Cryptography Best Practices

Prof. Bart Preneel
COSIC
Bart.Prneel(at)esat.kuleuven.be
http://homes.esat.kuleuven.be/~preneel
February 2012

© Bart Preneel. All rights reserved

Outline

• 1. Cryptology: concepts and algorithms
• 2. Cryptology: protocols
• 3. Public-Key Infrastructure principles
• 4. Networking protocols
• 5. New developments in cryptology
• 6. Cryptography best practices

Architectures (1a)

• Point to point
• Local
• Small scale

• Number of keys: 1 or \( n^2 \)
• Manual keying

Example: ad hoc PAN or WLAN

Architectures (2a)

• Centralized
• Small or large scale
• Manual keying

• Number of keys: \( n \)
• ! Central database: risk + big brother
• Non-repudiation of origin? (physical assumptions)

Example: WLAN, e-banking, GSM

Symmetric vs. Asymmetric Algorithms

• hardware costs: 3K–100K gates
• performance: 100 Mbit/s – 70 Gbit/s
• keys: 64-256 bits
• blocks: 64-128 bits
• power consumption: 20-30 \( \mu \)J/bit

• hardware costs: 100K-1M gates
• performance: 100 Kbit/s – 10 Mbit/s
• keys: 128-4096 bits
• blocks: 128-4096 bits
• power consumption: 1000-2000 \( \mu \)J/bit
Architectures (3a)
- Centralized
- Small or large scale
- Manual keying

• Number of keys: n + 1/session
• Not Central database: risk + big brother
• Non-repudiation of origin? (physical assumptions)

Example: LAN (Kerberos)

Architectures (4a)
- Decentralized
- Large scale

• Number of keys: n + N^2
• Risks?
• Trust
• Hard to manage

Example: network of LANs, GSM

Architectures (5a)
- Centralized
- Large scale
- Hierarchy

• Number of keys: n + N

Example: credit card and ATM

Architectures (1b)
- Point to point
- Worldwide
- Small networks

• No CA (e.g. PGP)

Example: P2P, international organizations

Architectures (2b)
- Centralized
- Large or small scale

• Reduced risk
• Non-repudiation of origin

Example: B2C e-banking

Architectures (3b)
- Centralized
- Small or large scale

• Reduced risk
• Non-repudiation of origin

Example: B2B and e-ID
Architectures (4b)

- Decentralized
- Large scale
- (Open)
- Key management architecture?
- Trust

Example: B2B, GSM interoperator communication

Architectures (5b)

- Centralized
- Large scale
- Hierarchy
- Open

Example: credit card EMV

When asymmetric cryptology?

- if manual secret key installation not feasible (also in point-to-point)
- open networks (no prior customer relation or contract)
- get rid of risk of central key store
- mutually distrusting parties
  - strong non-repudiation of origin is needed
- fancy properties: e-voting

Important lesson: on-line trust relationships should reflect real-word trust relationships

EMV Static Data Authentication (SDA)

- Three layers:
  - EPI
  - Issuers
  - Cards

Certificate for dynamic data authentication of a credit card

- Unique name owner
- Unique serial number
- Validity period
- Revocation information
- Public key
- Name of issuing CA
- CA’s Digital signature on the certificate
Using Cryptography Well

Bart Preneel

February 2012

Warning about EMV

http://www.cl.cam.ac.uk/research/security/banking/nopin/oakland10chipbroken.pdf

- Pin checking and authentication are not coupled
- EMV PIN verification “wedge” vulnerability

S.J. Murdoch, S. Drimer, R. Anderson, M. Bond,
IEEE Security & Privacy 2010

Network protocols

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>S/MIME</td>
</tr>
<tr>
<td>Presentation</td>
<td>TLS/SSL IPsec PPTP L2TP</td>
</tr>
<tr>
<td>Session</td>
<td></td>
</tr>
<tr>
<td>Transport</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>Data link</td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td></td>
</tr>
</tbody>
</table>

Where to put security?

