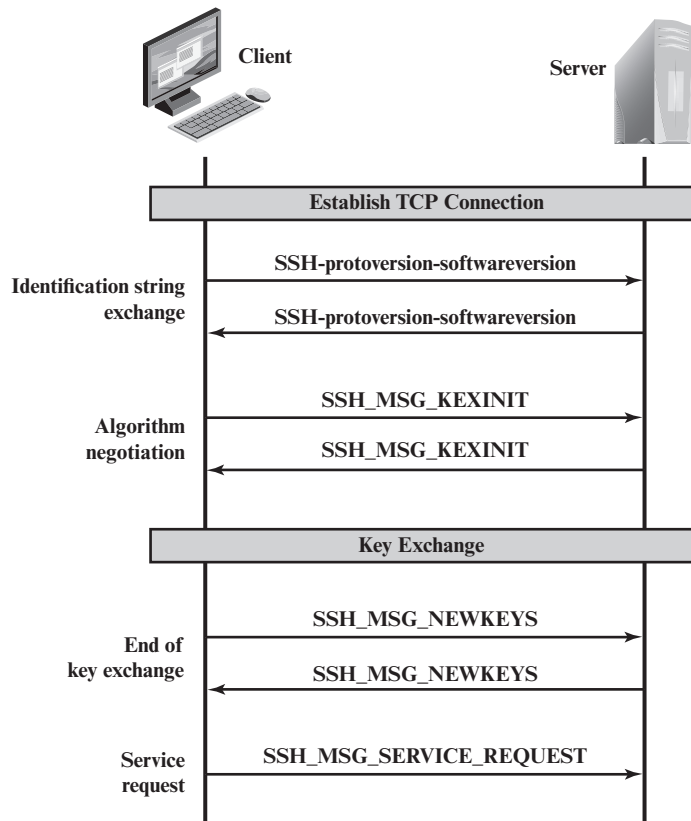


2. The host name-to-key association is certified by a trusted certification authority (CA). The client only knows the CA root key and can verify the validity of all host keys certified by accepted CAs. This alternative eases the maintenance problem, since ideally, only a single CA key needs to be securely stored on the client. On the other hand, each host key must be appropriately certified by a central authority before authorization is possible.

**PACKET EXCHANGE** Figure 6.9 illustrates the sequence of events in the SSH Transport Layer Protocol. First, the client establishes a TCP connection to the server. This is done via the TCP protocol and is not part of the Transport Layer Protocol. Once the connection is established, the client and server exchange data, referred to as packets, in the data field of a TCP segment. Each packet is in the following format (Figure 6.10).

- **Packet length:** Length of the packet in bytes, not including the packet length and MAC fields.
- **Padding length:** Length of the random padding field.
- **Payload:** Useful contents of the packet. Prior to algorithm negotiation, this field is uncompressed. If compression is negotiated, then in subsequent packets, this field is compressed.



