On-chain randomness in games

Abstract

On-chain randomness is vital for activities like games and lotteries, but it makes random numbers susceptible to hacking. This article discusses two key security concerns and preventive strategies in random number generation: (1) using oracles to integrate off-chain data streams for obtaining true random numbers, minimizing manipulation risks, and (2) employing on-chain multi-node random number generation.

By using oracles to introduce off-chain data sources for genuine randomness, manipulation risks are reduced, provided that third-party data is trustworthy and verifiable. On the other hand, verifiable on-chain multi-node random number generation invites multiple nodes to generate provable random responses. These are combined to form the final random number, which can't be manipulated or influenced, even if a node is compromised. Through reward and penalty mechanisms, more nodes are encouraged to participate, enhancing system security.

Introduction

The concept of On-chain randomness plays a significant role in various applications, such as gaming, lottery, and NFTs. In particular, its application in blockchain games is related to the generation of random rewards and gaming environments. For instance, in-game rewards can be distributed through random NFT minting, and randomness is also used to generate game maps and items. This enhances the challenge and depth of the game, preventing players from cheating by memorizing steps. Moreover, randomness can also impact players' action results, like combat outcomes in strategy games, or skill results in role-playing games.

The value of on-chain randomness in blockchain stems mainly from its unbiased and unpredictable nature, which has led to its widespread use in various blockchain games. With the growth of blockchain games, they have become a more potent economic force affecting both developers and players. Randomness plays an irreplaceable role in blockchain games, as the "reward randomness" it creates allows gamers to buy and sell in-game rewards through secondary markets, giving rise to a game economy (GameFi) where playing games can also earn money.

However, implementing randomness on the blockchain is not easy. Due to the decentralized and publicly verifiable nature of the blockchain, solutions that use on-chain data (e.g., block hashes, timestamps) as sources of randomness run a high risk of being cracked. On the other hand, if developers use non-public data for generating random numbers, they could potentially manipulate the generation of random numbers, violating the verifiable and transparent nature of the blockchain.

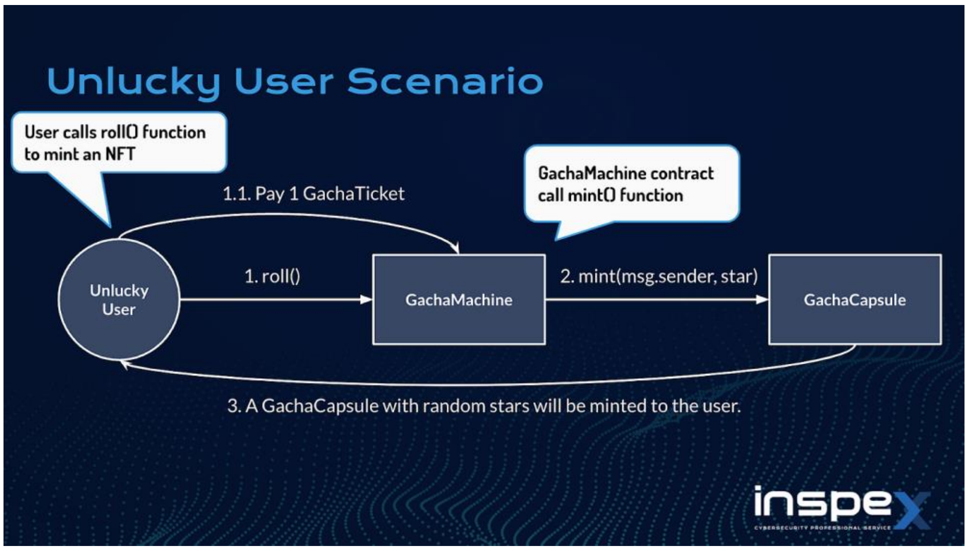
In this article, we will collect many on-chain randomness techniques and practical cases, interpret the technical background of various random number generation methods in detail, and analyze the attacks and losses suffered by various blockchain games due to the cracking of on-chain randomness. We will also observe the subsequent corrections of these projects, hoping to organize a relatively good solution for future blockchain games and applications related to on-chain randomness.

