Cryptography, Linux, and You

Hal Canary
h3 at halcanary dot org
http://halcanary.org/

SWFLUG, 2007-04-10

8CAFA2B3
E0B5 263B 6D18 08E5 E9D6 0AFF 7F03 9625 8CAF A2B3

Contents:

- ⇒ Cryptographic Hash Functions
- ⇒ Symmetric Cryptography
  - ⇒ Symmetric Encryption with GPG
  - ⇒ Passphrase Strength
  - ⇒ Encrypting large files with Aespipe
- ⇒ Public-key Cryptography
  - ⇒ Public-key Encryption with GPG
  - ⇒ Digital Signatures with GPG
  - ⇒ Encrypted communication with SSL
  - ⇒ Encrypted communication with SSH

Cryptographic Hash Functions

A **checksum** is a mathematical function that can be used to verify that an input has not been *accidentally* changed.

A **cryptographic hash function** has the additional property that it would be very hard for an attacker to make a change in the input and produce the same output.

Hash Function:	Bits:	Broken?
md5sum	128	Yes
sha1sum	160	Yes
sha256sum	256	No
sha512sum	512	No

Example:

```
$ echo "hello world" > test1
$ echo "hello world." > test2
$ sha1sum test1 test2
22596363b3de40b06f981fb85d82312e8c0ed511  test1
3337bbba15e6a05a29dd0fc658e0541ee185c024  test2
$ sha1sum test1 test2 > SHA1SUM.txt
$ sha1sum -c SHA1SUM.txt
test1: OK
test2: OK
```

```
$ cat test2
hello world.
$ echo "" >> test2
$ cat test2
hello world.

$ sha1sum -c SHA1SUM.txt
test1: OK
test2: FAILED
sha1sum: WARNING: 1 of 2 computed checksums did NOT match
$
```

Symmetric Cryptography

Symmetric Encryption with GPG

Symmetric encryption means that you use the same key to encrypt a message as to decrypt it. An example using GnuPG:

```
$ mkdir secretstuff
$ mv test1 test2 secretstuff/
$ tar -czf secretstuff.tgz secretstuff
$ gpg -c secretstuff.tgz
Enter passphrase:
Repeat passphrase:
$ ls -od secretstuff*
drwxrwxr-x 2 hal 4096 Mar 15 14:12 secretstuff
-rw-rw-r-- 1 hal  197 Mar 15 14:12 secretstuff.tgz
-rw-rw-r-- 1 hal  255 Mar 15 14:14 secretstuff.tgz.gpg
```

```
$ shred -n 2 -u secretstuff.tgz
$ ls -od secretstuff.*
-rw-rw-r-- 1 hal 255 Mar 15 14:14 secretstuff.tgz.gpg
$ gpg secretstuff.tgz.gpg
gpg: CAST5 encrypted data
Enter passphrase:
gpg: encrypted with 1 passphrase
gpg: original file name='secretstuff.tgz'
gpg: WARNING: message was not integrity protected
$ /bin/ls -od secretstuff.*
-rw-rw-r-- 1 hal 197 Mar 15 14:23 secretstuff.tgz
-rw-rw-r-- 1 hal 255 Mar 15 14:14 secretstuff.tgz.gpg
$
```

Passphrase Strength

http://www.iusmentis.com/security/passphrasefaq/strength/

.855	Nonsense phrase. betty was smoking tires in her peace of pipe organs and playing tuna fish.
1.05	A random bunch of characters. A6:o@6 Ls+\` uGX\%3y[k
1.34	Odd capitalization/punctuation and nonsense. Web oF thE Trust is BrokEn cAn You Glue it Back ToGether? and give it xRays.
.280	An average phrase. There is a sucker born every minute.
1.125	Random words. paper factors difference votes behind chain treaties never group
.761	Phrases with some random letters. Ignorance is bliss. spgemxk Education cures ignorance.

