Computer Science 418

Cryptography in Practice

Mike Jacobson

Department of Computer Science
University of Calgary

Week 13

Outline

Authenticity of Keys

Secure communication requires proper mechanisms for managing keys and ensuring their **authenticity**.

Mechanisms for ensuring authenticity of keys:

- A trusted third party
  - A key distribution center (session keys)
  - A certification authority (public keys)
- Peer authentication (e.g. PGP secure e-mail)
- Identity-based cryptography: your ID is your public key

The vast majority of key distribution systems involve a **trusted authority** to ensure authenticity of keys.

Symmetric Key Distribution

Symmetric schemes require both parties to share a common, secret key.

Possible distribution mechanisms:

- A selects a key and physically delivers to B. Secure, but cumbersome.
- Third party selects and physically delivers key to A and B. Also secure, but cumbersome.
- A and B can use a previous key to encrypt a new key. If one key is compromised, all subsequent keys are compromised.
- A commonly-trusted third party called a *key distribution center* (KDC) can relay the key between A and B via encrypted links (commonly used solution).

Key Distribution Centres

Idea:

- Each user holds a shared symmetric **master key** with the KDC
- Master key is used for distributing one-time **session keys**
- Encryption is performed with a session key that is destroyed at the end of the session

Advantages:

- Far fewer long-term keys than if each pair of entities holds a shared long-term key
- Compromise of a session key does not affect master key nor other sessions

Key Distribution Centres: Issues

Issues:

- Hierarchies of KDC's required for large networks, must trust each other
- Session key lifetimes should be limited for greater security
- All keys and entities (users and KDCs) must be authenticated (more later)

PKC Solutions

There are three main contributions in PKC:

- **Digital signatures** — for data origin authentication and non-repudiation
- **Key agreement protocols** — both parties contribute to the generation of a session key (eg. Diffie-Hellman)
- **Key transport** via hybrid encryption — party A generates a session key, encrypts and sends to B using a PKC (B has no control over the session key)

Main problem — user's public keys must be *authenticated* in order to prevent active attacks such as man-in-the-middle and impersonation.

Public-Key Distribution, I

1. **Point-to-point delivery over a trusted channel** such as personal exchange, registered mail, courier, etc.

   Problems: slow, inconvenient, potentially expensive.

2. **Direct access to a trusted public file** (public-key repository).

   Advantage: no user interaction.

   Problems:
   - The repository must be secure and tamper-proof (otherwise impersonation is still possible),
   - Users must have a secure channel (see Point 1) to initially register their public keys.

Public Key Distribution, II

3. An **on-line trusted server** dispenses public keys on request. The server signs the transmitted keys with its private key.
   Problems:
   - All users must know the server's public verification key,
   - The trusted server must be online and may become a bottleneck,
   - A communication link must be established with both the server and the intended recipient,
   - The server's public-key database may still be subject to tampering.
4. **Off-line server and certificates** (certification authorities).
5. Use of **systems implicitly guaranteeing authenticity** of public parameters (ID-based systems).

Option 5 is feasible, but has its own problems. We will focus on Option 4.

Public-Key Infrastructures

Definition 1 (Public-Key Infrastructure (PKI))

A set of techniques and procedures supporting authenticated key management for PKC. Specifically, a PKI supports:

- initialization of system users
- generation, distribution/authentication, and installation of public and private keys
- controlling the use of keys (eg. life cycles of session keys, public and private keys)
- update, revocation, and destruction of keys (eg. managing compromise of private keys)
- storage, backup/recovery, and archival of keys (eg. maintaining an audit trail)

Public-Key Certificates

Definition 2 (Public-Key Certificate)

A data structure consisting of a **data part** (containing at least the user ID and the corresponding public key) and a **signature part** consisting of the digital signature of a *certification authority* over the data part.

A certificate should also include information such as:

- A time-stamp indicating the currency of the certificate (to facilitate key changing and revocation)
- Additional information about the key (key generation algorithm, intended use)
- Key status (for revocation)
- Signature verification information (certification authority's name, signature algorithm used)

Certification Authorities

Definition 3 (Certification Authority (CA))

A trusted third party whose signature on a certificate vouches for the authenticity of the public key bound to the subject entity.

Idea: CA issues public key certificates that may be verified off-line. Users may exchange authentic public keys without having to contact the CA.

Example 1

X.509 is an IETF (Internet Engineering Task Force) standard for certificate-based authentication schemes (used in S/MIME, IPsec, SSL). **VeriSign** is an online CA service that uses X.509 certificates.

