Setting Up Working Environment

Ivan "Rambius" Ivanov

New York City BSD User Group

Septemer, 2019

What Will I Talk About

During the years I have asked for and given advice (mainly asked for), tips and tricks on how to work efficiently in Unix. The topics include SSH usage, X Window usage, processing large logs file to ease debugging and so on.

Starting Point

A desktop and a list of servers to maintain - wiki, bugtracker, build machines, QA machines. If we are lucky the desktop will have Unix as well; if not we need a tool like Cygwin or Putty.

Poking Around

```
$ ssh wiki
```

Two prompts will appear:

```
The authenticity of host 'wiki' can't be established.
ECDSA key fingerprint is SHA256:G+YEo1VU...
Are you sure you want to continue connecting (yes/no)?
Password for rambius@wiki:
```

We don't want to type "yes" and our password constantly.

Host Verification Prompt

Wen we type “yes”, the host’s public key is added in /.ssh/known_hosts.
Can we add it in advance? Yes!

ssh-keyscan

The command `ssh-keyscan` shows the SSH server's public key in format that can be added to `known_hosts`

```
ssh-keyscan -t rsa wiki
ssh-keyscan -t rsa wiki >> ~/.ssh/known_hosts
```

`-t` specifies the key's type - rsa, dsa, etc.

Avoiding duplicates in known_hosts

If we run the second `ssh-keyscan` command more than one time inadvertantly we will get duplicate entries in `known_host`. Not that it hurts but the command `ssh-keygen` can tell if a host key has been already added:

```
ssh-keygen -F wiki
```

returns 1 if the host is not added; dumps it otherwise

ssh-keygen and ssh-keyscan

Putting it all together:

```
host=wiki
if ! ssh-keygen -F $host
then
  ssh-keyscan -t rsa $host >> $known_hosts
fi
```

Host Verification Prompt - Conclusion (For Now)

We can loop over the list of hosts and use the snippet above we can prepopulate `known_hosts`. Still that does not guard us from MITM. We will discuss a better solution later.

Password Prompt

Passwords suck in general. They can expire, be forgotten, stolen or mistyped and are hard to automate. SSH keys are better - although they are more secure, they are also more convenient.

Generating SSH Keys

To generate a key pair on the local machine run

```
ssh-keygen
```

To prevent it from overwriting existing keys use

```
ssh-keygen -f <output_file>
```

Explaining ssh-keygen

By default the public key goes to `/.ssh/id_<type>.pub`, the private key to `/.ssh/id_<type>`.
The private key authenticates the user to the system, so it must be kept secret.
ssh-keygen prompt for a passphrase to encrypt the private key with with to increase its protection. I always put passphrases on my keys.

Authorizing a Key

The remote side must authorize the key before we can authenticate with it. The authorization consists of adding the public key to `~/.ssh/authorized_keys`. We need to distribute the public key to that remote file.

Distributing the Public Key

There are several ways to add the public key to `authorized_keys`:

- Copying the key to the remote box:

```
scp ~/.ssh/id_rsa.pub wiki:~/
ssh wiki "cat id_rsa.pub >> ~/.ssh/authorized_keys"
```

- Using SSH stdin and stdout

```
cat ~/.ssh/id_rsa.pub | ssh wiki \
  "cat - >> ~/.ssh/authorized_keys"
```

- Using `ssh-copy-id`

```
ssh-copy-id -i ~/.ssh/id_rsa.pub wiki
```

ssh-copy-id

I prefer `ssh-copy-id`, because it checks if the key is already in `authorized_keys`. `ssh-copy-id` prompts for the password and usually this is the only time I have to type my password.

Distributing the Public Key En Mass

If we use ssh-copy-id for multiple hosts, we will be prompted multiple times for the password. If the user authenticates with a same password everywhere (the case of Active Directory or LDAP user stores) we can be prompted once and use the password for all hosts.
I have implemented that scenatio using TCL / Expect. Expect is excellent for automating interactive programs.

Expect - Password Prompt

The following Expect procedure reads the password

```
proc promptpass {{msg "Password: "}} {
    stty -echo
    send_user -- $msg
    expect_user -re "(.*)\\n"
    send_user "\\n"
    stty echo
    return $expect_out(1,string)
}
```

It disables characters echoing, prints a prompt, captures the password and enables echoing again.

