---

19. SSL/TLS

19.1. Introduction

In this chapter we cover the SSL family of protocols, which includes SSL versions 2 and 3 and TLS. SSL version 2 is rapidly being replaced by version 3, so we will focus on SSL v3 and TLS, and only discuss v2 when its difference from v3 is interesting (such as the exportability tricks). SSL/TLS allows two parties to authenticate and establish a session key that is used to cryptographically protect the remainder of the session.

19.2. Using TCP

SSL/TLS is designed to run in a user-level process, and runs on top of TCP. As discussed in §16.1 *What Layer?*, running on top of layer 4 allows deployment of SSL/TLS in a user-level process rather than requiring OS changes. Using TCP (the reliable layer 4 protocol) rather than UDP (the datagram layer 4 protocol) makes SSL/TLS a little simpler, because it doesn't have to worry about timing out and retransmitting lost data (TCP does that for it). SSL/TLS could have retained the advantage of being easily deployable as a user-level process and still avoided the rogue packet problem discussed in section §16.1 *What Layer?* by running on top of UDP, and doing all the time-out/retransmission work of TCP within SSL/TLS, but the decision was made to live with the rogue packet problem and keep SSL/TLS simpler.

19.3. Quick History

SSL (Secure Sockets Layer) version 2 (version 1 was never deployed) was originally deployed in Netscape Navigator 1.1 by Netscape in 1995. Microsoft improved upon SSLv2, fixing some security problems, and introduced a similar protocol known as PCT (Private Communications Technology). Then Netscape substantially overhauled the protocol as SSLv3. The IETF, realizing it was bad for the industry to have three similar but incompatible protocols for the same purpose, introduced a fourth similar but incompatible protocol—TLS (Transport Layer Security).

As of the writing of this book, SSLv3 is the most commonly deployed. TLS tweaked the cryptographic algorithms for key expansion and authentication to make cryptographers happier with it. This made TLS noninteroperable with SSLv3. TLS also mandated unencumbered crypto (DH and DSS) rather than RSA. But there was only a window of 2 years in which DH/DSS was unencumbered before the RSA patent expired, which was not a sufficient lure to migrate many implementations away from the widely deployed SSLv3. It remains to be seen whether the world will ever migrate to TLS.

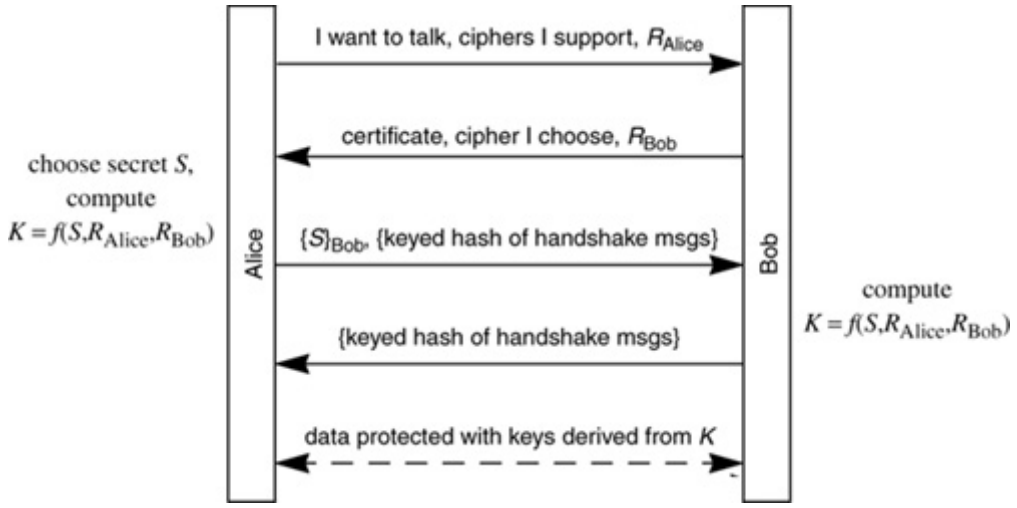
19.4. SSL/TLS Basic Protocol

Using the reliable octet stream service provided by TCP, SSL/TLS partitions this octet stream into records, with headers and cryptographic protection, to provide a reliable, encrypted, and integrity-protected stream of octets to the application. There are four types of records: user data, handshake messages, alerts (error messages or notification of connection closure), and change cipher spec (which should be a handshake message, but they chose to make it a separate record type).

In the basic protocol the client (Alice) initiates contact with the server (Bob). Then Bob sends Alice his certificate. Alice verifies the certificate, extracts Bob's public key, picks a random number S from which the session keys will be computed, and sends S to Bob, encrypted with Bob's public key. Then the remainder of the session is encrypted and integrity-protected with those session keys. (There are actually six secrets computed when encryption algorithms that require IVs are used—for each direction: integrity protection key, encryption key, and IV.) First we'll present a simplified form of the protocol, then discuss various issues in the full protocol, and finally discuss the details.

