Lecture 20: PGP, IPSec, SSL/TLS, and Tor Protocols

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 29, 2016
4:00pm

Goals:

- PGP: A case study in email security
- Key management issues in PGP
- Packet-level security with IPSec
- Transport Layer Security with SSL/TLS
- **Heartbeat Extension** to the SSL/TLS protocol
- The Tor protocol for anonymized routing

CONTENTS

20.1: PROVIDING SECURITY FOR INTERNET APPLICATIONS

- As described in detail in my previous lectures, there are three fundamental aspects to providing *information security* in internet applications:
  - authentication
  - confidentiality
  - key management

- As shown in Figure 1, information security may be provided at different layers in the internet suite of communication protocols:

  - We can provide security services in the Network Layer by using, say, the IPSec protocol, as shown in part (a) of Figure 1. While eliminating (or reducing) the need for higher level protocols to provide security, this approach, if solely relied upon, makes it difficult to customize the security policies to specific applications. It also takes away the management of security from the application developer.

Four Layer Representation of the TCP/IP Protocol Stack (See Lecture 16)

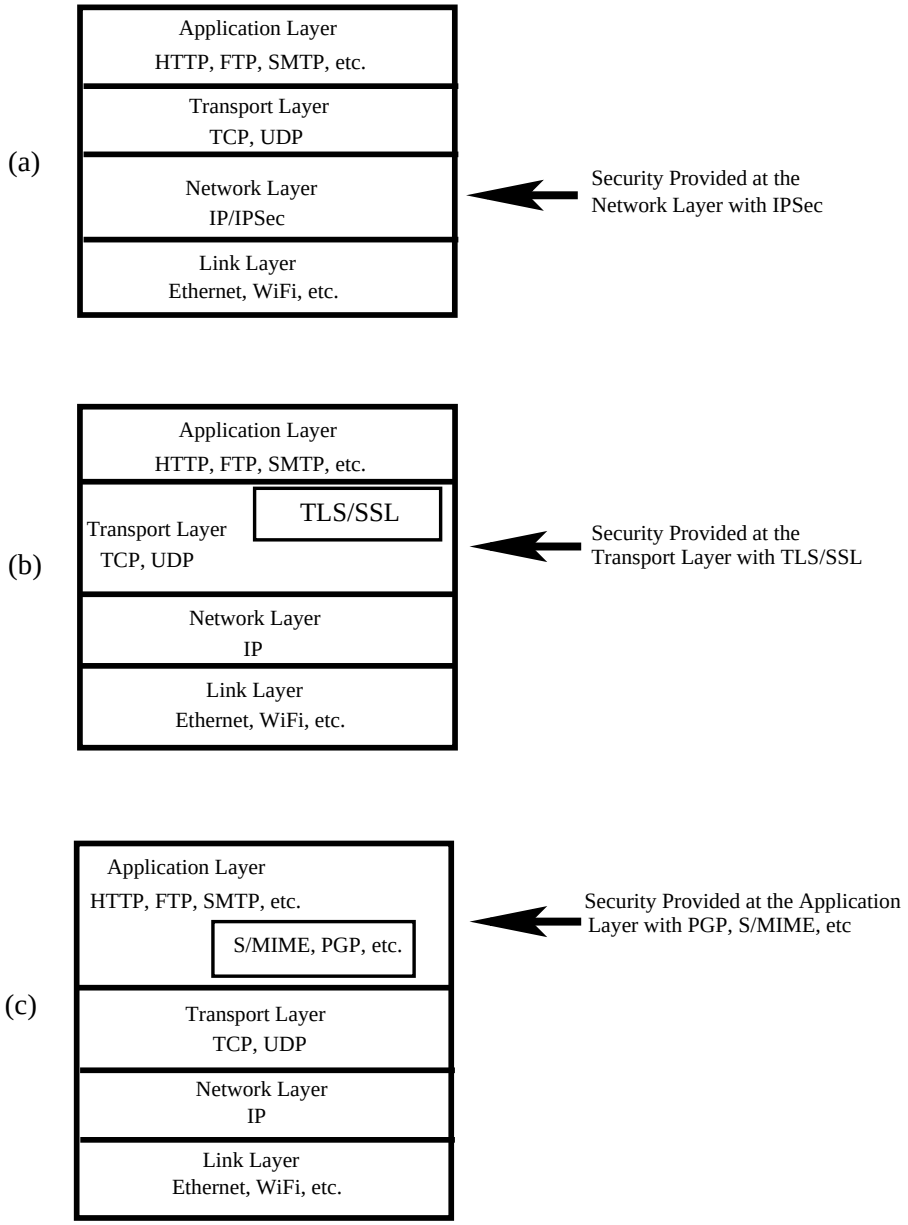


Figure 1: *Confidentiality and authentication for information security can be provided in three different layers in the TCP/IP protocol stack, as shown in this figure.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