- Application layer:
  - closer to user
  - more sophisticated/granular controls
  - end-to-end
  - but what about firewalls?
- Lower layer:
  - application independent
  - hide traffic data
  - but vulnerable in middle points
- Combine?

Where to put security? (2)

From: Bob@crypto.com
To: Alice@digicrime.com
Subject: Re: Can you meet me on Monday at 3pm to resolve the price issue?

This proposal is acceptable for me.
-- Bob

Security APIs

- Security module controls access to and processing of sensitive data
  - executes cryptographic commands, e.g. PIN checking, encryption,…
Master key/data key

- Load master 3DES key \( K_M \) (tightly controlled)
- Load data key:
  \[ 3DES_{K_M}(K_1) || 3DES_{K_M}(K_2) || 3DES_{K_M}(K_3) \]
- Send plaintext \( P \) and ask for encryption
  \[ DES_{K_1}(DES^{-1}_{K_2}(DES_{K_3}(P))) \]

Control vectors in the IBM 4758 (1)

- Potted in epoxy resin
- Protective tamper-sensing membrane, chemically identical to potting compound
- Detectors for temperature & X-Rays
- “Tempest” shielding for RF emission
- Low pass filters on power supply rails
- Multi-stage “latching” boot sequence
= STATE OF THE ART PROTECTION!

Features of the IBM 4758

- Control vector: type (e.g., PIN, data, MAC)
  store key of type type as \( E_{K_m + \text{"type"}}(k) \)
  - Output of encryption with key of type “PIN” is never allowed to leave the box
  - Output of encryption with key of type data, MAC, … may leave the box
- High security master key import: 3 shares
  - Import \( K_m \) as \( K_m = K_{m_A} + K_{m_B} + K_{m_C} \)

IBM 4758

Master key import

\[ K_{m_A} \quad K_{m_B} \quad K_{m_C} \]

\[ K_m = K_{m_A} + K_{m_B} + K_{m_C} \]
Fraudulous import

\[ K_{mA} \cdot K_{kB} \cdot K_{mC} = K_{mC} + \text{“data”} – \text{“PIN”} \]

\[ K_{m*} = K_{mA} + K_{mB} + K_{mC*} = K_{m} + \text{“data”} – \text{“PIN”} \]

The attack

Transport PIN key \( k \) from box 1 to box 2

1. Encrypt on box 1, type PIN:
   \[ x = E_{K_{m}} \cdot \text{“PIN”} \cdot (k) \]
2. Decrypt on box 2, type data:
   \[ D_{K_{m*}} \cdot \text{“DATA”} \cdot (x) = D_{K_{m}} \cdot K_{m*} \cdot \text{“PIN”} \cdot (x) = k \]

The system now believes that \( k \) is a key to decrypt data, which means that the result will be output (PINs are never output in the clear).

Lessons learned: security APIs

- Complex – 150 commands
- Need to resist to insider frauds
- Hard to design – can go wrong in many ways
- Need more attention
- Further reading: Mike Bond, Cambridge University
  http://www.cl.cam.ac.uk/users/mkb23/research.html

Key management

- Key establishment protocols
- Key generation
- Key storage
- Key separation (cf. Security APIs)

Key establishment protocols: subtle flaws

- Meet-in-the-middle attack
  - Lack of protected identifiers
- Reflection attack
- Triangle attack

Attack model:

Needham and Schroeder [1978]:

We assume that the intruder can interpose a computer in all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.
Meet-in-the-middle attack on Diffie-Hellman

- Eve shares a key $k_1$ with Alice and a key $k_2$ with Bob
- Requires active attack

$\alpha^{x_1} = (\alpha^{y_1})^{x_1} = (\alpha^{y_1})^{x_1}$
$\alpha^{y_2} = (\alpha^{x_2})^{y_2} = (\alpha^{x_2})^{y_2}$

