Public-Key Cryptography

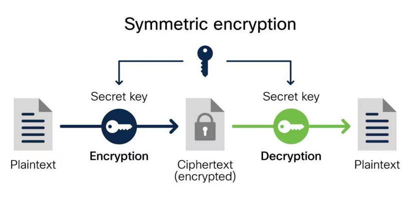
Charles Averill, Marty Vaneskahian

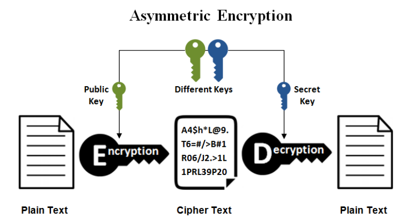
CSG CTF Bootcamp
The University of Texas at Dallas

October 2025

Asymmetric Encryption Overview

- Looked at *symmetric cryptography* last week, where key is same for encryption and decryption - very convenient but not always practical
- What if it's not convenient or safe to distribute my encryption keys to the people I want to be able to securely communicate with?
- **Key Idea**: What if we encrypt with a *public* key that everyone can know, but decrypt with a *private* key that only the decrypter knows?





RSA Overview

- RSA is the classic PKC algorithm and will serve nicely as an introductory example
- Phases: key **generation**, **distribution**, **encryption**, and **decryption**
- Public, private keys: $(n, e), (n, d)$ - n is RSA module, e is encryption exponent, d is decryption exponent

Key generation:

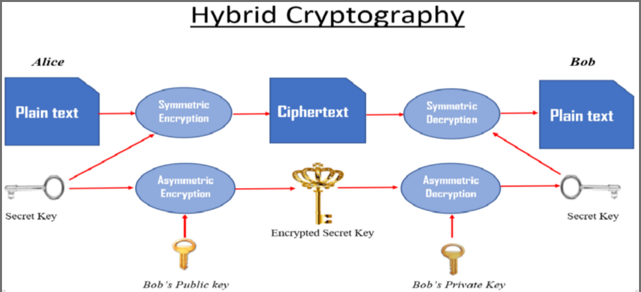
1. Select two distinct, large prime numbers p, q
2. $n := p \cdot q$
3. Select e s.t. e is **coprime**[1] to $\phi(n)$[2]
4. Select d s.t. $(e \cdot d) \mod \phi(n) = 1$ with Extended Euclidean Alg.

Key distribution: just distribute public key

[1] $coprime(n, m) \equiv GCD(n, m) = 1$
[2] $\phi(n) \equiv |\{x | x \leq n \land coprime(x, n)\}|$

RSA Encryption and Decryption

- Public, Private keys: $(n, e), (n, d)$
- **Encryption**: $c = m^e \mod n$
- **Decryption**: $m' = c^d \mod n$
- Generate random symmetric key k
- Encrypt message with k to get E_m
- Encrypt k with RSA to get E_k
- Send (E_k, E_m), receiver decrypts E_k to decrypt E_m to get message



Security Basis

- The security of RSA rests on two mathematical assumptions:
  1. **Very hard** to factor a large semiprime number $n = p \cdot q$
  2. Knowing only (n, e), it is **infeasible to compute** private exponent d
- If an attacker could factor n, they could compute

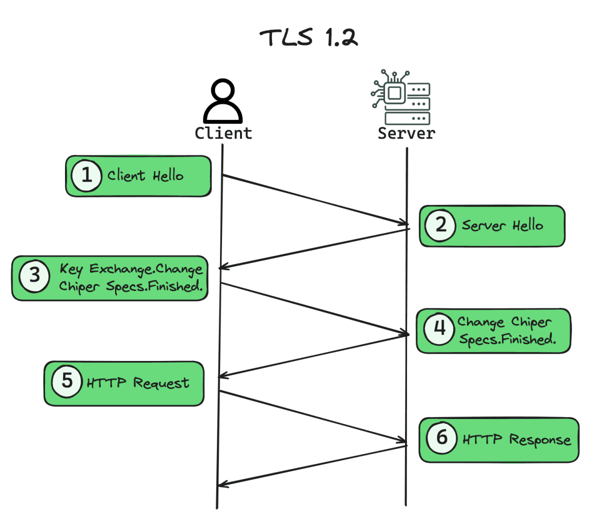
$$\phi(n) = (p-1)(q-1)$$

  and recover $d = e^{-1} \bmod \phi(n)$
- No known efficient algorithm for factoring large semiprimes
- **Key sizes**:
  - 2048-bit RSA $\approx$ 112-bit symmetric security
  - 4096-bit RSA $\approx$ 128-bit symmetric security
- **Quantum risk**: Shor's algorithm can factor efficiently on a large quantum computer, breaking RSA entirely