**Figure 6.9** SSH Transport Layer Protocol Packet Exchanges

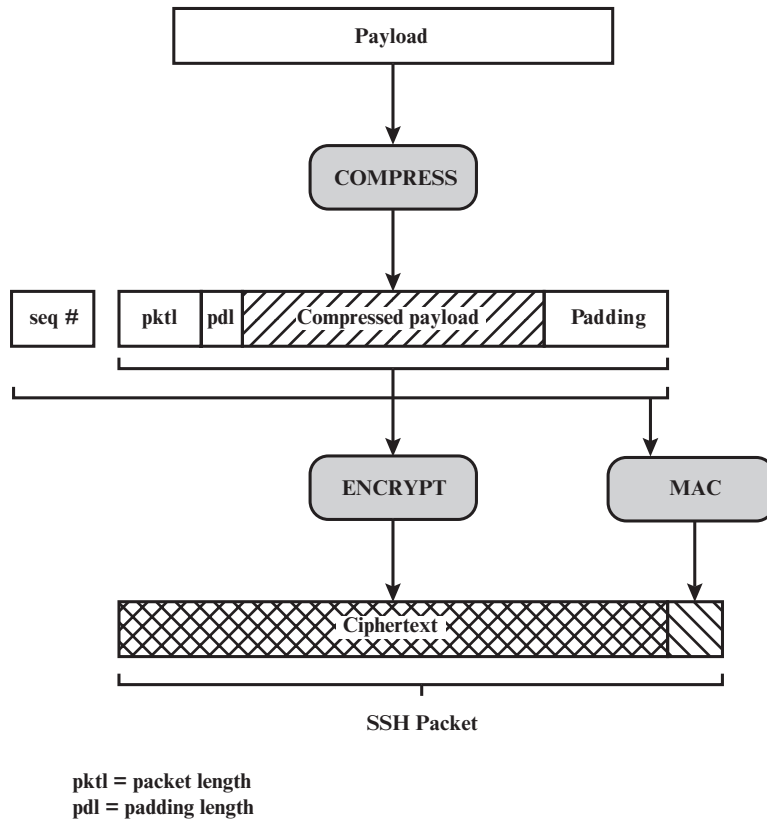


Figure 6.10 SSH Transport Layer Protocol Packet Formation

- **Random padding:** Once an encryption algorithm has been negotiated, this field is added. It contains random bytes of padding so that the total length of the packet (excluding the MAC field) is a multiple of the cipher block size, or 8 bytes for a stream cipher.
- **Message authentication code (MAC):** If message authentication has been negotiated, this field contains the MAC value. The MAC value is computed over the entire packet plus a sequence number, excluding the MAC field. The sequence number is an implicit 32-bit packet sequence that is initialized to zero for the first packet and incremented for every packet. The sequence number is not included in the packet sent over the TCP connection.

Once an encryption algorithm has been negotiated, the entire packet (excluding the MAC field) is encrypted after the MAC value is calculated.

The SSH Transport Layer packet exchange consists of a sequence of steps (Figure 6.9). The first step, the **identification string exchange**, begins with the client sending a packet with an identification string of the form:

```
SSH-protoversion-softwareversion SP comments CR LF
```

where SP, CR, and LF are space character, carriage return, and line feed, respectively. An example of a valid string is SSH-2.0-billsSSH\_3.6.3q3<CR><LF>. The server responds with its own identification string. These strings are used in the Diffie–Hellman key exchange.

Next comes **algorithm negotiation**. Each side sends an SSH\_MSG\_KEXINIT containing lists of supported algorithms in the order of preference to the sender. There is one list for each type of cryptographic algorithm. The algorithms include key exchange, encryption, MAC algorithm, and compression algorithm. Table 6.3 shows the allowable options for encryption, MAC, and compression. For each category, the algorithm chosen is the first algorithm on the client’s list that is also supported by the server.

The next step is **key exchange**. The specification allows for alternative methods of key exchange, but at present, only two versions of Diffie–Hellman key exchange are specified. Both versions are defined in RFC 2409 and require only one packet in each direction. The following steps are involved in the exchange. In this, C is the client; S is the server;  $p$  is a large safe prime;  $g$  is a generator for a subgroup of  $GF(p)$ ;  $q$  is the order of the subgroup;  $V_S$  is S’s identification string;  $V_C$  is

**Table 6.3** SSH Transport Layer Cryptographic Algorithms

Cipher		MAC algorithm	
3des-cbc*	Three-key 3DES in CBC mode	hmac-sha1*	HMAC-SHA1; digest length = key length = 20
blowfish-cbc	Blowfish in CBC mode	hmac-sha1-96**	First 96 bits of HMAC-SHA1; digest length = 12; key length = 20
twofish256-cbc	Twofish in CBC mode with a 256-bit key	hmac-md5	HMAC-MD5; digest length = key length = 16
twofish192-cbc	Twofish with a 192-bit key	hmac-md5-96	First 96 bits of HMAC-MD5; digest length = 12; key length = 16
twofish128-cbc	Twofish with a 128-bit key	Compression algorithm	
aes256-cbc	AES in CBC mode with a 256-bit key		
aes192-cbc	AES with a 192-bit key	none*	No compression
aes128-cbc**	AES with a 128-bit key	zlib	Defined in RFC 1950 and RFC 1951
Serpent256-cbc	Serpent in CBC mode with a 256-bit key		
Serpent192-cbc	Serpent with a 192-bit key		
Serpent128-cbc	Serpent with a 128-bit key		
arcfour	RC4 with a 128-bit key		
cast128-cbc	CAST-128 in CBC mode		

\* = Required

\*\* = Recommended

C's identification string;  $K_S$  is S's public host key;  $I_C$  is C's `SSH_MSG_KEXINIT` message and  $I_S$  is S's `SSH_MSG_KEXINIT` message that have been exchanged before this part begins. The values of  $p$ ,  $g$ , and  $q$  are known to both client and server as a result of the algorithm selection negotiation. The hash function `hash()` is also decided during algorithm negotiation.

