Protocol takes an application message to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, adds a header, and transmits the resulting unit in a TCP segment. Received data are decrypted, verified, decompressed, and reassembled before being delivered to higher-level users. The first step is fragmentation. Each upper-layer message is fragmented into blocks of 214 bytes (16,384 bytes) or less. Next, compression is optionally applied. Compression must be lossless and may not increase the content length by more than 6.2 / Transport Layer Security 193 1024 bytes.1 In TLSv2, no compression algorithm is specified, so the default compression algorithm is null. The next step in processing is to compute a message authentication code over the compressed data. TLS makes use of the HMAC algorithm defined in RFC 2104. Recall from Chapter 3 that HMAC is defined as HMACK(M) = H[(K+ ⊕ opad) ‘ H[(K+ ⊕ ipad) ‘ M]] where H = embedded hash function (for TLS, either MD5 or SHA-1) M = message input to HMAC K+ = secret key padded with zeros on the left so that the result is equal to the block length of the hash code (for MD5 and SHA-1, block length = 512 bits) ipad = 00110110 (36 in hexadecimal) repeated 64 times (512 bits) opad = 01011100 (5C in hexadecimal) repeated 64 times (512 bits) For TLS, the MAC calculation encompasses the fields indicated in the following expression: HMAC_hash(MAC_write_secret, seq_num ‘ TLSCompressed.type ‘ TLSCompressed.version ‘ TLSCompressed.length ‘ TLSCompressed.fragment) The MAC calculation covers all of the fields XXX, plus the field TLSCompressed.version, which is the version of the protocol being employed. Next, the compressed message plus the MAC are encrypted using symmetric encryption. Encryption may not increase the content length by more than 1024 bytes, 1 Of course, one hopes that compression shrinks rather than expands the data. However, for very short blocks, it is possible, because of formatting conventions, that the compression algorithm will actually provide output that is longer than the input. Figure 6.3 TLS Record Protocol Operation Application data Fragment Compress Add MAC Encrypt Append TLS record header 194 chapter 6 / Transport-Level Security so that the total length may not exceed 214 + 2048. The following encryption algorithms are permitted: Block Cipher Stream Cipher Algorithm Key Size Algorithm Key Size AES 3DES 128, 256 168 RC4-128 128 For stream encryption, the compressed message plus the MAC are encrypted. Note that the MAC is computed before encryption takes place and that the MAC is then encrypted along with the plaintext or compressed plaintext. For block encryption, padding may be added after the MAC prior to encryption. The padding is in the form of a number of padding bytes followed by a onebyte indication of the length of the padding. The padding can be any amount that results in a total that is a multiple of the cipher’s block length, up to a maximum of 255 bytes. For example, if the cipher block length is 16 bytes (e.g., AES) and if the plaintext (or compressed text if compression is used) plus MAC plus padding length byte is 79 bytes long, then the padding length (in bytes) can be 1, 17, 33, and so on, up to 161. At a padding length of 161, the total length is 79 + 161 = 240. A variable padding length may be used to frustrate attacks based on an analysis of the lengths of exchanged messages. The final step of TLS Record Protocol processing is to prepend a header consisting of the following fields: ■■ Content Type (8 bits): The higher-layer protocol used to process the enclosed fragment. ■■ Major Version (8 bits): Indicates major version of TLS in use. For TLSv2, the value is 3. ■■ Minor Version (8 bits): Indicates minor version in use. For TLSv2, the value is 1. ■■ Compressed Length (16 bits): The length in bytes of the plaintext fragment (or compressed fragment if compression is used). The maximum value is 214 + 2048. The content types that have been defined are change_cipher_spec, alert, handshake, and application_data. The first three are the TLSspecific protocols, discussed next. Note that no distinction is made among the various applications (e.g., HTTP) that might use TLS; the content of the data created by such applications is opaque to TLS. Figure 6.4 illustrates the TLS record format. Change Cipher Spec Protocol The Change Cipher Spec Protocol is one of the four TLS-specific protocols that use the TLS Record Protocol, and it is the simplest. This protocol consists of a single message (Figure 6.5a), which consists of a single byte with the value 1. The sole purpose of this message is to cause the pending state to be copied into the current state, which updates the cipher suite to be used on this connection. 