CSE 127: Intro to Computer Security

WI24

Lecture 14 - Public-Key Cryptography

Slide Credit: Kirill Levchenko, Stefan Savage, Deian Stefan, Nadia Heninger, Earlence Fernandes, Imani Munyaka

Announcements

HW/PA4 & PA5 Released

Feedback Results (next time)

PA3 - grades released next Tuesday

Today

- MAC/HMAC
- Key Exchange
- Public Key Encryption
- Digital Signatures

MACs

- Validate message integrity based on shared secret
- MAC: Message Authentication Code
  - Keyed function using shared secret
  - Hard to compute function without knowing key

$$a=MAC_k(m)$$

Message Authentication Codes (MACs)

- MACs provide message *integrity* and *authenticity*
- $MAC_K(M)$ – use symmetric encryption to produce short sequence of bits that depends on both the message (M) and the key (K)
- MACs should be resistant to *existential forgery*: Eve should not be able to produce a valid MAC for a message M' without knowing K
- To provide confidentiality, authenticity, and integrity of a message, Alice sends – $[E_K(M), MAC_K(E_K(M))]$ where $E_K(X)$ is encryption of X using key K
- Proves that M was encrypted (confidentiality and integrity) by someone who knew K (authenticity)

* In practice, you want to use different keys for E and MAC

Message Authentication Codes (MACs)

- MACs provide message *integrity* and *authenticity*
- $MAC_K(M)$ – use symmetric encryption to produce short sequence of bits that depends on both the message (M) and the key (K)
- MACs should be resistant to *existential forgery*: Eve should not be able to produce a valid MAC for a message M' without knowing K
- To provide confidentiality, authenticity, and integrity of a message, Alice sends – $[E_K(M), MAC_K(E_K(M))]$ where $E_K(X)$ is encryption of X using key K
- Proves that M was encrypted (confidentiality and integrity) by someone who knew K (authenticity)

* In practice, you want to use different keys for E and MAC

A Better MAC

- Objectives
  - Use available hash functions without modification
  - Easily replace embedded hash function as more secure ones are found
  - Preserve original performance of hash function
  - Easy to use

HMAC construction

- HMAC: MAC based on hash function

$$MAC_k(m) = H(k \oplus opad \,\|\, H(k \oplus ipad \,\|\, m))$$

  - HMAC-SHA256: HMAC construction using SHA-256

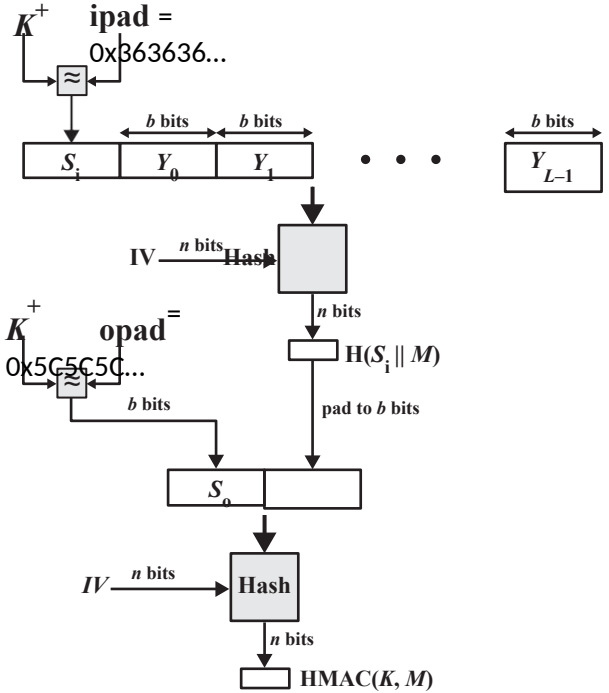
HMAC

- HMAC(k,M) =

  H(k⊕opad || H(k⊕ipad || M))

  – Attacker cannot extend MAC as before

  – Prove it to yourself



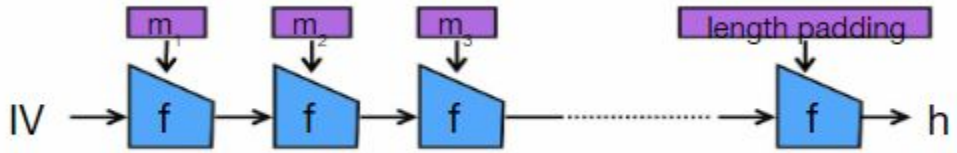
(from Stallings, Crypto and Net Securit_y_)

Creating a MAC from a Hash

- Consider the following simple hash-based MAC
  - $MAC_k(M) = h(k|M)$
- Suppose Eve wanted to append M' to M?
  - Goal: compute h(k|M|M') without knowing k
- Solution: Use h(k|M) as IV for next f iteration in h()
- Known as a Message Extension Attack

Length extension attack

- Merkle-Damgård construction: hash function from collision-resistant compression function f



- Attacker that can observe $MAC_k(m)$ can forge $MAC_k(m||padding||r)$ for an r of their choice

Other MAC constructions

- In 2009, Flickr required API calls to use authentication token that looked like:

  MD5(secret || arg1=val1&arg2=val2&...)

- Is $MAC_k(m) = H(k \,||\, m)$ a secure MAC?
  - ➤ No! If H is MD5, SHA1 or SHA2

Other MAC constructions

• In 2009, Flickr required API calls to use authentication token that looked like:

MD5(secret || arg1=val1&arg2=val2&...)

• Is $MAC_k(m) = H(k || m)$ a secure MAC?

- ➤ No! If H is MD5, SHA1 or SHA2
- ➤ Use HMAC!

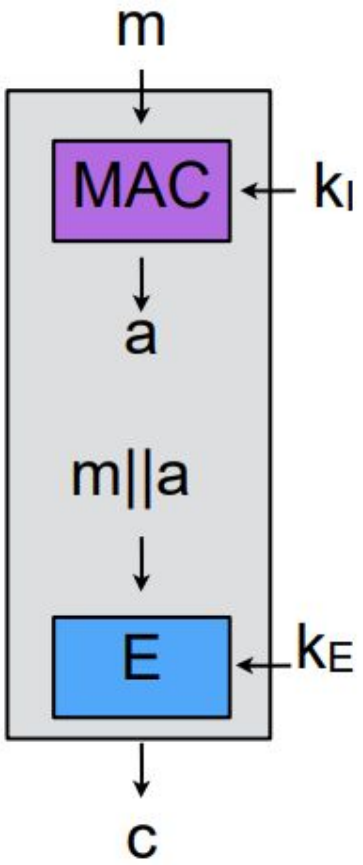
Combining Crypto has Implications

- Traditionally three ways of combining encryption and MAC as discrete operations:
  - Encrypt-then-MAC (EtM) [used in IPsec] {best}
  - Encrypt-and-MAC (E&M) [used in SSH] {bad}
  - MAC-then-Encrypt (MtE) [used in SSL/TLS] {padding attacks}
- All three are used in real protocols, but E&M and MtE require protocol modifications to be strongly unforgeable
- For more information, see:
  - https://en.wikipedia.org/wiki/Authenticated_encryption
  - https://eprint.iacr.org/2014/206 (EuroCrypt'14), starts the discussion that all three have problems in their generic composition

Combining MAC with encryption

MAC then Encrypt (SSL)

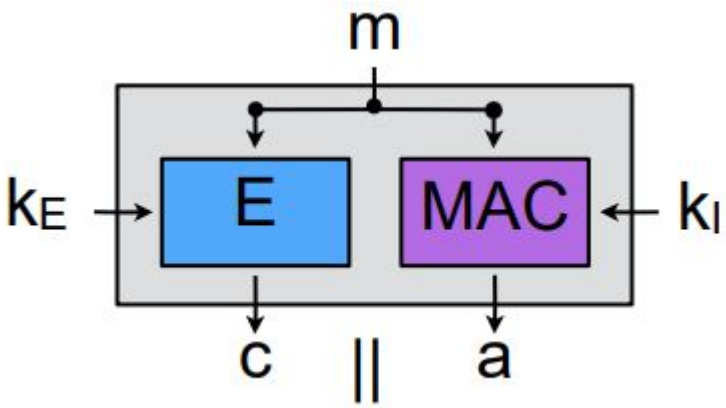
- Integrity for plaintext not ciphertext
- Issue: need to decrypt before you can verify integrity
- Hard to get right!



Combining MAC with encryption

Encrypt and MAC (SSH)

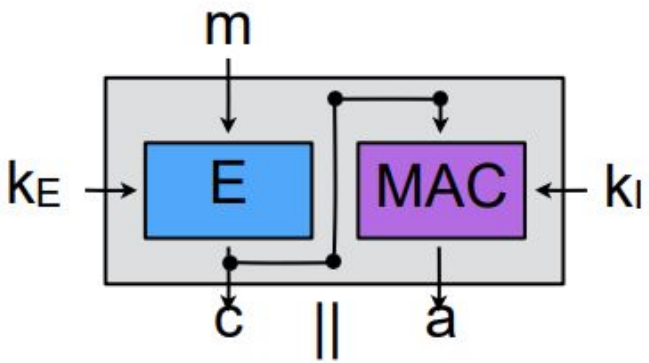
- Integrity for plaintext not ciphertext
- Issue: need to decrypt before you can verify integrity
- Hard to get right!



Combining MAC with encryption

Encrypt then MAC (IPSec)

- Integrity for plaintext and ciphertext
- Almost always right!



Authenticated Encryption (AE)

- Several modes of operation provide both encryption and authentication at the same time!
- Popular modes include
  - CCM (Counter with CBC-MAC)
  - GCM (Galois/Counter Mode)

AEAD construction

- Authenticated Encryption with Associated Data
  - AES-GCM, AES-GCM-SIV
- Always use an authenticated encryption mode
  - Combines mode of operation with integrity protection/ MAC in the right way

Galois/Counter Mode (GCM)

- Message is encrypted in a variant of CTR mode
- MAC comes from multiplication of ciphertext and key (H) in GF(2^{128}), in a chain similar to CBC

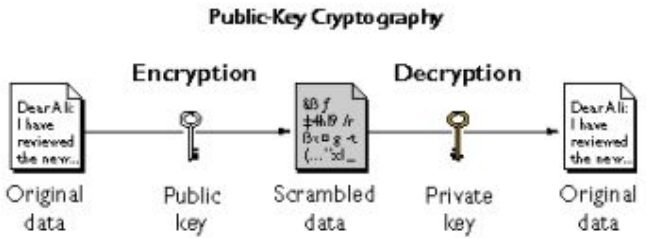
(https://en.wikipedia.org/wiki/Galois/Counter_Mode)

Asymmetric cryptography/public-key cryptography

Main insight: Separate keys for different operations.

Keys come in pairs, and are related to each other by the specific algorithm

- Public key: used to encrypt or verify signatures
- Private key: used to decrypt and sign



Public-key encryption

- Encryption: (public key, plaintext) → ciphertext

$$Enc_{pk}(m) = c$$

- Decryption: (secret key, ciphertext) → plaintext

$$Dec_{sk}(c) = m$$

Public Key - everyone has access to it
Private/Secret Key - only the owner of the key should have access to it

Public-key encryption

- Encryption: (public key, plaintext) → ciphertext

$$Enc_{pk}(m) = c$$

- Decryption: (secret key, ciphertext) → plaintext

$$Dec_{sk}(c) = m$$

Properties:

- Encryption and decryption are inverse operations:

Public-key encryption

- Encryption: (public key, plaintext) → ciphertext

$$Enc_{pk}(m) = c$$

- Decryption: (secret key, ciphertext) → plaintext

$$Dec_{sk}(c) = m$$

- Properties:
  - Encryption and decryption are inverse operations:

$$Dec_{sk}\ (Enc_{pk}(m)) = m$$

  - Secrecy: ciphertext reveals nothing about plaintext
    Computationally hard to decrypt without secret key
  - The point:
    Anybody with your public key can send you a secret message!
    Solves key distribution problem.

Modular Arithmetic

Modular Arithmetic

MATH MONKS

If A and B are two integers, and A is divided by B, then the relationship A = B x Q + R is written in modular arithmetic as

$$A \pmod{B} = R$$

here,

A = Dividend

B = Divisor

Q = Quotient

R = Remainder

Example

14 ÷ 3 = 4, remainder 2 ⇒ 14 (mod 3) = 2

(Webclicker)

13 mod 5 = X - what is X?

A. 1
B. 2
C. 3
D. 4
E. 5

(Webclicker)

13 mod 5 = X - what is X?

A. 1
B. 2
C. 3
D. 4
E. 5

$$\frac{13}{5} = 2 \text{ remainder } \mathbf{3}$$

$$13 \bmod 5 = \mathbf{3}$$

Modular Arithmetic Review

Division: Let n, d, q, r be integers.

$$\lfloor n/d \rfloor = q$$

$$n = qd + r \qquad 0 \le r < d$$

$$n \equiv r \bmod d$$

Facts about remainders/modular arithmetic:

Add: $(a \bmod d) + (b \bmod d) \equiv (a + b) \bmod d$

Subtract: $(a \bmod d) - (b \bmod d) \equiv (a - b) \bmod d$

Multiply: $(a \bmod d) \cdot (b \bmod d) \equiv (a \cdot b) \bmod d$

Modular Inverse: “Division” for modular arithmetic

If $a \cdot b \bmod d = c \bmod d$ we would like $c/b \bmod d = a \bmod d$.

