SSH tips and tricks

Why use SSH?

- In earlier times, servers were used to be remotely managed by non-secure methods like rlogin and telnet.
- The problem with these methods is that user credentials as well as other data get transferred to and from the server in clear text.
- SSH adds an encryption layer to the communication process to protect data from any possible leakage.
- SSH is not only applied to remote server management, but it can also secure other applications/protocols like VNC, and many other TCP/IP based programs and protocols.
- SSH works by encrypting any data packets that pass through its channel from a source host, and decrypt this data when received on the other side.

Basic usage

- If you want to connect to RedHat03 with a username sam, you'd type the following on the terminal:
  ```
  ssh -l sam RedHat03
  ```
  or
  ```
  ssh sam@RedHat03
  ```
- The username can be omitted if it is the same username you're logged in with:
  ```
  [sam@RedHat01 ~]# ssh RedHat03
  ```
- After successful connection, you can use RedHat03 to connect to another machine (say RedHat02) using FTP or telnet. But this time, the connection between you (on RedHat02) and RedHat03 is not secure unless you use SSH for the second time.
- You can also use SSH to transfer file securely between hosts using the scp command. For example:
  ```
  scp sam@RedHat03:myfile.txt .
  ```
  The above command will securely download myfile.txt from Sam's home directory (denoted by the colon ":") to the local current directory.
- The same thing holds for uploading files using scp:
  ```
  scp anotherfile.txt sam@RedHat03:
  ```
  This will upload anotherfile.txt from the current directory to Sam's home directory on RedHat03
- You can upload/download the file to a new name by specifying that name in the path instead of just the directory. For example:
  ```
  scp anotherfile.txt sam@RedHat03:newfilename.txt
  ```

Avoid man in the middle attacks

- Man-in-the-middle is one of the most common attacks on client-server connections.
- Consider the following example:
  - You are connecting to RedHat03 using SSH.
  - Unfortunately, the DNS server has been compromised. RedHat03 is now pointing at the IP of a machine used by the attacker
  - This machine has a “cooked” version of SSH-server
  - This cooked version will accept your username and password, then save them for the attacker
  - To cover up its tracks, the rogue SSH server will either disconnect your session and deny future access (perhaps with a under-maintenance message) until the attacker returns the DNS record to the way it was, or it may connect you to the real server and monitor your entire session.
- To combat such a scenario, SSH uses the known-hosts mechanism to verify the identity of the remote host. When you establish the connection for the first time, SSH asks you to accept (or deny) the identity of the remote server. If you chose to accept it, it gets stored in a file under ~/.ssh/known_hosts. Now whenever you want to connect this server, SSH will compare the identity of the server to the one stored, and will abort the connection if they are not identical.
- Of course this is not a 100% bullet-proof solution; as the attack may occur on the first-time connection (while exchanging the public key) but it’s better than none.

Key-based authentication

- What’s wrong with using passwords for SSH authentication? Well, a number of drawbacks:
  - A strong password (or a passphrase) can be a challenge to remember
  - It is prone to attacks by an altered SSH server on the remote host
  - It is difficult to be shared among a group of admins (for example the root password)
- For those reasons, SSH allows the use of a more secure and practical approach: public/private keys.
- A public key is the one placed “publicly” on the server. Anyone can view it
- A private key is the one that stays with the client (the user) no one should be allowed to access or view this key except the user.
- The private key must “fit” into the public key the same way a physical key first into its lock.
- To authenticate, the client presents the private key to the server, the server does some mathematical operations using the public key to ensure that the key is valid and the user can be authenticated

LAB: configure key-based authentication

- Target: configure SSH connections to RedHat03 server to use keys instead of username/password combination.
- On the client machine, use ssh-keygen to create the key pair:

  ```
  ssh-keygen
  ```
- You may want to enter a passphrase to protect the private key file on your local machine. You will be required to enter this passphrase whenever you want to use the private key to login.
- This will create two keys under .ssh directory in your home directory by default: id_rsa (the private key) and id_rsa.pub
- You need to transfer the public key to the remote host to be used for your authentication. You can either use the ssh-copy-id command as follows:

  ```
  ssh-copy-id sam@RedHat03
  ```

  Or you can manually add the key to the server by appending the contents of the `id_rsa.pub` to `~/.ssh/authorized_keys`
- Try to login now with your key:

  ```
  ssh -i id_rsa root@RedHat03
  ```

  You should be logged in without using a password
- Optionally, you can disable password authentication in `/etc/ssh/sshd_config` to force users to use key-based authentication and enforce the security policy

Secure FTP (sftp)

- It is not a “secure version” of the famous FTP program; it is a separate SSH subsystem that uses the same FTP commands to transfer files.
- There is no ASCII mode, only binary (in classic FTP, ASCII mode translated line endings between Windows and UNIX based machines)
- Sftp uses the same SSH credentials to open a file transfer session with the server. You can use the following commands:
  - `get` *`file`*: downloads the file
  - `put` *`file`*: uploads the file
  - `mget` *`file(s)`*: downloads multiple files.
  - `mput` *`file(s)`*: uploads multiple files.
- You can also use your private key with sftp (if the remote server is configured) using the following command:

```
sftp -o IdentityFile=~/.ssh/id_rsa RedHat03
```

- You can also place the path to your private key in `/etc/ssh/ssh_config` (don’t confuse this file with `/etc/ssh/sshd_config`)