Obtaining Public Keys

User B uses a public-key certificate to obtain the authentic public key of user A as follows:

1. Acquires the authentic public key of the CA (done only once, eg. pre-loaded in web browsers)
2. Acquires a public key certificate corresponding to A over an insecure channel such as a central database, or even directly from A
3. Verify the authenticity of the public key:
   (a) Verifies the currency of the certificate using the time-stamp
   (b) Verifies the signature on A's certificate using CA's public key
   (c) Verifies that the certificate has not been revoked
4. If all the checks succeed, accepts the public key in the certificate as A's public key

Requirements for the Scheme

1. Any participant can read a certificate to determine the name and public key
2. Any participant can verify that the certificate originated from the CA and is not counterfeit
3. Only the CA can create and update certificates
4. Any participant can verify the currency of the certificate

Main Issue / Problem: CA has to be trustworthy!

- not bad for small, closed deployment
- national or international level?

User Registration

Users must register with the CA in a secure manner (typically in person):

- The CA's public key (required for certificate verification) may be obtained at that time
- CA may generate user keys, or certify owner-generated keys (possibly without user revealing the private key)
- May store keys for backup

CA must verify the **binding** between the public and private keys.

CA Hierarchies

Large networks have hierarchies of CAs:

- Tree hierarchy — each node represents a principal whose public key is certified by its parent
- Leaf nodes — end users
- Non-leaf nodes — CAs at various levels and domains (e.g. country level has domains
  industry (.com)
  education (.edu)
  government (.gov)
  other organization (.org, .net)
- Two end users can obtain authentic public keys by finding a common ancestor node in the hierarchy

Certificate Revocation

Certificates may need to be revoked before they expire, for the following reasons:

- A user's private key is compromised
- A user is no longer certified by his current CA
- A CA's certificate is assumed to be compromised

Mechanisms for revocation:

- CA maintains a **certificate revocation list** (CRL), available online, signed by the CA
- Alternatively, incremental lists known as **Delta revocation lists** are disseminated through the hierarchy
- CA must time-stamp revocations — signatures issued prior to revocation date should be considered valid

Identity Based Cryptography

Motivation: an ideal e-mail system in which knowledge of a person's name (or e-mail address) alone is sufficient to

- send mail which can be read by that person only (secure),
- allow verification of signatures that could have only been produced by that person.

Idea (Shamir 1984): bind public keys directly to a user's identity

Definition 4 (Identity-based cryptosystem)

A PKC in which an entity's public identification information (unique name) plays the role of its public key.

Issues

Advantages: need for public key authentication is eliminated.

- Users need not exchange keys
- Public directories (files of public keys or certificates) need not be kept

If the wrong public user data is used, the cryptographic transformations fail.

Problem: how are the *private* keys generated? Recall that in order for a PKC to be secure, it must be computationally infeasible to compute the private key given the public key!

Private Key Generation

Answer:

- The system requires another piece of trap-door information: a **master key** that can be used to compute the private keys.
- The unique name (i.e. the public key) is used by a trusted authority to compute the entity's corresponding private key, using the master key.

Advantage: Trusted authority is only required during the set-up phase (to compute private keys)

Disadvantage: Users must trust the authority completely (it knows all the private keys)

Typical Application

- Users send encrypted messages using a public key derived from the recipient's ID and a **key validity period** (time stamp), using some publicly available function (e.g. converting concatenation of these two strings to appropriate length integer).
- Recipient requests the private key corresponding to a particular validity period from trusted authority

Incorporating a validity period into the public keys gives keys a lifetime, mitigating the problem of compromised keys (i.e. revocation).

Extreme solution: use a different public key for each message (unique time stamp instead of validity period).

Comparison to PKI

Both systems require a trusted third party. In ID-based cryptosystems, this authority always has access to the private keys.

In PKI, senders of messages / verifiers of signatures must obtain public keys of other users. In ID-based crypto, recipients / signers must obtain their own private keys.

Examples of ID Based Schemes

Signature Schemes

- Shamir (CRYPTO 1984) — based on RSA
- Feige, Fiat, and Shamir (J. Cryptology 1998) — based on computing square roots modulo pq (p and q large primes)

Encryption Schemes (good deal harder!)

- Boneh and Franklin (CRYPTO 2001) — uses the Weil pairing on elliptic curves.

  (first practical ID based encryption scheme)

Overview

Today, **authentication** is arguably the most important application of cryptography. Three main classifications:

- **Data-origin authentication** (digital signatures) — covered previously
- **Entity authentication** (client-server, user-host, process-host)
- **Authenticated key establishment**

KDCs and PKIs make up a general key authentication framework.

We still need protocols for ensuring *entity authentication* within these frameworks.

Authentication Protocols and Nonces

Definition 5 (Authentication protocol)

A sequence of one or more information exchanges used to convince parties of each others' identity.

Authentication may be one-way or mutual. Key issues:

- Confidentiality (e.g. to protect session keys)
- Timeliness (freshness) — to prevent **replay attacks** where a signed message is copied and later resent

Two means to ensure freshness: **nonces** and **time stamps**.

Definition 6 (Nonce)

A number or bit string that is used only once (usually in a particular message or protocol).

