

Foundations of Network and Computer Security

John Black

Lecture #9
Sep 21st 2004

CSCI 6268/TLEN 5831, Fall 2004

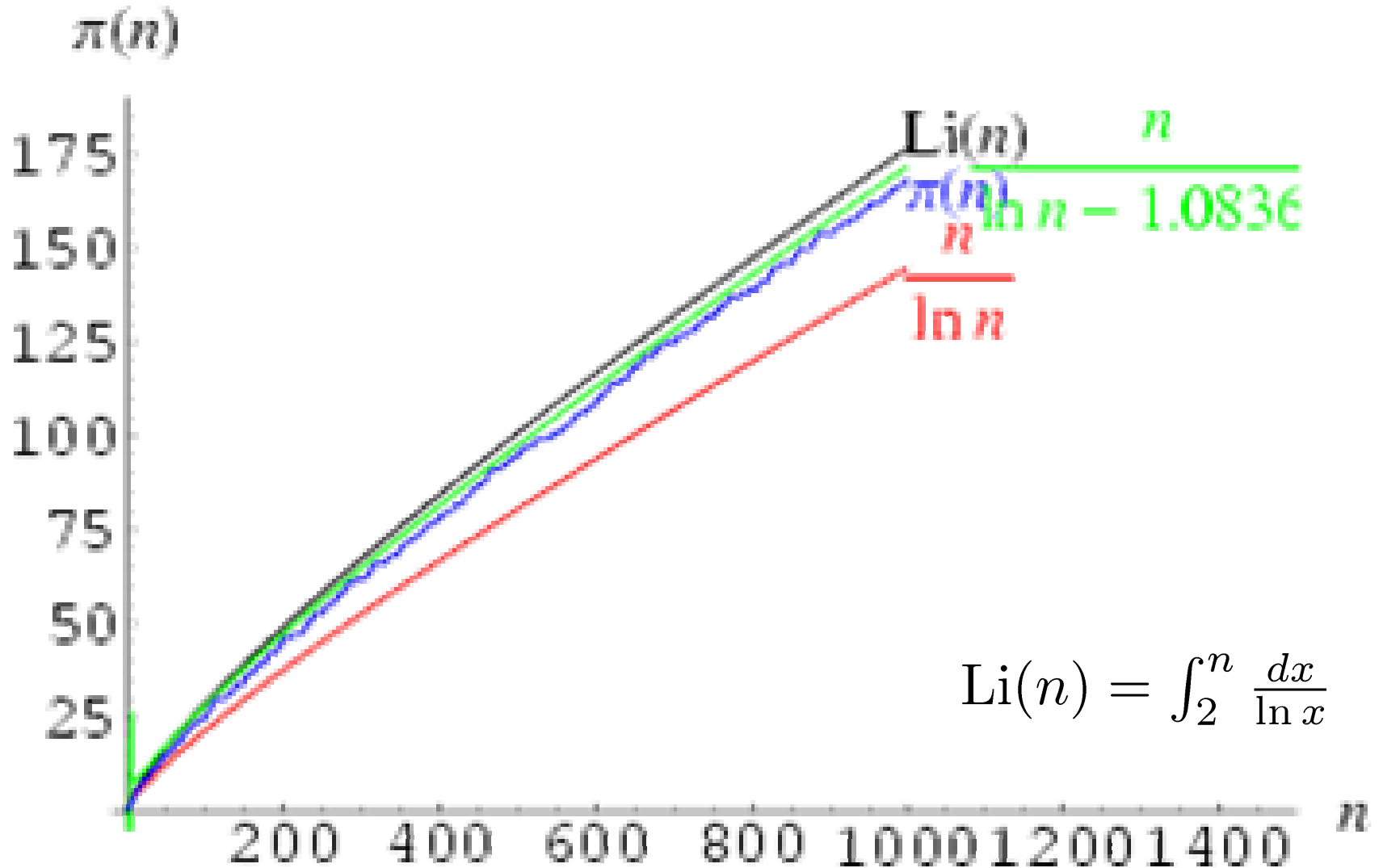
Announcements

- Quiz #2, Tuesday, Sept 28th
- Project #0 due Tuesday, Oct 5th
 - A few words about testing output
- Midterm, Thursday, Oct 14th
- Exams are closed notes, calculators allowed
- Remember to consult the class calendar

Prime Number Theorem

- Jeff asked last time about the distribution of primes
 - I gave a pretty non-rigorous answer; let me try again
 - PNT: $\pi(n) \sim n/\ln(n)$ where $\pi(n)$ is the number of primes smaller than n
 - In other words, $\lim_{n \rightarrow \infty} \pi(n) \ln(n)/n = 1$
 - What does this mean?
 - Primes get sparser as we go to the right on the number line

