Phase 2. Server Authentication and Key Exchange The server begins this phase by sending its certificate if it needs to be authenticated; the message contains one or a chain of X.509 certificates. The **certificate message** is required for any agreed-on key exchange method except anonymous Diffie–Hellman. Note that if fixed Diffie–Hellman is used, this certificate message functions as the server's key exchange message because it contains the server's public Diffie–Hellman parameters.

Next, a **server_key_exchange message** may be sent if it is required. It is not required in two instances: (1) The server has sent a certificate with fixed Diffie–Hellman parameters; or (2) RSA key exchange is to be used. The server_key_exchange message is needed for the following:

- **Anonymous Diffie–Hellman:** The message content consists of the two global Diffie–Hellman values (a prime number and a primitive root of that number) plus the server's public Diffie–Hellman key (see Figure 10.1).
- **Ephemeral Diffie–Hellman:** The message content includes the three Diffie–Hellman parameters provided for anonymous Diffie–Hellman plus a signature of those parameters.
- **RSA key exchange (in which the server is using RSA but has a signature-only RSA key):** Accordingly, the client cannot simply send a secret key encrypted with the server's public key. Instead, the server must create a temporary RSA public/private key pair and use the server_key_exchange message to send the public key. The message content includes the two parameters of the temporary RSA public key (exponent and modulus; see Figure 9.5) plus a signature of those parameters.

Some further details about the signatures are warranted. As usual, a signature is created by taking the hash of a message and encrypting it with the sender's private key. In this case, the hash is defined as

$$\text{hash}(\text{ClientHello.random} \parallel \text{ServerHello.random} \parallel \text{ServerParams})$$

So the hash covers not only the Diffie–Hellman or RSA parameters but also the two nonces from the initial hello messages. This ensures against replay attacks and misrepresentation. In the case of a DSS signature, the hash is performed using the SHA-1 algorithm. In the case of an RSA signature, both an MD5 and an SHA-1 hash are calculated, and the concatenation of the two hashes (36 bytes) is encrypted with the server's private key.

Next, a nonanonymous server (server not using anonymous Diffie–Hellman) can request a certificate from the client. The **certificate_request message** includes two parameters: certificate_type and certificate_authorities. The certificate type indicates the public-key algorithm and its use:

- RSA, signature only
- DSS, signature only
- RSA for fixed Diffie–Hellman; in this case the signature is used only for authentication, by sending a certificate signed with RSA
- DSS for fixed Diffie–Hellman; again, used only for authentication

The second parameter in the certificate_request message is a list of the distinguished names of acceptable certificate authorities.

The final message in phase 2, and one that is always required, is the **server_done message**, which is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response. This message has no parameters.

Phase 3. Client Authentication and Key Exchange Upon receipt of the `server_done` message, the client should verify that the server provided a valid certificate (if required) and check that the `server_hello` parameters are acceptable. If all is satisfactory, the client sends one or more messages back to the server.

If the server has requested a certificate, the client begins this phase by sending a **certificate message**. If no suitable certificate is available, the client sends a `no_certificate` alert instead.

Next is the **client_key_exchange message**, which must be sent in this phase. The content of the message depends on the type of key exchange, as follows:

- **RSA:** The client generates a 48-byte *pre-master secret* and encrypts with the public key from the server's certificate or temporary RSA key from a server_key_exchange message. Its use to compute a *master secret* is explained later.
- **Ephemeral or Anonymous Diffie–Hellman:** The client's public Diffie–Hellman parameters are sent.
- **Fixed Diffie–Hellman:** The client's public Diffie–Hellman parameters were sent in a certificate message, so the content of this message is null.

Finally, in this phase, the client may send a **certificate_verify message** to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability (i.e., all certificates except those containing fixed Diffie–Hellman parameters). This message signs a hash code based on the preceding messages, defined as

```
CertificateVerify.signature.md5_hash
    MD5(handshake_messages);
Certificate.signature.sha_hash
    SHA(handshake_messages);
```

where `handshake_messages` refers to all Handshake Protocol messages sent or received starting at `client_hello` but not including this message. If the user's private key is DSS, then it is used to encrypt the SHA-1 hash. If the user's private key is RSA, it is used to encrypt the concatenation of the MD5 and SHA-1 hashes. In either case, the purpose is to verify the client's ownership of the private key for the client certificate. Even if someone is misusing the client's certificate, he or she would be unable to send this message.