Standardized Authentication Protocols

ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) both standardized authentication primitives.

Notation:

A, B — identities of users A and B, respectively
M — identity of a masquerader (impersonator)
T — identity of a trusted authority
K_{XY} — session key shared by entities X and Y
$E_{K_{XY}}$ — symmetric encryption using key K_{XY}
TS_X — time stamp generated by entity X
N_X — nonce generated by entity X
$cert_X$ — public key certificate of entity X
sig_X — public key signature generated by entity X

ISO One-Pass Unilateral Authentication

Symmetric key version

1. $A \rightarrow B : E_{K_{AB}}(TS_A, B)$

Note: The primary function of E_K is to provide data integrity, not secrecy.

- Example problem: cannot detect rearrangement of plaintext blocks in ECB mode for example.

Better, use a MAC:

1. $A \rightarrow B : TS_A, B, MAC_{K_{AB}}(TS_A, B)$

Public key version

1. $A \rightarrow B : cert_A, TS_A, B, sig_A(TS_A, B)$

Use and Properties

Timeliness is assured via the time stamp TS_A that cryptographically integrates the current time into the message

B decrypts/verfies and authenticates A if the time stamp TS_A is within an acceptable range

Inclusion of B *binds* this run of the protocol to a session in which A is authenticating to B (as opposed to someone else)

A's involvement is guaranteed if B can successfully decrypt the message (B cold also check, for example, that the message came from an IP address known to belong to A)

Advantages and Disadvantages

Advantage: no real-time interaction between A and B

Disadvantage: clocks must be synchronized (and thus securely maintained). Alternatively, sequence numbers could be used, but this still requires synchronization

ISO Two-Pass Unilateral Authentication

Symmetric key version

1. **B's challenge**: $B \rightarrow A : B, N_B$
2. **A's response**: $A \rightarrow B : MAC_{K_{AB}}(N_B, B)$

Public key version

1. **B's challenge**: $B \rightarrow A : B, N_B$
2. **A's response**: $A \rightarrow B : cert_A, N_A, B, sig_A(N_A, N_B, B)$

Use and Properties

Timeliness is assured interactively via a **challenge-response** mechanism

B accepts A's response if upon decryption/verification, he successfully extracts his nonce N_B

- Intuition: A's cryptographic operation must have taken place *after* her receipt of B's nonce

Public key version: By including N_A, A ensures that she does not inadvertently sign a message of B's preparation (prevents chosen message attacks)

ISO Three-pass mutual authentication

Symmetric key version

1. $B \rightarrow A : B, N_B$
2. $A \rightarrow B : N_A, MAC_{K_{AB}}(N_A, N_B, B)$
3. $B \rightarrow A : MAC_{K_{AB}}(N_B, N_A, A)$

Public key version

1. $B \rightarrow A : B, N_B$
2. $A \rightarrow B : cert_A, N_A, B, sig_A(N_A, N_B, B)$
3. $B \rightarrow A : cert_B, A, sig_B(N_B, N_A, A)$

Use and Properties

Presence of N_B in the encrypted messages/signatures binds these messages to the same protocol execution.

The original version had B select a second nonce N'_B. This simply represents two interleaved unilateral protocols and succumbs to an **interleaving attack** with the result that:

- A believes B initiated the run and accepts B's identity
- B is still awaiting termination of the run

Interleaving Attack

Original Public key version

1. $B \rightarrow A : B, N_B$
2. $A \rightarrow B : cert_A, N_A, B, sig_A(N_A, N_B, B)$
3. $B \rightarrow A : cert_B, \mathbf{N'_B}, A, sig_B(\mathbf{N'_B}, N_A, A)$

Interleaving Attack (Wiener 1991)

1. $M \rightarrow A : B, N_B$ **(M masquerades as B)**
2. $A \rightarrow M : cert_A, N_A, B, sig_A(N_A, N_B, B)$ **(A believes M is B)**
   1. $M \rightarrow B : A, N_A$ **(M executes unilateral protocol with B while masquerading as A)**
   2. $B \rightarrow M : cert_B, N'_B, A, sig_B(N'_B, N_A, A)$
3. $M \rightarrow A : cert_B, N'_B, A, sig_B(N'_B, N_A, A)$

Authenticated Session Key Distribution Via KDC

Needham, Schroeder 1978

- Original KDC session key distribution protocol (basis of *Kerberos* session key distribution)
- Utilizes a challenge-response mechanism and symmetric encryption (no public key at all)
- T plays the role of the KDC; K_S is the session key

Needham-Schroeder Protocol

Protocol:

1. $A \rightarrow T : A, B, N_A$
2. $T \rightarrow A : E_{K_{AT}}(K_S, B, N_A, E_{K_{BT}}(K_S, A))$
3. $A \rightarrow B : E_{K_{BT}}(K_S, A)$
4. $B \rightarrow A : E_{K_S}(N_B)$
5. $A \rightarrow B : E_{K_S}(N_B - 1)$

Key distribution in steps 1,2,3
Mutual authentication of A and B in steps 4,5

Replay Attack on Needham-Schroeder

Suppose M has compromised a previous session key K'_S and has recorded message 3 from a previous run:

1. $T \rightarrow A : E_{K_{BT}}(K'_S, A)$.