Really good passphrases:

```
$ echo 128/8 | bc
16

$ head -c 16 /dev/random | hexdump -e "32/1 \"%02x\" \"\n\""
de4226f80c92e9de1030f4811b8b9a07

$ head -c 16 /dev/random | base64
3kIm+AyS6d4QMPSBG4uaBw==

$ head -c 18 /dev/random | base64
3kIm+AyS6d4QMPSBG4uaB0Gk
```

Encrypting large files with Aespipe

Why use Aespipe? It is much faster than GPG—this makes a difference for big files. This program can be found at
`http://loop-aes.sourceforge.net/`

What to use as a key? Gpg accepts any length, but aespipe wants a longer passphrase.

First create a passphrase and leave it in a file:

```
head -c 57 /dev/random | base64 > pass.txt
```

Aespipe can then use this passphrase:

```
tar cz secretstuff | aespipe -P pass.txt > secretstuff.tgz.aes
```

You can even gpg-encrypt the password file:

```
head -c 57 /dev/random | base64 | gpg -c -a > pass.gpg
tar cz secretstuff | aespipe -K pass.gpg > secretstuff.tgz.aes
```

To decrypt:

```
aespipe -d -P pass.txt < secretstuff.tgz.aes | tar xz
aespipe -d -K pass.gpg < secretstuff.tgz.aes | tar xz
```

Keep the keyfile in a safe place!

(I got $\sim$ 5 MbB/s using this method)

Public-key Crytography

Public-key Encryption with GPG

Public-key cryptography (PKC) uses *different keys* to encrypt and decrypt your message!

$$\text{encrypt} : (\text{cleartext}, \text{publickey}) \rightarrow \text{cyphertext}$$

$$\text{decrypt} : (\text{cyphertext}, \text{privatekey}) \rightarrow \text{cleartext}$$

OpenPGP is a standard for PKC and is based on the original PGP (pretty good privacy) program. GPG is a F/OSS implementation of the OpenPGP standard and is included in most distros. After you have generated a key pair, you will want to publish the public key and keep the private key safe.

There are two modes that you can use PKC. To encrypt a file for someone else, you will need their public key. Only they will be able to decrypt it because only they have their private key.

To sign a file, you will need your own private key. Anyone with your public key can verify that signature.

Use the command `gpg --gen-key` to generate a new key pair.

`gpg --armor --export 8CAFA2B3` will export the public side of the key in a form you can publish.

Send the key to a public keyserver so anyone can search for it:

```
gpg --send-keys 8CAFA2B3 --keyserver wwwkeys.eu.pgp.net
```

Get the fingerprint of your key with `gpg --fingerprint 8CAFA2B3`.

Encrypt to NAME and sign a file with

`gpg --sign --encrypt --recipient "NAME" FILE`.

Digital Signatures with GPG

$$\text{sign} : (\text{text}, \text{privatekey}) \rightarrow \text{signature}$$

$$\text{verify} : (\text{text}, \text{publickey}, \text{signature}) \rightarrow \{\text{pass or fail}\}$$

Things to do with digital signatures:

1) Signing a plain text document
2) Signing a sha1sum
3) Signing a random binary
4) Signing an email
5) Signing another public key

Signing a plain text document

Use `gpg --clearsign file.txt` to create a file called `file.txt.asc`. Use `gpg --verify` to check a signature. Here's an example:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt
mollit anim id est laborum.

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.7 (GNU/Linux)

iD8DBQFF+Z4kfw0WJYyvorMRCu/IAJ4o4ZCLKR2CJyEk2tTX6GnUznzW4ACfRTlZ
qLBEm0zTKzRhoDX7Yi4IXuE=
=cgGl
-----END PGP SIGNATURE-----
```

Signing a sha1sum

Use the command

```
sha1sum test1 test2 | gpg --clearsign > SHA1SUM
```

to produce a signed hash of these two files. Verify the signature with `gpg --verify SHA1SUM` and verify the checksum with `sha1sum -c SHA1SUM`.

Test them both with:

```
gpg < SHA1SUM | sha1sum -c
```

An example from Red Hat:

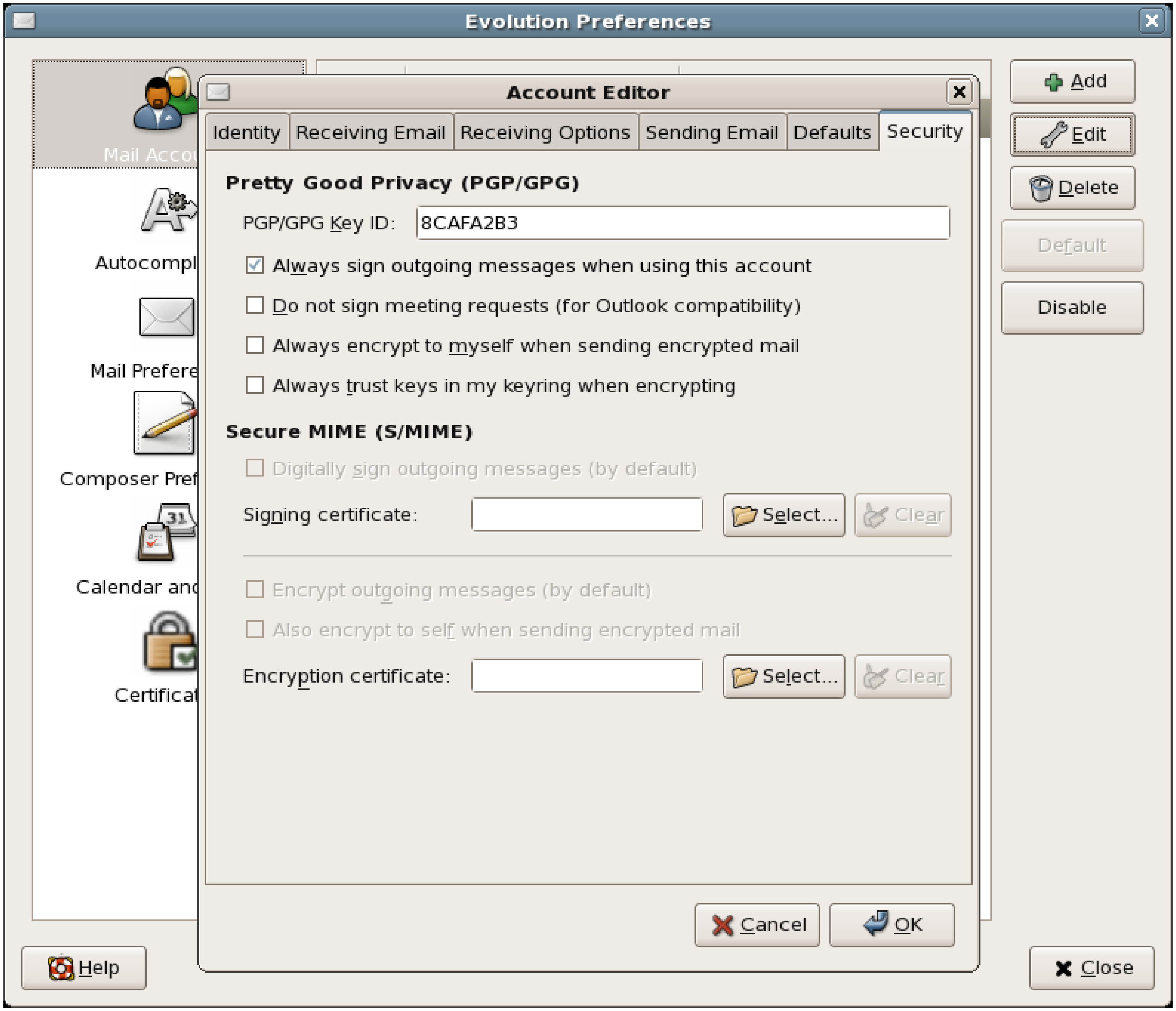
```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

