KEY DISTRIBUTION:
PKI and SESSION-KEY EXCHANGE
The public key setting

Bob’s secret key is $sk[B]$ and its associated public key is $pk[B]$. The public key setting assumes Alice is in possession of $pk[B]$.

\[
\begin{align*}
\text{Alice}^{pk[B]} & \quad \text{Bob} \\
C & \leftarrow^s E_{pk[B]}(M) \quad C \quad M \leftarrow D_{sk[B]}(C) \\
V_{pk[B]}(M, \sigma) & \quad M, \sigma \quad \sigma \leftarrow^s S_{sk[B]}(M)
\end{align*}
\]

Now Alice can encrypt a message $M$ under $pk[B]$ to get a ciphertext $C$ that Bob can decrypt using $sk[B]$.

Bob can sign a message $M$ using $sk[B]$ to get signature $\sigma$ that Alice can verify using $pk[B]$.

But how does Alice get $pk[B]$?
But who exactly are “Alice” and “Bob”? 

A question of **identity** that is central to key distribution and its security.
Names are bound to people via societal means

The question of identity

Alice

Alice Wonder
Alice Waters
Alice Walker
Alice Poker

Bob

Bob Sponge
Bob Builder
Bob Marley
Bob Hope

Established out-of-band binding of name to person
In TLS, Bob is a server

These are difficult issues

Established how?
What do `domain name` and `entity` even mean?

binding of domain name to entity
In TLS, Alice is a client

Alice’s identity could be her ip address or domain name.

Alice need not be an individual; she could be a corporation, a server herself, ...
How can Alice get Bob’s public key?

**A simple idea:** Bob generates his keys, and just sends $pk[B]$ to Alice with his identity “Bob” attached.

This enables:

- $C \xleftarrow{\$} \mathcal{E}_{pk[B]}(M)$
- $C \xrightarrow{} M \xleftarrow{} \mathcal{D}_{sk[B]}(C)$
- $\mathcal{V}_{pk[B]}(M, \sigma)$
- $M, \sigma \xleftarrow{} \sigma \xleftarrow{\$} \mathcal{S}_{sk[B]}(M)$
Entity-in-the-middle attack

Alice

\[ \text{Bob, } pk[A] \]

\[ (pk[A], sk[A]) \leftarrow \mathcal{K} \]

Adversary $A$

So:

\[ C \leftarrow \mathcal{E}_{pk[A]}(M) \]

\[ M \leftarrow \mathcal{D}_{sk[A]}(C) \]

\[ \mathcal{V}_{pk[A]}(M, \sigma) \]

\[ \sigma \leftarrow \mathcal{S}_{sk[A]}(M) \]

Adversary can decrypt ciphertexts intended for Bob.

Adversary can forge Bob's signatures.
PKI, CAs and certificates

**Goal:** Alice gets an **authentic** copy of Bob’s public key, meaning if \( pk \) claims to come from Bob, she has proof to that effect.

**Popular Solution:** The PKI (Public Key Infrastructure).

**Certificate authority:** Trusted entity that provides the above proof.

**Certificate:** The proof

**Note:** There are other ways to reach the goal: Bob could post his public key on his Facebook; post it on his personal or corporate webpage; include it as an attachment in his emails; put it on a keyserver like openpgp SGS; hand it to Alice in person; ...
Let’s Encrypt is a **free, automated, and open** Certificate Authority.

- [Get Started](https://letsencrypt.org)
- [Sponsor](https://letsencrypt.org/donate)
Root Certificates

Our roots are kept safely offline. We issue end-entity certificates to subscribers from the intermediates in the next section.

- Active
  - ISRG Root X1 (self-signed)

We’ve set up websites to test certificates chaining to our roots.

- ISRG Root X1 Valid Certificate
  - https://valid-isrgrootx1.letsencrypt.org/
- ISRG Root X1 Revoked Certificate
  - https://revoked-isrgrootx1.letsencrypt.org/
- ISRG Root X1 Expired Certificate
  - https://expired-isrgrootx1.letsencrypt.org/

Intermediate Certificates

IdenTrust has cross-signed our intermediates. This allows our end certificates to be accepted by all major browsers while we propagate our own root.