Denning, Sacco (1981) — M impersonates A as follows:

1. $M \rightarrow B : E_{K_{BT}}(K'_S, A)$ **(replay of old, valid message)**
2. $B \rightarrow M : E_{K'_S}(N_B)$
3. $M \rightarrow B : E_{K'_S}(N_B - 1)$

Result:

- B accepts K'_S as a valid session key shared with A
- M can continue to impersonate A successfully.

Denning's & Sacco's Proposed Fix

Uses a time stamp TS_T generated by T instead of A's nonce N_A:

1. $A \rightarrow T : A, B$
2. $T \rightarrow A : E_{K_{AT}}(K_S, B, \mathbf{TS_T}, E_{K_{BT}}(K_S, A, \mathbf{TS_T}))$
3. $A \rightarrow B : E_{K_{BT}}(K_S, A, \mathbf{TS_T})$
4. $B \rightarrow A : E_{K_S}(N_B)$
5. $A \rightarrow B : E_{K_S}(N_B - 1)$

Good news: replaying message 3 will no longer work, because B will reject the message if the current time differs greatly from TS_T.

Bad news: a **suppress-replay attack** is possible if B's clock is not tamper-proof. M proceeds as follows:

- Sets B's clock behind and suppress Message 3
- Sends Message 3 when B's clock corresponds to TS_T.

Fix — Combination of Nonces and Expiration Times

Let $time_B$ denote the expiration time for K_S (determined by B)

1. $A \rightarrow B : A, \mathbf{N_A}$
2. $B \rightarrow T : B, N_B, E_{K_{BT}}(A, \mathbf{N_A}, \mathbf{time_B})$
3. $T \rightarrow A : E_{K_{AT}}(B, \mathbf{N_A}, K_S, \mathbf{time_B}), E_{K_{BT}}(A, K_S, \mathbf{time_B}), N_B$
4. $A \rightarrow B : E_{K_{BT}}(A, K_S, \mathbf{time_B}), E_{K_S}(N_B)$

Nonces N_A and N_B assure both A and B of session timeliness

Only B needs to check $time_B$, so no clock synchronization needed

In Message 3, the block $E_{K_{BT}}(A, K_S, time_B)$ serves as a **ticket** that A can use to re-authenticate with B without interaction with T during the time limit specified by $time_B$:

1. $A \rightarrow B : E_{K_{BT}}(A, K_S, time_B), N'_A$
2. $B \rightarrow A : N'_B, E_{K_S}(N'_A)$
3. $A \rightarrow B : E_{K_S}(N'_B)$

Authenticated Diffie-Hellman Key Agreement

Diffie 1992

- Also referred to as *station-to-station* (STS) Protocol
- Basis of Internet Key Exchange (IKE) protocol component of IPsec

Public parameters:

- Large prime p, primitive root g of p
- CA's public key (for certificate validation)

Authenticated Diffie-Hellman: Protocol

Protocol (all "(mod p)"s omitted to avoid clutter):

1. $A \to B : g^x$ **(A selects random integer x)**
2. $B \to A : g^y, cert_B, E_K\left(sig_B(g^y, g^x)\right)$. Details:
   - B **selects random integer** y
   - B **computes shared key** $K = g^{xy}$
   - B **encrypts his signature on** g^y **and** g^x **using the key** K
   - **Upon receipt,** A **also can also compute** $K = g^{xy}$
3. $A \to B : cert_A, E_K\left(sig_A(g^x, g^y)\right)$

Note: g^x and g^y also playing the role of nonces to assure freshness

Services Provided

Mutual entity authentication

Mutual authenticated key agreement — each party contributes randomness to K, each party signs the key agreement material

Mutual key confirmation — both parties encrypt and decrypt with K

Perfect forward secrecy — compromise of one session key K or even one of the private keys used for signature generation does not compromise previous session keys as each session key is generated from one-time secrets

Note: original version had a minor flaw, succumbing to a **denial of service attack**.

Denial of Service Attack on STS

Lowe 1994: M masquerades as B to A and faces B as himself:

1. $A \to M : A, g^x$ **(A thinks M is B)**
   1. $M \to B : M, g^x$ **(M initiates protocol with B as himself)**
2. $B \to M : g^y, cert_B, E_K\left(sig_B(g^y, g^x)\right)$
   1. $M \to A : g^y, cert_B, E_K\left(sig_B(g^y, g^x)\right)$
      (A believes this message is from B due to the signature)
3. $A \to M : cert_A, E_K\left(sig_A(g^x, g^y)\right)$

Result:

- Denial-of-service against A : believes she shares a session key with B.
- B thinks he has participated in an incomplete run with M and is unaware that A is involved at all.
  Different from M simply blocking the last message, in which case B knows that A was trying to establish the session key.