6.2 / Transport Layer Security 195 Alert Protocol The Alert Protocol is used to convey TLS-related alerts to the peer entity. As with other applications that use TLS, alert messages are compressed and encrypted, as specified by the current state. Each message in this protocol consists of two bytes (Figure 6.5b). The first byte takes the value warning (1) or fatal (2) to convey the severity of the message. If the level is fatal, TLS immediately terminates the connection. Other connections on the same session may continue, but no new connections on this session may be established. The second byte contains a code that indicates the specific alert. The following alerts are always fatal: ■■ unexpected_message: An inappropriate message was received. ■■ bad_record_mac: An incorrect MAC was received. ■■ decompression_failure: The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length). ■■ handshake_failure: Sender was unable to negotiate an acceptable set of security parameters given the options available. ■■ illegal_parameter: A field in a handshake message was out of range or inconsistent with other fields. Figure 6.5 TLS Record Protocol Payload 1 (a) Change cipher spec protocol 1 byte Type (c) Handshake protocol 1 byte Length 3 bytes Content Ú 0 bytes (d) Other upper-layer protocol (e.g., HTTP) Opaque content Ú 1 byte Level (b) Alert protocol 1 byte 1 byte Alert Figure 6.4 TLS Record Format Content type Major version Minor version Compressed length Plaintext (optionally compressed) MAC (0, 16, or 20 bytes) Encrypted 196 chapter 6 / Transport-Level Security ■■ decryption_failed: A ciphertext decrypted in an invalid way; either it was not an even multiple of the block length or its padding values, when checked, were incorrect. ■■ record_overflow: A TLS record was received with a payload (ciphertext) whose length exceeds 214 + 2048 bytes, or the ciphertext decrypted to a length of greater than 214 + 1024 bytes. ■■ unknown_ca: A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA. ■■ access_denied: A valid certificate was received, but when access control was applied, the sender decided not to proceed with the negotiation. ■■ decode_error: A message could not be decoded, because either a field was out of its specified range or the length of the message was incorrect. ■■ export_restriction: A negotiation not in compliance with export restrictions on key length was detected. ■■ protocol_version: The protocol version the client attempted to negotiate is recognized but not supported. ■■ insufficient_security: Returned instead of handshake_failure when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. ■■ internal_error: An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue. The remaining alerts are the following. ■■ close_notify: Notifies the recipient that the sender will not send any more messages on this connection. Each party is required to send a close_notify alert before closing the write side of a connection. ■■ bad_certificate: A received certificate was corrupt (e.g., contained a signature that did not verify). ■■ unsupported_certificate: The type of the received certificate is not supported. ■■ certificate_revoked: A certificate has been revoked by its signer. ■■ certificate_expired: A certificate has expired. ■■

certificate_unknown: Some other unspecified issue arose in processing the certificate, rendering it unacceptable. ■■ decrypt_error: A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message. ■■ user_canceled: This handshake is being canceled for some reason unrelated to a protocol failure. ■■ no_renegotiation: Sent by a client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these messages would normally result in renegotiation, but this alert indicates that the sender is not able to renegotiate. This message is always a warning. 6.2 / Transport Layer Security 197 Handshake Protocol The most complex part of TLS is the Handshake Protocol. This protocol allows the server and client to authenticate each other and to negotiate an encryption and MAC algorithm and cryptographic keys to be used to protect data sent in a TLS record. The Handshake Protocol is used before any application data is transmitted. The Handshake Protocol consists of a series of messages exchanged by client and server. All of these have the format shown in Figure 6.5c. Each message has three fields: ■■ Type (1 byte): Indicates one of 10 messages. Table 6.2 lists the defined message types. ■■ Length (3 bytes): The length of the message in bytes. ■■ Content (# 0 bytes): The parameters associated with this message; these are listed in Table 6.2. Figure 6.6 shows the initial exchange needed to establish a logical connection between client and server. The exchange can be viewed as having four phases. Phase 1. Establish Security Capabilities Phase 1 initiates a logical connection and establishes the security capabilities that will be associated with it. The exchange is initiated by the client, which sends a client_hello message with the following parameters: ■■ Version: The highest TLS version understood by the client. ■■ Random: A client-generated random structure consisting of a 32-bit timestamp and 28 bytes generated by a secure random number generator. These values serve as nonces and are used during key exchange to prevent replay attacks. ■■ Session ID: A variable-length session identifier. A nonzero value indicates that the client wishes to update the parameters of an existing connection or to create a new connection on this session. A zero value indicates that the client wishes to establish a new connection on a new session. Message Type Parameters hello_request null client_hello version, random, session id, cipher suite, compression method server_hello version, random, session id, cipher suite, compression method certificate chain of X.509v3 certificates server_key_exchange parameters, signature certificate_request type, authorities server_done null certificate_verify