1. C generates a random number  $x$  ( $1 < x < q$ ) and computes  $e = g^x \bmod p$ . C sends  $e$  to S.
2. S generates a random number  $y$  ( $0 < y < q$ ) and computes  $f = g^y \bmod p$ . S receives  $e$ . It computes  $K = e^y \bmod p$ ,  $H = \text{hash}(V_C \| V_S \| I_C \| I_S \| K_S \| e \| f \| K)$ , and signature  $s$  on  $H$  with its private host key. S sends  $(K_S \| f \| s)$  to C. The signing operation may involve a second hashing operation.
3. C verifies that  $K_S$  really is the host key for S (e.g., using certificates or a local database). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes  $K = f^x \bmod p$ ,  $H = \text{hash}(V_C \| V_S \| I_C \| I_S \| K_S \| e \| f \| K)$ , and verifies the signature  $s$  on  $H$ .

As a result of these steps, the two sides now share a master key  $K$ . In addition, the server has been authenticated to the client, because the server has used its private key to sign its half of the Diffie–Hellman exchange. Finally, the hash value  $H$  serves as a session identifier for this connection. Once computed, the session identifier is not changed, even if the key exchange is performed again for this connection to obtain fresh keys.

The **end of key exchange** is signaled by the exchange of `SSH_MSG_NEWKEYS` packets. At this point, both sides may start using the keys generated from  $K$ , as discussed subsequently.

The final step is **service request**. The client sends an `SSH_MSG_SERVICE_REQUEST` packet to request either the User Authentication or the Connection Protocol. Subsequent to this, all data is exchanged as the payload of an SSH Transport Layer packet, protected by encryption and MAC.

**KEY GENERATION** The keys used for encryption and MAC (and any needed IVs) are generated from the shared secret key  $K$ , the hash value from the key exchange  $H$ , and the session identifier, which is equal to  $H$  unless there has been a subsequent key exchange after the initial key exchange. The values are computed as follows.

- Initial IV client to server:  $\text{HASH}(K \| H \| \text{"A"} \| \text{session\_id})$
- Initial IV server to client:  $\text{HASH}(K \| H \| \text{"B"} \| \text{session\_id})$
- Encryption key client to server:  $\text{HASH}(K \| H \| \text{"C"} \| \text{session\_id})$
- Encryption key server to client:  $\text{HASH}(K \| H \| \text{"D"} \| \text{session\_id})$
- Integrity key client to server:  $\text{HASH}(K \| H \| \text{"E"} \| \text{session\_id})$
- Integrity key server to client:  $\text{HASH}(K \| H \| \text{"F"} \| \text{session\_id})$

where `HASH()` is the hash function determined during algorithm negotiation.

## User Authentication Protocol

The User Authentication Protocol provides the means by which the client is authenticated to the server.

*MESSAGE TYPES AND FORMATS* Three types of messages are always used in the User Authentication Protocol. Authentication requests from the client have the format:

byte	SSH_MSG_USERAUTH_REQUEST (50)
string	user name
string	service name
string	method name
...	method specific fields

where user name is the authorization identity the client is claiming, service name is the facility to which the client is requesting access (typically the SSH Connection Protocol), and method name is the authentication method being used in this request. The first byte has decimal value 50, which is interpreted as SSH\_MSG\_USERAUTH\_REQUEST.

If the server either (1) rejects the authentication request or (2) accepts the request but requires one or more additional authentication methods, the server sends a message with the format:

byte	SSH_MSG_USERAUTH_FAILURE (51)
name-list	authentications that can continue
boolean	partial success

where the name-list is a list of methods that may productively continue the dialog. If the server accepts authentication, it sends a single byte message: SSH\_MSG\_USERAUTH\_SUCCESS (52).

*MESSAGE EXCHANGE* The message exchange involves the following steps.

