Arseny Reutov Follow
Application Security Researcher at Positive.com
Feb 1 · 10 min read

Predicting Random Numbers in Ethereum Smart Contracts

Ethereum has gained tremendous popularity as a platform for initial coin offerings (ICOs). However, it is used in more than just ERC20 tokens. Roulettes, lotteries, and card games can all be implemented using the Ethereum blockchain. Like any blockchain implementation, Ethereum is incorruptible, decentralized, and transparent. Ethereum allows running Turing-complete programs, which are usually written in Solidity, making it a “world supercomputer” in the words of the platform’s founders. All these features are especially beneficial in the context of computer gambling, in which user trust is crucial.

The Ethereum blockchain is deterministic and as such it imposes certain difficulties for those who have chosen to write their own pseudo-random number generator (PRNG), which is an inherent part of any gambling application. We decided to research smart contracts in order to assess the security of PRNGs written in Solidity and to highlight common design antipatterns that lead to vulnerabilities allowing prediction of the future state.

Our research was performed in the following steps:

1. 3,649 smart contracts were collected from etherscan.io and GitHub.
2. These contracts were then imported into the Elasticsearch open-source search engine.
3. Using the Kibana web UI for rich search and filtering, 72 unique PRNG implementations were found.
4. Based on manual assessment of each contract , 43 contracts were identified as vulnerable.

Vulnerable implementations

Analysis identified four categories of vulnerable PRNGs:

- 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

Let's examine each category and examples of vulnerable code.

PRNGs based on block variables

There are a number of block variables that may be wrongly used as a source of entropy:

- `block.coinbase` represents the address of the miner who mined the current block.
- `block.difficulty` is a relative measure of how difficult it was to find the block.
- `block.gaslimit` restricts maximum gas consumption for transactions within the block.
- `block.number` is the height of current block.
- `block.timestamp` is when the block was mined.

First of all, all these block variables can be manipulated by miners, so they cannot be used as a source of entropy because of the miners' incentive. More importantly, the block variables are obviously shared within the same block. So if an attacker's contract calls the victim contract via an internal message, the same PRNG in both contracts will yield the same outcome.

Example 1 (0x80ddae5251047d6ceb29765f38fed1c0013004b7):

```
// Won if block number is even
// (note: this is a terrible source of randomness, please
don't use this with real money)

bool won = (block.number % 2) == 0;
```

Example 2 (0xa11e4ed59dc94e69612f3111942626ed513cb172):

```
// Compute some *almost random* value for selecting winner
from current transaction.

var random = uint(sha3(block.timestamp)) % 2;
```

Example 3 (0xcC88937F325d1C6B97da0AFDbb4cA542EFA70870):

```
address seed1 = contestants[uint(block.coinbase) %
totalTickets].addr;
address seed2 = contestants[uint(msg.sender) %
totalTickets].addr;
uint seed3 = block.difficulty;
bytes32 randHash = keccak256(seed1, seed2, seed3);
uint winningNumber = uint(randHash) % totalTickets;
address winningAddress = contestants[winningNumber].addr;
```

PRNGs based on blockhash

Each block in the Ethereum blockchain has a verification hash. The Ethereum Virtual Machine (EVM) enables obtaining such blockhashes via the block.blockhash() function. This function expects a numeric argument that specifies the number of the block. In the course of research, we discovered that the result of block.blockhash() is frequently misused in PRNG implementations.

There are three major flawed variations of such PRNGs:

- `block.blockhash(block.number)` , which is the blockhash of the current block.
- `block.blockhash(block.number - 1)` , which is the blockhash of the last block.
- `block.blockhash()` of a block that is at least 256 blocks older than the current one.

Let’s examine each of these cases.

block.blockhash(block.number)

The `block.number` state variable allows obtaining the height of the current block. When a miner picks up a transaction that executes contract code, the `block.number` of the future block with this transaction is known, so the contract can reliably access its value. However, at the moment of transaction execution in the EVM, the blockhash of the block that is being created is not yet known for obvious reasons and the EVM will always yield zero.

Some contracts misinterpret the meaning of the expression `block.blockhash(block.number)` . In these contracts, the blockhash of the current block was deemed known at runtime and was used as a source of entropy.

Example 1 (0xa65d59708838581520511d98fb8b5d1f76a96cad):

```
function deal(address player, uint8 cardNumber) internal
returns (uint8) {
  uint b = block.number;
  uint timestamp = block.timestamp;
  return uint8(uint256(keccak256(block.blockhash(b), player,
cardNumber, timestamp)) % 52);
}
```

Example 2 (https://github.com/axiomzen/eth-random/issues/3):

```
function random(uint64 upper) public returns (uint64
randomNumber) {
  _seed = uint64(sha3(sha3(block.blockhash(block.number),
_seed), now));
```

```
    return _seed % upper;
  }
```

block.blockhash(block.number-1)

A certain number of contracts use another variation of blockhash-based PRNGs, relying on the blockhash of the last block. Needless to say, this approach is also flawed: an attacker can make an exploit contract with the same PRNG code in order to call the target contract via an internal message. The “random” numbers for the two contracts will be the same.