Fix of DoS Attack on STS

This DoS attack is significant if A is a server, as M can cause many false user authentications (and subsequent resource allocations).

Simple fix — include IDs of both participants in the signed messages:

1. $A \to B : g^x$
2. $B \to A : g^y, cert_B, E_K\left(sig_B(\mathbf{A}, \mathbf{B}, g^y, g^x)\right)$
3. $A \to B : cert_A, E_K\left(sig_A(\mathbf{A}, \mathbf{B}, g^x, g^y)\right)$

Previous attack fails: if B and M are included in B's response in message 2, then M cannot use this message to authenticate to A.

General principle (Abadi and Needham):

> *If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.*

Random Numbers in Cryptography

There are many uses of **random numbers** in cryptography:

- nonces in authentication protocols to prevent replay
- session keys
- public key generation
- key stream for a stream cipher

It is critical that these values be

- **statistically random** — independent, uniform distribution
- **unpredictable** — cannot infer future sequence on previous values)

How to Obtain Randomness?

The only source of true randomness is the real world.

- Find a regular but random event and monitor. Examples:
  - radioactive radiation
  - radio noise (white noise)
  - thermal noise in diodes
  - leaky capacitors
  - mercury discharge tubes, etc.
- Need special hardware in general (e.g. radiation counters)
- Can be slow and cumbersome
- Problems of bias or uneven distribution — have to compensate or use noisiest bits from each sample). One possibility: pass data through a cryptographically secure hash function.

Pseudo-Randomness

Published collections of random numbers also exist, but they are too limited and well-known for most uses.

In practice, one uses **pseudo-randomness**.

Definition 7 (Pseudorandom Number/Bit Generator (PRNG, PRBG))

An algorithmic technique to create sequences of statistically random numbers/bits, initialized with a random **seed**.

Statistical Randomness Tests

NIST FIPS 140-1 is a standard for randomness for PRBGs. It includes (among others) the following statistical tests, which take as input 20000 PRBG produced bits:

1. **Monobit test** — the number of '1's should be strictly between 9654 and 10346
2. **Poker test** — divide the sequence into 5000 non-overlapping blocks of length 4 and determine whether the frequencies of each length 4 block are as expected
3. **Runs test** — B_i (number of runs of '1's of length i) and G_i (number of gaps of length i) must fall into the expected intervals for $1 \leq i \leq 6$ (runs of length greater than 6 are counted as being length 6)
4. **Long run test** — no runs of length greater than 34

Cryptographically Secure PRBGs

Simple PRNG example: *linear congruential generator*

$$X_{n+1} = aX_n + c \pmod{m} .$$

- Advantage: outputs long statistically random sequences
- Disadvantage: fails unpredictability — it is too easy to reconstruct entire sequence given only a few values

Definition 8 (Cryptographically secure PRBG (CSPRBG))

Must pass the **next-bit test**: there is no polynomial time algorithm that, on input of the first k bits of an output sequence, can predict the $(k+1)$-st bit with probability significantly greater than $1/2$.

For all practical purposes, a CSPRBG is unpredictable.

Examples of CSPRNG

Non-Example: linear congruential PRNG

Simple Examples (idea: output of a strong hash function or block cipher is statistically random)

- $X_i = H(X_{i-1})$ where X_0 is a random seed (predictable, but good for distilling random bits from another source).
- $X_i = E_{K_m}(C+1)$ where K_m is a protected master key and C is a counter of period N. Seems to be computationally infeasible to predict next X_i if K_m is secret.

More Complicated Examples:

- ANSI standard X9.17 — three separate applications of 3DES $(E_{K_1} D_{K_2} E_{K_1})$ to generate a 64-bit random number
- Blum-Blum-Schub PRBG — satisfies the next-bit test under the assumption that the QRP is intractable

ANSI X9.17

Inputs:

DT_i — 64-bit representation of the current date and time
V_i — 64-bit seed value

