it must be longer than the submissions time to ensure that the last submit cannot calculate the final result).
10. Then You public the vdf function result to everybody, so that everybody can verify that the result is indeed calculate using the input

The reward is just the same as the answer propose.

But i just simplify to the core idea behind it, the beauty of this is in the math that implement it.

Verifiable delay function paper: https://eprint.iacr.org/2018/601.pdf

You can also check this paper which talk about 2 vdf:
https://crypto.stanford.edu/~dabo/pubs/papers/VDFsurvey.pdf

Share Improve this answer Follow edited Aug 28, 2019 at 16:39 answered Aug 28, 2019 at 16:32

haxerl
1,104 6 14

3

Securely generating a random number is difficult because of the deterministic nature of the EVM. There are many approaches. Which one works best depends on your security requirements and context. Here are a few:

Previous block hash

If your random number is not security critical, you can simply use the previous block hash. E.g. a random number from 0 to 99: `uint256(block.blockhash(block.number-1)) % 100`

Trusted oracle

Use an oracle you trust to fetch a number and perform a callback to your smart contract.

Commit and reveal

1. A generates a number r
2. A writes keccak256(r) to the smart contract
3. B commits to whatever event needs to be randomized
4. A reveals r

A must not be able to predict which event B will commit to.

A has proof that B could not have known the number when B committed to the event

B has proof that the number was generated before he committed to the random event

Share Improve this answer Follow answered Dec 16, 2019 at 19:36

Jesbus
10.1k 6 35 61

3

There are randomness oracles like Chainlink VRF available nowadays.

Share Improve this answer Follow

answered Jun 17, 2020 at 9:00
Mikko Ohtamaa
20.9k 6 58 122

2

Another way to generate random numbers could be to distribute various sources of randomness into the platform to seed the winning ticket. For instance we could use the cents of the closing prices of every stock traded in the main stock exchanges in the world. It would be nearly impossible to make all the stock exchanges work together to tamper the lotto.

The closing prices are shown by various services including Yahoo, Google, Bloomberg, etc. so they can work as a kind-of a public ledger to verify the prices are correct.

Also since so many direct traders, algorithms, company traders, etc. are involved changing those prices, it would be nearly impossible to predict what the very last traded price will be or wait for the very last second to be the last person to trade a stock.

Share Improve this answer Follow

answered Oct 17, 2017 at 18:34
awavi
121 1

2

Try the following algorithm to prevent cheating.

The idea is to use hash of a block but the block number is not known for miners/participants and organizers to prevent cheating.

1. On start organizer defines a random uint64 - **key number** to be used in the winner random number calculation. The uint number is big enough and known to organizer only on start.
2. Organizer fixes the **key number** by providing hash of sum - the number plus current block hash (as a salt). The key hash is stored in the Lottery contract and available for all the participants. Then key hash is used to prevent organizers cheating and fixes the key number.
3. Participants buy tickets and the Lottery contract stores all the participant addresses. The addresses are used on winner calculation as well.
4. When conditions are fulfilled (e.g. all the tickets are sold or necessary amount of blocks are generated or just after some time) organizer starts winner calculations.

5. Organizer provides the stored **key number** and confirms the number is exactly the same by calculating the key hash (entered on init).
6. The **key number** is added to the sum of addresses and we got mod 255 of the sum to detect number of block which hash is used to detect winner. Key number + address1 + address2 ... + addressN % 255 = number of winner block (the block hash is used to get winner number)

To person who want to guess winning number it is not possible to get the number - in the contract we have just hash of the number. The number is rather big uint64 (or could be even bigger) to prevent "unhashing" the number. For organizers the number is fixed and is futile because all the tickets owners addresses must be known to calculate the winner.

Opinions? Any holes in the logic?

I created a separate question but it could be answered here as well.

Share Improve this answer Follow

answered Jan 5, 2018 at 9:05
StanislavL
287 2 14

2

You need to use a third party to generate randomness.

Contract accounts only perform an operation when instructed to do so by an Externally Owned Account. So **it is not possible for a Contract account to be performing native operations like random number generation** or API calls – it can do these things only if prompted by an Externally Owned Account. This is because Ethereum requires nodes to be able to agree on the outcome of computation, which requires a guarantee of strictly deterministic execution.

read more: http://ethdocs.org/en/latest/introduction/what-is-ethereum.html#how-does-ethereum-work

Share Improve this answer Follow

answered Jul 21, 2018 at 22:06
Gabriel Vasile
131 2

1

Marco from Oraclize here. You could use the Oraclize Random Data Source, which leverages a Ledger Trusted Execution Environment.

You can read an introduction here and some examples here.

The Random Data Source is significantly safer than using Random.org with TLSNotary and using the blockhash to determine a random number, and it less difficulty and expensive than others commit-reveal schemes.

Share Improve this answer Follow

answered Oct 11, 2017 at 10:06
user34

Links are broken. – Chan-Ho Suh Oct 13, 2019 at 1:07