834fd761b9c0a5dc550d10d97307dac998103a68  FC-6-i386-rescuecd.iso
cc503d99c9d736af9052904a6ab14931b0850078  FC-6-i386-disc1.iso
3051710e6b2f1d17a14ede0ebb74761c29cda954  FC-6-i386-disc2.iso
5357ce21f8766db385b25923216a430b694bca5d  FC-6-i386-disc3.iso
d6133ab5ccf19431c14fd2ad85bce03c9834ef87  FC-6-i386-disc4.iso
6722f95b97e5118fa26bafa5b9f622cc7d49530c  FC-6-i386-DVD.iso
22327af62d6376916e209b0c4934540e14d5664a  FC-6-i386-disc5.iso
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.2.6 (GNU/Linux)

iD8DBQFFNo/utEJp0E8qb9IRAsf7AJ9ZqiDlKqJfAh8g5QHyDMmPOzNbTACfbyGw
hB8bkLBT+6ANW6y8iBmlxz8=
=O/Le
-----END PGP SIGNATURE-----
```

Signing a random binary

`gpg --armor --detach-sign FILE` will create a separate file (FILE.asc) with a signed checksum. This has a disadvantage over the sha1+gpg method in that you need a copy of GPG and your public key to verify the checksum.



Pretty Good Privacy (PGP/GPG)

PGP/GPG Key ID: 8CAFA2B3

☑ Always sign outgoing messages when using this account

□ Do not sign meeting requests (for Outlook compatibility)

□ Always encrypt to myself when sending encrypted mail

□ Always trust keys in my keyring when encrypting

