

# Unix System Administration

**Chris Schenk**

**Lecture 14 – Thursday Feb 28**

**CSCI 4113, Spring 2008**

# BIND Notes

- `/etc/bind/named.conf`
  - This is where you define what zones you are going to host on your DNS server
  - This is where you reference zone files directly
- Zone file configurations
  - Done in separate files! Not in `/etc/bind/named.conf`!
- Each view requires a root zone
  - And you may as well copy localhost and the others
  - Duplicate information in both views!

# SSH – Secure Shell

- Built on asymmetric & symmetric key encryption
- Provides secure communication
- Designed to replace insecure tools under unix
  - ssh (telnet, rsh), scp (rcp), sftp (ftp)
- Other utilities included
  - ssh-agent, ssh-keysign, ssh-keygen, and others
- Tunneling for other protocols, X windows
  - Provides encryption for other programs

# SSH – RSA Keys

- Asymmetric key relationship
  - Built on premise - factoring large numbers is difficult
- RSA provides an initial communication
  - Which is VERY slow compared to symmetric keys
  - Allows a secure transfer of symmetric keys
- Server always provides the public key for communication
  - “RSA key fingerprint is XX:XX...  
Are you sure you want to continue connecting  
(yes/no)?”

# SSH – RSA Keys (cont)

- Keys split into 'public' and 'private' halves
  - Each half is used for communication in a specific direction
- A message encrypted with the 'public' key can only be decrypted with the 'private' key
  - and vice-versa
- SSH server uses the 'private' key to send data to client
  - Client decodes with the public key originally presented

# SSH – Symmetric Keys

- Asymmetric keys are butt slow
  - Symmetric keys takeover after initial RSA exchange
  - Encryption algorithms (block ciphers) use a single shared key
- Key size depends on algorithm used
  - AES supports 128, 192, 256 bit key lengths
- Block size is the max length of input to the block cipher
  - Messages are split into parts equal to the block size
  - This is NOT the same as key size!

# SSH – Large Numbers

- Remember each bit doubles the keyspace
- Years till next ice age  $2^{14}$
- Age of the universe  $2^{34}$
- Number of atoms in the planet  $2^{170}$
- Number of atoms in the sun  $2^{190}$
- Number of atoms in the galaxy  $2^{223}$
- Number of atoms in the universe  $2^{265}$
- Number of bits in an RSA key  $2^{1024}$

# SSH – Block Ciphers

- A Block Cipher is an encryption algorithm
  - Takes a fixed sized input
  - Produces an output of EQUAL size to the input
- A “good” block cipher creates output that is seemingly indistinguishable from random
  - Output is based on the **input and the key**
  - What does “good” mean?
- Many different block ciphers available
  - 3DES, **AES**, Blowfish, and others

# SSH – Modes of Operation

- A block cipher must be reversible
  - Have to get the message back somehow!
  - encryption/decryption uses the shared key
  - The input message *ALWAYS* produces the same ciphertext
- A “mode of operation” is needed for security
  - Provides a way of further randomizing the ciphertext
  - Prevents frequency pattern attacks
- Common modes – CBC (cipher block chaining) and CTR (counter)

# SSH – sshd Configuration

- Usually found under `/etc/ssh/sshd_config`
  - Sometimes `/etc/openssh/sshd_config`
  - Server keys also found in this location
- Many different options for how the server behaves
  - Disallow root access, Disallow empty passwords
  - Allow only client ssh keypairs (no password authentication)
  - Set a login banner
  - And more