Let’s try this: let $a = 3$, $b = 2$, and $d = 4$

Modular Inverse: “Division” for modular arithmetic

If $a \cdot b \bmod d = c \bmod d$ we would like $c/b \bmod d = a \bmod d$.

Let’s try this: let a = 3, b = 2, and d = 4

3*2 mod4 = c mod 4

3*2 mod 4 = 6 mod 4

3 mod 4 = 3 mod

This doesn’t quite work, it says 3 = 1 mod 4!

Modular Inverse: “Division” for modular arithmetic

If $a \cdot b \bmod d = c \bmod d$ we would like $c/b \bmod d = a \bmod d$.

Let’s try this: let a = 3, b = 2, and d = 4
This doesn’t quite work, it says 3 = 1 mod 4!

Fix: For rationals, $\frac{a}{b} = a \cdot \frac{1}{b}$ $\quad b \cdot \frac{1}{b} = 1.$

Define modular inverse: $\frac{1}{b}$ means $b^{-1} \bmod d$.

- $b^{-1} \bmod d$ is a value such that $b \cdot b^{-1} \equiv 1 \bmod d$.
- Example: $3 \cdot (3^{-1} \bmod 5) \equiv$

Modular Inverse: “Division” for modular arithmetic

If $a \cdot b \bmod d = c \bmod d$ we would like $c/b \bmod d = a \bmod d$.

Let’s try this: let $a = 3$, $b = 2$, and $d = 4$

This doesn’t quite work, it says $3 = 1 \bmod 4$!

Fix: For rationals, $\frac{a}{b} = a \cdot \frac{1}{b}$ $b \cdot \frac{1}{b} = 1$.

Define modular inverse: means $b^{-1} \bmod d$.

- $b^{-1} \bmod d$ is a value such that $b \cdot b^{-1} \equiv 1 \bmod d$.
- Example: $3 \cdot (3^{-1} \bmod 5) \equiv 3 \cdot 2 \equiv 1 \bmod 5$.
- If $\gcd(a, d) = 1$ then a^{-1} is well defined.
- Efficient to compute.

Modular Inverse: “Division” for modular arithmetic

The modular inverse of A mod C is the B value that makes A * B mod C = 1

A= 3, B=?, C = 7

3 * 0 ≡ 0 (mod 7)
3 * 1 ≡ 3 (mod 7)
3 * 2 ≡ 6 (mod 7)
3 * 3 ≡ 9 ≡ 2 (mod 7)
3 * 4 ≡ 12 ≡ 5 (mod 7)
3 * 5 ≡ 15 (mod 7) ≡ 1 (mod 7) <------ FOUND INVERSE!
3 * 6 ≡ 18 (mod 7) ≡ 4 (mod 7)

Modular Inverse: “Division” for modular arithmetic

The Algorithm

The Euclidean Algorithm for finding GCD(A,B) is as follows:

- If A = 0 then GCD(A,B)=B, since the GCD(0,B)=B, and we can stop.
- If B = 0 then GCD(A,B)=A, since the GCD(A,0)=A, and we can stop.
- Write A in quotient remainder form (A = B·Q + R)
- Find GCD(B,R) using the Euclidean Algorithm since GCD(A,B) = GCD(B,R)

Modular exponentiation and discrete log

Modular exponentiation

- Over the integers, $g^a = g \cdot g \cdot g \ldots g$
- $g^a \bmod d$ it's the same:
- $g^a \bmod d = (((g \bmod d) \cdot g \bmod d) \ldots g \bmod d) \bmod d$
- Efficient to compute using the binary representation of a.

Modular exponentiation and discrete log

Modular exponentiation

- Over the integers, $g^a = g \cdot g \cdot g \ldots g$
- $g^a \bmod d$ it's the same:
- $g^a \bmod d = (((g \bmod d) \cdot g \bmod d) \ldots g \bmod d) \bmod d$
- Efficient to compute using the binary representation of a.

"Inverse" of modular exponentiation: Discrete log

- Over the reals, if $b^a = y$ then $\log_b y = a$.

Modular exponentiation and discrete log

Modular exponentiation

- Over the integers, $g^a = g \cdot g \cdot g \ldots g$
- $g^a \bmod d$ it's the same:
- $g^a \bmod d = (((g \bmod d) \cdot g \bmod d) \ldots g \bmod d) \bmod d$
- Efficient to compute using the binary representation of a.

"Inverse" of modular exponentiation: Discrete log

- Over the reals, if $b^a = y$ then $\log_b y = a$.
- Define discrete log similarly:
  - Input b, d, y, discrete log is a such that $b^a \equiv y \bmod d$.

Modular exponentiation and discrete log

Modular exponentiation

- Over the integers, $g^a = g \cdot g \cdot g \ldots g$
- $g^a \bmod d$ it's the same:
- $g^a \bmod d = (((g \bmod d) \cdot g \bmod d) \ldots g \bmod d) \bmod d$
- Efficient to compute using the binary representation of a.

"Inverse" of modular exponentiation: Discrete log

- Over the reals, if $b^a = y$ then $\log_b y = a$.
- Define discrete log similarly:
  - Input b, d, y, discrete log is a such that $b^a \equiv y \bmod d$.
- No known polynomial-time algorithm to compute this.

New Directions in Cryptography

Invited Paper

WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

Diffie-Hellman Key Exchange

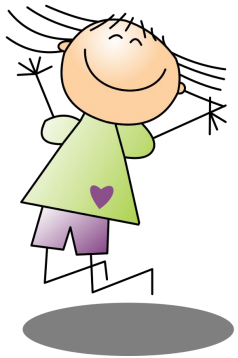
- Allows communicating parties with no prior knowledge to exchange shared secret keys over an insecure channel
- DH key exchange protocol allows exchange of a secret
- Protocol can be tweaked to turn into a public-key encryption scheme

Example:

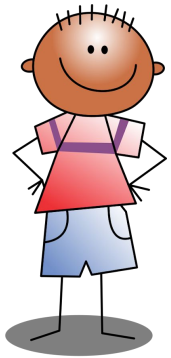
- Alice and Bob want to communicate
- Alice and Bob agree on:
    - Number p: big prime number (such as a 2048-bit number)
    - Generator g: small prime number (such as 2 and 3)
- Alice picks a random positive integer a < p
- Bob picks a random positive integer b < p

Symmetric cryptography

- Alice and Bob want to communicate

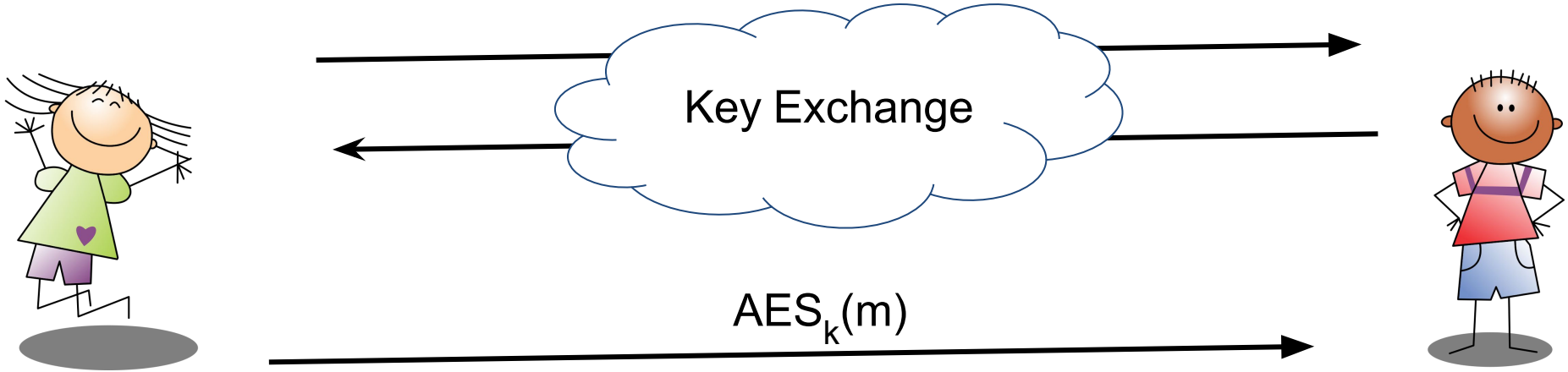


$AES_k(m)$



Public key crypto idea # 1: Key exchange

Solving key distribution without trusted third parties



k = shared secret

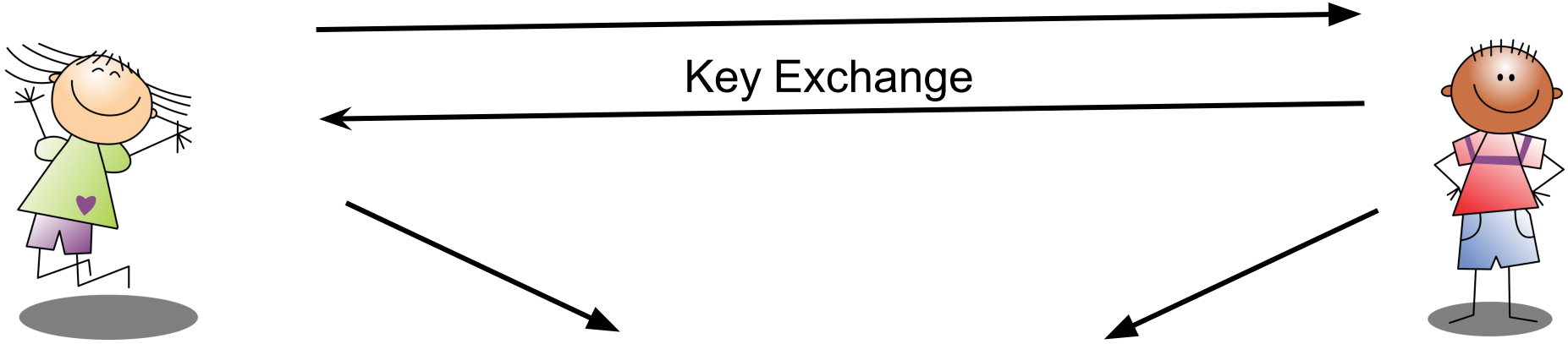
k = shared secret

Textbook Diffie-Hellman Key Exchange

Public Parameters

p a prime

g an integer mod$_p$



Alice and Bob agree on:

- Number p: big prime number (such as a 2048-bit number)
- Generator g: small prime number (such as 2 and 3)

Textbook Diffie-Hellman Key Exchange

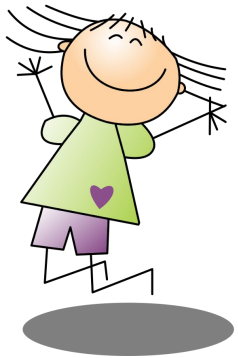
Public Parameters

p a prime

g an integer mod$_p$

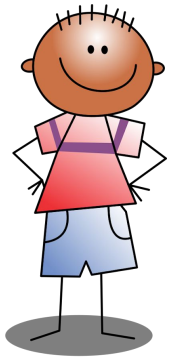
Alice picks a random positive integer a < p
Bob picks a random positive integer b< p