Related Work

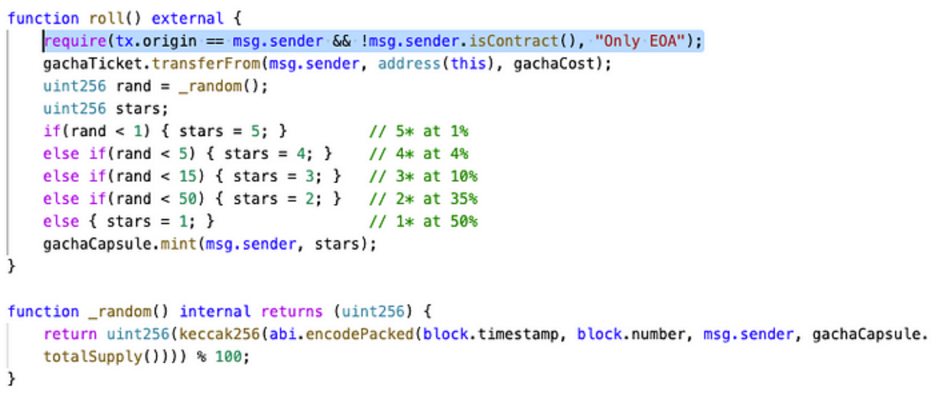
In the blockchain domain, the generation of random numbers is crucial for tasks like blockchain games and lotteries. However, this also makes random numbers a target for hackers. This article will introduce two common security issues related to random number generation, and how to effectively prevent them. First, we will analyze how to crack the vulnerability of randomly minting NFT rewards, and explain its mechanism with actual code examples. Second, we will explore how to exploit transaction reverts to break random numbers, illustrating the underlying hacker attacks with specific cases. By understanding these attack methods, we can better grasp the security issues of random number generation and take corresponding protective measures.

1. Generating Random Numbers Using On-Chain Information - Cracking Randomly Minted NFT Rewards

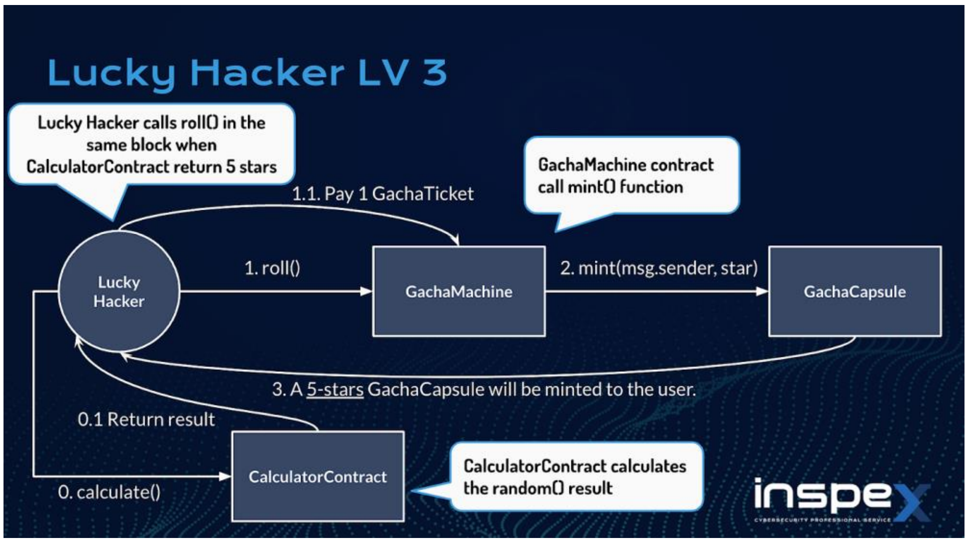
As one of the most common applications in blockchain games - randomly minting NFT rewards, we found a code example vulnerable to attack from the inspex cybersecurity audit company (https://github.com/InspexCo/gacha-lab), which helps us understand the poor randomness problem of the NFT minting mechanism. Under normal usage, users will call the roll() function and pay GachaTicket tokens as the minting cost. The GachaMachine contract will then randomly calculate the number of stars (rarity) and mint a GachaCapsule NFT for the user.


