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 $pk[B]$.

$$\begin{array}{lcl} \text{Alice}^{pk[B]} & & \text{Bob} \\ C \xleftarrow{\$} \mathcal{E}_{pk[B]}(M) & \xrightarrow{\ C\ } & M \leftarrow \mathcal{D}_{sk[B]}(C) \\ \mathcal{V}_{pk[B]}(M,\sigma) & \xleftarrow{\ M,\sigma\ } & \sigma \xleftarrow{\$} \mathcal{S}_{sk[B]}(M) \end{array}$$

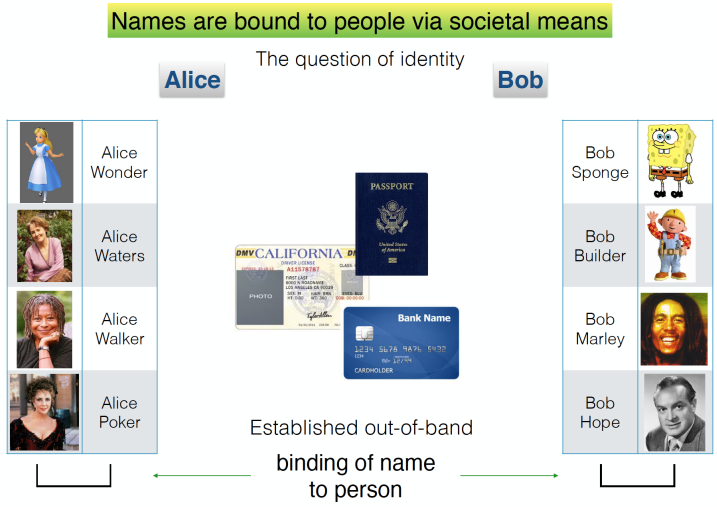
Now Alice can encrypt a message M under $pk[B]$ to get a ciphertext C that Bob can decrypt using $sk[B]$.

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

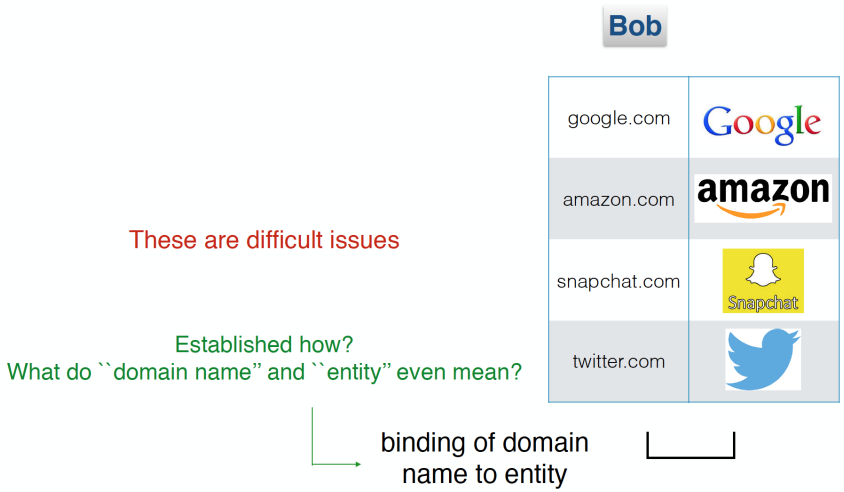
But how does Alice get $pk[B]$?

But who exactly are "Alice" and "Bob"?

A question of **identity** that is central to key distribution and its security.



In TLS, Bob is a server



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.

Alice		Bob
	$\xleftarrow{Bob, pk[B]}$	$(pk[B], sk[B]) \xleftarrow{\$} \mathcal{K}$

This enables:

$C \xleftarrow{\$} \mathcal{E}_{pk[B]}(M)$	$\xrightarrow{C}$	$M \leftarrow \mathcal{D}_{sk[B]}(C)$
$\mathcal{V}_{pk[B]}(M, \sigma)$	$\xleftarrow{M, \sigma}$	$\sigma \xleftarrow{\$} \mathcal{S}_{sk[B]}(M)$

Entity-in-the-middle attack

Alice		Adversary A
	$\xleftarrow{Bob, pk[A]}$	$(pk[A], sk[A]) \xleftarrow{\$} \mathcal{K}$

So:

$C \xleftarrow{\$} \mathcal{E}_{pk[A]}(M)$	$\xrightarrow{C}$	$M \leftarrow \mathcal{D}_{sk[A]}(C)$
$\mathcal{V}_{pk[A]}(M, \sigma)$	$\xleftarrow{M, \sigma}$	$\sigma \xleftarrow{\$} \mathcal{S}_{sk[A]}(M)$

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.

