Schneier Chapter 3: Basic Protocols

Chris Brown
University of Rochester Computer Science Department

August 2, 2001

1 Key Exchange

Given you want to use a session key, how to spread it around to the participants?

Key Exchange with Symmetric Cryptography

A and B on network, each shares a secret key with the KDC (Trent or T). How to get them there... indeed a problem!

- A requests session key from T to talk with B
- T generates random session key and encrypts it once with A's key and once with B's key. Sends both to Alice.
- A decrypts her copy and sends B his copy.
- B decrypts his copy
- A and B use the key to communicate.

Trent is obviously both the point to attack and a bottleneck.

KE with PK

As before, easy... now can send signed encrypted msgs to those you don't even know.

- A gets B's public key from KDC
- A generates random session key, encrypts using Bob's public key, sends it to B.
- B decrypts this message with his private key
- A and B then both use same session key for conversation.

Man in the Middle Attack

Given a powerful Mallory:

- A sends B her public key. M intercepts it and sends B his own public key in place of A's
- B sends A his public key, Mallory intercepts it and sends A his own key in place of B's.
- A sends message to B. M intercepts, decrypts with his private key, encrypts with B's public key, and ships it on to B.
- Likewise if B sends to A.

A and B can't really guarantee they are talking to each other, so M can do this impersonation thing. M is invisible unless he causes delays. Rivest and Shamir invented the Interlock protocol to foil the man in the middle.

- A sends B her public key, B sends A his.
- A encrypts her msg, sends half of it to B
- B does the same thing, sends half of his encrypted msg to A.
- A sends other half to B, who puts them together and decrypts with his private key.
- B sends other half of his msg to A, who puts it together with 1st half and decrypts with her private key.

could send 1st half, 2nd half, or every other bit, or 1st "half" could be a one-way hash fn of the msg, 2nd half the encrypted msg itself.

M clearly has a problem since he can't decrypt 1/2 the message, or re-encrypt to send on.

Key Exchange with Digital Signatures

This foils middle man attack too. Trent signs A's and B's public keys, including a certificate of ownership. A and B get their keys and verify T's signature. Now they have a public key they know belongs to the right person and they can exchange session keys, etc.

M can't impersonate A or B since he doesn't know their private keys. He can't substitute his public key because his is signed by Trent as belonging to him (Mallory). If M breaks into the KDC he get's Trent's key and can falsely sign new keys, but he can't decrypt session keys or read messages. He COULD create phony signed keys to fool A and B, so he can do a man in the middle attack. However he has to be able to intercept and exchange messages, which isn't always easy (say on the radio). Computers are easier.

Key and Message Transmission

Can send message without key exchange protocol first. This is a hybrid system and is most common use of PKC (as in PGP): it combines with signatures, timestamps, etc.

- A generates random session key K and encrypts M using K.
- A gets B's public key from database
- A encrypts K with Bob's public key
- Alice sends encrypted message and the encrypted session key to B...she can sign it if she's worried about men in the middle. $E_K(M), E_B(K)$
- B decrypts K using private key
- B decrypts M using K.

This easily generalizes to sending message to several people... you just send key along encrypted in B,C,D's... private keys. Each of and only the people whose public keys are used can in turn use their private keys to get the session key out.

2 Authentication

How does a computer system know who you are? PIN or password. obvious problems. Usually people forget PINS and passwords, but of course they are stolen from computers all the time too...

But computer doesn't need to know the password, just that it's valid. Sounds like a one-way hash situation, or a one-way function in general. Host performs a function on password and compares that with what it's stored. I think UNIX does that too.

Dictionary Attacks, Salt

Presumably the one-way function is public (UNIX's would be available, say, or LINUX's... UNIX, btw, does terrifying encryption... 25+ DES-like encryptions of a string of 64 zeroes, and then salt...) So simply generate likely passwords and encrypt them and compare those with the list in the machine. Matches show valid passwords. This is a *Dictionary attack.

