CNS Lecture 9

Security through mathematics

more public key crypto

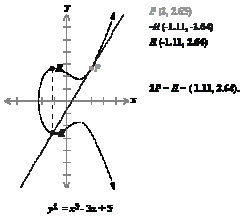
ECC

Applied crypto

ssh

pgp

PKI



Assignment 8 due 10/28/06

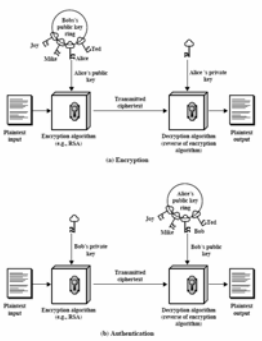
You are here ...

Attacks & Defenses	Cryptography	Applied crypto
• Risk assessment✓	•Random numbers✓	•SSH
• Viruses✓	•Hash functions✓	•PGP
• Unix security✓	MD5, SHA,RIPEMD	•S/Mime
• authentication✓	•Classical + stego✓	•SSL
• Network security	•Number theory✓	•Kerberos
Firewalls,vpn,IPsec,IDS	•Symmetric key✓	•IPsec
• Forensics	DES, Rijndael, RC5	•Crypto APIs
	•Public key	•Coding securely
	RSA, DSA, D-H,ECC	

Public key crypto

- **Diffie-Hellman key establishment**
  - Discrete logs
    $X = g^x$ **mod** p
- **RSA**
  - Factoring integers, $n = pq$
    $c = m^e$ **mod** n
- **DSS**
  - Discrete logs, sign only
- **ECC**

BIG integers (1000-bit)



OpenSSL

- genrsa, gendh, gendsa -- generate keys

```
genrsa -des3 -out ca.key 1024
```

- rsautl -- encrypt/decrypt sign/verify
- Plus the hash (md5, sha) and encrypt (AES, DES) commands
- API for doing rand, big numbers, find prime, D-H, encrypt/decrypt, sign/verify

ECC – elliptic curve cryptography

- **Based on elliptic curve arithmetic (old field of mathematics)**
- **Form of public key encryption**
  - More security per bit than any other public key crypto
  - Efficient hardware implementations
  - Suitable for cryptocards, cell phones, PDAs
  - Free software (few? licensing restrictions)
  - Strength not based on factoring (just in case ☺)
  - Strength/operation similar to Diffie-Hellman
  - Mathematics more complex than RSA/D-H, so smaller keys and faster (10x) in hardware

Stacking balls

- Great moments in elliptic curves
- How many balls do you need to stack in the arrangement 1 on top of 4 on top of 9 on top of 16 ...

$1^2 + 2^2 + 3^2 + 4^2 \ldots + x^2$ so that they form a perfect square when collapsed? The sum is $x(x+1)(2x+1)/6$

So you want $y^2 = (2x^3 + 3x^2 + x)/6$ an elliptic curve!

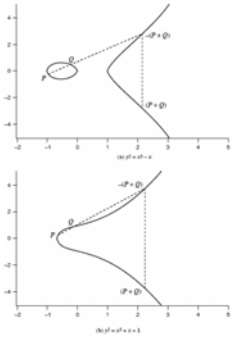
Lots of (x,y) solutions, including negative and fractional tennis balls, but only two solutions in whole tennis balls! ☺

(1,1) or (24,70)

4900 balls in a pyramid 24 balls high, or 70 by 70 square

Elliptic curves

- Elliptic curves are NOT ellipses
- Described by cubic equations of the form
  $y^2 = x^3 + ax + b$ E(a,b)
  P = (x,y) is a point on the curve if (x,y) is in E(a,b)
  -P = (x, -y)
  forms a group over addition if $(4a^3 + 27b^2) \neq 0$
  there is an additive identity O
- Addition over E(a,b) (geometric)
  P + Q where P and Q are points in E(a,b)
  Draw line thru P and Q, where line intersects curve (R), the result is the mirror image (reflection) of R
  Algebraically
  $x_R = \lambda^2 - x_P - x_Q$
  $y_R = -y_P + \lambda(x_P - x_R)$
  where $\lambda = (y_Q - y_P)/(x_Q - x_P)$

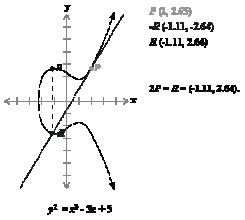


Elliptic curve addition

- To double a point, P+P = 2P =R, (geometric) draw a tangent to P, its reflection is R
- Multiplication is defined as repeated additions nP = P+P+ … +P
- Algebraically

$$x_R = \left(\frac{3x_P^2 + a}{2y_P}\right)^2 - 2x_P$$

$$y_R = \left(\frac{3x_P^2 + a}{2y_p}\right)(x_P - x_R) - y_p$$



Certicom's ECC tutorial

Elliptic curves over Z_p

Equations of form (prime curves)

$y^2 \bmod p = (x^3 + ax + b) \bmod p$ $E_p(a,b)$

forms a group over addition if $(4a^3 + 27b^2) \bmod p \neq 0$ (p is a BIG prime)

Variables and coefficients are in the set of integers [0 … p-1]

Rules for addition:

$x_R = (\lambda^2 - x_P - x_Q) \bmod p$

$y_R = (\lambda(x_P - x_R) - y_P) \bmod p$

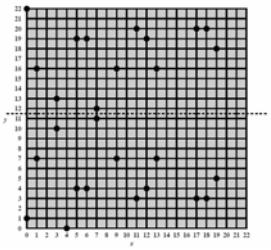
where $\lambda = ((y_Q - y_P)/(x_Q - x_P)) \bmod p$ if $P \neq Q$

or $\lambda = ((3x_P^2 + a)/(2y_P)) \bmod p$ if $P = Q$

Example: $E_{23}(1,1)$

$y^2 = x^3 + x + 1 \bmod 23$

(0, 1)	(6, 4)	(12, 19)
(0, 22)	(6, 19)	(13, 7)
(1, 7)	(7, 11)	(13, 16)
(1, 16)	(7, 12)	(17, 3)
(3, 10)	(9, 7)	(17, 20)
(3, 13)	(9, 16)	(18, 3)
(4, 0)	(11, 3)	(18, 20)
(5, 4)	(11, 20)	(19, 5)
(5, 19)	(12, 4)	(19, 18)



For real crypto, BIG integers (160 bits)

ECC encryption

- $E_q(a,b)$ and point G are published
- each user selects random private key n_a, public key is $K_a = n_a G$
- encode message as a point M on the curve (tricky but doable, PKCS #13)
- Alice encrypts M to Bob using Bob's public key K_b
  generate random r, send the following message to Bob
  $C_m = \{X, Y\}$ where $X = rG$ and $Y = M + rK_b$
- Bob decrypts by calculating
  $Y - n_b X = M + rK_b - n_b rG = M + rn_b G - n_b rG = M$
- Easy to calculate $X = rG$
  attacker has to find r given X and G -- real hard for large 160-bit primes!
  - Sort of discrete logarithm problem for elliptic curves
  - Repeated additions (rather than multiplications as in D-H)
  - Best method to find k given X, use Pollard Rho method, exponential time algorithm with complexity $O(\sqrt{n})$

ECC discrete logs

In the elliptic curve group defined by

$y^2 = x^3 + 9x + 17$ over F_{23},

What is the discrete logarithm k of Q = (4,5) to the base P = (16,5)?