Use Case: TLS

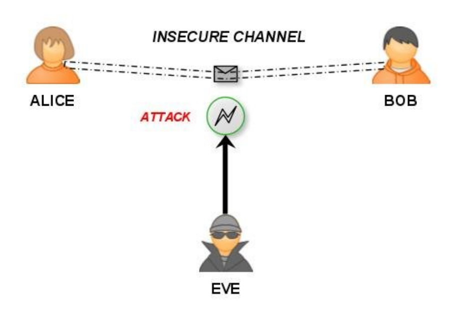
- **Transport Layer Security (TLS)** secures HTTPS
- RSA historically used in TLS for two purposes:
  1. **Key exchange**: client generates a random session key, encrypts it with server's RSA public key
  2. **Authentication**: server proves its identity by presenting RSA-signed digital certificate issued by a trusted CA
- Resulting shared symmetric key used to encrypt session data
- **Limitations:**
  - No *forward secrecy*: if private key is later compromised, past sessions can be decrypted
  - Modern TLS (1.3) replaces RSA key exchange with **Elliptic Curve Diffie–Hellman (ECDHE)**, while keeping RSA or ECDSA for authentication

Use Case: TLS

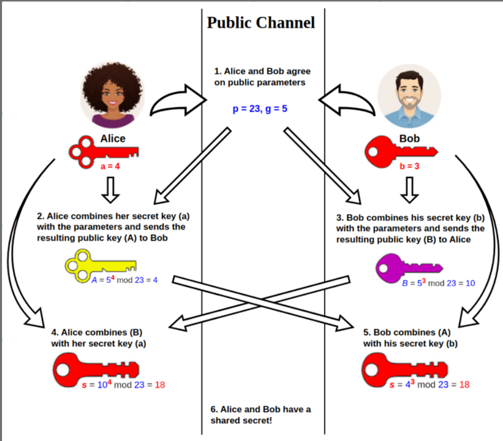

TLS 1.2
Client
Server
1 Client Hello
2 Server Hello
3 Key Exchange.Change Chiper Specs.Finished.
4 Change Chiper Specs.Finished.
5 HTTP Request
6 HTTP Response

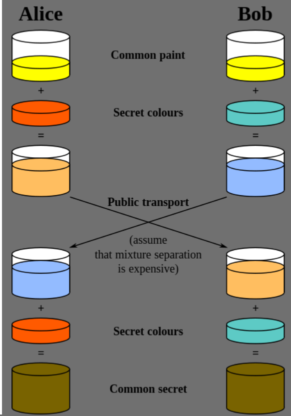
Key Exchange Overview

- Sometimes we want to use symmetric encryption, but we haven't shared keys beforehand and we only have an insecure channel - how to share keys?
- Before asymmetric encryption was invented, this was pervasive
- *Key exchange protocols* describe how to safely share keys over insecure channels



Diffie-Hellman Key Exchange





Key Exchange and RSA

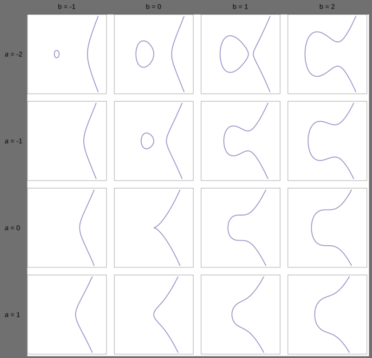
- Why bother with key exchange algorithms when we have asymmetric encryption?
- RSA is way slower than AES
- RSA keys are huge and using it everywhere would waste bandwidth
- If RSA private key is ever compromised, all following traffic is compromised. With DH, ephemeral keys are possible

Elliptic Curves

- An **elliptic curve** over a finite field $\mathbb{F}_p$ is defined by an equation:

$$y^2 = x^3 + ax + b \pmod{p}$$

- Curve parameters a, b must satisfy $4a^3 + 27b^2 \neq 0$ (no singular points)
- Points on the curve form can combine to form other curve points



Why Elliptic Curves?

- Elliptic curves are a structure with a hard problem:
  - Points on the curve form a group (you can "add" points together)
  - Given a point P and a multiple $Q = k \cdot P$, it's easy to compute Q from k (multiplication)
  - But extremely hard to find k given only P and Q — this is the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**
- ECC exploits this hard problem to make **public-key cryptography**:
  - Private key $= k$
  - Public key $= Q = k \cdot P$
  - Only someone who knows k can reverse operations
- Result: strong security with much smaller keys than RSA

Why Bother with ECC?

- ECC offers **equivalent security** to RSA with much smaller keys
- Example comparison (approximate security equivalence):
  - 2048-bit RSA $\approx$ 224-bit ECC
  - 3072-bit RSA $\approx$ 256-bit ECC
  - 4096-bit RSA $\approx$ 384-bit ECC
- Advantages:
  - Lower bandwidth
  - Faster computations
  - Less storage and energy consumption (ideal for mobile/IoT)

Elliptic Curve Discrete Logarithm Problem (ECDLP)

- Security of ECC relies on the **ECDLP**:

$$\text{Given } P, Q = k \cdot P, \text{ find } k$$

- P is a point on the curve, Q is a multiple of P, k is unknown scalar
- No known efficient classical algorithm to solve ECDLP for large curves
- Quantum risk: Shor's algorithm breaks ECC as well, similar to RSA

ECC Key Exchange

- **Elliptic Curve Diffie–Hellman (ECDH)** allows two parties to derive a shared secret
- Process:
  1. Agree on curve E and base point G
  2. Alice chooses secret a, computes $A = a \cdot G$
  3. Bob chooses secret b, computes $B = b \cdot G$
  4. Exchange A and B, compute shared secret:

$$S = a \cdot B = b \cdot A$$

- Shared secret can then derive symmetric session keys for fast encryption (AES)