15.6.1 OpenSSH

Click one of the buttons to take you to that part of the video.

OpenSSH 0:00-1:05

In the very early days of UNIX, and Linux as well, we used a variety of different tools to establish network connections between systems. For example, you could access the shell prompt of a remote system over the network using Telnet, rlogin, or rshell. Likewise, you could copy files back and forth between systems using RCP or FTP. But there was a key problem with using these tools.

Understand that each of these utilities had one glaring weakness. Network services, such as Telnet, rlogin, RCP, rshell, and FTP, transmit data as clear text. This is bad because it means anyone running a sniffer can easily capture data that's being transferred back and forth over the network segment between two systems.

For example, they could capture your username and your password as you authenticate to the remote system. And as you transfer files back and forth between systems, a sniffer can easily capture the contents of these transmissions.

Unencrypted Communications 1:06-2:48

In this example, let's suppose that I have a client system over here, and I need to remotely manage my Linux server over here on the network, and I'm lazy and I don't want to get up from my desk and walk all the way into the server room to access the keyboard and console of that server.

So what I do is fire up a Telnet client on my workstation, and I establish a Telnet session between my workstation and my server. In order to access the console of the Linux server over here, I have to provide my username and my password.

And as I'm working, I realize that I need to perform some root level tasks. So I use the su command to switch to my root user account over here on the Linux system. Of course when I do that, I have to provide my password for the root user. I go ahead and complete my work and then I log out.

Well, during this whole time, it is possible for someone over here who has not good intentions to actually capture these packets as they are being transferred back and forth between the server and my workstation, using the Telnet protocol.

Because Telnet does not encrypt the information, it would be very easy for this nefarious person to capture both my username and my password when I initially established my connection between the systems, using Telnet.

Then, when I ran the su command, they would also be able to capture the root user's password. This is not a good thing because in the situation, the attacker right here would have everything that he or she needed to gain full, unfettered access to my Linux server. Bad, bad, bad.

OpenSSH Components 2:49-4:19

To keep this from happening, you can install and use the OpenSSH package. In using OpenSSH, you can complete the same management tasks that we've already talked about, but do so securely, because OpenSSH provides encryption.

Basically, OpenSSH provides all the functionality we used earlier with Telnet, rlogin, RSH, RCP, and FTP, and so on, but it does so with encryption implemented.

In order to do this, OpenSSH provides several encryption-enabled components. The first is the sshd daemon. This is the SSH server that allows remote access to the system. We also have the SSH client. This is the client that we use to connect to the sshd daemon on a different system.

If we need to copy files securely between two systems, we can use the scp command. scp stands for secure copy. Using scp, we can send files between systems, and the contents of those files is encrypted. Even if a nefarious person is able to capture the packets to sniff them, it will all be a jumbled mess to them because they don't have the key needed to decrypt the transmission.

You can accomplish the same thing with SFTP. It uses a secure form of FTP to transfer files between systems. And you can also use slogin. This is not a tool I use very often. I prefer to use SSH client myself, but if you need to, you can also use slogin to access the shell prompt of a remote system over the network, and do so securely.

Encrypted Communications (SSHv1) 4:20-8:59

In order to establish a secure connection, OpenSSH actually uses two sets of keys. First, it uses a private/public key pair to provide public key encryption, and it also uses secret key encryption.

Remember, with private/public key encryption, the private key remains on the host, while the public key is freely distributed to anybody else on the system that needs to communicate securely with that system.

The important thing to remember is that any data encrypted with the public key right here can be decrypted only with the private key, which always remains on the source host. It never gets distributed. Because of this, we can freely give out the public key.

We can allow an attacker to capture the public key, because even if somebody were to encrypt data with the public key, the attacker would not be able to decrypt it because you cannot decrypt something that was encrypted with the public key, with the public key.

It can only be decrypted with the private key. In addition, we also use a key pair called the symmetric encryption, using secret key. That's where both hosts have the same key, and the same key can be used to both encrypt and decrypt.