Phase 4. Finish Phase 4 completes the setting up of a secure connection. The client sends a **change_cipher_spec message** and copies the pending CipherSpec into the current CipherSpec. Note that this message is not considered part of the Handshake Protocol but is sent using the Change Cipher Spec Protocol. The client then immediately sends the **finished message** under the new algorithms, keys, and secrets.

The finished message verifies that the key exchange and authentication processes were successful. The content of the finished message is:

PRF(master_secret, finished_label, MD5(handshake_messages) || SHA-1 (handshake_messages))

where `finished_label` is the string "client finished" for the client and "server finished" for the server.

In response to these two messages, the server sends its own `change_cipher_spec` message, transfers the pending to the current CipherSpec, and sends its finished message. At this point, the handshake is complete and the client and server may begin to exchange application-layer data.

Cryptographic Computations

Two further items are of interest: (1) the creation of a shared master secret by means of the key exchange; and (2) the generation of cryptographic parameters from the master secret.

Master Secret Creation The shared master secret is a one-time 48-byte value (384 bits) generated for this session by means of secure key exchange. The creation is in two stages. First, a `pre_master_secret` is exchanged. Second, the `master_secret` is calculated by both parties. For `pre_master_secret` exchange, there are two possibilities.

- **RSA:** A 48-byte pre_master_secret is generated by the client, encrypted with the server's public RSA key, and sent to the server. The server decrypts the ciphertext using its private key to recover the pre_master_secret.
- **Diffie–Hellman:** Both client and server generate a Diffie–Hellman public key. After these are exchanged, each side performs the Diffie–Hellman calculation to create the shared pre_master_secret.

  Both sides now compute the `master_secret` as

  master_secret =
  PRF(pre_master_secret, "master secret", ClientHello.random || ServerHello .random)

where `ClientHello.random` and `ServerHello.random` are the two nonce values exchanged in the initial hello messages.

The algorithm is performed until 48 bytes of pseudorandom output are produced. The calculation of the key block material (MAC secret keys, session encryption keys, and IVs) is defined as

key_block =
PRF(SecurityParameters.master_secret, "key expansion",
SecurityParameters.server_random || SecurityParameters.client_random)

until enough output has been generated.

Generation of Cryptographic Parameters CipherSpecs require a client write MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV, and a server write IV, which are generated from the master secret in that order. These parameters are generated from the master secret by hashing the master secret into a sequence of secure bytes of sufficient length for all needed parameters.

The generation of the key material from the master secret uses the same format for generation of the master secret from the pre-master secret as

$$\begin{aligned}\text{key\_block} = \text{MD5}(&\text{master\_secret} \parallel \text{SHA}(\text{`A'} \parallel \text{master\_secret} \parallel \\ &\text{ServerHello.random} \parallel \text{ClientHello.random})) \parallel \\ \text{MD5}(&\text{master\_secret} \parallel \text{SHA}(\text{`BB'} \parallel \text{master\_secret} \parallel \\ &\text{ServerHello.random} \parallel \text{ClientHello.random})) \parallel \\ \text{MD5}(&\text{master\_secret} \parallel \text{SHA}(\text{`CCC'} \parallel \text{master\_secret} \parallel \\ &\text{ServerHello.random} \parallel \text{ClientHello.random})) \parallel \ldots\end{aligned}$$

until enough output has been generated. The result of this algorithmic structure is a pseudorandom function. We can view the `master_secret` as the pseudorandom seed value to the function. The client and server random numbers can be viewed as salt values to complicate cryptanalysis (see Chapter 11 for a discussion of the use of salt values).

