

KEY DISTRIBUTION: PKI and SESSION-KEY EXCHANGE

The public key setting

B 's secret key is $sk[B]$ and its associated public key is $pk[B]$. The public key setting **assumes** A is in possession of $pk[B]$.

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

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

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

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

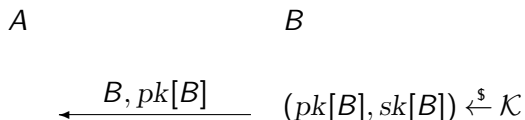
But who exactly are “A” and “B”?

Typically, as in most uses of TLS, B is a server. Its identity B is an associated domain name or ip address, for example $B = \text{google.com}$.

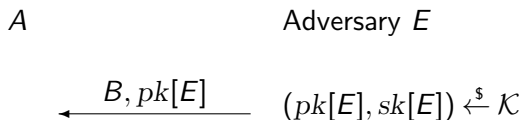
A is a client, also with an associated ip address.

How does A get B 's public key?

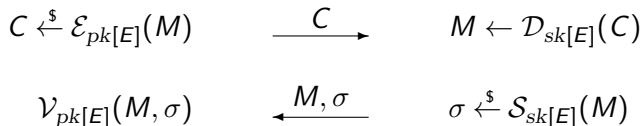
How about: B runs a prescribed key-generation algorithm \mathcal{K} to generate $(pk[B], sk[B])$. It sends $(B, pk[B])$ to A .



Entity-in-the-middle attack



So:



Adversary E can decrypt ciphertexts intended for B and can forge B 's signatures. Adversary effectively becomes B .

Goal: A gets an **authentic** copy of B 's public key, meaning if pk claims to come from B , then A has a 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: B could post its public key on its Facebook; post it on its personal or corporate webpage; include it as an attachment in its emails; put it on a keyserver like openpgp SKS; hand it to A in person; ...

Let's Encrypt



The screenshot shows the homepage of the Let's Encrypt website. At the top, the browser address bar displays "https://letsencrypt.org". The navigation menu includes "Documentation", "Get Help", "Donate", "About Us", and "Languages". The main content area features a large banner with a geometric background. The banner text reads: "Let's Encrypt is a **free, automated, and open** Certificate Authority." Below this text are two buttons: "Get Started" and "Sponsor".

https://letsencrypt.org

LINUX FOUNDATION COLLABORATIVE PROJECTS

Let's Encrypt

Documentation Get Help Donate About Us Languages

Let's Encrypt is a **free, automated, and open** Certificate Authority.

Get Started Sponsor

Some other 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

- B generates $(pk, sk) \stackrel{\$}{\leftarrow} \mathcal{K}$ by running a key-generation algorithm \mathcal{K}
- B sends its identity B , and pk , to CA
- CA does identity check to ensure pk is B 's
- B proves knowledge of sk to CA
- CA issues certificate to B
- B sends certificate to A
- A verifies certificate and extracts B 's public key pk

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.

B sends its identity B (domain name, ip address, email address, ...) and its public key pk to the certificate authority (CA).

Upon receiving (B, pk) the CA performs some checks to ensure pk is really B 's key.

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

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

Example: If B 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 B sign or decrypt something under sk to ensure that B knows sk . This ensures B has not copied someone else's public key.

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

$$\text{CERT}[B] = (\text{CERTDATA}, \sigma),$$

where σ is the CA's signature on CERTDATA, computed under the CA's secret key $sk[\text{CA}]$, and CERTDATA contains:

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

The certificate $\text{CERT}[B]$ is returned to B .

Certificate usage

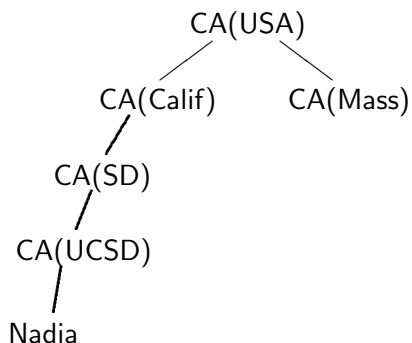
B can send $\text{CERT}[B]$ to A , who is assumed to have the CA's public key $pk[\text{CA}]$, and now will:

- Parse $\text{CERT}[B]$ as $(\text{CERTDATA}, \sigma) \leftarrow \text{CERT}[B]$
- Check that $\mathcal{V}_{pk[\text{CA}]}(\text{CERTDATA}, \sigma) = 1$
- Extract $(pk, B, \text{expiry}, \dots) \leftarrow \text{CERTDATA}$
- Check certificate has not expired
- Check that B is the desired identity
- ...