Under normal circumstances, certificates issued by Let’s Encrypt will come from “Let’s Encrypt Authority X3”. The other intermediate, “Let’s Encrypt Authority X4”, is reserved for disaster recovery and will only be used should we lose the ability to issue with “Let’s Encrypt Authority X3”. The X1 and X2 intermediates were our first generation of intermediates. We’ve replaced them with new intermediates that are more compatible with Windows XP.

- Active
  - Let’s Encrypt Authority X3 (IdenTrust cross-signed)
  - Let’s Encrypt Authority X3 (Signed by ISRG Root X1)
# Some certificate authorities

<table>
<thead>
<tr>
<th>Rank</th>
<th>Issuer</th>
<th>Usage</th>
<th>Market share</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IdenTrust</td>
<td>20.4%</td>
<td>39.7%</td>
</tr>
<tr>
<td>2</td>
<td>Comodo</td>
<td>17.9%</td>
<td>34.9%</td>
</tr>
<tr>
<td>3</td>
<td>DigiCert</td>
<td>6.3%</td>
<td>12.3%</td>
</tr>
<tr>
<td>4</td>
<td>GoDaddy</td>
<td>3.7%</td>
<td>7.2%</td>
</tr>
<tr>
<td>5</td>
<td>GlobalSign</td>
<td>1.8%</td>
<td>3.5%</td>
</tr>
<tr>
<td>6</td>
<td>Certum</td>
<td>0.4%</td>
<td>0.7%</td>
</tr>
<tr>
<td>7</td>
<td>Actalis</td>
<td>0.2%</td>
<td>0.3%</td>
</tr>
<tr>
<td>8</td>
<td>Entrust</td>
<td>0.2%</td>
<td>0.3%</td>
</tr>
<tr>
<td>9</td>
<td>Secom</td>
<td>0.1%</td>
<td>0.3%</td>
</tr>
<tr>
<td>10</td>
<td>Let's Encrypt</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>11</td>
<td>Trustwave</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>12</td>
<td>WISeKey Group</td>
<td>&lt; 0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>13</td>
<td>StartCom</td>
<td>&lt; 0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>14</td>
<td>Network Solutions</td>
<td>&lt; 0.1%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
Certificate process

- Bob generates $sk$ and $pk$ by locally running key-generation
- Bob sends his identity and $pk$ to CA
- CA does identity check to ensure $pk$ is Bob’s
- Bob proves knowledge of $sk$ to CA
- CA issues certificate to Bob
- Bob sends certificate to Alice
- Alice verifies certificate and extracts Bob’s public key $pk$
Bob locally runs a prescribed key-generation algorithm to generate his keys: \((pk, sk) \leftarrow \mathcal{K}\)
Key generation example: RSA Key generation with openssl

Generating a private RSA key

1. Generate an RSA private key, of size 2048, and output it to a file named key.pem:

   ```
   $ openssl genrsa -out key.pem 2048
   Generating RSA private key, 2048 bit long modulus
   ........+++
   ............................................................+++ 
   e is 65537 (0x10001)
   ```

