

# Unix System Administration

**Chris Schenk**

**Lecture 17 – Thursday Mar 15**

**CSCI 4113, Spring 2007**

# SSH – ssh-keygen

- Create an SSH keypair
- % `ssh-keygen -t <type> [-f filename] [-b bits]`
  - `ssh-keygen -t rsa`
  - If options are omitted you may be prompted
  - Default keysize is 1024 bits (2<sup>1024</sup>)
- Optionally you can enter a passphrase
  - Stronger protection of your logins, highly recommended
- View a key's fingerprint (very useful)
  - `ssh-keygen -l ~/.ssh/id_rsa.pub`

# SSH – ssh-agent

- Keeps track of keypairs in memory
  - You open your ssh keys once at the beginning with passphrase
  - All passphrases tracked thereafter for the duration of ssh-agent
  - Useful if you ssh to one machine more than once
    - And want to keep access very secure!
- ssh-agent runs in the background
  - Any new shells you spawn are tracked by ssh-agent
    - May not be true for older versions

# SSL – Secure Sockets Layer

- Originally developed by Netscape
- Uses public-key cryptography to transmit data
  - Originally RSA was not included in SSL because of patent laws
  - Non-commercial use or full blown licenses for RSA
  - RSA patent expired in 2000, now included in OpenSSL
- Trust established with SSL Certificates
  - Trust is managed by a Certificate Authority (CA)
  - Solves a huge scalability problem with public keypairs

# Public Keys – The Scaling Problem

- We use public keys to talk securely with servers
- 'First try risk' exists with accepting a public key
  - Seen in SSH, but would also be true with browsers
    - Without VALID certificates, that is
- To avoid the 'first try risk' problem, we can include public keys in all web browsers
  - This is impossible with the number of servers on the web
  - We need a way to allow our public key to be accepted
    - And still maintain trust with the client

# SSL Certificates

- Enter certificates to solve the scaling problem
  - Embed a small(er) set of public keys in browsers
    - Instead of every web server's public key embedded in the browser (which would be hundreds of thousands)
  - An authority deems set of public keys to be trusted
- Each site manages their own certificate that is signed by the CA
  - The client checks if the CA has signed the key
  - If signature is valid, accept the public key for use

# OpenSSL – Command Line Tools

- Curiously enough, it's called `openssl`
  - Tons of different commands available in the library
- Used to do a number of things
  - Encrypt and decrypt files with a specific algorithm
    - e.g. Use RSA or DES3 to manage private data
  - Calculate cryptographic checksums
  - Create and manipulate X.509 certificates
    - This is the primary use of OpenSSL for the web

# OpenSSL – RSA Key Generation

- Use 'genrsa' option in openssl to create keys
  - schenkc@schenktop:~\$ **openssl genrsa -aes128 -out chris-priv.pem 1024**  
Generating RSA private key, 1024 bit long modulus  
.....++++++  
.....++++++  
e is 65537 (0x10001)  
Enter pass phrase for chris-priv.pem:  
Verifying - Enter pass phrase for chris-priv.pem:  
schenkc@schenktop:~\$
- You must have a block cipher to privatize key!
  - Otherwise the key is left completely unprotected
  - We use aes128 in the above example

# OpenSSL – Key File

- What does our secret key look like?

