Where to get random bits?

Unix-like Systems:
- Typically have: `/dev/random` and `/dev/urandom`.
- Cryptographic PNGs, continually re-seed from various sources of entropy.
- Treat like files of random bits you can read from. But, like real files, reading from them can fail. So include lots of sanity checks when you use them.
- The details of `/dev/random` and `/dev/urandom` differ from system to system.
- Typically, they maintain entropy pools of “random” bits draw from system behavior (e.g., i/o devices, network and user activity, etc.)
  
  !!! These pools may be empty at boot time, and this can cause problems.

Windows: `CryptGenRandom()`

Intel processors: `RDRAND` — draws from a hardware source of randomness

RSA

I sincerely hope you'll never have to implement RSA from scratch.

If you're asked to, run as fast as you can and question the sanity of the person who asked you to do so.

It took decades for cryptographers and engineers to develop RSA implementations that are fast, sufficiently secure, and hopefully free of debilitating bugs, so you really don't want to reinvent RSA.

J.-P. Aumanson. *Serious Cryptography*
Textbook RSA Encryption

**Setup**
- Each user $U_i$:
  - Picks two large (random) primes $p_{U_i}$ and $q_{U_i}$ (with $p_{U_i} \neq q_{U_i}$).
  - Computes $n_{U_i} = p_{U_i} \cdot q_{U_i}$ and $\phi(n_{U_i}) = (p_{U_i} - 1) \cdot (q_{U_i} - 1)$.
  - Picks $e_{U_i} \in \{1, \ldots, \phi(n_{U_i}) - 1\}$ with $\gcd(e_{U_i}, \phi(n_{U_i})) = 1$.
  - Computes $d_{U_i} = e_{U_i}^{-1} \mod \phi(n_{U_i})$.
  - Publishes $e_{U_i}$ and $n_{U_i}$.
  - Keeps $d_{U_i}, p_{U_i}, q_{U_i},$ and $\phi(n_{U_i})$ secret.

**Encryption**
- Bob wants to send $m \in \mathbb{Z}_n$ to Alice.
  - Computes $c = m^{e_{U_i}} \mod n_{U_i}$.
  - Sends $c$ to Alice.

**Decryption**
- Alice wants to decrypt $c$.
  - Computes $m = c^{d_{U_i}} \mod n_{U_i}$.

RSA with Optimal Asymmetric Encryption Padding (RSA-OAEP)

For 2048-bit RSA, this scheme uses:
- $H \in \{0,1\}^{256}$, a constant of the scheme
- A pseudo-random generator $g : \{0,1\}^{256} \rightarrow \{0,1\}^{1864}$
- A hash-function $h : \{0,1\}^{1864} \rightarrow \{0,1\}^{256}$ (e.g., SHA-256)

**Encryption of $m_0 \in \{0,1\}^{1520}$**
- $\bar{0} = 270$-many $0$’s
- $r_0 \in \{0,1\}^{256}$
- $m_1 \leftarrow H(001 \parallel m_0)$
- $m_2 \leftarrow g(r_0) \oplus m_1$
- $r_1 \leftarrow r_0 \oplus h(m_2)$
- $x \leftarrow 00 \parallel r_1 \parallel m_2$
- $c \leftarrow x^e \mod n$

**Decryption of $c$**
- $x \leftarrow c^d \mod n$ // $x = 00\|r_1\|m_2$
- $r_0 \leftarrow r_1 \oplus h(m_2)$
- $m_1 \leftarrow g(r_0) \oplus m_2$ // $m_1 = H(\bar{0}01 \parallel m_0$)
- return $m_0$

RSA-OAEP, Continued

**Encryption of $r_0, m_1, r_1, m_2 \in \{0,1\}^{256}$**
- $r_0 \in \{0,1\}^{256}$
- $m_1 \leftarrow H(\bar{0}01 \parallel m_0)$
- $m_2 \leftarrow g(r_0) \oplus m_1$
- $r_1 \leftarrow r_0 \oplus h(m_2)$
- $x \leftarrow 00 \parallel r_1 \parallel m_2$
- $c \leftarrow x^e \mod n$

**Decryption of $c$**
- $x \leftarrow c^d \mod n$ // $x = 00\|r_1\|m_2$
- $r_0 \leftarrow r_1 \oplus h(m_2)$
- $m_1 \leftarrow g(r_0) \oplus m_2$ // $m_1 = H(\bar{0}01 \parallel m_0$)
- return $m_0$