2. Extract the public key from the key pair, which can be used in a certificate:

   ```
   $ openssl rsa -in key.pem -outform PEM -pubout -out public.pem
   writing RSA key
Generating a private EC key

1. Generate an EC private key, of size 256, and output it to a file named key.pem:

   $ openssl ecparam -name prime256v1 -genkey -noout -out key.pem

2. Extract the public key from the key pair, which can be used in a certificate:

   $ openssl ec -in key.pem -pubout -out public.pem
   read EC key
   writing EC key

After running these two commands you end up with two files: key.pem and public.pem. These files are referenced in various other guides on this page when dealing with key import.
Bob sends his identity *Bob* (domain name, ip address, email address, ...) and his public key *pk* to the certificate authority (CA).

Upon receiving *(Bob, pk)* the CA performs some checks to ensure *pk* is really Bob’s key.

**Example:** If *Bob* is a domain name, then the CA sends Bob a challenge and checks that he can put it on the webpage of the domain name.

**Example:** If *Bob* is an email address, then the CA sends an email to that address with a link for Bob to click to verify that he owns the address.

**Example:** If *Bob* is a passport or driver’s license, the CA may be able to verify it physically, out of band.

**Proof of knowledge of secret key:** The CA might have Bob sign or decrypt something under *pk* to ensure that Bob knows the corresponding secret key *sk*. This ensures Bob has not copied someone else’s key.
Certificate Issuance

Once CA is convinced that $pk$ belongs to $Bob$, it forms a certificate

$$CERT_{Bob} = (CERTDATA, \sigma),$$

where $\sigma$ is the CA’s signature on $CERTDATA$, computed under the CA’s secret key $sk[CA]$, and $CERTDATA$ contains:

- Bob’s public key $pk$, and its type (RSA, EC, ...)
- Identity $Bob$ of Bob
- Name of CA
- Expiry date of certificate
- ...

The certificate $CERT_{Bob}$ is returned to Bob.
Bob can send \( CERT_{Bob} \) to Alice who is assumed to have the CA’s public key \( pk[CA] \) and now will:

- Parse it as \( (CERTDATA, \sigma) \leftarrow CERT_{Bob} \)
- Check that \( \nu_{pk[CA]}(CERTDATA, \sigma) = 1 \)
- Extract \( (pk, Bob, expiry, \ldots) \leftarrow CERTDATA \)
- Check certificate has not expired
- \( \ldots \)

If all is well, Alice accepts the certificate and is ready to use the public key \( pk \) therein.

**How does Bob get \( pk[CA] \)?** CA public keys are embedded in software such as your browser.
Certificate hierarchies

\[
\begin{array}{c}
\text{CA(USA)} \\
\text{CA(Calif)} \quad \text{CA(Mass)} \\
\text{CA(SD)} \\
\text{CA(UCSD)} \\
\text{Mihir}
\end{array}
\]

\[
\begin{align*}
\text{CERT}_{\text{Mihir}} &= \text{CERT}[\text{CA(USA)} : \text{CA(Calif)}] \\
& \quad \text{CERT}[\text{CA(Calif)} : \text{CA(SD)}] \\
& \quad \text{CERT}[\text{CA(SD)} : \text{CA(UCSD)}] \\
& \quad \text{CERT}[\text{CA(UCSD)} : \text{Mihir}]
\end{align*}
\]

\[
\text{CERT}[X : Y] = (pk[Y], Y, \ldots, S_{sk[X]}(pk[Y], Y, \ldots))
\]

To verify \(\text{CERT}_{\text{Mihir}}\) you need only \(pk_{\text{CA[USA]}}\).
Why certificate hierarchies?

• It is easier for CA(UCSD) to check Mihir’s identity (and issue a certificate) than for CA(USA) since Mihir is on UCSD’s payroll and UCSD already has a lot of information about him.

• Spreads the identity-check and certification job to reduce work for individual CAs

• Browsers need to have fewer embedded public keys. (Only root CA public keys needed.)