– We can provide security in a higher layer, but still in a manner that is agnostic with regard to specific applications, by adding security-related features to TCP packets. This can be done with a Session Layer protocol like the Secure Sockets Layer (SSL/TLS). This is shown in part (b) of Figure 1. [*As stated in Section 16.2 of Lecture 16, in a 4-layer presentation of the TCP/IP protocol stack, the SSL/TLS protocol is usually placed in the Application Layer. However, again as stated in Lecture 16, more accurately speaking, the SSL/TLS protocol belongs to the Session Layer in the 7-layer OSI model of the TCP/IP stack.*] [Note that the firewall security provided by `iptables`, as presented in Lecture 18, also operates at the transport layer of the protocol stack. However, that is primarily defensive security. That is, `iptables` based firewall security is not meant for making information secure through authentication and confidentiality services.]

– We can embed security in the application itself, as shown in part (c) of Figure 1. The applications PGP, S/MIME, etc., in that figure are all security aware. [The proxy servers, as presented in Lecture 19, can also provide security at the application level. However, as with `iptables`, that is again primarily defensive security in the form of access control. It is generally not the job of the proxy servers to provide authentication and confidentiality services.]

- In each of the three different layers mentioned above, authentication can be provided by public-key cryptography (see Lecture 12) and by secure transmission of message digests or message authentication codes (see Lecture 15). [As mentioned previously in Lecture 15, authentication means **two** things: When information is received from a source, authentication means that the source is indeed as alleged in the information. Authentication also means that the information

was not altered along the way. The latter type of authentication is also referred to as maintaining data integrity.]

- Again in each of the three different layers, confidentiality can be provided by symmetric key cryptography (see Lecture 9).

- However, when public-key cryptography is used for authentication at any layer, the key-management issues in all layers can be made complicated by the fact that users are allowed to have multiple public keys.

- In this lecture, we will present PGP as an example of Application Layer security, IPSec for Network Layer security, and SSL/TLS for Session Layer security.

- About the vocabulary used in the rest of this lecture, note that the internet standards often use **octet** for a **byte** and not infrequently **datagram** for a **packet**. We will consider an octet to be synonymous with a byte and a packet to be synonymous with a datagram. [Strictly speaking, a **byte** is the smallest unit for memory addressing. A special-purpose computing device may, for example, use 6-bit bytes. For us, a **byte** will always contain 8 bits. About packets vs. datagrams, a **packet** is a generic name for the data that is kept together during transmission through a network. As discussed in Lecture 16, the IP Layer receives a TCP segment from the TCP Layer and, if the TCP segment is too long, fragments it into smaller packets that are acceptable to the routers. Before security processing can be applied, it is often necessary to reassemble these packets back into the original TCP segments. In the context of TCP/IP protocols,

most folks use **packet** to denote what is sent down by the IP Layer to the Link Layer at the sending end and what is sent up by the Link Layer to the IP Layer at the receiving end. Additionally, most folks use **TCP segment** and **datagram** interchangeably.]

20.2: APPLICATION LAYER SECURITY — PGP FOR EMAIL SECURITY

- PGP stands for Pretty Good Privacy. It was developed originally by Phil Zimmerman. However, in its incarnation as **OpenPGP**, it has now become an open-source standard. The standard is described in the document RFC 4880.