signature client_key_exchange parameters, signature finished hash value Table 6.2 TLS Handshake Protocol Message Types 198 chapter 6 / Transport-Level Security ■■ CipherSuite: This is a list that contains the combinations of cryptographic algorithms supported by the client, in decreasing order of preference. Each element of the list (each cipher suite) defines both a key exchange algorithm and a CipherSpec; these are discussed subsequently. ■■ Compression Method: This is a list of the compression methods the client supports. After sending the client_hello message, the client waits for the server_ hello message, which contains the same parameters as the client_hello Figure 6.6 Handshake Protocol Action Client Server Phase 1 Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers. Phase 2 Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase. Phase 3 Client sends certificate if requested. Client sends key exchange. Client may send certificate verification. Phase 4 Change cipher suite and finish handshake protocol. Note: Shaded transfers are optional or situation-dependent messages that are not always sent. finished change_cipher_spec finished change_cipher_spec certificate_verify client_key_exchange certicate server_hello_done certicate_request server_key_exchange certicate server_hello client_hello Time 6.2 / Transport Layer Security 199 message. For the server_hello message, the following conventions apply. The Version field contains the lowest of the version suggested by the client and the highest supported by the server. The Random field is generated by the server and is independent of the client's Random field. If the SessionID field of the client was nonzero, the same value is used by the server; otherwise the server's SessionID field contains the value for a new session. The CipherSuite field contains the single cipher suite selected by the server from those proposed by the client. The Compression field contains the compression method selected by the server from those proposed by the client. The first element of the Ciphersuite parameter is the key exchange method (i.e., the means by which the cryptographic keys for conventional encryption and MAC are exchanged). The following key exchange methods are supported. ■■ RSA: The secret key is encrypted with the receiver's RSA public key. A publickey certificate for the receiver's key must be made available. ■■ Fixed Diffie–Hellman: This is a Diffie–Hellman key exchange in which the server's certificate contains the Diffie–Hellman public parameters signed by the certificate authority (CA). That is, the public-key certificate contains the Diffie–Hellman public-key parameters. The client provides its Diffie– Hellman public-key parameters either in a certificate, if client authentication is required, or in a key exchange message. This method results in a fixed secret key between two peers based on the Diffie–Hellman calculation using the fixed public keys. ■■ Ephemeral Diffie–Hellman: This technique is used to create ephemeral (temporary, one-time) secret keys. In this case, the Diffie–Hellman public keys are exchanged and signed using the sender's private RSA or DSS key. The receiver can use the corresponding public key to verify the signature. Certificates are used to authenticate the public keys. This would appear to be the most secure of the three Diffie–Hellman options because it results in a temporary, authenticated key. ■■ Anonymous Diffie–Hellman: The base Diffie–Hellman algorithm is used with no authentication. That is, each side sends its public Diffie–Hellman parameters to the other with no authentication. This approach is vulnerable to man-in-the-middle attacks, in which the attacker conducts anonymous Diffie– Hellman with both parties. Following the definition of a key exchange method is the CipherSpec, which includes the following fields: ■■ CipherAlgorithm: Any of the algorithms mentioned earlier: RC4, RC2, DES, 3DES, DES40, or IDEA ■■ MACAlgorithm: MD5 or SHA-1 ■■ CipherType: Stream or Block ■■ IsExportable: True or False ■■ HashSize: 0, 16 (for MD5), or 20 (for SHA-1) bytes ■■ Key Material: A sequence of bytes that contain data used in generating the write keys ■■ IV Size: The size of the Initialization Value for Cipher Block Chaining (CBC) encryption 200 chapter 6 / Transport-Level Security 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 hash(ClientHello.random ‘ ServerHello.random ‘ 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 6.2 / Transport Layer Security 201 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. 202 chapter 6 / Transport-Level Security 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. 6.2 / Transport Layer Security 203 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 key_block = MD5(master_secret ‘ SHA(=A> ‘ master_secret ‘ ServerHello.random ‘ ClientHello.random)) ‘ MD5(master_secret ‘ SHA(=BB> ‘ master_secret ‘ ServerHello.random ‘ ClientHello.random)) ‘ MD5(master_secret ‘ SHA(=CCC> ‘ master_secret ‘ ServerHello.random ‘ ClientHello.random)) ‘ c 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 P_hash(secret, seed) = HMAC_hash(secret, A(1) ‘ seed) ‘ HMAC_hash(secret, A(2) ‘ seed) ‘ HMAC_hash(secret, A(3) ‘ seed) ‘ where A() is defined as A(0) = seed A(i) = HMAC_hash(secret, A(i - 1)) 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. 