- Message 1. Alice says she would like to talk (but doesn't identify herself), and gives a list of cryptographic algorithms she supports, along with a random number R_{Alice}, that will be combined with the S in message 3 to form the various keys.
- Message 2. Bob sends Alice his certificate, a random number R_{Bob} that will also contribute to the keys, and responds with one of the ciphers Alice listed in message 1 that Bob also supports.
- Message 3. Alice chooses a random number S (known as the **pre-master secret**) and sends it, encrypted with Bob's public key. She also sends a hash of the **master secret** K and the hand-shake messages, both to prove she knows the key and to ensure that tampering of the handshake messages would be detected. The hash is an (unnecessarily) complex function based on one of the early versions of HMAC. To ensure that the keyed hash Alice sends is different from the keyed hash Bob sends, each side includes a constant ASCII string in the hash. The initiator constant is CLNT in SSLv3 and client finished in TLS. The constant Bob will hash into the stew is SRVR in SSLv3 and server finished in TLS. Surprisingly (and unnecessarily), the keyed hash is sent encrypted and integrity-protected. The keys used for encrypting the keyed hash, like the rest of the data in the session will be, are derived from hashing K, R_{Alice}, and R_{Bob}. The keys used for transmission are known as *write* keys, and the keys used for receipt are known as *read* keys. So, for instance, Bob's write-encryption key is Alice's read-encryption key. The keys are encryption, integrity, and IV in each direction, so there are six keys derived. (In SSLv2 there are only two session keys, one in each direction, each used both for integrity protection and encryption.)

Protocol 19-1. (simplified) SSLv3/TLS



- Message 4. Bob proves he knows the session keys, and ensures that the early messages arrived intact, by sending a keyed hash of all the handshake messages, encrypted with his write-encryption key, and integrity-protected with his write-integrity key. Since the session keys are derived from *S* this proves he knows Bob's private key, because he needed it in order to extract *S*.

At this point, Alice has authenticated Bob, but Bob has no idea to whom he's talking. As deployed today, authentication is seldom mutual—the client authenticates the server but the server does not authenticate the client. In theory SSL/TLS could be used for mutual authentication and the protocol allows optional authentication of the client if the client has a certificate. But in the most common case today, if the application on the server wishes to authenticate the user, it's usually done by having the user Alice send her name and password to the server Bob, cryptographically protected with the session keys.

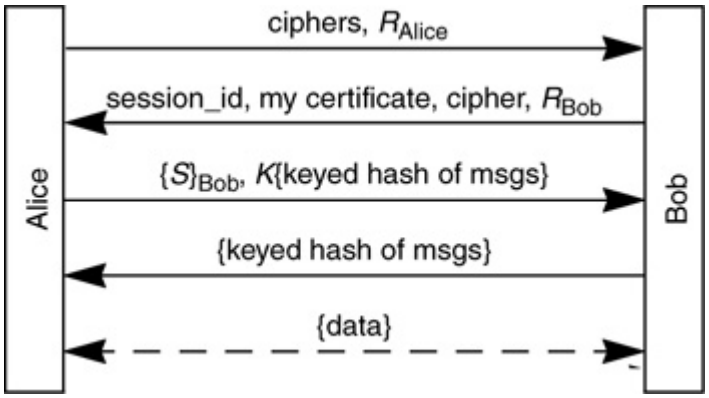
19.5. Session Resumption

SSL/TLS assumes that a session is a relatively long-lived thing from which many "connections" can be cheaply derived. This is because it was designed to work with HTTP 1.0, which has a tendency to open a lot of TCP connections between the same client and server (one per item on the web page).

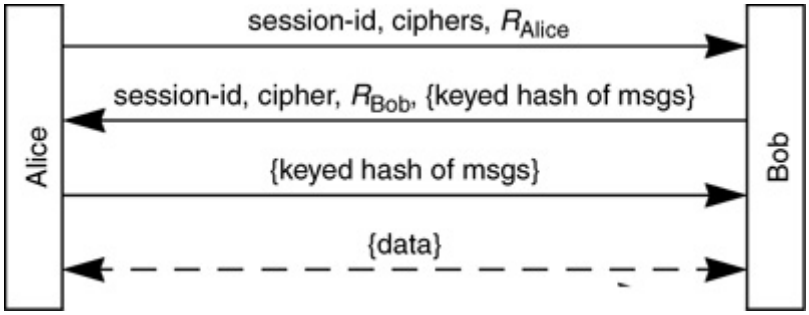
The per-session master secret is established using expensive public key cryptography. Multiple connections can be cheaply derived from that master secret by doing a handshake that involves sending nonces (so that session keys will be unique), but avoids public key operations.

SSL/TLS allows session resumption, but it is not stateless at Bob as in Lotus Notes (see §16.9 *Session Resumption*). If Bob wants to allow a session to resume, i.e., wants to be able to have multiple connections based on that session, he sends the client (in message 2) a session_id (which is not secret), and stores (session_id, master secret). If Alice presents (in message 1) a session_id that Bob remembers, they can skip the public key portion of the handshake. The shortened handshake will only succeed if Alice and Bob remember the same master secret. If Bob does not recognize the session_id, he returns a different session_id in message 2 (or leaves it out entirely if he wants to make the session non-resumable).

Protocol 19-2. Session initiation if no previous state



Protocol 19-3. Session resumption if both sides remember session-id



It might seem odd for Alice, when resuming the session, to send a set of ciphers rather than just the cipher used in the session. Even if Bob has lost state about the session, wouldn't he make the same choice of cipher, given the same choices, as he made when the session was created? Not necessarily, because perhaps his policy has changed. So Alice is allowed, when resuming a session, to send a set of choices, but the set must at least contain the cipher that had been chosen previously for that session.

19.6. Computing the Keys

The secret *S* sent in the first exchange is the pre-master secret. It is shuffled with the two *R*s to produce the master secret *K*. In other words $K = f(S, R_{Alice}, R_{Bob})$. For each connection (including the first) the master secret is shuffled with the two *R*s to produce the six keys used for that connection (for each side: encryption, integrity, and IV), i.e., each of the keys is $g_i(K, R_{Alice}, R_{Bob})$.