Entity authentication

- Alice and Bob share a secret $k$

Entity authentication: reflection attack

- Eve does not know $k$ and wants to impersonate Bob

Needham-Schroeder (1978)

- Alice and Bob have each other’s public key $PA$ and $PB$

Lowe’s attack on Needham-Schroeder (1995)

- Alice thinks she is talking to Eve
- Bob thinks he is talking to Alice

Lowe’s attack on Needham-Schroeder (1995)

- Eve is a legitimate user = insider attack
- Fix the problem by inserting B in message 2

- Prudent engineering practice (Abadi & Needham): include names of principals in all messages
- IKE v2 – plausible deniability: don’t include name of correspondent in signed messages: http://www.ietf.org/proceedings/02nov/I-D/draft-ietf-ipsec-soi-features-01.txt

Rule #1 of protocol design

Don’t!

Why is protocol design so hard?

- Understand the security properties offered by existing protocols
- Understand security requirements of novel applications
- Understanding implicit assumptions about the environment underpinning established properties and established security mechanisms

And who are Alice and Bob anyway?

- Users?
- Smart cards/USB tokens of the users?
- Computers?
- Programs on a computer?

If Alice and Bob are humans, they are vulnerable to social engineering

Random number generation

- “The generation of random numbers is too important to be left to chance”
- John Von Neumann, 1951: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"
- Used for
  - Key generation
  - Encryption and digital signatures (randomization)
  - Protocols (nonce)

Key generation: overview
Key generation: hardware entropy sources

- radioactive decay
- reverse biased diode
- free running oscillators
- radio
- audio, video
- hard disk access time (air turbulence)
- manually (dice)
- lava lamps

Risk: physical attacks, failure

Key generation: software entropy sources

- system clock
- elapsed time between keystrokes or mouse movements
- content of input/output buffers
- user input
- operating system values (system load, network statistics)
- interrupt timings

Risk: monitoring, predictable

Key generation: monitoring

- Statistical tests (NIST FIPS 140)
- typical tests: frequency test, poker test, run’s test
- necessary but not sufficient
- 5 lightweight tests to verify correct operation continuously
- stronger statistical testing necessary during design phase, after production and before installation

State update

- Keep updating entropy pool and extracting inputs from entropy pool to survive a state compromise
- Combine both entropy pool and existing state with a non-invertible function (e.g., SHA-512, $x^2 \mod n, \ldots$)

Output function

- One-way function of the state since for some applications the random numbers become public
- A random string is not the same as a random integer $\mod p$
- A random integer/string is not the same as a random prime

What not to do

- use $\text{rand}()$ provided by programming language or O/S
- restore entropy pool (seed file) from a backup and start right away
- use the list of random numbers from the RAND Corporation
- use numbers from http://www.random.org/
- 66198 million random bits served since October 1998
- use digits from $\pi, e, \pi/e, \ldots$
- use linear congruential generators [Knuth]
  - $x_{n+1} = ax_n + b \mod m$
RSA moduli

- Generate a 1024-bit RSA key
  Use random bit generation to pick random an integer \( r \) in the interval \([2^{12}, 2^{13}-1]\)
  If \( r \) is even: \( r := r + 1 \)
  Do \( r := r + 2 \) until \( r \) is prime; output \( p \)
  Do \( r := r + 2 \) until \( r \) is prime; output \( q \)

What is the problem?

How to store keys

- Disk: only if encrypted under another key
  - But where to store this other key?
- Human memory: passwords limited to 48-64 bits and passphrases limited to 64-80 bits
- Removable storage: Floppy, USB token, iButton, PCMCIA card
- Cryptographic co-processor: smart card USB token
- Cryptographic co-processor with secure display and keypad
- Hardware security module
- PUFs: Physical Uncloneable Functions

Secure key storage with non-initialized SRAM

Implementation attacks

cold boot attack

- Why break cryptography? Go for the key, stupid!
- Data reminence in DRAMs
  Least We Remember: Cold Boot Attacks on Encryption Keys [Halderman-Schoen-Heninger-Clark-Paul-Calandrino-Feldman-Appelbaum-Felten’08]
  - Works for AES, RSA,…
  - Products: BitLocker, FileVault, TrueCrypt, dm-crypt, loop-AES