One (brute force) way to find k is to compute multiples of P until Q is found. The first few multiples of P are:

P = (16,5) 2P = (20,20) 3P = (14,14) 4P = (19,20) 5P = (13,10)
6P = (7,3) 7P = (8,7) 8P = (12,17) 9P = (4,5)

Since 9P = (4,5) = Q, the discrete logarithm of Q to the base P is k = 9.

In a real application, k and the modules would be large enough (e.g. 160 bits) such that it would be infeasible to determine k in this manner.

ECC key establishment

D-H equivalent, though 5 to 10 times faster.

n "additions" rather than n multiplications

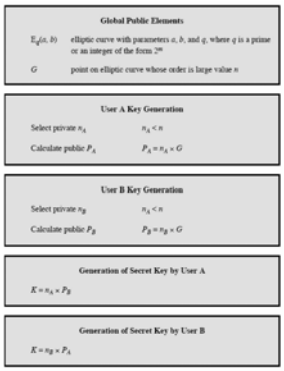
The "generator", base point, G, has a high order, nG = O (n is large)

Algorithms/encodings:

PKCS 13

ANSI X9.62 X9.63

ECDH in OpenSSL (osi)



ECC digital signatures

- TLS and OpenSSL ECDSA
- Algorithm like DSA, generate a verifier of the hash of message
- Sign:
  - Given equation info $E_p(a,b)$ and base point G order n
  - given message hash h
  - Given Alice's private/public key k_A and Q_A (where $Q_A = k_A G$)
  - Generate big random integer z and point on curve $Z = zG = (x,y)$
  - set $r = x \bmod n$ and $s = z^{-1}(h + rk_A) \bmod n$
  - Send message and verifier pair r and s
- Verify
  - Regenerate hash h', calculate $u_1 = h's^{-1}$ $u_2 = rs^{-1} \bmod n$
  - Calculate point on curve $Z' = u_1G + u_2Q_A$
    Z' == Z if h == h' → verified hash the same

$$u_1G + u_2Q_A = h's^{-1}G + rs^{-1}k_AG = h'z(h+rk_A)^{-1}G + rz(h+rk_A)^{-1}k_AG$$
$$= zG\,(h+rK_A)^{-1}(h' + rk_A)$$
$$= zG \rightarrow \text{a point } (x', y')$$

signature valid if $x' \bmod n == r$

Elliptic curves over GF(2^m)

polynomial arithmetic (again!), cubic equation where variables and coefficients are all in GF(2^m) (m =160 for today's crypto)

$y^2 + xy = x^3 + ax^2 + b$ (group if b is non-zero)

Rules for addition: (uses polynomial arithmetic)

P ≠ Q

$x_R = \lambda^2 + \lambda + x_P + x_Q + a$

$y_R = \lambda(x_P + x_R) + x_R + y_P$

where $\lambda = (y_Q + y_P)/(x_Q + x_P)$

P = Q

$x_R = \lambda^2 + \lambda + a$

$y_R = x_P^2 + (\lambda+1)x_R$

where $\lambda = x_P + y_P/x_P$

- Efficient in hardware (add is XOR, multiply is shifts and XORs)

Certicom's ECC tutorial

Orthonormal basis is even faster in hardware for squaring (just a rotate).

Poly arithmetic over GF(2^4)

$g^0 = (0001)$ $g^1 = (0010)$ $g^2 = (0100)$ $g^3 = (1000)$
$g^4 = (0011)$ $g^5 = (0110)$ $g^6 = (1100)$ $g^7 = (1011)$
$g^8 = (0101)$ $g^9 = (1010)$ $g^{10} = (0111)$ $g^{11} = (1110)$
$g^{12} = (1111)$ $g^{13} = (1101)$ $g^{14} = (1001)$ $g^{15} = (0001)$

- Mod irreducible polynomial $(x^4 + x + 1)$
- Elements of set are {0000,0001,0010,0011, … 1111}
- Addition and subtraction just XOR
- Multiplication is done mod $x^4 + x + 1$
- generator g (e.g. 0010) is an element whose powers g^i generate the entire set
- Multiplicative inverse of g^i is $g^{-i \bmod 15}$

$g^7 = 1011$ $g^{-7 \bmod 15} = g^8 = 0101$, to check does 1011 x 0101 = 0001 ?

$(x^3 + x + 1)(x^2 + 1)$ **mod** $(x^4 + x + 1)$

$x^5 + x^2 + x + 1$ (divide by $x^4 + x + 1$), get remainder of 1, so g7 is multiplicative inverse of g^8

ECC over GF(2^4) -- example

- Irreducible polynomial $x^4 + x + 1$
  generator g = (0010)
- Elliptic curve
  $y^2 + xy = x^3 + g^4x^2 + 1$

$g^0 = (0001)$ $g^1 = (0010)$ $g^2 = (0100)$ $g^3 = (1000)$
$g^4 = (0011)$ $g^5 = (0110)$ $g^6 = (1100)$ $g^7 = (1011)$
$g^8 = (0101)$ $g^9 = (1010)$ $g^{10} = (0111)$ $g^{11} = (1110)$
$g^{12} = (1111)$ $g^{13} = (1101)$ $g^{14} = (1001)$ $g^{15} = (0001)$

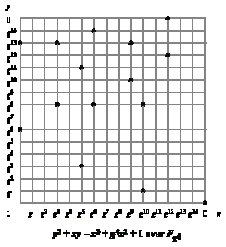
Is (g^5, g^3) a point on the curve?

$(g^3)^2 + g^5g^3 = (g^5)^3 + g^4(g^5)^2 + 1$

$g^6 + g^8 = g^{15} + g^{14} + 1$

(1100) + (0101) = (0001) + (1001) + (0001)

(1001) = (1001)



ECC challenge

- Certicom challenge over Z_p and/or over GF(2^m)

Challenge	End Date	Elliptic Curve Operations	Iterations per second	Machine Days
ECC2-79	Dec. 16, 1997	1.7 x 1012	170000	116
ECC2-89	Feb. 9, 1998	1.8 x 1013	187000	1114
ECC2-95	May 21, 1998	2.2 x 1013	149000	1709
ECC2-109	April 27, 2004	2600 computers 17 months		
ECCp-79	Dec. 6, 1997	1.4 x 1012	314000	52
ECCp-89	Jan. 12, 1998	2.4 x 1013	388000	716
ECCp-97	Mar. 18, 1998	2.0 x 1014	361000	6412
ECCp-109	Nov 6, 2002			29.7M (50K x 594)

- ECC2-97 sept 99, ECC 97 bit private key found, 16000 MIPS-yrs (more than 512-bit RSA challenge). used 740 computers, 40 days

Key Size	MIPS-Years
150	3.8×10^{10}
205	7.1×10^{18}
234	1.6×10^{28}

(a) Elliptic Curve Logarithms using the Pollard rho Method

Key Size	MIPS-Years
512	3×10^{4}
768	2×10^{8}
1024	3×10^{11}
1280	1×10^{14}
1536	3×10^{16}
2048	3×10^{20}

(b) Integer Factorization using the General Number Field Sieve

equivalent key sizes		
Symmetric	RSA	ECC
40		
56	400	
64	512	
80	768	
90	1024	160
120	2048	210
128	3072	256
256	15360	512

Using ECC

- Excellent for hardware implementations
- OpenSSL supports ECC in the API and at the command level
  - ECCDH and ECCDSA over either GF(p) or GF(2^m)
  - Lots of different curves with different strengths (160 to 571 bits)
  - API includes routines for encoding/conversion and SSL/TLS support
  - Commands for keygen/encodings
  - ECC in X509 certs
- OpenSSL ECC performance in https
  - Also see openssl speed
- The test of time?