Note that this is unnecessarily complex in the case of using RSA, since on the first connection, the *R*s get shuffled in twice. It would have been fine with RSA to just use *S* the way *K* is used; for each connection, combine *S* with that connection's *R*s to produce the six keys used for that connection. However, there are other authentication methods that would wind up always computing the same pre-master secret *S*. For instance, with Diffie-Hellman using a fixed Diffie-Hellman number as your public key, the same two parties (Alice and Bob) will always compute the same *S*. If *S* were kept around in memory, it's possible that malicious software might steal it. So it's safer to hash it with some nonces to produce *K* (the master secret), since if *K* is stolen, it will only affect communication between Alice and Bob during the single session.

The keyed hashes are keyed with *K*, and are sent protected with the data keys (i.e., encrypted using the encryption key and IV, and integrity-protected with the integrity key).

The *R*s are 32 octets long, and it is recommended (but not enforced) that the first 4 octets not be randomly chosen, but instead be the Unix time (seconds since January 1, 1970) when the message was generated. This ensures (assuming that at least a second has elapsed) that Alice and Bob will choose different 32-octet *R*s each time a session under the same master secret is resumed, even if their random number generators are really bad or they're *really* unlucky. If the same *R*s were chosen with the same master secret, an attacker might be able to successfully replay packets.

19.7. Client Authentication

As deployed today it is unusual for clients to have certificates. However, it is possible in SSL/TLS for the server (Bob) to request that the client (Alice) authenticate herself. This is done by having Bob send a "certificate request" in message 2. Alice, upon seeing the request, sends her certificate, and her signature on a hash of the handshake messages, proving she knows the private key associated with the public key in the certificate.

19.8. PKI as Deployed by SSL

As deployed today, the client typically comes configured with public keys of various "trusted" organizations (trusted by the browser vendor, not necessarily by the user). The user at the client machine can modify this list, adding or deleting keys. The server sends a certificate to the client, and if it's signed by one of the CAs on the client's list, the client will accept the certificate. If the server presents a certificate signed by someone not on the list (such as a self-signed certificate), the user is typically presented with a pop-up box informing him that the certificate couldn't be verified because it was signed by an unknown authority, and would the user like to look at the certificate and/or import the signer onto the list of trusted root CAs? We discuss this model of PKI along with others in Chapter 15 *PKI (Public Key Infrastructure)*.

If the server wishes to authenticate the client, it sends a certificate request in which it specifies the X.500 names of the CAs it trusts, and the types of keys it can handle (e.g., RSA vs. DSS). This is a strange piece of asymmetry, since the client does not get to specify to the server what sort of certificate chain or key type it wants. And SSL/TLS insists on X.500 names, even though X.500 names are rarely if ever used on the Internet, and DNS names are now supported by the PKIX standard. The convention that browsers commonly use for equating a DNS name with an X.500 name is that the common name portion of the X.500 name holds the DNS name of the server and all the other name components in the X.500 name are ignored.

Another issue with the certificate request is that the name of a CA may not be sufficient description, since there might be several keys associated with a name, even possibly incorrectly associated with the name. It would have been better to have sent hashes of public keys.

In SSLv3 and TLS, a chain of certificates can be sent instead of a single certificate.

19.9. Version Numbers

The SSL/TLS designers had a strange idea about how to use a version number field. SSLv2 has a 2-octet field known as VERSION NUMBER which is set to 0.2 (0 in the high-order octet, 2 in the low-order octet). SSLv3 has a field known as VERSION NUMBER which is set 3.0 (3 in the high-order octet, 0 in the low-order octet). TLS has a field known as VERSION NUMBER which is set to 3.1 (3 in the high-order octet, 1 in the low-order octet). SSLv3 moved the VERSION NUMBER field! So you can't distinguish v2 and v3 messages by looking at the VERSION NUMBER field. An SSLv3 client cannot send an SSLv3 *client-hello* message, unless it knows somehow that the server it is contacting speaks SSLv3, since an SSLv2 server will misparse the packet and get confused. So it sends a v2 client-hello message, but fills in the VERSION NUMBER field as 3.0.

SSLv2 didn't specify what you do if you see a higher version number. Many implementations simply ignore the version number (that means they parse the message, not worrying if the VERSION NUMBER field indicates a higher version of the protocol than they support). SSLv3 and TLS implementations depend on this behavior. This implies of course that everything the client needs to say can be expressed in v2 format, which is true, and which makes you wonder why they needed to change the format for v3.

So an SSLv2 server that receives a message with a higher version number than it supports happily parses the message, assuming that nobody would have changed the message format in a later version of the protocol (even though the designers did just that for version 3!).

The designers of the higher versions were lucky that it is possible to distinguish a v2 message from a v3 or TLS message (if they hadn't moved the VERSION NUMBER field it would have been easy). For instance, it happens that the first octet of the v2 *client-hello* will be greater than 128, and the first octet of the v3 *client-hello* will be something between 20 and 23. There might indeed be other octets in the message that would distinguish v2 and v3. The spec doesn't say how to differentiate the two, which is dangerous, since it's conceivable someone might, in a future version, change the format again, and assume that implementations are differentiating versions based on the first octet when in fact they're looking elsewhere.

So SSLv3 clients send an SSLv2 *client-hello* message, as long as there might be a server out there still speaking SSLv2. Which means that in reality the SSLv3 *client-hello* never gets used, (except when resuming a session with a server that the client discovered was SSLv3). In the SSLv2 *client-hello* message, the v3 client puts the value 3.0. If the server speaks v3, it will respond with a v3 message. However, if the server is v2, it thinks its own version number is 0.2, and will interpret version 3.0 as 3×256+0, or 768. Wow! It never occurs to the SSLv2 server that the message might have gotten redesigned in between version 2 and version 768!