• 3 B G3N R RI T K Y  
 NCRY T D  
D K Inf : 128 CBC,974F4776 18993 47368C14F3D B4D38  
  
6m8D kM B 9IDg mQT v MFT CKUF LmCTI5L55B8b=F7=Jl4 b/2K  
O MMl kL On8.5 =F1 vm3/+f kmE C9j bu /VT9j DKh/u=NMCv+ L  
Oiu=g Hj5Q JnK u v J Q0j HM T mFF5D G/F+HOMD GgM+I C B  
+O72K Ulv f Gh=J 6D 6 mvG3bCi3f6 5IT2 oU Ou v CROg9 Tliuf  
3D=/DR7Dk =UIbnv 8 DFv GI/LUU mY 7 3FIH46GRk6 mN1u B  
UYfT m i21 //5J7 f 2+f Cf00gGUF nB hkium u g Bgg 3F7b f  
u6B7 h m+g OT Y YT5 hQ=MT0LF mKKiO GFui 2nYG IC+Rjf DYBLf  
B GuM DDQ TY C 1 M 7C 105 2ufRvM I 7 +3B KQ 7 H k2+ G6u /BGG  
Ilg U 4 Nmh7 N9j OIRk/7 H7 7 gO9n OYBD0 9IU Q HL5 j /7 5  
h f /G Dn U lhdUL l 1 C1F0NH4 / Omjm5l4 b3kgJ+ L 7 R 3  
nvyh+ugFj 3IL+ DuM5 299 k /vCm G6k7 i5l2gg+=1 7 91Ygm b2  
E4 DT 8uYY +b F D 3 M h=0 1u8 0ni4mDC 4lQ9 H O0CD 1 5 Kfl  
-2lGI L hn 3hKI2R BII89JC u u GGDfkD+G7i nNT74L UYM On Y 424Q  
ND R RI T K Y

- Not very readable, is it?

# OpenSSL – Key File (cont)

- To read our key, we need to run openssl again

```
hknk@hknk:~$ openssl rsa -in chris-priv.pem -text -noout
Private-Key: (1024 bit)
modulus:
  00:f0:5b:90:93:97:56:b0:38:b1:b8:76:32:
  3f:13:38:87:13:5f:25:12:0b:58:22:
  3ublikey: 65537 (010001)
  iv:
  0:f:6:8:8:34:f:2:80:b:37:b:31:63:6f:38:
  3f9:97:4:05:28:f:1b:57:4f:34:70:20:f:7:
  3m1: ...
  m2: ...
  37n:n: ...
  7n:n: ...
  7fffi:n: ...
```

- All fields here are used in RSA encryption/decryption

# OpenSSL – CSR: Certificate Request

- Certificate not valid until signed by an authority
  - Usually a company (Verisign) will do this for cash
    - Large number of CAs around to take your money
- To obtain a certificate, first create a 'request'
  - Has your name, email, other info, your public key and signature
  - CA will sign this request if properly formatted and your information matches who you are
  - The CA supposedly checks to see if you are a valid business
    - And checks periodically as well

# OpenSSL – Creating a CSR

- You use your private key to create the request

```
chris@chris:~$ openssl req -key chris-priv.pem -new -out chris-req.pem
```

...

```
C=un N m (2 l U :US  
vin N m (full n m) :Colorado  
L=li N m (g, i) :Boulder  
O=gn n N m (g, m n) In i gi L :University of Colorado  
O=gn n I Uni N m (g, n) :Department of Computer Science  
C=mm n N m (g, YOUR n m) :schenktop.cs.colorado.edu  
m il :Christopher.Schenk@Colorado.EDU
```

```
chris@chris:~$
```

# OpenSSL – Viewing a CSR

- Very similar to viewing your key

```
chank@chank:~$ openssl req -in chris-req.pem -text -noout
Certificate Request
Data:
  version: 0 (0)
  subject: C=UK, T=City, L=Building, O=University of City, OU=Department
  CN=chank, email=chank@citybuilding.com, c=UK, ou=Department
  subjectKeyInfo:
    subjectKeyIdentifier: 3
    publicKey:
      RSA Public Key: (1024 bit)
      Modulus: (1024 bit)
        00:f0:5b:90:93:97:56:b0:38:b1:87:63:2:
        f:13:87:1:3:5f:2:51:20:b5:82:2: ...
  signatureAlgorithm: sha1WithRSAEncryption
  signature: 3
  subjectKeyIdentifier: 65537 (010001)
  subjectKeyIdentifier:
    subjectKeyIdentifier: 3
    publicKey:
      RSA Public Key: (1024 bit)
      Modulus: (1024 bit)
        00:f0:5b:90:93:97:56:b0:38:b1:87:63:2:
        f:13:87:1:3:5f:2:51:20:b5:82:2: ...
```

# OpenSSL - CSRs

- A CSR is self-signed (signed by you). Why?
  - Ensures that the CSR author (you) has the private key corresponding to the public key in the CSR
    - If this wasn't enforced, I could grab anyone else's public key to use
- Why does the CA sign your public key?
  - Well, this is the whole point
  - The authority will only sign your key if you're trustworthy
    - For credit card handling, personal data, etc
- What happens if a CA does not sign your key?