Certificates on Mac: keychain

Keychain Access: Keychains
- login
- Local Items
- System
- System Roots

Name:
- AddTrust External CA Root
- auth.resnet.ucsd.edu
- auth.ucsd.edu
- auth.ucsd.edu
- com.apple.idms.appleid.pr0d.46414e6a564e2f746958484d6d4475473070582b2b513d3d
- com.apple.idms.appleid.pr0d.46414e6a564e2f746958484d6d4475473070582b2b513d3d
- InCommon RSA Server CA
- InCommon Server CA
- member: B3F2F72E-E369-43F3-97C6-B51C88991470 76EF51EC-49C1-47F6-A069-EF6FED3300FF
- ucsb-secure.wireless.ucsd.edu
- USERTrust RSA Certification Authority
- USERTrust RSA Certification Authority
- www.schlossbensberg.com
A particular certificate

<table>
<thead>
<tr>
<th>Subject Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country or Region</td>
<td>US</td>
</tr>
<tr>
<td>Postal Code</td>
<td>92093</td>
</tr>
<tr>
<td>State/Province</td>
<td>CA</td>
</tr>
<tr>
<td>Locality</td>
<td>La Jolla</td>
</tr>
<tr>
<td>Street Address</td>
<td>9500 Gilman Drive</td>
</tr>
<tr>
<td>Organization</td>
<td>University of California, San Diego</td>
</tr>
<tr>
<td>Organizational Unit</td>
<td>UCSD</td>
</tr>
<tr>
<td>Common Name</td>
<td>auth.ucsd.edu</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Issuer Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country or Region</td>
<td>US</td>
</tr>
<tr>
<td>State/Province</td>
<td>MI</td>
</tr>
<tr>
<td>Locality</td>
<td>Ann Arbor</td>
</tr>
<tr>
<td>Organization</td>
<td>Internet2</td>
</tr>
<tr>
<td>Organizational Unit</td>
<td>InCommon</td>
</tr>
<tr>
<td>Common Name</td>
<td>InCommon RSA Server CA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serial Number</th>
<th>Version</th>
<th>Signature Algorithm</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B1 26 07 A2 0D 08 E2 27 6E A0 9C 97 47 D0 DF 87</td>
<td>3</td>
<td>SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not Valid Before</th>
<th>Not Valid After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday, April 19, 2018 at 5:00:00 PM Pacific Daylight Time</td>
<td>Sunday, April 19, 2020 at 4:59:59 PM Pacific Daylight Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Key Info</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>RSA Encryption (1.2.840.113549.1.1.1)</td>
</tr>
<tr>
<td>Parameters</td>
<td>None</td>
</tr>
<tr>
<td>Public Key</td>
<td>256 bytes : C4 AD 44 82 D1 A1 84 0F ...</td>
</tr>
<tr>
<td>Exponent</td>
<td>65537</td>
</tr>
<tr>
<td>Key Size</td>
<td>2048 bits</td>
</tr>
<tr>
<td>Key Usage</td>
<td>Encrypt, Verify, Wrap, Derive</td>
</tr>
<tr>
<td>Signature</td>
<td>256 bytes : 41 01 7D F8 D1 80 AC E8 ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extension</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Usage</td>
<td>(2.5.20.15)</td>
</tr>
<tr>
<td>Critical</td>
<td>YES</td>
</tr>
<tr>
<td>Usage</td>
<td>Digital Signature, Key Encipherment</td>
</tr>
</tbody>
</table>
Suppose Bob wishes to revoke his certificate $CERT_{Bob}$, perhaps because his secret key was compromised.

- Bob sends $CERT_{Bob}$ and revocation request to CA
- CA checks that request comes from Bob
- CA puts ($CERT_{Bob}$, Revocation date) on its Certificate Revocation List (CRL)
- This list is disseminated.

Before Alice accepts Bob’s certificate she should check that it is not on the CRL.

The OCSP (Online Certificate Status Protocol) is one-way to do this.
Revocation Issues

- November 22: Bob’s secret key compromised
- November 24: Bob’s $CERT_{Bob}$ revoked