Suppose a v3 client somehow knew it was talking to a v3 server. Then it would be able to send a v3 *client-hello*. But that means that the server had better be able to tell the difference between a v2 *client-hello* and a v3 *client-hello*, since v3 clients that know that the server speaks v3 will send v3 *client-hello*s, while v2 clients or v3 clients that don't know what version the server speaks will send v2 *client-hello*s.

How does an SSLv3 client encode what it would like in an SSLv2 *client-hello*? Luckily there's nothing in the SSLv3 *client-hello* that can't easily be encoded in an SSLv2 *client-hello*. One interesting field is CIPHER SUITE, which had a 3-octet encoding in SSLv2 and a 2-octet encoding in SSLv3. It's easy to fit a 2-octet value into a 3-octet field. The reverse would have been more challenging. (See section §19.10 *Negotiating Cipher Suites*.)

It would be nice if the SSLv3 designers learned their lesson about version numbers based on the kludginess of v2/v3 interoperability, and had a fixed place in the packet for version number, and specified that a node that didn't support the higher version number should not attempt to parse the packet further but instead send back a message specifying the largest version it supported. But SSLv3 (and TLS) do the same thing SSLv2 did. If they see a higher version number, they happily parse the packet as if all future messages will be compatible with their version.

SSLv2 does not integrity-protect the *client-hello*. Therefore it's possible for an active attacker to modify a v3 client's v2 *client-hello* by changing the version number from 768 (SSLv3) or 769 (TLS) to 2. SSLv3 and TLS have a cute method of detecting this tampering despite the fact that the SSLv2 handshake doesn't explicitly protect against it. SSLv3 and TLS specify 3s as the least significant eight octets of the PKCS #1 padding (see §6.3.6.1 *Encryption*) when the client sends the pre-master secret (*S*) to the server.

So, if a v3 or TLS server receives a *client-hello* with version 2 and with the *S* value padding having 3s in the least significant eight octets, it breaks the connection and should issue an error. There is a 1 in 2^{64} chance that a v2 client might just happen to choose padding with the least significant eight octets being 3s, in which

case its connection will get mysteriously rejected.

19.10. Negotiating Cipher Suites

A cipher suite is a complete package (encryption algorithm, key length, integrity checksum algorithm—whatever needs to be defined in order to completely specify all the crypto SSL/TLS will need). Cipher suites are predefined and each is assigned a numeric value. In SSLv2 the value is 3 octets long, but in SSLv3 and TLS, it's 2 octets. This mercifully limits the number of suites that could potentially be defined in SSLv3/TLS to about 65000.

There are about 30 defined cipher suites, and there are 256 values reserved for private use (the ones with the most significant octet = ff_{16}. Private use values that have no way of coordinating their use are dangerous. You can define your own suite and choose any number out of the 256 reserved numbers for describing it. But there's no guarantee someone else won't define their own private suite and choose the same number. If two such systems attempt to talk, they'll think they are agreeing on the same suite, but in fact they will not interoperate.

In v2, each cipher suite had a 3-octet value. In v3, each is 2 octets. The way to specify a v3 cipher suite in the 3-octet v2 field is to set the most significant octet to 0, and have the v3 cipher suite value in the remaining 2 octets. Luckily, no v2 cipher suites had been defined with the most significant octet=0, so there was a convenient block of 2^{16} unused cipher suite values.

19.10.1. Who Makes the Decision?

In SSLv2, Bob returns the subset of Alice's suggested cipher suites that he is willing to support, but lets Alice make the final choice and announce it in message 3. (This is kind of silly, since Alice already said she'd support everything in her list. Bob should just pick one. Indeed one of the enhancements in SSLv3 is that Bob does make the choice, from the list Alice sent. If there's more than one that both he and Alice find acceptable, he makes the decision. And he could in theory make the decision in v2 by only returning one cipher suite rather than many.)

Not all supportable crypto suites are equal. The side choosing the suite will choose the most desirable (perhaps the strongest, or the fastest) one it can from the set of crypto suites that both sides support. An exportable suite is chosen only as a last resort (since the exportable suites are purposely weak).

19.10.2. Cipher Suite Names

Although cipher suites have 2-octet (or 3-octet in SSLv2) identifiers, they are referred to in the spec by names such as SSL_RSA_EXPORT_WITH_DES40_CBC_SHA:

- SSL means SSLv3, (the SSLv2 cipher suites have names starting with SSL2, e.g., SSL2_RC4_128_WITH_MD5),
- RSA means RSA,
- EXPORT means it's exportable (i.e., weak crypto, exportable before the rules were relaxed),
- WITH means the names weren't long enough,
- DES40 means DES with 40 bit keys,
- CBC means CBC-mode encryption, and
- SHA means HMAC-SHA is used for the MAC.

19.11. Negotiating Compression Method

Due to patent issues, only one compression method has been defined, which is NULL (type=0), the compression method that does not change the data. It is the belief of the TLS committee that implementing the NULL compression method will not infringe on any patents.

19.12. Attacks Fixed in v3

19.12.1. Downgrade Attack

In SSLv2, there is no integrity protection for the initial handshake, so an active attacker can remove the cipher suites with strong encryption from the list of requested cipher suites, causing Alice and Bob to agree upon a weaker cipher. In SSLv3 this was fixed by adding a *finished* message to the end of the initial handshake in which each side sends a digest of the messages in the handshake.

19.12.2. Truncation Attack

SSLv3 added a *finished* message to indicate that there is no more data to send. V2 depended on the TCP connection closing. But the TCP connection close is not cryptographically protected, so an attacker could close the connection by sending a TCP close message, and the other side would not be able to know that the session was abnormally terminated. Some applications have a way of specifying when the data is finished, but applications that don't might be fooled into incorrect behavior.