Pseudorandom Function TLS makes use of a pseudorandom function referred to as PRF to expand secrets into blocks of data for purposes of key generation or validation. The objective is to make use of a relatively small, shared secret value but to generate longer blocks of data in a way that is secure from the kinds of attacks made on hash functions and MACs. The PRF is based on the data expansion function (Figure 6.7) given as

$$\begin{aligned}\text{P\_hash}(\text{secret}, \text{seed}) = &\text{HMAC\_hash}(\text{secret}, \text{A}(1) \parallel \text{seed}) \parallel \\ &\text{HMAC\_hash}(\text{secret}, \text{A}(2) \parallel \text{seed}) \parallel \\ &\text{HMAC\_hash}(\text{secret}, \text{A}(3) \parallel \text{seed}) \parallel\end{aligned}$$

where A() is defined as

$$\begin{aligned}\text{A}(0) &= \text{seed} \\ \text{A}(i) &= \text{HMAC\_hash}(\text{secret}, A(i-1))\end{aligned}$$

The data expansion function makes use of the HMAC algorithm with either MD5 or SHA-1 as the underlying hash function. As can be seen, `P_hash` can be iterated as many times as necessary to produce the required quantity of data. For example, if `P_SHA256` was used to generate 80 bytes of data, it would have to be iterated three times (through A(3)), producing 96 bytes of data of which the last 16 would be discarded. In this case, `P_MD5` would have to be iterated four times, producing exactly 64 bytes of data. Note that each iteration involves two executions of HMAC, each of which in turn involves two executions of the underlying hash algorithm.

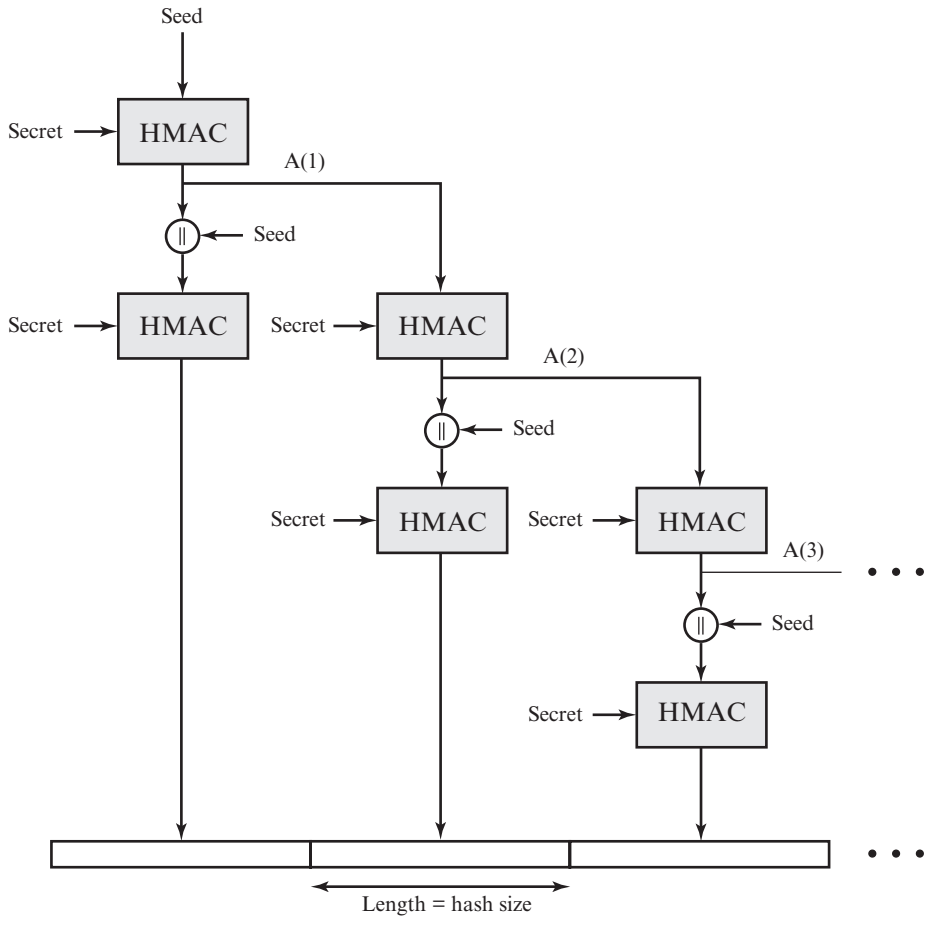

Figure 6.7 TLS Function `P_hash(secret, seed)`

To make PRF as secure as possible, it uses two hash algorithms in a way that should guarantee its security if either algorithm remains secure. PRF is defined as

$$\text{PRF(secret, label, seed)} = \text{P\_<hash>(secret, label} \| \text{seed)}$$

PRF takes as input a secret value, an identifying label, and a seed value and produces an output of arbitrary length.

