Schneier Chapter 2: Protocol Building Blocks

Chris Brown
University of Rochester Computer Science Department

July 24, 2001

1 Protocols

From Schneier starting with Chapter 2...

Protocol is sequence of steps by 2 or more people to accomplish a task. It's a multi-agent algorithm if you will. Examples are buying groceries, using a trusted third party as an intermediary when I give you a check for a product and you want to be sure my check is OK and I want to make sure I get my product from you. Or you can listen to two modems going through a protocol to negotiate things like connection speed every time you dial up a computer phone connection.

- Everyone involved knows the protocol and all its steps in advance.
- Everyone involved agrees to follow protocol.
- Protocol must be unambiguous and well-defined and correctly understood.
- Protocol must be complete: an action for every possible situation.
- It should not be possible to do more or learn more than what is specified in the protocol.

So they're like algorithms with all the pitfalls.

Alice, Bob, Carol, Dave: people in 2 to 4-party protocols. Alice leads off, Bob responds. Eve : eavesdropper. Mallory: Malicious active attacker. Trent: trusted arbitrator. Walter: warden to guard A and B in some protocols. Peggy: prover. Victor: verifier.

*Arbitrated protocols have a trusted third party, like a banker or lawyer in the middle. Computer arbitrators have some problems: can't look them in the eye, network must charge for the service, there are delays, arbitrator in middle makes bottleneck, one point to subvert. *Adjudicated protocols – executed only when there is some dispute, as in going to small claims court. *Self-enforcing protocol guarantees fairness, no disputes, cheating doesn't work, etc.

Protocol Attacks: can attack encryption algorithms, or their implementation, or the protocol. Here we care about the last. Just eavesdropping is a *passive attack, aimed at obtaining information. Like a ciphertext-only cryptographic attack. Hard to detect so try to prevent. *Active attacks try to alter protocols: introduce, delete, replay, substitute messages; pretend to be someone else; disrupt comm. channel; alter computer files. If one of the parties is attacking the protocol he's a *cheater. *Passive cheaters follow protocol but try to get more information than they're supposed to. *Active cheaters disrupt ongoing protocol somehow.

2 Symmetric Cryptography Communications

For A to send B an encrypted message, need a protocol like this:

1. A and B agree on cryptosystem
2. A and B agree on a key
3. A encrypts her msg with key
4. A sends ciphertext to B
5. B decrypts with key and correct algorithm

If Eve only hears step 4, she needs to decrypt. She wants to hear 1 and 2 as well as 4. Ideally cryptosystems are best if the secrecy only resides in the key. So we could assume Eve knows 1. *Key management is therefore a Big Issue....how to distribute or agree on keys in a secure way.

Mallory could break the cipher and then pretend to be Alice, break the comm. channel somehow, replace messages with garbage that would decrypt to confusing gibberish.

Alice could give copy of key to Eve, mail transcripts of his mail to the National Enquirer, and Bob could do same sort of thing, so A and B have to trust each other here.

So symmetric key systems... need keys distributed in secret. Big Deal, often by courier. Losing a key is disastrous...hence lead-covered code books on ships. If one key per pair of communicators, need O(N*N) keys for N people.

3 One-Way Function

This is a brief qualitative treatment. *One-way functions are a fundamental building block for protocols and are central to public-key cryptography. They are easy to compute but (very very) hard to invert, like breaking a plate. They may not exist, but it seems they do. So we'll pretend. To be useful we need *trapdoor one-way functions, which have a secret component (or key) that makes them easy to invert if you know the key. E.g. of a key is instructions for assembling a printed circuit board out of a bag of small components.

4 One-way hash functions

Or compression function, checksum, message integrity check, etc. Same as in Data Structures and hash-tables. Hash function maps a variable-length *pre-image to a fixed-length *hash value...e.g. the XOR of all the input bytes. Usually used to check if a candidate pre-image could be the real pre-image.

One-way hash is easy to compute from a pre-image, but it's hard to find a pre-image that gives you a particular hash value (NOT like the XOR example above!). Ideally, hashes are *collision "free", meaning it's hard to generate two pre-images with same hash value. So A could verify that B has a file without having to send it and do a diff if his hash value for his file matches hers for her file.

5 Comms with public key cryptography

We know the deal here: A and B have two keys, a public and a private. Can't (it's computationally difficult to) deduce the private key from the public one. Everyone encrypts messages for A with her public key. Noone but A, using her private key, can decrypt her messages. So encryption is the easy direction of a trap-door one-way function. Decryption is the hard direction UNLESS you have a key, in which case the trapdoor makes it easy.

