On-Chain Randomness

Generating pseudo-random values in Move is similar to solutions in other languages. A Move function can create a new instance of RandomGenerator and use it for generating random values of different types, for example, generate_u128(&mut generator), generate_u8_in_range(&mut generator, 1, 6) , or,

Random has a reserved address 0x8 . See random.move for the Move APIs for accessing randomness on Sui.

Although Random is a shared object, it is inaccessible for mutable operations, and any transaction attempting to modify it fails.

Having access to random numbers is only one part of designing secure applications, you should also pay careful attention to how you use that randomness. To securely access randomness:

Be aware that some resources that are available to transactions are limited. If you are not careful, an attacker can break or exploit your application by deliberately controlling the point where your function runs out of resources.

Concretely, gas is such a resource. Consider the following vulnerable code:

Observe that the gas costs of a transaction that calls insecure_play depends on the value of win . An attacker could call this function with a gas budget that is sufficient for the "happy flow" but not the "unhappy one", resulting in it either winning or reverting the transaction (but never losing the payment).

The Random API does not automatically prevent this kind of attack, and you must be aware of this subtlety when designing your contracts.

Other limited resources per transaction that you should consider are:

For many use cases this attack is not an issue, like when selecting a raffle winner, or lottery numbers, as the code running is independent of the randomness. However, in the cases where it can be problematic, you can consider one of the following:

While composition is very powerful for smart contracts, it opens the door to attacks on functions that use randomness. Consider for example a betting game that uses randomness for rolling dice:

An attacker can deploy the next function:

The attacker can now call attack with a guess, and always revert the fee transfer if the guess is incorrect.

To protect against composition attacks, define your function as a private entry function so functions from other modules cannot call it.

The Move compiler enforces this behavior by rejecting public functions with Random as an argument.

A similar attack to the one previously described involves PTBs even when play_dice is defined as a private entry function. For example, consider the entry play_dice(guess: u8, fee: Coin, r: &Random, ctx: &mut TxContext): Ticket { … } function defined earlier, the attacker can publish the function

and send a PTB with commands play_dice(...), attack(Result(0)) where Result(0) is the output of the first command. As before, the attack takes advantage of the atomic nature of PTBs and always reverts the entire transaction if the guess was incorrect, without paying the fee. Sending multiple transactions can repeat the attack, each one executed with different randomness and reverted if the guess is incorrect.

To protect against PTB-based composition attacks, Sui rejects PTBs that have commands that are not TransferObjects or MergeCoins following a MoveCall command that uses Random as an input.

RandomGenerator is secure as long as it's created by the consuming module. If passed as an argument, the caller might be able to predict the outputs of that RandomGenerator instance (for example, by calling bcs::to_bytes(&generator) and parsing its internal state).

The Move compiler enforces this behavior by rejecting public functions with RandomGenerator as an argument.

If you want to call roll_dice(r: &Random, ctx: &mut TxContext) in module example , use the following code snippet:

random.move

Limited resources and

Be aware that some resources that are available to transactions are limited. If you are not careful, an attacker can break or exploit your application by deliberately controlling the point where your function runs out of resources.

Concretely, gas is such a resource. Consider the following vulnerable code:

Observe that the gas costs of a transaction that calls insecure_play depends on the value of win . An attacker could call this function with a gas budget that is sufficient for the "happy flow" but not the "unhappy one", resulting in it either winning or reverting the transaction (but never losing the payment).

The Random API does not automatically prevent this kind of attack, and you must be aware of this subtlety when designing your contracts.

Other limited resources per transaction that you should consider are:

For many use cases this attack is not an issue, like when selecting a raffle winner, or lottery numbers, as the code running is independent of the randomness. However, in the cases where it can be problematic, you can consider one of the following:

While composition is very powerful for smart contracts, it opens the door to attacks on functions that use randomness. Consider for example a betting game that uses randomness for rolling dice:

An attacker can deploy the next function:

The attacker can now call attack with a guess, and always revert the fee transfer if the guess is incorrect.

To protect against composition attacks, define your function as a private entry function so functions from other modules cannot call it.

The Move compiler enforces this behavior by rejecting public functions with Random as an argument.