Key Exchange



$g^a \bmod p$

$g^b \bmod p$

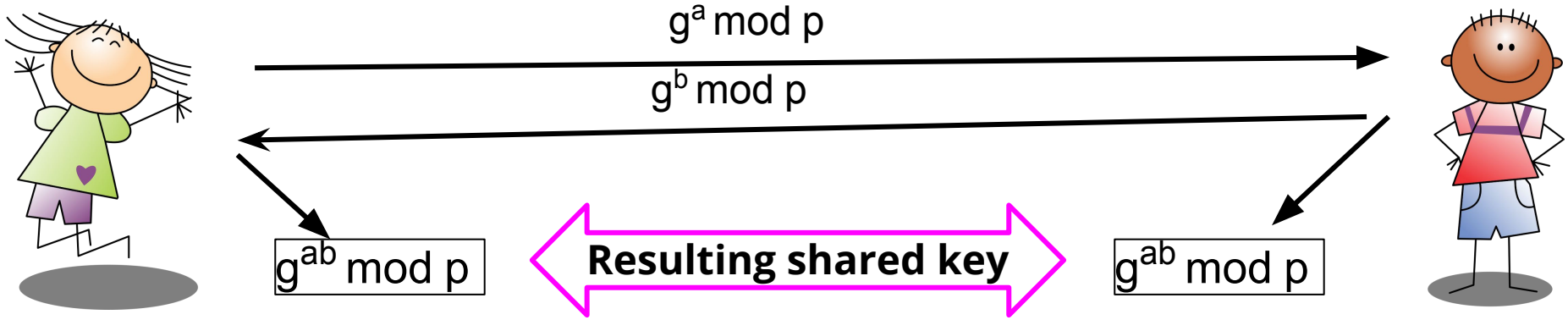


$g^{ab} \bmod p$

$g^{ab} \bmod p$

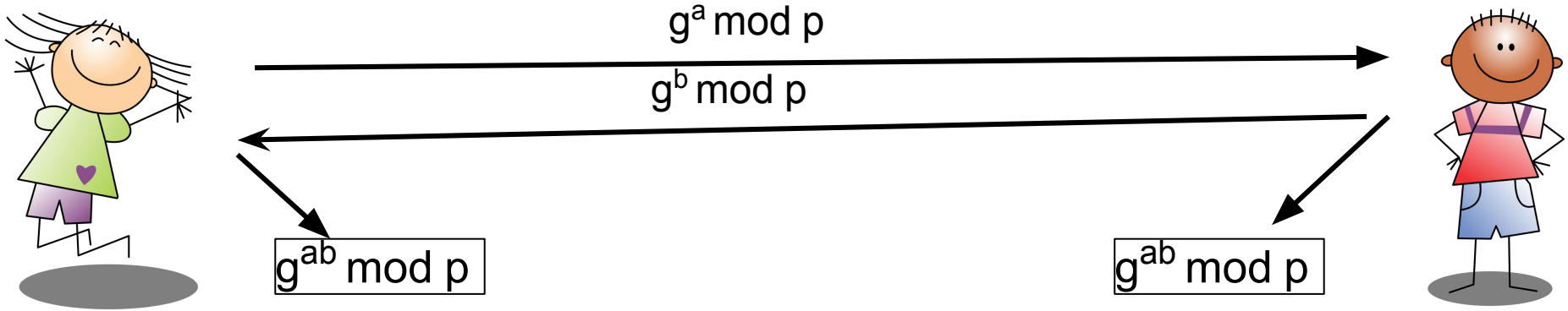
Note: $(g^a)^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p (g^b)^a \bmod p$.

Diffie-Hellman Security



► Most efficient algorithm for passive eavesdropper to break: Compute discrete log of public values $g^a \bmod p$ or $g^b \bmod p$.

Diffie-Hellman Security



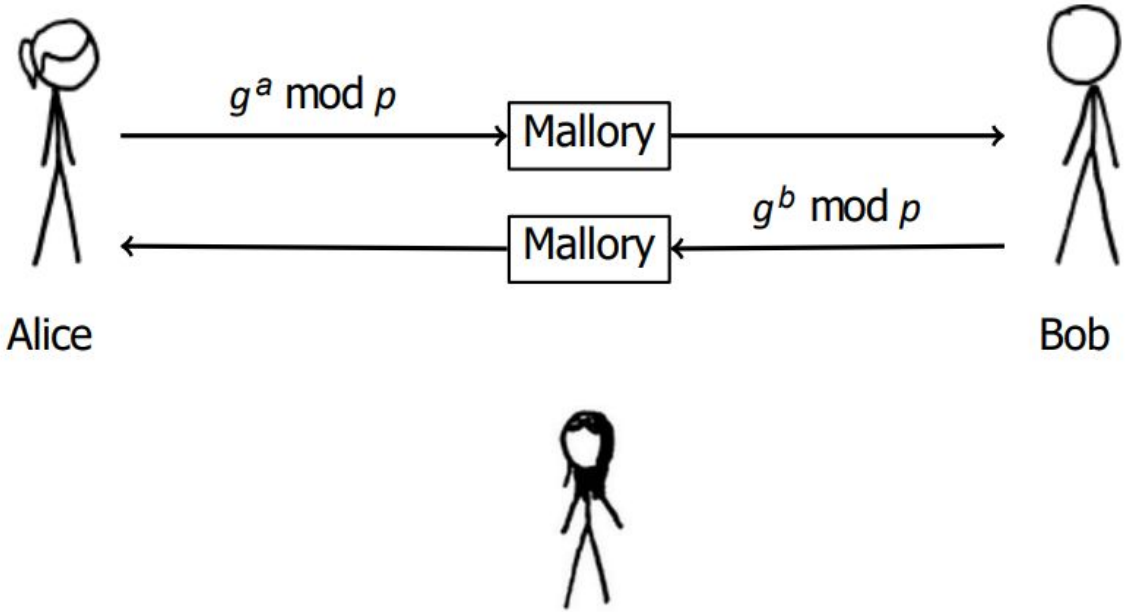
► Most efficient algorithm for passive eavesdropper to break: Compute discrete log of public values $g^a \bmod p$ or $g^b \bmod p$.

► Parameter selection: p should be ≥ 2048 bits

► Do not implement this yourself ever: discrete log is only hard for certain choices of p and g.

► Best current choice: Use elliptic curve Diffie-Hellman. (Similar idea, more complicated math.)

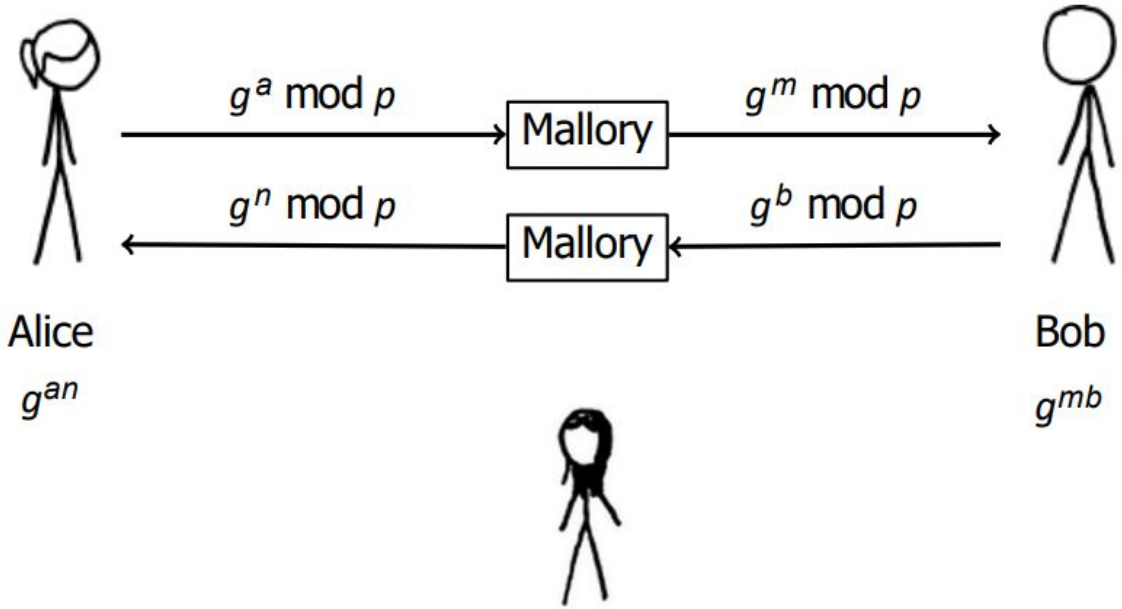
Diffie-Hellman insecure against man-in-the-middle



Active adversary can modify Diffie-Hellman messages in transit and learn both shared secrets.

Allows transparent MITM attack against later encryption.

Diffie-Hellman insecure against man-in-the-middle



Active adversary can modify Diffie-Hellman messages in transit and learn both shared secrets.

Allows transparent MITM attack against later encryption.

Fix: Need to authenticate messages.

Computational complexity for integer problems

► Integer multiplication is efficient to compute.

► There is no known polynomial-time algorithm for general-purpose factoring.

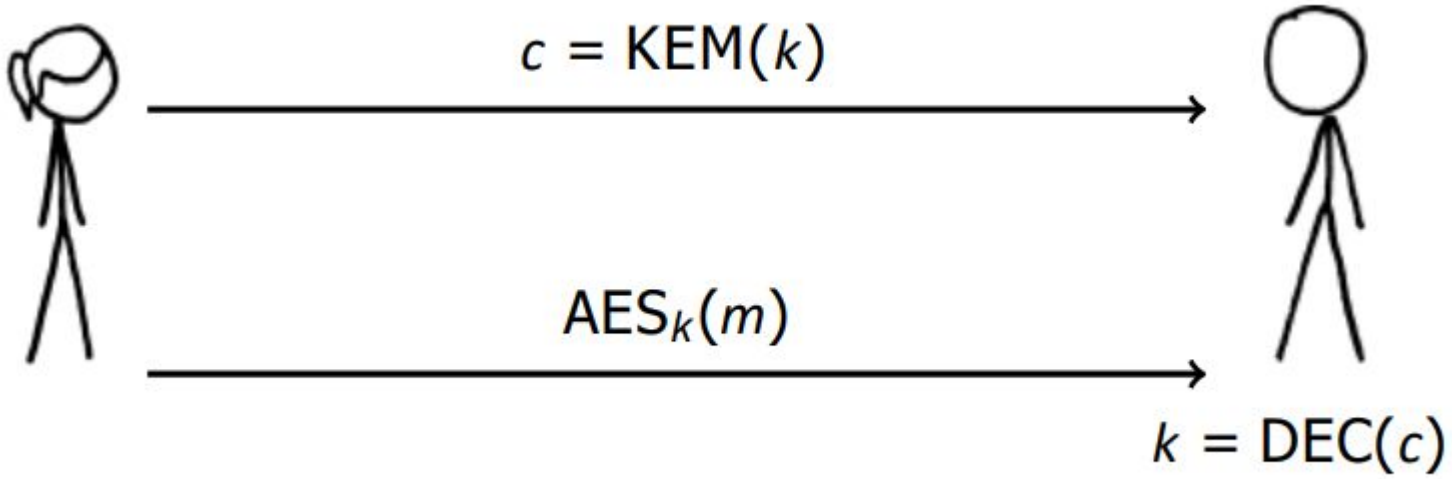
► Efficient factoring algorithms for many types of integers. Easy to find small factors of random integers.

► Modular exponentiation is efficient to compute.

► Modular inverses are efficient to compute.

Idea # 2: Key encapsulation/public-key encryption

Solving key distribution without trusted third parties



Diffie Hellman & Modular Math

A= value Alice sends to Bob
B= value Bob sends to Alice
g= non-zero integer agreed upon by the group
a= Alice secret
b= bob secret
p= large prime agreed upon by the group

$$\underbrace{A \equiv g^a \pmod{p}}_{\text{Alice computes this}} \quad \text{and} \quad \underbrace{B \equiv g^b \pmod{p}}_{\text{Bob computes this}}.$$

Diffie Hellman & Modular Math

$$\underbrace{A \equiv g^a \pmod{p}}_{\text{Alice computes this}} \quad \text{and} \quad \underbrace{B \equiv g^b \pmod{p}}_{\text{Bob computes this}}.$$

$$\underbrace{A' \equiv B^a \pmod{p}}_{\text{Alice computes this}} \quad \text{and} \quad \underbrace{B' \equiv A^b \pmod{p}}_{\text{Bob computes this}}.$$

Example 2.7 from textbook (?)

Alice and Bob agree to use the prime p = 941 and the primitive root g = 627.

Alice chooses the secret key a = 347 and computes $A = 390 \equiv 627^{347} \pmod{941}$.

Similarly, Bob chooses the secret key b = 781 and computes $B = 691 \equiv 627^{781} \pmod{941}$.

Alice sends Bob the number 390 and Bob sends Alice the number 691. Both of these transmissions are done over an insecure channel, so both A = 390 and B = 691 should be considered public knowledge.

The numbers a = 347 and b = 781 are not transmitted and remain secret. Then Alice and Bob are both able to compute the number .

$470 \equiv 627^{347 \cdot 781} \equiv A^{b} \equiv B^{a} \pmod{941}$, so 470 is their shared secret

Practice

Using the Diffie-Hellman Key Exchange, find the shared key between Kim and John if the prime, P, is 23 and primitive modulo, g, is 5. Note that Kim's secret key, a, is 4 and John's secret key, b, is 3.

Group Exercise

Using the Diffie-Hellman Key Exchange, find the shared key between Alice and Bob if the prime, P, is 11 and primitive modulo, g, is 2. Note that Alice's secret key, a, is 4 and Bob's secret key, b, is 5.

