Is 68 a random number? There's something wrong with the question. Randomness is not a property of an individual number but rather of a system or process. Randomness is very often the source of insecurity in systems, as if the underlying pseudorandom process is insecure, then the entire system will be as well.

What's "good enough" for randomness in cryptography? **Random ≈ unpredictable**. The goal of generating a random key is to prevent the adversary from doing some computation that relies on the key, then what we need is for that key to be unpredictable.

Unpredictability is contingent on questions like "to whom?" and "when?". We typically discuss it in context.

Pseudorandom Generator (PRG)

Definition (PRG)**.** *Takes a small random "seed" as input (typically 256-bits). Generates a long output sequence that is indistinguishable from random.*

Defined by a game:

- *Mallory sees either a random function or a PRG with a random seed.*
- *Mallory guesses which of the two she saw.*
- *Can Mallory win more than half the time? If no, then it's a "secure PRG".*

The PRG has a "hidden state" s and operates as follows:

1. Seed goes into the *init* function, which produces initial state s_0.
2. Apply $out(s_0)$ to produce $output_0$.
3. Call the *advance* function, which produces s_1, the next hidden state of the generator.
4. Continue for as long as you'd like.

We can define a particular generator by defining the *init*, *out*, and *advance* functions. Outputs will be of a particular size. You can slice, buffer, and concatenate as necessary.

Another useful property: **no backtracking**. If an adversary learns the hidden state of the generator at any time, it cannot recover earlier output. For example, if the adversary can invert the *advance* function, then they can produce the seeds used earlier on in the process.

Example: PRG #1 (allows backtracking)

Assume we have PRF f. This PRG just keeps a counter and increments it over time.

- $init \rightarrow (seed, 0)$

- $advance : (seed, k) \rightarrow (seed, k+1)$
- $out : f(seed, k)$

If you see the state any any time, just decrement the counter to get previous outputs.

Example: PRG #2 (no backtracking)

- $init \rightarrow seed$
- $advance : seed \rightarrow f(s, 0)$
- $out : f(s, 1)$

If you see $f(s, 0)$, you cannot retrieve s, which was the previous PRF key.

Pseudorandomness as a System Service

Starting point: PRG inside the system. But there are two problems:

1. Where does the seed come from?
2. How do we recover if the state leaks? (We might not even know it leaked, which suggests that we should "always be recoverable".)

Add an operation to our PRG, **recover**: $(state, data) \rightarrow state$. The goal is to use some data that the adversary doesn't know about to come up with new, unpredictable states constantly. You can also get an initial seed by recovering (i.e., just compute $recover(0, data)$).

To recover:

1. **Collect** unpredictable data.
    - Pick up the exact history of key presses (i.e., which key and at which time it was pressed).
    - Pick up the exact path of the mouse as it moves across the screen.
    - Take periodic screenshots.
    - Pick up the exact history of network packet traffic.
    - Pick up internal temperature of the computer.
    - Pick up mic or camera input.
    - *Most of this will be low quality data, highly correlated.*
2. **Extract** compact set of bits.

- Run SHA256(data).

Recovery process consists of:

1. Add state to the randomness pool.
2. Take SHA256(data) output and make it the new state.

In practice, it's really hard to estimate how much randomness we have in the pool; we often try to be as conservative as possible in recovery, waiting until we almost certainly have way too much data before running the recovery process.

Message Confidentiality

Process:

1. Alice inputs *plaintext* and *key* into an encryption function E.
2. Bob puts the *ciphertext* and *key* through the decryption function D.
3. Eve wants to intercept and read the *ciphertext*.

The **one-time pad** gives us confidentiality:

- $E(k, x) = k \oplus x$
- $D(k, x) = k \oplus x$

If k is chosen randomly, then the ciphertexts are indistinguishable from random. So the one-time pad is secure, but unfortunately, the key must be as big as the message and it cannot be reused.

We have multiple definitions of security in this context. We'll use the following.

Definition (Semantic Security)**.** *Defined through a game against Mallory. We pick a secret key k. Mallory sends us x_0, and we send back $E(k, x_0)$, repeat n times. Then Mallory gets to guess y_0, y_1. We flip a coin to pick a bit b. We send back $E(k, y_b)$. Mallory attempts to guess b and wins if she can do better than random (i.e., if there's no such strategy for Mallory that's effective with non-negligible advantage, then the encryption process is secure).*

This doesn't say that Mallory is unable to tamper with or forge messages; she just can't tell them apart.

The one-time pad fails this test unless we extend it and use it in pieces.

Definition (Stream Cipher)**.** *Start with a fixed-size random k. Add a "nonce" (unique but non-secret value). Use $(k||nonce)$ to seed a PRG and XOR the message with the output of that PRG.* ***Rule****: Don't re-use a $(key, nonce)$ pair. If you follow this rule, you get* ***semantic security****.*

In practice, you want *both* confidentiality and integrity.

Confidentiality & Integrity

Can we combine a cipher (for confidentiality) and a MAC (for integrity)? Here are several ways to do so:

1. $E(x||M(x))$, as used by SSL/TLS.
2. $E(x)||M(E(x))$, as used by IPSec.
3. $E(x)||M(x)$, as used by SSH.

It's disappointing that different secure systems do this differently, but it's reflective of history: up until 2001, we didn't know how to do this properly. However, **#2 is the winner**.

Theorem 0.1. *If E is semantically secure and M is a secure MAC, then $E(x)||M(E(x))$ is secure (in the desired sense: eavesdropper can't learn anything about x, nor can they tamper with the ciphertext to get Bob to accept a message that Alice didn't send–essentially, you don't break the properties enforced by E and M individually).*

The other options (#1, #3) are only secure with certain choices of E and M.