A similar attack to the one previously described involves PTBs even when play_dice is defined as a private entry function. For example, consider the entry play_dice(guess: u8, fee: Coin, r: &Random, ctx: &mut TxContext): Ticket { … } function defined earlier, the attacker can publish the function

and send a PTB with commands play_dice(...), attack(Result(0)) where Result(0) is the output of the first command. As before, the attack takes advantage of the atomic nature of PTBs and always reverts the entire transaction if the guess was incorrect, without paying the fee. Sending multiple transactions can repeat the attack, each one executed with different randomness and reverted if the guess is incorrect.

To protect against PTB-based composition attacks, Sui rejects PTBs that have commands that are not TransferObjects or MergeCoins following a MoveCall command that uses Random as an input.

RandomGenerator is secure as long as it's created by the consuming module. If passed as an argument, the caller might be able to predict the outputs of that RandomGenerator instance (for example, by calling bcs::to_bytes(&generator) and parsing its internal state).

The Move compiler enforces this behavior by rejecting public functions with RandomGenerator as an argument.

If you want to call roll_dice(r: &Random, ctx: &mut TxContext) in module example , use the following code snippet:

random.move

Use (non-public)

While composition is very powerful for smart contracts, it opens the door to attacks on functions that use randomness. Consider for example a betting game that uses randomness for rolling dice:

An attacker can deploy the next function:

The attacker can now call attack with a guess, and always revert the fee transfer if the guess is incorrect.

To protect against composition attacks, define your function as a private entry function so functions from other modules cannot call it.

The Move compiler enforces this behavior by rejecting public functions with Random as an argument.

A similar attack to the one previously described involves PTBs even when play_dice is defined as a private entry function. For example, consider the entry play_dice(guess: u8, fee: Coin, r: &Random, ctx: &mut TxContext): Ticket { … } function defined earlier, the attacker can publish the function

and send a PTB with commands play_dice(...), attack(Result(0)) where Result(0) is the output of the first command. As before, the attack takes advantage of the atomic nature of PTBs and always reverts the entire transaction if the guess was incorrect, without paying the fee. Sending multiple transactions can repeat the attack, each one executed with different randomness and reverted if the guess is incorrect.

To protect against PTB-based composition attacks, Sui rejects PTBs that have commands that are not TransferObjects or MergeCoins following a MoveCall command that uses Random as an input.

RandomGenerator is secure as long as it's created by the consuming module. If passed as an argument, the caller might be able to predict the outputs of that RandomGenerator instance (for example, by calling bcs::to_bytes(&generator) and parsing its internal state).

The Move compiler enforces this behavior by rejecting public functions with RandomGenerator as an argument.

If you want to call roll_dice(r: &Random, ctx: &mut TxContext) in module example , use the following code snippet:

random.move

Programmable transaction block (PTB) restrictions

A similar attack to the one previously described involves PTBs even when play_dice is defined as a private entry function. For example, consider the entry play_dice(guess: u8, fee: Coin, r: &Random, ctx: &mut TxContext): Ticket { … } function defined earlier, the attacker can publish the function

and send a PTB with commands play_dice(...), attack(Result(0)) where Result(0) is the output of the first command. As before, the attack takes advantage of the atomic nature of PTBs and always reverts the entire transaction if the guess was incorrect, without paying the fee. Sending multiple transactions can repeat the attack, each one executed with different randomness and reverted if the guess is incorrect.

To protect against PTB-based composition attacks, Sui rejects PTBs that have commands that are not TransferObjects or MergeCoins following a MoveCall command that uses Random as an input.

RandomGenerator is secure as long as it's created by the consuming module. If passed as an argument, the caller might be able to predict the outputs of that RandomGenerator instance (for example, by calling bcs::to_bytes(&generator) and parsing its internal state).

The Move compiler enforces this behavior by rejecting public functions with RandomGenerator as an argument.

If you want to call roll_dice(r: &Random, ctx: &mut TxContext) in module example , use the following code snippet:

random.move

Instantiating

RandomGenerator is secure as long as it's created by the consuming module. If passed as an argument, the caller might be able to predict the outputs of that RandomGenerator instance (for example, by calling bcs::to_bytes(&generator) and parsing its internal state).

The Move compiler enforces this behavior by rejecting public functions with RandomGenerator as an argument.

If you want to call roll_dice(r: &Random, ctx: &mut TxContext) in module example , use the following code snippet:

random.move

Accessing

If you want to call roll_dice(r: &Random, ctx: &mut TxContext) in module example , use the following code snippet:

random.move

Related Links

random.move