- November 25: Alice sees CRL

In the period Nov. 22-25, $CERT_{Bob}$ might be used and Alice might be accepting as authentic signatures that are really the adversary’s. Also Alice might be encrypting data for Bob which the adversary can decrypt.

In practice, CRLs are large and revocation is a problem.
Certificate Transparency

Home

Google's Certificate Transparency project fixes several structural flaws in the SSL certificate system, which is the main cryptographic system that underlies all HTTPS connections. These flaws weaken the reliability and effectiveness of encrypted Internet connections and can compromise critical TLS/SSL mechanisms, including domain validation, end-to-end encryption, and the chains of trust set up by certificate authorities. If left unchecked, these flaws can facilitate a wide range of security attacks, such as website spoofing, server impersonation, and man-in-the-middle attacks.

Certificate Transparency helps eliminate these flaws by providing an open framework for monitoring and auditing SSL certificates in nearly real time. Specifically, Certificate Transparency makes it possible to detect SSL certificates that have been mistakenly issued by a certificate authority or maliciously acquired from an otherwise unimpeachable certificate authority. It also makes it possible to identify certificate authorities that have gone rogue and are maliciously issuing certificates.

Because it is an open and public framework, anyone can build or access the basic components that drive Certificate Transparency. This is particularly beneficial to Internet security stakeholders, such as domain owners, certificate authorities, and browser manufacturers, who have a vested interest in maintaining the health and integrity of the SSL certificate system.
Keys are stored on keyservers like openpgp SGS.
Mihir Bellare

Shared key setting

\[ M \leftarrow D_K(C) \quad C \quad C \leftarrow E_K(M) \]

\[ \sigma \leftarrow T_K(M) \quad M, \sigma \quad V_K(M, \sigma) \]

Alice and Bob can

- send each other encrypted data
- verify each other’s MACs

Can be preferable to public key setting because computation costs are lower.

But how do Alice and Bob get a shared key?
Diffie-Hellman Key Exchange

Let $G = \langle g \rangle$ be a cyclic group of order $m$ and assume $G, g, m$ are public quantities.

Alice

$x \leftarrow \mathbb{Z}_m; \quad X \leftarrow g^x$

$K \leftarrow Y^x$

Bob

$y \leftarrow \mathbb{Z}_m; \quad Y \leftarrow g^y$

$K \leftarrow X^y$

$Y^x = (g^y)^x = g^{xy} = (g^x)^y = X^y$

This enables Alice and Bob to agree on a common key $K$ which can subsequently be used, say to encrypt:

Alice

$M \leftarrow D_K(C)$

Bob

$C \leftarrow E_K(M)$

Mihir Bellare
Security of DH Key Exchange under Passive Attack

Eavesdropping adversary gets $X = g^x$ and $Y = g^y$ and wants to compute $g^{xy}$. But this is the (presumed hard) CDH problem.

Conclusion: DH key exchange is secure against passive (eavesdropping) attack.
Security of DH Key Exchange under Active Attack

Entity-in-the-middle attack:

\[ E \]

\[
\begin{align*}
E & \leftarrow X \leftarrow g^x \\
K & \leftarrow Y^x \\
M & \leftarrow D_K(C) \\
C & \leftarrow E_K(M)
\end{align*}
\]

Bob

Alice, X

Bob, Y

y \leftarrow Z_m; Y \leftarrow g^y

K \leftarrow X^y

Adversary $E$ impersonates Alice so that:

- Bob thinks he shares $K$ with Alice but $E$ has $K$
- $E$ can now decrypt ciphertexts Bob intends for Alice

