April 29, 2024

Blockchain Final Project: Attacking Ethereum Lottery Game “1000 Guess”

Final Report

Authors:

Kyri Christensen
Domenic Lo Iacono
Sierra Kennedy

Background

Certain blockchain platforms like Ethereum allow for the use of private variables within smart contracts, these variables are not truly private in the conventional sense. Despite attempts to obscure them from direct public view, they can still be accessed or inferred through various means. Tools like Hardhat offer functions such as ethers.provider.storageAt() that enable developers to inspect storage slots within smart contracts, where variables are stored in 32-byte increments. Additionally, analyzing the blockchain's history and transaction data can reveal patterns or dependencies that disclose supposedly private information.

Pseudo-random number generators (PRNGs) are algorithms that are used to generate sequences of numbers that appear random but are deterministic. This means they produce the same sequence given the same initial conditions or seed value. These generators are crucial in various computational tasks, simulations, and cryptographic algorithms. Despite the appearance of randomness, PRNGs can exhibit predictability under certain circumstances. One key factor is their reliance on a starting value (seed), which, if known, can allow someone to reproduce the entire sequence of generated numbers. Some PRNG algorithms might exhibit patterns or cycles over time, making them vulnerable to prediction if their internal workings are understood.

Introduction

The "1000 Guess" game was a distinctive lottery-style decentralized application (dapp) deployed on the Ethereum blockchain, showcasing the innovative capabilities of smart contracts. This game operated through Ethereum's decentralized infrastructure, where participants placed bets based on their position in the sequence of players. For instance, if a player was the fifth to place a bet, their guess would be the number 5. The winning number was determined by the

position of the winning player in the sequence of bets, adding an element of anticipation and timing to the gameplay. The player who correctly guessed the winning number based on their position would receive the accumulated Ether from all bets placed. This setup created an interactive and engaging gaming experience within the Ethereum ecosystem, demonstrating how blockchain technology can facilitate transparent and secure gaming interactions governed by smart contract logic.

Problem Statement

The challenges encountered with the "1000 Guess" game underscore the complexities of maintaining privacy and fairness within blockchain-based applications. Despite attempts to keep certain variables confidential, such as the seed used for the pseudo-random number generator (PRNG), the game's reliance on the transparent nature of the public blockchain introduced vulnerabilities. All contract variables, including the supposedly private seed, were visible on the blockchain ledger, making it possible for knowledgeable users to analyze this information and predict outcomes. This situation highlights the critical importance of considering transparency and security implications when designing and deploying blockchain-based games and applications. Developers must implement robust privacy measures and carefully assess the exposure of sensitive data to ensure fairness and integrity in decentralized environments, where code execution and data visibility are immutable and publicly accessible.

Solution

This project consists of three components. First, we implement the vulnerable version of the 1000 Guess game. Testing at this stage will confirm that the game functions as intended in

the absence of a malicious user looking to exploit the lottery calculations. Ideally, no participant in this game has a higher rate of winning than any other if the same users randomly make bets over many games. Developing this contract gives us an understanding of the components that factor into the calculation of the winning number and how different variables can be affected.

In the second stage, we developed an exploit for the vulnerable contract. This involves automating the retrieval of internal data and ensuring that the attacker only places a bet when they are sure their betting position is the winning position.

The third stage provides a modified version of the 1000 Guess game that is not vulnerable. By using an oracle to provide random numbers rather than a calculation based on internal variables, an attacker cannot predict whether they will be betting in the winning position. The exploit used in stage two should not allow an attacker to win with a greater likelihood than the other players of the game.

Project Implementation

Phase 1: Game implementation

The game was implemented as a modified version of the 1000 Guess game originally playable on the blockchain before the identified vulnerability caused it to be shut down. In the original implementation, once 1000 players placed a bet, a function would calculate a winner and reward them with the entire value of the contract minus a developer fee. For this project, the contract was modified to calculate a winner after the 10th player bet for ease of testing.