- PGP is widely used for protecting data in long-term storage. In this lecture, though, our focus is primarily on email security. [As I also mention in Lecture 22, in these days when it is so easy for your information to be stolen from your computer through malware, at the least you should keep all your personal information in a GPG encrypted file. GPG, which stands for Gnu Privacy Guard, is an implementation of OpenPGP (RFC 4880). To encrypt a file called `myinfo.txt`, all you have to do is to run a command like '`gpg --cipher-algo AES256 -c myinfo.txt`'. You will be prompted for a passphrase that is used to create the needed encryption key. This command will place its output in a file named `myinfo.txt.gpg`. You can decrypt the encrypted file at any time by calling '`gpg myinfo.txt.gpg`'. Do '`gpg -help`' for the different command line options that go with the `gpg` command. I should also mention that it is easy to use several text editors seamlessly with GPG. **IMPORTANT:** After you have used the `gpg` command in the manner indicated, make sure you delete the original file with the `srm` command that stands for "secure remove". What `srm` does amounts to wiping clean the part of disk memory that was occupied by the file you just encrypted.]

- PGP's operation consists of **five services**:

1. **Authentication Service:** Sender authentication consists of the sender attaching his/her digital signature to the email and the receiver verifying the signature using public-key cryptography. Here is an **example** of authentication operations carried out by the sender and the receiver:

   i At the sender's end, the SHA-1 hash function is used to create a 160-bit message digest of the outgoing email message. See Lecture 15 for SHA-1.

   ii The message digest is encrypted with RSA using the sender's private key and the result **prepended** to the message. The composite message is transmitted to the recipient.

   iii The receiver uses RSA with the sender's public key to decrypt the message digest.

   iv The receiver compares the locally computed message digest with the received message digest.

   The above description was based on using a RSA/SHA based digital signature. PGP also support DSS/SHA based signatures. DSS stands for **Digital Signature Standard**. [See

Section 13.6 of Lecture 13 and Section 14.13 of Lecture 14 for DSS.] Additionally, the above description was based on attaching the signature to the message. PGP also supports **detached signatures** that can be sent separately to the receiver. Detached signatures are also useful when a document must be signed by multiple individuals.

2. **Confidentiality Service:** This service can also be used for encrypting disk files. As you'd expect on the basis of the discussion in Lecture 13, PGP uses symmetric-key encryption for confidentiality. The user has a choice of three different block-cipher algorithms for this purpose: CAST-128, IDEA, or 3DES, with CAST-128 being the default choice. [Like DES, **CAST-128** is a block cipher that uses the Feistel cipher structure (see Lecture 3 for what is meant by the Feistel structure). The block size in CAST-128 is 64-bits and the key size varies between 40 and 128 bits. Depending on the key size, the number of rounds used in the Feistel structure is between 12 and 16, it being the latter when the key size exceeds 80 bits. Obviously, as you'd expect, how each round of processing works in CAST is different from how it works in DES. But, overall, as in DES, each round carries out a series of substitutions and permutations in the incoming data. **IDEA** (International Data Encryption Algorithm) is also a block cipher. IDEA uses 64-bit blocks and 128 bit keys. The cipher uses 8 rounds of processing on the input bit blocks (and an additional half round), each round consisting of substitutions and permutations.]

   – The block ciphers are used in the **Cipher Feedback Mode** (CFB) explained in Lecture 9.

– The 128-bit encryption key, called the **session key**, is generated for each email message separately.

– The session key is encrypted using RSA with the receiver's public key. Alternatively, the session key can also be established using the **ElGamal** algorithm. (See Section 13.6 of Lecture 13 for the ElGamal variant of the Diffie-Hellman algorithm.)

– What is put on the wire is the email message after it is encrypted first with the session key and then with the receiver's public key.

– If confidentiality and sender-authentication are needed simultaneously, a digital signature for the message is generated using the hash code of the message plaintext and appended to the email message before it is encrypted with the session key. (See the previously shown PGP's authentication service.)

3. Compression Service: By Default PGP compresses the email message after appending the signature but before encryption. This makes long-term storage of messages and their signatures more efficient. This also decouples the encryption algorithm from the message verification procedures. Compression is carried out with the ZIP algorithm.

