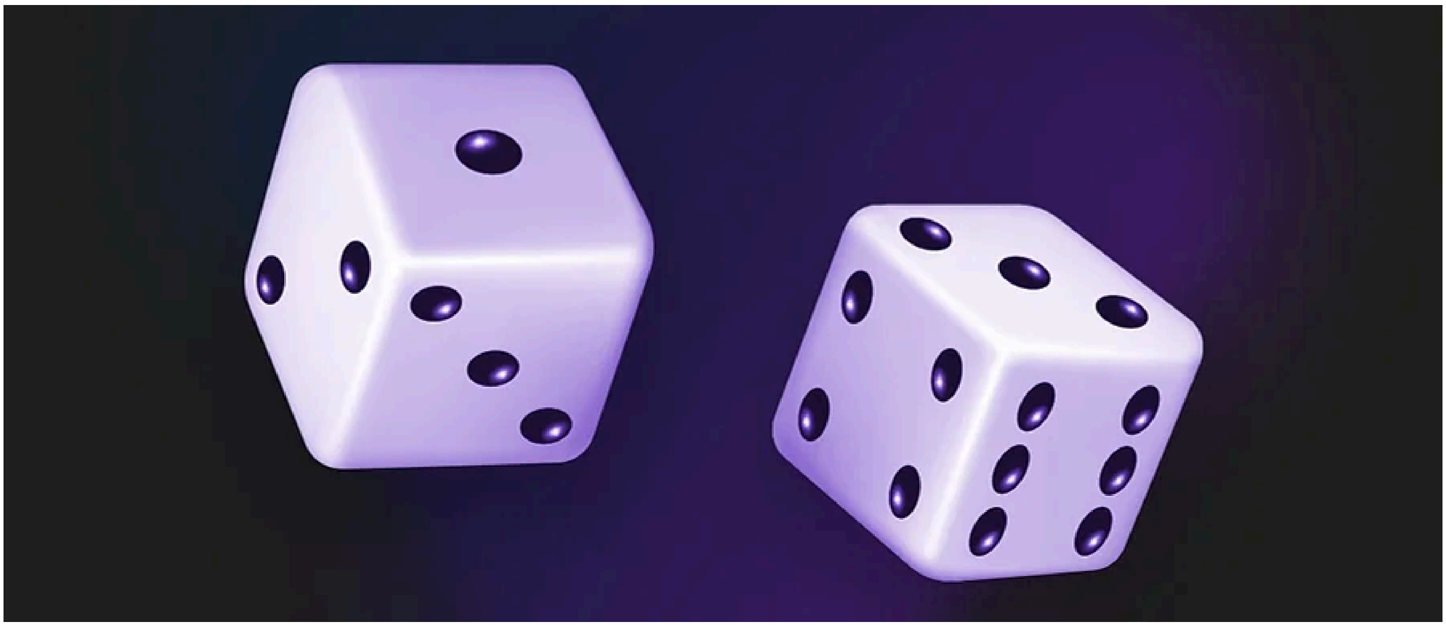
Dec 2, 2022 4 min read

Generate a random number with Solidity on the blockchain

Updated: Dec 21, 2022


Random Number

Randomness is tricky on the blockchain because the blockchain is deterministic, but randomness requires non-determinism (otherwise it becomes predictable). This article assumes the user has some familiarity with solidity already, especially the operations block.number(), block.hash(), and digital signatures.

If you use data like block.timestamp or the previous blockhash, then someone can use a smart contract to predict if a transaction will result in the desired outcome or not. Capture the ether has a sequence of hacks to test you on this.

If you need random numbers, there are some design patterns you can consider.

Commit Reveal

Although blockchain transactions are perfectly deterministic at the time of execution, they cannot predict the future. Specifically, the future blockhash cannot be predicted (with a caveat described below).

It works like this: a transaction commits that 20 blocks in the future from the current block n, whatever the blockhash is at 20 + n, that blockhash will be the random number. Since you cannot predict the blockhash of block n + 2, where n is the current block, then block n + 20 is considered random.

