A Mini-Course on the Theory of Cryptography

Charles Rackoff

Department of Computer Science, University of Toronto

Bonn, June 17-21, 2006

(Part 5)

Session-Key exchange

We showed how two parties who share a random session key can use a pseudo-random function generator to talk securely over a totally insecure channel.

How should they share a session key?

People spawn processes, and pairs of processes will invoke a protocol in order to share a session key. Only "matching" processes should be able to succeed at sharing a session key, and the session key shared by two processes should look like a random, independent string, even to an adversary who has total control of the network.

Alternative wording: An adversary who is allowed to see session keys of processes of his choice should not be allowed to learn anything about any session key that he shouldn't know. (He *should* know the session keys he's seen, as well as those that are supposed to match them.)

Our setting will be the PKI (public key infrastructure) setting.

Note: Our actual definition will use a style similar to what we've used before. An alternative is the "simulation" style or the "universal composability" style of Canetti.

What exactly is a key exchange protocol?

There is a security parameter 1^n.
A "person" will be an n-bit string. (We will let the Adversary choose the names.) The world will have good-guy persons that the adversary will be trying to break; good-guy people will spawn processes that properly follow the protocol, whereas bad-guy people can do whatever they like, and are really just pseudonyms for the adversary.

A person A spawns a process of type $\langle A, M, b \rangle$:
M is the name of the person the process wants to share a key with;
$b \in \{0, 1\}$ is the "role" of the process, since we need asymmetry so the processes can use the exchanged key consistently after the exchange.

(M can be good, bad, or even A itself.)

We have a polynomial-time key generating process
$\textsc{Gen}(1^n, \text{random bits}) = (pub, pri)$.

A process of type $\langle A, M, b\rangle$ has as input: A, pri_A, M, pub_M.
It interacts with what it thinks is an M process,
and eventually outputs a session key.

ADV chooses the names of both good-guys and bad-guys.
Keys are chosen for good-guys by $\textsc{Gen}$, and ADV sees their public keys. ADV chooses the public keys of bad guys however he likes.

ADV creates $\langle A, M, b\rangle$ processes where A is a good-guy, and interacts with them however he likes.

We will assume that the number of "flows" or rounds is fixed. We will also assume that whether a process begins by reading or writing is fixed. We will also assume that the number of bits read or written on each flow depends only on the security parameter 1^n.

We insist that, in the absence of an adversary, the protocol be correct, in the obvious sense. That is, say that people A and B have generated their keys correctly using GEN. Then if the processes $\langle A, B, 0\rangle$ and $\langle B, A, 1\rangle$ interact correctly using the protocol, then neither outputs FAIL, and the session keys they output are identical.

ADV can choose to see session keys output by some of the processes; he sees every output that is FAIL.

For one of the processes of type $\langle A, M, b\rangle$ – where M is a good-guy – that outputs a session key, ADV will choose it as the *challenge* process; with probability 1/2, ADV will either be given the session key or a random string of the right length.

ADV continues his attack, and then finally tries to guess if the challenge string was the session key or random.

Then ADV's probability of success should be negligibly above 1/2.

EXCEPT: If ADV ever chooses to see the session key of a process $\langle M, A, \bar{b}\rangle$ that *matches* the challenge process, and the session key equals the challenge string, then ADV *loses*.

What is the philosophy behind letting the ADV see some of the session keys?

If such a key doesn't give away the challenge, then the process this key belongs to might start engaging in a protocol using the session key, and the adversary can interact with that protocol and learn something about the session key.

Towards an example

Let p be an n-bit prime, $p-1=2q$ where q is prime. $\mathbb{Z}_p^*$ is the group consisting of $\{1,2,\ldots,p-1\}$ under multiplication mod p.

Let $g \in \mathbb{Z}_p^*$ be a generator of a subgroup G of size q.
The *Decisional Diffie-Hellman* (DDH) assumption (with respect to G) states that: For every polynomially bounded adversary D who is given p and g,

D can't distinguish between (g^x, g^y, g^{xy}) and (g^x, g^y, g^z), where x, y, z are chosen randomly from $\{0,1,\ldots,q-1\}$ and all arithmetic is mod p.

That is, the difference between the probability of D accepting from the first distribution is negligibly different from the probability of D accepting from the second distribution.

Let us also assume a function $h : \{0,1\}^n \to \{0,1\}^{n/2}$ with the property that $h(G)$ induces a nearly uniform distribution on $\{0,1\}^{n/2}$. (See last time.)

Protocol 1

For the public and private keys we use the public and private keys of a secure signature scheme.

Informal Description:
$\langle A, B, 0\rangle$ and $\langle B, A, 1\rangle$ behave as follows:

A chooses random $x \in \{0, \ldots, q-1\}$, computes g^x and signs it and 0 to B,

B chooses random $y \in \{0, \ldots, q-1\}$, computes g^y and signs it and 1 to A;

then both of them output as the session key $h(g^{xy})$.

More Formal Description:

$\langle A, B, b \rangle$ behaves as follows:

A chooses random $x \in \{0, \ldots, q-1\}$,
computes $\alpha = g^x \mathrm{mod} p$,
sends $[\alpha, \textsc{Sign}_A(\alpha, b)]$.