Keys: two 56-bit DES keys K_1, K_2

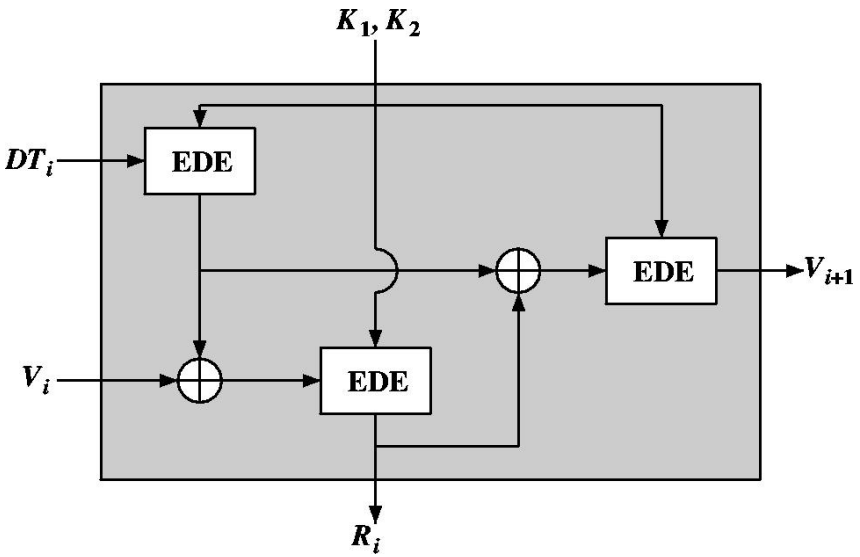
Outputs:

R_i — 64-bit pseudorandom bit string
V_{i+1} — seed value for next iteration

Formulas:

$$R_i = 3DES_{K_1,K_2}\left(V_i \oplus 3DES_{K_1,K_2}(DT_i)\right)$$
$$V_{i+1} = 3DES_{K_1,K_2}\left(R_i \oplus 3DES_{K_1,K_2}(DT_i)\right)$$

Diagram of ANSI X9.17



Properties of ANSI X9.17

ANSI X9.17 is one of the strongest PRNGs:

- 112-bit key
- Each round depends on the current date/time DT_i and the seed value V_i which is distinct from the previous pseudorandom number R_i produced
- Even if R_i is compromised, can't derive V_{i+1} without "'inverting" $3DES$

Appears unpredictable in practice (no proof).

Can substitute 3DES with any secure block cipher (eg. AES)

Common Mistakes with PRNGs

The seed must have sufficient **entropy** to make it unpredictable.

The following are all real life (bad) examples!

1. Generating a random 512-bit prime using a 32-bit seed for the random number generator. The entropy of the resulting prime is only 32 bits — easy to exhaustively try all possible seeds.
2. Generating a random 512-bit prime by calling a system PRNG that produces 32-bit random numbers, padding with 0s to 512 bits, and looking for the smallest prime greater than the number. This approach also has only 32 bits of entropy.
3. Instead of padding with zeros, call the system PRNG 16 times to generate 16 32-bit random numbers. This still has only 32 bits of entropy.

Fourth Bad Example — Kerberos 4

Kerberos 4 generates DES session keys by using a PRNG, seeded with a 32-bit value, to produce two 32-bit random numbers.

Problem: only 32 bits of entropy (should be 56)

Bigger problem: seed is the XOR of 5 random 32-bit numbers:

- time of day in seconds since Jan. 1, 1970
- fractional part of the current time in microseconds
- process ID of Kerberos server process
- cumulative count of session keys produced so far
- host id of the machine on which Kerberos is running

Entropy of each of these quantities: between 1 to 20 bits

Thus, Kerberos 4 seed has only 20 **bits of entropy** — it is easy to test all 2^{20} possible values in seconds! (Better in Kerberos 5.)

Possible Fix to Kerberos 4

Compute a hash on the concatenation of the 5 values.

- Every bit of randomness contributes to every bit of the session key
- Successive applications of the hash function will produce further pseudorandom bits (but with no more total entropy than the seed)

See the Internet Engineering Task Force's (IETF) Request For Comments RFC 1750 "Randomness Recommendations for Security" for more information about guidelines for deploying random number generators. Section 6 covers recommendations for software-based strategies for example.

Fifth Bad Example — Factoring RSA Moduli

1. Scrounge the internet for lots of RSA public keys with moduli $n_1, n_2, \ldots$
2. Compute $\gcd(n_i, n_j)$ for lots of $i \neq j$

You'd be surprised how many of the moduli you can factor!

Problem: too many people use the same primes, obtained via bad PRNG.

NIST Recommendations

Moral: The number of bits of entropy must correspond to the overall bit security of the system.

Example: 1024-bit RSA provides 80 bits of security, so the seed material for the PRNG must have at least 80 bits of entropy.