# OpenSSL – Signed Certificates

- What do we have so far?
  - An X.509 certificate signed by the CA
    - Contains our public key, name, email, and other info
  - A private key in a password-protected file
    - Don't lose the password or your certificate may become useless
- What else do we need?
  - We need to be able to verify the CA's signature on the cert
    - Which means we need the CA's verification key

# OpenSSL – CA Verification Key

- CA's verification key IS a certificate
  - CA generates a self-signed 'root' certificate
  - Certificate has the verification key (aka public key)
  - This certificate is embedded in your web browser!
  - Used to validate public keys from other sources
- Root certificate is freely distributed to everyone else
  - You can add in other CA certificates to your browser if you want
    - “Unable to verify certificate...do you want to accept?”

# Digital Signatures

- We keep talking about 'signing' certificates
  - How do we actually accomplish this?
- Signatures require a public/private keypair
  - We can encrypt a piece of data with our private key
  - No one else has our private key, difficult to forge ciphertext
- But private/public key algorithms are block ciphers
  - Means fixed input gives the same fixed size output
- We don't want our signature to be as long as the message

# Cryptographic Hashes

- $h = h^{-1} = h(m)$
- A 'hash' by itself takes a variable length input
  - Always produces a fixed size output
  - MD5, SHA1, CRC, and others
- Cryptographic hashes are functions that add certain security properties suitable for some information security
  - Verifying file integrity
  - Signing email to demonstrate originality
  - Password hashes

# Cryptographic Hash Properties

- Cryptographic hashes have three general properties
- Preimage resistance
  - Given a hash value  $h$ , it is difficult to find the original message
- Second preimage resistance
  - Given a hash value  $h$  and its message  $m_1$ , it is difficult to find another input  $m_2$  such that  $h = h(m_1) = h(m_2)$
- Collision resistance
  - It should be difficult to find two messages  $m_1$  and  $m_2$  such that  $h(m_1) = h(m_2)$

# Cryptographic Hashes and Signatures

- Digital Signatures follow a Hash-Then-Sign paradigm
- Remember, a block cipher produces output the same size as its input
  - We want to sign our data, but we want to keep the sig. small
  - In fact, it'd be better to keep it a fixed size!
- So let's first hash our data **BEFORE** signing
  - Takes the input to the block cipher to a fixed size
    - Size depends on the hash used (MD5, SHA1, etc)
- Why don't we hash **AFTER** using block cipher?

# MD5 – A Broken Cryptographic Hash?

- **Let's take a look at the wikipedia post:**
  - Because MD5 makes only one pass over the data, if two prefixes with the same hash can be constructed, a common suffix can be added to both to make the collision more reasonable. And because the current collision-finding techniques allow the preceding hash state to be specified arbitrarily, a collision can be found for any desired prefix -- for any given string of characters X, two colliding files can be determined which both begin with X. All that is required to generate two colliding files is a template file, with a 128-byte block of data aligned on a 64-byte boundary, that can be changed freely by the collision-finding algorithm.
- **Byte boundaries and template files are mentioned**

# MD5 – A Broken Cryptographic Hash?

- Let's look at this a little further
  - <http://www.cits.rub.de/MD5Collisions/>
  - Site has an example of a hash collision for two documents
- A 'prefix' was mentioned on the previous slide
  - This prefix can be modified while the rest of the document is left the same
- What type of attack is this?
  - Think in terms of the cryptographic hash properties from earlier
- Does this make MD5 bad to use for hashes?