Unlucky User Scenario
User calls roll() function to mint an NFT
1.1. Pay 1 GachaTicket
GachaMachine contract call mint() function
Unlucky User
1. roll()
GachaMachine
2. mint(msg.sender, star)
GachaCapsule
3. A GachaCapsule with random stars will be minted to the user.
inspex
CYBERSECURITY PROFESSIONAL SERVICE

However, when the sources of randomness all use on-chain information or are cracked by hackers, as in this example where only msg.sender, block.timestamp, block.number, and fixed information in the game (gachaCapsule) are used as the sources for the hash value, hackers can implement the same random logic in their own deployed CalculatorContract. They can predict the random result of the roll() function with the contract they deployed, performing extra checks on the rarity (star). When the random result calculated by CalculatorContract is the rarest five stars, they can execute the roll() function in the same block immediately afterwards, allowing them to obtain the rarest NFT.


```
function roll() external {
    require(tx.origin == msg.sender && !msg.sender.isContract(), "Only EOA");
    gachaTicket.transferFrom(msg.sender, address(this), gachaCost);
    uint256 rand = _random();
    uint256 stars;
    if(rand < 1) { stars = 5; }          // 5* at 1%
    else if(rand < 5) { stars = 4; }     // 4* at 4%
    else if(rand < 15) { stars = 3; }    // 3* at 10%
    else if(rand < 50) { stars = 2; }    // 2* at 35%
    else { stars = 1; }                  // 1* at 50%
    gachaCapsule.mint(msg.sender, stars);
}

function _random() internal returns (uint256) {
    return uint256(keccak256(abi.encodePacked(block.timestamp, block.number, msg.sender, gachaCapsule.
    totalSupply()))) % 100;
}
```


Lucky Hacker LV 3
Lucky Hacker calls roll() in the same block when CalculatorContract return 5 stars
1.1. Pay 1 GachaTicket
GachaMachine contract call mint() function
Lucky Hacker
1. roll()
GachaMachine
2. mint(msg.sender, star)
GachaCapsule
3. A 5-stars GachaCapsule will be minted to the user.
0.1 Return result
0. calculate()
CalculatorContract
CalculatorContract calculates the random() result
inspex

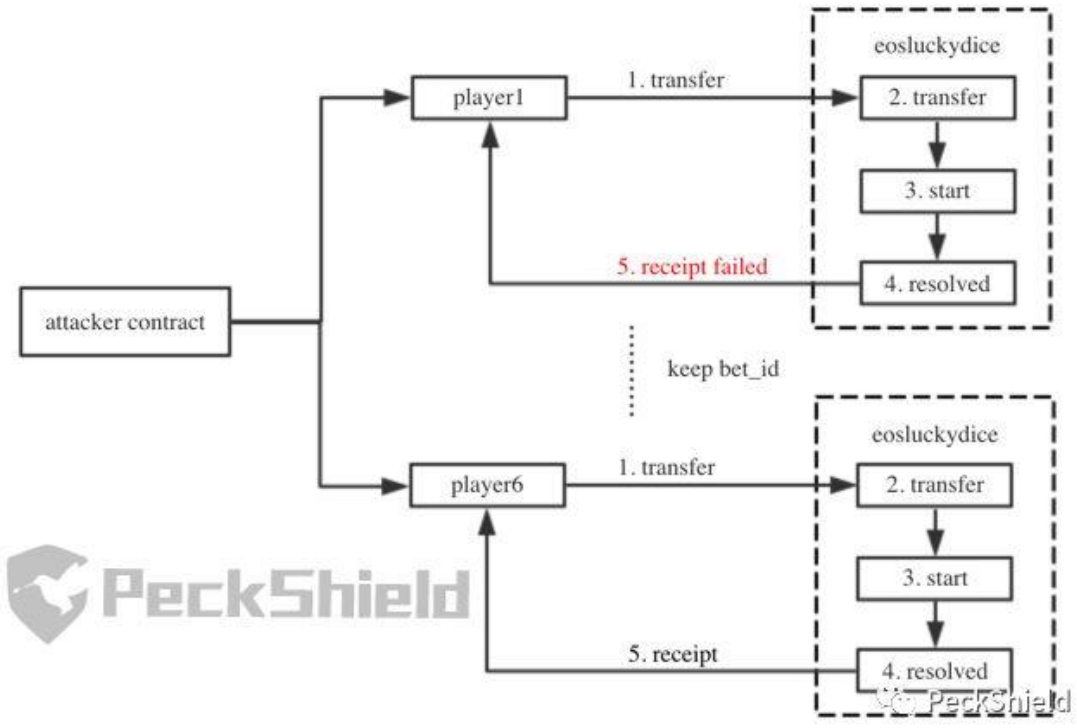
2. Exploiting Transaction Revert to Break Random Numbers