Protocol for PKC:

1. A and B agree on a cryptosystem
2. B sends A his public key
3. A encrypts msg with that key and sends to B
4. B decrypts with his private key

Boom! No key management! Secret is the *asymmetric cryptosystem. Eve can listen to whole protocol and still be helpless. Often there's a database of public keys so steps 1 and 2 collapse to looking up his key.

6 Hybrid Cryptosystems

But! Public key systems are slow, maybe 1000 times slower than DES. And! PKC is vulnerable to chosen-plaintext attacks. if you know that C = E(P) where P is one of a number of known plaintexts (like a dollar amount less than 1,000,000), you can encrypt all n plaintexts and compare with the encrypted message actually sent. Or even knowing a ciphertext does NOT correspond to a particular known plaintext message could be useful...

So practically, a *hybrid system is used (as in Pretty Good Privacy(TM)). You use public-key cryptography to share a *session key, which is good for this communications session only, and which is a key for a symmetric cipher like DES.

1. B sends A his public key
2. A generates a random session key K, encrypts it with B's public key, and sends it to B.
3. B decrypts with his private key, gets session key.
4. Both A and B now encrypt comms using session key.

So no symmetric key sits around for long.

7 Digital Signatures

Ideal signature: authentic and convincing, unforgeable, not reusable, unalterable, unrepudiatable. How do that with computers? In fact one of the earliest proposed applications of digital signatures was to verify nuclear test ban treaties. Each nation had seismometers, and the host nation needed to be sure they were only sending relevant data, while the other nation needed to know the data was not tampered with. With digital signatures, the host nation can read but not alter the data, and the monitoring data knows data is inviolate.

Symmetric Ciphers and Arbitrator

Trent shares key K_A with Alice, K_B with Bob, (well-established and reusable).

1. A encrypts msg to B with K_A, sends to Trent
2. Trent decrypts msg with K_A
3. Trent encrypts msg, along with stmt that he got it from A and is passing it on, with K_B and sends bundle to B.
4. B decrypts whole smash with K_B, has msg and certification.

Trent knows msg is from A since K_A worked.

This meets criteria above. Most interesting is what if B takes T's certification and attaches it to another message he purports is from A – trying to impersonate T in a way. A would object, and T or another arbitrator would ask for the msg and A's encrypted msg – that is, the same information that T must have had. B is in trouble since he doesn't know K_A and can't produce the encrypted version of A's message, so he can never justify sending out the bundle.

To forward A's msg to C requires B sending it back to T, who verifies that it was indeed from A (he checks his database) and then sends it on to C in the usual bundle. This all works but it's hard on T and he's always in the middle, and how would you build a T you would trust? What if someone broke into the signature database?

Signing with Public Key

There are signature algorithms (like DSA) that are only for signing, but you can use regular public-private key this way:

1. A encrypts doc with private key, thus signing doc.
2. A sends signed doc to B
3. B decrypts doc with A's public key, thus verifying signature
4.

Voila! no Trent, and it satisfies the criteria. Note signature is not reusable since it's all mixed in with the document by the cryptosystem. Likewise altering document means it won't be verified by public key. Usually timestamps are added so doc. can't be used at arbitrary time in future.

Signing with Public Key and One-Way Hash Fns

Notice that in the last protocol the entire doc had to be encrypted with private key and we know that's expensive, so...

1. A and B agree in a one-way hash fun and digital signature algorithm
2. A encrypts the hash of the document with her private key. This is the signature.

3. A sends doc and signed hash to B

4. B produces hash of doc, decrypts the signed hash with A's public key. If the match, doc. is valid and signed.

Here the signature is separate from doc. Smaller storage requirements: in fact an archival storage system can just store submitted document hashes, and can verify existence by checking hashes. For instance, copyrights, priority disputes, etc. Simply maintain a timestamp and hash of the file. So you can copyright a document that you actually keep secret! (Have to go public to prove the copyright applies to this particular doc, I guess...)

8 Algorithms and Terminology

Digital signature algorithms are public-key algorithms, with secret info to sign and public information to verify signatures. Because of RSA being a common technique, the locution "encrypting with a private key" has crept in for signing and "decrypting with public key" for verification. There are algs that can be used for signing but not encryption though.

