ð² Randomness

Randomness was introduced in the Nitrogen testnet. Contracts in Fhenix can get random numbers by calling one of the randomness functions inFHE.sol . These functions are:

import

{ FHE , euint8 , euint16 , euint32 , euint64 , euint128 , euint256 }

from

"@fhenixprotocol/contracts/FHE.sol" ;

euint8 randomValue

FHE . randomEuint8 ( ) ; euint16 randomValue = FHE . randomEuint16 ( ) ; euint32 randomValue = FHE . randomEuint32 ( ) ; euint64 randomValue = FHE . randomEuint64 ( ) ; euint128 randomValue = FHE . randomEuint128 ( ) ; euint256 randomValue = FHE . randomEuint256 ( ) ; Note that the random values are returned as encrypted values. This is a fundamental quality of randomness generation, because if the returned value was plaintext, then it would be possible to simulate the execution and predict the random value.

To see the randomness functions as part of a full example take a look at the ng-binary-guessing-game example repo.

Best practice: Ensure caller is not a Contractâ

When acting upon the resulting random numbers, it is important to keep the following scenario in mind.

Suppose we have a simple game contract that you can send funds to, and with a probability ofP=0.5 (or, 50% chance), you will receive double the funds back.

contract

RandomLucky

{ function

play ( )

external

payable

{ require (msg . value

0 ,

"You need to send some FHE") ;

// Generate a random encrypted number euint8 outcome = FHE . randomEuint8 ( ) ; uint8 outcomeDecrypted = outcome . decrypt ( ) ;

// If the outcome is even, send double the value back to the sender if

(outcomeDecrypted %

2

==

0)

{ uint prize = msg . value *

2 ; require (address (this) . balance

= prize ,

"Contract does not have enough balance") ; payable (msg . sender) . transfer (prize) ; } // If the outcome is odd, the contract keeps the value }

// Fallback function to receive FHE receive ( )

external

payable

{ } } An adversary could could call the randomness consumer function, check the result of the random, and revert the transaction if that result were not favorable.

In this case:

contract

Adversary

{ RandomLucky game ;

constructor (address gameAddress)

{ game =

RandomLucky (gameAddress) ; }

// Function to attack the RandomLucky contract function

attack ( )

public

payable

{ require (msg . value

0 ,

"Must send some FHE to attack") ;

// Store the initial balance of the contract uint256 initialBalance =

address (this) . balance ;

// Call the play function of the RandomLucky contract game . play { value : msg . value } ( ) ;

// Check if the balance did not increase if

(address (this) . balance <= initialBalance)

{ revert ("Did not win, reverting transaction") ; } } } To prevent this kind of attacks, it is recommended to not allow contracts to call functions that act upon random numbers, like so:

modifier

callerIsUser ( )

{ require (tx . origin == msg . sender ,

"The caller is another contract") ; _ ; }

function

play ( ) callerIsUser { . . . } If your randomness consumer functionmust be callable by another contract, it is recommended to split the consumption and reveal into separate functions:

struct

UserData

{ uint256 amount ; euint8 outcome ; uint256 block ; bool revealed ; }

mapping

(address

=> UserData)

private userData ;

function

play ( )

external

payable

{ require (msg . value

0 ,

"You need to send some FHE") ; require (userData [msg . sender] . amount ==

0 ,

"Already playing")

// Store the amount played and the outcome to be revealed later userData [msg . sender]

=

UserData ({ amount : msg . value , outcome : FHE . randomEuint8 ( ) , block : block . number , revealed :

false }) ; }

function

reveal ( )

external

{ UserData storage data = userData [msg . sender] ;

// Ensure that random number cannot be consumed and revealed in the same block require (block . number

data . block ,

"Cannot reveal in same block") ; require (! data . revealed ,

"Already revealed") ;

uint8 outcomeDecrypted = data . outcome . decrypt ( ) ; if

(outcomeDecrypted %

2

==

0)

{ uint256 prize = data . amount *

2 ; require (address (this) . balance

=

(prize) ,

"Contract does not have enough balance") ; payable (msg . sender) . transfer (prize) ; }

data . amount =

0 ; data . revealed =

true ; } Warning

Randomness in View functionsâ

Randomness is guaranteed to be unique for each transaction, but not for eacheth_call . Specifically, two eth_calls to the same contract, on the same block may receive the same random value (more on this below). How does it work? Random

generation takes as input a seed, and returns a random number which is unique for each seed and key sets.

To cause each random to be different for each transaction, the seed is created from a) the contract address, b) the transaction hash, and c) a counter that gets incremented each transaction.

seed = hash(contract_address, tx_hash, counter) For eth calls, which don't have a tx_hash nor can use a counter, we use the block hash instead, that's why two quick subsequent calls to the same contract may return the same random number.

seed = hash(contract_address, block_hash) Edit this page

Previous ðªµ Console.log Next ð§ââï¸ Types and Operations