4. **E-Mail Compatibility Service:** Since encryption, even when it is limited to the signature, results in arbitrary binary strings, and since network message transmission is character oriented, we must represent binary data with ASCII strings. PGP uses **Base64** encoding for this purpose. [Base64 encoding is referred to as Radix 64 encoding in the PGP documentation. As you should already know from our previous references to this form of encoding multimedia objects, it has emerged as probably the most common way to transmit binary data over a network. To briefly review Base64 again (at the risk of beating a dead horse), it first segments the bytes of the object that needs to be encoded into 6-bit words. The $2^6 = 64$ different possible 6-bit words are represented by printable characters as follows: The first 26 are mapped to the uppercase letters A through Z, the next 26 to the lowercase a through z, the next 10 to the digits 0 through 9, and the last two to the characters '/' and '+'. This causes each triple of adjoining bytes to be mapped into four ASCII characters. The Base64 character set includes a 65^{th} character, '=', to indicate how many characters the binary string is short of being an exact multiple of 3 bytes. When the binary string is short one byte, that is indicated by terminating the Base64 string with a single '='. And when it is short two bytes, the termination becomes '=='.]

5. **Segmentation Service:** For long email messages (these are generally messages with attachments), many email systems place restrictions on how much of the message will be transmitted as a unit. For example, some email systems segment long email messages into $50,000$ byte segments and transmit each segment separately. PGP has built-in facilities for such segmentation and re-assembly.

- Figure 2 shows the three different modes in which PGP can be used for secure email exchange. The top diagram is for when only

authentication is desired, the middle when only confidentiality is needed, and the bottom when both are wanted. The notation *R*64 in the figure is for conversion to Radix 64 ASCII format (which, as already mentioned, is the same as what is accomplished by Base-64 ecoding).