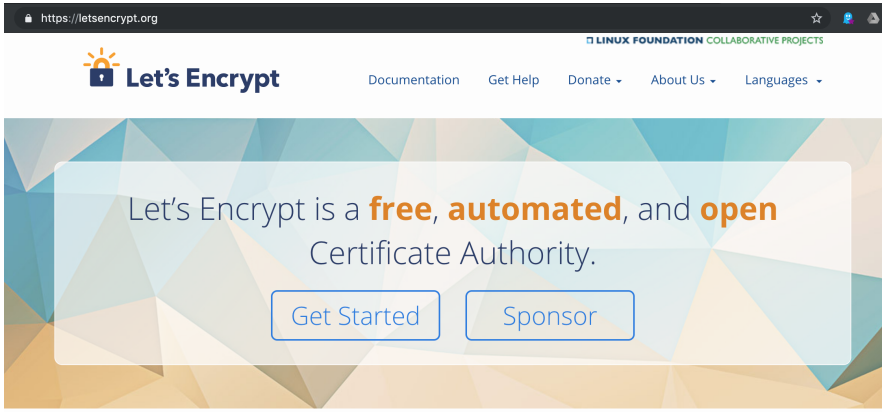
Popular Solution: The PKI (Public Key Infrastructure).

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



Let's Encrypt

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

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

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*

Key generation

Bob locally runs a prescribed key-generation algorithm to generate his keys: $(pk, sk) \xleftarrow{\$} \mathcal{K}$