204 chapter 6 / Transport-Level Security 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 PRF(secret, label, seed) = P_6hash7(secret, label ‘ 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). Figure 6.7 TLS Function P_hash(secret, seed) Secret Seed Seed A(1) HMAC Secret Secret Length = hash size Secret Seed A(2) HMAC HMAC Secret Seed A(3) HMAC HMAC Secret HMAC 6.2 / Transport Layer Security 205 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 206 chapter 6 / Transport-Level Security 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 6.3 / HTTPS 207 –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 208 chapter 6 / Transport-Level Security 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 6.4 / Secure Shell (SSH) 209 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. Figure 6.8 SSH Protocol Stack SSH User Authentication Protocol SSH Transport Layer Protocol TCP IP Internet protocol provides datagram delivery across multiple networks. Transmission control protocol provides reliable, connectionoriented end-to-end delivery. Provides server authentication, condentiality, and integrity. It may optionally also provide compression. Authenticates the client-side user to the server. SSH Connection Protocol Multiplexes the encrypted tunnel into several logical channels. M 210 chapter 6 / Transport-Level Security 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 Client Server SSH-protoversion-softwareversion Identication string exchange Algorithm negotiation End of key exchange Service request SSH-protoversion-softwareversion SSH_MSG_KEXINIT SSH_MSG_KEXINIT SSH_MSG_NEWKEYS SSH_MSG_NEWKEYS SSH_MSG_SERVICE_REQUEST Establish TCP Connection Key Exchange 6.4 / Secure Shell (SSH) 211 ■■ 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 Figure 6.10 SSH Transport Layer Protocol Packet Formation pdlpktl pktl = packet length pdl = padding length # qes gniddaP Payload SSH Packet Compressed payload Ciphertext COMPRESS ENCRYPT MAC 212 chapter 6 / Transport-Level Security 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. 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 MAC algorithm hmac-sha1* HMAC-SHA1; digest length = key length = 20 hmac-sha1-96** First 96 bits of HMACSHA1; digest length = 12; key length = 20 hmac-md5 HMAC-MD5; digest length = key length = 16 hmac-md5-96 First 96 bits of HMAC-MD5; digest length = 12; key length = 16 Compression algorithm none* No compression zlib Defined in RFC 1950 and RFC 1951 Cipher 3des-cbc* Three-key 3DES in CBC mode blowfish-cbc Blowfish in CBC mode twofish256-cbc Twofish in CBC mode with a 256-bit key twofish192-cbc Twofish with a 192-bit key twofish128-cbc Twofish with a 128-bit key aes256-cbc AES in CBC mode with a 256-bit key aes192-cbc AES with a 192-bit key aes128-cbc** AES with a 128-bit key 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 6.4 / Secure Shell (SSH) 213 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 6 x 6 q) and computes e = gx mod p. C sends e to S. 2. S generates a random number y(0 6 y 6 q) and computes f = gy mod p. S receives e. It computes K = ey mod p, H = 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 mod p, H = 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: HASH(K ‘ H ‘ ;A< ‘ session_id) ■■ Initial IV server to client: HASH(K ‘ H ‘ ;B< ‘ session_id) ■■ Encryption key client to server: HASH(K ‘ H ‘ ;C< ‘ session_id) ■■ Encryption key server to client: HASH(K ‘ H ‘ ;D< ‘ session_id) ■■ Integrity key client to server: HASH(K ‘ H ‘ ;E< ‘ session_id) ■■ Integrity key server to client: HASH(K ‘ H ‘ ;F< ‘ session_id) where HASH() is the hash function determined during algorithm negotiation. 214 chapter 6 / Transport-Level Security 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. 6.4 / Secure Shell (SSH) 215 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.2 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. 2 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. 216 chapter 6 / Transport-Level Security 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. 6.4 / Secure Shell (SSH) 217 ■■ 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 Figure 6.11 Example of SSH Connection Protocol Message Exchange Client Server SSH_MSG_CHANNEL_OPEN Open a channel Data transfer Close a channel SSH_MSG_CHANNEL_OPEN_CONFIRMATION SSH_MSG_CHANNEL_DATA SSH_MSG_CHANNEL_DATA SSH_MSG_CHANNEL_DATA SSH_MSG_CHANNEL_DATA SSH_MSG_CHANNEL_CLOSE Establish Authenticated Transport Layer Connection 218 chapter 6 / Transport-Level Security 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 Figure 6.12 SSH Transport Layer Packet Exchanges Client Server Client application Unsecure TCP connection (a) Connection via TCP TCP entity x y Server application TCP entity Client application Secure SSH tunnel (b) Connection via SSH tunnel SSH entity x y Server application SSH entity TCP Unsecure TCP connection entity a b TCP entity 6.4 / Secure Shell (SSH) 219 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. 220 chapter 6 / Transport-Level Security 6.5 Key Terms, Review Questions, And Problems Key Terms Alert protocol Change Cipher Spec protocol Handshake protocol HTTPS (HTTP over SSL) Master Secret Secure Shell (SSH) Secure Socket Layer (SSL) Transport Layer Security (TLS) Review Questions 6.1 What are the advantages of each of the three approaches shown in Figure 6.1? 6.2 What protocols comprise TLS? 6.3 What is the difference between a TLS connection and a TLS session? 6.4 List and briefly define the parameters that define a TLS session state. 6.5 List and briefly define the parameters that define a TLS session connection. 6.6 What services are provided by the TLS Record Protocol? 6.7 What steps are involved in the TLS Record Protocol transmission? 6.8 Give brief details about different levels of awareness of a connection in HTTPS. 6.9 Which protocol was replaced by SSH