$\pi(n)$ versus $n/\ln(n)$



Sample Calculation

- Let's say we're generating an RSA modulus and we need two 512-bit primes
 - This will give us a 1024-bit modulus n
- Let's generate the first prime, p
 - Question: if I start at some random 512-bit odd candidate c , what is the probability that c is prime?
 - Ans: about $1/\ln(c) \approx 1/350$
 - Question: what is the expected number of candidates I have to test before I find a prime, assuming I try every odd starting from c ?
 - Ans: each number has a $1/350$ chance, but I'm testing only odd numbers, so my chance is $1/175$; I therefore expect to test 175 numbers on average before I find a prime
 - Of course I could do more sieving (eliminate multiples of 3, 5, etc)

Back to SSL/TLS

- SSL
 - Secure Socket Layer
 - Designed by Paul Kocher, consulting for Netscape
- TLS
 - Transport Layer Security
 - New version of SSL, and probably what we should call it (but I'm used to SSL)
- Used for web applications (https)
 - But also used many other places that aren't as well-known

TLS – Sketch

- Let's start by trying to design TLS ourselves and see what else we'll need
 - This will end up being only a sketch of the very complex protocol TLS actually is
- We want:
 - Privacy, authentication
 - Protection against passive and active adversaries
- We have:
 - Symmetric/asymmetric encryption and authentication
 - Collision-resistant hash functions

A First Stab

- First we need a model
 - Client/Server is the usual one
 - Client and Server trust each other
 - No shared keys between client and server
 - Assuming a shared key is not realistic in most settings
 - Adversary is active (but won't try DoS)
- Server generates RSA key pair for encryption
 - pk_S, sk_S
 - S subscript stands for "Server"

A First Stab (cont)

- Now client C comes along and wants to communicate with server S
 - C sends SSL HELLO to initiate session
 - S responds by sending pk_S
 - C sends credit card number encrypted with pk_S
 - S decrypts credit card number with sk_S and charges the purchase
- What's wrong here?

Our First Protocol: Problems

- There are tons of problems here
 - We don't know how to encrypt $\{0,1\}^*$, only how to encrypt elements of Z_n^*
 - Ok, say we solve that problem (there are ways)
 - It's really SLOW to use RSA on big messages
 - Ok, we mentioned this before... let's use symmetric cryptography to help us
 - There is no authentication going on here!
 - Adversary could alter pk_S on the way to the client
 - We'd better add some authentication too
- Let's try again...

Second Stab

- C says Hello
- S sends pk_S to C
- C generates two 128-bit session keys
 - K_{enc} , K_{mac} , used for encryption and MACing
- C encrypts (K_{enc}, K_{mac}) with pk_S and sends to S
- S recovers (K_{enc}, K_{mac}) using sk_S and both parties use these “session keys” to encrypt and MAC all further communication