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
```


Key generation Example: EC Key generation with `openssl`

Generating a private EC key

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

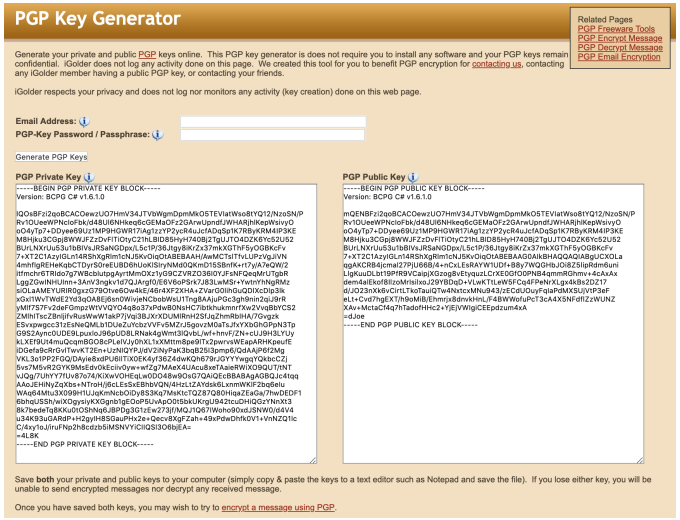
```
$ openssl ecparam -name prime256v1 -genkey -noout -out key.pem
```

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.

Key generation Example: On-line key generation tools



Checks

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 σ 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.

Certificate usage

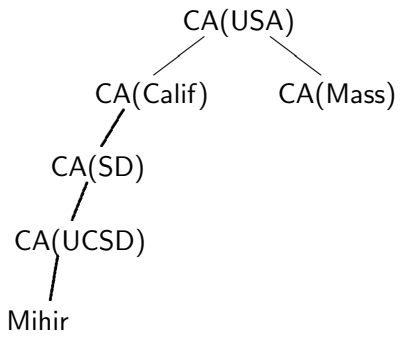
Bob can send $CERT_{Bob}$ to Alice who is assumed to have the CA's public key $pk[CA]$ and now will:

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

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



$\underline{CERT_{Mihir}}$

$CERT[CA(USA) : CA(Calif)]$
$CERT[CA(Calif) : CA(SD)]$
$CERT[CA(SD) : CA(UCSD)]$
$CERT[CA(UCSD) : Mihir]$

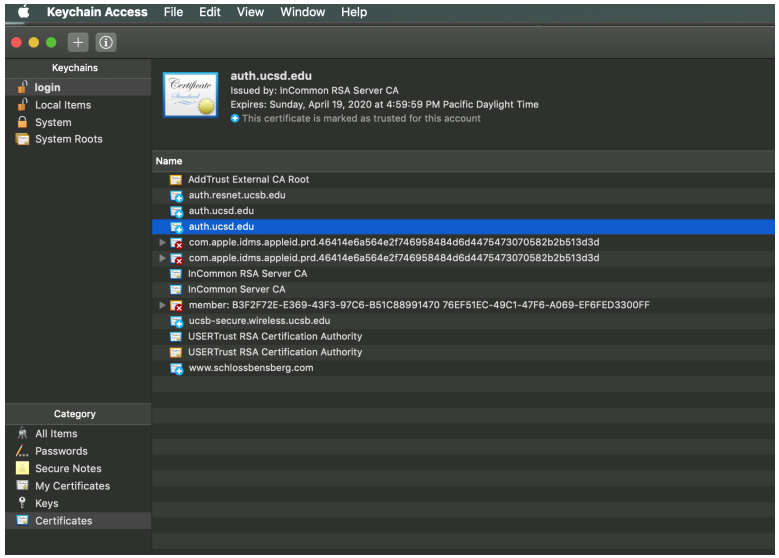
$CERT[X : Y] = (pk[Y], Y, \ldots, \mathcal{S}_{sk[X]}(pk[Y], Y, \ldots))$

To verify $CERT_{Mihir}$ you need only $pk_{CA[USA]}$.

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`



A particular certificate

Revocation

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.

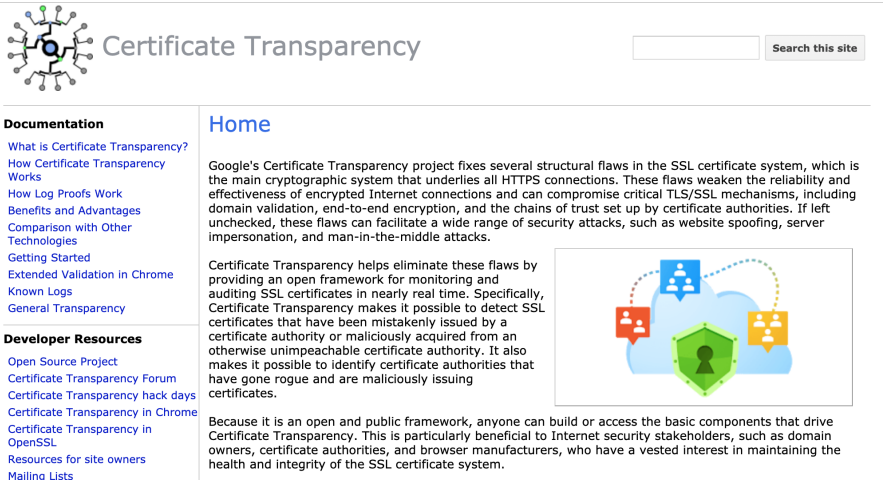
Revocation Issues

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

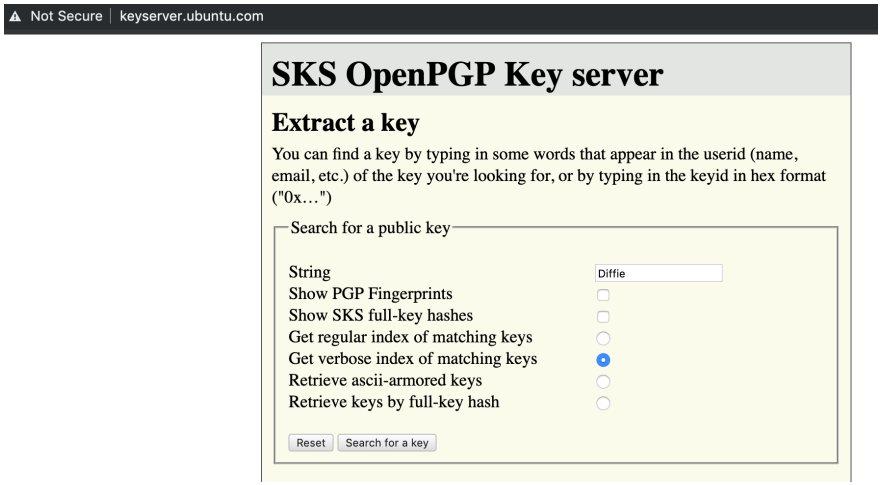
In the period Nov. 22-25, $CERT_{Bob}$ might be used and Alice might be accepting as authentic signatures that are really the adversary's. Also Alice might be encrypting data for Bob which the adversary can decrypt.

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

Certificate transparency



Dissemination of public key via server



Search on SGS key-server

Search results for 'diffie'