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

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

Textbook RSA Encryption

[Rivest Shamir Adleman 1977]

Public Key pk

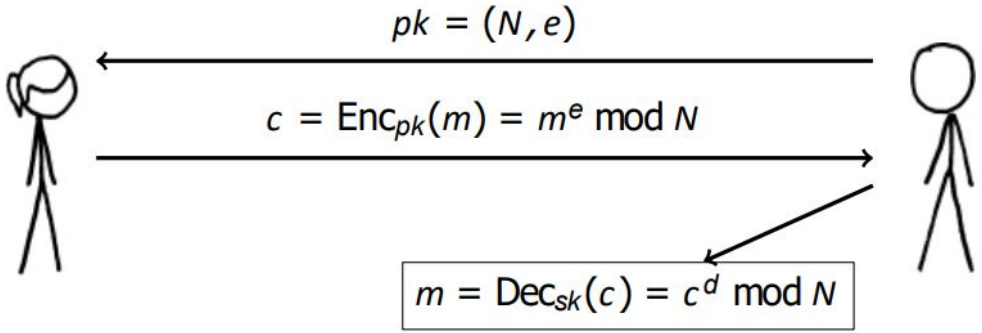
N = pq modulus

e encryption exponent

Secret Key sk

p, q primes

d decryption exponent



Dec(Enc(m)) $= m^{ed} \bmod N \equiv m^{1+k\varphi(N)} \equiv m \bmod N$ by Euler's theorem ($m^{\varphi(N)} \equiv 1 \bmod N$).

Textbook RSA Encryption

[Rivest Shamir Adleman 1977]

In this case Bob and Alice have separate pks

Public Key pk

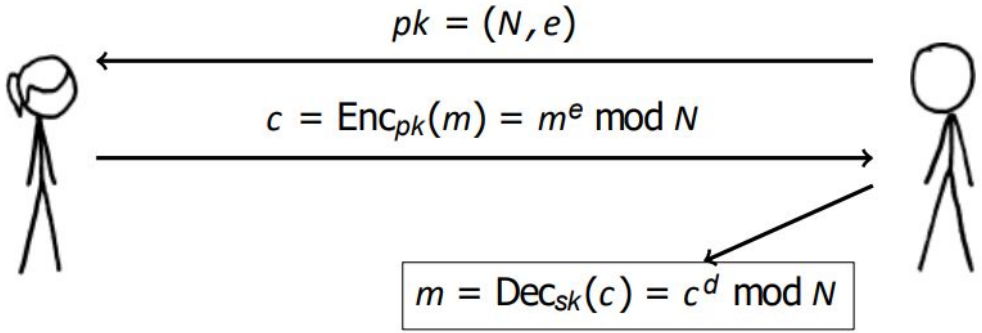
N = pq modulus

e encryption exponent

Secret Key sk

p, q primes

d decryption exponent



Dec(Enc(m)) $= m^{ed} \bmod N \equiv m^{1+k\varphi(N)} \equiv m \bmod N$ by Euler's theorem ($m^{\varphi(N)} \equiv 1 \bmod N$).

RSA Security

- Best algorithm to break RSA: Factor N and compute d.
- Factoring is not efficient in general.
- Current key size recommendations: N should be ≥ 2048 bits.
- Do not ever implement this yourself. Factoring is only hard for some integers, and textbook RSA is insecure.

Textbook RSA is super insecure

Unpadded RSA encryption is homomorphic under multiplication.

Textbook RSA is super insecure

Unpadded RSA encryption is **homomorphic** under multiplication.

Homomorphic encryption systems allow data to be analyzed and processed on a ciphertext rather than the underlying data itself (you can do computations w/ the ciphertext without needing to decrypt first). Also, encrypted data can be accessed but never decrypted (might be seen but hard to understand). (IEEE)

Textbook RSA is super insecure

Unpadded RSA encryption is homomorphic under multiplication.

Attack: Malleability

capability of being shaped or formed by alternate forces

Given a ciphertext $c = Enc(m) = m^e \bmod N$, attacker can forge ciphertext $Enc(ma) = ca^e \bmod N$ for any a.

Basically attackers message encrypted time the ciphertext

A is attacker controlled

Textbook RSA is super insecure

Unpadded RSA encryption is homomorphic under multiplication.

Attack: Malleability

Given a ciphertext $c = \text{Enc}(m) = m^e \bmod N$, attacker can forge ciphertext $\text{Enc}(ma) = ca^e \bmod N$ for any a.

Attack: Chosen ciphertext attack

Given a ciphertext $c = \text{Enc}(m)$ for unknown m, attacker asks for $\text{Dec}(ca^e \bmod N) = d$ and computes $m = da^{-1} \bmod N$.

Textbook RSA is super insecure

Unpadded RSA encryption is homomorphic under multiplication.

Attack: Malleability

Given a ciphertext $c = \text{Enc}(m) = m^e \bmod N$, attacker can forge ciphertext $\text{Enc}(ma) = ca^e \bmod N$ for any a.

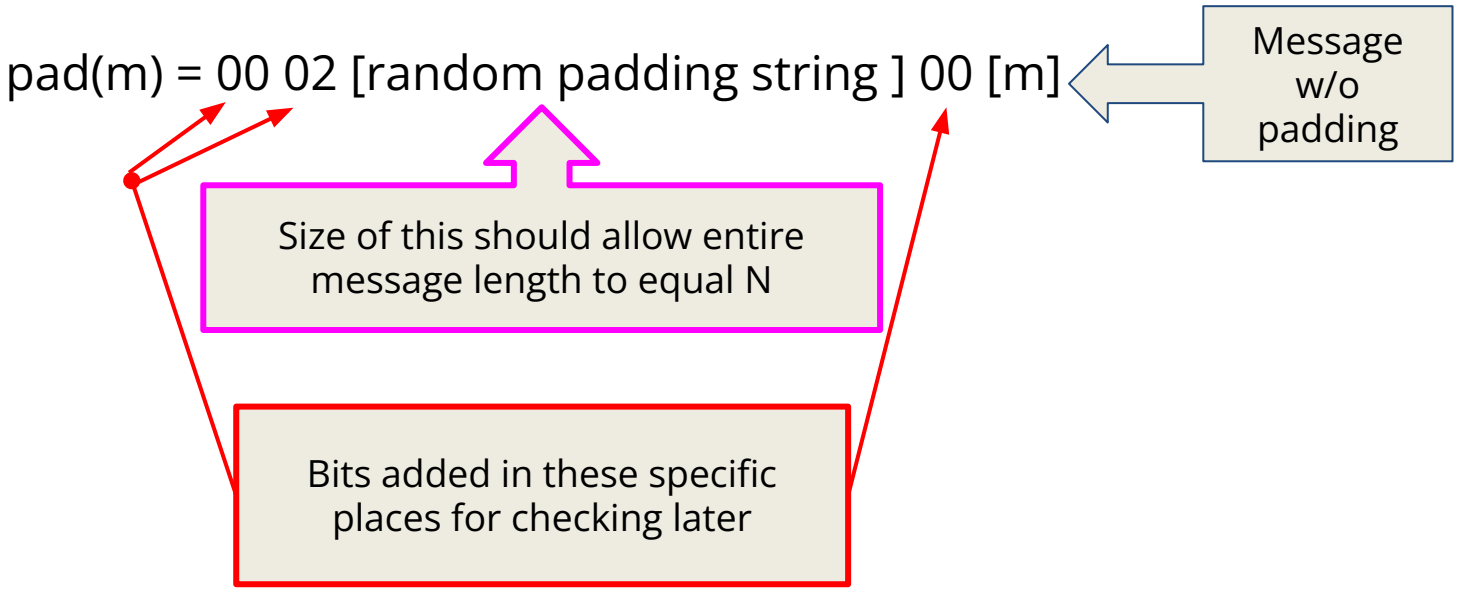
Attack: Chosen ciphertext attack

Given a ciphertext $c = \text{Enc}(m)$ for unknown m, attacker asks for $\text{Dec}(ca^e \bmod N) = d$ and computes $m = da^{-1} \bmod N$.

Fix: always use padding on messages.

RSA PKCS #1 v1.5 padding

Most common implementation choice even though it is insecure



RSA PKCS #1 v1.5 padding

Most common implementation choice even though it is insecure

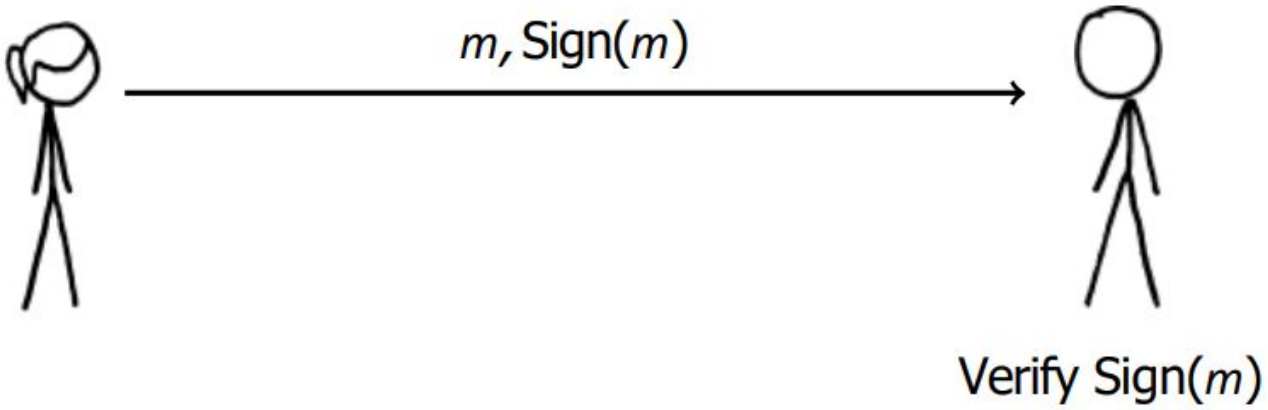
pad(m) = 00 02 [random padding string] 00 [m]

► Encrypter pads message, then encrypts padded message using RSA public key: $Enc_{pk}(m) = pad(m)^e \bmod N$

► Decrypter decrypts using RSA private key, strips off padding to recover original data: $Dec_{sk}(c) = c^d \bmod N = pad(m)$

PKCS#1v1.5 padding is vulnerable to a number of padding attacks. It is still commonly used in practice

Idea #3: Digital Signatures



Bob wants to verify Alice's signature using only a public key.

- Signature verifies that Alice was the only one who could have sent this message.
- Signature also verifies that the message hasn't been modified in transit.

Digital Signatures

- Signing: (secret key, message) → signature

$$\text{Sign}_{sk}(m) = s$$

- Verification: (public key, message, signature) → bool

$$\text{Verify}_{pk}(m, s) = \text{true} \mid \text{false}$$

Signature properties:

- Verification of signed message succeeds:

Digital Signatures

- Signing: (secret key, message) → signature

$$\text{Sign}_{sk}(m) = s$$

- Verification: (public key, message, signature) → bool

$$\text{Verify}_{pk}(m, s) = \text{true} \mid \text{false}$$

Signature properties:

- Verification of signed message succeeds:
  - $\text{Verify}_{pk}(m, \text{Sign}_{sk}(m)) = \text{true}$
- Unforgeability: Can't compute signature for message m that verifies with public key without corresponding secret key.
- The point:
  - Anybody with your public key can verify that you signed something!

Textbook RSA Signatures

[Rivest Shamir Adleman 1977]

Public Key pk

N = pq modulus

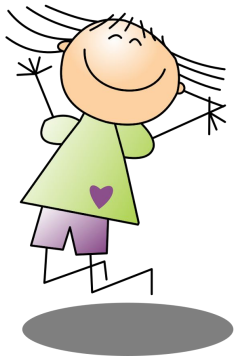
e public exponent

Secret Key sk

p, q primes

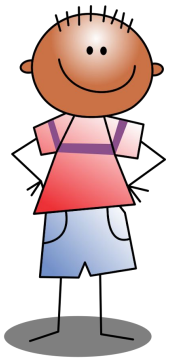
d private exponent

$(d = e^{-1} \bmod (p-1)(q-1))$



pk = (N, e)

m, s = Sign(m) = $m^d \bmod N$



Verify(m, s): $m = s^e \bmod N$

Works for the same reason RSA encryption does.

Textbook RSA signatures are super insecure

Attack: Signature forgery

1. Attacker wants Sign(x).
2. Attacker computes $z = xy^e \bmod N$ for some y.
3. Attacker asks signer for $s = \text{Sign}(z) = z^d \bmod N$.
4. Attacker computes $\text{Sign}(x) = sy^{-1} \bmod N$.

Countermeasures:

- Always use padding with RSA.
- Sign hash of m and not raw message m.

Positive viewpoint:

- Blind signatures: Lots of neat crypto applications.

RSA PKCS #1 v1.5 signature padding

Most widely used padding scheme in practice

```
pad(m) = 00 01 [FF FF FF ... FF FF] 00 [data H(m)]
```

- Signer hashes and pads message, then signs padded message using RSA private key.
- Verifier verifies using RSA public key, strips off padding to recover hash of message.

Q: What happens if a decrypter doesn't correctly check padding length?

A: Bleichenbacher low exponent signature forgery attack.

https://www.youtube.com/watch?v=2xspZfXI_nY

Bleichenbacher RSA Signature Forgery

pad(m) = 00 01 [FF FF FF ... FF FF] 00 [data H(m)]

If victim shortcuts padding check: just looks for padding format but doesn't check length, and signature uses e = 3:

1. Construct a perfect cube over the integers, ignoring N, such that

   x = 0001FF . . . FF00[hash of forged message][garbage]

2. Compute s such that $s^3 = x$.

   (Easy way: set garbage to zero and take cube root, i.e., $s = rxl^{1/3}$.)