Secure MIME (S/MIME)

□ Digitally sign outgoing messages (by default)

Signing certificate: ____ Select... Clear

□ Encrypt outgoing messages (by default)

□ Also encrypt to self when sending encrypted mail

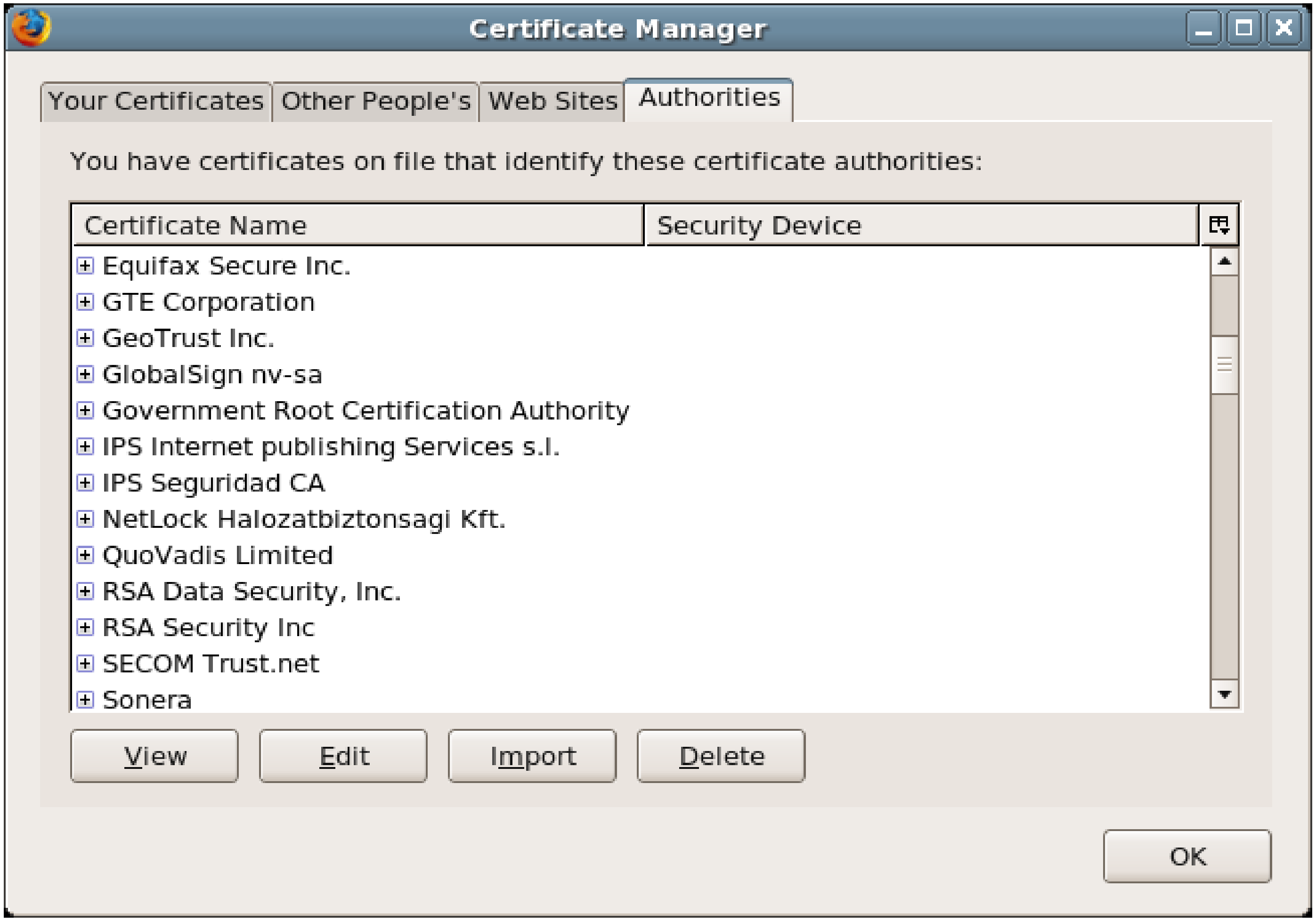
Encryption certificate: ____ Select... Clear

Encrypted communication with SSL

SSL stands for “Secure Socket Layer.” It is used for https communication and makes use of PKC. Every server has a public-private key pair. Most of the time, your browser decides to trust a server because that server gives a copy of its public key that has been digitally signed by a certifying authority (CA) that your browser has been programmed to trust:

The CA is there to prevent a man-in-the-middle attack.

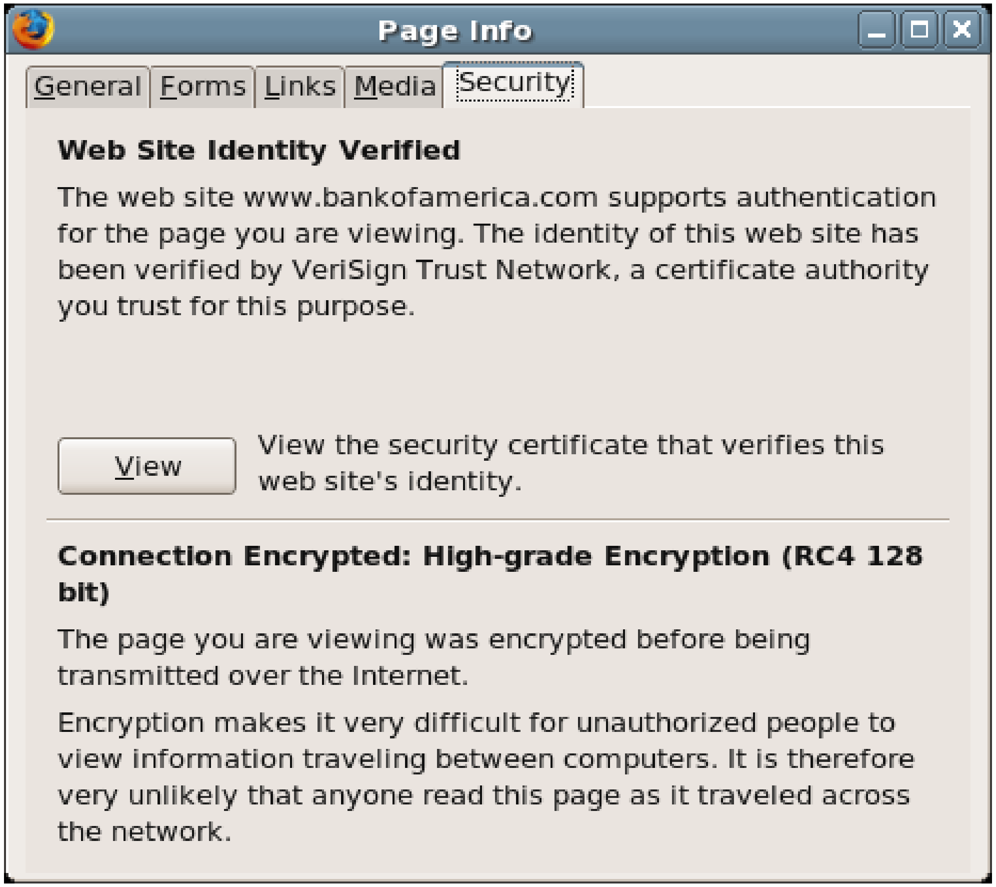
Screenshot from Firefox:



You have certificates on file that identify these certificate authorities:

Certificate Name	Security Device
⊞ Equifax Secure Inc.	
⊞ GTE Corporation	
⊞ GeoTrust Inc.	
⊞ GlobalSign nv-sa	
⊞ Government Root Certification Authority	
⊞ IPS Internet publishing Services s.l.	
⊞ IPS Seguridad CA	
⊞ NetLock Halozatbiztonsagi Kft.	
⊞ QuoVadis Limited	
⊞ RSA Data Security, Inc.	
⊞ RSA Security Inc	
⊞ SECOM Trust.net	
⊞ Sonera	

Screenshot from Firefox:



Web Site Identity Verified

The web site www.bankofamerica.com supports authentication for the page you are viewing. The identity of this web site has been verified by VeriSign Trust Network, a certificate authority you trust for this purpose.

View the security certificate that verifies this web site's identity.

Connection Encrypted: High-grade Encryption (RC4 128 bit)

The page you are viewing was encrypted before being transmitted over the Internet.

Encryption makes it very difficult for unauthorized people to view information traveling between computers. It is therefore very unlikely that anyone read this page as it traveled across the network.

Encrypted communication with SSH

SSH stands for ”Secure SHell.” There are several implementations of the standard. Since there are no central CAs for SSH, you need to manually verify a server’s key fingerprint before trying to log on. In this sense, it is like PGP.

I carry around a slip of paper with my server’s SSH key fingerprint on it.