The contract accepts bets from players, and after the 10th player, it uses a combination of internal variables and characteristics of the current block to determine a winner out of those who have bet. When created, it was assumed that since the variables were private, one could not read

them to calculate the winning position before the game concluded. When a bet is placed, the addGuess() function is called. This updates the value of currentHash, the calculation of which is shown in Figure 1. If this guess is the 10th, it then uses this value to calculate the winning position in an array of betters.

Figure 1: addGuess() and calculation of currentHash

```
function _addGuess() private {
    require(msg.value >= bet, "Place a bigger bet");
    currentHash = sha256(abi.encodePacked(block.timestamp, block.coinbase, block.difficulty, currentHash));
    if ((numGuesses + 1) <= arraySize) {
        guesses[numGuesses].addr = msg.sender;
        numGuesses = numGuesses + 1;
        if (numGuesses >= arraySize) {
            endGame();
        }
    }
}
```

Figure 2: Calculating lotteryNum to decide the winner

```
function endGame() private {
    state = State.Ended;
    uint256 blockTimestamp = block.timestamp;
    uint256 lotteryNum = (uint256(currentHash) + blockTimestamp) % (maxGuess + 1);
    findWinner(lotteryNum);

    uint256 prize = getLotteryWinnings();
    payable(winner).transfer(prize);
    payable(developer).transfer(address(this).balance);

    numGuesses = 0;
    gameIndex += 1;
    state = State.Started;
    startTime = block.timestamp;
}
```

Figure 3: Using lotteryNum to get winner in array of players

```
function findWinner(uint256 lotteryNum) private {
    uint256 win = lotteryNum % numGuesses;
    winner = guesses[win].addr;
}
```

Phase 2: The Attack

The attack involved using the Hardhat ethers functionality to access private variables stored in the smart contract of the "1000 Guess" game. By obtaining the current hash variable from the game's smart contract, we were able to calculate the random number that determines the winning player's index. Since we developed the code for the game, we knew how the random number was generated and applied the same algorithms. The random number is generated using the SHA256 hash function applied to specific block information including the timestamp, Coinbase (address that receives mining rewards), difficulty level, and the hash of the previous guess, which we accessed through the storage function. This hash value is then used with modulo division to determine the final index value that corresponds to the winning guess.

The attack strategy involved checking whether the last guess would result in a win. If it did, the attack would place the final guess and secure the lottery winnings. It's important to recognize that this attack method has a limited success rate, specifically a 1 in 10 chance of winning, considering the game's reduced number of guesses from 1000 to 10.

Figure 4: Accessing Private Variables

```
// (address target, bytes32 curhash, uint256 arraySize, uint attackerInd)
const curhash = ethers.provider.getStorage(game, 0);
```

Figure 5: Calculating Winning Bet

```
curhash = sha256(abi.encodePacked(block.timestamp, block.coinbase, block.difficulty, curhash));
uint256 lotteryNum = (uint256(curhash) + block.timestamp) % (maxGuess + 1);
uint256 i = lotteryNum % arraySize;

if(attackerInd != i) {
    revert(Strings.toString(i));
}

(bool success, ) = target.call{value: 1 ether}(abi.encodeWithSignature("addGuess()"));
if (!success)
{
    revert("Call failed");
}
emit Won(address(this).balance);
payable(owner).transfer(address(this).balance);
```

Phase 3: Secure Game

Utilizing Chainlink VRF and Remix, a secure version of the Thousand Guess game has been implemented along with some quality of life improvements.