1. The client sends a SSH\_MSG\_USERAUTH\_REQUEST with a requested method of none.
2. The server checks to determine if the user name is valid. If not, the server returns SSH\_MSG\_USERAUTH\_FAILURE with the partial success value of false. If the user name is valid, the server proceeds to step 3.
3. The server returns SSH\_MSG\_USERAUTH\_FAILURE with a list of one or more authentication methods to be used.
4. The client selects one of the acceptable authentication methods and sends a SSH\_MSG\_USERAUTH\_REQUEST with that method name and the required method-specific fields. At this point, there may be a sequence of exchanges to perform the method.

5. If the authentication succeeds and more authentication methods are required, the server proceeds to step 3, using a partial success value of true. If the authentication fails, the server proceeds to step 3, using a partial success value of false.
6. When all required authentication methods succeed, the server sends a `SSH_MSG_USERAUTH_SUCCESS` message, and the Authentication Protocol is over.

**AUTHENTICATION METHODS** The server may require one or more of the following authentication methods.

- **publickey:** The details of this method depend on the public-key algorithm chosen. In essence, the client sends a message to the server that contains the client's public key, with the message signed by the client's private key. When the server receives this message, it checks whether the supplied key is acceptable for authentication and, if so, it checks whether the signature is correct.
- **password:** The client sends a message containing a plaintext password, which is protected by encryption by the Transport Layer Protocol.
- **hostbased:** Authentication is performed on the client's host rather than the client itself. Thus, a host that supports multiple clients would provide authentication for all its clients. This method works by having the client send a signature created with the private key of the client host. Thus, rather than directly verifying the user's identity, the SSH server verifies the identity of the client host—and then believes the host when it says the user has already authenticated on the client side.

## Connection Protocol

The SSH Connection Protocol runs on top of the SSH Transport Layer Protocol and assumes that a secure authentication connection is in use.<sup>2</sup> That secure authentication connection, referred to as a **tunnel**, is used by the Connection Protocol to multiplex a number of logical channels.

**CHANNEL MECHANISM** All types of communication using SSH, such as a terminal session, are supported using separate channels. Either side may open a channel. For each channel, each side associates a unique channel number, which need not be the same on both ends. Channels are flow controlled using a window mechanism. No data may be sent to a channel until a message is received to indicate that window space is available.

---

<sup>2</sup>RFC 4254, *The Secure Shell (SSH) Connection Protocol*, states that the Connection Protocol runs on top of the Transport Layer Protocol and the User Authentication Protocol. RFC 4251, *SSH Protocol Architecture*, states that the Connection Protocol runs over the User Authentication Protocol. In fact, the Connection Protocol runs over the Transport Layer Protocol, but assumes that the User Authentication Protocol has been previously invoked.

The life of a channel progresses through three stages: opening a channel, data transfer, and closing a channel.

When either side wishes to **open a new channel**, it allocates a local number for the channel and then sends a message of the form:

```

byte      SSH_MSG_CHANNEL_OPEN
string    channel type
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
....      channel type specific data follows

```

where uint32 means unsigned 32-bit integer. The channel type identifies the application for this channel, as described subsequently. The sender channel is the local channel number. The initial window size specifies how many bytes of channel data can be sent to the sender of this message without adjusting the window. The maximum packet size specifies the maximum size of an individual data packet that can be sent to the sender. For example, one might want to use smaller packets for interactive connections to get better interactive response on slow links.

If the remote side is able to open the channel, it returns a `SSH_MSG_CHANNEL_OPEN_CONFIRMATION` message, which includes the sender channel number, the recipient channel number, and window and packet size values for incoming traffic. Otherwise, the remote side returns a `SSH_MSG_CHANNEL_OPEN_FAILURE` message with a reason code indicating the reason for failure.

Once a channel is open, **data transfer** is performed using a `SSH_MSG_CHANNEL_DATA` message, which includes the recipient channel number and a block of data. These messages, in both directions, may continue as long as the channel is open.

When either side wishes to **close a channel**, it sends a `SSH_MSG_CHANNEL_CLOSE` message, which includes the recipient channel number.

Figure 6.11 provides an example of Connection Protocol Message Exchange.

**CHANNEL TYPES** Four channel types are recognized in the SSH Connection Protocol specification.

- **session:** The remote execution of a program. The program may be a shell, an application such as file transfer or e-mail, a system command, or some built-in subsystem. Once a session channel is opened, subsequent requests are used to start the remote program.
- **x11:** This refers to the X Window System, a computer software system and network protocol that provides a graphical user interface (GUI) for networked computers. X allows applications to run on a network server but to be displayed on a desktop machine.