- $(r_0, m_1) \sim (r_1, m_2)$ is an example of an all-or-nothing transformation.
- To recover $m_1$, you need to recover the entire $r_0$ and the entire $m_2$.
  - Because of $h$, you need the entire $m_2$ to recover $r_0$ from $r_1$.
  - Because of $g$, you need the entire $r_0$ to recover $m_1$ from $m_2$.
- So, figuring out just part of $x$ does you no good.
Textbook RSA Signatures: Trivial Forgeries

Setup for RSA Signatures
Just like RSA-encryption

Signing
Bob wants to sign a \( m \in \mathbb{Z}_n \).
1. Computes \( s = m^{d_B} \mod n_B \).
2. Sends \((m, s)\) to Alice.

Verifying
Alice wants to check \((m, s)\)
1. Tests \( m \equiv s^{e_B} \mod n_B \).

Trivial Forgery
For all \( n_U \) and \( d_U \) and for \( x = 0, 1, (n_U - 1) \):
\( x^{d_U} \equiv x \mod (n_U) \).
So we can forge signatures for \( m = 0, 1, (n_U - 1) \) without knowing \( d_U \).

Textbook RSA Signatures: Blinding Attack

Setup for RSA Signatures
Just like RSA-encryption

Signing
Bob wants to sign a \( m \in \mathbb{Z}_n \).
1. Computes \( s = m^{d_B} \mod n_B \).
2. Sends \((m, s)\) to Alice.

Verifying
Alice wants to check \((m, s)\)
1. Tests \( m \equiv (s^{e_B} \mod n_B) \).

Blinding Attack
Suppose \( m \) is a message Alice would not sign.
Suppose you find \( r \) such that Alice would sign message \( r^{e_A}m \mod n_A \).
Have Alice sign \( r^{e_A}m \) with signature \( s = (r^{e_A}m)^{d_A} \mod n_A \).
Then:
\[
\begin{align*}
s \cdot r^{-1} &\equiv (r^{e_A}m)^{d_A} \cdot r^{-1} \\
&\equiv (r \cdot m^{d_A}) \cdot r^{-1} \\
&\equiv m^{d_A} \mod n_A
\end{align*}
\]
= Alice’s signature on \( m \)

Full Domain Hash Signatures & The Probabilistic Signature Scheme

Full Domain Hash Signatures
\( Hash = \) a good crypto-hash function

Signing
Bob wants to sign an \( m \).
1. Computes \( x = Hash(m) \).
2. Computes \( s = x^{d_B} \mod n_B \).
3. Sends \((m, s)\) to Alice.

Verifying
Alice wants to check \((m, s)\)
1. Tests \( Hash(m) \equiv s^{e_B} \mod n_B \).

Problem
RSA-FDH is not randomized, so it is open to certain attacks.

Probabilistic Signature Scheme
A scheme similar to RSA-OAEP that \is randomized, but a lot more complex than RSA-FDH.

Flaws in RSA Implementations: Low Entropy Primes, 1

- In 2012 researchers scanned are large chunk of the net and collected public keys from TLS certificates and SSH hosts. They found a fair number of systems with either:
  - identical RSA moduli
    If Alice and Bob have \( n_A = n_B \), then they can compute each other’s decryption exponents.
  - similar RSA moduli (i.e., a shared prime in the moduli)
    If \( n_A = p \cdot q \) and \( n_B = p' \cdot q' \), then \( \gcd(n_A, n_B) = p \) and \( q = n_A / p \) and \( q' = n_B / p \).

How did this happen?
- Many systems determine RSA keys at boot-time.

```
prng.seed(seed)
p = prng.generate_random_prime()
q = prng.generate_random_prime()
n = p*q
```

- What happens when two systems with the same seed run this code?
Flaws in RSA Implementations: Low Entropy Primes, 2

- In 2012 researchers scanned a large chunk of the net and collected public keys from TLS certificates and SSH hosts. They found a fair number of systems with either:
  - identical RSA moduli
    
    If Alice and Bob have $n_A = n_B$, then they can compute each other’s decryption exponents.
  - similar RSA moduli (i.e., a shared prime in the moduli)
    
    If $n_A = p \cdot q$ and $n_B = p \cdot q'$, then $\gcd(n_A, n_B) = p$ and $q = n_A/p$ and $q' = n_B/p$.

How did this happen?
- Many systems determine RSA keys at boot-time.

