How Cryptosystems Work Today

• **Layered approach:** Cryptographic protocols (like “CBC mode encryption”) built on top of cryptographic primitives (like “block ciphers”)

• **Flavors of cryptography:** Symmetric (private key) and asymmetric (public key)

• Public algorithms (Kerckhoff’s Principle)

• Security proofs based on assumptions (*not this course*)

• Be careful about inventing your own! (If you just want to use some crypto in your system, use vetted libraries!)
The Cryptosystem Stack

• Primitives:
  • AES / DES / etc
  • RSA / ElGamal / Elliptic Curve (ed25519)

• Modes:
  • Block modes (CBC, ECB, CTR, GCM, ...)
  • Padding structures

• Protocols:
  • TLS / SSL / SSH / etc

• Usage of Protocols:
  • Browser security
  • Secure remote logins
Kerckhoff’s Principle

• Security of a cryptographic object should depend only on the secrecy of the secret (private) key.

• Security should not depend on the secrecy of the algorithm itself.

• Foreshadow: Need for randomness – the key to keep private
Flavors of Cryptography

• Symmetric cryptography
  • Both communicating parties have access to a shared random string $K$, called the key.

• Asymmetric cryptography
  • Each party creates a public key $pk$ and a secret key $sk$.
  • Hard concept to understand, and revolutionary! Inventors won Turing Award 😊
Both communicating parties have access to a shared random string $K$, called the key.
Asymmetric Setting

Each party creates a public key $pk$ and a secret key $sk$. 
Received April 4, 1977

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman

Abstract
An encryption method is presented with the novel property that publicly revealing an encryption key does not thereby reveal the corresponding decryption key. This has two important consequences:

1. Couriers or other secure means are not needed to transmit keys, since a message can be enciphered using an encryption key publicly revealed by the intended recipient. Only he can decipher the message, since only he knows the corresponding decryption key.

2. A message can be “signed” using a privately held decryption key. Anyone can verify this signature using the corresponding publicly revealed encryption key. Signatures cannot be forged, and a signer cannot later deny the validity of his signature. This has obvious applications in “electronic mail” and “electronic funds transfer” systems.
Flavors of Cryptography

• Symmetric cryptography
  • Both communicating parties have access to a shared random string $K$, called the key.

• Asymmetric cryptography
  • Each party creates a public key $pk$ and a secret key $sk$. 
Flavors of Cryptography

• Symmetric cryptography
  • Both communicating parties have access to a shared random string $K$, called the key.
  • Challenge: How do you privately share a key?

• Asymmetric cryptography
  • Each party creates a public key $pk$ and a secret key $sk$.
  • Challenge: How do you validate a public key?
Flavors of Cryptography

• Symmetric cryptography
  • Both communicating parties have access to a shared random string $K$, called the key.
  • Challenge: How do you privately share a key?

• Asymmetric cryptography
  • Each party creates a public key $pk$ and a secret key $sk$.
  • Challenge: How do you validate a public key?

• Key building block: Randomness – something that the adversaries won’t know and can’t predict and can’t figure out
Ingredient: Randomness

• Many applications (especially security ones) require randomness

• Explicit uses:
  • Generate secret cryptographic keys
  • Generate random initialization vectors for encryption

• Other “non-obvious” uses:
  • Generate passwords for new users
  • Shuffle the order of votes (in an electronic voting machine)
  • Shuffle cards (for an online gambling site)
C’s rand() Function