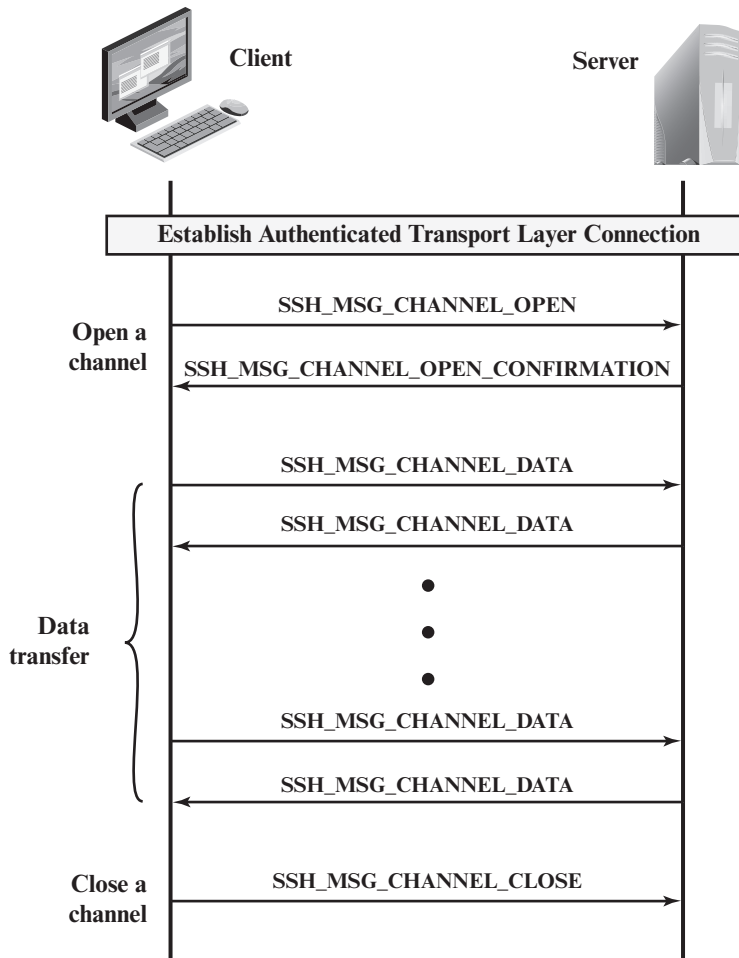


Figure 6.11 Example of SSH Connection Protocol Message Exchange

- **forwarded-tcpip:** This is remote port forwarding, as explained in the next subsection.
- **direct-tcpip:** This is local port forwarding, as explained in the next subsection.

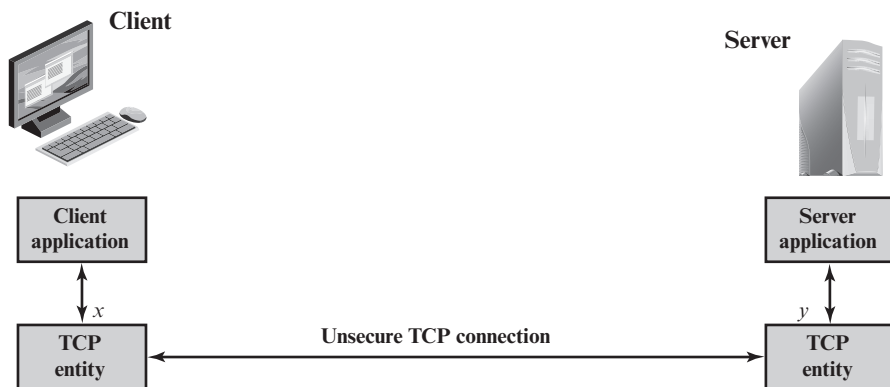
**PORT FORWARDING** One of the most useful features of SSH is port forwarding. In essence, port forwarding provides the ability to convert any insecure TCP connection into a secure SSH connection. This is also referred to as SSH tunneling. We need to know what a port is in this context. A **port** is an identifier of a user of TCP. So, any application that runs on top of TCP has a port number. Incoming TCP traffic is delivered to the appropriate application on the basis of the port number. An application may employ multiple port numbers. For example, for the Simple Mail Transfer Protocol (SMTP), the server side generally listens on port 25, so an