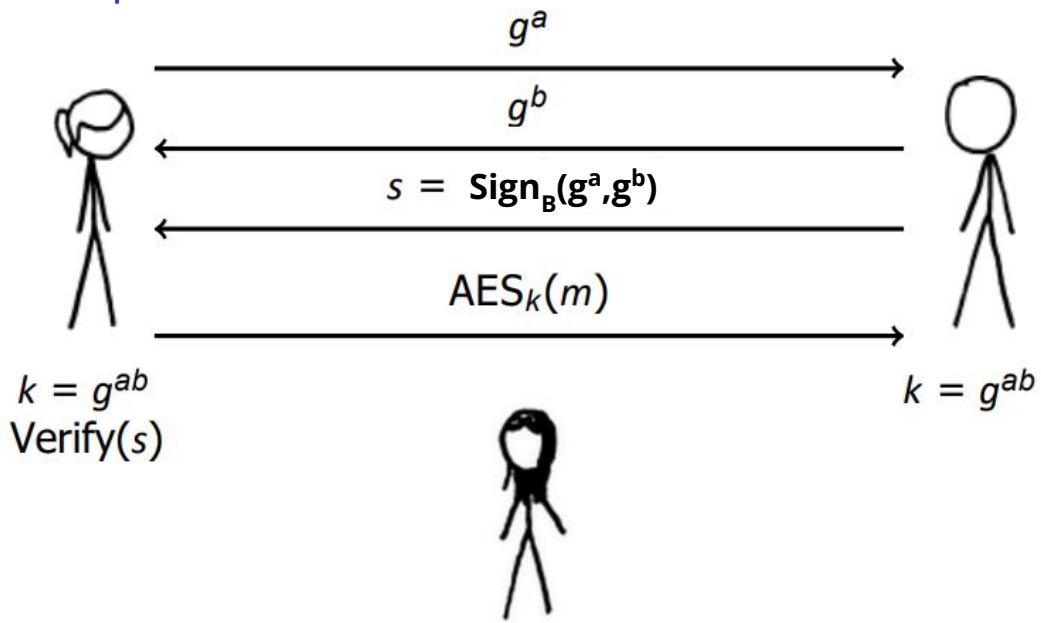
3. Lazy implementation validates bad signature!

Security for RSA signatures

- Same as RSA encryption.
- Recommendation: Use ECDSA or ed25519 instead.

Putting it all together

How public-key cryptography is used in practice



- Diffie-Hellman used to negotiate shared session key.
- Alice verifies Bob's signature to ensure that key exchange was not man-in-the-middled.
- Shared secret used to symmetrically encrypt data.

Public-key cryptography and quantum computers

Right now, all public-key cryptography used in the real world involves three “hard” problems:

- Factoring
- Discrete log mod primes
- Elliptic curve discrete log

All of these problems can be solved efficiently by a general-purpose quantum computer.

Big standardization effort now to develop replacements:

- Lattice-based cryptography
- Multivariate cryptography
- Hash-based signatures
- Supersingular isogeny Diffie-Hellman

These will likely be used more in the real world in the next few

Reminder: Cryptographic primitives

	Symmetric crypto	Public-key crypto
Confidentiality	Symmetric encryption (e.g. AES)	Public-key encryption (e.g. RSA)
Integrity/Authenticity	MACs	Digital signatures

Reminder: Network Attacker Threat Model

Network Attacker:

- Controls infrastructure: Routers, DNS
- Eavesdrops, injects, drops, or modifies packets

Examples:

- Wifi at internet cafe
- Internet access at hotels

Goal: Establish a secure channel to a host that ensures

- Confidentiality and Integrity of messages
- Authentication of the remote host

SSL / TLS

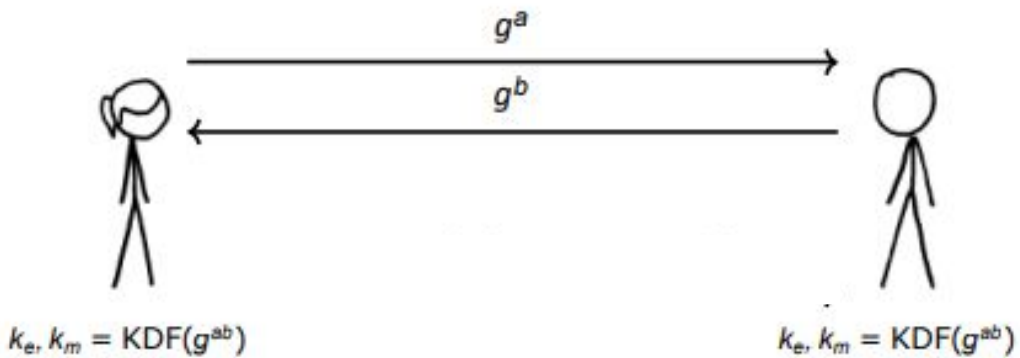
- Secure Socket Layer (SSL)
  - v2 Developed by Netscape Navigator in 1995
  - v3 released in 1996
- Transport Layer Security (TLS)
  - Released as RFC in 1999
  - Attempt to standardize the protocol
- Basic idea: A program can replace socket creation with a “secure socket” to get authentication, confidentiality and integrity
- HTTPS = HTTP + SSL/TLS

Common cryptographic network protocols

- **TLS(Transport Layer Security)**
  - Used to provide an encryption wrapper around HTTP to make HTTPS, and for many other application layer protocols.
  - Security goals: Authenticate server, confidentiality and integrity of traffic
- **SSH (Secure Shell)**
  - Use to access remote machines
  - Security goals: Authenticate server and client, confidentiality and integrity of traffic
- **IPsec (Internet Protocol Security)**
  - Provides an encrypted, authenticated alternative to IP
  - Commonly used for VPNs (Virtual Private Networks)
  - Security goals: client and server authentication, authenticate headers, optionally encrypt headers, ensure confidentiality and integrity of payloads

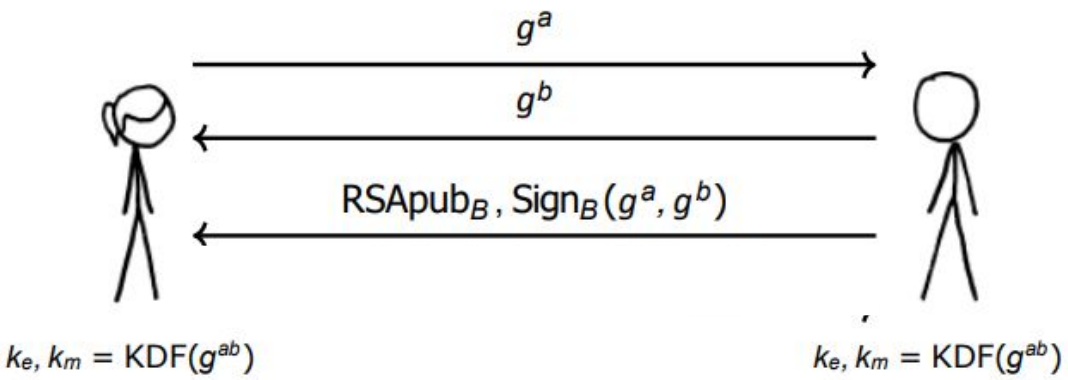
Constructing a secure encrypted channel

- To negotiate shared symmetric keys: Diffie-Hellman key exchange. Key Derivation Function (KDF) maps shared secret to symmetric key.



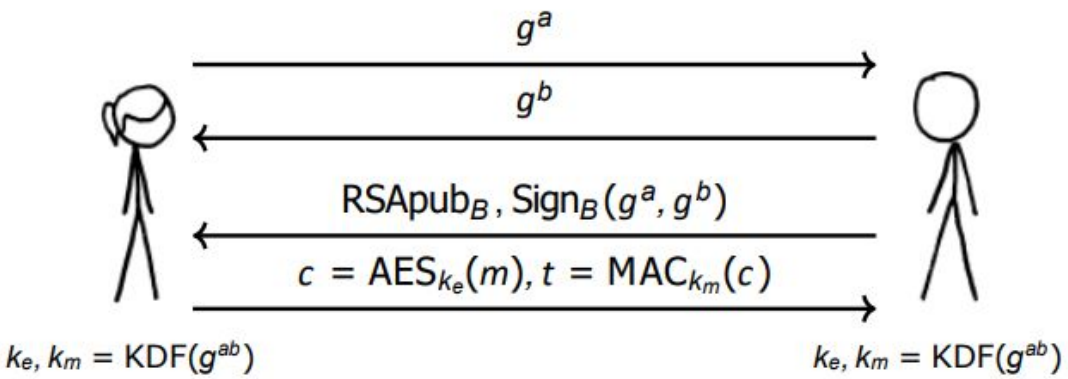
Constructing a secure encrypted channel

• To negotiate shared symmetric keys: Diffie-Hellman key exchange. Key Derivation Function (KDF) maps shared secret to symmetric key.
• To ensure authenticity of endpoints: Digital Signatures



Constructing a secure encrypted channel

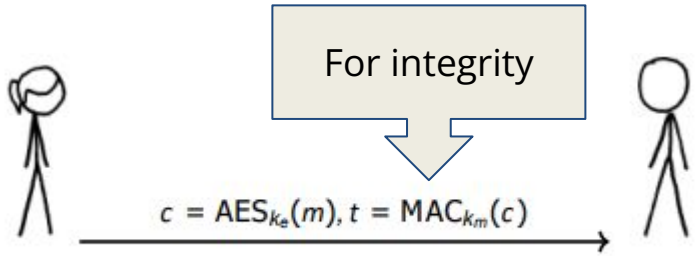
• To ensure confidentiality and integrity: Encrypt and MAC data
• To negotiate shared symmetric keys: Diffie-Hellman key exchange. Key Derivation Function (KDF) maps shared secret to symmetric key.
• To ensure authenticity of endpoints: Digital Signatures



Constructing a secure encrypted channel

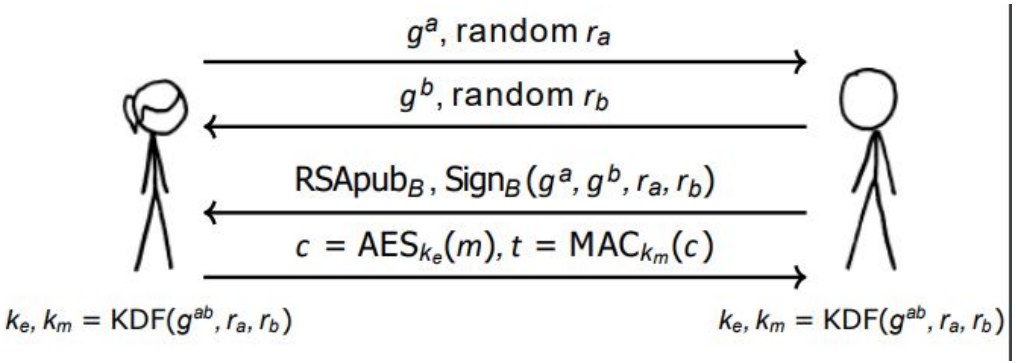
- To ensure confidentiality and integrity: Encrypt and MAC data

Why? Consider the issues of ONLY using AES with a shared key.



Constructing a secure encrypted channel

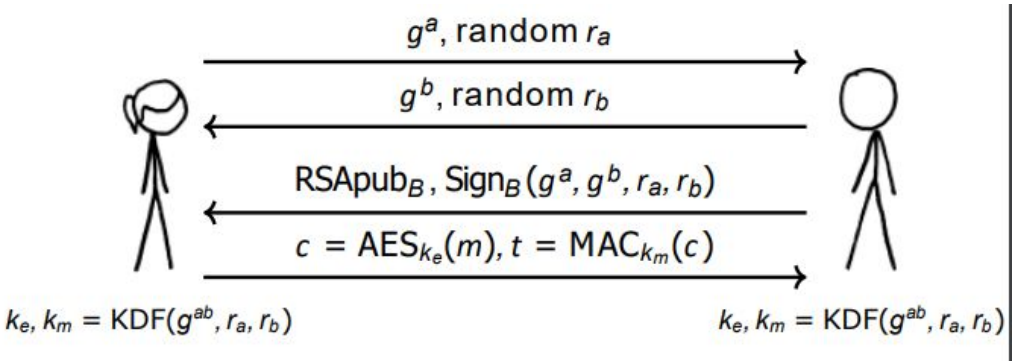
- To ensure confidentiality and integrity: Encrypt and MAC data
- To negotiate shared symmetric keys: Diffie-Hellman key exchange. Key Derivation Function (KDF) maps shared secret to symmetric key.
- To ensure authenticity of endpoints: Digital Signatures
- To ensure an adversary can't reuse a signature later, add some random unique values ("nonces")



This is not exactly what TLS looks like, but it's similar.