and why? Which version is currently in the process of being standardized? 6.10 List and briefly define the SSH protocols. Problems 6.1 In SSL and TLS, why is there a separate Change Cipher Spec Protocol rather than including a change_cipher_spec message in the Handshake Protocol? 6.2 What purpose does the MAC serve during the change cipher spec TLS exchange? 6.3 Consider the following threats to Web security and describe how each is countered by a particular feature of TLS. a. Brute-Force Cryptanalytic Attack: An exhaustive search of the key space for a conventional encryption algorithm. b. Known Plaintext Dictionary Attack: Many messages will contain predictable plaintext, such as the HTTP GET command. An attacker constructs a dictionary containing every possible encryption of the known-plaintext message. When an encrypted message is intercepted, the attacker takes the portion containing the encrypted known plaintext and looks up the ciphertext in the dictionary. The ciphertext should match against an entry that was encrypted with the same secret key. If there are several matches, each of these can be tried against the full ciphertext to determine the right one. This attack is especially effective against small key sizes (e.g., 40-bit keys). c. Replay Attack: Earlier TLS handshake messages are replayed. d. Man-in-the-Middle Attack: An attacker interposes during key exchange, acting as the client to the server and as the server to the client. e. Password Sniffing: Passwords in HTTP or other application traffic are eavesdropped. f. IP Spoofing: Uses forged IP addresses to fool a host into accepting bogus data. 6.5 / Key Terms, Review Questions, And Problems 221 g. IP Hijacking: An active, authenticated connection between two hosts is disrupted and the attacker takes the place of one of the hosts. h. SYN Flooding: An attacker sends TCP SYN messages to request a connection but does not respond to the final message to establish the connection fully. The attacked TCP module typically leaves the “half-open connection” around for a few minutes. Repeated SYN messages can clog the TCP module. 6.4 Based on what you have learned in this chapter, is it possible in TLS for the receiver to reorder TLS record blocks that arrive out of order? If so, explain how it can be done. If not, why not? 6.5 For SSH packets, what is the advantage, if any, of not including the MAC in the scope of the packet encryption? 222 7.1 Wireless Security Wireless Network Threats Wireless Security Measures 7.2 Mobile Device Security Security Threats Mobile Device Security Strategy 7.3 IEEE 802.11 Wireless LAN Overview The Wi-Fi Alliance IEEE 802 Protocol Architecture IEEE 802.11 Network Components and Architectural Model IEEE 802.11 Services 7.4 IEEE 802.11i Wireless LAN Security IEEE 802.11i Services IEEE 802.11i Phases of Operation Discovery Phase Authentication Phase Key Management Phase Protected Data Transfer Phase The IEEE 802.11i Pseudorandom Function 7.5 Key Terms, Review Questions, and Problems Chapter Wireless Network Security 7.1 / Wireless Security 223 This chapter begins with a general overview of wireless security issues. We then focus on the relatively new area of mobile device security, examining threats and countermeasures for mobile devices used in the enterprise. Then, we look at the IEEE 802.11i standard for wireless LAN security. This standard is part of IEEE 802.11, also referred to as Wi-Fi. We begin the discussion with an overview of IEEE 802.11, and then we look in some detail at IEEE 802.11i. 7.1 Wireless Security Wireless networks, and the wireless devices that use them, introduce a host of security problems over and above those found in wired networks. Some of the key factors contributing to the higher security risk of wireless networks compared to wired networks include the following [MA10]: ■■ Channel: Wireless networking typically involves broadcast communications, which is far more susceptible to eavesdropping and jamming than wired networks. Wireless networks are also more vulnerable to active attacks that exploit vulnerabilities in communications protocols. ■■ Mobility: Wireless devices are, in principal and usually in practice, far more portable and mobile than wired devices. This mobility results in a number of risks, described subsequently. ■■ Resources: Some wireless devices, such as smartphones and tablets, have sophisticated operating systems but limited memory and processing resources with which to counter threats, including denial of service and malware. ■■ Accessibility: Some wireless devices, such as sensors and robots, may be left unattended in remote and/or hostile locations. This greatly increases their vulnerability to physical attacks. Learning Objectives After studying this chapter, you should be able to: ◆ Present an overview of security threats and countermeasures for wireless networks. ◆ Understand the unique security threats posed by the use of mobile devices with enterprise networks. ◆ Describe the principal elements in a mobile device security strategy. ◆ Understand the essential elements of the IEEE 802.11 wireless LAN standard. ◆ Summarize the various components of the IEEE 802.11i wireless LAN security architecture. 