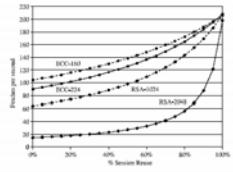


Figure 1: Throughput v/s session reuse plot.

ECC keys and OpenSSL

- Your ECC public/private key includes info describing the equation and coefficients $E_q(a,b)$ and base point G
- Your private key is a big random integer n (keep this secret)
- Your public key is the point on the curve nG (and equation info)
- The ECC standards (and OpenSSL) provide lots of "suitable" curves to choose from (see `openssl ecparam -list_curves`)
- To generate an ECC key pair

```
openssl ecparam -out ec_key.pem -name prime192v1 –genkey
```

and you can have a CA sign you public ECC key and create X509 cert

Your ECC keys

```
openssl ec -in ec_key.pem -noout -text
read EC key
Private-Key: (192 bit)
priv:
    56:7d:0a:64:7e:ed:44:b0:ea:2a:1a:90:3a:3c:d5:
    9e:47:1a:61:49:87:25:73:f8
pub:
    04:b0:d2:ca:ae:e9:d0:f8:f3:95:00:97:40:a3:d4:
    9b:89:2e:93:ab:bb:a2:49:75:ce:b2:22:a4:a6:be:
    9b:31:ae:10:f2:ce:a2:13:16:8d:61:c7:29:91:ab:
    27:56:10:50
ASN1 OID: prime192v1
```

Your pub key is actually a "point" (x,y), so there is some encoding

The OID tells the reader which equation used {p,a,b,G,n,h}

OpenSSL ECC API

- OpenSSL version 0.9.8 or later
- ECDH

```
EC_KEY *ecckey = EC_KEY_new_by_curve_name(nid);
const EC_GROUP *group;
group = EC_KEY_get0_group(ecckey);
EC_KEY_generate_key(ecckey);  //pub key
EC_POINT_get_affine_coordinates_GFp(group,
          EC_KEY_get0_public_key(ecckey), x, y,ctx)
ECDH_compute_key(…) // shared secret
```

- ECDSA

```
EC_KEY    *key ;        ECDSA_SIG *signature;
EVP_DigestFinal(&md_ctx, digest, &dgst_len);
signature = ECDSA_do_sign(digest, 20, key);
if (ECDSA_do_verify(digest, 20, signature, key) != 1)
```


Public key crypto

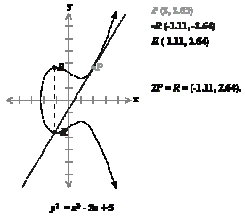
- public key crypto meets our security requirements
  - privacy
  - integrity
  - authentication
  - non-repudiation
- choose from RSA, ElGamal, ECC or sign with DSA
- key agreement with D-H
- need random numbers, primes, generators, BigInteger software
- standards (IEEE P1363 and PKCS)
- symmetric-key crypto (DES, AES,...) still needed for speed
- Solves symmetric key distribution problem
  - but carrying your private key around is still a problem
- public key management (PKI) still unsolved ... (later)

- Diffie-Hellman
  - Key agreement
  - Discrete logs
- RSA
  - Sign/encrypt
  - Factoring integers
- DSS
  - Sign
  - Discrete logs
- ECC
  - Sign/encrypt/key agreement
  - Elliptic curves

Number theory for crypto✓

- Mod arithmetic, gcd, CRT (shift cipher, Hill, RSA, D-H, ECC)
- Polynomial arithmetic over GF(2ⁿ) (LFSR, ECC, AES, CRC)
- Primality tests, irreducible polynomials, generators
- Random number generation (keys, IV, blinding, k for DSS)
- **BIG Integer** arithmetic
- Nonlinear Boolean functions (Bent)
- Factoring and discrete logs
- Elliptic curves
- Computational complexity

Security through mathematics



Crypto toolbox✓

You have tools for building secure applications

- fast symmetric key encryption
- hash functions
- random numbers, prime testing
- public key crypto
- Big integer math libraries/methods
- algorithms for message authentication, key exchange, user authentication
- rules for encoding, padding, interoperability (more later)
- no standard API (more later)

next: applied crypto

Applied crypto – secure applications

security features ≠ secure features

Evaluation criteria

- privacy, integrity, availability
- encryption (symmetric and public key)
- message authentication (hashes)
- user authentication (mutual?)
- key size, distribution, re-new
- encoding/protocols
- random number generation
- prime generation
- API
- replay defense
- offline attacks
- Strong software engineering (secure design and coding)

SSH & PGP

ssh and pgp are sample applications that use public key crypto, hashing, symmetric key encryption, random numbers, and do their own public key management

ssh

- crypto negotiation
- session key establishment
- authentication protocols
- public and secret key
- manual public key mgt.

pgp

- message authentication/encryption
- public and secret key
- user-based key certification and mgt., key rings, trust
- Message envelopes

ssh

Secure Shell

- public domain encrypted shell
- runs on most UNIX systems
- Windows client
- replaces rsh rlogin rcp ftp
- can tunnel other TCP services (X)
- developed in Finland, now OpenSSH
- uses RSA/DSA, DES/3DES/IDEA/RC4/Blowfish/TSS
- optional compression (gzip)
- v1 uses crc32 for MAC (encrypted)
- Newer versions based on OpenSSL lib (not SSL protocol)

components: sshd ssh scp ssh-keygen ssh-agent sftp

SSH features

- strong/mutual authentication (RSA) defeats IP impersonation and DNS hacks
- encryption defeats session hijacking, passwords in the clear, privacy violations
- uses encrypted tunnel for other services
- public key generation and session key generation require strong random (unpredictable) numbers

ssh support files

HOST /etc/

- sshd_config what's allowed
- ssh_host_key ssh_host_key.pub (ssh_host_dsa_key) – at install time
- ssh_random_seed
- ssh_known_hosts
- (server key -- regenerated each hour, not saved)