If all is well, A accepts the certificate and is ready to use the public key pk therein.

How does A get $pk[\text{CA}]$? CA public keys are embedded in software such as your browser, or, on Apple, in the keychain.

Certificate hierarchies



CERT[Nadia]

CERT[CA(USA) : CA(Calif)]

CERT[CA(Calif) : CA(SD)]

CERT[CA(SD) : CA(UCSD)]

CERT[CA(UCSD) : Nadia]

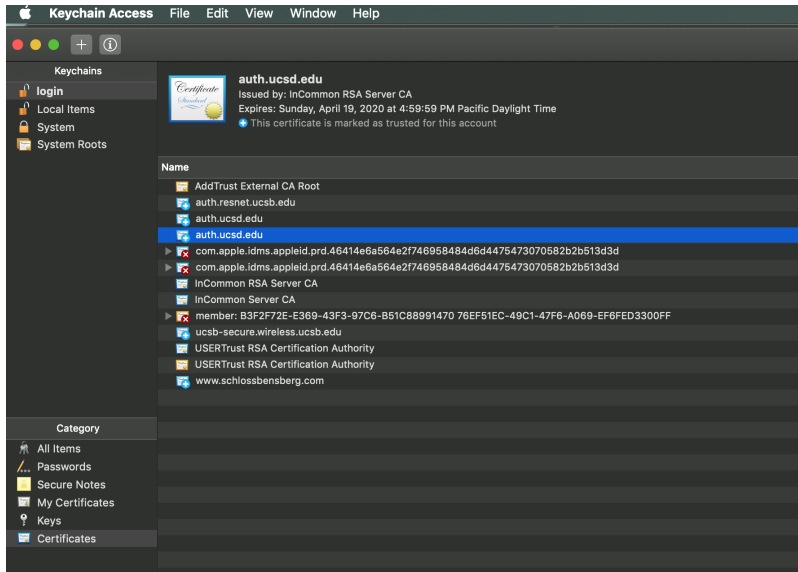
$\text{CERT}[X : Y] = ((pk[Y], Y, \dots), \mathcal{S}_{sk[X]}((pk[Y], Y, \dots)))$

To verify CERT[Nadia] you need only $pk[CA[USA]]$.

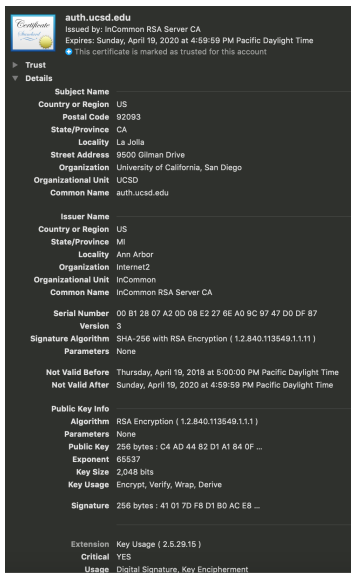
Why certificate hierarchies?


- It is easier for CA(UCSD) to check Nadia's identity (and issue a certificate) than for CA(USA) since Nadia is on UCSD's payroll and UCSD already has a lot of information about her.
- 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



 **auth.ucsd.edu**
Issued by: InCommon RSA Server CA
Expires: Sunday, April 19, 2020 at 4:59:59 PM Pacific Daylight Time
• This certificate is marked as trusted for this account

▶ Trust

▼ Details

Subject Name

Country or Region 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 auth.ucsd.edu

Issuer Name

Country or Region US
State/Province MI
Locality Ann Arbor
Organization Internet2
Organizational Unit InCommon
Common Name InCommon RSA Server CA

Serial Number 00 B1 28 07 A2 0D 08 E2 27 6E A0 9C 97 47 D0 DF 87
Version 3

Signature Algorithm SHA-256 with RSA Encryption (1.2.840.113549.1.1.1)
Parameters None

Not Valid Before Thursday, April 19, 2018 at 5:00:00 PM Pacific Daylight Time
Not Valid After Sunday, April 19, 2020 at 4:59:59 PM Pacific Daylight Time

Public Key Info

Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters None
Public Key 256 bytes : C4 AD 44 82 D1 A1 84 0F ...
Exponent 65537
Key Size 2,048 bits
Key Usage Encrypt, Verify, Wrap, Derive

Signature 256 bytes : 41 01 7D F8 D1 80 AC E8 ...

