D6: Bad Randomness

Portland State
Computer Science

#6: Bad Randomness

- also known as **nothing is secret**
  - Contract code is public
  - Execution of contracts deterministic
- Random number generation can not be done secretly
  - PNRG algorithm whose code and seed is public
  - Results in ability to predict any random numbers generated

Walkthrough scenario

- A **victim contract** uses the block number as a source of randomness for a game.
- An attacker creates a **malicious contract** that checks if the current block number is a winner
  - If so, it calls the **victim contract** in order to win
  - Since the call will be part of the same transaction, the block number will remain the same on both contracts.
- The attacker only has to call his/her **malicious contract** until it wins.

Example #1

Hrishi Olickel Follow
Jun 21, 2016 · 7 min read

Why Smart Contracts Fail: Undiscovered bugs and what we can do about them

ones programmers are used to. For example, the contract **TheRun** uses the current timestamp in order to generate random numbers and award a jackpot based on the result. Similarly, a betting contract may run for a predetermined

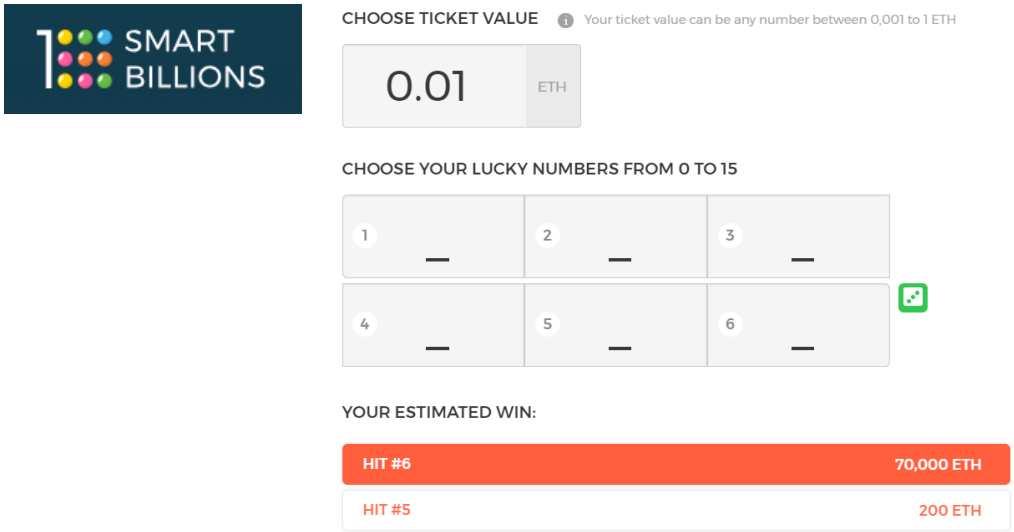
Unfortunately, this timestamp can be manipulated by a colluding miner. He may adjust the timestamp provided by a few seconds, changing the output of the contract to his benefit. In **TheRun**, the output of the random number

Common errors

- Using known or predictable block variables as a source of entropy
  - `block.coinbase` (address of miner who mined the block)
  - `block.difficulty` (puzzle difficulty solved)
  - `block.gaslimit` (maximum gas for a transaction)
  - `block.number` (current level/height in chain)
  - `block.timestamp` (wall-clock time of mined block given in seconds since the epoch)

- `block.blockhash(block.number)`
  - Always 0
  - Current `block.number` is known, but its hash (as the current block), is not known while executing contract function (i.e. known only after block is actually mined)!
  - https://github.com/axiomzen/eth-random/issues/3
- `block.blockhash(block.number - 1)`
  - Blockhash of the last block (or any prior block)
  - Known to an adversary who can craft an attacking contract to leverage it using the exact same code that the victim contract uses
- Calling `block.blockhash()` (conveniently abstraction provided by Solidity…)
  - Hides underlying fact that EVM returns 0 on blocks more than 256 blocks old!
  - Smart Billions bug…next slide

Smart Billions bug



1.3k

Posted by u/supr3m 1 year ago

SmartBillions lottery contract just got hacked!

Someone made it in the "hackathon" (lol). The hacker could withdraw 400 ETH before the owners, who wrote "the successful hacker keeps ALL of the 1500 ETH reward", withdrew quickly the remaining 1100 ETH, that happened 5min before the next transaction (from the "hacker") would have emptied the whole contract. So that's already a lie from

How did it happen? Their lottery functions were flawed, if you place a bet (systemPlay() function) with betting on number value "0" and then call the won() function after 256+ blocks (after you placed the bet) the returning value will be "0" so you would have bet on "000000" and result would be "000000" and baaam you have the jackpot. The lucky guys first bet was "1" so "000001" and result after 256+ blocks calling won() would be "000000" so he matched 5 correctly which is 20000x and with 0.01ETH bet amount a win of 200ETH. He managed to pull that 2 time and corrected to "0" and for that

Code vulnerability example #1

- A private **`seed`** is used in combination with an **`iteration`** number and the **`keccak256`** hash function to determine if the caller wins.
- Even though the **`seed`** is **`private`**, it must have been set via a transaction at some point in time and thus is visible on the blockchain.
- Attacker knows exactly which iterations will win

```
uint256 private seed;

function play() public payable {
  require(msg.value >= 1 ether);
  iteration++;
  uint randomNumber = uint(keccak256(seed + iteration));
  if (randomNumber % 3918507 == 0) {
    msg.sender.transfer(this.balance);
  }
}
```

Code vulnerability example #2

- `block.blockhash` used to generate random number using current `block.number`
- Value is 0
- First call wins

```
function play() public payable {
  require(msg.value >= 1 ether);
  if (block.blockhash(block.number) % 3918507 == 0) {
    msg.sender.transfer(this.balance);
  }
}
```

Code vulnerability example #3

- `block.coinbase`, `block.difficulty`, and `msg.sender` used to generate random number
  - Data sources are public (`block.difficulty`)
  - Data sources can be manipulated directly by miners giving them an advantage (`block.coinbase`, `msg.sender`)
  - Miner can calculate number and insert a winning transaction

```
1 function generateRandom() public returns (uint) {
2   address seed1 = contestants[uint(block.coinbase) % totalTickets].addr;
3   address seed2 = contestants[uint(msg.sender) % totalTickets].addr;
4   uint seed3 = block.difficulty;
5   bytes32 randHash = keccak256(seed1, seed2, seed3);
6   uint winningNumber = uint(randHash) % totalTickets;
7   return winningNumber;
8 }
```

Remediation

- Commit reveal (Bit-commitment)
  - 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.
  - But, can refuse to reveal if you know you've lost!
    - Contracts must be written to penalize such behavior
  - https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620

SI CTF Lab 3.1 (D6_LockBox)
SI CTF Lab 3.2 (D6_HeadsOrTails, D6_Lottery)