The ecosystem of the EOS chain developed by Block.one once thrived, and the blockchain games on the EOS chain successfully attracted more people to participate in on-chain interactions, including EOSBet, EOSCast, FFgame, EOSDice, EOSWin, etc. However, the relentless hacker attacks posed a serious threat to the EOS ecosystem at that time. Blockchain security company PeckShield pointed out in a report that the principle behind most hacker attacks is related to random number vulnerabilities.

Taking the dice game on EOS as an example, referring to [2] to reconstruct from the hacker's perspective, the hacker's attack method was analyzed.

All elements that influence the generation of random numbers in this game are related to on-chain information, including: transaction hash ID, transaction block height, block ID prefix, bet_id (lottery number allocated by the game), etc. Although the game uses information from two delayed transactions as part of the random number, that is, it tries to ensure unpredictability by using information from future blocks and adds the lottery number randomly allocated by the game, however, when a hacker attacks, as long as it is in the same block, these "future" block information are constant and can be revealed. At the same time, in this example, the hacker used the revert error generated when the contract call function fails, allowing the bet_id with a high chance of winning to be retained.

Therefore, as shown in the flowchart below, the hacker deployed several attack contracts at the same time. Players 1 to 5 only bet the minimum amount, while player 6 bets the maximum amount. In the contracts of players 1 to 5, the current bet_id and the winning probability of the hash combined with other block information will be calculated respectively. If the winning probability is not high, it will be drawn normally to change the bet_id. If the calculated winning probability is high, the revert generated by the function call failure will be used. The bet_id with the highest chance of winning is retained to player 6 and bets the maximum amount, ultimately successfully cracking the blockchain dice game.



In order to combat this type of attack, PeckShield suggests that developers should remove variables that attackers can control in the generation of random numbers in DApp and add uncontrollable factors provided by external contracts to participate in the generation of random numbers. Also, avoid the draw action and notification action in the same transaction, thereby avoiding the revert of the transaction status, and thereby blocking attacks from hackers.

Method

Next, we will attempt to propose two existing solutions to the problem of random number generation in the blockchain environment: (1) using oracle to introduce off-chain data flow and (2) verifiable on-chain multi-node random number generation. Using an oracle, such as the API provided by Provable Things, you can obtain real random numbers from external data sources and ensure the security of data through encryption technology. In the case where the data provided by a third party is trustworthy and verifiable, it effectively reduces the risk of manipulation. On the other hand, verifiable on-chain multi-node random number generation, such as Chainlink Verifiable Random Function (VRF), invites multiple nodes to generate verifiable random responses, and then combines the responses of each node to form the final random number. Even if the node is compromised, it cannot manipulate or influence the generation result of the random number, and encourages more nodes to participate through the reward and punishment mechanism, improving the security of the overall system.

1. Use Oracle - Introducing Trustworthy Third-Party Off-Chain Data Flow

Using an Oracle API provided by Provable Things, you can provide random data unrelated to the on-chain status by pushing data through the off-chain data stream

Because the blockchain is a deterministic environment, all nodes must reach the same result to maintain the consistency of the network. However, the generation of real random numbers requires the introduction of a certain degree of uncertainty, which is relatively difficult to achieve in a blockchain environment. In this case, we usually need to rely on external data sources, also known as oracles, to obtain random numbers. Using oracles has many advantages:

1. Provide true randomness: Oracles (like Provable) can obtain real random numbers from external data sources outside the blockchain. These data sources may be random number generators based on physical phenomena, or other trusted third-party services.

2. Security: Oracles use encryption technology to ensure the security of data. This means that even if the data is intercepted during transmission, its randomness and credibility will not be affected.