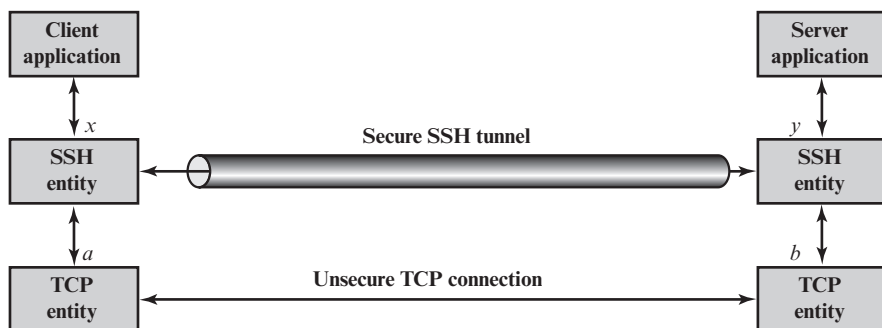
incoming SMTP request uses TCP and addresses the data to destination port 25. TCP recognizes that this is the SMTP server address and routes the data to the SMTP server application.

Figure 6.12 illustrates the basic concept behind port forwarding. We have a client application that is identified by port number  $x$  and a server application identified by port number  $y$ . At some point, the client application invokes the local TCP entity and requests a connection to the remote server on port  $y$ . The local TCP entity negotiates a TCP connection with the remote TCP entity, such that the connection links local port  $x$  to remote port  $y$ .

To secure this connection, SSH is configured so that the SSH Transport Layer Protocol establishes a TCP connection between the SSH client and server entities, with TCP port numbers  $a$  and  $b$ , respectively. A secure SSH tunnel is established



(a) Connection via TCP



(b) Connection via SSH tunnel

Figure 6.12 SSH Transport Layer Packet Exchanges

over this TCP connection. Traffic from the client at port  $x$  is redirected to the local SSH entity and travels through the tunnel where the remote SSH entity delivers the data to the server application on port  $y$ . Traffic in the other direction is similarly redirected.

SSH supports two types of port forwarding: local forwarding and remote forwarding. **Local forwarding** allows the client to set up a “hijacker” process. This will intercept selected application-level traffic and redirect it from an unsecured TCP connection to a secure SSH tunnel. SSH is configured to listen on selected ports. SSH grabs all traffic using a selected port and sends it through an SSH tunnel. On the other end, the SSH server sends the incoming traffic to the destination port dictated by the client application.

The following example should help clarify local forwarding. Suppose you have an e-mail client on your desktop and use it to get e-mail from your mail server via the Post Office Protocol (POP). The assigned port number for POP3 is port 110. We can secure this traffic in the following way:

1. The SSH client sets up a connection to the remote server.
2. Select an unused local port number, say 9999, and configure SSH to accept traffic from this port destined for port 110 on the server.
3. The SSH client informs the SSH server to create a connection to the destination, in this case mailserver port 110.
4. The client takes any bits sent to local port 9999 and sends them to the server inside the encrypted SSH session. The SSH server decrypts the incoming bits and sends the plaintext to port 110.
5. In the other direction, the SSH server takes any bits received on port 110 and sends them inside the SSH session back to the client, who decrypts and sends them to the process connected to port 9999.

With **remote forwarding**, the user’s SSH client acts on the server’s behalf. The client receives traffic with a given destination port number, places the traffic on the correct port and sends it to the destination the user chooses. A typical example of remote forwarding is the following. You wish to access a server at work from your home computer. Because the work server is behind a firewall, it will not accept an SSH request from your home computer. However, from work you can set up an SSH tunnel using remote forwarding. This involves the following steps.

1. From the work computer, set up an SSH connection to your home computer. The firewall will allow this, because it is a protected outgoing connection.
2. Configure the SSH server to listen on a local port, say 22, and to deliver data across the SSH connection addressed to remote port, say 2222.
3. You can now go to your home computer, and configure SSH to accept traffic on port 2222.
4. You now have an SSH tunnel that can be used for remote logon to the work server.