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

Suppose B wishes to revoke its certificate $\text{CERT}[B] = (\text{CERTDATA}, \sigma)$, perhaps because its secret key sk , corresponding to the pk in CERTDATA , was compromised. Then:

- B sends $\text{CERT}[B]$ and revocation request to CA, signed under sk
- CA verifies the signature under pk
- CA puts $(\text{CERT}[B], \text{RevocationDate})$ on its Certificate Revocation List (CRL)
- This list is disseminated.

Before A accepts B 's certificate, A should check that it is not on the CRL. The OCSP (Online Certificate Status Protocol) is one way to do this.

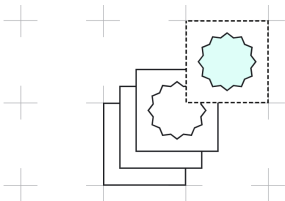
Revocation Issues

- November 22: B 's secret key compromised
- November 24: B 's $CERT[B]$ revoked
- November 25: A sees CRL

$CERT[B]$ might be used in the November 22-25 range, compromising security.

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

Certificate transparency



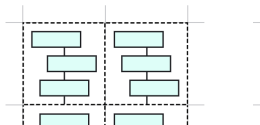
Who watches the watchers?

Historically, user agents determined if CAs were trustworthy through audits by credentialled third parties. But these tended to look at operational practices and historical performance rather than technical correctness. Such audits can't catch everything. Before CT, there could be a significant time lag between a certificate being wrongly issued, and a CA doing something about it.

That's where
Certificate Transparency
comes in.

Independent, reliable logs

CT depends on independent, reliable logs because it is a distributed ecosystem. Built using Merkle trees, logs are publicly verifiable, append-only, and tamper-proof.



SKS OpenPGP Key server

Extract a key

You can find a key by typing in some words that appear in the userid (name, email, etc.) of the key you're looking for, or by typing in the keyid in hex format ("0x...")

Search for a public key

String	<input type="text" value="0xDEADBEEF"/>
Show PGP Fingerprints	<input type="checkbox"/>
Show SKS full-key hashes	<input type="checkbox"/>
Get regular index of matching keys	<input type="radio"/>
Get verbose index of matching keys	<input checked="" type="radio"/>
Retrieve ascii-armored keys	<input type="radio"/>
Retrieve keys by full-key hash	<input type="radio"/>

Submit a key

You can submit a key by simply pasting in the ASCII-armored version of your key and clicking on submit.

SKS is a new [OpenPGP](#) keyserver. The main innovation of SKS is that it includes a highly-efficient reconciliation algorithm for keeping the keyserver synchronized.

[SKS statistics](#)

Session key exchange

A large part of secure communication over the Internet is through protocols like TLS (`https`).

Here, public-key cryptography is not used to directly secure data.

Rather, public-key cryptography is used in a *session-key exchange* that provides (client) A and (server) B with a shared (symmetric) *session key* K .

Data is then secured under K using an authenticated encryption scheme $\mathcal{AE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$:

$$\begin{array}{ccc} A^K & & B^K \\ M \leftarrow \mathcal{D}_K(C) & \xleftarrow{C} & C \xleftarrow{\$} \mathcal{E}_K(M) \end{array}$$

Session key exchange

Why session keys, as opposed to directly securing data with public-key cryptography?

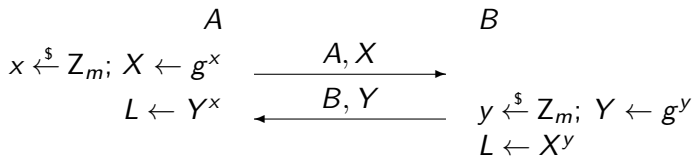
One reason is performance: symmetric cryptography is more efficient than asymmetric cryptography.

More fundamentally, it reflects the Internet architecture in which A and B will engage in multiple, sometimes concurrent communication sessions.

The session key exchange paradigm gives each such session a *fresh* session key, making its security independent of that of other sessions.

Recall Diffie-Hellman Key Exchange

Let $G = \langle g \rangle$ be a cyclic group of order m in which the CDH problem is hard. Let $H: \{0, 1\}^* \rightarrow \{0, 1\}^k$ be a hash function.

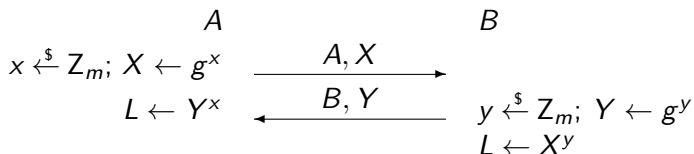


$$Y^x = (g^y)^x = \underbrace{g^{xy}}_L = (g^x)^y = X^y$$

This enables A and B to agree on the common k -bit key $K = H(L) = H(g^{xy})$.