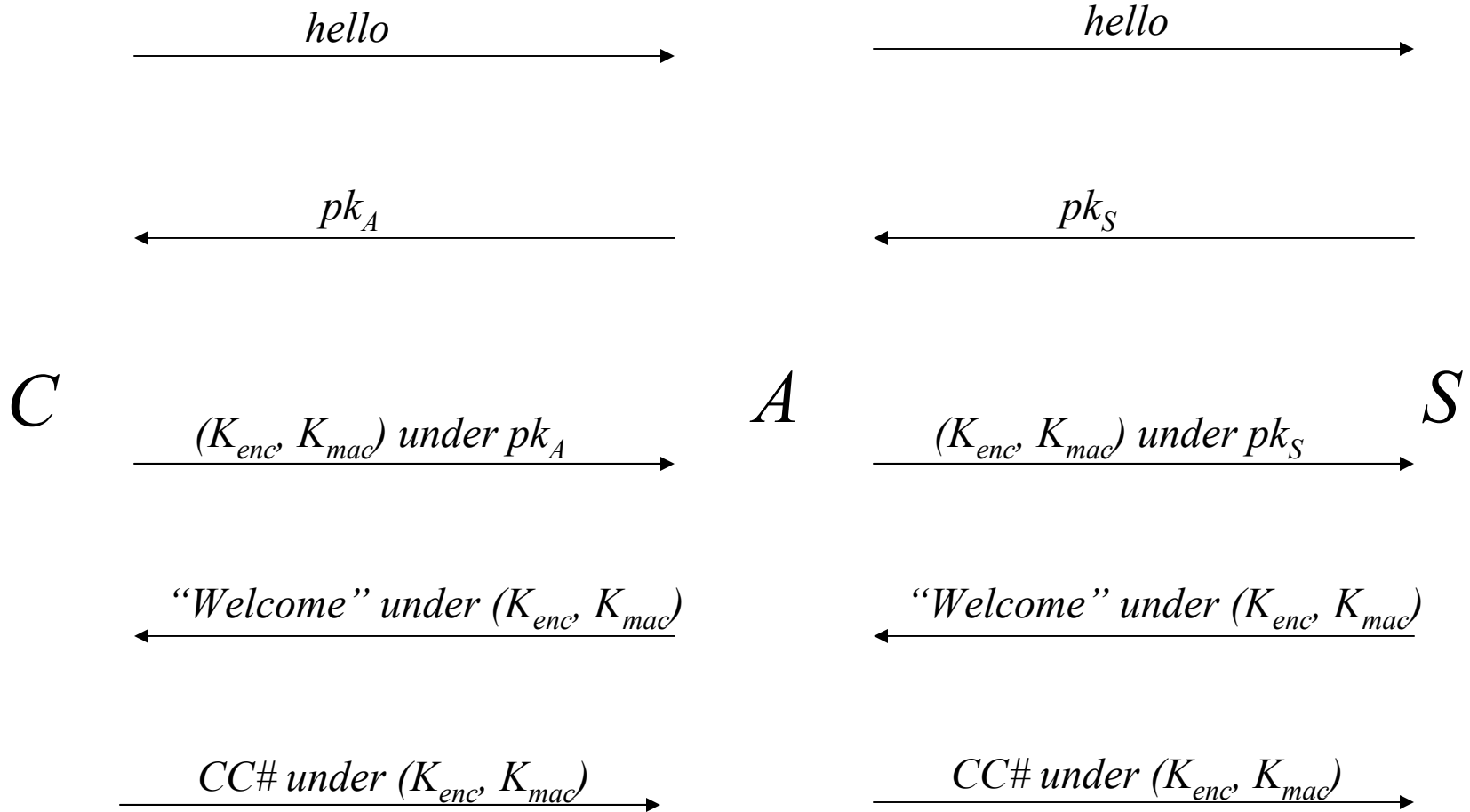
Second Stab (cont)

- Problems?
 - Good news: we're a lot more efficient now since most crypto is done with symmetric key
 - Good news: we're doing some authentication now
 - Bad news: Man-in-the-Middle attack still possible
 - Frustratingly close
 - If we could get pk_s to the client, we'd be happy

Man in the Middle

- Let's concretely state the problem
 - Suppose an adversary A generates pk_A and sk_A
 - Now S sends pk_S to C, but A intercepts and sends pk_A to C instead
 - C responds with (K_{enc}, K_{mac}) encrypted under pk_A and sends to S
 - A intercepts, decrypts (K_{enc}, K_{mac}) using sk_A and re-encrypts with pk_S then sends on to S
 - A doesn't have to use (K_{enc}, K_{mac}) here... any keys would do
 - Idea is that A proxies the connection between C and S and reads/alters any traffic he wishes

MitM Attack



How do we Stop This?

- Idea:
 - Embed pk_S in the browser
 - A cannot impersonate S if the public key of S is already held by C
 - Problems:
 - Scalability (10,000 public keys in your browser?)
 - Key freshening (if a key got compromised and it were already embedding in your browser, how would S update?)
 - New keys (how do you get new keys? A new browser?)
 - Your crypto is only as reliable as the state of your browser (what if someone gets you to install a bogus browser?)
- (Partial) Solution: Certificates

Certificates: Basic Idea

- Enter the “Certification Authority” (CA)
 - Some trusted entity who signs S 's public key
 - Well-known ones are Verisign, RSA
 - Let's assume the entity is called “CA”
 - CA generates keys vk_{CA} and sk_{CA}
 - CA signs pk_S using sk_{CA}
 - CA's vk_S is embedded in all browsers
 - Same problem with corrupted browsers as before, but the scaling problem is gone

New Protocol

- C sends Hello
- S sends pk_S and the signature of CA on pk_S
 - These two objects together are called a “certificate”
- C verifies signature using vk_{CA} which is built in to his browser
- C generates (K_{enc}, K_{mac}) , encrypts with pk_S and sends to S
- S decrypts (K_{enc}, K_{mac}) with sk_S
- Session proceeds with symmetric cryptography

SSH (A Different Model)

- SSH (Secure SHell)
 - Replacement for telnet
 - Allows secure remote logins
- Different model
 - Too many hosts and too many clients
 - How to distribute pk of host?
 - Can be done physically
 - Can pay a CA to sign your keys (not likely)
 - Can run your own CA
 - More reasonable, but still we have a bootstrapping problem

SSH: Typical Solution

- The most common “solution” is to accept initial exposure
 - When you connect to a host for the first time you get a warning:
 - “Warning: host key xxxxxx with fingerprint xx:xx:xx is not in the .ssh_hosts file; do you wish to continue? Saying yes may allow a man-in-the-middle attack.” (Or something like that)
 - You take a risk by saying “yes”
 - If the host key *changes* on your host and you didn’t expect that to happen, you will get a similar warning
 - And you should be suspicious

Key Fingerprints

- The key fingerprint we just saw was a hash of the public key
 - Can use this when you're on the road to verify that it's the key you expect
 - Write down the fingerprint on a small card and check it
 - When you log in from a foreign computer, verify the fingerprint
 - Always a risk to log in from foreign computers!