USER .ssh/

- identity d u p q u= p^{-1} mod q
- identity.pub (ascii) bits e n
  id_rsa or id_dsa (plus .pub) for OpenSsh
- random_seed
- known_hosts
  hostname/pub key
- authorized_keys
  pubkey + label

ssh config files

sshd_config -- what sshd will allow

```
Port 22
HostKey /etc/ssh_host_key
RandomSeed /etc/ssh_random_seed
ServerKeyBits 768
LoginGraceTime 600
KeyRegenerationInterval 3600
PermitRootLogin no
X11Forwarding yes
SyslogFacility DAEMON
RhostsAuthentication no
RhostsRSAAuthentication yes
RSAAuthentication yes
PasswordAuthentication yes
PermitEmptyPasswords yes
```


ssh protocol

- TCP port 22 (sshd)
- server sends version info
- client replies with version info
- server sends host and server pub keys, cookie, supported encryption and authentication
- client checks known_hosts
- client selects and sends encryption mode, cookie, and random session key encrypted with host AND server pub key
- server sends confirmation encrypted with session key
- server is authenticated
- host key added to your known_hosts on client, warns if no match
- Subsequent messages:
  - Encrypted
  - CRC used for integrity (v1)
  - all messages have some random padding

Next, user authentication begins (either UNIX password or user RSA key)

- new server keys each hour

ssh authentication

- Installation generates host pub/priv key ssh-keygen
- RSA host authentication (mutual)
  - use .rhosts/.shosts or hosts.equiv
  - Server or user's known_hosts must have client host's pub key
  - server sends random challenge (cookie) encrypted with client host's pub key
  - client responds with MD5 hash of challenge
- user RSA authentication
  - user generates pub/priv key ssh-keygen
  - stored in .ssh/
  - private key encrypted with IDEA/3DES using passphrase
  - authorized_keys in your .ssh/ on server
  - server sends random challenge (cookie) encrypted with your pub key
  - client responds with MD5 hash of challenge
- UNIX password (but session is already encrypted)
- Handy authentication agent (ssh-agent) – only type your passphrase once
  - E.g., in .login start your X server with ssh-agent startx
  - When you start login, from an xterm, ssh-add (asks for your passphrase)
  - All ssh commands using user-RSA key will be handled automatically

An ssh (v1) session

```
ssh -v
SSH Version 1.2.17 [i686-unknown-linux], protocol version 1.5.
Standard version.  Does not use RSAREF.
Reading configuration data /etc/ssh_config
ssh_connect: getuid 23673 geteuid 0 anon 0
Connecting to whisper [128.169.93.200] port 22.
Allocated local port 1022.
Connection established.
Remote protocol version 1.5, remote software version 1.2.26
Waiting for server public key.
Received server public key (768 bits) and host key (1024 bits).
Host 'whisper' is known and matches the host key.
Initializing random; seed file /home/thistle/dunigan/.ssh/random_seed
Encryption type: idea
Sent encrypted session key.
Received encrypted confirmation.
Trying rhosts or /etc/hosts.equiv with RSA host authentication.
Remote: Rhosts/hosts.equiv authentication refused: client user 'dunigan',
  server user 'dunigan', client host 'thistle.epm.ornl.gov'.
Server refused our rhosts authentication or host key.
No agent.
Trying RSA authentication with key 'dunigan@icarus'
Received RSA challenge from server.
Enter passphrase for RSA key 'dunigan@icarus':
```


Random numbers in ssh

- Used for prime generation/testing, cookie, session key

From randome.c

```
random_get_noise_from_command(state, uid, "ps laxww 2>/dev/null");
if (time(NULL) - start_time < 30)
  random_get_noise_from_command(state, uid, "ps -al 2>/dev/null");
if (time(NULL) - start_time < 30)
  random_get_noise_from_command(state, uid, "ls -alni /tmp/. 2>/dev/null");
if (time(NULL) - start_time < 30)
  random_get_noise_from_command(state, uid, "w 2>/dev/null");
if (time(NULL) - start_time < 30)
  random_get_noise_from_command(state, uid, "netstat -s 2>/dev/null");
if (time(NULL) - start_time < 30)
  random_get_noise_from_command(state, uid, "netstat -an 2>/dev/null");
if (time(NULL) - start_time < 30)
  random_get_noise_from_command(state, uid, "netstat -in 2>/dev/null");
```

- then mixes using MD5, some randomness saved in file random_seed
- uses GNU's multiprecision library
- Openssh uses openssl lib's for random (based on /dev/random)

Prime tests in ssh

Generating primes p and q for RSA (rsa.c)

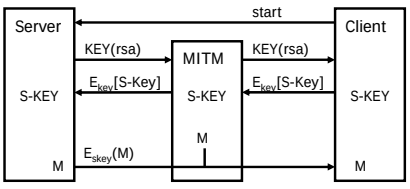
- generate big-integer random number
- set 2 highest bits, low bit
- See if divisible by small primes (1050) (print . if passed test)
- fermat test for witness 2
  - not prime if $2^n \bmod n \neq 2$
  - if passed test, print +
- 20 Miller-Rabin tests (GMP)
  - print + (distance is number of tries x 2)
- confirm $p \neq q$ and not too close and are relatively prime
- confirm RSA encrypt/decrypt works
- If any of the tests fail, generate a new random number (+2) and start over …

Key generation

```
cetus3a% ssh-keygen -b 1024 -f test -N ''
Initializing random number generator...
Generating p:  ..................................................
.........................++ (distance 1046)
Generating q:  .......++ (distance 114)
Computing the keys...
Key generation complete.
Your identification has been saved in test.
Your public key is:
1024 33 14393845200713617792488580070948770049655935640447476454297355052341192 5
98882643223198822733183490417817736100393742900939658914396345257607272735660285
82461520936180182589295082705770544338898103088003747712927386236806723033247571
58624715199863545171529247567102098771947439443619083792269274291408340652383
  dunigan@cetus3a
Your public key has been saved in test.pub
```


SSH v1 vulnerabilities

- Cipher text exposes size of username and password
- Vulnerable to man-in-the-middle attack
  - –Assumes client doesn't have server's key already
- Use v2



ssh updates

- ssh v2 (OpenSSH)
- SecurID, s/key, and Kerberos support
- blowfish, AES, RC4
- SHA1
- D-H, DSA
- use /dev/random and OpenSSL's libssl
- new protocols, sftp
- uses hash (hmac-md5 or hmac-sha) instead of CRC
- speedups

- IETF's secure shell (secsh)
- Also versions that will use public key/ssl or one-time passwords (fob)

- **Use ssh** and get latest version (there are/have been bugs/holes)

PGP implementation

- key generation
- files and key representation (pgformat.doc)
- signatures
- encryption
- encoding (radix-64 ASCII)
- trust model
- uses RSA, IDEA, MD5, ZIP (DSA, 3DES with GNUpg)
- big number lib (mpilib.c)
- multi OS, multi architecture

- RFC 3156

PGP services

- Tool for encrypting/signing files
- Used with email
- Based on public keys and a web of trust

Function	Algorithms Used	Description
Digital signature	DSS/SHA or RSA/SHA	A hash code of a message is created using SHA-1. This message digest is encrypted using DSS or RSA with the sender's private key and included with the message.
Message encryption	CAST or IDEA or Three-key Triple DES with Diffie-Hellman or RSA	A message is encrypted using CAST-128 or IDEA or 3DES with a one-time session key generated by the sender. The session key is encrypted using Diffie-Hellman or RSA with the recipient's public key and included with the message.
Compression	ZIP	A message may be compressed, for storage or transmission, using ZIP.
Email compatibility	Radix 64 conversion	To provide transparency for email applications, an encrypted message may be converted to an ASCII string using radix 64 conversion.
Segmentation	–	To accommodate maximum message size limitations, PGP performs segmentation and reassembly.