Expect - The Main Procedure 1

The main procedure takes a file with one host per line and calls `ssh-copy-id` on each host:

```
set pass [promptpass]
set hf [lindex $argv 0]
set hh [open $hf]
set timeout 60
set pubkey "$env(HOME)/.ssh/id_rsa.pub"
while {[gets $hh host] >= 0} {
    spawn ssh-copy-id -i $pubkey $host
    expect {
        "*?assword*" { send "$pass\n" }
    }
}
close $hh
```

Expect - The Main Procedure 2

`spawn` runs `ssh-copy-id`. `expect` waits until `ssh-copy-id` outputs "Password:" or "password:" and then sends the password.
I needed to increase `spawn`'s timeout, because some of my hosts were slower.
Another tool called `sshpass` can also supply the password to `ssh-copy-id`. I used `expect` because it was preinstalled.

Where All This Comes Handy

Once I screwed up my window manager's configuration and decided to run `rm -rf *` in its configuration directory. It turned out I ran it in `/.ssh`. I was so happy I could easily restore all SSH setup.

SSH Passphrases

At this point we handled host verication and key authentication. If we login we will get a prompt about the private key's passphrase:

```
$ ssh wiki
Enter passphrase for key '/home/.../id_rsa':
```

The passphrase is used to decrypt the private key. Upon entering it the key is "unlocked" and we can log in.

Passphrases vs Passwords

The main difference between passwords and passphrases is that the password travels to the remote side; the passphrase never leaves the local machine. In fact, using SSH keys no private information travels to the remote side.
The passwords cannot be “preloaded”. The encrypted key can be unlocked in advance using SSH agents.

SSH Agents and Passphrases

`ssh-agent` is an authentication agent that holds the private key for key authentication. When started, it holds no key. `ssh-add` prompts for the passwphrase, unlocks the key and adds it to a running SSH agent.

SSH Agent - A Typical Usage

```
$ eval `ssh-agent`
$ ssh-add
Enter passphrase for /home/.../id_rsa
Identity added: /home/.../id_rsa (/home/.../id_rsa)
$ ssh wiki
wiki$:
```

At that point of time we can ssh with no prompts from the current terminal.

Reusing a SSH agent 1

The SSH agent communicates with `ssh-add` and `ssh` by setting `SSH_AGENT_PID` and `SSH_AUTH_SOCK` environment variables. If those variable are not set, for example when we start a new terminal (not from the current one), `ssh-add` and `ssh` cannot find the ssh agent even if it is running.

Reusing a SSH agent 2

I have tried sharing ssh agent's variables by

```
ssh-agent > ~/.ssh/agent
```

and in then sourcing `~/.ssh/agent` in `.profile`, but that does not work if `ssh-agent` is not running at all. We could examine `ps` if it is running and then start it, but that got complicated (even more with X Window) and I never made it work reliably.

Keychain to reuse ssh-agent

keychain allows the re-use of a single ssh-agent between terminals, shell and X Window sessions and cron jobs. The usage is simple:

```
$ keychain ~/.ssh/id_rsa
# Supply the passphrase
$ . ~/.keychain/‘hostname‘-sh
```

We source /.keychain/‘hostname‘-sh from .profile and we will have an ssh-agent available.

ssh-agent and ssh-copy-id

I recently discovered that if a key is loaded into an SSH agent, `ssh-copy-id` fetch it from the agent and copy it to a host.

```
eval `ssh-agent`
ssh-add ~/.ssh/id_rsa
# or
keychain ~/.ssh/id_rsa
```

```
ssh-copy-id <host>
```

that is there is no need to supply `-i` option to `ssh-copy-id`.

X Window

Most of the boxes I work with have many X clients installed on them. I find it convenient to run xterm, emacs, etc remotely.
The X Window terminology is a little bit reverse. The server is the display where the application are drawn, often the local desktop. The X clients run on the remote machines.

X Window - The Server

`startx` starts the X Server on the local machine. `~/.xinitrc` contains additional initialization such as what clients should be run (at least one xterm) or what window manager.

X Window - The Clients

The X client connect to the X server over the network using (insecure) connection. The DISPLAY variable (on the X client's box) contains the location of the X server.
The commands `xhost` and `xauth` provide ways to authenticate X clients to the X server.