X.509 Certificates

- X.509 is a format for a certificate
 - It contains a public key (for us, at least), email address, and other information
 - In order to be valid, it must be signed by the CA
 - In this class, our grader Mazdak, will be the CA

Project #1

- The next phase of the project
 - Won't be assigned for a while, but here is a heads-up
 - You will generate an RSA pk,sk pair using OpenSSL (**genrsa** command)
 - Your private key should be password protected
 - PEM stands for “Privacy Enhanced Mail” and is the default format used by OpenSSL

```
% openssl genrsa -out john-priv.pem 1024
Generating RSA private key, 1024 bit long modulus
.....++++++
.+++++++
e is 65537 (0x10001)
```

What does secret key look like?

```
-----BEGIN RSA PRIVATE KEY-----  
fFbkGjYxpp9dEpiq5p6lQ/Dm/Vz5X2Kpp2+1lqFCKXLzxc8Z8zL7Xgi3oV5RUtSl  
wFjkiJaPP7fyo/X/Swz0L0lQKVQ7RDUe9NpnwTUBV44rtQVsSWfbgzda9MAQT945  
wBI270AJWYQTApEeM2JhgvqCSptdIn9paC9yeIzXLxwqrnlLCscGKncX53y3J3QG  
KP1UqujpdTY9FRMvbL6bM5cn1bQ16pSbjntgFi5q4sdcwBNiWveFy5BNf4FnWtk6  
KdAQ4jFeZqnwR3eAP0kdleosucPNZMxoQKafsi19bGi9BDdR4FoBdHy+K1sbXEm0  
Z5+mcVPIITmB9MgUQLZ/AFguXHsxGDih74es2Ahe60ACxWlqe4nfFxikXJfJw8EY  
9nzw8xSZV5ov66BuT6e/K5cyrd2r0mlUb9gooYoVZ9UoCfO/C6mJcs7i7MWRNakv  
tC1Ukt9FqVF14Bcr1oB4QEEk1oWW3QU2TArCWQKc67sVcSBuvMJjBd18Q+8AZ7GY  
Jtt4rcOEb0/EUJuMauv4XlAQkiJcQ46qQjtkUo346+XMeRjWuUyQ/e5A/3Fhprat  
7C10relDQonVi5WoXrEUTKeoaJgggZaeFhdpoe6DQePSWfLKB06u7qpJ6Gr5XAd  
NnBoHEWBYH4C0YcGm77OmX7CbPaZiIrha/WU7mHUBXPUHDCOhyYQK8uisADKfmEV  
XEzyl3iK6hF3cJFDZJ5BBmI774AoBsB/vahLquBUjSPtDruic24h6n2ZXcGCLiyc  
redr80iGRJ0r6XF85GYKU082vQ6TbSXqBgM5Llotf53gDZjMdT71eMxI4Fj3PH91  
-----END RSA PRIVATE KEY-----
```

(Not very useful, is it?)

OpenSSL RSA Private Key

```
% openssl rsa -in john-priv.pem -text -noout
```

```
Private-Key: (1024 bit)
```

```
modulus:
```

```
00:a3:8d:60:56:df:75:52:50:62:fb:6b:09:3a:2e:  
e4:46:4e:e3:e2:d2:fe:c5:43:52:71:5a:47:ed:26:. . .  
63:29:27:38:bf:df:cc:cd:0b
```

```
publicExponent: 65537 (0x10001)
```

```
privateExponent:
```

```
7f:09:7c:50:5e:27:c9:f5:28:bd:33:29:aa:a8:eb:  
a4:f4:f8:2b:a2:4a:44:3d:03:97:8a:51:9e:12:29:. . .  
19:7f:28:b4:ff:70:f8:99
```

```
prime1:
```

```
00:d9:12:85:e4:c5:6f:23:7a:19:7c:34:81:1a:20:  
ac:80:ae:9a:0d:24:a8:ca:9d:43:06:7a:26:a1:02:. . .  
0c:8f:a5:8d:9f
```

```
prime2: ...
```

```
exponent1: ...
```

```
exponent2: ...
```

```
coefficient: ...
```

Challenge Problem #2: Figure out what these are!

