Roteiro

Aula: Servidor SSH (Secure Shell)

O protocolo SSH é utilizado como um servidor de acesso remoto. Utilizaremos o servidor OpenSSH na disciplina - https://www.openssh.com

Sobre o SSH (Livro: https://www.computer-networking.info):

3.7 Remote login

One of the initial motivations for building computer networks was to allow users to access remote computers over the networks. In the 1960s and 1970s, the mainframes and the emerging minicomputers were composed of a central unit and a set of terminals connected through serial lines or modems. The simplest protocol that was designed to access remote computers over a network is probably *telnet* **RFC 854**. *telnet* runs over TCP and a telnet server listens on port *23* by default. The TCP connection used by telnet is bidirectional, both the client and the server can send data over it. The data exchanged over such a connection is essentially the characters that are typed by the user on the client machine and the text output of the processes running on the server machine with a few exceptions (e.g. control characters, characters to control the terminal like VT-100, ...) . The default character set for telnet is the ASCII character set, but the extensions specified in **RFC 5198** support the utilization of Unicode characters.

From a security viewpoint, the main drawback of *telnet* is that all the information, including the usernames, passwords and commands, is sent in cleartext over a TCP connection. This implies that an eavesdropper could easily capture the passwords used by anyone on an unprotected network. Various software tools exist to automate this collection of information. For this reason, *telnet* is rarely used today to access remote computers. It is usually replaced by *ssh* or similar protocols.

3.7.1 The secure shell (ssh)

The secure shell protocol was designed in the mid 1990s by T. Ylonen to counter the eavesdropping attacks against *telnet* and similar protocols [Ylonen1996]. *ssh* became quickly popular and system administrators encouraged its usage. The original version of *ssh* was freely available. After a few years, his author created a company to distribute it commercially, but other programmers continued to develop an open-source version of *ssh* called OpenSSH. Over the years, *ssh* evolved and became a flexible applicable whose usage extends beyond remote login to support features such as file transfers, protocol tunneling, ... In this section, we only discuss the basic features of *ssh* and explain how it differs from *telnet*. Entire books have been written to describe *ssh* in details [BS2005]. An overview of the protocol appeared in [Stallings2009].

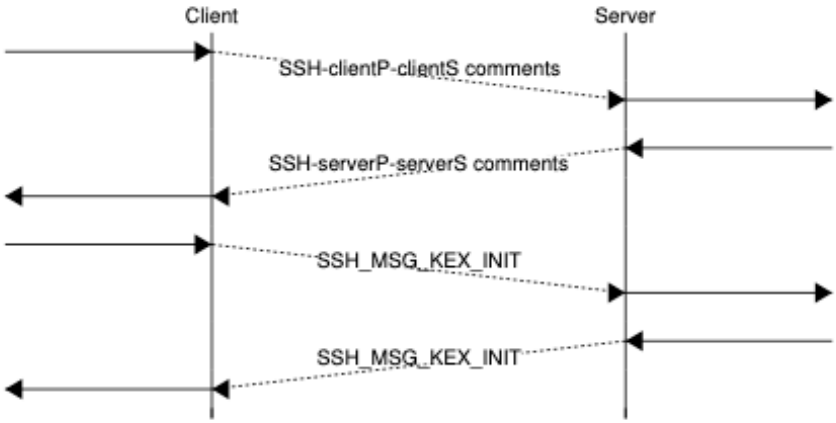
The *ssh* protocol runs directly above the TCP protocol. Once the TCP bytestream has been established, the client and the server exchange messages. The first message exchanged is an ASCII line that announces the version of the protocol and the version of the software implementation used by the client and the server. These two lines are useful when debugging interoperability problems and other issues.

The next message is the `SSH_MSG_KEX_INIT` message that is used to negotiate the cryptographic algorithms that will be used for the `ssh` session. It is very important for security protocols to include mechanisms that enable a negotiation of the cryptographic algorithms that are used. First, these algorithms provide different levels of security. Some algorithms might be considered totally secure and are recommended today while they could become deprecated a few years later after the publication of some attacks. Second, these algorithms provide different levels of performance and have different CPU and memory impacts.

In practice, an ssh implementation supports four types of cryptographic algorithms :

- key exchange
- encryption
- Message Authentication Code (MAC)
- compression

The IANA maintains a list of the cryptographic algorithms that can be used by ssh implementations. For each type of algorithm, the client provides an ordered list of the algorithms that it supports and agrees to use. The server compares the received list with its own list. The outcome of the negotiation is a set of four algorithms[1] that will be combined for this session.



This negotiation of the cryptographic algorithms allows the implementations to evolve when new algorithms are proposed. If a client is upgraded, it can announce a new algorithm as its preferred one even if the server is not yet upgraded.