```
Type bits/keyID     cr. time   exp time   key expir
------------------------------------------------------------------------------

pub  3072R/B8760337 2016-05-24

uid Joe Diffie <bigmoondog187@yahoo.com>
sig  sig3  B8760337 2016-05-24 _________ _________ [selfsig]

sub  3072R/89B91C8B 2016-05-24
sig sbind  B8760337 2016-05-24 _________ _________ [ ]

sub  3072R/C1A6676E 2016-05-24
sig sbind  B8760337 2016-05-24 _________ _________ [ ]

------------------------------------------------------------------------------

pub  2048D/186FE65A 2015-06-20

uid Manuel Rachad Haj-Saleh Ramírez (Clave Diffie-Hellman) <manuel@hajsaleh.es>
sig  sig3  186FE65A 2015-06-20 _________ 2019-06-20 [selfsig]

sub  2048g/C4FE1190 2015-06-20
sig sbind  186FE65A 2015-06-20 _________ 2019-06-20 [ ]

------------------------------------------------------------------------------

pub  1024D/E1CF6414 2014-12-20

uid diffie-hell-test2048 <diffie-hell-test2048@test.cn>
sig  sig   E1CF6414 2014-12-20 _________ _________ [selfsig]

sub  2048g/9709418D 2014-12-20
sig sbind  E1CF6414 2014-12-20 _________ _________ [ ]

------------------------------------------------------------------------------

pub  2048R/48EF2F1F 2012-08-21

uid diffie <diffie@mailvault.com>
sig  sig3  48EF2F1F 2012-08-21 _________ _________ [selfsig]

sub  2048R/5BE9521D 2012-08-21
sig sbind  48EF2F1F 2012-08-21 _________ _________ [ ]