But Notice no Password!

- Shouldn't leave your private key lying around without password protection; let's fix this

```
% openssl genrsa -aes128 -out john-priv.pem 1024
Generating RSA private key, 1024 bit long modulus
.....++++++
.....++++++
e is 65537 (0x10001)
Enter pass phrase for john-priv.pem:
Verifying - Enter pass phrase for john-priv.pem:

% openssl rsa -in john-priv.pem -text -noout
Enter pass phrase for john-priv.pem:
Private-Key: (1024 bit)
modulus:
    00:ca:40:b9:ef:31:c2:84:73:ab:ef:e2:6d:07:17... ..
```

What does key look like now?

This private key file is encrypted

```
-----BEGIN RSA PRIVATE KEY-----  
Proc-Type: 4, ENCRYPTED  
DEK-Info: AES-128-CBC,1210A20F8F950B78E710B75AC837599B
```

```
fFbkGjYxpp9dEpiq5p61Q/Dm/Vz5X2Kpp2+1lqFCKXLzxc8Z8zL7Xgi3oV5RUtS1  
wFjkiJaPP7fyo/X/Swz0LO1QKVQ7RDUe9NpnwTUBV44rtQVsSWfbgzdA9MAQT945  
wBI27OAJWYQTApEeM2JhgvqCSptdIn9paC9yeIzXLxwqrnlLCscGKncX53y3J3QG  
KP1UqujpdTY9FRMvbL6bM5cn1bQ16pSbjntgFi5q4sdcwBNiWveFy5BNf4FnWtk6  
KdAQ4jFeZqnwR3eAP0kdleosucPNZMxoQKafsi19bGi9BDdR4FoBdHy+K1sbXEm0  
Z5+mcVPIITmB9MgUQLZ/AFguXHsxGDih74es2Ahe6OACxWlqe4nfFxikXJfJw8EY  
9nzw8xSZV5ov66BuT6e/K5cyrd2r0mlUb9gooYoVZ9UoCfO/C6mJcs7i7MWRNakv  
tC1Ukt9FqVF14Bcr1oB4QEeK1oWW3QU2TArCWQKc67sVcSBuvMJjBd18Q+8AZ7GY  
Jtt4rcOEb0/EUJuMauv4XlAQkiJcQ46qQjtkUo346+XMerjWuUyQ/e5A/3Fhprat  
7C10relDQonVi5WoXrEUTKeoaJgggZaeFhdpoe6DQePSWfLKB06u7qpJ6Gr5XAd  
NnBoHEWBYH4C0YcGm77OmX7CbPaZiIrha/WU7mHUBXPUHDCOhyYQK8uisADKfmEV  
XEzyl3iK6hf3cJFDZJ5BBmI774AoBsB/vahLquBUjSPtDruic24h6n2ZXcGCLiyc  
redr8OiGRJ0r6XF85GYKUO82vQ6TbSXqBgM5Llotf53gDZjMdT71eMxI4Fj3PH91  
-----END RSA PRIVATE KEY-----
```

CSR: Certificate Request

- You will generate a CSR
 - Certificate Request
 - Has your name, email, other info, your public key, and you sign it
- Send your CSR to the CA
 - CA will sign it if it is properly formatted
 - His signature overwrites your signature on the CSR
- Once CA signs your CSR it becomes a certificate

Creating a CSR

```
% openssl req -key john-priv.pem -new -out john-req.pem
Enter pass phrase for john-priv.pem:
You are about to be asked to enter information that will
  be incorporated into your certificate request.
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Colorado
Locality Name (eg, city) []:Boulder
Organization Name (eg, company) [Internet Widgits Pty
  Ltd]:University of Colorado
Organizational Unit Name (eg, section) []:Computer Science
Common Name (eg, YOUR name) []:John Black
Email Address []:jrblack@cs.colorado.edu
```

(Leave the rest blank)

This outputs the file **john-req.pem** which is a cert request

Viewing a CSR

```
% openssl req -in john-req.pem -text -noout
```

```
Certificate Request:
```

Note: not password protected



```
Data:
```

```
Version: 0 (0x0)
```

```
Subject: C=US, ST=Colorado, L=Boulder, O=University of Colorado,  
OU=Computer Science, CN=John Black/emailAddress=jrblack@cs.colorado.edu
```

```
Subject Public Key Info:
```

```
Public Key Algorithm: rsaEncryption
```

```
RSA Public Key: (1024 bit)
```

```
Modulus (1024 bit):
```

```
00:ca:40:b9:ef:31:c2:84:73:ab:ef:e2:6d:07:17:
```

```
83:5e:96:46:24:25:38:ed:7a:60:54:58:e6:f4:7b:
```

```
...
```

```
27:de:00:09:40:0c:5e:80:17
```

```
Exponent: 65537 (0x10001)
```

```
Attributes:
```

```
a0:00
```

```
Signature Algorithm: md5WithRSAEncryption
```