Constructing a secure encrypted channel

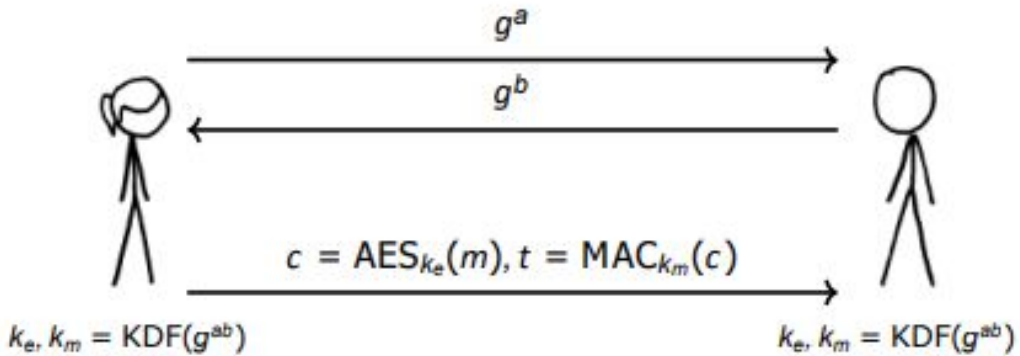
- To ensure confidentiality and integrity: Encrypt and MAC data
- To negotiate shared symmetric keys: Diffie-Hellman key exchange. Key Derivation Function (KDF) maps shared secret to symmetric key.
- To ensure authenticity of endpoints: Digital Signatures
- To ensure an adversary can't reuse a signature later, add some random unique values ("nonces")



How does Alice know to trust Bob's public signing key?

Constructing a secure encrypted channel

- To ensure confidentiality and integrity: Encrypt and MAC data
- To negotiate shared symmetric keys: Diffie-Hellman key exchange. Key Derivation Function (KDF) maps shared secret to symmetric key.



Public Key Infrastructure: Establishing Trust in Keys

Ways to establish trust in keys:

- Meet in person to exchange keys.
    - Not practical at scale over the internet

Public Key Infrastructure: Establishing Trust in Keys

Ways to establish trust in keys:

- **Fingerprint verification**
  - Verify a cryptographic hash of a public key through a separate channel, or "**trust on first use"** (TOFU).
  - This is used by SSH for host keys.

```
$ ssh elk.sysnet.ucsd.edu
The authenticity of host 'elk.sysnet.ucsd.edu (137.110.222.162)' can't be
established.
ED25519 key fingerprint is SHA256:rI/PqZezDo18EbK8U8/HXesuO7iCoNUGa+8r3t3qGxw.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Public Key Infrastructure: Establishing Trust in Keys

Ways to establish trust in keys:

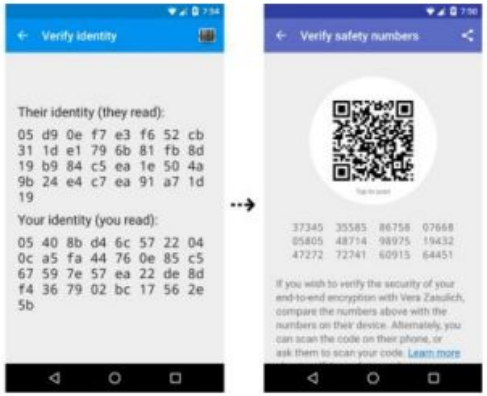
- **Fingerprint verification**
  - Verify a cryptographic hash of a public key through a separate channel, or "**trust on first use"** (TOFU).
  - This is used by SSH for host keys.

Trust On First Use (TOFU) is a security model in which a client needs to create a trust relationship with an unknown server. To do that, clients will look for identifiers (for example public keys) stored locally. If an identifier is found, the client can establish the connection. If no identifier is found, the client can prompt the user to determine if the client should trust the identifier. (Mozilla)

Public Key Infrastructure: Establishing Trust in Keys

Ways to establish trust in keys:

- **Fingerprint verification**
  - Verify a cryptographic hash of a public key through a separate channel, or "**trust on first use"** (TOFU).
  - This is used by SSH for host keys.
  - This is also used by encrypted messaging apps like Signal



Public Key Infrastructure: Establishing Trust in Keys

Ways to establish trust in keys:

- Certificate Authorities
  - A certificate authority (CA) is a kind of commercial trusted intermediary.
  - Certificate Authorities verify public keys and sign them in exchange for money.
  - If you trust the certificate authority, you transitively trust the keys it signs.
  - This is used for TLS, software signing keys.

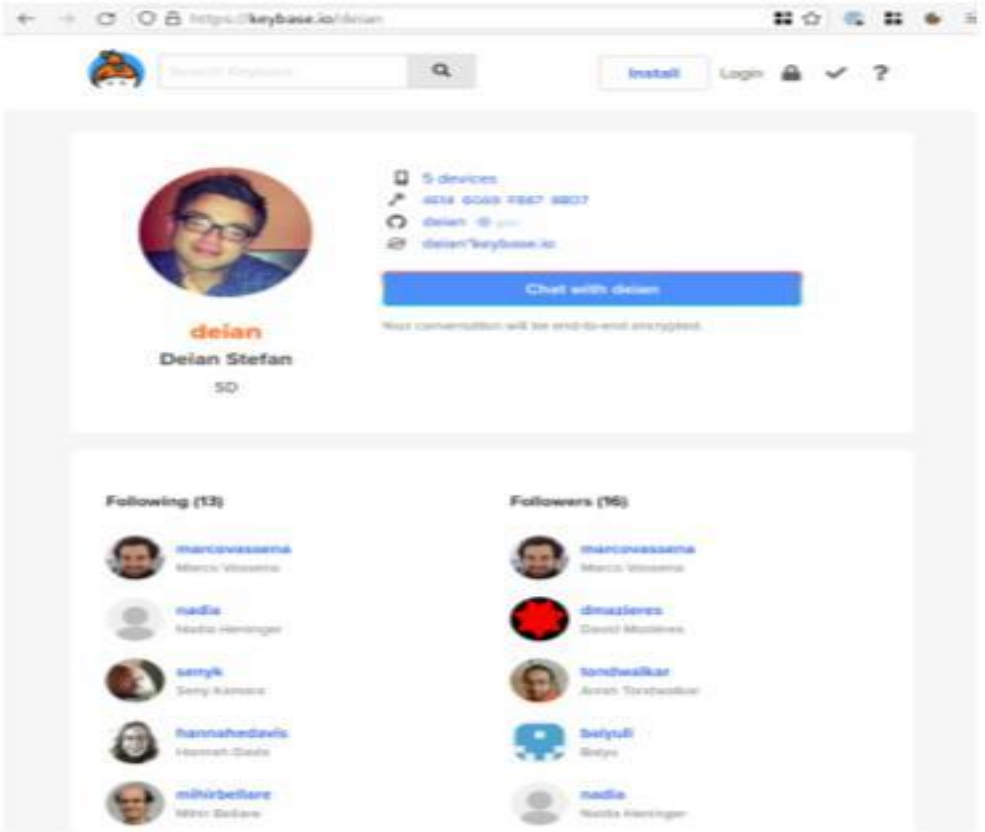
Public Key Infrastructure: Establishing Trust in Keys

Ways to establish trust in keys:

- Web of Trust (e.g., PGPG)
  - In a web of trust, you establish trust in intermediaries of your choice.
  - You then transitively trust the keys they sign.