------------------------------------------------------------------------------
```


Shared key setting

$$\begin{array}{lcl} \text{Alice}^K & & \text{Bob}^K \\ M \leftarrow \mathcal{D}_K(C) & \xleftarrow{\quad C \quad} & C \xleftarrow{\$} \mathcal{E}_K(M) \\ \sigma \xleftarrow{\$} \mathcal{T}_K(M) & \xrightarrow{\quad M, \sigma \quad} & \mathcal{V}_K(M, \sigma) \end{array}$$

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.

$$\begin{array}{rcl} \text{Alice} & & \text{Bob} \\ x \xleftarrow{\$} \mathbf{Z}_m;\ X \leftarrow g^x & \xrightarrow{\quad \text{Alice}, X \quad} & \\ K \leftarrow Y^x & \xleftarrow{\quad \text{Bob}, Y \quad} & y \xleftarrow{\$} \mathbf{Z}_m;\ Y \leftarrow g^y \\ & & K \leftarrow X^y \end{array}$$

$$Y^x = (g^y)^x = \underbrace{g^{xy}}_{K} = (g^x)^y = X^y$$

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

$$\begin{array}{rcl} \text{Alice} & & \text{Bob} \\ M \leftarrow \mathcal{D}_K(C) & \xleftarrow{\quad C \quad} & C \xleftarrow{\$} \mathcal{E}_K(M) \end{array}$$

Security of DH Key Exchange under Passive Attack

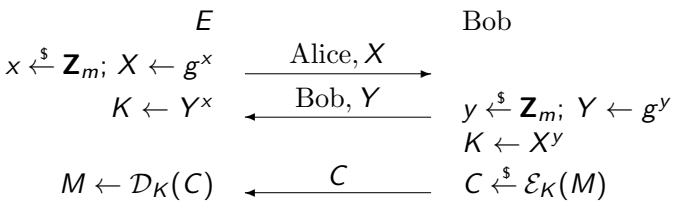
$$\begin{array}{rcl} \text{Alice} & & \text{Bob} \\ x \xleftarrow{\$} \mathbf{Z}_m;\ X \leftarrow g^x & \xrightarrow{\quad \text{Alice}, X \quad} & \\ K \leftarrow Y^x & \xleftarrow{\quad \text{Bob}, Y \quad} & y \xleftarrow{\$} \mathbf{Z}_m;\ Y \leftarrow g^y \\ & & K \leftarrow X^y \end{array}$$

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:



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

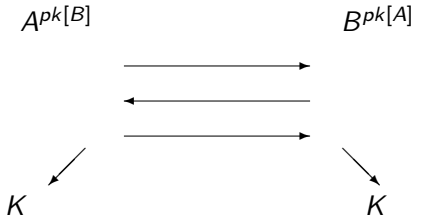
Why session keys?

- 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, ...

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

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

$$A, R_A \rightarrow$$
$$\leftarrow R_B, C, B, \mathrm{Sign}_B(A, B, R_A, R_B, C) \quad C \xleftarrow{\$} \mathcal{E}_A(K)$$
$$A, \mathrm{Sign}_A(A, B, R_A, R_B) \rightarrow$$

Session key K is chosen by B.

R_A, R_B are random nonces.

$\mathrm{Sign}_P(M)$ 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 $sk[A]$.

Beyond KE2

Modern session key exchange protocols enhance KE2 to incorporate many other security features including

- **Forward security:** Exposure of long-lived secret keys should not compromise PRIOR session keys
- **Anonymity:** Eavesdropper cannot figure out identity (public key) of client authenticating to some server.

Passwords

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

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

This is called a dictionary attack.

Password usage

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.

Preventing dictionary attacks: Password selection

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%.

Top 25 most common passwords by year according to SplashData

Rank	2011[4]	2012[5]	2013[6]	2014[7]	2015[8]	2016[3]	2017[9]	2018[10]
1	password	password	123456	123456	123456	123456	123456	123456
2	123456	123456	password	password	password	password	password	password
3	12345678	12345678	12345678	12345	12345678	12345	12345678	123456789
4	qwerty	abc123	qwerty	12345678	qwerty	12345678	qwerty	12345678
5	abc123	qwerty	abc123	qwerty	12345	football	12345	12345
6	monkey	monkey	123456789	123456789	123456789	qwerty	123456789	111111
7	1234567	letmein	111111	1234	football	1234567890	letmein	1234567
8	letmein	dragon	1234567	baseball	1234	1234567	1234567	sunshine
9	trustno1	111111	iloveyou	dragon	1234567	princess	football	qwerty
10	dragon	baseball	adobe123[a]	football	baseball	1234	iloveyou	iloveyou
11	baseball	iloveyou	123123	1234567	welcome	login	admin	princess
12	111111	trustno1	admin	monkey	1234567890	welcome	welcome	admin
13	iloveyou	1234567	1234567890	letmein	abc123	solo	monkey	welcome
14	master	sunshine	letmein	abc123	111111	abc123	login	666666
15	sunshine	master	photoshop[a]	111111	1qaz2wsx	admin	abc123	abc123
16	ashley	123123	1234	mustang	dragon	121212	starwars	football
17	bailey	welcome	monkey	access	master	flower	123123	123123
18	passw0rd	shadow	shadow	shadow	monkey	passw0rd	dragon	monkey
19	shadow	ashley	sunshine	master	letmein	dragon	passw0rd	654321
20	123123	football	12345	michael	login	sunshine	master	!@#$%^&*
21	654321	jesus	password1	superman	princess	master	hello	charlie
22	superman	michael	princess	696969	qwertyuiop	hottie	freedom	aa123456
23	qazwsx	ninja	azerty	123123	solo	loveme	whatever	donald
24	michael	mustang	trustno1	batman	passw0rd	zaq1zaq1	qazwsx	password1
25	Football	password1	000000	trustno1	starwars	password1	trustno1	qwerty123

Preventing dictionary attacks: avoiding image revelation

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.