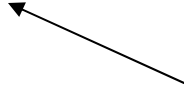
```
32:e1:3f:e2:12:47:74:88:a3:f9:f4:44:8a:f3:b7:4e:d1:14:
```

```
1f:0b:be:b8:19:be:45:40:ed:5b:fb:ab:9b:01:e8:9a:26:0c:
```

```
...
```

```
9c:e0
```

CSR is signed by you



CSRs

- Why is your CSR signed by you?
 - Ensures that the CSR author (you) have the private key corresponding to the public key in the CSR
 - If we didn't do this, I could get the CA to sign anyone's public key as my own
 - Not that big a deal since I can't decrypt things without the corresponding private key, but still we disallow this
- Why does the CA sign your public key
 - Well, because that's his reason for existence, as discussed previously
 - Ok, let's say I email my CSR to Mazdak and he signs it... then what?

Sample Certificate

-----BEGIN CERTIFICATE-----

MIIDkDCCAnigAwIBAgIBCzANBgkqhkiG9w0BAQQFADCBgTEQMA4GA1UEAxMHSm9o
biBDQTERMA8GA1UECBMIQ29sb3JhZG8xCzAJBgNVBAYTAIVTMSYwJAYJKoZIhvcN
AQkBFhdqcmJsYWNrQGNzLmNvbG9yYWRvLmVkdTEIMCMA4GA1UEChMcUm9vdCBDZXJ0
aWZpY2F0aW9uIEF1dGhvcml0eTAeFw0wMzExMTMyMDQ1MjFaFw0wNDExMTIyMDQ1
MjFaMIGFMRIwEAYDVQQDEwIUZXN0IFVzZlIxETAPBgNVBAgTCENvbG9yYWRvMQsw
CQYDVQQGEwJVUzEjMCEGCSqGSIb3DQEJARYUdGVzdEBjcy5jb2xvcmFkby5lZHUx
FjAUBgNVBAoTDVVuaXYgQ29sb3JhZG8xEjAQBgNVBAAsTCUNTQ0kgNDgzMDCCASlw
DQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL1k6hJ9gwXIUYHiFOm6OHOf+8Y0
o1b7WOexYfNDWm9H0I79o0wVgDj7waOgt4hz2FE2h+gArfGY5VsaSzmCH0EA4kDS
m/sPob3HTVpbIFwlbXTV7hC0OxOzRs8lphDdj1vaNDSnOwqOS1ADCfIdaGEh9WKi
rEdFdriiu7v1bw+c1ByM57v9aHO7RslswR9EnRFZPWYa8GpK+St0s8bZVf98lOOk
H8HiliyVSt5IAXRMnlxhYMG89tkkuCAwxgDD+7WqyETYxY0UCg/joFV4IKcC7W1b
CmvxsY6/H35UpGgv0anCkjp0mKY/YWB9KXwrR8NHC7/hacij0YNiV77EIMCAwEA
AaMNMAswCQYDVR0TBAlwADANBgkqhkiG9w0BAQQFAAOCAQEAZr4hdQPcGnAYmk++
0bQ4UKILXj9wr7UZdgz3DKJNpMPkFjzU6wvJrd1C8KIKfJC63TKHJ7svmdZwTCB2
hNUFy8kbe2KvNWQiGoX3PaY1eo3auLzli8lXpQn+W/p1z3MhtpQqNllqzG8G1o50
QP2yAyj2V0rnwIRL3kZ7ibvXRnSB1Bz+6zJLLAQR4kTQD2EfxLhpks+iSE+m58PV
tfck25o2IMJYYLAdtoNGjcFG9/aDk+GHbsx8LP/va6B6BlzB3vrefuQvBu+7j/mz
aXP7QkuGYf1r4yyOiuMYnw0kwp5xndDKTzORsxksHQk5AWfBXrDdGPZrb6i1UIOq
U/P3+A==

-----END CERTIFICATE-----

Ooh...how useful!