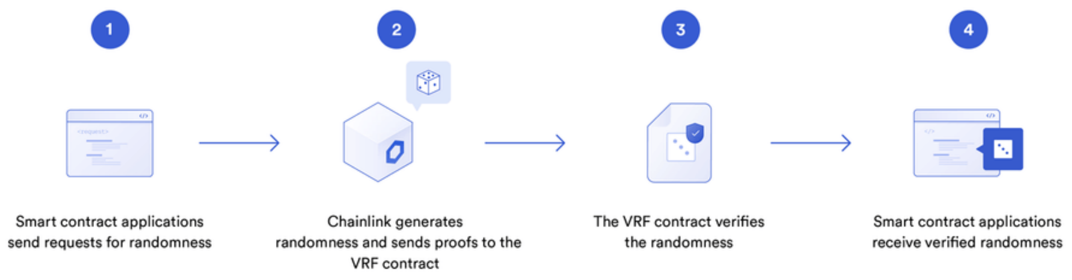
3. Reduce the risk of manipulation: Because the random numbers provided by the oracle come from external data sources outside the blockchain, they will not be manipulated by other participants in the blockchain network. Compared with random number generation that only relies on internal data in the blockchain (such as block hash or timestamp), this can significantly reduce the risk of manipulation.

Code example:
https://github.com/EPJ-coding/bdaf-final/blob/main/Orcale_sample_code.sol

2. Verifiable On-chain Multi-node Random Number Generation - Chainlink Verifiable Random Function

Chainlink Verifiable Random Function (VRF) is implemented based on this verifiable random function paper[4]. VRF will send invitations to multiple nodes when generating random numbers. Each node will generate a provable random response after receiving the request (using a public key, private key, and the hash value of the seed to generate a random number). The responses of each node will then be XORed to form the final random number. The working principle of the VRF is to combine the block data that is still unknown at the time of request with the private key of each node to generate random numbers and cryptographic proof. Each node uses its private key when generating randomness and verifies it on the chain using blockchain wallet signatures and proof verification functions before sending it to the user contract. Even if the node is compromised, it cannot manipulate or control the result of random number generation, because the cryptographic proof on the chain will fail, which will be immediately and permanently visible on the blockchain. Users no longer rely on nodes that stop responding and/or do not provide randomness with valid proof.

At the same time, Chainlink uses a token reward and punishment mechanism to encourage more node providers to join, provide more difficult to break randomness, and punish low-quality and/or dishonest node operators, and remove them from future randomness sources. , Provide higher security for smart contract developers and their users.


1
Smart contract applications send requests for randomness
2
Chainlink generates randomness and sends proofs to the VRF contract
3
The VRF contract verifies the randomness
4
Smart contract applications receive verified randomness

Code example:
https://github.com/EPJ-coding/bdaf-final/blob/main/VRF_sample_code.sol

Conclusion

In confronting the challenges posed by random number generation within the blockchain ecosystem, we have suggested two viable and currently employed strategies. The first strategy involves the integration of an oracle, which leverages trusted external data sources for the generation of random numbers. This method offers robust data security and significantly diminishes the potential for manipulation. The second strategy involves the use of verifiable, on-chain, multi-node random number generation. This system assures that even if a single node becomes compromised, the overall generation of random numbers remains unaffected. The security of the system is further reinforced by a reward and punishment mechanism.

It is undeniable that randomness plays a critical role in a myriad of blockchain applications, yet this also introduces security risks associated with random number generation. With the application of strategies such as oracle and multi-node random number generation, we are able to sustain randomness while simultaneously safeguarding the system's security and reliability.

Looking towards the future, continuous development and optimization of these strategies are essential as the blockchain landscape and its associated security threats continue to evolve. Furthermore, exploring new technologies and approaches to enhance the security and integrity of random number generation on the blockchain is crucial. This includes research into advanced cryptographic techniques, the development of more sophisticated consensus algorithms, and the incorporation of quantum computing capabilities for random number generation.

Reference

[1] https://inspexco.medium.com/how-hackers-can-become-lucky-in-nft-minting-822f48d4b917
[2] https://mp.weixin.qq.com/s/WTiwaIbB7N2VDKKgcXo0Fg
[3] https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/#post-title
[4] Včelák, Jan et al. “Making NSEC5 Practical for DNSSEC.” (2022)
[5] https://github.com/provable-things/ethereum-examples/blob/master/solidity/random-datasource/randomExample.sol
[6] https://docs.chain.link/getting-started/intermediates-tutorial