Once the cryptographic algorithms have been negotiated, the key exchange algorithm is used to negotiate a secret key that will be shared by the client and the server. These key exchange algorithms include some variations over the basic algorithms. As an example, let us analyze how the Diffie Hellman key exchange algorithm is used within the ssh protocol. In this case, each host has both a private and a public key.

- the client generates the random number a and sends $A = g^a \mod p$ to the server
- the server generates the random number b. It then computes $B = g^b \mod p$, $K = B^a \mod p$ and signs with its private key $hash(V_{Client}||V_{Server}||KEX\_INIT_{Client}||KEX\_INIT_{Server}||Server_{pub}||A||B||K)$ where V_{Server} (resp. V_{Client}) is the initial messages sent by the client (resp. server), KEX_INIT_{Client} (resp. KEX_INIT_{Server}) is the key exchange message sent by the client (resp. server) and A, B and K are the messages of the Diffie Hellman key exchange
- the client can recompute $K = A^b \mod p$ and verify the signature provided by the server

This is a slightly modified authenticated Diffie Hellman key exchange with two interesting points. The first point is that when the server authenticates the key exchange it does not provide a certificate. This is because ssh assumes that

[1] For some of the algorithms, it is possible to negotiate the utilization of no algorithm. This happens frequently for the compression algorithm that is not always used. For this, both the client and the server must announce null in their ordered list of supported algorithms.

the client will store inside its cache the public key of the servers that it uses on a regular basis. This assumption is valid for a protocol like ssh because users typically use it to interact with a small number of servers, typically a few or a few tens. Storing this information does not require a lot of storage. In practice, most ssh clients will accept to connect to remote servers without knowing their public key before the connection. In this case, the client issues a warning to

the user who can decide to accept or reject the key. This warning can be associated with a fingerprint of the key, either as a sequence of letters or as an ASCII art which can be posted on the web or elsewhere[2] by the system administrator of the server. If a client connects to a server whose public key does not match the stored one, a stronger warning is issued because this could indicate a man-in-the-middle attack or that the remote server has been compromised. It can also indicate that the server has been upgraded and that a new key has been generated during this upgrade.

The second point is that the server authenticates not only the result of the Diffie Hellman exchange but also a hash of all the information sent and received during the exchange. This is important to prevent *downgrade attacks*. A *downgrade attack* is an attack where an active attacker modifies the messages sent by the communicating hosts (typically the client) to request the utilization of weaker encryption algorithms. Consider a client that supports two encryption schemes. The preferred one uses 128 bits secret keys and the second one is an old encryption scheme that uses 48 bits keys. This second algorithm is kept for backward compatibility with older implementations. If an attacker can remove the preferred algorithm from the list of encryption algorithms supported by the client, he can force the server to use a weaker encryption scheme that will be easier to break. Thanks to the hash that covers all the messages exchanged by the server, the downgrade attack cannot occur against `ssh`. Algorithm agility is a key requirement for security protocols that need to evolve when encryption algorithms are broken by researchers. This agility cannot be used without care and signing a hash of all the messages exchanged is a technique that is frequently used to prevent downgrade attacks.

Note: Single use keys

Thanks to the Diffie Hellman key exchange, the client and the servers share key K. A naive implementation would probably directly use this key for all the cryptographic algorithms that have been negotiated for this session. Like most security protocols, `ssh` does not directly use key K. Instead, it uses the negotiated hash function with different parameters[3] to allow the client and the servers to compute six keys from K :

- a key used by the client (resp. server) to encrypt the data that it sends
- a key used by the client (resp. server) to authenticate the data that it sends
- a key used by the client (resp. server) to initialize the negotiated encryption scheme (if required by this scheme)

It is common practice among designers of security protocols to never use the same key for different purposes. For example, allowing the client and the server to use the same key to encrypt data could enable an attacker to launch a replay attack by sending to the client data that it has itself encrypted.

At this point, all the messages sent over the TCP connection will be encrypted with the negotiated keys. The `ssh` protocol uses messages that are encoded according to the Binary Packet Protocol defined in **RFC 4253**. Each of these messages contains the following information :

- `length` : this is the length of the message in bytes, excluding the MAC and length fields
- `padding length` : this is the number of random bytes that have been added at the end of the message.
- `payload` : the data (after optional compression) passed by the user
- `padding` : random bytes added in each message (at least four) to ensure that the message length is a multiple of the block size used by the negotiated encryption algorithm
- `MAC` : this field is present if a Message Authentication Code has been negotiated for the session (in practice, using `ssh` without authentication is risky and this field should always be present). Note that to compute the MAC, an `ssh` implementation must maintain a message counter. This counter is incremented by

[2] For example, **RFC 4255** describes a DNS record that can be used to associate an `ssh` fingerprint to a DNS name.
[3] The exact algorithms used for the computation of these keys are defined in **RFC 4253**

one every time a message is sent and the MAC is computed with the negotiated authentication algorithm using the MAC key over the concatenation of the message counter and the cleartext message. The message counter is not transmitted, but the recipient can easily recover its value. The `MAC` is computed as $mac = MAC(key, sequence\_number||unencrypted\_message)$ where the key is the negotiated authentication key.