Viewing a Certificate

```
% openssl x509 -in john-cert.pem -text -noout
```

```
Certificate:
```

```
Data:
```

```
Version: 3 (0x2)
```

```
Serial Number: 1 (0x1)
```

```
Signature Algorithm: md5WithRSAEncryption
```

```
Issuer: C=US, ST=CO, L=DENVER, O=UCB, OU=CS,  
CN=MAZDAK/emailAddress=mazdak.hashemi@colorado.edu
```

```
Validity
```

```
Not Before: Sep 17 20:57:44 2004 GMT
```

```
Not After : Sep 12 20:57:44 2005 GMT
```

```
Subject: C=US, ST=Colorado, L=Boulder, O=University of Colorado, OU=Computer  
Science, CN=John Black/emailAddress=jrblack@cs.colorado.edu
```

```
Subject Public Key Info:
```

```
Public Key Algorithm: rsaEncryption
```

```
RSA Public Key: (1024 bit)
```

```
Modulus (1024 bit):
```

```
00:ca:40:b9:ef:31:c2:84:73:ab:ef:e2:6d:07:17:
```

```
83:5e:96:46:24:25:38:ed:7a:60:54:58:e6:f4:7b:. . .
```

```
27:de:00:09:40:0c:5e:80:17
```

```
Exponent: 65537 (0x10001)
```

```
Signature Algorithm: md5WithRSAEncryption
```

```
97:4a:20:ea:a7:5a:4d:4c:77:b9:3e:c0:49:9b:ab:8f:6f:02:
```

```
53:24:a9:71:97:2c:1f:e8:e4:eb:d0:f6:6a:7c:74:30:1d:9e: . . .
```

```
3a:59
```

← *Again, no encryption*

↖ *Now it's the CA's signature*

What have we Accomplished?

- We have an X.509 cert
 - It contains our public key, name, email, and other stuff
 - It is signed by the CA
- You have a private key in a password-protected file
 - Don't lose this file or forget the password!
- What else do we need?
 - We need to be able to verify the CA's signature on a public key!
 - We therefore need the CA's verification key

CA's Verification Key is a Cert!

- The CA generates a self-signed “root certificate”
 - This is his verification key (aka public key) which he signs
 - This certificate is what is embedded in your browser
 - This certificate is used to validate public keys sent from other sources
 - Mazdak's root certificate will be used to validate all public keys for our class

Mazdak's Root Cert

-----BEGIN CERTIFICATE-----

MIIDYjCCAsugAwIBAgIBADANBgkqhkiG9w0BAQQFADCBgzELMAkGA1UEBhMCVVMx
CzAJBgNVBAGTAkNPMQ8wDQYDVQQHEwZERU5WRVlxDDAKBgNVBAoTA1VDQjEELMAkG
A1UECxMCQ1MxDzANBgNVBAMTBk1BWkRBSzEqMCgGCSqGSIb3DQEJARYbbWF6ZGFr
Lmhhc2hlbWIAY29sb3JhZG8uZW11MB4XDTA0MDkxNzlyNTQwOVVoXDTA3MDkxNzly
NTQwOVVowgYMxCzAJBgNVBAYTAIVTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGREVO
VkVSMQwwCgYDVQQKEwNVQ0lxCzAJBgNVBAsTAKNTMQ8wDQYDVQQDEwZNQVpEQUx
KjAoBgkqhkiG9w0BCQEWG21hemRhay5oYXNoZW1pQGNvbG9yYWRvLmVkdTCBnzAN
BgkqhkiG9w0BAQEFAAOBjQAwGyKCGYEA1A8ClwTUxKI/ehlgMeTpU1gUmVIF/vXh
lYbBwz0CvXisMGq5U6JnGyianLmd+IJAE6NoSaEP3A4FZmDR0Aw5abM695PT4zyS
7J01jE8AfRIRe83yKQ/EwQDsn/pYZvD5DXsqL2GQj58GggAdX0qNy2fK0yum8zj5
t7KQ14tjmQMCAwEAAaOB4zCB4DAdBgNVHQ4EFgQU/Rp1mIPXUOwwteoAuXx4JrVf
vuYwgbAGA1UdIwSBqDCBpYAU/Rp1mIPXUOwwteoAuXx4JrVfvuahgYmkgYYwgYMx
CzAJBgNVBAYTAIVTMQswCQYDVQQIEwJDTzEPMA0GA1UEBxMGREVOVkVSMQwwCgYD
VQQKEwNVQ0lxCzAJBgNVBAsTAKNTMQ8wDQYDVQQDEwZNQVpEQUxKjAoBgkqhkiG
9w0BCQEWG21hemRhay5oYXNoZW1pQGNvbG9yYWRvLmVkdYIBADAMBgNVHRMEBTAD
AQH/MA0GCSqGSIb3DQEBBAUAA4GBALTQurLtBbGJB1aarA+xmfgm7JPOK7exljAi
SuWuVpaG+C3lQWfrZwVdRYSQ4zIRUQzoi5AnEv5TYoI18mM8xJA5FVCyTZZEMmv9
z1torlhq17Xuydg+YGNobUaw5eVdzsxpJCS0oiwhfRhQRZ59RY10TpwSux1Xd/O
asesXE4O

-----END CERTIFICATE-----

How to Distribute the Root Cert?