So at the protocol level we don't care. We just write signing with private Key K as $S_K(M)$ and verification with corresponding public key is $V_K(M)$. The *(digital) signature is the bitstring attached by the algorithm (e.g. a one-way hash of the entire document, timestamp, etc.). The entire protocol, the receiver being sure of the identity of the sender and the integrity of the message, is *authentication.

Multiple signatures

1. Alice and Bob sign separate copies... bulky!

2. A signs, B signs the A-signed doc. OK, but can't verify A's sig. without verifying B's as well.

With one-way hash:

1. A signs hash of doc

2. B signs hash of doc

3. B sends his signature to A

4. A sends doc and 2 signatures to Carol

5. C verifies either A's or B's sig or both.

Nonrepudiation

A can cheat. She signes, then anonymously publishes out her private key and claims it was stolen. Or whatever. She can *repudiate, saying someone else signed the doc with her key. Timestamps can help. This however is why there is the idea of private keys residing in tamper-proof modules. You'd like at least that old signatures are not invalidated by disputes involving more recent ones. Solution uses Trent.

Digital Signatures and Encryption

A protocol that delivers secure and authentic documents, like an envelope and a signature.

1. A signs msg with private key: $S_A(M)$
2. A encrypts the signed msg with Bob's public key and send it to him: $E_B(S_A(M)))$
3. B decrypts message with his private key and gets $D_B(E_B(S_A(M))) = S_A(M)$
4. then he verifies with A's public key:

Signing inside the security (envelope) is natural, although note we sometimes sign envelopes FOR security (say on recommendation letters). But you can't get inside these digital encryptions with teakettle either. This way can't remove sig. from msg. Also this way Bob can's be accused of mixing up different letters from different envelopes and presenting them as being paired. Might have different encryption and signing keys, in fact it makes sense since you can hand over encryption key w/o compromising your signatures.

Timestamps are useful, for instance in extending the protocol to acknowledge receipt of message (like a certified letter, return receipt requested).

1. A signs msg with private key, encrypts the signed msg with Bob's public key and send it to him: $E_B(S_A(M)))$
2. B decrypts message with his private key and verifies signature as above: $V_A(D_B(E_B(S_A(M)))) = M$
3. Bob then signs message with his private key, encrypts with Alice's pulic key and sends it back to A:

   $E_A(S_B(M))$
4. Alice decrypts and verifies Bob's signature with his public key, checks to make sure he sent back the same message she sent.

If the same algorithm is used for encryption and digital signature verification, a Resend Attack can read other people's mail.

Mallory is in the system with his own public and private keys. He intercepts and records A's message to Bob. Later he sends the message to Bob, claiming it came from Mallory. Bob decrypts and tries to verify with M's public key, but gets garbage:

$E_M(D_B(E_B(D_A(M)))) = E_M(D_A(M))$,

still he goes ahead (maybe some script is running) and goes on with protocol, sending M a receipt:

$E_M(D_B(E_M(D_A(M))))$.

At this point M just has to decrypt this with his own private key and encrypt it with Bob's public key, then decrypt it again with his private key and then encrypt it with Alices' public key and he's got M.

To beat this resend attack, don't use same op for signature and encryption. different keys for each op would work, or different algorithms, or time stamps (so incoming, outgoing messages are different), as do signatures with one-way hash functions.

Public Key management

How to get someone else's public key? Easiest is a secure database. Need it to be write-protected for all but Trent– e.g. if Mallory substitute his public key for Bob's. Suppose M substituted his public key for B's during transmission from the database? Trent can sign each public key with his private key. When he does this, he's known as a Key Distribution Center or Key Certification Authority. But Alice has the KDC public key stored...suppose M substituted his own public key for that? He could impersonate the KDC.

9 Random Numbers

Most computer random number generators are not really very random. (a good treatment is in Knuth, probably Vol. 1. Might be fun to take some statistics on your favorite generator, look for patterns, etc. Basically, computers are deterministic and algorithms only have finite state. So an algorithm can produce a *pseudo-random sequence generator. For practical purposes want one with long enough period to look random. Various tests: about same number of each digit (0,1), about half the runs of 1's and 0's should be of length 1, 1/2 of length 2, etc. Should not be compressable. It's possible to get periods of 2**256 bits, which are good enough for most practical purps.

*Cryptographically secure pseudorandom sequences have higher criteria: in particular it must be unpredictable, even given complete knowledge of the algorithm and all previous bitstream bits. Idea is to make such generators resistant to attack.

*Really random. Can't be reproduced.