224 chapter 7 / Wireless Network Security In simple terms, the wireless environment consists of three components that provide point of attack (Figure 7.1). The wireless client can be a cell phone, a Wi-Fi–enabled laptop or tablet, a wireless sensor, a Bluetooth device, and so on. The wireless access point provides a connection to the network or service. Examples of access points are cell towers, Wi-Fi hotspots, and wireless access points to wired local or wide area networks. The transmission medium, which carries the radio waves for data transfer, is also a source of vulnerability. Wireless Network Threats [CHOI08] lists the following security threats to wireless networks: ■■ Accidental association: Company wireless LANs or wireless access points to wired LANs in close proximity (e.g., in the same or neighboring buildings) may create overlapping transmission ranges. A user intending to connect to one LAN may unintentionally lock on to a wireless access point from a neighboring network. Although the security breach is accidental, it nevertheless exposes resources of one LAN to the accidental user. ■■ Malicious association: In this situation, a wireless device is configured to appear to be a legitimate access point, enabling the operator to steal passwords from legitimate users and then penetrate a wired network through a legitimate wireless access point. ■■ Ad hoc networks: These are peer-to-peer networks between wireless computers with no access point between them. Such networks can pose a security threat due to a lack of a central point of control. ■■ Nontraditional networks: Nontraditional networks and links, such as personal network Bluetooth devices, barcode readers, and handheld PDAs, pose a security risk in terms of both eavesdropping and spoofing. ■■ Identity theft (MAC spoofing): This occurs when an attacker is able to eavesdrop on network traffic and identify the MAC address of a computer with network privileges. ■■ Man-in-the middle attacks: This type of attack is described in Chapter 3 in the context of the Diffie–Hellman key exchange protocol. In a broader sense, this attack involves persuading a user and an access point to believe that they are talking to each other when in fact the communication is going through an intermediate attacking device. Wireless networks are particularly vulnerable to such attacks. Figure 7.1 Wireless Networking Components Endpoint Wireless medium Access point 7.1 / Wireless Security 225 ■■ Denial of service (DoS): This type of attack is discussed in detail in Chapter 10. In the context of a wireless network, a DoS attack occurs when an attacker continually bombards a wireless access point or some other accessible wireless port with various protocol messages designed to consume system resources. The wireless environment lends itself to this type of attack, because it is so easy for the attacker to direct multiple wireless messages at the target. ■■ Network injection: A network injection attack targets wireless access points that are exposed to nonfiltered network traffic, such as routing protocol messages or network management messages. An example of such an attack is one in which bogus reconfiguration commands are used to affect routers and switches to degrade network performance. Wireless Security Measures Following [CHOI08], we can group wireless security measures into those dealing with wireless transmissions, wireless access points, and wireless networks (consisting of wireless routers and endpoints). Securing Wireless Transmissions The principal threats to wireless transmission are eavesdropping, altering or inserting messages, and disruption. To deal with eavesdropping, two types of countermeasures are appropriate: ■■ Signal-hiding techniques: Organizations can take a number of measures to make it more difficult for an attacker to locate their wireless access points, including turning off service set identifier (SSID) broadcasting by wireless access points; assigning cryptic names to SSIDs; reducing signal strength to the lowest level that still provides requisite coverage; and locating wireless access points in the interior of the building, away from windows and exterior walls. Greater security can be achieved by the use of directional antennas and of signal-shielding techniques. ■■ Encryption: Encryption of all wireless transmission is effective against eavesdropping to the extent that the encryption keys are secured. The use of encryption and authentication protocols is the standard method of countering attempts to alter or insert transmissions. The methods discussed in Chapter 10 for dealing with DoS apply to wireless transmissions. Organizations can also reduce the risk of unintentional DoS attacks. Site surveys can detect the existence of other devices using the same frequency range, to help determine where to locate wireless access points. Signal strengths can be adjusted and shielding used in an attempt to isolate a wireless