NIST's Recommendations for Security levels:

RSA modulus (in bits)	1024	2048	3072	8192	15360
Hash function size (in bits)	160	224	256	384	512
Security level (in bits)	80	112	128	192	256

Security level: key length for block cipher providing equivalent level of difficulty to break

Two Real Life Examples

We will now see two applications that put much of what we've learned together:

- **PGP**

  "**P**retty **G**ood **P**rivacy" — a secure e-mail system developed by Phil Zimmerman (`www.philzimmermann.com`)
- **SSH**

  Secure **Sh**ell – a PKC based access control system for remote login and file transfer

Features of PGP

Website: `www.pgp.com`

- Originally available for free world-wide, since 2010 owned and sold by Symantec
- Runs on most platforms
- Uses best available cryptographic primitives
- Not developed by government nor standards organization
- Compatible with most e-mail programs
- Automatically segments large messages (to accommodate message size limitations)
- Users may have multiple public keys, each identified by its 64 low order bits (*key ID*, denoted ID_K)

Cryptographic Services Provided by PGP

Authentication, data integrity, non-repudiation, using digital signatures

- DSS/SHA-1 (1024 bit keys), RSA/SHA-1 (768 to 3072 bit keys) and others are supported

Confidentiality, using a variation of CFB encryption (see documentation for details)

- A wide variety of block ciphers including AES are supported
- El Gamal and RSA for key transport (hybrid encryption)

Sending Secure E-Mail Using PGP

A sends an authenticated, encrypted message M to B as follows:

1. Computes the signature $S = D_A(H(M))$ on the SHA-1 hash of M
2. Compresses (S, M) using ZIP
3. Generates a random one time key K_{OT} to be used to encrypt *only* this message M
4. Uses K_{OT} to encrypt a time stamp TS, the key ID $ID_K(E_A)$ of her public key, the signature S, and the message M:

$$C = E_{K_{OT}}(TS, ID_K(E_A), S, M)$$

5. Encrypts K_{OT} using B's public key and sends the corresponding key ID $ID_K(E_B)$, encryption of K and C to B:

$$(ID_K(E_B), E_B(K_{OT}), C)$$

Receiving Secure E-Mail Using PGP

Upon receipt of $(ID_K(E_B), E_B(K_{OT}), C)$, B decrypts and verifies:

1. Decrypts the one time key $K_{OT} = D_B(E_B(K_{OT}))$
2. Recovers the signature S and message M by computing $D_{K_{OT}}(C) = (TS, ID_K(E_A), S, M)$
3. Checks that the time stamp TS is within an acceptable limit
4. Verifies the authenticity of M by comparing $H(M)$ with $E_A(S)$

Some features:

- The key IDs allow A and B to use the correct public keys when they have multiple public/private key pairs
- The use of time stamps prevents replay attacks
- No session keys are needed as each symmetric key is used to encrypt only one message

Types of Keys in PGP

One-time encryption keys

- Generated by PGP via PRNG
- Used for encryption of the current message only

Public/private keys

- User's public/private key pairs, other users' known public keys
- Generated by PGP via PRNG
- Used for signature generation and verification

Pass phrase

- Generated by user
- Used for encrypting and storing user's private keys

Generation of One Time Keys

Generated pseudorandomly as follows (based on ANSI X9.17):

- Compute random 128-bit initial key K_0 and random 64-bit IV X_0
- Compute pseudorandom 64-bit blocks X_i $(i = 1, 2, \dots)$ as $X_i = E_{K_0}(X_{i-1})$
- Use two pseudorandom blocks X_i, X_{i+1} as the 128-bit block one time key K_{OT}

Seed generation:

- Generate a 256-byte buffer of true random data by measuring the latency between key strokes and their content
- Pass this data through a hash function for whitening

Same seed generation for generation of public/private key pairs

Private Key Rings

A user's public/private key pairs are stored in a *private key ring* maintained and stored by the user.

Each entry corresponds to one public/private key pair and contains:

- time-stamp (time of key creation)
- key ID
- public key (generated by PGP)
- private key (generated by PGP)
- user ID — different ID's, typically email addresses, may be assigned to different public/private keys

Stored in encrypted form using a block cipher. Access key is the SHA-1 hash value of a user generated secret pass phrase. To retrieve a private key, the user gets prompted to enter the pass phrase.

Public Key Rings

Other users' public keys known to a user are stored in a *public key ring*.

Can obtain keys from:

- secure public channels, CAs, etc...,
- *plus* from a mutually trusted individual.

Novel aspect of PGP: authenticity of public keys is **decentralized**

- PGP provides a mechanism for *quantifying* trust.
- no central trusted authority!

Main Problem

Public Key Ring Entries

Each entry corresponds to one known public key and contains:

- user ID (email address), time-stamp, key ID, public key.
- *owner trust field* — is this public key trusted to sign other certificates? User-assigned — higher value indicates higher degree of trust.
- *key legitimacy field* — higher value indicates higher trust in the binding of public key to user ID. Computed by PGP as a function of signature trust fields.
- *digital signatures* — zero or more signatures, each vouching for the authenticity of the key to ID binding of this public key.
- *signature trust fields* — each indicates the degree of trust in one signature. Higher value indicates a higher degree of trust in the signature's author. The key legitimacy field is a function of the signature trust fields.