PGP keys

- select a distinguished name (user id)
- initially generate RSA public/private key pair
  - –key strokes for random source (original PGP)
  - –find RSA primes and exponents
  - –large integers (1024 bit)
- private key encrypted with IDEA/AES using passphrase in .pgp/secring.pgp (.gnupg/ for GnuPG)
- public key and other's public keys on your public key ring .pgp/pubring.pgp
- keyid is last 64 bits of RSA modulus n
- key fingerprint is MD5 hash of public key (verifier)

PGP keys

public key

- creation timestamp
- validity period
- algorithm (RSA)
- modulus n
- exponent e

user id

- length of userid string
- ASCII userid (e.g. email address)

Can have multiple userid's associated with a public key

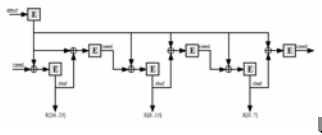
private key

- creation timestamp
- validity period
- algorithm (RSA)
- modulus n
- exponent e
- secret exponent d
- factor p
- factor q
- $u = p^{-1} \bmod q$

Multiprecision integers: number of bits (16-bit number), bytes (MSB first)

Random numbers in PGP

- Used for key generation, session keys, IV
- noise.c random.c randpool.c (v 2.6.2)
  - –key strokes, high res time (dtbuf)
  - –randomness retained in **randseed.bin** (rseed)
  - –data from file pre/post washed with CFB
  - –updated any time user does keyboard input
  - –updated with MD5 of file being encrypted
  - –data stirred with MD5
  - –uses X9.17 but with CAST-128
  - –**randseed.bin** updated



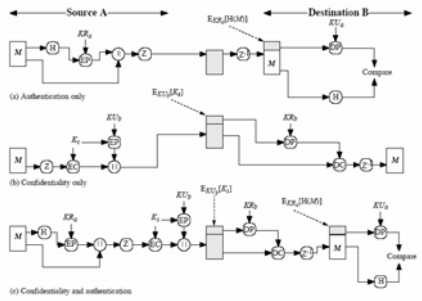
gpg randomness

- Random numbers for public key generation and message keys
- *cipher/random.c* based on Gutman's paper
- Uses /dev/urandom, seed file + pid, time, and clock
- Mixes pool with RIPEMD-160
- Wipes stack and prefers "secure memory" (no swap)
- Pool updated whenever key requested for encryption or secure hash (DSS k)
- State track current entropy of pool
- Application can request strong entropy (slower)
- Saves pool to file ~/.gnupg/**random_seed**

PGP primes

genprime.c
- generate random n-bit number p
- set two hi-bits to 1, use 3 mod 4 numbers (Blum)
- verify p is not divisible by all primes less than 2000
- perform four Fermat tests, for any random x
  if $x^{p-1} \bmod p \neq 1$ then p is not prime
  if fails, add 4 and try again
- find prime q that is not too close to p
- verify RSA encrypt/decrypt works !
- this is what takes time when you first generate your PGP keys

PGP crypto



PGP signatures

Signing a message (-sat)
- hash (MD5) message, sig type, and time
- encrypt hash plus some padding with your private key (PKCS 1)
- encode in radix-64 ASCII for email transport (canonical)
- includes 24-bit CRC with pad (version 1)
- receiver can verify by re-computing hash, decrypting signature with your public key and comparing hash values
- -sa uses ASCII encoding (NOT encryption), need PGP to decode
- -sat leaves in clear text, don't need PGP to read.

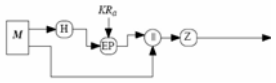
To make a detached signature file (e.g., for authentication of a .tar file)

gpg –sb mystuff.tar

Creates mystuff.tar.sig

```
-----BEGIN PGP SIGNED MESSAGE-----
whatever
-----BEGIN PGP SIGNATURE-----
Version: 2.6.2

iQCVAwUBOLQjXux/7s+CagJ9AQHFtAP/dMHUq6HmWFqqtrpDPt3U+5ol2CNqhQDA
jgmxKD8laDp/UJIPN2PYvrLXCjDLmVxRuLaba3GJDrC2hQ1PC4W0zsgJyHsQo2Un
R7a/s3RbiT/WwmKC0eImopqYUeBxOBeas2pj0wtCcDgcQoQxlhbvGQCETqVDNUjF
GjAztMHl1gM=
=BKJ4
-----END PGP SIGNATURE-----
```



PGP signature

signature type
timestamp
key id of signer (64 bits)
algorithms (RSA, MD5)
verifier (first 2 bytes of hash)
encrypted hash

Encrypted hash is RSA-encrypted multiprecision integer

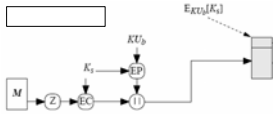
Signing a key (-ks)

says you believe this public key is associated with this user id
-hash (MD5) of public key pkt and user id pkt, sig type, and time
-encrypt hash plus some random padding with your private key
(ref PKCS 1)

signed key is a "certificate"

PGP encryption

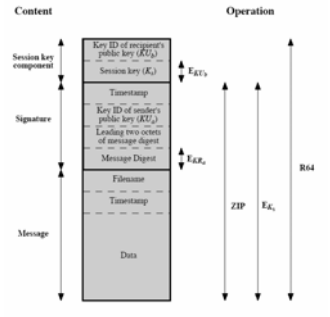
Encrypting a message (-ea)



- generate a message key (random)
- compress (ZIP) message
- encrypt message with message key
  - IDEA, 64-bit CFB (faster than RSA)
  - IV of 0
  - 8 random bytes, verifier (2B), message
- encrypt (RSA) message key with recipient's public key (one per recipient)
  - keyid (64 bits)
  - encrypted IDEA key (PKCS 1)
- encode in radix-64 ASCII (-a)

Encrypting a file (-c)
- IDEA/3DES/AES
- MD5 hashed passphrase as key
- options for over-writing plaintext file
- PGP is also careful about zeroing out secrets in memory.

PGP message format



PGP compression

- Why compress?
  - –Hide plain text
  - –Reduce message size
    - Faster encryption/decryption
    - Less storage space
    - Faster transmission
- PGP compresses with ZIP
  - –Replaces repeated substrings with a short "code word"
  - –Compressed data includes the code mappings and the compressed data

PGP encoding – radix-64

- Encoding binary data as ASCII (pgp, s/mime) – keep mail programs happy
  - –uuencode
  - –radix-64
- radix-64 encodes 6-bits at time (6 bits to 8-bit character)

6-bit value	character encoding	6-bit value	character encoding	6-bit value	character encoding	6-bit value	character encoding
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/
						(pad)	=

Attacking PGP

- Decrypting a message
  - –attack 3DES/IDEA/blowfish
- Recover the message key encrypted with the receiver's pub key
  - –attack RSA
  - –attack random number generator/protocol
- Forge a message
  - –attack RSA
  - – find collision on MD5/SHA
- Steal your private-key file
  - –Attack symmetric key encryptor (3DES etc.)
  - –Dictionary attack on your passphrase
- Keyboard sniffer to get your passphrase
- Trojan horse to get your passphrase

PGP trust

How do you know it's Tom's key?