New attack on keys in memory (21/02/08)

- Key is stored in DRAM when machine is in sleep or hibernation
- Option 1: Reboot from a USB flash drive with O/S and forensic tools (retaining the memory image in DRAM), scan for the encryption keys and extract them.
- Option 2: physically remove the DRAM
  - Cool DRAM using compressed-air canister (-50 C) or liquid nitrogen (-196 C)
- Solution: hardware encryption or 2-factor authentication
How to back-up keys

- Backup is essential for decryption keys
- Security of backup is crucial
- Secret sharing: divide a secret over n users so that any subset of t users can reconstruct it

Implementing digital signatures is hard

- ElGamal
- RSA

The risks of ElGamal (1/3)

- ElGamal-type signatures (including DSA, ECDSA)
- public parameters: prime number p, generator g (modulo p operation omitted below)
- private key x, public key y = g^x
- signature (r,s)
  - generate temporary private key k and public key r = g^k
  - solve s from h(m) ≡ x r + k s mod (p−1)
- verification:
  - Signature verification: 1 < r < p and h(m) ≡ y^r r^s mod p

The risks of ElGamal (2/3)

- long term keys: y = g^x
- short term keys: r = g^k
- the value k has to be protected as strongly as the value x
  - Ex 1: NIST had to redesign the DSA FIPS standard because of a subtle flaw in the way k was generated [Bleichenbacher'01]
  - Ex 2: attack on ElGamal as implemented in GPG [Nguyen'03]

The risks of ElGamal (3/3)

- y = g^x
- signature:
  - r = g^k
  - h(m) ≡ x r + k s mod (p−1)
- what if k would be the same every time?
  - h(m1) ≡ x r + k s mod (p−1)
  - h(m2) ≡ x r + k s mod (p−1)
- 2 linear equations in 2 unknowns: easy to solve: yields the signing key x
- one solution: choose k = h(m || x)

How to sign with RSA?

- public key: (n,e)
- private key: d
- s = t^d mod n = t^{1/e} mod n
- But
  - message M is often larger than modulus n
  - RSA(x*y) = RSA(x)*RSA(y)
  - RSA(0) = 0, RSA(1) = 1, ...
- Solution: hash and add redundancy
  - PKCS #1
  - RSA-PSS
RSA Signatures: PKCS #1 v1.5 [source: RSA Labs]

- public key: (n,e)
- private key: d

\[ t = 00 01 ff \ldots ff 00 \]

- Generation of RSA signature on M: \( s = t^e \mod n \)
- Verification of RSA signature \( s \) on M
  \[ t = s^d \mod n \text{ and check that } t \text{ has the required format} \]

Problem: most signature verification software would accept a signature on M of the following form:

- consider RSA with public exponent \( e = 3 \)
- for any hash value \( H \), it is easy to compute a string “Magic” such that the above string is a perfect cube of 3072 bits
- consequence:
  - one can sign any message \( (H) \) without knowing the private key
  - this signature works for any public key that is longer than 3072 bits
- vulnerable: OpenSSL, Mozilla NSS, GnuTLS

Other ways to fool CAs
- [Moxie Marlinspike’09] Black Hat
  - browsers may accept bogus SSL certs
  - CAs may sign malicious certs
- certificate for \( \text{www.paypal.com/uk/kuleuven.be} \) will be issued if the request comes from a kuleuven.be admin
- response by PayPal: suspend Moxie’s account
  - http://www.theregister.co.uk/2009/10/06/paypal_banishes_ssl_hacker/

Fix of Bleichenbacher’s attack
- Write proper verification code (but the signer cannot know which code the verifier will use)
- Use a public exponent that is at least 32 bits
- Upgrade – finally – to RSA-PSS