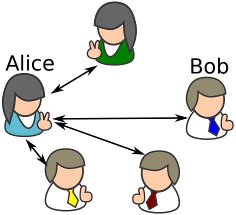
50.020 Security
Lecture 14 - Key Establishment

This lecture

- This lecture:
  - More design challenges
  - Key establishment schemes
- Some content in these slides based on slide set of "Understanding Cryptography" by C. Paar and J. Petzl

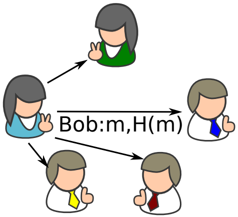
Design challenge: SUTD Secure Comm System



- Goal: design a secure chat application for SUTDents on campus
  - Provides confidentiality, integrity
- Wireless devices and communication can be used
- Staff/Student ID as source/destination for message sending/receiving
- How to build this in a secure way?
  - What kind of infrastructure is required?
  - Which/how many messages have to be exchanged?
  - What happens if a new student joins SUTD?

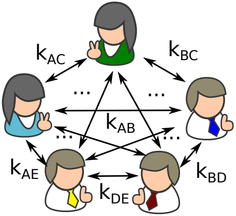
Solution example

- Insecure solution (without any key):



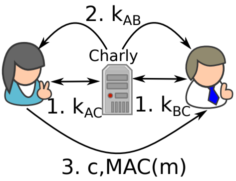
- Alice computes hash value of m: H(m)
- Alice broadcasts 1 message: "Bob:m,H(m)"
- Bob receives message, validates H(m)
- What are the security problems here?

Distributed solution idea



- Everyone has a shared key with everyone
- Messages can be sent directly
- To send *m* to Bob, Alice does the following:
  - Look up the shared key k_{AB}
  - Compute the MAC(m,k_{AB})
  - Encrypt m to obtain ciphertext *c* (e.g. using AES with k_{AB})
  - Send $< c, MAC(m, k_{AB}) >$ to Bob
  - Bob decrypts c using k_{AB}, and verifies MAC(m,k_{AB})

Centralized Solution idea



- One server (Charly), everyone gets a shared key with server
- *n* keys pre-shared (somehow)
- If Alice wants to send message to Bob
  - Alice securely asks Charly for a new key K_{AB}
  - Alice uses that key to encrypt, and compute MAC for the message
  - Bob gets k_{AB} securely from Charly, decrypts message
- Delay+overhead for server communication
- We also need to trust the server (with our shared keys)

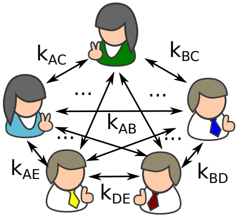
Conclusion from design challenge

- Given symmetric encryption and MAC, confidentiality and integrity is easy
- The main practical challenge is key establishment!
- This challenge appears frequently in real systems
  - How to authenticate users for SUTD WiFi?
  - How to set up secure communication for ad hoc users?
- We will now look at ways to solve this key establishment problem in practise

Key establishment

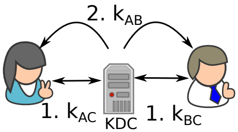
- Four major ways to establish shared keys:
  - Pre-share $n(n-1)/2$ keys
  - "Magic" secure *out-of-band* channel
  - Centralized key distribution center
  - Dynamic peer-to-peer establishment

Pre-sharing keys



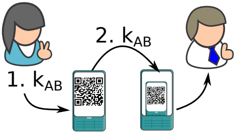
- Logistical and operational nightmare. Assume n=1001, i.e., there are 1001 users:
  - The number of keys needed is $(n * (n-1))/2 \approx 1$ Million
  - Storage for each user: 1000*128 bit=16kB (not a a big issue, maybe)
  - If a new user joins, all n students need one additional key. . . (might be a big issue)

Key Distribution Centers



- Each new user has pre-shared key with KDC
- KDC can create k_{AB} on demand and send to Alice and Bob
- The *Kerboros* protocol uses such a KDC
- Problems with KDC
  - If KDC gets compromised, past keys could be disclosed
  - KDC is single point of failure
  - Communication overhead and load for KDC

Out-of-Band channel



- An additional channel, that is "more secure" for some reason
- Could be a phone line and voice communication
- Could be a visual channel (comparing random strings)
- Example:
  - One user/device creates a random key as 32 Hex numbers
  - Second user reads this key, and enters it into his device
- Problems
  - Scalability, automation, user errors,...

Dynamic key establisment

- What if we could find a protocol to automatically negotiate keys?
- Alice and Bob could run it to derive a shared key
- Ideally, both would contribute "randomness"
  - This prevents an attacker from re-using keys
- What is the problem for such a protocol?

Dynamic key establisment

- What if we could find a protocol to automatically negotiate keys?
- Alice and Bob could run it to derive a shared key
- Ideally, both would contribute "randomness"
  - This prevents an attacker from re-using keys
- What is the problem for such a protocol?

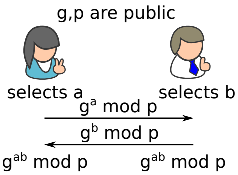
- Without any pre-existing key:
  - No integrity protection is possible
  - No authenticity can be verified
  - No confidentiality is possible
- Public key protocols solve this problem!