So is this a suitable session key exchange protocol? Are we done?

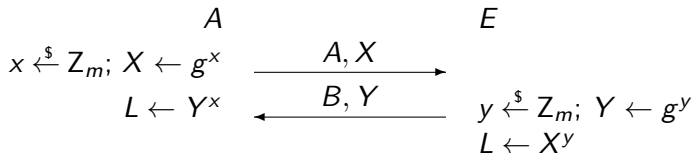
DH Key Exchange is secure under Passive Attack



A passive adversary is one that observes the communication, acquiring $X = g^x$ and $Y = g^y$, and wants to compute $K = H(g^{xy})$. But to do so requires solving the CDH problem, which is here assumed hard.

DH Key Exchange is insecure under Active Attack

Entity-in-the-middle attack:



Adversary E impersonates B . A thinks it shares $K = H(L)$ with B , but in fact A shares K with E .

If A now encrypts, under K , a message intended for B , then E can decrypt the ciphertext and recover the message.

So DH key exchange does not solve the session key exchange problem. However, we will see that it will be a useful tool in achieving forward security in session key exchange ...

Session key exchange requirements

We consider the unilateral, public-key setting. Here B has a certificate $\text{CERT}[B]$ and corresponding public and secret keys $pk[B], sk[B]$. A is not assumed to have a certificate or corresponding keys.

This is the most common setting for TLS, where B is a server like `google.com` and A is a client.

The session key exchange should result in a session key K , known to both A and B , and satisfying:

- Authenticity: A really shares K with B , not some other entity
- Secrecy: The adversary does not know K .

This must hold even if the adversary knows session keys of other sessions and is active, meaning in complete control of the communication.

These basic requirements are supplemented by various others including forward secrecy, anonymity, ...

Session key exchange secrecy

Secrecy: The adversary E cannot distinguish the true session key K from a random string of the same length.

Suppose the protocol terminates and a party X outputs a session key K . Now we let

$$b \xleftarrow{\$} \{0, 1\}; K_1 \leftarrow K; K_0 \xleftarrow{\$} \{0, 1\}^{|K|}; b' \leftarrow E(K_b)$$

Then the adversary's advantage $2 \Pr[b = b'] - 1$ should be small.

This must hold even if the adversary has obtained the session key of all other instances except the one partnered with X , and when the adversary is active, in charge of all communication.

Warning: This is not a formal definition, just a glimpse of it.

Session key exchange landscape

Session-key exchange is a subtle problem.

Easy to specify protocols, hard to get them right.

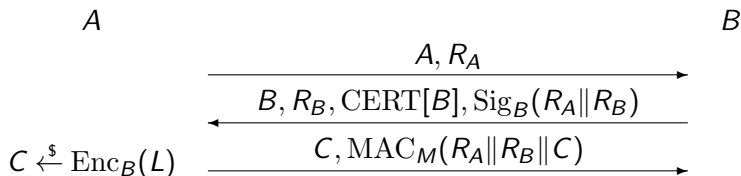
Many security requirements, many proposed protocols, many attacks.

Definitions and provable security treatment started with [BR93] and continued with [BCK98,BPR00,CK01,CK02,...].

Today, standards look for proof-based support.

The TLS 1.3 session key exchange protocol is based on the Sigma protocol of [Kr03].

Protocol KE1



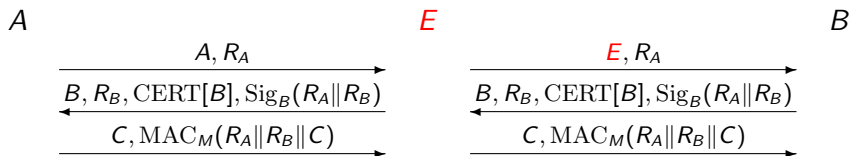
R_A, R_B , called *nonces*, are randomly chosen by the parties.

$\text{Sig}_B(X)$ is B 's signature on X , computed under $sk[B]$ and verifiable under the $pk[B]$ that is in $\text{CERT}[B]$.

L is randomly chosen by A . Session key is $K = H_1(L)$ and MAC key is $M = H_2(L)$ where H_1, H_2 are public hash functions.