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_entropy()
q = prng.generate_random_prime()
n = p*q
```

- What happens when two systems with the same seed run this code?

The Bellcore Attack on RSA-Chinese-Remainder-Theorem

A fault-injection attack, forces an error in the execution of an algorithm by altering a circuit’s voltage or shooting a laser at part of the circuit.

- Recall that in using the CRT to compute $x^d$, you compute
  - $x_p = y^s \mod p$
  - $x_q = y^t \mod q$
  - Then $x = (x_p \cdot q \cdot (q^{-1} \mod p) + x_q \cdot p \cdot (p^{-1} \mod q)) \mod n$

- Suppose we force a mistake in the computation of $x_q$, getting a value $x'_q$.
- Let $x' = (x_p \cdot q \cdot (q^{-1} \mod p) + x'_q \cdot p \cdot (p^{-1} \mod q)) \mod n$, which is a multiple of $p$.
- Therefore, $p = \gcd(n, x - x')$ and $q = n/p$.
- Randomized versions of RSA are safe against this attack. (Why?)

Diffie-Hellman

- Diffie-Hellman is a key agreement protocol.
- Used extensively all over the net.

Possible Attacks on a Key Agreement Protocol

- The eavesdropper
  The attacker sees all messages exchanged and can modify/drop/inject messages.

- The data leak
  The attacker learns the session key and all temporary secrets for a few runs of the protocol, but doesn’t know any long-term secrets.

- The breach/corruption
  The attacker learns the long-term key of one or more party.
Security Goals of a Key Agreement Protocol

**Authentication**
Each party should be able to authenticate the other.

**Key control**
No party should be able to choose/restrict the final shared secret.

**Forward secrecy**
Even if all long-term secrets are exposed, shared secrets from previous protocol-runs cannot be computed.

**Resistance to key-compromise impersonation**
If Eve learns Alice’s long-term key, then the protocol should protect against Eve impersonating Alice.

Anonymous Diffie–Hellman

**Setup**
p, a large prime (Pub) and a, a prim. elem. of \( Z^*_p \) (Pub)

**Alice**
- Picks \( x \) \( \text{ran} \in Z^*_{p-1} \) (Priv.) and sends \( a^x \pmod p \) to Bob
- Computes \( k = (a^y)^x = a^{x+y} \pmod p \)
  - Computes \( k = (a^x)^y = a^{xy} \pmod p \).

**Bob**
- Picks \( y \) \( \text{ran} \in Z^*_{p-1} \) (Priv.) and sends \( a^y \pmod p \) to Alice
- Computes \( k = (a^x)^y = a^{xy} \pmod p \).

Recall: This is vulnerable to eavesdropper (man-in-the-middle) attacks.

Authenticated Diffie–Hellman (ADH): Strengths

**Setup**
p, a large prime (Pub) and \( g \), a prim. elem. of \( Z^*_p \) (Pub)

**Alice**
- Picks \( x \) \( \text{ran} \in Z^*_{p-1} \) (Priv.), computes \( k_A = g^x \pmod p \) & \( s_A = \text{sig}_A(k_A) \)
  - Sends \( (k_A, s_A) \) to Bob
- Computes \( k = (k_B)^x = g^{xy} \pmod p \) and verifies \( s_B \).

**Bob**
- Picks \( y \) \( \text{ran} \in Z^*_{p-1} \) (Priv.) computes \( k_B = g^y \pmod p \) & \( s_B = \text{sig}_B(k_B) \)
  - Sends \( (k_B, s_B) \) to Alice
- Computes \( k = (k_A)^y = g^{xy} \pmod p \) and verifies \( s_A \).

- The keys for the signatures are long-term keys.
- Authentication stops the man-in-the-middle attack.
- ADH provides forward secrecy; past session keys are safe.
- ADH prevents any party from controlling the value of \( g^{xy} \).

Authenticated Diffie–Hellman (ADH): Weaknesses

**Setup**
p, a large prime (Pub) and \( g \), a prim. elem. of \( Z^*_p \) (Pub)

**Alice**
- Picks \( x \) \( \text{ran} \in Z^*_{p-1} \) (Priv.), computes \( k_A = g^x \pmod p \) & \( s_A = \text{sig}_A(k_A) \)