- It's ridiculous for me to ask you to write this down, right?
 - If I email it to you, it might get altered by an adversary
 - If I put it on the web page, it might get altered by an adversary
 - Ok, this is probably not a REAL concern for us, but we're practicing being paranoid
 - What can we do?

Distributing the Root Cert

- Fingerprint the root certificate!
 - We'll just distribute the fingerprint as a verification check
 - The cert itself will be distributed via some insecure means
 - The fingerprint will use a collision-resistant hash function, so it cannot be altered
 - But now we have to distribute the fingerprint
 - This you can write down, or I can hand you a hardcopy on a business card, etc
 - People used to have a fingerprint of their PGP public key on their business cards at conferences... haven't seen this in a while though

Root Cert Fingerprint

```
% openssl x509 -in cacert.pem -fingerprint -noout
```

MD5 Fingerprint =

**FE:EF:31:32:22:1D:93:29:
6C:14:2E:79:73:63:9A:02**

- Please write this down now
- And, yes, some is going to point out that perhaps my powerpoint was infiltrated during the night, so I'll check against my hardcopy

Overall Idea of the Project

- Each student has a cert containing a public key corresponding to his private key
- Each student knows the verification key of the CA
- Student A wants to send secure mail message M to student B
 - A obtains B's cert and verifies it is correctly signed by the CA
 - A chooses a random session key K and RSA encrypts using B's public key (from B's cert)
 - A writes out the encrypted K followed by M encrypted symmetrically, then signs each of these with her private key and sends to B
- B receives all of this and...
 - Obtains A's cert and verifies it is signed by CA
 - B verifies A's signature on the message
 - B uses his private key to decrypt K (session key used by A)
 - B uses K to decrypt M

Sample Message from A to B

-----BEGIN CSCI 6268 MESSAGE-----

hjh2vkeSGpWehAwgMOEbKomsW3lTd8BBBRefFchbAZpnbc+07wcI8OT0g9WP9iPV
K92xbzAiVlAN7ZFOWlx/iX2XQIbUQBU6kl7NOyPTtSZ/5+9JHVDY1TFZG3cGtVj5
SeJ97+kvuWkZvNcKjAeclYbRYpXRGwRmqPtz+o5WYWqWmqPV6lQWjbn4Jc+w2Gcl
FKR7t0Zsi5RcnEwIn+cZtuTe3QWW4/inMGMBFgbXjA2E6VU7zn62BdBHh7S1/oBR
tt84Rr4/oXXJhrEASdZJEdGw8trh0FPd48ioHElT7TNGMx4YJKHBV1+EMjTcHwdN
DCr29AZ2QyDh/pHYqvJmVg==

RSA Encrypted Session Key K

AES-128-CBC encrypted message M

U2FsdGVkX1/QUjgfw4jEV34P/Efn8Ub7NDzV5QL+uWoeDb1spQiz2BiPqQEalacb
CD2+XgD36FmmcP9WxD0dQ63AlX2K4t4SdSyTT8uk9YpdUC0thqCXfKdGM6P0u7Xx
gBxP0s0mtcNFKbcpwmiEp5K8ayGHsYW5lM2veFclVL75xReQGA8fkjZ3OQQeR+nz
nQTg2Hniyaniwbb1lYgBmyWQ4bsVK5UDG0iYab100cvPULFZXrMmK4aumMNTc+0Z
+Syj4FaPzUphhebhuhsU29tahd8hL9DZQ5ZuzZiZi5hy0nG5z45FHktap/bwwOGC

Iu3mRM6ZqoTVVanTqf0cBaRA5c+XJbhuxLxjs44viFKSKENmZ7pEPZtdisvd/aq2
weZblamCy2jnP0xQioI8Lc/zkno5XRW21bGH3kWeG8kMuOrBKVyms2FOEpsI0TH0
UIzck095R4jnPUI+e7S85z1Wx1ToyMI3Ub/Mee3MyIt60H2r2LC4sp9C01Yn4tYN
pA4ULy3DhFy4z9x4bX+aU+bSymiqf5JvSjMXS/zQYERW+1fhOKnU3fI518mE9Gbx
tJBJJmjnPxWhWpSJjvG7qEady/Pibcd8YPXn3NZ7j1mU8SgYog9vwJwz3fsKaCS6
AP4LTLN9ef5Hb/STtvA+ow==

-----END CSCI 6268 MESSAGE-----

RSA signature on first two chunks