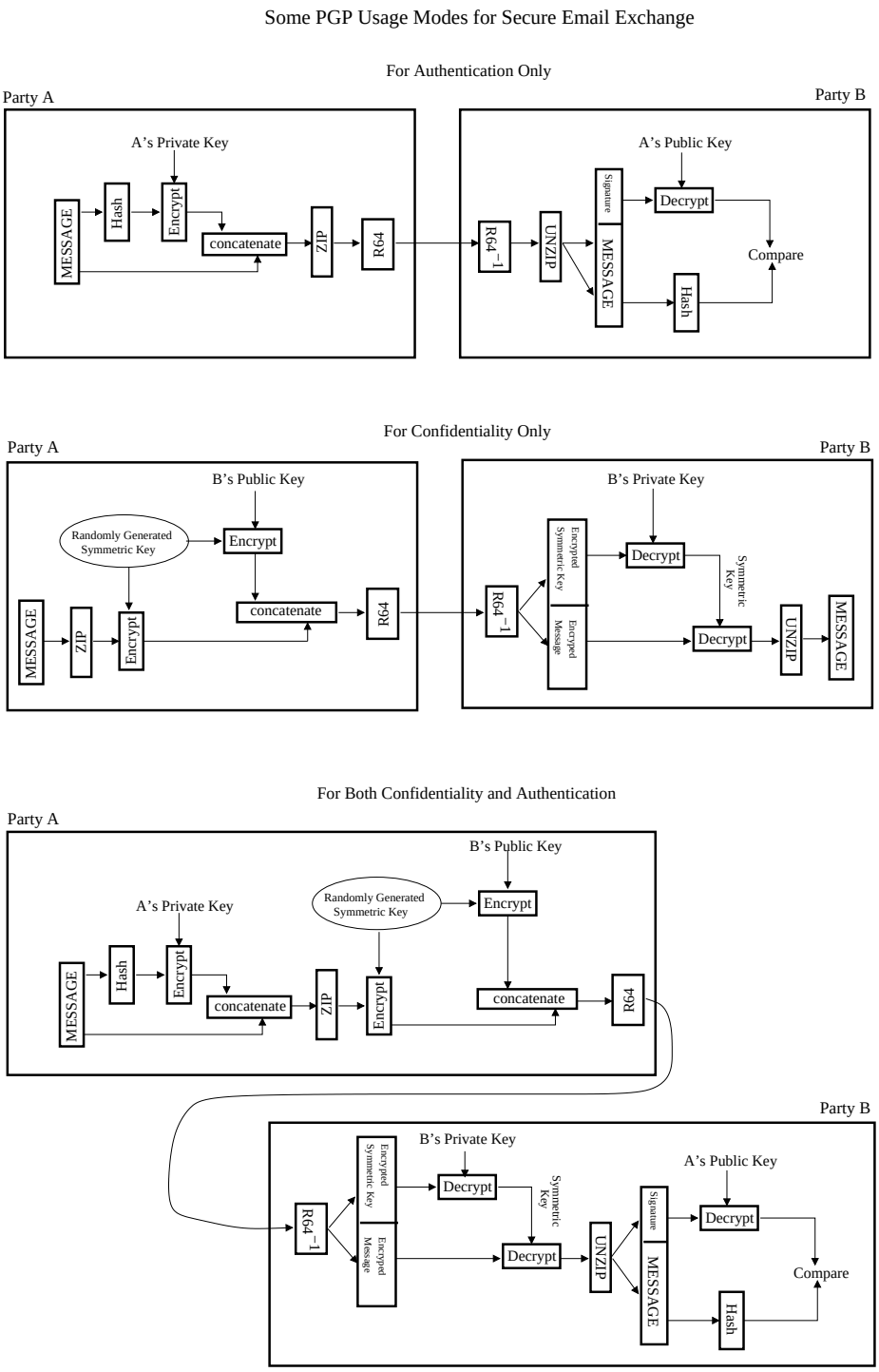


Figure 2: *The three different modes in which PGP can be used for secure email exchange.* (*This figure is from "Computer and Network Security" by Avi Kak*)

20.2.1: Key Management Issues in PGP and PGP's Web of Trust

- As you have already seen, public key encryption is central to PGP. It is used for authentication and for confidentiality. A sender uses his/her private key for placing his/her digital signature on the outgoing message. And a sender uses the receiver's public key for encrypting the symmetric key used for content encryption for ensuring confidentiality.

- We can expect people to have multiple public and private keys. This could happen for a number of practical reasons. For example, an individual may wish to retire an old public key, but, to allow for a smooth transition, may decide to make available both the old and the new public keys for a while.

- So PGP must allow for the possibility that the receiver of a message may have stored multiple public keys for a given sender. This raises the following procedural questions:

  – Let's say PGP uses one of the public keys made available by the recipient, how does the recipient know which public key it is?

– Let's say that the sender uses one of the multiple private keys that the sender has at his/her disposal for signing the message, how does the recipient know which of the corresponding public keys to use?

- Both of these problems can be gotten around by the sender also sending along the public key used. The only problem here is that it is wasteful in space **because the RSA public keys can be hundreds of decimal digits long.**

- The PGP protocol solves this problem by using the notion of a relatively short **key identifiers** (key ID) and requiring that every PGP agent maintain *its own list* of paired private/public keys in what is known as the Private Key Ring; and a list of the public keys *for all its email correspondents* in what is known as the Public Key Ring. Examples of private and public key rings are shown in Figure 3.

- The keys for a particular user are uniquely identifiable through a combination of the user ID and the key ID.

- The key ID associated with a public key consists of its least significant 64 bits. [This way the key ID is always just 8 bytes long. The entries for the keys and their IDs shown in Figure 3 are in hex. Each hex string begins with the least significant byte. Therefore, the sixteen hex characters in a key ID will always be the same as the first sixteen hex

Private Key Ring Table:

User ID	Key ID	Public Key	Encrypted Private Key	Timestamp
kak@abc.com	EA132....43	EA132....43....A21	34ABF23.....A9	041908−11:30
⋮	⋮	⋮	⋮	⋮

Public Key Ring Table:

User ID	Key ID	Public Key	Producer Trust	Certificate	Certificate Trust	Key Legitimacy	Timestamp
kak@abc.com	EA132....43	EA132....43....A21	Full	−−−	−−−	Full	041908−11:30
zaza@foo.com	132AB....02	132AB....02....23A	Full	−−−	−−−	Full	−−−
toto@bar.com	231DA....02	231DE....02....33B	Full	Zaza's	Full	Full	−−−
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: *Examples of the public and the private key rings for a user.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

characters of the public key. The public key ring table always include entries for the public keys of the owner of the public key ring despite the fact that the same information is contained in the private key ring table for the owner.]

- Going back to private key ring shown in Figure 3, for security reasons, PGP stores the private keys in the table in an encrypted form so that the keys are only accessible to the user who owns them. [PGP can use any of the three block ciphers at its disposal, CAST-128, IDEA, and 3DES, with CAST-128 serving as the default choice, for this encryption. The encryption algorithm asks the user to enter a passphrase. The pass-phrase is hashed with SHA-1 to yield a 160-bit hash code. The first 128 bits of the hash code are used as the encryption key by the CAST-128 algorithm. Both the passphrase and the hash code are immediately discarded.]