- verify key (voice, fingerprint)
  or someone you "trust" has signed it (a "certificate")
- PGP asks you (or gpg --edit-key)

```
Do you want to certify any of these keys yourself (y/N)?
Are you sure key belongs to xxxx?
Would you trust xxxx as an introducer (1-4)?
```

- two trusted objects
  - –certainty of key (KEYLEGIT). It is Bob's key
  - –trust in user as an introducer (OWNERTRUST)

Trust levels	
1.	I don't know
2.	I do NOT trust
3.	I trust marginally
4.	I trust fully

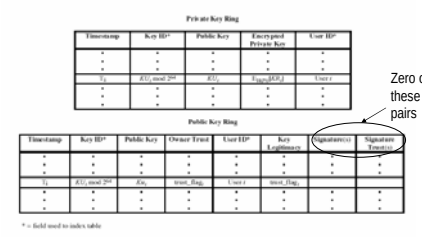
PGP web of trust

- key is trusted if signed by people/keys you trust
- PGP will recalculate trust periodically
- key may become trusted if "enough" good certificates
- extracted key includes certificates (key envy) (no trust info)
- key signing parties
- what if no trust ?
- compromised keys (sig type)
- scalable?
- web/mail key servers (Bill Clinton)
  - –Lots of signatures saying it's legit (JoeTheHacker, BobThehacker ….)

When you sign Bob's key, you're saying you trust that this key belongs to Bob.

Will contrast with certificates signed by a Certificate Authority (CA) and a formal Public Key Infrastructure (PKI)

PGP key rings



Accessed by either User ID or Key ID

- **Timestamp** when entry was generated
- **Key ID** least significant 64 bits of pub key (last 64 bits of modulus n)
- **User ID** identifies owner, e.g. email, multiple user ID's can be associated with single public key

PGP key rings

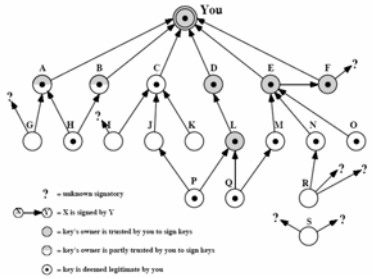
- public key ring includes keys and certificates (others who have signed the key)

certificate = signed key, means signer verifies that this key belongs to user

```
        OWNERTRUST    KEYLEGIT    SIGTRUST    SIGTRUST
(pub key, trust)(Tom@, trust)(sig1,trust)(sig2,trust)...
(pub key, trust)(Bob@, trust)(sig1,trust)(sig2,trust)...
(pub key, trust)(Alice@, trust)(sig1,trust)(sig2,trust)...
(pub key, trust)compromise(Bill@, trust)(sig1,trust)(sig2,trust)...
```

- **OWNERTRUST** first trust is introducer trust (manual) (1-4)
- **KEYLEGIT** key ownership trust, calculated from certificates
- **SIGTRUST** certificate's trust copied from signer's OWNERTRUST (if signer's key is on our ring)
- you could sign it (ultimate trust)
- Periodically PGP re-calculates the SIGTRUST and KEYLEGIT fields

PGP trust model



You have signed A-F
You assigned full trust D,E,F,L
You assigned partial trust A,B
You don't trust C to sign
You think P,Q,N,O are legit keys

PGP calculated keylegit for H
S is orphaned (from key server?)

Revoking your public key

- Someone has figured out your passphrase and copied your private key, so they can now sign things saying they are YOU!
- You issue a **revocation certificate**
  - –just a signature certificate with a bit to indicate the key is revoked
  - –Badguy could issue one too, but what's the point
- If you forget your passphrase or lose your private key ☹, then you can't issue the revocation certificate – some suggest making a revocation certificate and stashing it away.

But how do you distribute the revocation cert.? ....
ah, therein lies the problem – more PKI stuff next time

Key management

- key generation
- key length
- key lifetime, archiving
- key distribution
  - –secret key
  - –KDC
  - –PKI
- protecting private/secret keys
- Public Key Infrastructure (PKI) issues
- key revocation
- key recovery

Key distribution

- secret keys -- out of band, splitting
- key update using old key ☹ or key encrypting key
- Diffie-Hellman
- individual distribution (PGP)
- public key server/directory (PGP/X509)
- KDC (Kerberos/DCE) -- must be secure
  - –get keys from trusted server with secure protocol
- public key certificates
  - –signed (by whom?)
  - –simple directory service

SSH/PGP keys

- ssh -- manual key mgt., no certificates, encrypt priv. key
- pgp -- manual key distribution, web of trust (friends), certificate is signed hash of (id, pub key), key ring contains others signature on your key … doesn't really scale

A PGP "certificate" – says signer believes this public key belongs to the specified userid

```
version
sig. type
time
validity period
key ID of signer
public key alg. (RSA, DSA)
hash alg. (MD5, SHA)
verifier
signed hash of (userid, public key)
```


PKI

public key infrastructure

- **certificate authority (CA)** -- designated key signing authority
- distinguished names and format/encodings
- certificate and CRL distribution (X.509, LDAP, email)
- certification policy (ID assurance)

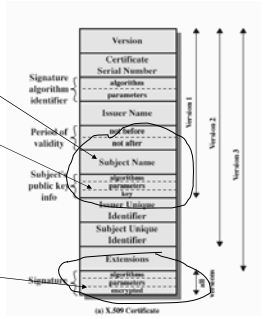
PKI – how it works

- you generate your pub/priv pair
- present yourself and public key to CA
- CA signs your public key (certificate), certifying your distinguished name (X.500) with your key

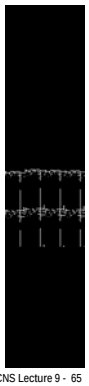
  C=US,O=UTK,OU=CS,CN=dunigan
- certificate distributed (don't need secure repository)
- CA key "known" (e.g., built in to your browser)
- others fetch your certificate and perhaps CA's certificate
- others trust your signature if they trust your CA
- should fetch/check revoked key list (CRL) too
- might have more than one name/certificate

Public Key certificate (X509)

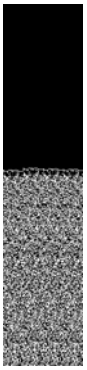
- Certificate == a link between your identity and your public key
- based on X509.3
- uses ASN.1 /DER encoding
- can use SHA/DSA, KEA, RSA/MD5, (D-H?)
- Start/expire dates (netscape expirations)
- v3 extensions include info on key and policy, subject and issuer attributes, and constraints on certification path and whether "required"
- keys for hosts, DNS, applications, URLs, email names
- Cert extensions for authorization
- Signature at end is hash of certificate signed (encrypted) with certificate authority's private key



MD5 collisions and X509



- Recall, code exists to create colliding files with MD5!
- MD5 signature on 2 X509 cert files
  - –Right, 2 files eventually differ (avalanche)
  - –Left, 2 files collide, but end up with same hash
  - –Collision software has to search a while to find right stuff to add to 2nd file
  - –Takes advantage of length extension property of hash algorithms
- Bad news! Use SHA-256?