Note: Authenticating messages with HMAC

ssh is one example of a protocol that uses Message Authentication Codes (MAC) to authenticates the messages that are sent. A naive implementation of such a MAC would be to simply use a hash function like SHA-1. However, such a construction would not be safe from a security viewpoint. Internet protocols usually rely on the HMAC construction defined in **RFC 2104**. It works with any hash function (H) and a key (K). As an example, let us consider HMAC with the SHA-1 hash function. SHA-1 uses 20 bytes blocks and the block size will play an important role in the operation of HMAC. We first require the key to be as long as the block size. Since this key is the output of the key generation algorithm, this is one parameter of this algorithm.

HMAC uses two padding strings : *ipad* (resp. *opad*) which is a string containing 20 times byte `0x36` (resp. byte `0x5C`). The HMAC is then computed as $H[K \oplus opad, H(K \oplus ipad, data)]$ where $\oplus$ denotes the bitwise XOR operation. This computation has been shown to be stronger than the naive $H(K, data)$ against some types of cryptographic attacks.

Among the various features of the `ssh` protocol, it is interesting to mention how users are authenticated by the server. The `ssh` protocol supports the classical username/password authentication (but both the username and the password are transmitted over the secure encrypted channel). In addition, `ssh` supports two authentication mechanisms that rely on public keys. To use the first one, each user needs to generate his/her own public/private key pair and store the public key on the server. To be authenticated, the user needs to sign a message containing his/her public key by using his/her private key. The server can easily verify the validity of the signature since it already knows the user's public key. The second authentication scheme is designed for hosts that trust each other. Each host has a public/private key pair and stores the public keys of the other hosts that it trusts. This is typically used in environments such as university labs where each user could access any of the available computers. If Alice has logged on `computer1` and wants to execute a command on `computer2`, she can create an `ssh` session on this computer and type (again) her password. With the host-based authentication scheme, `computer1` signs a message with its private key to confirm that it has already authenticated Alice. `computer2` would then accept Alice's session without asking her credentials.

The `ssh` protocol includes other features that are beyond the scope of this book. Additional details may be found in [BS2005].

Prática:

- Saber se o serviço está instalado na maquina:
  - dpkg -l | grep ssh
- Buscar pacotes ssh:
  - apt-cache search ssh server
- Instalar o pacote ssh:
  - apt-get install openssh-server
- Controle do processo do ssh:
  - /etc/init.d/ssh <start|stop|restart>
  - ou service ssh <start|stop|restart|status>
  - ou start|stop ssh
  - ou systemctl start ssh

    - systemclt <disable|enable> ssh /** desabilitar a inicialização do serviço
    - sudo systemctl restart sshd.service
- Verificação de processos ativos:
    - ps aux | grep ssh
- Cliente:
    - Acesso remoto: ssh -p <porta> <user>@<ip-server>
    - Windows: Putty
        - <path to putty>/putty.exe -ssh <user>@<ip> port
- Arquivo de configuração do openssh-server:
    - etc/ssh/sshd_config
- Buscando pela porta relacionada ao SSH
    - netstat -anp | grep ssh
- Acesso a aplicações gráficas
    - ssh -X <user>@<ip>
- Exemplo de utilização com SCP:
    - scp faz cópia de arquivos utilizando o ssh
    - enviar p servidor: scp -P <porta> <arquivo> <user>@<ip>:/<pasta>
    - copiar do servidor: scp -P <porta> <user>@<ip>:/<pasta/arquivo> /home (.)
    - scp -r $HOME/diretorio <user>@<ip>:/<pasta>

Atividade SSH

- Desinstale o ssh da sua mv.
- Crie um usuario classmate<seu-id>, no grupo aulaRedes2, com a senha classmate<seu-id>.
- Reinstale o ssh.
- Forneça o usuário/senha de sua máquina para o colega ao lado e solicite um usuário/senha da a máquina do colega.
- Acesse remotamente via ssh a máquina do colega:
    - Crie a pasta /aulaRedes2
    - e subpastas, ex: aulaSSH, servico02, servico03
- Feche a conexão ssh.
- Crie em sua maquina local o arquivo helpSSH dentro de um dos diretórios criado pelo colega em sua máquina. Utilize um dos editores de texto estudados em sala.
    - Adicione no arquivo comandos utilizados na aula ssh e que gostaria de registrar.
- Envie o arquivo utilizando SCP para a pasta aulaSSH da maquina remota.
- Peça ao colega que acrescente novas informações ao arquivo helpSSH e que lhe avise quando terminar.
- Recupere a nova versão do arquivo via SCP.