environment from competing nearby transmissions. Securing Wireless Access Points The main threat involving wireless access points is unauthorized access to the network. The principal approach for preventing such access is the IEEE 802.1X standard for port-based network access control. The standard provides an authentication mechanism for devices wishing to attach to a LAN or wireless network. The use of 802.1X can prevent rogue access points and other unauthorized devices from becoming insecure backdoors. Section 5.3 provides an introduction to 802.1X. 226 chapter 7 / Wireless Network Security Securing Wireless Networks [CHOI08] recommends the following techniques for wireless network security: 1. Use encryption. Wireless routers are typically equipped with built-in encryption mechanisms for router-to-router traffic. 2. Use antivirus and antispyware software, and a firewall. These facilities should be enabled on all wireless network endpoints. 3. Turn off identifier broadcasting. Wireless routers are typically configured to broadcast an identifying signal so that any device within range can learn of the router’s existence. If a network is configured so that authorized devices know the identity of routers, this capability can be disabled, so as to thwart attackers. 4. Change the identifier on your router from the default. Again, this measure thwarts attackers who will attempt to gain access to a wireless network using default router identifiers. 5. Change your router’s pre-set password for administration. This is another prudent step. 6. Allow only specific computers to access your wireless network. A router can be configured to only communicate with approved MAC addresses. Of course, MAC addresses can be spoofed, so this is just one element of a security strategy. 7.2 Mobile Device Security Prior to the widespread use of smartphones, the dominant paradigm for computer and network security in organizations was as follows. Corporate IT was tightly controlled. User devices were typically limited to Windows PCs. Business applications were controlled by IT and either run locally on endpoints or on physical servers in data centers. Network security was based upon clearly defined perimeters that separated trusted internal networks from the untrusted Internet. Today, there have been massive changes in each of these assumptions. An organization’s networks must accommodate the following: ■■ Growing use of new devices: Organizations are experiencing significant growth in employee use of mobile devices. In many cases, employees are allowed to use a combination of endpoint devices as part of their day-to-day activities. ■■ Cloud-based applications: Applications no longer run solely on physical servers in corporate data centers. Quite the opposite, applications can run anywhere—on traditional physical servers, on mobile virtual servers, or in the cloud. Additionally, end users can now take advantage of a wide variety of cloud-based applications and IT services for personal and professional use. Facebook can be used for an employee’s personal profiles or as a component of a corporate marketing campaign. Employees depend upon Skype to speak with friends abroad or for legitimate business video conferencing. Dropbox and Box can be used to distribute documents between corporate and personal devices for mobility and user productivity. 7.2 / Mobile Device Security 227 ■■ De-perimeterization: Given new device proliferation, application mobility, and cloud-based consumer and corporate services, the notion of a static network perimeter is all but gone. Now there are a multitude of network perimeters around devices, applications, users, and data. These perimeters have also become quite dynamic as they must adapt to various environmental conditions such as user role, device type, server virtualization mobility, network location, and time-of-day. ■■ External business requirements: The enterprise must also provide guests, third-party contractors, and business partners network access using various devices from a multitude of locations. The central element in all of these changes is the mobile computing device. Mobile devices have become an essential element for organizations as part of the overall network infrastructure. Mobile devices such as smartphones, tablets, and memory sticks provide increased convenience for individuals as well as the potential for increased productivity in the workplace. Because of their widespread use and unique characteristics, security for mobile devices is a pressing and complex issue. In essence, an organization needs to implement a security policy through a combination of security features built into the mobile devices and additional security controls provided by network components that regulate the use of the mobile devices. Security Threats Mobile devices need additional, specialized protection measures beyond those implemented for other client devices, such as desktop and laptop devices that are used only within the organization’s facilities and on the organization’s networks. SP 800-14 (Guidelines for Managing and Securing Mobile Devices in the Enterprise, July 2012) lists seven major security concerns for mobile devices. We examine each of these in turn. Lack of Physical Security Controls Mobile devices are typically under the complete control of the