```
$ gpg --edit-key dickey@invisible-island.net
gpg (GnuPG) 2.2.29; Copyright (C) 2021 Free Software Foundation, Inc. This is free
software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
pub  dsa1024/702353E0F7E48EDB
     created: 2004-01-05  expires: 2022-07-05  usage: SC trust: unknown
                          validity: unknown
sub  elg1024/0296C3D9E4374AE1
     created: 2004-01-05  expires: never       usage: E
[ unknown] (1). Thomas Dickey <dickey@invisible-island.net> gpg> trust
pub  dsa1024/702353E0F7E48EDB
     created: 2004-01-05  expires: 2022-07-05  usage: SC trust: unknown
                          validity: unknown
sub  elg1024/0296C3D9E4374AE1
               created: 2004-01-05  expires: never       usage: E
          [ unknown] (1). Thomas Dickey <dickey@invisible-island.net>
Please decide howfar you trust this user to correctly verify other users' keys (by looking at
passports, checking fingerprints from different sources, etc.)
  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menuYour decision?
```

A more modern and practical WoT: Keybase



TLS: Transport Layer Security

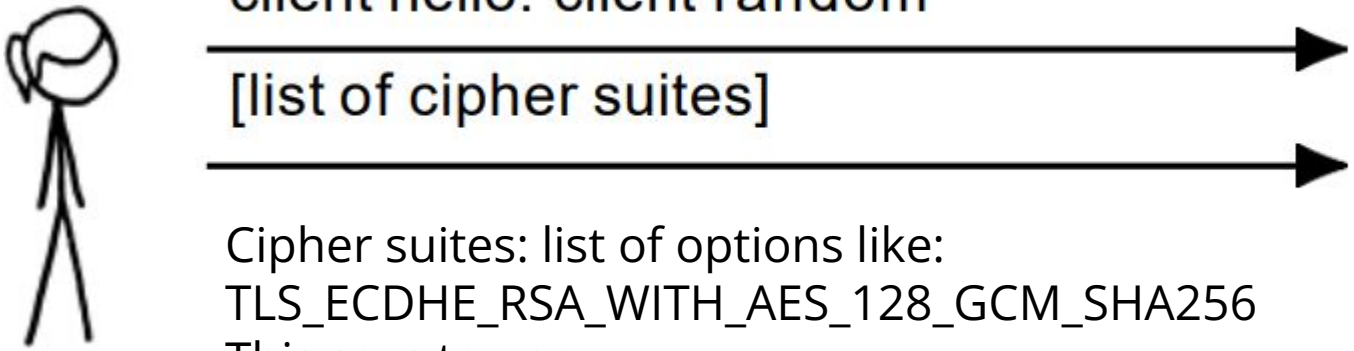
- TLS provides an encrypted channel for application data.
- Used for HTTPS: HTTP inside of a TLS session
- Used to be called SSL (Secure Sockets Layer) in the 90s.

  SSL 1.0 Terribly insecure; never released.
  SSL 2.0 Released 1995; terribly insecure, deprecated in 2011
  SSL 3.0 Released 1996; insecure, deprecated in 2015.
  TLS1.0 Released 1999; deprecated in 2020.
  TLS1.1 Released 2006; deprecated in 2020.
  TLS1.2 Released 2008. Ok.
  TLS 1.3 Standardized in August 2018 and is being rolled out now; major change from TLS1.2.

Deprecated - no longer recommended for use

TLS 1.2 with Diffie-Hellman Key Exchange

Step 1: The client (browser) tells the server what kind of cryptography it supports.



Cipher suites: list of options like:
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
This says to use

- elliptic curve Diffie-Hellman for key exchange
- RSA digital signatures
- 128-bit AES for symmetric encryption
- GCM (Galois Counter Mode) AES mode of operation
- SHA-256 for hash function

TLS 1.2 with Diffie-Hellman Key Exchange

Step 1: The client (browser) tells the server what kind of cryptography it supports.

client hello: client random

[list of cipher suites]

Cipher suites: list of options like:
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

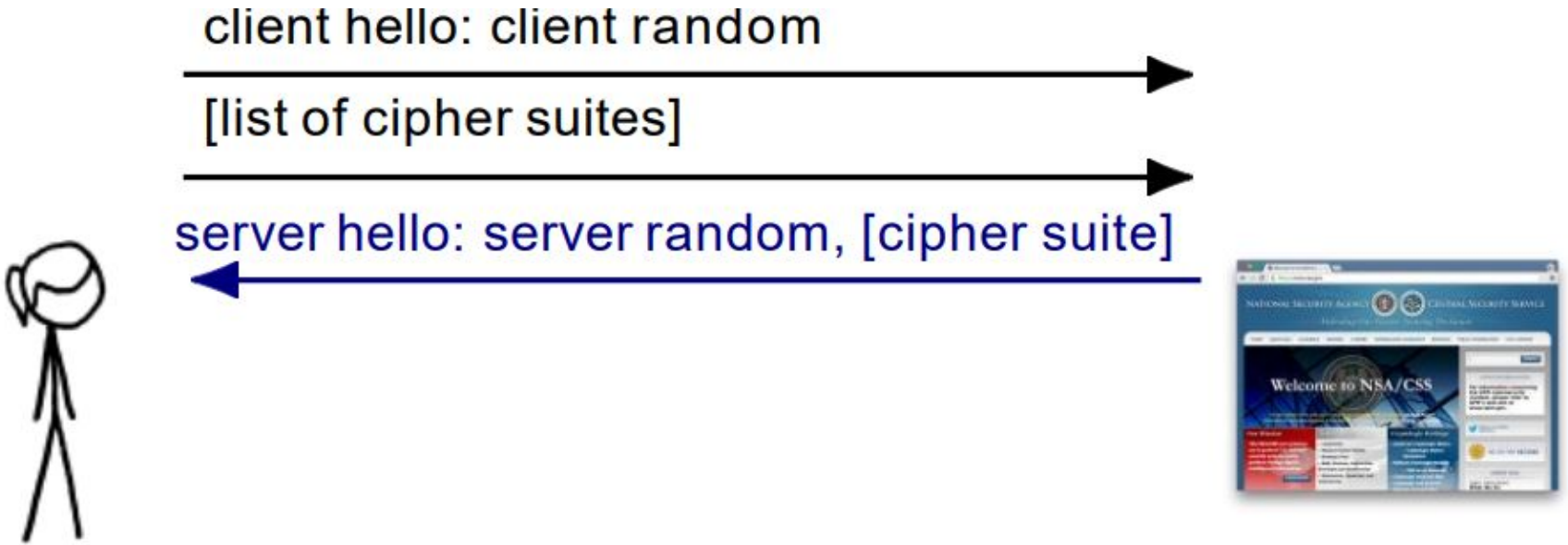
Server cipher suite configuration can be confusing and difficult for sysadmins. Many insecure options like

TLS_DHE_RSA_WITH_DES_CBC_SHA
or
TLS_NULL_WITH_NULL_NULL

Subtle protocol errors around cipher suite negotiation.

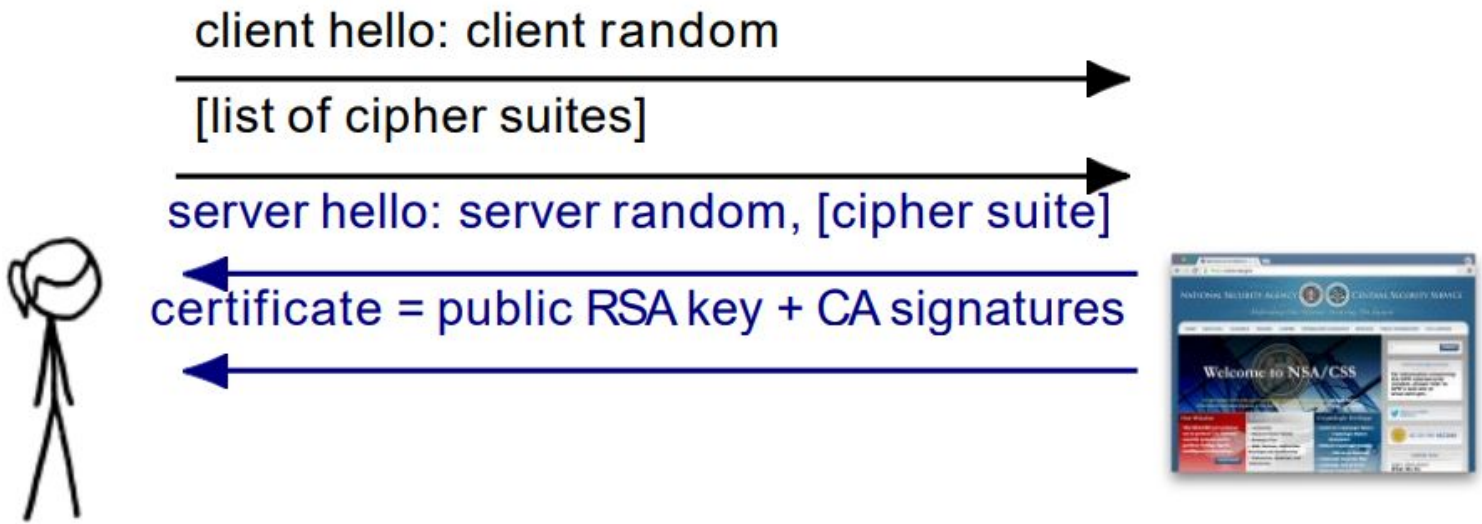
TLS 1.2 with Diffie-Hellman Key Exchange

Step 2: The server tells the client which kind of cryptography it wishes to use.



TLS 1.2 with Diffie-Hellman Key Exchange

Step 3: The server sends over its certificate which contains the server's public key and signatures from a certificate authority.



Certificates and Certificate Authorities in TLS

Certificate - digital object used to authenticate an entity

Certificate Authority - an entity that issues a certificate

Certificates signed by CAs.

Browsers come with set of trusted CAs.

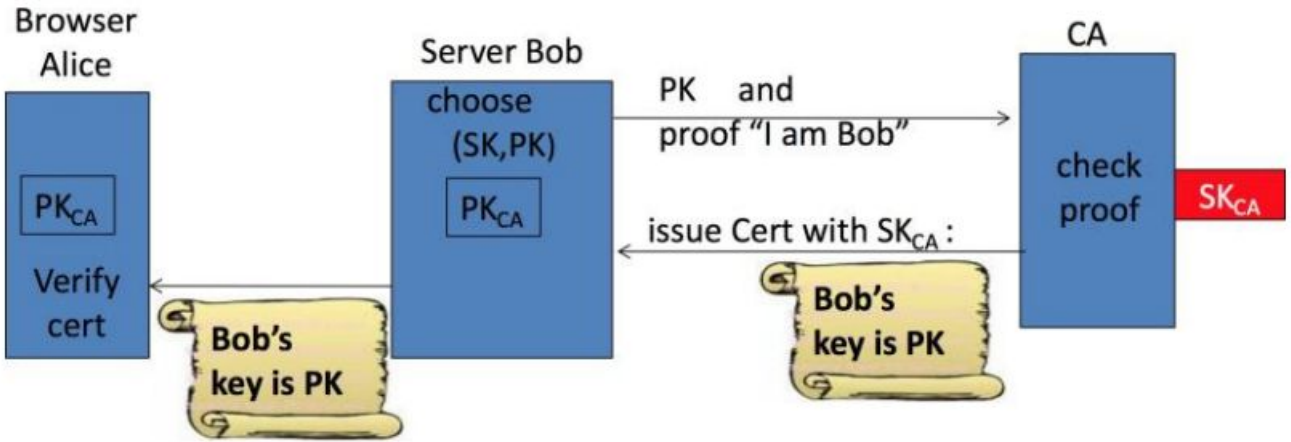
Certificates typically valid for 3 months to multiple years.

Certificates and Certificate Authorities in TLS

Website public keys are encoded into certificates.

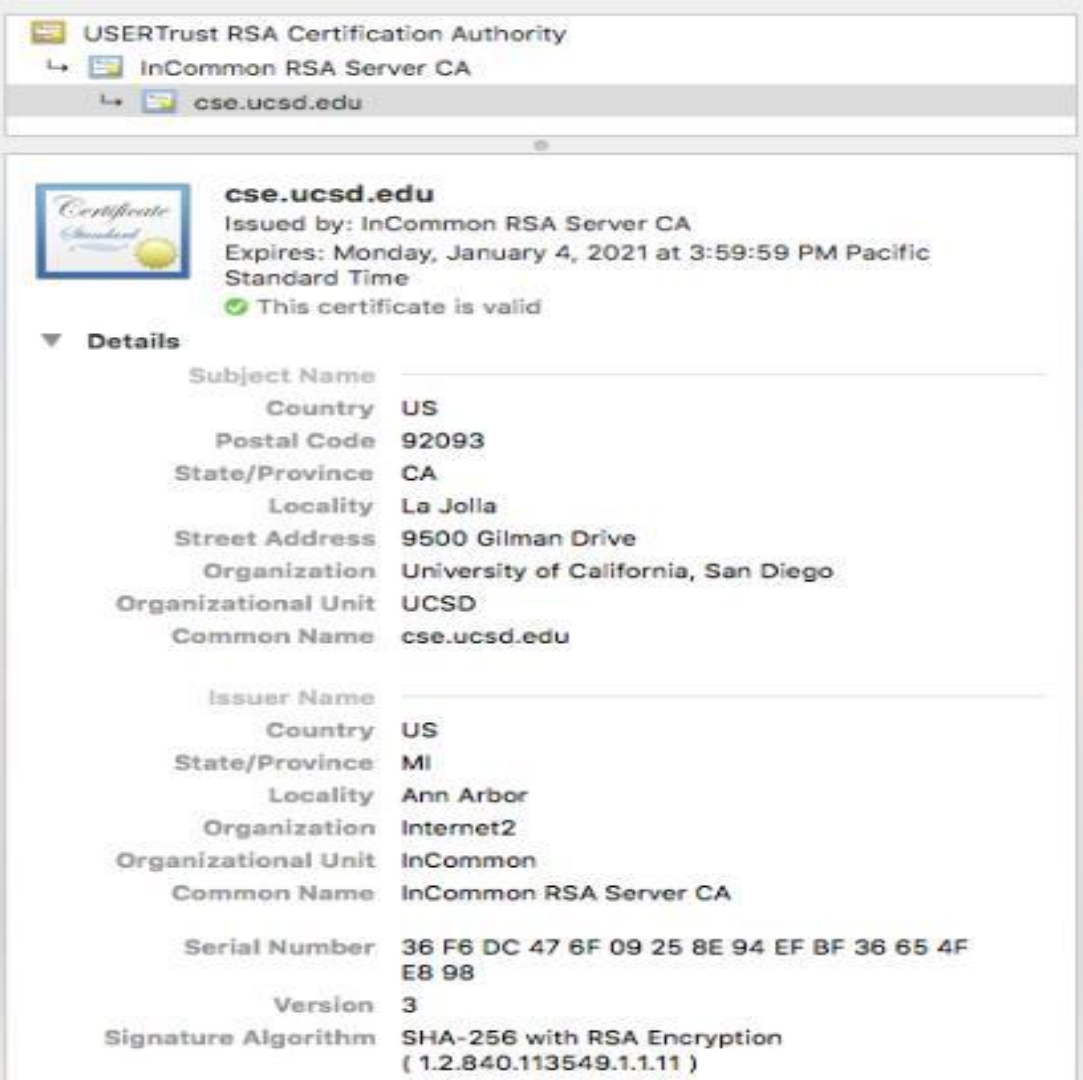
Certificates signed by CAs.
Browsers come with set of trusted CAs.

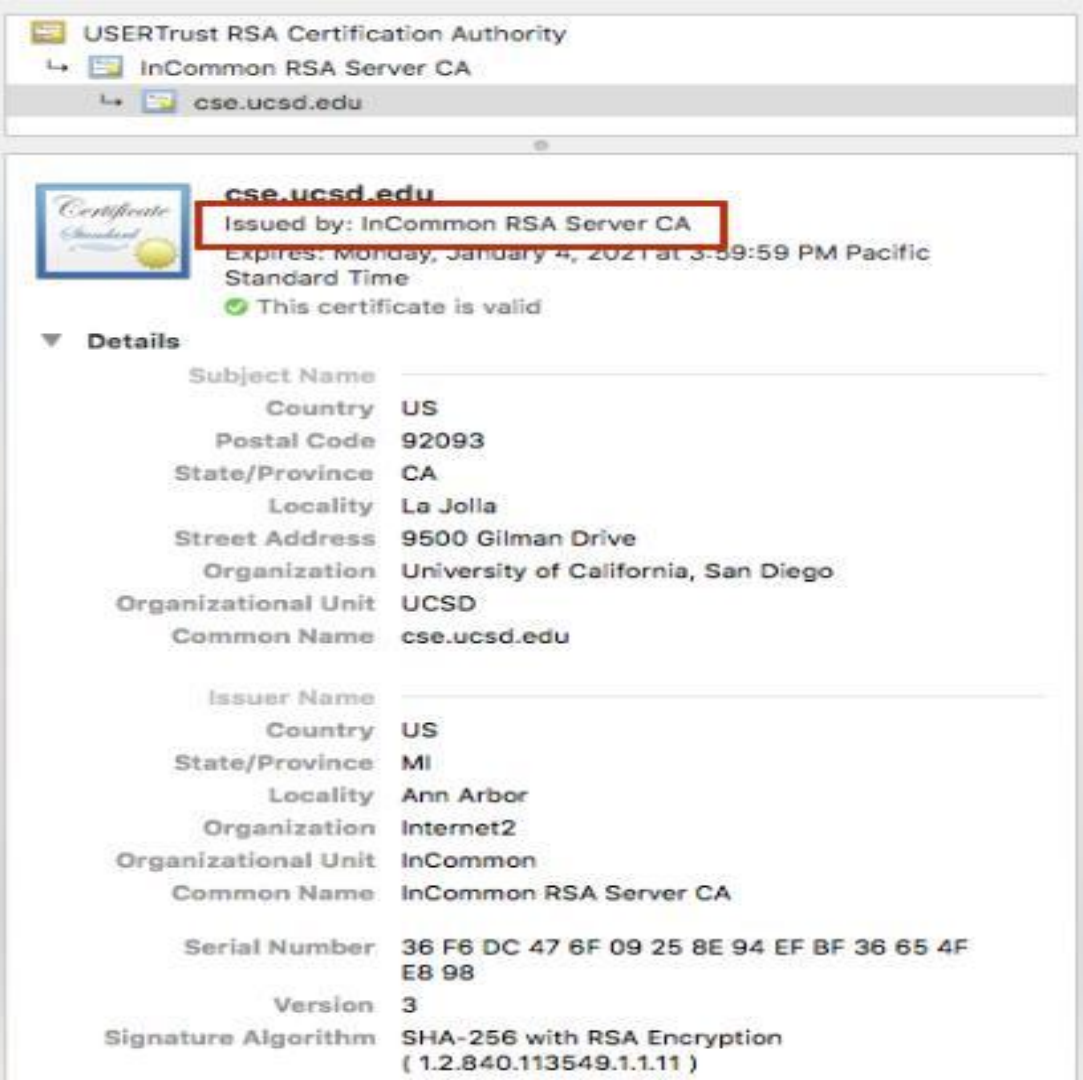
To verify a certificate, browsers verify chain of digital certificates back to trusted root CA.