Heartbeat Protocol

In the context of computer networks, a heartbeat is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a system. A heartbeat protocol is typically used to monitor the availability of a protocol entity. In the specific case of TLS, a Heartbeat protocol was defined in 2012 in RFC 6250 (*Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*).

The Heartbeat protocol runs on top of the TLS Record Protocol and consists of two message types: `heartbeat_request` and `heartbeat_response`. The use of the Heartbeat protocol is established during Phase 1 of the Handshake protocol (Figure 6.6). Each peer indicates whether it supports heartbeats. If heartbeats are supported, the peer indicates whether it is willing to receive `heartbeat_request` messages and respond with `heartbeat_response` messages or only willing to send `heartbeat_request` messages.

A `heartbeat_request` message can be sent at any time. Whenever a request message is received, it should be answered promptly with a corresponding `heartbeat_response` message. The `heartbeat_request` message includes payload length, payload, and padding fields. The payload is a random content between 16 bytes and 64 Kbytes in length. The corresponding `heartbeat_response` message must include an exact copy of the received payload. The padding is also random content. The padding enables the sender to perform a path MTU (maximum transfer unit) discovery operation, by sending requests with increasing padding until there is no answer anymore, because one of the hosts on the path cannot handle the message.

The heartbeat serves two purposes. First, it assures the sender that the recipient is still alive, even though there may not have been any activity over the underlying TCP connection for a while. Second, the heartbeat generates activity across the connection during idle periods, which avoids closure by a firewall that does not tolerate idle connections.

The requirement for the exchange of a payload was designed into the Heartbeat protocol to support its use in a connectionless version of TLS known as Datagram Transport Layer Security (DTLS). Because a connectionless service is subject to packet loss, the payload enables the requestor to match response messages to request messages. For simplicity, the same version of the Heartbeat protocol is used with both TLS and DTLS. Thus, the payload is required for both TLS and DTLS.

SSL/TLS ATTACKS

Since the first introduction of SSL in 1994, and the subsequent standardization of TLS, numerous attacks have been devised against these protocols. The appearance of each attack has necessitated changes in the protocol, the encryption tools used, or some aspect of the implementation of SSL and TLS to counter these threats.

Attack Categories We can group the attacks into four general categories:

- **Attacks on the handshake protocol:** As early as 1998, an approach to compromising the handshake protocol based on exploiting the formatting and implementation of the RSA encryption scheme was presented [BLEI98]. As countermeasures were implemented the attack was refined and adjusted to not only thwart the countermeasures but also speed up the attack [e.g., BARD12].
- **Attacks on the record and application data protocols:** A number of vulnerabilities have been discovered in these protocols, leading to patches to counter the new threats. As a recent example, in 2011, researchers Thai Duong and Juliano Rizzo demonstrated a proof of concept called BEAST (Browser Exploit Against SSL/TLS) that turned what had been considered only a theoretical vulnerability

into a practical attack [GOOD11]. BEAST leverages a type of cryptographic attack called a chosen-plaintext attack. The attacker mounts the attack by choosing a guess for the plaintext that is associated with a known ciphertext. The researchers developed a practical algorithm for launching successful attacks. Subsequent patches were able to thwart this attack. The authors of the BEAST attack are also the creators of the 2012 CRIME (Compression Ratio Info-leak Made Easy) attack, which can allow an attacker to recover the content of web cookies when data compression is used along with TLS [GOOD12]. When used to recover the content of secret authentication cookies, it allows an attacker to perform session hijacking on an authenticated web session.