19.13. Exportability

The acrobatics we had to go through back in the dark ages of export control! Even if export controls have been permanently relaxed, we have to understand the mechanisms they engendered, since that is what is deployed and what must be implemented in order to be compatible with what was deployed. The simplest thing to have done with export controls was to always use weak crypto, even domestically. But it was legal to deploy strong crypto domestically, so complex mechanisms were built in order to make something that could be as secure as possible within the political constraints, and have the domestic and exportable versions interoperate.

19.13.1. Exportability in SSLv2

SSLv2 domestic clients supported 128-bit encryption. But the U.S. government limited exportable cryptographic keys to 40 bits. Some "exportable" suites were defined for SSLv2 which effectively used a 40-bit encryption key even though the actual cryptographic algorithm used a 128-bit key. In these exportable suites, instead of sending a 128-bit secret encrypted with the server's public key, the client sends a 40-bit secret encrypted with the server's public key, and sends an additional 88 bits in the clear. So in a non-exportable suite, there will be 128 secret bits. In an exportable suite, there will be 40 secret bits and 88 non-secret bits. So in either case there are 128 bits, which are known in SSLv2 as the **client master key** (not a terrific name for a quantity in which most of the bits are not secret). This key is used to produce the session keys.

Interestingly, the export laws were that encryption was limited to 40-bit secret keys and 512-bit RSA encryption keys, the latter normally used to encrypt secret keys. This is kind of weird, because if the encryption key is 40 bits, why should they care if it's sent encrypted in a really long RSA key? Or likewise, if they felt they could break 512-bit RSA, why couldn't you use a long encryption key as long as it had been sent across the wire encrypted with a short (512-bit) RSA key? But non-weird was never an important consideration in export rules.

So exportable servers had 512-bit RSA keys. Domestic servers had large RSA keys (e.g., 1024 bits). If a domestic server were talking to an exportable client, the client would wind up sending its 40-bit secret in the domestic server's large RSA key, which was technically illegal but apparently nobody noticed until after the protocol was approved and shipped. At that point, they agreed to let SSLv2 continue to be exported, with clients that would encrypt a 40-bit key in a 1024-bit RSA key, though they warned the designers they would not approve the same approach in SSLv3.

19.13.2. Exportability in SSLv3

In SSLv3 the pre-master secret sent by the client (and encrypted with the server's public key) is always a 48-octet number, but only 46 octets are random. The other two octets are used for version number. In the domestic version the pre-master secret is hashed with the *R*s to produce the master secret, and the master secret is hashed with the *R*s to produce the stream from which the six data protection keys are extracted.

If the cipher suite negotiated is exportable, then the integrity keys are computed as before, because export rules did not preclude strong integrity, but only 40 bits are extracted for each of the two encryption keys. And (due to someone's misunderstanding of the export rules), the IVs are computed solely from the two *R*s, so for the exportable cipher suites, the IVs are not secret. The export rules would have allowed strong IVs, but non-secret IVs are not a security vulnerability anyway.

But the 40 bits are only seeds for the encryption keys. These 40-bit seeds are hashed with the two *R*s to produce 128-bit keys. The reason for this step is similar to the reason for using salt: although there is still only a work factor of 2^{40} for breaking any individual encrypted session, you couldn't build up a table of 2^{40} keys for all sessions, since although there are only 2^{40} possible keys in any one session, the total number of possible keys is 2^{128} (for encryption algorithms with 128-bit keys).

But wait! 40-bit encryption keys wasn't sufficient to meet export restrictions. It was also a crime to send the 40-bit secret encrypted with an RSA key larger than 512 bits. A domestic server would have a 1024-bit RSA key, and might be speaking to an exportable client. It wouldn't be legal for the client to send its pre-master secret encrypted with a 1024-bit RSA key.

SSLv3 solves this problem by having the server (Bob) generate at random a 512-bit RSA key. This key can be changed periodically and is signed by Bob using his long-term key. The 512-bit key is known as the **ephemeral key**. In SSLv2 the client chose the cipher suite, but in SSLv3, since the server chooses the cipher suite, Bob figures out from the client's list of cipher suites in message 1 (and his own cipher suite capabilities) if he needs to be speaking exportable crypto. So in message 2, if Bob chooses an exportable cipher suite, then, in addition to sending his regular certificate (the one certifying his long-term key), Bob sends his 512-bit ephemeral key, signed using his long-term key. The client then sends the pre-master secret encrypted with Bob's ephemeral key.

This could have been implemented in a way that improved performance by having Bob sign the 512-bit ephemeral key once and use it for many connections. He would then only have to do a 512-bit private key operation for each connection instead of having to use the longer key. But doing so would have introduced a security vulnerability since someone who learned or broke a single 512-bit key could impersonate Bob indefinitely. Rather than putting an expiration time under the signature and risking clock synchronization problems, the protocol requires that Bob sign a combination of the ephemeral public key and the two nonces.

Aside from the performance penalty, in a perverse way, this hack, created for exportability, enhances security, because it allows weak-key perfect forward secrecy. Once Bob forgets the ephemeral private key, it would require breaking the 512-bit ephemeral public key or the 40-bit encryption key in order to decrypt the conversation, whereas with the non-exportable cipher suites, someone obtaining Bob's long-term private key would be able to directly decrypt previous conversations. So ironically, the NSA would have been better off with the SSLv2 solution. Although this perfect forward secrecy solution would have been useful in the non-exportable case, it is not supported in any server.