user, and are used and kept in a variety of locations outside the organization’s control, including off premises. Even if a device is required to remain on premises, the user may move the device within the organization between secure and nonsecured locations. Thus, theft and tampering are realistic threats. The security policy for mobile devices must be based on the assumption that any mobile device may be stolen or at least accessed by a malicious party. The threat is twofold: A malicious party may attempt to recover sensitive data from the device itself, or may use the device to gain access to the organization’s resources. Use of Untrusted Mobile Devices In addition to company-issued and companycontrolled mobile devices, virtually all employees will have personal smartphones and/or tablets. The organization must assume that these devices are not trustworthy. That is, the devices may not employ encryption and either the user or a third party may have installed a bypass to the built-in restrictions on security, operating system use, and so on. 228 chapter 7 / Wireless Network Security Use of Untrusted Networks If a mobile device is used on premises, it can connect to organization resources over the organization’s own in-house wireless networks. However, for off-premises use, the user will typically access organizational resources via Wi-Fi or cellular access to the Internet and from the Internet to the organization. Thus, traffic that includes an off-premises segment is potentially susceptible to eavesdropping or man-in-the-middle types of attacks. Thus, the security policy must be based on the assumption that the networks between the mobile device and the organization are not trustworthy. Use of Applications Created by Unknown Parties By design, it is easy to find and install third-party applications on mobile devices. This poses the obvious risk of installing malicious software. An organization has several options for dealing with this threat, as described subsequently. Interaction with Other Systems A common feature found on smartphones and tablets is the ability to automatically synchronize data, apps, contacts, photos, and so on with other computing devices and with cloud-based storage. Unless an organization has control of all the devices involved in synchronization, there is considerable risk of the organization’s data being stored in an unsecured location, plus the risk of the introduction of malware. Use of Untrusted Content Mobile devices may access and use content that other computing devices do not encounter. An example is the Quick Response (QR) code, which is a two-dimensional barcode. QR codes are designed to be captured by a mobile device camera and used by the mobile device. The QR code translates to a URL, so that a malicious QR code could direct the mobile device to malicious Web sites. Use of Location Services The GPS capability on mobile devices can be used to maintain a knowledge of the physical location of the device. While this feature might be useful to an organization as part of a presence service, it creates security risks. An attacker can use the location information to determine where the device and user are located, which may be of use to the attacker. Mobile Device Security Strategy With the threats listed in the preceding discussion in mind, we outline the principal elements of a mobile device security strategy. They fall into three categories: device security, client/server traffic security, and barrier security (Figure 7.2). Device Security A number of organizations will supply mobile devices for employee use and preconfigure those devices to conform to the enterprise security policy. However, many organizations will find it convenient or even necessary to adopt a bring-your-own-device (BYOD) policy that allows the personal mobile devices of employees to have access to corporate resources. IT managers should be able to inspect each device before allowing network access. IT will want to establish configuration guidelines for operating systems and applications. For example, “rooted” or “jail-broken” devices are not permitted on the network, and mobile 7.2 / Mobile Device Security 229 devices cannot store corporate contacts on local storage. Whether a device is owned by the organization or BYOD, the organization should configure the device with security controls, including the following: ■■ Enable auto-lock, which causes the device to lock if it has not been used for a given amount of time, requiring the user to re-enter a four-digit PIN or a password to re-activate the device. ■■ Enable password or PIN protection. The PIN or password is needed to unlock the device. In addition, it can be configured so that e-mail and other data on the device are encrypted using the PIN or password and can only be retrieved with the PIN or password. ■■ Avoid using auto-complete features that remember user names or passwords. ■■ Enable remote wipe. ■■ Ensure that SSL protection is enabled, if available. ■■ Make sure that software, including operating systems and applications, is up to date. ■■ Install antivirus software as it becomes available. Figure 7.2 Mobile Device Security Elements Firewall Firewall limits scope of data and application access Authentication and access control protocols used to verify device and user and establish limits on access