- With regard to the public key ring shown in Figure 3, the fields **Producer Trust**, **Key Legitimacy**, **Certificate**, and **Certificate Trust** are to assess how much trust to place in the public keys belonging to other people. [If A has B's public key in the ring, but the key really belongs to C (in the sense that C is the legitimate owner of the corresponding private key), then B can send messages to A and forge C's signature, assuming that B has also stolen C's private key. A would think a message was from C whereas it is really from B and any encrypted messages from A to C would be readable by B.]

- How to designate trust is implementation dependent. In the rest of the explanation here, we will use the symbolic values **full**, **partial**, and **none** for expressing the degree of trust.

- A unique feature of PGP is its own notion of a "certificate authority" for authenticating the binding between a public key and its owner. This notion is based on PGP's **web of trust** that is a *bottom-up* approach to establishing trust for authentication. [This is to be contrasted with the *top-down* approaches of Public Key Infrastructure (PKI) that we talked about in Lecture 13. As presented in that lecture, PKI is based on Certificate Authorities (CA) that are arranged in a strict hierarchy for establishing trust. In PKI, the trust can only flow downwards from the root node (that must always be trusted implicitly) to the CAs at the other nodes that descend from the root node.]

- In PGP's **web of trust**, a user's public key can be signed by any other user. See Lecture 13 for what is meant by signing a public key. For example, in user `kak`'s public key ring shown in Figure 3, `toto`'s public-key was signed by `zaza`. The same table shows that the user `kak` fully trusts `zaza` presumably because `zaza` handed its public key to `kak` directly (say, over the phone). Because the fully-trusted `zaza` endorses the new user `toto`'s public key, `toto` also becomes a fully-trusted email correspondent for the user `kak`. For proper operation of the web of trust, it is important that everyone who signs a public key for another submits the signature to a central key server.

- Because there is no hierarchy of trust in PGP, it is possible that a user will receive two different certificates for a new email correspondent, say one that the receiver will trust fully and the other that the receiver may trust only partially. Whether or not to trust such a potential email correspondent is up to the receiver of

the certificates. [As explained in Lecture 13, a certificate is simply a public key digitally signed by its endorser through his/her private key.]

- The entry stored in the **Public Key** field is where the public key is stored.

- The entry in the **Producer Trust** field of the Public Key Ring table indicates the extent to which the owner of a particular public key can be trusted to sign other certificates. This will generally be one of three values: **full**, **partial**, or **none**.

- The **Certificate** field holds the certificate(s) that authenticates the entry in the public key field. The third row in the Public Key Ring in Figure 3 shows that `toto` public key was signed by `zaza`. That is, `zaza` supplied the certificate that authenticated `toto`'s public key. In other words, `zaza` used its private key to digitally sign `toto` public key and sent that signed document to `kak`. The entry in the **Certificate** field holds that certificate.

- The **Certificate Trust** field indicates how much trust a user wants to place in the entry in the **Certificate** field.

- For a given public key, the value for the **Key Legitimacy** field is automatically derived by PGP from the value(s) stored for the **Certificate Trust** field(s) and a predefined weight for each

symbolic value for certificate trust. Recall that an individual may receive multiple signed certificates for a new potential email correspondent from others in a web of trust.

- Figure 4, based on a figure in Chapter 15 of "Cryptography and Network Security" by William Stallings, shows the general format of a PGP message. As the figure shows, a PGP message consists of three components: a session key component, a signature component, and the actual email message itself. Perhaps the only unexpected entry is the "leading two bytes of message digest." This is to enable the recipient to determine that the correct public key (of the sender) was used to decrypt the message digest for authentication. These two bytes also serve as a 16-bit **frame check sequence** for the actual email message. The message digest itself is calculated using SHA-1.

- **In modern usage of PGP**, creation of the web of trust is facilitated by the availability of free publicly available PGP Keyservers (v. 7.0) at various places around the world. In order to upload your key to such a server, one typically creates a GPG (Gnu Privacy Guard) key though the following steps: [As mentioned at the beginning of Section 20.2, Gnu Privacy Guard (abbreviated GnuPG or GPG) is an implementation of the OpenPGP standard (RFC 4880).]