19.13.3. Server Gated Cryptography/Step-Up

The U.S. allowed an exported client to use strong cryptography when talking to some servers doing financial transactions. This meant that the client had to have all the code necessary for doing strong cryptography. But it would only use strong cryptography if the server's certificate specified that the server was one with which the client was allowed to speak strong crypto. This concept was implemented by both Netscape and Microsoft, but called **SGC** (**Server Gated Cryptography**) by Microsoft and **Step-Up** by Netscape. The protocols are slightly different, and the certificate extensions are different, since they were independently developed.

Whereas the usual certificates would be honored if signed with a chain starting with any of the trust anchors, the SGC certificate had to be signed by Verisign or Thawte (but Verisign bought Thawte so it was just Verisign), since this wasn't a matter of whom the client trusted, but whom the U.S. government trusted. Whereas the set of trust anchors for regular certificates could be modified by the user, the trusted SGC-certificate issuer(s) were wired into the implementation.

In the Step-Up exchange between an exportable client (Alice) with Step-Up capability and a server (Bob) with the Step-Up extension in its Verisign-signed certificate, there is the usual initial exchange (see Protocol 19-1) in which Alice offers only weak (exportable) crypto suites, Bob selects one of the weak suites, and they complete an exchange with a 40-bit key K_1. But Alice notices the Step-Up extension in Bob's certificate, so she continues with another handshake, with the handshake protected with K_1. (Once an SSL connection is established, handshake messages are allowed as well as data messages, so it is possible to conduct a new handshake protected by the keys established by the previous handshake.) In the new exchange, Alice proposes strong crypto and Bob accepts a strong cipher, and the new handshake completes, this time agreeing upon a strong key K_2. Then Alice sends a ChangeCipherSpec message, indicating a change from the weak crypto suite and key K_1 to the strong crypto suite and key K_2.

This might seem like a lot of messages, but this protocol would allow a server without special Step-Up code to speak strong crypto as long as it is given a certificate with the Step-Up option. Microsoft's SGC protocol is more efficient, but requires the server to have code that supports a new SGC exchange in addition to a certificate with the SGC extension.

In SGC, Alice's message 1 offers weak crypto, and Bob's message 2 sends his certificate, and has him choose one of Alice's weak crypto suite offers. Alice notices the SGC extension in Bob's Verisign-signed certificate, and Alice then sends a new *client-hello*, this time offering strong crypto suites. This would confuse a server that did not have special SGC code, because it wouldn't expect a second *client-hello* within the same TCP connection.

19.14. Encoding

The packet encodings in the SSL/TLS specifications are expressed in a pseudo-ASN.1 syntax that is very unpleasant to read and makes it difficult to imagine what the packets actually look like. Hopefully you will appreciate our translating them into pictures here. The problem with the ugly syntax in the spec is that not only does it give the reader a headache, but it resulted in weird formats, for instance redundant fields, and implementation bugs (e.g., see §19.14.2.4 *ClientKeyExchange*).

Since SSL/TLS runs on top of TCP, SSL/TLS can send an arbitrary-length chunk. TCP will handle breaking it up into packets and reassembling it. SSL/TLS has two layers of chunking itself, which operate somewhat independently. The unit of cryptographic protection is the **record**, which contains a header and a body, where the body is cryptographically protected if the protocol is far enough along. Records are limited to 2^{14} octets by the specification, but there are buggy implementations that send records of up to 2^{16}-1 octets so a conservative implementation should be prepared to accept records up to that size. The parts of the handshake are divided into **messages**, which also contain a header and a body. In principle, a single message could be packaged in multiple records or multiple messages could be placed in a single record. In practice, most implementations combine the two layers and put one message in each record.

It would have been simpler (and more efficient) for each message to be its own record type, and to have only a single header on each handshake message. But instead the designers chose to have four record types, with most of the handshake messages being enclosed in records of type handshake. A handshake record can contain multiple messages in it, whereas the other records are really free-standing messages. If the server is going to send a *ServerHello*, a *Certificate*, and a *ServerHelloDone* (all of which are handshake messages), these could be sent as three records, or lumped together into a single record, or two of them could be in one record and the third in another record.

The four record types are:

- 20=ChangeCipherSpec. This is really part of the handshake, and it's rather strange to have it be its own record type. The reason they did this is that after the *ChangeCipherSpec* message the data is encrypted and integrity-protected with the key just agreed upon. It would not be possible for only part of a record to be encrypted, so *ChangeCipherSpec* had to be the last thing in a record. They could have had this be a handshake message, mandating that it be the final message in the record. But instead they said that it would be its own record type.
- 21=alert (any sort of notification)
- 22=handshake (any of many messages which is part of the handshake)
- 23=application_data (encrypted and integrity-protected data sent after the handshake is complete)

The record header, which is never encrypted, is:

# octets	
1	record type
2	version number
2	length

Following the record header is the contents of that record, which might be encrypted and integrity-protected. In the case of the handshake record, the record contains handshake messages, each message starting with a 1-octet type and a 3-octet length. It's somewhat mysterious why they would choose a 3-octet length field for the messages and a 2-octet length field for the record, but in theory if there were a message that was bigger than 2^{14} octets, it could be split into several records. So the 2-octet length of the record length field does not constrain messages to be smaller than 2^{16} octets. All multi-octet integers are sent in big-endian order (also known as network byte order).