  - Sends \( (k_A, s_A) \) to Bob
- Computes \( k = (k_B)^x = g^{xy} \pmod p \) and verifies \( s_B \).

**Bob**
- Picks \( y \) \( \text{ran} \in Z^*_{p-1} \) (Priv.) computes \( k_B = g^y \pmod p \) & \( s_B = \text{sig}_B(k_B) \)
  - Sends \( (k_B, s_B) \) to Alice
- Computes \( k = (k_A)^y = g^{xy} \pmod p \) and verifies \( s_A \).

- ADH is not secure against replay attacks.
  (You need to add key confirmation: nonces + challenges)
- ADH is also not secure against data leaks
  - If Eve learns short-term secrets, \( x \) and \( y \), Eve can impersonate either Alice or Bob.
  - If Eve learns both: (i) Alice’s short-term secret (i.e., \( x \)) and (ii) Alice’s long-term secret (the key to \( \text{sig}_A \)), then Eve can impersonate Alice.
**Menezes–Qu–Vanstone (MQV): Diffie–Hellman on Steroids**

**Setup**
- $p$ a prime with a hard discrete-log problem.
- $g$ a primitive element of $\mathbb{Z}_p^*$.

For each user $U$:
- a private long-term key $\ell_U$ & a public key $L_U = g^{\ell_U} \pmod{p}$.

**Alice**
- Picks $x \overset{\text{ran}}{\in} \mathbb{Z}_{p-1}^*$.
- Sends Bob $X = g^x \pmod{p}$.

**Bob**
- Picks $y \overset{\text{ran}}{\in} \mathbb{Z}_{p-1}^*$.
- Sends Alice $Y = g^y \pmod{p}$.

**Alice**
- Computes $K = (Y \cdot L_U^x)^{x + \ell_y} \pmod{p}$.

**Bob**
- Computes $K' = (X \cdot L_Y^x)^{y + \ell_x} \pmod{p}$.

**Claim: $K = K'$**

$$(Y \cdot L_U^x)^{x + \ell_y} = (g^y \cdot (g^{\ell_y})^y)^{x + \ell_y} \pmod{p}$$

$$(X \cdot L_Y^x)^{y + \ell_x} = (g^x \cdot (g^{\ell_y})^x)^{y + \ell_x} \pmod{p}$$

$g^{xy}$ should determine (but not be) the shared secret
- The elements of $\mathbb{Z}_p^*$ have too much structure as $(k + 1)$-bit numbers, where $2^k < p < 2^{k+1}$.
- Take SHA3($g^{xy} \pmod{p}$) as the shared secret; the hash function will spread out the randomness.

Legacy Diffie–Hellman in TSL
- TSL uses Diffie–Hellman for session key generation.
- TSL allows different flavors of Diffie–Hellman, including Anonymous Diffie–Hellman, which is subject to man-in-the-middle attacks!

**Diffie–Hellman in Practice, 1**

$p$ a prime with a hard discrete-log problem.
- $g$ a primitive element of $\mathbb{Z}_p^*$.

For each user $U$:
- a private long-term key $\ell_U$ & a public key $L_U = g^{\ell_U} \pmod{p}$.

**Alice**
- Picks $x \overset{\text{ran}}{\in} \mathbb{Z}_{p-1}^*$.
- Sends Bob $X = g^x \pmod{p}$.

**Bob**
- Picks $y \overset{\text{ran}}{\in} \mathbb{Z}_{p-1}^*$.
- Sends Alice $Y = g^y \pmod{p}$.

**Alice**
- Computes $K = (Y \cdot L_U^x)^{x + \ell_y} \pmod{p}$.

**Bob**
- Computes $K' = (X \cdot L_Y^x)^{y + \ell_x} \pmod{p}$.

MQV cannot be broken by Eve learning temporary secrets ($x$ and $y$).
- Suppose Eve learns Alice’s key $\ell_A$, Previous shared secrets are safe because they involved temporary secrets ($x$ and $y$).

There is a forward secrecy attack; to defend against it you need to add key confirmation.
- MQV is rarely used in practice: too complex, too many patents.

**Diffie–Hellman in Practice, 2**


- The authors found that people were ignoring the “Suppose $p$ is a prime with a hard-discrete log problem” proviso.
- Many Diffie–Hellman implementations where accepting primes $p$ such that $(p – 1)$ had many small divisors. This leads to troubles.
- As a result of the paper, lots of patching of major systems occurred in 2016 (e.g., the OpenSSL toolkit).
- The way to guard against this is to demand that $(p – 1)/2$ is itself prime. (Why would this help?)