- **Attacks on the PKI:** Checking the validity of X.509 certificates is an activity subject to a variety of attacks, both in the context of SSL/TLS and elsewhere. For example, [GEOR12] demonstrated that commonly used libraries for SSL/TLS suffer from vulnerable certificate validation implementations. The authors revealed weaknesses in the source code of OpenSSL, GnuTLS, JSSE, ApacheHttpClient, Weberknecht, cURL, PHP, Python and applications built upon or with these products.
- **Other attacks:** [MEYE13] lists a number of attacks that do not fit into any of the preceding categories. One example is an attack announced in 2011 by the German hacker group The Hackers Choice, which is a DoS attack [KUMA11]. The attack creates a heavy processing load on a server by overwhelming the target with SSL/TLS handshake requests. Boosting system load is done by establishing new connections or using renegotiation. Assuming that the majority of computation during a handshake is done by the server, the attack creates more system load on the server than on the source device, leading to a DoS. The server is forced to continuously recompute random numbers and keys.

The history of attacks and countermeasures for SSL/TLS is representative of that for other Internet-based protocols. A "perfect" protocol and a "perfect" implementation strategy are never achieved. A constant back-and-forth between threats and countermeasures determines the evolution of Internet-based protocols.

TLSv1.3

In 2014, the IETF TLS working group began work on a version 1.3 of TLS. The primary aim is to improve the security of TLS. As of this writing, TLSv1.3 is still in a draft stage, but the final standard is likely to be very close to the current draft. Among the significant changes from version 1.2 are the following:

- TLSv1.3 removes support for a number of options and functions. Removing code that implements functions no longer needed reduces the chances of potentially dangerous coding errors and reduces the attack surface. The deleted items include:
  - –Compression
  - –Ciphers that do not offer authenticated encryption
  - –Static RSA and DH key exchange
  - –32-bit timestamp as part of the Random parameter in the client_hello message

–Renegotiation
–Change Cipher Spec Protocol
–RC4
–Use of MD5 and SHA-224 hashes with signatures

- TLSv1.3 uses Diffie–Hellman or Elliptic Curve Diffie–Hellman for key exchange and does not permit RSA. The danger with RSA is that if the private key is compromised, all handshakes using these cipher suites will be compromised. With DH or ECDH, a new key is negotiated for each handshake.
- TLSv1.3 allows for a "1 round trip time" handshake by changing the order of message sent with establishing a secure connection. The client sends a Client Key Exchange message containing its cryptographic parameters for key establishment before a cipher suite has been negotiated. This enables a server to calculate keys for encryption and authentication before sending its first response. Reducing the number of packets sent during this handshake phase speeds up the process and reduces the attack surface.

These changes should improve the efficiency and security of TLS.

6.3 HTTPS

HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server. The HTTPS capability is built into all modern Web browsers. Its use depends on the Web server supporting HTTPS communication. For example, some search engines do not support HTTPS.

The principal difference seen by a user of a Web browser is that URL (uniform resource locator) addresses begin with https:// rather than http://. A normal HTTP connection uses port 80. If HTTPS is specified, port 443 is used, which invokes SSL.

When HTTPS is used, the following elements of the communication are encrypted:

- URL of the requested document
- Contents of the document
- Contents of browser forms (filled in by browser user)
- Cookies sent from browser to server and from server to browser
- Contents of HTTP header

HTTPS is documented in RFC 2818, *HTTP Over TLS*. There is no fundamental change in using HTTP over either SSL or TLS, and both implementations are referred to as HTTPS.

Connection Initiation

For HTTPS, the agent acting as the HTTP client also acts as the TLS client. The client initiates a connection to the server on the appropriate port and then sends the TLS ClientHello to begin the TLS handshake. When the TLS handshake has

finished, the client may then initiate the first HTTP request. All HTTP data is to be sent as TLS application data. Normal HTTP behavior, including retained connections, should be followed.

There are three levels of awareness of a connection in HTTPS. At the HTTP level, an HTTP client requests a connection to an HTTP server by sending a connection request to the next lowest layer. Typically, the next lowest layer is TCP, but it also may be TLS/SSL. At the level of TLS, a session is established between a TLS client and a TLS server. This session can support one or more connections at any time. As we have seen, a TLS request to establish a connection begins with the establishment of a TCP connection between the TCP entity on the client side and the TCP entity on the server side.