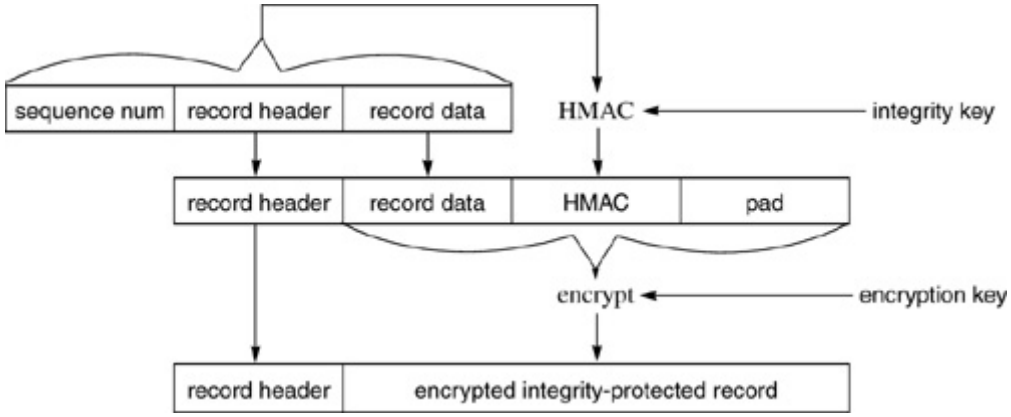
19.14.1. Encrypted Records

Records sent after a *ChangeCipherSpec* record are cryptographically protected with the negotiated cipher suite. Integrity protection is provided using HMAC based on MD5 or SHA-1. Encryption uses any of a variety of encryption algorithms, all of which are block ciphers in CBC mode except RC4, which is a stream cipher.

HMAC takes two arguments, a key and data. The key used is the session integrity key for that direction. The data on which the HMAC is based consists of a 64-bit sequence number followed by the record header followed by the data. The sequence number is not actually transmitted. It only is used in the calculation of the HMAC. It protects against ordering and replay, but does not need to be explicitly transmitted because SSL runs over a reliable protocol (TCP). If TCP failed somehow and lost, reordered, or duplicated data, the two sides would disagree about the sequence number, the integrity check would fail, and this would result in a connection failure.

If a block cipher is to be used for encryption, the DATA | HMAC is padded to a multiple of the block size. Then DATA | HMAC | padding is encrypted. If a block cipher is used, the initial IV is computed along with the key and the final block of each record is used as the IV for the next.

Figure 19-4. Cryptographically protected record format



For TLS, HMAC is computed as specified in RFC 2104. For SSLv3, the MAC is computed in a similar fashion, but is different both because the HMAC design was still evolving and they got an early draft and because a typographical error resulted in 40 octets of padding in a spot where there should have been 44. Neither of these differences is likely to affect the security, but they do complicate the documentation.

To summarize, what is transmitted is the record header, followed by an encrypted blob which, when decrypted, consists of the data followed by the HMAC, followed by padding including the padding length.

19.14.2. Handshake Messages

A handshake record (type 22) contains handshake messages. The specification allows someone to break a handshake message into pieces, and send each piece in a separate handshake record. It is also permissible for the handshake record to contain multiple handshake messages. For instance, the server might send a *ServerHello*, a *Certificate*, a *CertificateRequest*, and a *ServerHelloDone*, all within the same handshake record. The handshake messages are *ClientHello*, *ServerHello*, *ClientKeyExchange*, *Certificate*, *ServerHelloDone*, *HandshakeFinished*, *CertificateRequest*, *CertificateVerify*, and *ServerKeyExchange*.

19.14.2.1. ClientHello

# octets	
1	type=1
3	length
2	version number
32	random (R_{Alice})
1	length of session_id (or 0 if field absent)
variable	session_id
2	length of cipher suite list (in octets)
variable	sequence of cipher suites, each a 2-octet type
1	length of compression list (in octets)
variable	sequence of compression methods, each a single octet

This message contains the optional session_id (to allow a session to be resumed; see §19.5 *Session Resumption*), R_{Alice}, and Alice's list of cipher suites and compression methods.

19.14.2.2. ServerHello

# octets	
1	type=2
3	length
2	version number
32	random (R_{Bob})
1	length of session_id (or 0 if field absent)
variable	session_id
2	chosen cipher
1	chosen compression method

This message contains R_{Bob}, an optional session_id (to allow the session to be resumable), and Bob's choice of cipher and compression method.

19.14.2.3. ServerHelloDone

# octets	
1	type=14
3	length=0

This indicates that the server is finished sending its handshake messages. It is the final piece of message 2 in Protocol 19-1. It just has a type octet (=14) and a length field (3 octets of 0).

19.14.2.4. ClientKeyExchange

# octets	
1	type=16
3	length
2	length (missing in SSLv3, present in TLS)
variable	encrypted pre-master secret

This is the message in which Alice sends the pre-master secret encrypted with the server's public key. Note the extra length field. It serves no purpose with RSA or Diffie-Hellman. It was in the SSLv3 specification, but not in any implementations. TLS copied the SSLv3 spec, so left the extra length field in. As we said earlier, it is an artifact of the packet specification language they invented, and so difficult to read and visualize that it resulted not only in extra fields but implementation bugs.

Redundant length fields are a problem. At best it's extra work to set and look at them, and extra octets on the wire. And what are you supposed to do if they disagree? Also, note that the second length field is only 2 octets long and the first one is 3 octets long. If the length never needs to be larger than can be expressed in two octets, why is the other length field 3 octets?

Although the second length field is redundant in RSA and Diffie-Hellman, there are key exchange algorithms such as FORTEZZA that require multiple variable length fields to express the encrypted pre-master secret. Also, future algorithms might require more than 2^{16} octets of information. The information following the 3-octet length field is algorithm dependent, so they could have left out the 2nd length field for RSA and Diffie-Hellman, but had the data for FORTEZZA consist of (length, value) pairs.