Example 1 (0xF767fCA8e65d03fE16D4e38810f5E5376c3372A8):

```
//Generate random number between 0 & max
```

```
uint256 constant private FACTOR =
1157920892373161954235709850086879078532699846656405640394575840079131296399;
function rand(uint max) constant private returns (uint256
result){
  uint256 factor = FACTOR * 100 / max;
  uint256 lastBlockNumber = block.number - 1;
  uint256 hashVal =
uint256(block.blockhash(lastBlockNumber));
  return uint256((uint256(hashVal) / factor)) % max;
}
```

Blockhash of a future block

A better approach is to use the blockhash of some future block. The implementation scenario is as follows:

- The player makes a bet and the house stores the block.number of the transaction.
- In a second call to the contract, the player requests that the house announces the winning number.
- The house retrieves the saved block.number from storage and gets its blockhash, which is then used to generate a pseudo-random number.

This approach works only if an important requirement is met. The Solidity documentation warns about the limit of saved blockhashes that the EVM is able to store:

Note

The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Therefore, if a second call was not made within 256 blocks and there is no validation of the blockhash, the pseudo-random number will be known beforehand—the blockhash will be zero.

The most well-known case of this weakness being exploited is the hack of the SmartBillions lottery. The contract had insufficient validation of the `block.number` age, which resulted in 400 ETH being lost to an unknown player who waited for 256 blocks before revealing the predictable winning number.

Blockhash with a private seed

In order to increase entropy, some of the analyzed contracts employed an additional seed deemed private. One such case is the Slotthereum lottery. The relevant code is as follows:

```
bytes32 _a = block.blockhash(block.number - pointer);
for (uint i = 31; i >= 1; i--) {
  if ((uint8(_a[i]) >= 48) && (uint8(_a[i]) <= 57)) {
    return uint8(_a[i]) - 48;
  }
}
```

The variable pointer was declared as private, meaning that other contracts cannot access its value. After each game, the winning number between 1 and 9 was assigned to this variable, which was then used as an offset of the current `block.number` when retrieving the blockhash.

Being transparent in nature, the blockchain must not be used to store secrets in plaintext. Although private variables are protected from other contracts, it is possible to get the contents of contract storage off-chain. For instance, popular Ethereum client web3 has the API method `web3.eth.getStorageAt()`, which allows retrieving storage entries at the specified indices.

Given this fact, it is trivial to extract the value of the private variable pointer from the contract storage and supply it as an argument to an exploit:

```
function attack(address a, uint8 n) payable {
  Slotthereum target = Slotthereum(a);
  pointer = n;
  uint8 win = getNumber(getBlockHash(pointer));
  target.placeBet.value(msg.value)(win, win);
}
```

Front-running

In order to receive the maximum reward, miners choose transactions to create a new block based on the cumulative gas used by each transaction. The order of transaction execution in a block is determined by the gas price. The transaction with the highest gas price will be executed first. So by manipulating the gas price, it is possible to get a desired transaction executed ahead of all others in the current block. This may constitute a security issue—commonly referred to as front-running—when a contract's execution flow depends on its position in a block.

Consider the following example. A lottery uses an external oracle to get pseudo-random numbers, which are used to determine the winner from among the players who submitted their bets in each round. These numbers are sent unencrypted. An attacker may observe the pool of pending transactions and wait for the number from the oracle. As soon as the oracle's transaction appears in the transaction pool, an attacker sends a bet with a higher gas price. The attacker's transaction was made last in the round, but thanks to the higher gas price, is actually executed before the oracle's transaction, making the attacker victorious. Such a task was featured in the ZeroNights ICO Hacking Contest.

Another example of a contract prone to front-running is the game called "Last is me!". Every time a player buys a ticket, that player claims the last seat and the timer starts counting down. If nobody buys the ticket within a certain number of blocks, the last player to "take a seat" wins the jackpot. When the round is about to finish, an attacker may observe the transaction pool for other contestants' transactions and claim the jackpot by means of a higher gas price.

Towards a safe(r) PRNG

There are several approaches for implementing safer PRNGs on the Ethereum blockchain:

- External oracles
- Signidice
- Commit–reveal approach

External oracles: Oraclize

Oraclize is a service for distributed applications that provides a bridge between the blockchain and the external environment (Internet). With Oraclize, smart contracts can request data from web APIs such as currency exchange rates, weather forecasts, and stock prices. One of the most prominent use cases is the ability of Oraclize to serve as a PRNG. Some of the analyzed contracts used Oraclize to obtain random numbers from random.org via the URL connector. This scheme is depicted in Figure 1.