Salt is to foil dictionary attacks. Salt is the computer encryptor's own random string concatenated to the password before the one-way function. Store salt and one-way function result. If enouth salt bits, dict. attack infeasible. The salt idea generalizes to an *initialization vector, used in cipher block chaining modes to get the encryption started (or the initial key in an auto-key cipher). Anyway, Mallory has to concatenate each user's salt value with each possible password in M's dictionary to generate possible matches, instead of one big batch run of the encryption value over the dictionary. So it's a dictionary search per user, not a dictionary search per `/u/`. UNIX uses 12 bits of salt. There have been some very successful attacks on UNIX pws... 40% on a given host in a week. list of 732,000 pws with 4096 salt values estimates 30% success on any host. [Schneier p. 53]. So salt is to protect the system, not to protect any particular single user against a dictionary attack.

SKEY

We use SKEY here at UR – Ever wondered why you work through it backwards?

Anyway SKEY is meant to fix the problem of sending your password in clear over the phone lines. Actually it doesn't do that, it just fixes it so you can't begin to use your password until you've done the SKEY authentication. You could still lose your password to Eve.

Alice or the sys. admin. enters a random number R , and the computer produces $f(R)$, $f(f(R))$, etc. (call these numbers $x_1, x_2, ..., x_100$). This is the list you (Alice) keep. Computer stores x_101 in the clear, associated with Alice in a login database. Now when A wants to login, she enters her name and x_100. Computer calculates $f(x_100)$, which should equal its stored x_101. If it does, OK: A is authenticated and the computer replaces its number opposite her name with the newly-entered x_100. Each time, the computer can ask for the number it wants by its subscript number. Also as you may know we don't use numbers we turn them into a list of random words like MAKO QUIZ FARO GLUT. Eve can't learn anything by listening since each number only used once and the function is one-way. The database doesn't help, either.

Authentication with PK

Sending your password to the computer (say, over a phone line) is very dangerous. Salt doesn't help. Someone on the network who has access to processor memory can steal it before it's encrypted, as well. So... host keeps file of all users' public keys, and each user keeps his private key as usual. A weak protocol then would be:

- Host sends Alice a random string
- A encrypts string with private key, sends it to host along with her name.
- Host looks up public key in database, decrypts.
- If result is the string host sent, initially, host grants A access.
-

No one can impersonate A since they don't have her key, and she never transmits her key, unlike a password. However, the first step is bad. Chosen ciphertext attacks are powerful, and effective against RSA for instance...Mallory, pretending to be the host could get A to encrypt strings to help M find her key. So...

- A performs some computation based on random numbers and her key and sends the result to the host.
- Host sends A a different random number.
- Alice makes more computations on the random numbers she generated, the random number from the host, AND her key, and sends that to the host.
- The host does some computation on the numbers received from her and her public key to verify that she indeed knows her private key.
- If she does, she's authenticated.

But A may not trust the host any more than the host trusts A, so the host should be able to authenticate that it is who it says it is. For more on the above protocol, see Schneier Chapters 21.2, 21.3.

Mutual Authentication with Interlock

Alice and Bob each have a password the other knows: P_A an P_B respectively. This won't work

- A and B trade public keys
- Alice encrypts P_A with B's public key and sends it to Bob
- Bob encrypts P_B with A's public key and sends it to Alice
- A decrypts, verifies it is correct, so does B

Because of this man in the middle attack. If Mallory can get in and substitute his public key for A's (as far as B is concerned) and B's (as far as A is concerned), he can read and re-send their messages invisibly, and Mallory thus learns P_A and P_B undetectably. Interlock can help... various nefarious ways of improving the attack however.

SKID2, SKID3: commercial symmetric crypt. ID protocols.

3 Authentication and Key Exchange

A and B are on a network. How can they exchange keys and authenticate each other so they can communicate securely despite Mallory? Most of these protocols assume Trent shares a different secret key with each participant, all in place before protocol starts.

This section of Schneier describes several (nine) commercial and other published protocols, such as Wide-mouth Frog and Kerberos. Some are found wanting. Here are two example treatments

Wide-Mouth Frog: simplest system using trusted server. A and B share secret key with Trent, used only for key distribution, not for message encryption. To get a session key to Bob takes two steps.