Conclusion: DH key exchange is insecure against active attack
When is key agreement possible?

In the presence of an active adversary, it is impossible for Alice and Bob to
  • start from scratch, and
  • exchange messages to get a common key unknown to the adversary

Why? Because there is no way for Bob to distinguish Alice from the adversary.

Alice and Bob need some a priori “information advantage” over the adversary. This typically takes the form of long-lived keys.
Settings and long-lived keys

- Public key setting: $A$ has $pk_B$ and $B$ has $pk_A$
- Symmetric setting: $A$, $B$ share a key $K$
- Three party setting: $A$, $B$ each share a key with a trusted server $S$.

These keys constitute the long-lived information.
Session keys: The “real” key distribution problem

In practice, Alice and Bob will engage in multiple communication “sessions.” For each, they

- First use a session-key distribution protocol, based on their long-lived keys, to get a fresh, authentic session key;
- Then encrypt or authenticate data under this session key for the duration of the session

Why session keys?

- In public-key setting, efficient cryptography compared to direct use of long-lived keys
- Security attributes, in particular enabling different applications to use keys in different ways and not compromise security of other applications
Session key distribution

- Hundreds of protocols
- Dozens of security requirements
- Lots of broken protocols
- Protocols easy to specify and hard to get right
- Used ubiquitously: SSL, TLS, SSH, ...
Question: What is desired security attribute of session key?
Question: What is desired security attribute of session key?

Answer: It should be indistinguishable from random to adversary

At end of protocol:

\[ b \leftarrow \{0, 1\}; \quad \alpha_0 \leftarrow \alpha; \quad \alpha_1 \leftarrow \{0, 1\}^{\|\alpha\|} \]
Session key exchange in public key setting

Most important type of session key exchange in practice, used in all communication security protocols: SSL, SSH, TLS, IPSEC, 802.11, ...
Protocol KE1: Basic Exchange

\[ A^{pk[B]} \]

\[ \xrightarrow{A, R_A} \]

\[ \xleftarrow{R_B, C, B, \text{Sign}_B(A, B, R_A, R_B, C)} \]

\[ \xrightarrow{A, \text{Sign}_A(A, B, R_A, R_B)} \]

\[ B^{pk[A]} \]

\[ C \leftarrow \mathcal{E}_A(K) \]

Session key \( K \) is chosen by \( B \).

\( R_A, R_B \) are random nonces.

\( \text{Sign}_P(M) \) is \( P \)'s signature of \( M \) under \( sk[P] \). It is verifiable given \( pk[P] \).

\( \mathcal{E}_A(\cdot) \) is encryption under \( A \)'s public key \( pk[A] \), decryptable using \( sk[A] \).
Forward secrecy

\[ A^{pk[B]} \]

\[ A, R_A \]

\[ R_B, C, B, \text{Sign}_B(A, B, R_A, R_B, C) \rightarrow \]

\[ A, \text{Sign}_A(A, B, R_A, R_B) \rightarrow \]

\[ C_B \]

\[ B^{pk[A]} \]

\[ C \leftarrow \mathcal{E}_A(K) \]

\[ C_B \leftarrow \mathcal{E}_K(M) \]

Nov. 20: Adversary $E$ records above flows.
Dec. 18: $A$'s, system compromised and $sk[A]$ exposed.
Dec. 19: $A$ revokes $pk[A]$ so that no further damage is done but cannot prevent $E$ from $K \leftarrow D_{sk[A]}(C); M \leftarrow D_K(C_B)$.

Can we achieve forward secrecy: Privacy of communication done prior to exposure of $sk[A]$ is not compromised?
KE2: Adding forward secrecy

Session key is \( K = H(A, B, g^a, g^b, g^{ab}) \).

Adversary \( E \) records above flows on Nov. 20. On Dec. 18, \( sk[A] \) is exposed. This allows \( E \) to forge \( A \)'s signatures, but \( A \) can address this by revoking \( pk[A] \). But \( sk[A] \) does not help \( E \) obtain \( K \).