X Forwarding

An X protocol connection can be forwarded through an SSH connection to provide more security and stronger authentication.
When using SSH with X11 forwarding `sshd` runs `auth` on user's behalf to add it to `.Xauthority`. `ssh` also sets the DISPLAY variable.

Running X Client Remotely

From an `xterm` (or another terminal emulator) running on the local machine (the X server) we connect to the box with the X client over ssh with trusted X11 forwarding:

```
$ ssh -Y xclientbox
xclientbox$ xterm &
xclientbox$ emacs &
```

`xterm` and `emacs` appear in the X server on our desktop.

X Server and keychain

To make sure the ssh command above and all other ssh invokations have an SSH agent we can provide one in .xinitrc:

```
sshkey=$HOME/.ssh/id_rsa
if ! ssh-add -l 2>&1 | grep -q $sshkey
then
  keychain --timeout 600 $sshkey
fi
if [ -f $HOME/.keychain/‘hostname‘-sh ]
then
  source $HOME/.keychain/‘hostname‘-sh
fi
openbox-session & wmpid=$!
wait $wmpid
```

X Server and `keychain` - Explanations

`ssh-add -l` lists the keys loaded in the SSH agent. If the key is not added or an SSH agent is not running, we run `keychain`, source its file and run the window manager.

Shortcuts in X

For no particular reason I am using Openbox with fbpanel. I access two servers most frequently so I have added shortcuts to them in fbpanel. The shortcut calls

```
xterm -hold -e /bin/sh -c "ssh <host>"
```

Host Verification Revisited

The only place where SSH needs human intervention is when it makes the very first initial connection to a host. It cannot possible decide if the public key presented by the remote host really belongs to it or it is a MITM.

Funny Story with a Support Person

I had a chat with a support person from a web hosting company. He told me to ssh to a host. I prompty did so and I was greeted by the message with to approve or reject the host's fingerprint. I had time to waste so I asked the person to supply the fingerprint so that I could compare it. I waited 20 mins and he was still could not send it...

Host Verification Bad Practices

I have seen people fighting host verification out of confusion or lack of knowledge. Usually they set `StrictHostKeyCheckng=no` or `UserKnownHostFile=/dev/null`.

The Solution - SSHFP

The solution is to move host verification out of the users. SSHFP allows storing SSH fingerprint in a DNS server. After DNSSEC is used to sign zone we get assurance that the fingerprints are genuine.
Thus the people who usually provision the boxes can also add their fingerprints to the DNS infrastructure.

A Case Study with OpenBSD and Unbound

I have a spare APU board, where I setup OpenBSD and Unbound to set SSHFP.
DISCLAIMER: I am no expert in DNS, so take the following slides with a grain of salt.

Unbound Setup 1

The first step is to establish the initial key to sign the zone.
`unbound-anchor` command creates the initial `root.key`.
`unbound-anchor` should also run on startup to update that file.
`unbound.conf` should have the location of that file as well:

```
server:
...
auto-trust-anchor-file: "/var/unbound/db/root.key"
```

Unbound Setup 2

Once the zone is signed SSHFP fields for the host should be added. `ssh-keygen -r <host>` can print them:

```
$ ssh-keygen -r <host>
denica IN SSHFP 1 1 e2cf36c947...
denica IN SSHFP 1 2 4140de402c...
denica IN SSHFP 2 1 9df77afeec...
denica IN SSHFP 2 2 311d5be4ff...
denica IN SSHFP 3 1 d1924f8977...
denica IN SSHFP 3 2 1be604a355...
denica IN SSHFP 4 1 0fe1724ba5...
denica IN SSHFP 4 2 d0bdf938f7...
```

Unbound Setup 3

Copy the fields to `unbound.conf`

```
server:
...
local-zone: "zone." static
local-data: <host>.zone. IN A 192.168.1.2
local-data: "apu.supernova IN SSHFP 1 2 ..."
local-data: "apu.supernova IN SSHFP 2 2 ..."
local-data: "apu.supernova IN SSHFP 3 2 ..."
local-data: "apu.supernova IN SSHFP 4 2 ..."
```

Restart unbound

SSH Client Verifying the Fingerprints from DNS

Run `ssh` with `VerifyHostKeyDNS=yes`

```
ssh -o VerifyHostKeyDNS=yes <host>
```

It should verify the host against the DNS and will not prompt even if the host's key is not in `known_hosts`.