```
$ (cd /etc/ssh/;for x in s*.pub;do ssh-keygen -l -f $x;done)
1024 11:70:ad:d8:15:ec:75:89:22:c1:b7:dc:b3:30:e1:10 ssh_host_dsa_key.pub
2048 67:57:91:96:66:60:9b:f0:b0:90:1a:a6:76:12:b7:c5 ssh_host_key.pub
2048 55:be:0d:d2:7f:9d:2e:3f:a6:2d:03:fa:a4:b6:09:7b ssh_host_rsa_key.pub
```

```
$ ssh example.com
The authenticity of host 'example.com (71.3.117.142)' can't be
established.
RSA key fingerprint is 55:be:0d:d2:7f:9d:2e:3f:a6:2d:03:fa:a4:b6:09:7b.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'example.com,71.3.117.142' (RSA) to
the list of known hosts.
```

Fun things to do with SSH.

1) Copy files:

```
scp FILE hal@example.com:.
scp hal@example.com:FILE .
```

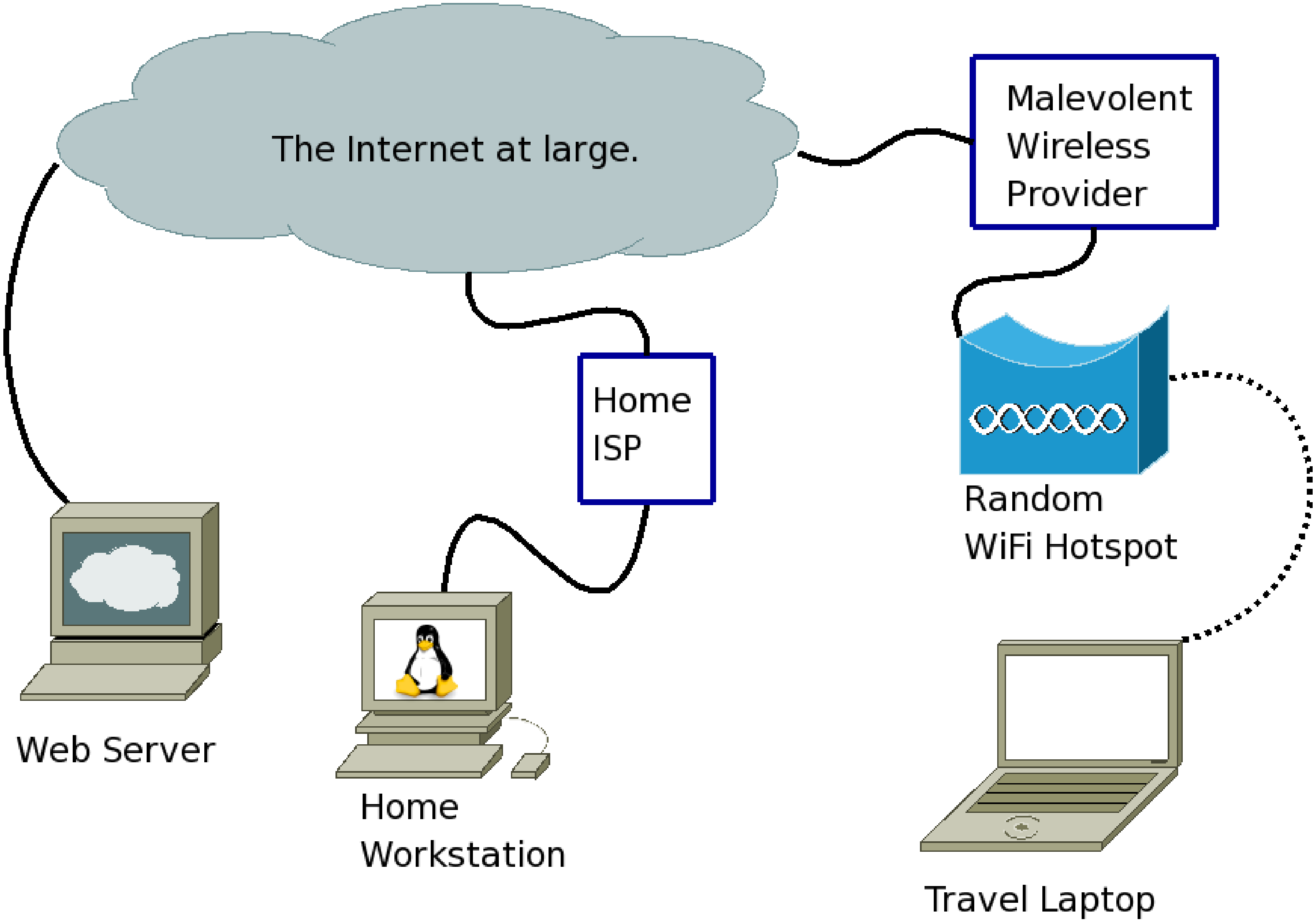
2) Copy directories:

```
scp -r directory hal@example.com:
rsync -e ssh -avz directory hal@example.com:
```

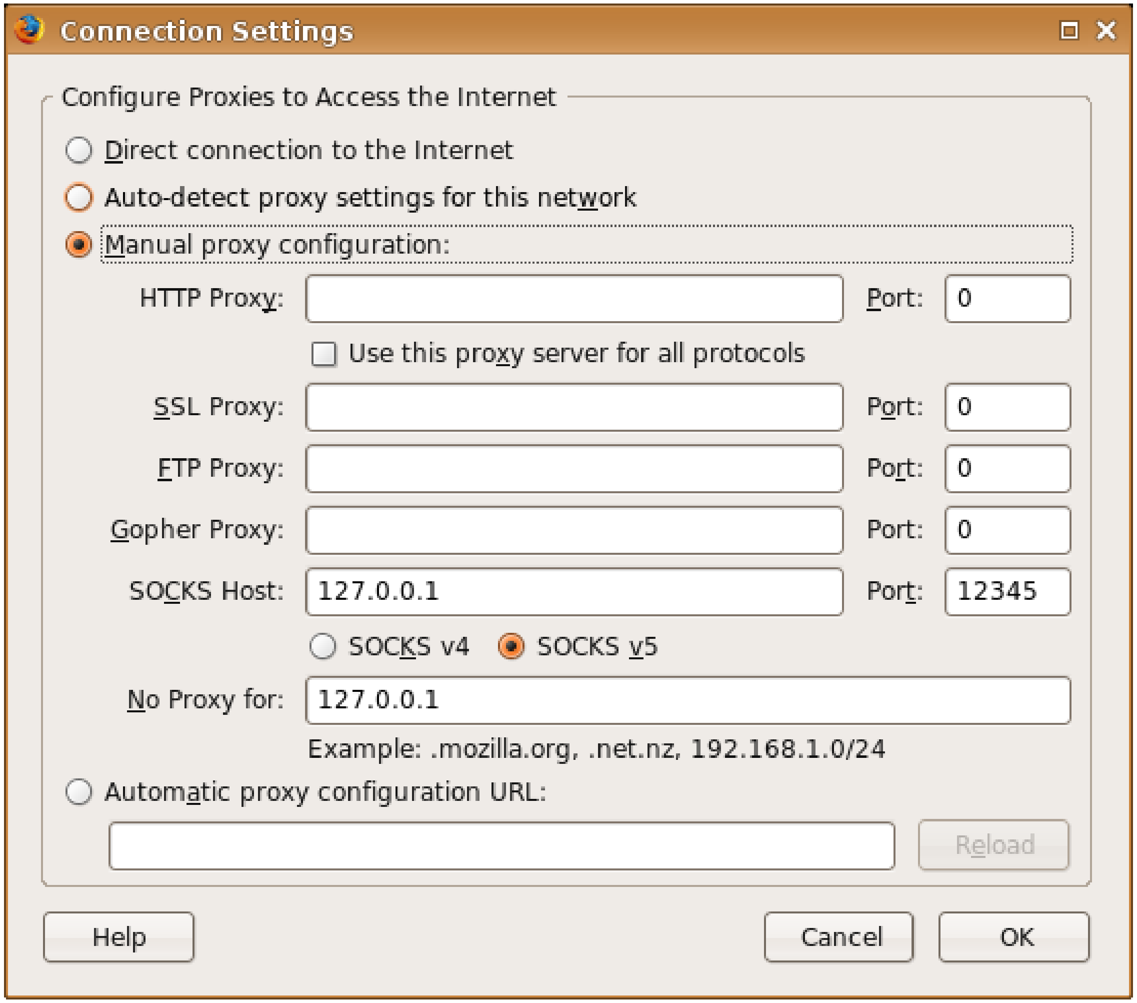
(note that rync can either connect to a remote sshd daemon or a remote rsync daemon!)

3) Forward ports.

```
ssh -Y hal@example.com
ssh -D 12345 hal@example.com
```


The Internet at large.
Malevolent Wireless Provider
Home ISP
Random WiFi Hotspot
Web Server
Home Workstation
Travel Laptop

Firefox⟶Edit⟶Preferences⟶Advanced⟶Network⟶Settings



Connection Settings

Configure Proxies to Access the Internet

○ Direct connection to the Internet

○ Auto-detect proxy settings for this network

● Manual proxy configuration:

HTTP Proxy: Port: 0

□ Use this proxy server for all protocols

SSL Proxy: Port: 0

FTP Proxy: Port: 0

Gopher Proxy: Port: 0

SOCKS Host: 127.0.0.1 Port: 12345

○ SOCKS v4 ● SOCKS v5

No Proxy for: 127.0.0.1

Example: .mozilla.org, .net.nz, 192.168.1.0/24

○ Automatic proxy configuration URL:

Conclusions?