Encoding overview

- ASN.1/DER/BER
- PKCS
- PEM + base64
- S/MIME
- Also IETF PKIX and IEEE P1363 , specs for public key crypto (PKC)
  - –Objective: interoperable security
  - –Scope (functional specs):
    - Crypto parameters, encodings, and keys
    - Key agreement (D-H, MQV)
    - Signatures (RSA, ECC, DSA, N-R, R-W)
    - Encryption (RSA, ECC)
  - –Variants
    - EKE, SPEKE, SRP
    - Lattice-based PKC

ASN.1/DER encoding used for X509 certs

- ASN.1 – notation for describing data structures (C-like)
- Distinguished Encoding Rules (DER) subset of Basic Encoding Rules (BER)
  Tag, length, value encoding for strings, big integers, date, time,etc.
  platform-independent data representation

```
0x01 -- [0000|0001], [BOOLEAN] GetRequest.lock
0x01 -- [0000|0001], length 1
0x00 -- [0000|0000] value FALSE

0x04 -- [0000|0100], [OCTET STRING] GetRequest.url
0x16 -- [0001|0110],length 22
[/ses/magic/moxen.html] - value
```

Slow/ugly in software (libraries provided in OpenSSL)

X509v3 ASN.1 cert

```
Certificate ::= SEQUENCE {
  tbsCertificate TBSCertificate,
  signatureAlgorithm AlgorithmIdentifier,
  signatureValue BIT STRING
}

TBSCertificate ::= SEQUENCE {
  version [0] EXPLICIT Version DEFAULT v1,
  serialNumber CertificateSerialNumber,
  signature AlgorithmIdentifier,
  issuer Name,
  validity Validity,
  subject Name,
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL,
           -- If present, version shall be v2 or v3
  subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL,
           -- If present, version shall be v2 or v3
  extensions [3] EXPLICIT Extensions OPTIONAL
           -- If present, version shall be v3
}
```


X509 in OpenSSL

- Create RSA pub/private key (no protection on private key)
- Saved in PEM format file (later)
- You can also pre-specify info in a config file for openssl

```
openssl req -new -x509 -days 3650 -nodes -out mycert.pem -keyout mykey.pem
Generating a 1024 bit RSA private key
..........................................++++++
.......++++++
writing new private key to 'mykey.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [GB]:US
State or Province Name (full name) [Berkshire]:TN
Locality Name (eg, city) [Newbury]:Knoxville
Organization Name (eg, company) [My Company Ltd]:UT
Organizational Unit Name (eg, section) []:CS
Common Name (eg, your name or your server's hostname) []:Bubba
Email Address []:bubba@utk.edu
```


X509 in OpenSSL (examining a certificate)

```
openssl x509 -in mycert.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=US, ST=TN, L=Knoxville, O=UT, OU=CS, CN=Bubba/emailAddress=bubba@utk.edu
        Validity
            Not Before: Apr 21 21:55:08 2006 GMT
            Not After : Apr 18 21:55:08 2016 GMT
        Subject: C=US, ST=TN, L=Knoxville, O=UT, OU=CS,
 CN=Bubba/emailAddress=bubba@utk.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:e5:a1:29:2f:89:7c:08:24:2e:22:55:1c:c4:01:
                    f4:68:52:f7:94:59:3b:c4:d8:82:01:65:c9:db:88:
                    …
                    cc:51:8c:32:98:5c:a9:b7:2f
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
            A7:A1:3F:1D:47:BC:C9:01:C1:78:AD:D1:5C:72:86:C3:81:5A:21:44
            X509v3 Authority Key Identifier:
            keyid:A7:A1:3F:1D:47:BC:C9:01:C1:78:AD:D1:5C:72:86:C3:81:5A:21:44
            DirName:/C=US/ST=TN/L=Knoxville/O=UT/OU=CS/CN=Bubba/emailAddress=bubba@utk.edu
            serial:00

            X509v3 Basic Constraints:
            CA:TRUE
    Signature Algorithm: md5WithRSAEncryption
        9e:f1:29:9a:17:0f:d5:90:2a:e2:04:8c:1a:7d:42:9c:72:20:  ….
```


ECC in X509 cert

```
Certificate:
  Data: Version: 1 (0x0) Serial Number: ed:f9:85:3f:69:9e:ac:e1
  Signature Algorithm: ecdsa-with-SHA1
  Issuer: C=US, ST=CA, L=Mountain View, O=Sun Microsystems
  Validity Not Before: Dec 6 21:30:14 2005 GMT
    Not After : Jan 14 21:30:14 2010 GMT
  Subject: C=US, ST=CA, L=Mountain View, O=Sun Microsystems
  Subject Public Key Info:
  Public Key Algorithm: id-ecPublicKey Public-Key: (161 bit)
  pub:
  04:eb:e0:7c:77:eb:bf:f7:95:d0:41:89:35:7f:e9:
  67:d5:b9:a4:63:55:85:2b:16:d7:17:2e:ab:05:d3:
  8c:f7:c8:bd:dd:c3:84:dc:c1:29:d0
  ASN1 OID: secp160r1
  Signature Algorithm: ecdsa-with-SHA1
  30:2d:02:15:00:80:81:d1:c8:84:88:18:ee:76:44:b9:33:e4:
  e0:0d:1a:30:4b:dc:2a:02:14:31:76:dd:e9:26:83:4c:72:1d:
  2c:04:6b:f2:66:fa:4f:12:a5:b8:94
 -----BEGIN CERTIFICATE-----
 MIICEzCCAdICCQDt+YU/aZ6s4TAJBgcqhkjOPQQBMIGrMQswCQYDVQQGEwJVUzEL
 MAkGA1UECAwCQ0ExFjAUBgNVBAcMDU1vdW50YWluIFZpZXcxJjAkBgNVBAoMHVN1
 ....
 SM49BAEDMAAwLQIVAICB0ciEiBjudkS5M+TgDRowS9wqAhQxdt3pJoNMch0sBGvy
 ZvpPEqW4lA==
-----END CERTIFICATE-----
```


PKCS

Public-Key Cryptography Standard

encoding standard (not really, owned/motivated by RSA)

- How to encode public keys (big integers), signatures, algorithms
- Rules for padding

uses ASN.1

(1) RSA encodings
(3) Diffie-Hellman
(5) secret key encryption
(6) key certificate syntax (deprecated by X.509v3)
(7) digital envelopes (see also IETF PKIX CMS)
(8) private key syntax
(9) attributes for 6,7,8, 10
(10) certification request format
(11) crypto token API (competitor: Microsoft CSP)
(12) key file format
(13) ECC (signature, key agreement, encryption, encoding, algorithms)
(14) Pseudo random number generation
(15) Crypto tokens

PKCS 1 RSA

How to use RSA (RFC 2437) encodings for
- public key/private key
- signature
- short-encrypted message (key)
- short signed message (MAC)

Format of message to be encrypted (bytes)

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|2|8-random bytes |0| data (e.g., session key)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Format of message to be signed

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|1|8 bytes of FF  |0| ASN.1 digest type and digest
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

deals with several threats

- random padding defeats guessable messages
- diff. padding for each recipient defeats multiple copies when e=3
- numbers not smooth (small primes)
- interoperable encodings