19.14.2.5. ServerKeyExchange

# octets	
1	type=12
3	length
2	length of modulus
variable	modulus
2	length of exponent
variable	exponent
2	length of signature
variable	signature

This message is used in SSLv3 and TLS for export approval, in which the server signs a (short) ephemeral public key, signed with its long-term key. Instead of sending an X.509 certificate, it is just a signature on the new key. It is also used when the server's long-term public key can only be used for signing, for instance when the server's key is a DSS key.

19.14.2.6. CertificateRequest

# octets	
1	type=13
3	length
1	length of key type list
variable	list of types of keys (each one octet long)
2	length of CA name list
2	length of 1st CA name
variable	1st CA name
2	length of 2nd CA name
variable	2nd CA name
2	length of 3rd CA name
variable	3rd CA name
	...
2	length of last CA name
variable	last CA name

This message is sent by the server to request that the client send a certificate and authenticate. See §19.8 *PKI as Deployed by SSL*. There are several defined "key types", which not only specify the kind of key, but the kind of authentication method the user will use to authenticate. The most commonly used is type 1=RSA signing.

19.14.2.7. Certificate

# octets	
1	type=11
3	length
3	length (unnecessary field)
3	length of first certificate
variable	first certificate
variable	more (3-octet length, certificate) pairs, if more than one certificate is being sent

This message contains one or more certificates. Note this also contains a redundant length field. The second length field will always be 3 less than the first length field (since the first length field includes the second length field in the number of octets it is counting).

19.14.2.8. CertificateVerify

# octets	
1	type=15
3	length
2	length of signature
variable	signature

This message is sent by the client to prove it knows its private key. The signature (in the case of RSA, which is really all that's deployed), is a modified PKCS #1 RSA-signed quantity consisting of both the MD5 of the handshake messages and the SHA of the handshake messages. The modification to PKCS #1 is that the ASN.1 encoded digest type is omitted and the MD5 and the SHA digests are concatenated.

19.14.2.9. HandshakeFinished

# octets	
1	type=20
3	length=36 or 12
36 or 12	digest

This message ensures integrity of the exchange as well as proving knowledge of the key. In SSLv3 the digest, strangely, is a 36-octet quantity consisting of a concatenation of a 16-octet keyed MD5 hash and 20-octet keyed SHA hash of information including the handshake messages. In TLS the digest is 12 octets computed with a complex combination of MD5 and SHA.

19.14.3. ChangeCipherSpec

# octets	
1	record type=20
2	version number
2	length (set to 1)
1	ChangeCipherSpecType (set to 1)

This isn't a handshake message, but is instead its own record type. It indicates that all records following this will be protected with the ciphers agreed upon as of this message. The final field (CHANGECIPHERSPECTYPE) is mysterious. Only one value (1) has been defined, which means *change to the cipher suite we've agreed to now*. It's unclear when you would want to say anything other than that. But, if someone thinks of a new thing to say, they can define a new value for that field.

19.14.4. Alerts

An alert is sent to advise the other side of some condition. Most alerts are error messages, with a severity level of either 1=warning, or 2=fatal. The one defined alert which is not an error indication is the *closure* alert, in which one side notifies the other that it has no more data to send. This was added in SSLv3 because of a security vulnerability in SSLv2 known as the truncation attack. SSLv2 assumed the other side was finished sending data if the TCP connection closed. However, because TCP commands are not cryptographically protected, there is no way to know whether the TCP connection closed because the other side closed the TCP connection when it finished sending all its data, or an attacker sent the TCP command to close the connection.

19.15. Further Reading

Aside from the RFCs (RFC 2246 for TLS, www.netscape.com/eng/security/SSL_2.html for SSLv2, and home.netscape.com/eng/security/ssl3/index.html for SSLv3), we recommend the book *SSL and TLS: Designing and Building Secure Systems* by Eric Rescorla, published by Addison Wesley, 2000. The book goes beyond the specs in offering implementation tips and showing packet traces.

19.16. Easy Homework

1. What are the implications of SSL being implemented at layer 4, contrasted with IPsec's being implemented at layer 3?
2. If there is no integrity protection on Alice's initial message, how can an active attacker force Alice and Bob to agree on weaker crypto?
3. Compare the session resumption mechanisms in SSL and Lotus Notes.

19.17. Homework

1. As explained in §19.9 *Version Numbers*, the VERSION NUMBER field in v2 messages is in a different place from the VERSION NUMBER in v3 messages. Therefore, if a v3 client is not sure what version number the server is, it will send a v2-formatted *ClientHello* with the VERSION NUMBER field marked as 3.0, and hope that a v2 server will ignore the VERSION NUMBER and otherwise process the message. If the v3 client *does* know the server is v3, it will send a v3-formatted *ClientHello*. This explains why a v3 server must be able to receive either v2 or v3 messages, and distinguish them. Under what scenario would a v3 client not be able to predict whether it will receive a v2 message or a v3 message from a particular server?
2. Compare the performance of doing PFS as implemented in SSLv3, vs. the recommended modification in §19.13.2 *Exportability in SSLv3* vs. doing a Diffie-Hellman exchange for each connection. Assume the ephemeral key pair is of adequate length, rather than done to meet export rules. Assume also that PFS doesn't need to be "perfect", but rather the ephemeral key can change, say, every hour, and the cost of generating the key pair can be amortized over all the connection requests that occur during that hour.
3. What is the advantage, in the exportable SSLv3 case, of hashing the 40-bit secret with two non-secret values to produce a 128-bit key? How many keys would have to be tested to brute-force break a single session?