$\text{Enc}_B(L)$ is encryption of L under B 's public key $pk[B]$. Decryption uses $sk[B]$.

Identity mis-binding attack on KE1



A accepts B and thinks it shares K with B .

But B accepts E and thinks it shares K with E .

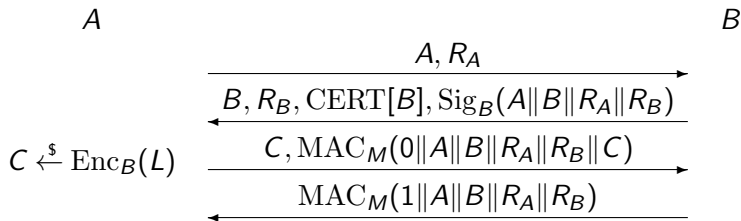
This is viewed as a problem, even though E does not know K , because there is a mis-binding of identities.

A good definition would view this as a successful attack.

A good protocol should ensure that if A accepts B with K , then B either accepts A with K , or accepts nobody with K or a key related to K .

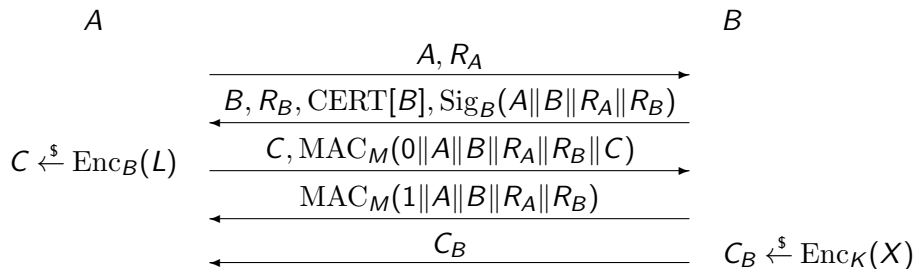
Protocol KE2

Identity mis-binding is circumvented by inclusion of identities in the signature and the MAC, and addition of a MAC from the server:



Session key is $K = H_1(A\|B\|R_A\|R_B\|L)$ and MAC key is $M = H_2(A\|B\|R_A\|R_B\|L)$.

KE2 is not forward secure



Nov. 20: Adversary E records above flows.

Dec. 18: E compromises B 's system and obtains $sk[B]$

Dec. 19: B revokes $\text{CERT}[B]$, and thus $pk[B]$

However, at any time after Dec. 18, E can obtain session key K and decrypt C_B to obtain X via: $K \leftarrow \text{Dec}_{sk[B]}(C)$; $X \leftarrow \text{Dec}_K(C_B)$.

This is a violation of what's called *forward secrecy*.

Forward secrecy asks that exposure of $sk[B]$ does not allow recovery of session keys K exchanged prior to the time of exposure.

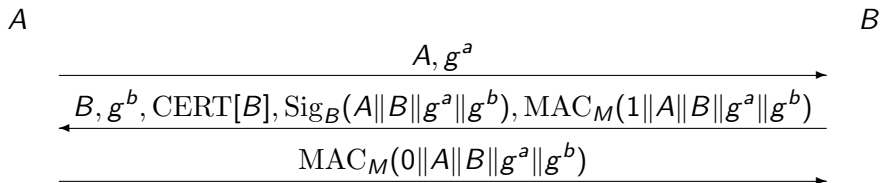
This is achieved using the DH key exchange inside the session key exchange protocol.

Forward secrecy is considered necessary in modern session key exchange, and is present in the TLS 1.3 protocol.

Session-key exchange protocols using DH for forward secrecy are often called authenticated DH key exchange protocols.

Protocol KE3

Let $G = \langle g \rangle$ be a cyclic group of order m in which the CDH problem is hard.

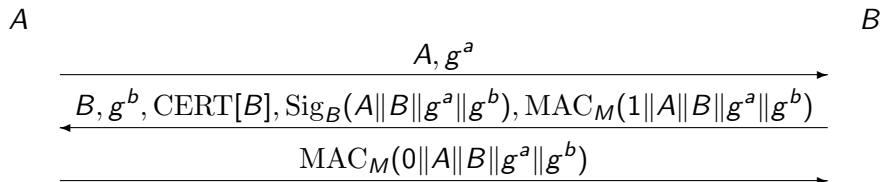


Here $a, b \xleftarrow{\$} Z_m$ are chosen by A, B , respectively, and g^a, g^b play the role of nonces.

$\text{Sig}_B(X)$ is B 's signature on X , computed under $sk[B]$ and verifiable under the $pk[B]$ that is in $\text{CERT}[B]$.