A receives $[\beta, \sigma]$;
if $\textsc{Ver}_B([\beta, \bar{b}], \sigma) = 0$ then output FAIL;
else, output $h(\beta^x \mathrm{mod} p)$.

How to break Protocol 1

Our ADV will work as follows:

ADV first chooses two good-guy names A and B.
ADV also chooses a bad-guy name U, and chooses a public key pub_U and a corresponding private key pri_U using GEN.

ADV creates processes $\langle A, B, 0\rangle$ and $\langle B, U, 1\rangle$.

ADV reads $[\alpha, \sigma_1]$, from the A process and
$[\beta, \sigma_2]$ from the B process.

ADV sends $[\beta, \sigma_2]$ to the A process.
ADV sends $[\alpha, \text{SIGN}_U(\alpha, 0)]$ to the B process.

Both processes will output the same session key, say k.
So ADV looks at session key of the B process,
and challenges the (*nonmatching*) A process.
He knows that the challenge string he receives is the session key (rather than random) iff it is equal to k.

Protocol 2

We fix this problem by having each party also sign the name of the other party.

$\langle A, B, b\rangle$ behaves as follows:

A chooses random $x \in \{0, \ldots, q-1\}$,
computes $\alpha = g^x \mathrm{mod} p$,
sends $[\alpha, \textsc{Sign}_A(\alpha, B, b)]$ to B.

A receives $[\beta, \sigma]$;
if $\textsc{Ver}_B([\beta, A, \bar{b}], \sigma) = 0$ then output FAIL;
else, output $h(\beta^x \mathrm{mod} p)$.

We can prove this is secure, assuming DDH.

Protocol 2 happens to also satisfy the nice property of *forward security*. This means that if at some point in the future, after the key exchange has been completed, an adversary breaks into (say) A's computer and learns A's secret key, this does not cause insecurity to the past key exchange or to the session that used (or is using) it. This is because the key was used only to sign, not to encrypt.

Protocol 2 has the bad property that neither party is sure that he has actually shared a key. (Note that it is impossible that *both* can be sure.)

Public Key Encryption primitive

We have the following polynomial-time algorithms.

- $\textsc{KeyGen}(1^n, \text{random bits}) = (pub, pri)$.
- $\textsc{Enc}(pub, m, \text{random bits})$ for $m \in \{0,1\}^n$.
- $\textsc{Dec}(pri, e)$; this can be an n-bit message, or the symbol FAIL.

Correctness: If $\textsc{KeyGen}$ can output (pub, pri) and $\textsc{Enc}_{pub}(m)$ can output e, then $\textsc{Dec}(pri, e) = m$.

Security of public key encryption primitive

Our very strong definition is CCA security:
security against chosen cipher-text attack.

For every polynomially bounded D:

$\textsc{KeyGen}(1^n)$ is run to obtain (pub, pri) and pub is given to D.

D then queries $\textsc{Dec}_{pri}$ for a while.

D then chooses two n-bit messages M^0 and M^1 and sees
$\textsc{Enc}_{pub}(M^b) = e$ for a random b.

D then queries $\textsc{Dec}_{pri}$, except for e, for a while.

D then outputs b'.
$q_D(n)$ is the probability that $b' = b$.
Then $q_D(n)$ is negligibly above $1/2$.

Suggestions for secure public key encryption primitives

- RSA standard.

  It is very efficient, and provably secure based only on the standard RSA assumption

  ...

  or would be if it were implemented using a "random oracle".

  Instead of the "random oracle", someone's favorite function is used instead.

- Cramer/Shoup and later improvements.
  Reasonably efficient and provably secure based on DDH.

Challenge (a bit tricky)

Say we have a secure public key encryption primitive for encrypting strings of length n.

Come up with a secure public key encryption primitive for encrypting strings of length, say, $2n$.

Protocol 3

$\textsc{Gen}(1^n)$ works as follows:

Generate a key pair $(spub, spri)$ for a secure signature scheme.

Generate a key pari $(epub, epri)$ for a secure public key encryption primitive.

The public key for the protocol will be $(spub, epub)$, and the private key will be $(spri, epri)$.

Process $\langle A, B, 0\rangle$ works as follows:

- Choose a random n-bit string r and send it.
- (To understand this part, see role-1 process protocol below.)
  Receive strings $[\alpha, \beta]$ of appropriate lengths.

  If $\textsc{Ver}_B([\alpha, r, A], \beta) = 0$ then output FAIL.

  Compute $\gamma = \textsc{Dec}_A(\alpha)$.
  If $\gamma =$ FAIL or γ does not end with B then output FAIL.
- Use $k =$ the first half of γ as the session key.

Process $\langle B, A, 1\rangle$ works as follows:

- Receive an n-bit string r.
- Choose a random n-bit string k.
  Compute $\alpha = \textsc{Enc}_A(k, B)$.
  Compute $\beta = \textsc{Sign}_B([\alpha, r, A])$.
  Send strings $[\alpha, \beta]$.
- Use k as the session key.

This is secure. Can you prove it?
Do you see why it would not be secure if, for example, we omitted B from the stuff encrypted in α?