You are allowed a lookback of 256 blocks to get the hash function, so the user must initiate a second transaction sometime between block n + 20 and n + 276. The second transaction is the reveal. Of course, the user can see if the random number came in their favor or not, so the application must be configured such that the user is incentivized to send the second transaction if it is favorable to them.

For example, a fair coin flip would pay out the user if, say, the blockhash was an even number. The user would not do the second transaction if they saw the blockhash was odd, but they don't win anything anyway. To win, the blockhash must be even, and only then will they be bothered to send the second transaction.

This scheme can be tampered with by block producers. Although block producers cannot force an exact hash value, they can re-order transactions until the block hash produces an even number (or something else that is favorable to them).

You can defend against this behavior by having the user commit the hash of a secret number at block n. Upon block n + 20, the user reveals the pre-image of the hash and the preimage is concatenated with the blockhash. The concatenation of those two values is hashed, and that hash is used as the random number.

The block producer cannot know the preimage of the hash, so they cannot tamper with it. But it still is not block-producer proof.

If the block producer is playing the lottery, they can commit a known secret number, then tamper with the blockhash such that the final outcome is favorable.

Now that Ethereum has moved to proof of stake, this attack is harder to pull off, because the malicious producer must be the block producer at the exact block of the reveal.

But if you want to be safe against malicious block producers, you should use chainlink VRF (described next).

A financially equipped attacker can exploit this scheme if sufficiently motivated. Let's say we pay out a player if the blockhash is even. The attacker can flood the network with high gas transactions to prevent the reveal transaction between blocks 20 and 276. Remember, the blockhash produces zero if the look back is greater than 256. This would be very expensive for the attacker, but it is still a possible attack vector.

Chainlink VRF

Much has already been written online about how to generate random numbers with Chainlink VRF (Verifiable Random Function). Their documentation is very good and easy to follow. But here is how it works in a nutshell.

The smart contract that wants a random number calls the chainlink smart contract requesting a random number (and paying some LINK to cover the cost).

Chainlink will accept the request and wait a specified number of blocks and call back the contract that requested the random number. Chainlink's algorithm for generating the random number is transparent, so anyone can validate it was created in a fair manner.

Chainlink cannot do this in on transaction, or a malicious player could revert the transaction if they get an outcome they don't like.

Due to the possibility of chain reorganization, the application specify that the callback happen further into the future for high value use cases.

Chainlink initiates the second transaction, so this saves the user the trouble of authorizing a second transaction. However, there is a literal price to this convenience, as not only must the user pay the gas cost, they (or the application) must also pay LINK token to use the service.

Offchain signature

I have to give credit to a conversation I had with gaspack.xyz who came up with the core of this idea.

On obvious UX problem with the above solutions is that they require a delay of some sort, and potentially two transactions. In a blockchain game, players might not appreciate the delay.

How can you do this without creating a vulnerable smart contract?

A semi-decentralized way to get random numbers is to have an offchain random number generator create ***and cryptographically sign*** a random number, the sender, and a future blocknumber.

That random number will be concatenated with the future blockhash, and the resulting string will be hashed. This produces the random number. The smart contract responsible for distributing the reward verifies the signature against the sender and block number.

Even if the offchain random number generator isn't perfectly random, or even slightly malicious, it can't predict future blockhashes, and doesn't know what it's random number will be concatenated with. Because the signature is only valid at a particular block, the player cannot wait for a favorable block before they roll the dice.

There are three ways this scheme can be weak:

1. The miner controls both the random number generator and is the block producer at the specified time. As long as the block producer and the random number generator don't collude, this scheme is safe.
2. The random number produces the same number over and over. In this case, the miner can easily predict how he or she should tamper with the hash.

3. There needs to be a way for players to avoid getting several random numbers so they can try until results are favorable. This necessarily requires some kind of off-chain gating which will likely be less transparent and secure.

It is possible to mitigate these drawbacks to a certain degree by further decentralizing the random number producer. For example, you could use block hashes on the bitcoin network as a source of random numbers. This will make it transparent if funny business is going on.