Let $L = g^{ab}$ be the DH key. Then session key is $K = H_1(A||B||g^a||g^b||L)$ and MAC key is $M = H_2(A||B||g^a||g^b||L)$ where H_1, H_2 are as before.

Protocol KE3



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

Compromise of $sk[B]$ only gives E the ability to forge signatures. Even given $sk[B]$, it cannot recover the DH key $L = g^{ab}$ from a prior exchange, and thus cannot distinguish from random the session key $K = H_1(A\|B\|g^a\|g^b\|L)$.

Accordingly this provides forward secrecy.

This is roughly the core of the unilateral session-key exchange in the TLS 1.3 handshake. It is based on Sigma [Kr03].

A password is a human-memorizable key.

Attackers can form a set D of possible passwords called a dictionary such that

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

then the target password pwd can be found via:

For all $\text{pwd}' \in D$ do

 If $f(\text{pwd}') = \overline{\text{pwd}}$ then return pwd'

This is called a dictionary, or brute-force, attack.

Password usage

Passwords are in widespread use for client authentication to Internet services and servers like gmail, Amazon, Internet banking, ...

Most of us have more passwords than we can remember.

Passwords are communicated over TLS. The main threat is dictionary attacks arising from the adversary obtaining the image $\overline{\text{pwd}} = f(\text{pwd})$ of the target password pwd under some public function f .

Studies show that many users select poor passwords, meaning ones that fall into attacker dictionaries. And attackers get better and better at making dictionaries. So preventing dictionary attacks is important for security.

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	lovrme	whatever	donald
24	michael	mustang	trustno1	batman	passw0rd	zaq1zaq1	qazwsx	password1
25	Football	password1	000000	trustno1	starwars	password1	trustno1	qwerty123

A protocol for Password Authenticated Key Exchange (PAKE) assumes client A has a password pwd and server B has either pwd or its hash under a public hash function.

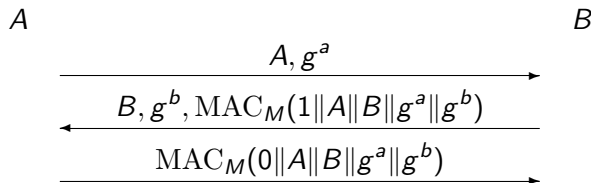
The parties interact to arrive at a common session key K satisfying authenticity, secrecy, forward secrecy and also *security against off-line dictionary attacks*.

This means the protocol never reveals an image $\overline{\text{pwd}} = f(\text{pwd})$ of pwd under a public function f . So even if the password is in the dictionary, the off-line dictionary attack is infeasible.

Roughly, one adversary interaction with one of the parties can eliminate at most one candidate password from the dictionary.

Authentication here is mutual, and no PKI / certificates are assumed.

Protocol KE4

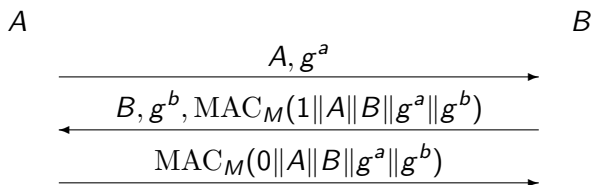


Client A has password pwd that is known to server B .

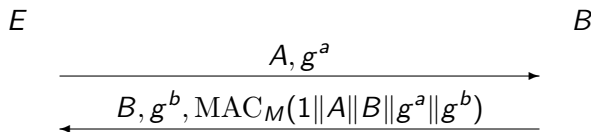
Let $L = g^{ab}$ be the DH key. Then the session key and MAC keys are $K = H_1(A\|B\|g^a\|g^b\|L\|\text{pwd})$ and $M = H_2(A\|B\|g^a\|g^b\|L\|\text{pwd})$, respectively.

Is this secure against dictionary attack?

Protocol KE4



A successful dictionary attack by adversary E is possible, as follows:



E has A, B, g^a, g^b and also $L = g^{ab} = (g^b)^a$. Let

$$f(\text{pwd}) = \text{MAC}_{\text{H}_2}(A\|B\|g^a\|g^b\|L\|\text{pwd})(A\|B\|g^a\|g^b).$$

This f is a public function of the password, allowing E to mount the dictionary attack.

History and status of PAKE

The first protocols were by Bellare and Merritt, 1992.

Definitions and proven-secure protocols begin with [BPR00].

Large literature.

A representative modern PAKE protocol is OPAQUE [JKX18].