• C has a built-in random function: `rand()

```c
unsigned long int next = 1;
/* rand: return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

• Problem: don’t use `rand()` for security-critical applications!
  • Given a few sample outputs, you can predict subsequent ones
mamajoe: Hey guys, Big B is in!
More details: “How We Learned to Cheat at Online Poker: A Study in Software Security”
PS3 and Randomness

Hackers obtain PS3 private cryptography key due to epic programming fail? (update)


- 2010/2011: Hackers found/released private root key for Sony’s PS3
- Key used to sign software – now can load any software on PS3 and it will execute as “trusted”
- Due to bad random number: same “random” value used to sign all system updates
A recent example: keypair

https://securitylab.github.com/advisories/GHSL-2021-1012-keypair/

- keypair is a JS library for generating (asymmetric) keypairs

The output from the Lehmer LCG is encoded incorrectly. The specific line with the flaw is:

\[ b.putByte(String.fromCharCode(next & 0xFF)) \]

The definition of putByte is

\[ [...]putByte = function(b) { this.data += String.fromCharCode(b); }; \]

Since we are masking with 0xFF, we can determine that 97% of the output from the LCG are converted to zeros. The only outputs that result in meaningful values are outputs 48 through 57, inclusive.

The impact is that each byte in the RNG seed has a 97% chance of being 0 due to incorrect conversion. When it is not, the bytes are 0 through 9.
How might we get “good” random numbers?
Obtaining Pseudorandom Numbers

• For security applications, want “cryptographically secure pseudorandom numbers”

• Libraries include cryptographically secure pseudorandom number generators (CSPRNG)
Obtaining Pseudorandom Numbers

• Linux:
  • /dev/random – blocking (waits for enough entropy)
  • /dev/urandom – nonblocking, possibly less entropy
  • getrandom() – syscall! – by default, blocking

• Internally:
  • Entropy pool gathered from multiple sources
    • e.g., mouse/keyboard/network timings
  • Challenges with embedded systems, saved VMs
Obtaining *Random* Numbers

• Better idea:
  • AMD/Intel’s *on-chip random number generator*
    • RDRAND

• Hopefully no hardware bugs!
Back to encryption
Confidentiality: Basic Problem

Given (Symmetric Crypto): both parties know the same secret.

Goal: send a message confidentially.

Ignore for now: How is this achieved in practice??
One weird bit-level trick

• XOR!
  • Just XOR with a random bit!

• Why?
  • Uniform output
  • Independent of ‘message’ bit
One-Time Pad

Key is a random bit sequence as long as the plaintext

Encrypt by bitwise XOR of plaintext and key:
\[ \text{ciphertext} = \text{plaintext} \oplus \text{key} \]

Decrypt by bitwise XOR of ciphertext and key:
\[ \text{ciphertext} \oplus \text{key} = (\text{plaintext} \oplus \text{key}) \oplus \text{key} = \text{plaintext} \]

Cipher achieves **perfect secrecy** if and only if there are as many possible keys as possible plaintexts, and every key is equally likely \( \text{(Claude Shannon, 1949)} \)
Advantages of One-Time Pad

• Easy to compute
  • Encryption and decryption are the same operation
  • Bitwise XOR is very cheap to compute

• As secure as theoretically possible
  • Given a ciphertext, all plaintexts are equally likely, regardless of attacker’s computational resources
  • ...as long as the key sequence is truly random
    • True randomness is expensive to obtain in large quantities
  • ...as long as each key is same length as plaintext
    • But how does sender communicate the key to receiver?
Problems with the One-Time Pad?

- Breakout Discussions
- What potential security problems do you see with the one-time pad?
- (Try not to look ahead and next slides)
- Recall two key goals of cryptography: confidentiality and integrity
Dangers of Reuse

Learn relationship between plaintexts
\[ C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = (P_1 \oplus P_2) \oplus (K \oplus K) = P_1 \oplus P_2 \]
Problems with One-Time Pad

• (1) Key must be as long as the plaintext
  • Impractical in most realistic scenarios
  • Still used for diplomatic and intelligence traffic

• (2) Insecure if keys are reused
  • Attacker can obtain XOR of plaintexts
Integrity?

Key is a random bit sequence as long as the plaintext

Encrypt by bitwise XOR of plaintext and key:
\[ \text{ciphertext} = \text{plaintext} \oplus \text{key} \]

Decrypt by bitwise XOR of ciphertext and key:
\[ \text{ciphertext} \oplus \text{key} = (\text{plaintext} \oplus \text{key}) \oplus \text{key} = \text{plaintext} \oplus (\text{key} \oplus \text{key}) = \text{plaintext} \]
Problems with One-Time Pad

• (1) Key must be as long as the plaintext
  • Impractical in most realistic scenarios
  • Still used for diplomatic and intelligence traffic

• (2) Insecure if keys are reused
  • Attacker can obtain XOR of plaintexts

• (3) Does not guarantee integrity
  • One-time pad only guarantees confidentiality
  • Attacker cannot recover plaintext, but can easily change it to something else