PKCS 5 – secret key

Password-Based Cryptography Standard
- Version 1.5, 1993, Version 2.0, 1999
- Oriented towards protection of private keys
- Does not specify a standard for password format
- Protocols for salt, hash, padding
- Encryption schemes
  - –PBES1 with DES or RC2 in CBC plus padding
  - –PBES2 with arbitrary encryption scheme
- MAC scheme

Pad with 1 to blocksize bytes
Pad value is number of pad bytes
x x x x x 3 3 3
If plaintext multiple of blocksize, then an extra block is added.
Example, OpenSSL EVP

PKCS 7 – crypto message syntax

- Compatible with PEM, uses ASN.1 and BER (tag,length,value)
- Syntax for digital signatures and envelopes (recursive)
- 6 content types

  data, signed data, enveloped data, signed-enveloped data, digested data, encrypted data

```
EnvelopedData ::= SEQUENCE {
          version Version,
          recipientInfos RecipientInfos,
          encryptedContentInfo EncryptedContentInfo }
RecipientInfos ::= SET OF RecipientInfo
EncryptedContentInfo ::= SEQUENCE {
         contentType ContentType,
         contentEncryptionAlgorithm ContentEncryptionAlgorithmIdentifier,
         encryptedContent [0] IMPLICIT EncryptedContent OPTIONAL }
```


PCKS 8 – private key syntax

PrivateKeyInfo
- –version
- –privateKeyAlgorithm
- –privateKey
- –Attributes

- encryptedPrivateKeyInfo
  - –encryptionAlgorithm
  - –encryptedData
- privateKeyInfo BER-encoded and encrypted
- Usually encrypted using PKCS 5

PKCS 10 – certification request

- Certification Request
  - –certificationRequestInfo

    Version, subjectName, subjectPublicKeyInfo, attributes
  - –signatureAlgorithm
  - –signature
- Signed with private key corresponding to public key in request
- very RSA specific
- RFC 2511 defines a different format:

  certificate request message format

PKCS 12 – personal info exchange format

Example: IE/netscape personal key export

types of information
- – PKCS 8 shrouded key
- – Private key
- – Certificates
  - X.509v3
  - SDSI
- – CRLs
  - X.509
- – Secret
  - Whatever

recursive composition of these

Each of these can be
- •Plaintext
- •Enveloped

  encrypted using a secret key which is encrypted using a public key
- •Encrypted
  - -secret key encrypted
  - -usually password derived

    use PKCS 5 and a password formatting standard which is part of PKCS 12

The entire stuff is then either
- •Signed and accompanied with signing certificate
- •MAC'ed
  - -PKCS 5 based and accompanied with salt and iteration count

PEM

Privacy Enhanced Mail
- proposed in '87
- defined in RFC's 1421-1424
  - 1421 message encryption/authentication
  - 1422 certificate-based key mgt (X509)
  - 1423 formats and definitions
  - 1424 certificate services
- provides privacy, authentication, message integrity
- messages may be ordinary, clear signed, encoded signed, encrypted
- canonical form and ascii encodings
- implementations: RIPEM, TIS/PEM, MOSS/MIME
- Specifies a detailed certification hierarchy ⊗

S/MIME has replaced PEM for email attachments

PEM encodings used by OpenSSL for key and certificate encodings – base 64 encoding of DER.

OpenSSL and PEM

- OpenSSL uses PEM files (ASCII/base64 encoding) for certificates and CRL's, can concatenate info in the files
- cat mycert.pem

```
-----BEGIN CERTIFICATE-----
MIIDNjCCAp+gAwIBAgIBADANBgkqhkiG9w0BAQQFADB2MQswCQYDVQQGEwJVUzEL
MAkGA1UECBMCVE4xEjAQBgNVBAcTCUtub3h2aWxsZTELMAkGA1UEChMCVVQxCzAJ
BgNVBAsTAkNTMQ4wDAYDVQQDEwVCdWJiYTEcMBoGCSqGSIb3DQEJARYNYnViYmFA
dXRrLmVkdTAeFw0wNjA0MjEyMTU1MDhaFw0xNjA0MTgyMTU1MDhaMHYxCzAJBgNV
BAYTAlVTMQswCQYDVQQIEwJUTjESMBAGA1UEBxMJS25veHZpbGxlMQswCQYDVQQK
EwJVVDELMAkGA1UECxMCQ1MxDjAMBgNVBAMTBUJ1YmJhMRwwGgYJKoZIhvcNAQkB
Fg1idWJiYUB1dGsuZWR1MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDloSkv
iXwIJC4iVRzEAfRoUveUWTvE2IIBZcnbiMcFMJ4OccVtnmbaKcWhSmskcGV66yAk
R5vYkSt3Owl7UYK+dmPkNX2yMqIeAHSC7ek9FUpux/8m1DoebhuTNRxLCg9X1H9Z
VqfvYdRIQVG4TKwaYZNpKw7QMcxRjDKYXKm3LwIDAQABo4HTMIHQMB0GA1UdDgQW
BBSnoT8dR7zJAcF4rdFccobDgVohRDCBoAYDVR0jBIGYMIGVgBSnoT8dR7zJAcF4
rdFccobDgVohRKF6pHgwdjELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAlROMRIwEAYD
VQQHEwlLbm94dmlsbGUxCzAJBgNVBAoTAlVUMQswCQYDVQQLEwJDUzEOMAwGA1UE
AxMFQnViYmExHDAaBgkqhkiG9w0BCQEWDWJ1YmJhQHV0ay5lZHWCAQAwDAYDVR0T
BAUwAwEB/zANBgkqhkiG9w0BAQQFAAOBgQCe8SmaFw/VkCriBIwafUKcciAzIrjx
DanSQ8InamtykJxrUmabIT2TwjnvHfYZFB/liAK5FUFVtqa6G34F5hLx8SDMtE8/
6boqEHKfqq0FlBPVHP0UJs6qxOCfs/wAc9+hH1kkL0xhOpUZX+AP2oFdAH3xh2RY
I8vz6wnuRrAwWQ==
-----END CERTIFICATE-----
```


Creating certificates

- Your web browser can create you a public/private key pair AND send it to a CA for signing (costs money $$)
- Or use OpenSSL to create your own CA (or Java keytool)
  - –genrsa -- key generation
  - –verify -- verify cert
  - –pkcs12 -- convert cert encodings (PKCS12 ←→ PEM)
  - –x509 -- cert management including signing
  - –req -- generate cert or CSR

Next time

- PKI
- S/MIME
- Web security and SSL

	Lectures
1.	Risk, viruses
2.	UNIX vulnerabilities
3.	Authentication & hashing
4.	Random #s classical crypto
5.	Block ciphers DES, RC5
6.	AES, stream ciphers RC4, LFSR
7.	**MIDTERM** ⊗
8.	Public key crypto RSA, D-H
9.	ECC, PKCS, ssh/pgp
10.	PKI, SSL
11.	Network vulnerabilities
12.	Network defenses, IDS, firewalls
13.	IPsec, VPN, Kerberos, secure OS
14.	Secure coding, crypto APIs
15.	review