There is no public-key encryption here, only signatures.

All standard protocols use DH to get forward security in ways like this.
A password is a human-memorizable key.

Attackers are capable of forming a set $D$ of possible passwords called a dictionary such that

- If the target password $pw$ is in $D$ and
- The attacker knows $\overline{pw} = f(pw)$, the image of $pw$ under some public function $f$.

then the target password can be found via

\[
\text{for all } pw' \in D \text{ do}
\]
\[
\text{if } f(pw') = \overline{pw} \text{ then return } pw'
\]

This is called a dictionary attack.
Fact is that in spite of all the great crypto around, a significant fraction of our security today resides in passwords: bank ATM passwords; login passwords; passwords for different websites; ...

Few of us today have cryptographic keys; but we all have more passwords than we can remember!

Passwords are convenient and entrenched.

Preventing dictionary attacks is an important concern.
Systems try to force users to select “good” passwords, meaning ones not in the dictionary. But studies show that a significant fraction of user passwords end up being in the dictionary anyway.

Attackers get better and better at building dictionaries.

Good password selection helps, but it is unrealistic to think that even the bulk of passwords are well selected, meaning not in the dictionary.
Popular passwords

In 2016, the 25 most common passwords made up more than 10% of surveyed passwords, with the most common making up 4%.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>password</td>
<td>password</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
</tr>
<tr>
<td>2</td>
<td>123456</td>
<td>123456</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>password</td>
</tr>
<tr>
<td>3</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
<td>12345</td>
<td>12345678</td>
<td>12345</td>
<td>12345678</td>
<td>12345678</td>
</tr>
<tr>
<td>4</td>
<td>qwerty</td>
<td>abc123</td>
<td>qwerty</td>
<td>12345678</td>
<td>qwerty</td>
<td>12345678</td>
<td>qwerty</td>
<td>12345678</td>
</tr>
<tr>
<td>5</td>
<td>abc123</td>
<td>qwerty</td>
<td>abc123</td>
<td>qwerty</td>
<td>12345</td>
<td>football</td>
<td>12345</td>
<td>12345</td>
</tr>
<tr>
<td>6</td>
<td>monkey</td>
<td>monkey</td>
<td>123456789</td>
<td>123456789</td>
<td>123456789</td>
<td>qwerty</td>
<td>123456789</td>
<td>123456789</td>
</tr>
<tr>
<td>7</td>
<td>1234567</td>
<td>letmein</td>
<td>111111</td>
<td>1234</td>
<td>football</td>
<td>1234567890</td>
<td>letmein</td>
<td>1234567</td>
</tr>
<tr>
<td>8</td>
<td>letmein</td>
<td>dragon</td>
<td>1234567</td>
<td>baseball</td>
<td>1234</td>
<td>1234567</td>
<td>sunshine</td>
<td>1234567</td>
</tr>
<tr>
<td>9</td>
<td>trustno1</td>
<td>111111</td>
<td>iloveyou</td>
<td>dragon</td>
<td>1234567</td>
<td>princess</td>
<td>football</td>
<td>qwerty</td>
</tr>
<tr>
<td>10</td>
<td>dragon</td>
<td>baseball</td>
<td>adobe123[a]</td>
<td>football</td>
<td>baseball</td>
<td>1234</td>
<td>iloveyou</td>
<td>iloveyou</td>
</tr>
<tr>
<td>11</td>
<td>baseball</td>
<td>iloveyou</td>
<td>123123</td>
<td>1234567</td>
<td>welcome</td>
<td>login</td>
<td>admin</td>
<td>princess</td>
</tr>
<tr>
<td>12</td>
<td>111111</td>
<td>trustno1</td>
<td>admin</td>
<td>monkey</td>
<td>1234567890</td>
<td>welcome</td>
<td>welcome</td>
<td>admin</td>
</tr>
<tr>
<td>13</td>
<td>iloveyou</td>
<td>1234567</td>