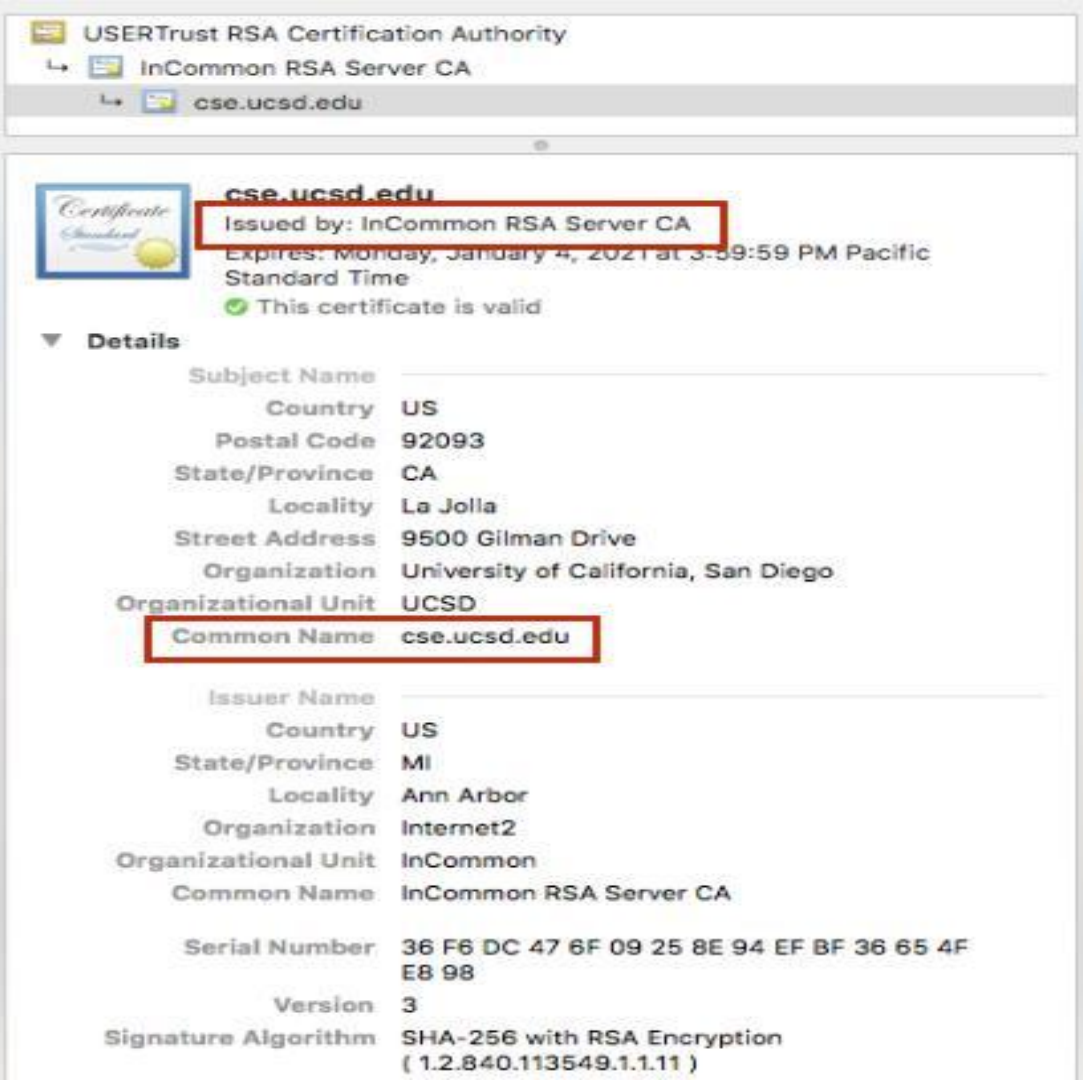
Certificates typically valid for 3 months to multiple years.

Sample certificate


USERTrust RSA Certification Authority
InCommon RSA Server CA
cse.ucsd.edu
cse.ucsd.edu
Issued by: InCommon RSA Server CA
Expires: Monday, January 4, 2021 at 3:59:59 PM Pacific Standard Time
This certificate is valid
Details
Subject Name
Country US
Postal Code 92093
State/Province CA
Locality La Jolla
Street Address 9500 Gilman Drive
Organization University of California, San Diego
Organizational Unit UCSD
Common Name cse.ucsd.edu
Issuer Name
Country US
State/Province MI
Locality Ann Arbor
Organization Internet2
Organizational Unit InCommon
Common Name InCommon RSA Server CA
Serial Number 36 F6 DC 47 6F 09 25 8E 94 EF BF 36 65 4F E8 98
Version 3
Signature Algorithm SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)



Who are we trusting?

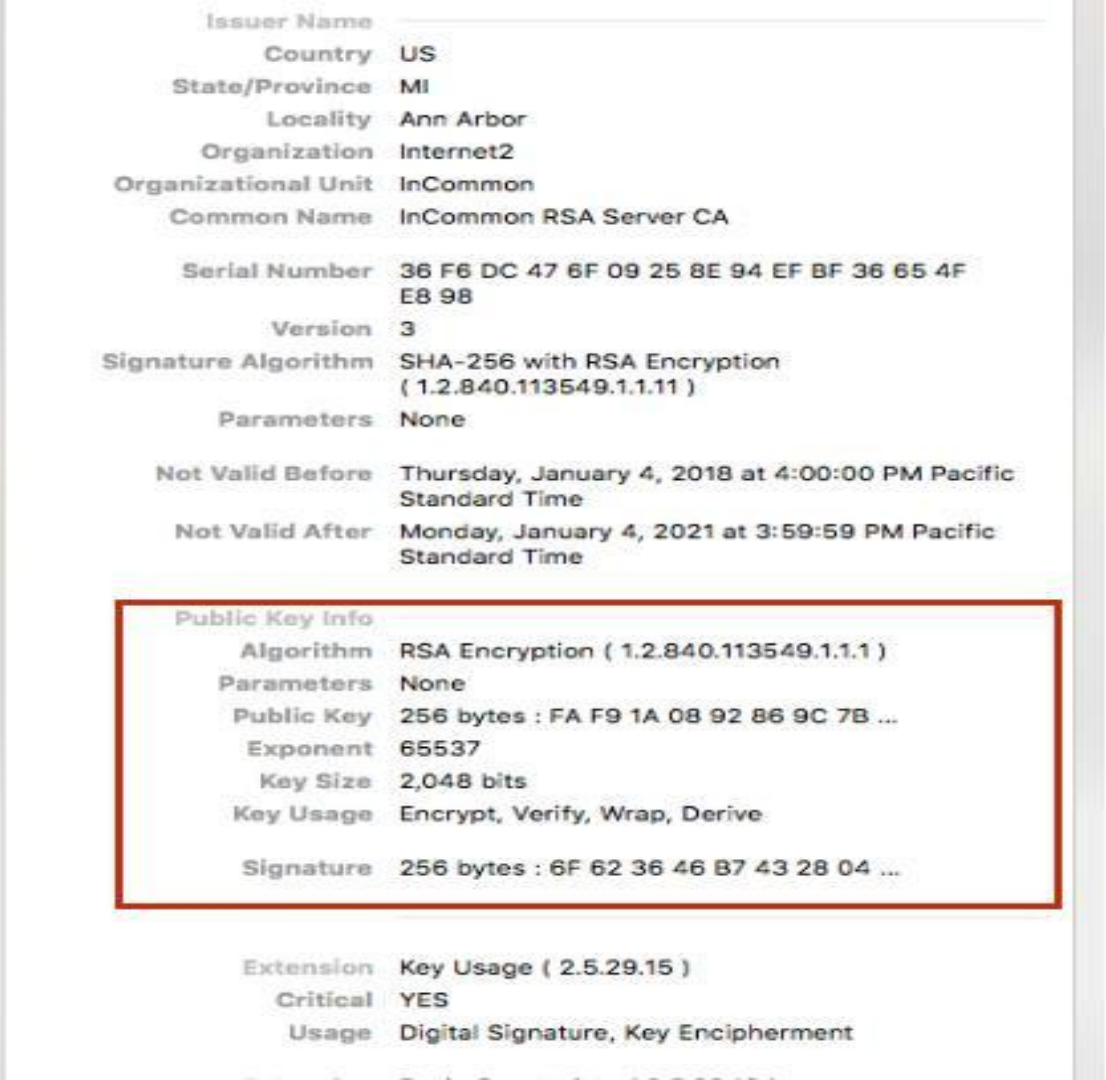


Who are we trusting?

Who is this cert for?

Key ID	1E 05 A3 77 8F 6C 96 E2 5B 87 4B A6 B4 86 AC 71 00 0C E7 38
Extension	Subject Alternative Name (2.5.29.17)
Critical	NO
DNS Name	cse.ucsd.edu
DNS Name	cs.ucsd.edu
DNS Name	www-cs.ucsd.edu
DNS Name	www-cse.ucsd.edu
DNS Name	www.cs.ucsd.edu
DNS Name	www.cse.ucsd.edu
Extension	Certificate Policies (2.5.29.32)
Critical	NO
Policy ID #1	(1.3.6.1.4.1.5923.1.4.3.1.1)
Qualifier ID #1	Certification Practice Statement (1.3.6.1.5.5.7.2.1)
CPS URI	https://www.incommon.org/cert/repository/cps_ssl.pdf
Policy ID #2	(2.23.140.1.2.2)
Extension	CRL Distribution Points (2.5.29.31)
Critical	NO
URI	http://crl.incommon-rsa.org/InCommonRSAServerCA.crl
Extension	Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
Critical	NO
Method #1	CA Issuers (1.3.6.1.5.5.7.48.2)
URI	http://crt.usertrust.com/InCommonRSAServerCA_2.crt
Method #2	Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
URI	http://ocsp.usertrust.com

Who is this cert for?



Issuer Name

Country	US
State/Province	MI
Locality	Ann Arbor
Organization	Internet2
Organizational Unit	InCommon
Common Name	InCommon RSA Server CA
Serial Number	36 F6 DC 47 6F 09 25 8E 94 EF BF 36 65 4F E8 98
Version	3
Signature Algorithm	SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)
Parameters	None
Not Valid Before	Thursday, January 4, 2018 at 4:00:00 PM Pacific Standard Time
Not Valid After	Monday, January 4, 2021 at 3:59:59 PM Pacific Standard Time

Public Key Info

Algorithm	RSA Encryption (1.2.840.113549.1.1.1)
Parameters	None
Public Key	256 bytes : FA F9 1A 08 92 86 9C 7B ...
Exponent	65537
Key Size	2,048 bits
Key Usage	Encrypt, Verify, Wrap, Derive
Signature	256 bytes : 6F 62 36 46 B7 43 28 04 ...

Extension	Key Usage (2.5.29.15)
Critical	YES
Usage	Digital Signature, Key Encipherment

CSE's pub key info

Key ID	1E 05 A3 77 8F 6C 96 E2 5B 87 4B A6 B4 86 AC 71 00 0C E7 38
Extension	Subject Alternative Name (2.5.29.17)
Critical	NO
DNS Name	cse.ucsd.edu
DNS Name	cs.ucsd.edu
DNS Name	www-cs.ucsd.edu
DNS Name	www-cse.ucsd.edu
DNS Name	www.cs.ucsd.edu
DNS Name	www.cse.ucsd.edu
Extension	Certificate Policies (2.5.29.32)
Critical	NO
Policy ID #1	(1.3.6.1.4.1.5923.1.4.3.1.1)
Qualifier ID #1	Certification Practice Statement (1.3.6.1.5.5.7.2.1)
CPS URI	https://www.incommon.org/cert/repository/cps_ssl.pdf
Policy ID #2	(2.23.140.1.2.2)
Extension	CRL Distribution Points (2.5.29.31)
Critical	NO
URI	http://crl.incommon-rsa.org/InCommonRSAServerCA.crl
Extension	Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
Critical	NO
Method #1	CA Issuers (1.3.6.1.5.5.7.48.2)
URI	http://crt.usertrust.com/InCommonRSAServerCA_2.crt
Method #2	Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
URI	http://ocsp.usertrust.com

Where we should check for revocation information