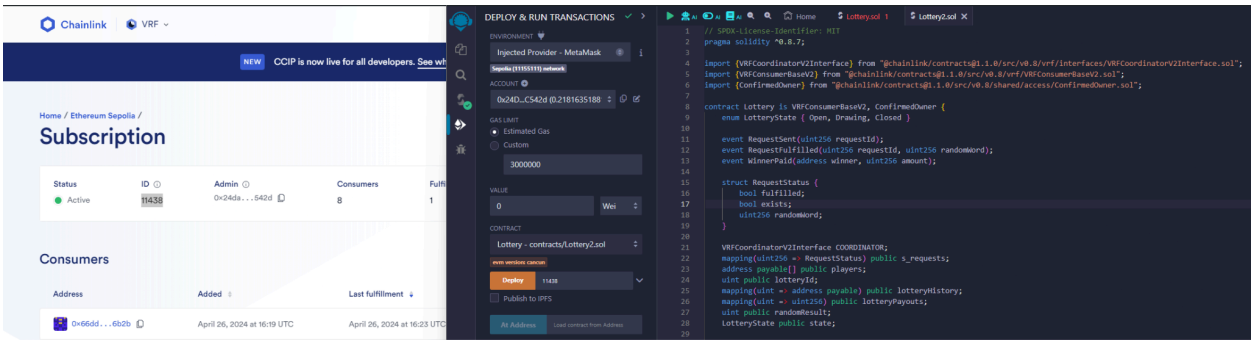
To start, the game no longer requires that a certain amount of guesses have to be made before the lottery is decided. This not only allowed for easier testing when showing the effectiveness of the security but allowed for more flexibility for the lottery owner. Another key difference compared to the previous contracts is the environment. Through Remix, the Sepolia testnet was accessed through the injection of a Metamask wallet as opposed to the use of Hardhat. Additionally, all testing was done manually through the Remix GUI.

Next, exploring the process of successfully generating a random number to be used in the lottery. Using the provided documentation from Chainlink: https://docs.chain.link/vrf/v2/subscription/examples/get-a-random-number, the process of generating a random number was simple and repeatable. As mentioned previously the Sepolia testnet was utilized by injecting a Metamask wallet. By utilizing testnet ETH and LINK a prepaid VRF subscription was put into place.

Figure 6: Chainlink subscription page

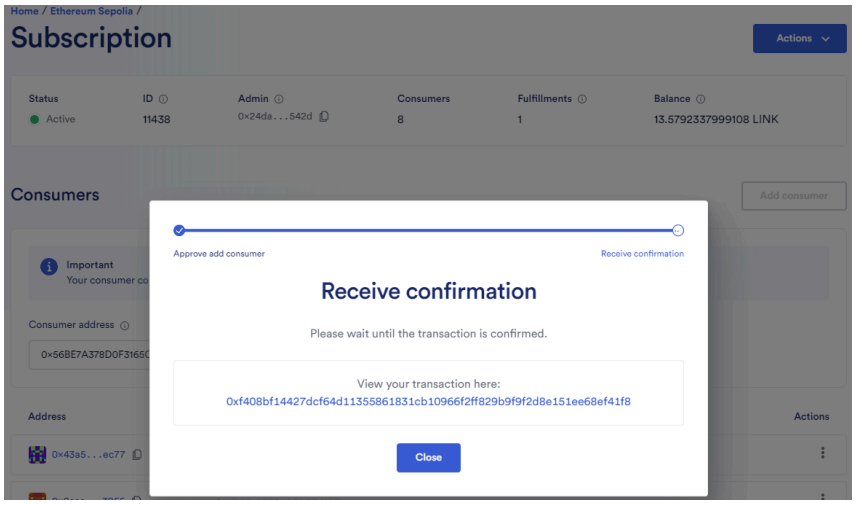
Following the creation of the subscription the consumer which is the Lottery contract, can now be put into place. By inheriting from the VRF2.0 Consumer contract the Lottery contract is able to use the subscription after being deployed with the subscription ID.

Figure 7: Code and deployment through Remix



After the contact has been deployed successfully the contract must be added to the subscription as a consumer through the web interface by giving the contract address.

Figure 8: Consumer added for VRF functionality



At this point the contract is now completely set up and verifiable random numbers from Chainlink's oracle are now implemented in the Lottery contract. As a result any attack that relies on the use of hard coded or insecurely stored private variables to determine the winner are no longer possible.

Sources

Original 1000 Guess Game:

https://etherscan.io/address/0x386771ba5705da638d889381471ec1025a824f53#readContract

Chainlink Docs:

https://docs.chain.link/vrf/v2/subscription/examples/get-a-random-number