Diffie-Hellman Key Exchange: Introduction

Diffie-Hellman key exchange

- Setting: Alice and Bob want to establish shared key
  - Can only communicate over public channel
  - Both can compute modular exponentiation (to discuss today)
  - Attacker is passive eavesdropper
  - How to agree on key without attacker learning it?
- Public parameters:
  - Large prime p
  - Some integer g

Diffie-Hellman key exchange (Description)



- They each create a random private number a (resp. b)
- They now publicly announce $g^a \mod p$ (resp. $g^b \mod p$)
- Alice receives $g^b \mod p$ and *exponentiates* with secret a
- Bob receives $g^a \mod p$ and *exponentiates* with secret b
- Both end up with the same value $g^{ab} \mod p$
- Attacker cannot compute $g^{ab} \mod p$ or learn a, b
- $g^{ab} \mod p$ is now the shared key for Alice and Bob

Diffie-Hellman Key Exchange: Why Is It Secure?

Discrete Logarithm Problem

- Why is this secure?
- Attacker cannot find a from g^a mod p
  - Although she has g and p !
- This is called the *discrete logarithm problem*

Definition (Discrete Logarithm Problem)

- Given finite cyclic group G of order p-1 and an element r, and a (multiplicative) generator element g
- The DLP is: find $1 \leq x \leq p-1$, such that $g^x \equiv r \bmod p$
- This can also be written as: $x = \log_g r \bmod p$

Discrete Logarithm Problem

How to compute the discrete logarithm?

- One solution (brute force): iteratively multiply the generator element, until reaching result of r:
  - $r = g \times g \times g \times g \times \ldots$ [x times]

Why is the computation of DLP hard?

- Modular multiplication is expensive
  - and x can be quite big, and can be any of the p-1 elements (for example, p is 2048 bits long)
- Algorithms for DLP solving:
  - Brute force linear search
  - Shank's Baby-Step-Giant-Step algorithm
  - Pollard's Rho Method
  - Pohlig-Hellman method

Where is the DLP used?

- Core of many cryptographic schemes
- Discrete logarithm problem is (very) hard in some well-chosen groups, and thus secure-enough cryptosystems can be built.
- Examples of the well-chosen finite cyclic groups:
  - The original Diffie-Hellmann key exchange uses $\mathbb{Z}_p$
  - *Elgamal* encryption and the *Digital Signature* algorithm use $\mathbb{Z}_p$
  - *Elliptic-Curve-Cryptography* uses a group defined by an elliptic curve equation

Diffie-Hellman Key Exchange Implementation: How To Compute Modular Exponentiation?

How to compute modular exponentiation

- There is no "exponentiation" instruction on CPU
- There is no "take modular remainder" instruction on CPU
- One example algorithm to computer modular exponentiation: "Square-And-Multply algorithm"
- We show an example first before presenting the algorithm.

Example for Square-And-Multiply

Example

How to compute: $3^5 \ mod \ 11$*?*

```
def squAndMult(3, 5, 11):
  res=1
  for i in '101':
    res=res*res % 11
      if (i=='1'):
        res=res*3 % 11
  return res
```

Example for Square-And-Multiply

Example

How to compute: $3^5 \ mod\ 11$*?*

```
def squAndMult(3, 5, 11):
  res=1
  for i in '101':
    res=res*res % 11
      if (i=='1'):
        res=res*3 % 11
  return res
```

- After first round of for-loop:
  - res=3

Example for Square-And-Multiply

Example

How to compute: $3^5 \ mod \ 11$*?*

```
def squAndMult(3, 5, 11):
  res=1
  for i in '101':
    res=res*res % 11
      if (i=='1'):
        res=res*3 % 11
  return res
```

- After first round of for-loop:
  - res=3

- After second round of for-loop:
  - res=9

Example for Square-And-Multiply

Example

How to compute: $3^5 \ mod \ 11$?

```
def squAndMult(3, 5, 11):
  res=1
  for i in '101':
    res=res*res % 11
      if (i=='1'):
        res=res*3 % 11
  return res
```

- After first round of for-loop:
  - res=3
- After second round of for-loop:
  - res=9
- After third round of for-loop:
  - 81 % 11=4
  - 4*3 % 11=1
  - $->$ res is 1

How to compute modular exponentiation: Square-And-Multply algorithm

Square-And-Multply algorithm (MSB first,i.e., bits are read from left to right)

```
def squAndMult(m, k, n):
  res=1
  for i in bin(k)[2:]:# step bitwise through key
    res=res*res %n
      if (i=='1'):
        res=res*m % n
  return res
```

How to compute modular exponentiation: Square-And-Multiply algorithm

- Expensive ops: Square and Multiply
- Operations executed: Square, Multiply, Square, Square, Multiply

Conclusion

- Keys are fundamentally required for confidentiality and authentication
- They need to be established or exchanged by different ways:
  - Pre-distribution
  - Centralized KDC
  - Use of out-of-band channels
  - Dynamic negotiation, for example, by DHKE
- DHKE introduces *public key* cryptography
  - More on public key cryptography in the next lectures