First what happens is the SSH client creates a connection with a system where the SSH daemon is running. And it does this on IP port 22. sshd running over here. We run SSH at the command prompt of this system to start the client and connect to this server.

When we do, the SSH server sends its public key to the SSH client system. The SSH server uses a host key pair to store its private and public keys. These are shown right here.

We have /etc/ssh/ssh_host_key. That's the private key. And /etc/ssh/ssh_host_key.pub. That's the public key. This is the one that gets sent over to the client system. The client system receives the public key from the SSH server, and it will check to see if it already has a copy of that key.

The SSH client stores keys from other systems-- public keys, I should say-- from other systems in the file shown here: /etc/ssh/ssh_known_host. These are keys that can be used by anyone on the system. Individual users can also store these keys in their /home directory, in the /.ssh hidden directory, in a file named known_host.

If the client looks in either of these files and finds that it does not have a copy of this public key that's being sent from the server, it will prompt the user to say, "Is it okay if I add it?" And most likely, you'll say, "Yeah, go ahead and do this."

Once done, the client over here now trusts the server over here. And so it will generate a 256-bit secret key.

Remember that with the secret key, the keys are the same on both the sending and receiving systems. The data that's encrypted with that key can also be decrypted with that same key.

In order to get this key from this system over to this system, we have to protect it, because if this attacker here in the middle were to get ahold of that secret key, he or she would be able to decrypt our communications between the hosts. Here's the cool thing that SSH does, and it's really quite clever.

What it does is use the public key that we got down here from the server to encrypt this key right here that's generated on the SSH client. And we go ahead and send that key from the client to the server running the sshd daemon in encrypted format, encrypted with the public key.

Because remember, that transmission being encrypted with the public key can be decrypted only with this private key over here. This guy in the middle does not have that private key.

By doing this, we end up with the same secret key on both systems. Now we can use that key to encrypt and decrypt data being transmitted between these two hosts. Once this is done, the user over here running the client can authenticate to the remote server, and the username and the passwords that are being sent between this system and this system are encrypted with this secret key.

The attacker can sniff that transmission if they want, but they won't be able to read it because it will be obscured. It will be mangled. It will just appear as gibberish to them.

Encrypted Communications (SSHv2) 9:00-11:02

I do need to point out, before we go any further, that the process we just looked at works for SSH1. In SSH2 and later, things are a little bit different. First of all, the host key files used on the server are different, because we can actually have two different sets of private and public keys.

We can have an RSA key pair and we can have a DSA key pair. The key pair used will depend upon which encryption mechanism--either RSA or DSA--was used to initially create them.

In addition, with SSH1, we actually transmitted the secret key in encrypted form between the hosts. In SSH2, we actually don't do that. Instead, what we use is something called a Diffie-Hellman key agreement.

This Diffie-Hellman key agreement is used to negotiate a secret key on both ends. Because these two systems are using Diffie-Hellman key agreement, they are actually able both to come up with the same key.

The key itself is never actually sent over the network medium, as it was with SSH1, which increases the overall security of the system. Because there is always the risk, although remote, there's always the risk that maybe this guy over here in the middle was somehow able to get the private key off of this system.

If that were to happen, rare though it may be, if that were to be able to happen, then the attacker in the middle here could capture the packets and be able to decrypt the secret key that was encrypted with the public key on the client as it's being sent back to the server, be able to extract the secret key from it, and then be able to hack into the session, and be able to decrypt the packets as they are being sent.

With the SSH2 version of the protocol, that key is never sent. So this attacker in the middle never is able to grab ahold of it. There's never that risk of it happening.

After this, the secure channel is negotiated, and the user can authenticate to the SSH server from the SSH client, just like we saw with SSH1. The data can then be securely transferred between both systems.

Summary 11:03-11:14

That's it for this lesson. In this lesson, you were introduced to OpenSSH. We first discussed the weaknesses associated with older remote access technologies, like Telnet. Please don't ever use Telnet. Then we discussed how OpenSSH uses encryption to securely transfer data between systems.