<td>letmein</td>
<td>abc123</td>
<td>solo</td>
<td>monkey</td>
<td>welcome</td>
<td>666666</td>
</tr>
<tr>
<td>14</td>
<td>master</td>
<td>sunshine</td>
<td>letmein</td>
<td>abc123</td>
<td>111111</td>
<td>abc123</td>
<td>login</td>
<td>666666</td>
</tr>
<tr>
<td>15</td>
<td>sunshine</td>
<td>master</td>
<td>photoshop[a]</td>
<td>111111</td>
<td>1qa2wxa</td>
<td>admin</td>
<td>abc123</td>
<td>abc123</td>
</tr>
<tr>
<td>16</td>
<td>ashley</td>
<td>123123</td>
<td>1234</td>
<td>mustang</td>
<td>dragon</td>
<td>121212</td>
<td>starwars</td>
<td>football</td>
</tr>
<tr>
<td>17</td>
<td>bailey</td>
<td>welcome</td>
<td>monkey</td>
<td>access</td>
<td>master</td>
<td>flower</td>
<td>123123</td>
<td>123123</td>
</tr>
<tr>
<td>18</td>
<td>passwdWord</td>
<td>shadow</td>
<td>shadow</td>
<td>shadow</td>
<td>monkey</td>
<td>passw0rd</td>
<td>dragon</td>
<td>monkey</td>
</tr>
<tr>
<td>19</td>
<td>shadow</td>
<td>ashley</td>
<td>sunshine</td>
<td>master</td>
<td>letmein</td>
<td>dragon</td>
<td>passw0rd</td>
<td>654321</td>
</tr>
<tr>
<td>20</td>
<td>123123</td>
<td>football</td>
<td>12345</td>
<td>michael</td>
<td>login</td>
<td>sunshine</td>
<td>master</td>
<td>l@#$%^&amp;*</td>
</tr>
<tr>
<td>21</td>
<td>654321</td>
<td>jesus</td>
<td>password1</td>
<td>superman</td>
<td>princess</td>
<td>master</td>
<td>hello</td>
<td>charlie</td>
</tr>
<tr>
<td>22</td>
<td>superman</td>
<td>michael</td>
<td>princess</td>
<td>696696</td>
<td>qwertyuiop</td>
<td>hottie</td>
<td>freedom</td>
<td>aa123456</td>
</tr>
<tr>
<td>23</td>
<td>qazwxs</td>
<td>ninja</td>
<td>azerty</td>
<td>123123</td>
<td>solo</td>
<td>loveme</td>
<td>whatever</td>
<td>donald</td>
</tr>
<tr>
<td>24</td>
<td>michael</td>
<td>mustang</td>
<td>trustno1</td>
<td>batman</td>
<td>passw0rd</td>
<td>zaq1zaq1</td>
<td>qazwxs</td>
<td>password1</td>
</tr>
<tr>
<td>25</td>
<td>Football</td>
<td>password1</td>
<td>000000</td>
<td>trustno1</td>
<td>starwars</td>
<td>password1</td>
<td>trustno1</td>
<td>qwerty123</td>
</tr>
</tbody>
</table>
An alternative approach is to ensure that usage of a password $pw$ never reveals an image $\overline{pw} = f(pw)$ of $pw$ under a public function $f$. Then, even if the password is in the dictionary, the dictionary attack cannot be mounted.

**Password-based session-key exchange:**

- $A, B$ share a password $pw$.
- They want to interact to get a common session key.
- The protocol should resist dictionary attack: adversary should be unable to obtain an image of $pw$ under a public function.
Doesn’t prevent dictionary attacks

\[ \begin{align*}
A^{pw} & \xrightarrow{A, g^a} \sigma \xrightarrow{B, g^b, MAC_{pw}(1, A, B, g^a, g^b)} \xleftarrow{A, MAC_{pw}(0, A, B, g^a, g^b)} B^{pw}
\end{align*} \]

Session key is \( K = H(A, B, g^a, g^b, g^{ab}) \).

**Dictionary attack is possible:** Let \( f \) be defined by

\[ f(x) = MAC_x(1, A, B, g^a, g^b) \]

Then, letting \( D \) be the dictionary of candidate passwords, get the target password \( pw \) via

for all \( pw' \in D \) do

if \( f(pw') = \sigma \) then return \( pw' \)
Protocol KE4: [BPR00]

\[ A^{pw} \]

\[ A, E_{pw}(g^a) \]

\[ B, E_{pw}(g^b), H(1, A, B, g^a, g^b, g^{ab}) \]

\[ A, H(2, A, B, g^a, g^b, g^{ab}) \]

\[ B^{pw} \]

\[ E : PW \times G \rightarrow G \] is a block cipher over group \( G \) and key space \( PW \) of all possible passwords; the session key is \( K = H(0, A, B, g^a, g^b, g^{ab}) \).

This has been proven secure against dictionary attack [BPR00].