Obtaining Public Keys, I

A inserts a new public key (certificate with attached signatures) into his public key ring as follows:

1. The owner trust field is assigned (byte value).
   - If the public key belongs to A, a value of "ultimate trust" is assigned.
   - Otherwise, user selects one of "unknown," "untrusted," "marginal trust," or "complete trust".

2. Signature trust fields are assigned (byte values). For each attached signature:
   - If the signature's author is unknown, the signature trust field is assigned "unknown user".
   - Otherwise, the signature trust field is assigned the corresponding owner trust field.

Obtaining Public Keys, II

3. The key legitimacy field is evaluated based on the signature trust fields:
   - If at least one signature trust field is "ultimate trust," the key legitimacy is assigned "complete".
   - Otherwise, key legitimacy is derived from a weighted sum, where weights are assigned to trust values as follows:

     weight of $1/X$ to "always trusted" signature trust fields,
     weight of $1/Y$ to "usually trusted" signature trust fields,

     where X and Y user-configurable parameters.
   - If total weight is ≥ 1, the key legitimacy is assigned "complete". So in the absence of "ultimate trust", one needs

     at least X "always trusted" signatures or
     at least Y "usually trusted" signatures or
     some suitable combination thereof.

     If total weight is < 1, the key legitimacy is assigned "not trusted" or "marginally trusted".

Consistency of Public Key Rings and Key Revocation

This scheme makes it possible to trust the authenticity of a user's public key, but not to trust the user to sign other users' keys.

Key revocation:

- PGP occasionally checks public key rings for consistency.
- A user can revoke his public keys by issuing a *key revocation certificate* — a signed certificate with a revocation flag set — and sending it to as many users as possible.
- The recipients must then update their public key rings accordingly.

PGP: Usability?

Not in wide-spread use (poor usability).

SSH (Secure Shell)

Website: `www.tectia.com` (`www.ssh.com` redirects to there)

SSH is an access control system consisting of 3 components:

1. **SSH Transport Layer Protocol**
   - unilateral authentication (server to client) — client downloads server's public key
   - establishment of shared session key for secure communication
2. **SSH User Authentication Protocol**
   - unilateral authentication (client to server) protected by shared session key
3. **SSH Connection Protocol**
   - interactive applications protected by shared session key

Once the secure channel is set up in step 1, the other two are relatively straightforward.

SSH TLP — TCP Connection Establishment

1. $C \rightarrow S : V_C$
2. $S \rightarrow C : V_S$
3. $C \rightarrow S : I_C$
4. $S \rightarrow C : I_S$

Steps 1 & 2: identification

- V_C, V_S: client's and server's SSH protocol and software versions

Steps 3 & 4: algorithm negotiation

- I_C, I_S: lists of algorithms supported for key agreement, encryption, MAC, compression

For each category, the algorithm chosen is the first one listed in I_C that is also listed in I_S.

SSH TLP — Key Agreement

Unilaterally authenticated Diffie-Hellman, server S to client C:

Protocol (all "mod p"s omitted):

1. $C \rightarrow S : g^x$
2. $S \rightarrow C : K_S,\ g^y,\ sig_S\big(H(V_C, V_S, I_C, I_S, K_S, g^x, g^y, K)\big)$ where
   - K_S — server's public key
   - $K = g^{xy}$ — session key
   - V_C, V_S — SSH protocol & Software versions
   - I_C, I_S — algorithm lists
3. C verifies authenticity of K_S and validates the server's signature

Note that K_S is not authenticated (beware of man-in-the-middle attacks!)

Management and Validation of Server's Public Keys

Two approaches:

1. Use public-key certificates
   - Problem: PKI not widely deployed
2. Current solution: each client maintains a local database (e.g. `$HOME/.ssh/known_hosts`) containing associations between servers and public keys.

   Suggested methods to ensure authenticity of stored public keys:
   - carry authenticated copy on removable storage media (e.g. a USB key or token)
   - obtain public key over an insecure channel, verify over phone (read out hash of obtained public key)

   Not perfect, but a huge improvement over `rlogin, rsh, rftp, telnet` etc (which have no security!)

SSH TLP Secure Channel

Once authenticated Diffie-Hellman is completed, server and client have a shared session key and hence a secure channel.

Services of the secure channel:

- Confidentiality
  - 3DES-CBC required
  - AES128-CBC recommended
- Data Integrity
  - HMAC-SHA1 required
  - HMAC-SHA1-96 recommended

SSH User Authentication and Connection Protocols

SSH User Authentication Protocol:

- Unilateral authentication (client to server) over the secure channel established by SSH TLP
- Authentication is based on the user proving possession of some cryptographic credential:
  - Public key challenge/response required (private key derived from user's pass phrase)
  - Password based authentication should also be supported

SSH Connection Protocol:

- standard interactive shell applications over the mutually authenticated secure channel established by the previous two components

Cryptography in the Real World is Hard!