# 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 -f ~/.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?

- -----BEGIN RSA PRIVATE KEY-----

Proc-Type: 4, ENCRYPTED

DEK-Info: AES-128-CBC, 974F4776A18993A47368C14F3DEB4D38

```
6m8DSakMwBw9lDgzmQTwdovwcMFTACKUFzqtjILmCTI5L55B8bVF7VJl4Eb/e2Kt
OSMMlxAkLW0n8q5paVF1wvm3/+fhrkmFwC9iAZbuo/vT9jPDKh/uVNMCv+Et dtWL
EOizuVzgsWHj5QXlxxnKsZuWvZJopQ0jcdHMdoTsmFF5DZG/F+H0mDwGgM+IXCxBy
+O72KEcU1vePzfZsGhVJd6DdE6ZmvGbCi3f6Zc5IT2pqUAsOuAvccROg9tTliuf
3DV/DR7DkZdVUIbnvS8WxdrtDFvVawcPGI/LUUEmYd7x3FIH46GRk6WmN1ueBaxt
UYfTt7mci21r//5J7pfbrc2+ftWqCf00gGUFEnBVXahkiumWaiucgrBggd3F7bpf
u6B7ohtIcom+gsOTPYZVYT5rhQVMT0LFrmKKiOaEcGFuiPS2nYGXIC+RjfcDYBLf
BzGuMcDDQpXTYACex1AMr7CzIO59ufRvMsIozyzc+3BsKQeoHek2+eG6uAr/BGGe
IljgtgUd4VENmh7r+N9jOIRk/7WxPH7Xog09nZOYBD0Vz9lUaQWcSHL5sjrV/7P5
hZfy/GasDnaUslhDULSlo1sxW4eC1F0NH4z/oVxOmjm5l4Eb3kgJ+pLoRds1Aetd
nvvh+ugFjP3IL+VqDuM5e299qdkVe/vCmycaG6koi5l2gg+V1aot91YgmEWpb2sX
F4cyDTA8uYYe+baFyDX3XMaqhV0plu8XZ0ni4mDCW4lQ9roHayo00CDs1dW5yKfI
A2lGlaLWzhnwhKl2RECI3II89JCsuduEGGdfkd+G7ianNT74LcaUYMWO n r Yc424Q
-----END RSA PRIVATE KEY-----
```

– Not very readable, is it?

# OpenSSL – Key File (cont)

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

- schenkc@schenktop:~\$ **openssl rsa -in chris-priv.pem -text -noout**

Enter pass phrase for chris-priv.pem:

Private-Key: (1024 bit)

modulus:

00:f0:5b:90:c9:c3:97:c5:6b:03:8b:ab:87:63:d2:

ef:13:da:87:c1:3a:de:5f:e2:51:20:ba:c5:82:c2: ...

publicExponent: 65537 (0x10001)

privateExponent:

0d:fd:d6:8c:d8:34:f2:8a:0b:37:cb:31:63:6f:38:

f9:97:e4:05:2c:8f:1b:57:ca:4f:34:70:20:ef:7a: ...

prime1: ...

prime2: ...

exponent1: ...

exponent2: ...

coefficient: ...

- 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

```
• schenkc@schentop:~$ openssl req -key chris-priv.pem -new -out chris-req.pem
Enter pass phrase for chris-priv.pem:
...
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) []:Department of Computer Science
Common Name (eg, YOUR name) []:schentop.cs.colorado.edu
Email Address []:Christopher.Schenk@Colorado.EDU
...
schenkc@schentop:~$
```

# OpenSSL – Viewing a CSR

- Very similar to viewing your key

- schenkc@schenktop:~\$ **openssl req -in chris-req.pem -text -noout**  
Certificate Request:

Data:

Version: 0 (0x0)

Subject: C=US, ST=Colorado, L=Boulder, O=University of Colorado, OU=Department of Computer Science, CN=schenktop.cs.colorado.edu/emailAddress=Christopher.Schenk@Colorado.EDU

Subject Public Key Info:

Public Key Algorithm: rsaEncryption

RSA Public Key: (1024 bit)

Modulus (1024 bit):

00:f0:5b:90:c9:c3:97:c5:6b:03:8b:ab:87:63:d2:  
ef:13:da:87:c1:3a:de:5f:e2:51:20:ba:c5:82:c2: ...

Exponent: 65537 (0x10001)

Attributes:

a0:00

Signature Algorithm: sha1WithRSAEncryption

3b:0e:e7:65:ba:a9:ce:96:be:3a:ba:35:ac:59:42:f1:51:7b: ...

# 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?