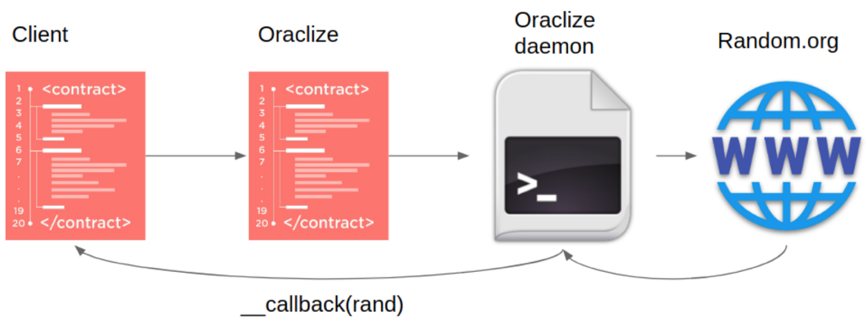

Figure 1. Oraclize scheme of operation

The key drawback of this approach is that it is centralized. Can we trust the Oraclize daemon not to tamper with the results? Can we trust random.org and all its underlying infrastructure? Although Oraclize provides TLSNotary verification of the results, it can be used only off-chain—in case of a lottery, only after a winner has been chosen. A better use of Oraclize is as a “random” data source using Ledger proofs that can be verified on-chain.

External oracles: BTCRelay

BTCRelay is a bridge between Ethereum and Bitcoin blockchains. Using BTCRelay, smart contracts in the Ethereum blockchain can request future Bitcoin blockhashes and use them as a source of entropy. One project that uses BTCRelay as a PRNG is The Ethereum Lottery.

The BTCRelay approach is not safe against the miner incentive problem. Although this approach sets a higher barrier compared to relying on Ethereum blocks, it simply takes advantage of the fact that the price of Bitcoin is higher than Ethereum, thus reducing but not eliminating the risk of cheating by miners.

Signidice

Signidice is an algorithm based on cryptographic signatures that can be used as a PRNG in smart contracts involving only two parties: the player and the house. The algorithm works as follows:

- The player makes a bet by calling a smart contract.
- The house sees the bet, signs it with its private key, and sends the signature to the smart contract.
- The smart contract verifies the signature using the known public key.
- This signature is then used to generate a random number.

Ethereum has a built-in function `ecrecover()` for verifying ECDSA signatures on-chain. However, ECDSA cannot be used in Signidice since the house is able to manipulate input parameters (specifically, parameter *k*) and thus affect the resulting signature. A proof-of-concept of such cheating has been created by Alexey Pertsev.

Fortunately, with release of the Metropolis hardfork, a modular exponentiation operator has been introduced. This allows implementing RSA signature verification, which unlike ECDSA does not allow manipulating input parameters to find a suitable signature.

Commit–reveal approach

As the name implies, the commit–reveal approach consists of two phases:

- A "commit" stage, when the parties submit their cryptographically protected secrets to the smart contract.
- A "reveal" stage, when the parties announce cleartext seeds, the smart contract verifies that they are correct, and the seeds are used to generate a random number.

A proper commit–reveal implementation should not rely on any single party. Although players do not know the original seed submitted by the owner, and their chances are equal, the owner may also be a player, due to which players cannot trust the owner.

A better implementation of the commit–reveal approach is Randao. This PRNG collects hashed seeds from multiple parties, and each party is paid a reward for participation. Nobody knows the others' seeds so the result is truly random. However, a single party refusing to reveal the seed will result in denial of service.

Commit–reveal can be combined with future blockhashes. In this case, there are three sources of entropy:

1. owner's sha3(seed1)
2. player's sha3(seed2)
3. a future blockhash

The random number is then generated as follows: `sha3(seed1, seed2, blockhash)`. Thus, the commit–reveal approach solves the miner incentive problem: a miner can decide on the blockhash but does not know the owner's and player's seeds. It also solves the owner incentive problem: an owner knows only the owner's own seed, but the player's seed and future blockhash are unknown. In addition, this approach solves the case when a person is both owner and miner: that person decides on the blockhash and knows the owner's seed, but does not know the player's seed.

Conclusion

Secure PRNG implementation in the Ethereum blockchain remains a challenge. As our research suggests, developers tend to use their own implementations due to the lack of ready-made solutions. But when creating these implementations, it is easy to make a mistake because the blockchain has limited sources of entropy. When designing a PRNG, developers should be sure to first understand each party's incentive and only then choose an appropriate approach.