Connection Closure

An HTTP client or server can indicate the closing of a connection by including the following line in an HTTP record: `Connection: close`. This indicates that the connection will be closed after this record is delivered.

The closure of an HTTPS connection requires that TLS close the connection with the peer TLS entity on the remote side, which will involve closing the underlying TCP connection. At the TLS level, the proper way to close a connection is for each side to use the TLS alert protocol to send a `close_notify` alert. TLS implementations must initiate an exchange of closure alerts before closing a connection. A TLS implementation may, after sending a closure alert, close the connection without waiting for the peer to send its closure alert, generating an "incomplete close". Note that an implementation that does this may choose to reuse the session. This should only be done when the application knows (typically through detecting HTTP message boundaries) that it has received all the message data that it cares about.

HTTP clients also must be able to cope with a situation in which the underlying TCP connection is terminated without a prior `close_notify` alert and without a `Connection: close` indicator. Such a situation could be due to a programming error on the server or a communication error that causes the TCP connection to drop. However, the unannounced TCP closure could be evidence of some sort of attack. So the HTTPS client should issue some sort of security warning when this occurs.

6.4 SECURE SHELL (SSH)

Secure Shell (SSH) is a protocol for secure network communications designed to be relatively simple and inexpensive to implement. The initial version, SSH1 was focused on providing a secure remote logon facility to replace TELNET and other remote logon schemes that provided no security. SSH also provides a more general client/server capability and can be used for such network functions as file transfer and e-mail. A new version, SSH2, fixes a number of security flaws in the original scheme. SSH2 is documented as a proposed standard in IETF RFCs 4250 through 4256.

SSH client and server applications are widely available for most operating systems. It has become the method of choice for remote login and X tunneling and

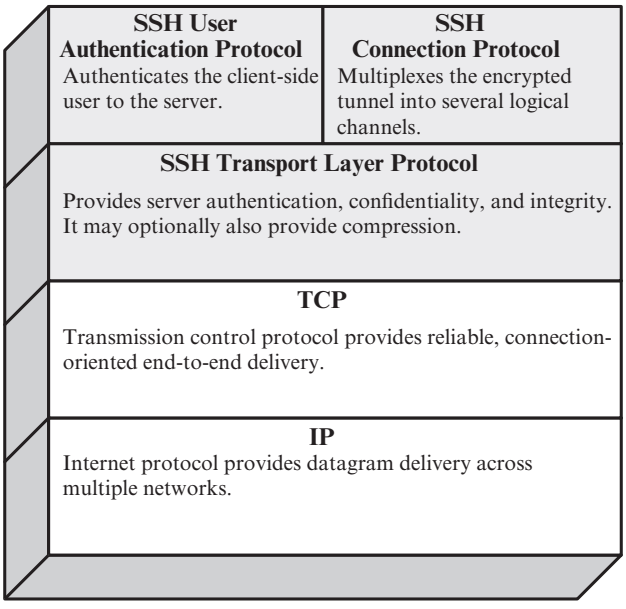

Figure 6.8 SSH Protocol Stack

is rapidly becoming one of the most pervasive applications for encryption technology outside of embedded systems.

SSH is organized as three protocols that typically run on top of TCP (Figure 6.8):

- **Transport Layer Protocol:** Provides server authentication, data confidentiality, and data integrity with forward secrecy (i.e., if a key is compromised during one session, the knowledge does not affect the security of earlier sessions). The transport layer may optionally provide compression.
- **User Authentication Protocol:** Authenticates the user to the server.
- **Connection Protocol:** Multiplexes multiple logical communications channels over a single, underlying SSH connection.

Transport Layer Protocol

Host Keys Server authentication occurs at the transport layer, based on the server possessing a public/private key pair. A server may have multiple host keys using multiple different asymmetric encryption algorithms. Multiple hosts may share the same host key. In any case, the server host key is used during key exchange to authenticate the identity of the host. For this to be possible, the client must have a priori knowledge of the server's public host key. RFC 4251 dictates two alternative trust models that can be used:

1. The client has a local database that associates each host name (as typed by the user) with the corresponding public host key. This method requires no centrally administered infrastructure and no third-party coordination. The downside is that the database of name-to-key associations may become burdensome to maintain.