- Alice concatenates a timestamp, Bob's name, random session key, and encrypts whole thing with the key she shares with Trent, sending it to Trent along with her name: $A, E_A(T_A, B, K)$.
- Trent decrypts. The then concatenates a new timestamp, Alice's name, and the random key and sends that to Bob: $E_B(T_B, A, K)$.

This protocol assumes A is competent to generate good session keys, which isn't easy.

Kerberos: Version 5 has same assumptions as above. It's a variant of Needham-Schroeder, dealt with in detail in Schneier. Part of the motivation is to prevent *replay attacks, in which Mallory records old messages and use them later to try to subvert the protocol. We need to be sure that messages are really part of the current protocol. The timestamps in Kerberos are to defeat a security hole in Needham-Schroeder in which Mallory gets access to an old session key.

- A sends T her and B's identity: A, B.

- T makes a msg containing timestamp, lifetime L, random session key K. He encrypts this in the key he shares with Bob. Then he encrypts the timestamp, lifetime, session key, and Bob's identity and encrypts them in the key he shares with Alice. He sends both encrypted messages to Alice: $E_A(T, L, K, B), E_B(T, L, K, A)$
- A generates message with her identity and the timestamp, encryhpts it in K, and sends it to Bob. Alice also sends Bob the message from Trent encrypted in B's key. $E_K(A, T), E_B(T, L, K, A)$.
- B creates message consisting of timestamp plus one, encrypts it in K and sends to Alice: $E_K(T + 1)$.

This assumes all clocks synchronized with Trents. In practice they need only to be approximately synched, and you check for replay attacks within the uncertainty interval.

The conclusion here is that there are many authentication and key-exchange protocols in the literature, and they have been subjected to the normal amount of academic examination and critique. Some have been broken, some not. Conclusions: Protocols fail if designers are too clever and try to do without names, random numbers, etc. Make everything explicit. Optimizing performance depends on assumptions: authenticated time is good if you have it, but you may not. Protocol choice depends on underlying comm. architecture. Minimize size or number of msgs, for instance? Everyone connected or only pairwise?

4 Formal Verification

Lots of work on verifying protocols formally, expecially authentication and key exchange

- Use specification languages and verification tools designed for operating systems or algorithm or maybe even hardware design verification.
- Use expert systems for protocol design and investigation.
- Use logics for analysis of reasoning and belief. This is most popular, and includes BAN logic.
- Formalize the term-rewriting properties of crypto system and reason with that.

If some of this sounds like what you learned in the good old AI course, that's because everything is connected, as I always say. For a little more, see Schneier.

5 Multiple Key PK

PK protocols generalize to multiple recipients, as when you might have 100 operatives in the field and you'd like to be able to send a message to any subset without needing 2**100 keys, for instance. See Schneier.

6 Secret Splitting and Sharing

If you want to send different parts of a message to different recipients who have to cooperate to read it, here's a protocol to split a msg between two people so each piece means nothing but together they can reconstruct it.

- Trent generates random string R the same length as the message M.
- Trent XOR's M with R to generate S.
- Trent gives R to Alice and S to Bob.
- Alice and Bob XOR their pieces together to reconstruct the message.

So T basically uses a 1-time pad and gives the ciphertext to one person and the pad to another. Scheme extends to more people by using more random strings (details for the reader to work out).

If you want to send the codes to unleash massive nuclear strikes in such a manner that any three of five generals can accomplish the launch, that's possible to, and in a generalized version as well. You can do what you need with arrangements of keys and a mechanical arrangement of locks (details left to reader). An * (m,n)-threshold scheme divides message up into n pieces, called *shadows or *shares, such that any m of them can reconstruct msg. However there are ways to cheat. See Schneier, esp. Chapter 22.

7 Protecting Database with Cryptography

Suppose you have a data base of names and addresses you want to keep private. You'd like, say, to make it easy to extract the address of one person you know but hard to extract them all. Use one-way hash and symmetric encryption. Store the full name and address info encrypted by the last name, along with a field that is the last name hashed with the function. If I want to find your record, I search database for your hashed name and decrypt what I find with your last name. Takes a full phone-book dictionary attack to get everyone in the list.