SSH configuration options

- The SSH daemon can be configured to tighten the system security even more.
- The configuration file is located in `/etc/ssh/sshd_config`. The following are some of the available options (you can read the full documentation using `man -5 sshd_config`). You have to restart sshd service after any change to this file:
  - `Port 22` you can change this to the port where you want SSH to be listening at
  - `ListenAddress 0.0.0.0` if you have multiple network interfaces, you can configure sshd to listen only on one or more interfaces by specifying the IP address of the interface. The default is to listen on all the available.
  - `Protocol 2` use SSH v2. You can opt to use SSH v1 but it is highly recommended to keep using SSH v2 because of massive security enhancements
  - `PermitRootLogin yes` many sys admins choose to set this to no denying direct root user login over the network. The administrator has to log on with a normal account and use sudo or su to execute *rootly* commands.
  - `PasswordAuthentication yes` enable/disable password based authentication. You may want to set that to no if you are using key based authentication.
  - `AllowUsers|AllowGroups` If either of those exists, only the mentioned users/groups will be allowed to access the server using SSH.
  - `DenyUsers | DenyGroups` if wither of those exists, all users/groups will be allowed access to the server except the ones mentioned
  - ClientAliveInterval 600
    ClientAliveCountMax 0
    Those two lines configure the maximum idle time before the user's SSH session will automatically terminate. In this example, the timeout is set to 600 seconds (5 minutes)

Tunneling application traffic

- You can apply SSL security to insecure applications by using TCP forwarding
- SSH will also add a layer of authentication which provides more protection
- The application packets travel through an SSH “wrapper” which may also encrypts and may compress them.
- This may be useful also in situations when SSH is the only allowed protocol/port on the firewall; other protocols can be tunneled through SSH.
- This requires that the application be configured to use another port than then native one. For example, an SMTP client that is configured to use port 25 must be configured to use the port that you used in tunneling (say 2525)
- Forwarding works by directing an application client traffic to an SSH tunnel instead of directly to the network. The server on the other side uses SSH to receive and decrypt this traffic and directs it to the original listening port.
- Both the client and the server are not aware that their traffic is being tunneled, accordingly, they behave as expected.
- Most TCP/IP applications can make use of tunneling with the exception of FTP. You can use another totally different implementation like SFTP or vsFTP to achieve secure file transfer.

LAB: secure telnet connection using SSH tunneling

- Target: connect to RedHat03 from RedHat01 using telnet through an SSH tunnel
- On RedHat03 turn on the firewall an block port 23 (telnet port)
```
iptables -A INPUT -p udp --dport 23 -j DROP
```
- Ensure that you cannot connect to RedHat03 using telnet
```
telnet RedHat03
```
- On RedHat01 establish an SSH tunnel that will map the local port 1234 (you can choose any port number) on localhost to port 23 on RedHat03
```
ssh -L 1234:localhost:23 RedHat03 -N
```
- Try to connect to RedHat03 from RedHat01 using port 1234 on localhost
```
telnet localhost 1234
```

LAB: using an intermediate server

- Sometimes the situation is more complicated, you may not have direct access to the destination host, but you do have access to a server that is allowed to communicate with the destination host (an intermediate server). Tunneling can help you here.
- Configure the firewall on RedHat03 to deny any traffic except from RedHat02 (192.168.0.251)

```
iptables -A INPUT -s 192.168.0.251 -j ACCEPT
iptables -A INPUT -j DROP
```

- Confirm that you cannot access RedHat03 from RedHat01 but you can access it from RedHat02
- Establish a tunnel between RedHat01 and RedHat03 through RedHat02

```
ssh -L 1234:RedHat03:23 RedHat02 -N
```

- Test the connection using telnet, you should be connected to RedHat03

```
telnet localhost 1234
```

Remote tunneling

- Consider the following situation: you are behind a firewall that denies any inbound connections. But you can access the outside world. You want to be able to access the server remotely. You can use “remote tunneling” to do this.
- Reverse tunneling works by establishing an SSH connection to a server, this server can be used as a tunnel to route traffic back to the originating one.
- The same rules that apply to “local” tunneling applies to remote tunneling, just replace –L with –R

LAB: use remote tunneling to access a machine that is behind firewall

- Target: RedHat03 cannot be accessed from outside because it is behind a firewall. However, it is allowed to access other machines. You are going to make a remote tunnel to be able to access it through the tunneled machine.
- Configure the firewall on RedHat03 to deny all inbound traffic
  ```
  iptables -A INPUT -j DROP
  ```
- Establish a remote SSH tunnel from RedHat03 to RedHat02 using a defined port: 1234
  ```
  ssh -R 1234:localhost:23 RedHat01
  ```
- Using RedHat01, try to establish a telnet connection to RedHat03 using port 1234 on the localhost
  ```
  telnet localhost 1234
  ```

LAB: using dynamic tunneling (SOCKS proxy)

- In the previous examples, local and remote ports where explicitly mapped and defined. In some situations you may not know which ports the SSH tunnel has to negotiate. That is, you need a SOCKS proxy.
- To use a SOCKS proxy, the application must be aware of how to deal with it. The most famous example of those is web browsers. In this lab we are going to configure Firefox on a machine with no Internet access to browse the Internet through an SSH tunnel to a machine that does have Internet access.
- On the client machine establish the SSH tunnel as follows:
  ```
  ssh -D 8080 root@192.168.0.253
  ```
- The configure Firefox on the same machine to use a SOCKS proxy of:
  ```
  localhost:8080
  ```
- Ensure that you can browse the Internet.