  – create a new `.gnupg` directory at the top level of your home directory.

Party A sends a PGP message to party B

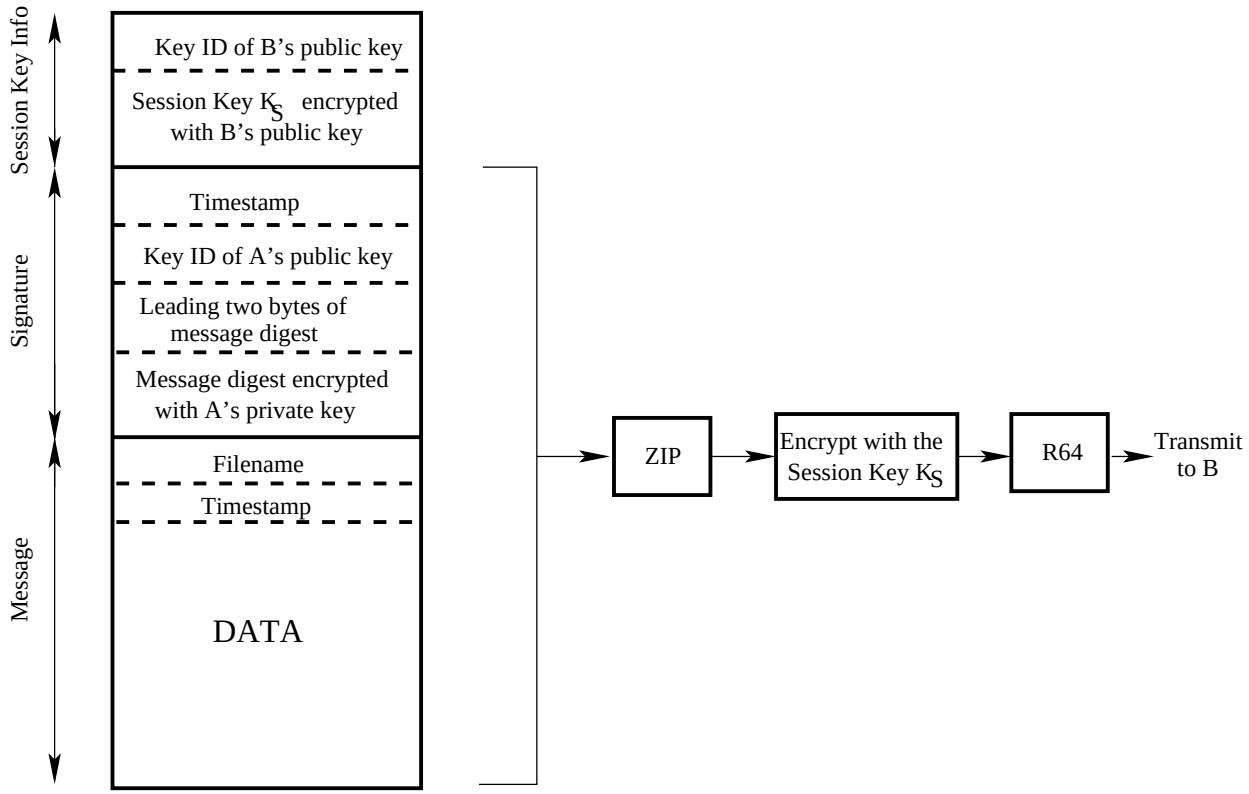


Figure 4: *The general format of a PGP message.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

– Using the following call, execute the `gpg` key generation command to create a public/private key pair:

```
gpg --gen-key
```

You will be prompted for what type of keys you want. The default is "RSA and RSA". Go with the default. You will be prompted for the size of the modulus for the RSA key. The default is 2048. Go with the default. You will also be prompted for when the key should expire. I went for the default, as indicated by '0', which stands for "keys do not expire". Subsequently, you will be prompted for what User-ID to use to identify your key. The User-ID is a concatenation of your "Real Name', a "Comment", and your email address. I left out the comment and went with "Avi Kak ⟨kak@purdue.edu⟩" for the User-ID. Finally, you'll be prompted for a passphrase to protect your key.

– After you have supplied the information mentioned above, `gpg` will create a key pair for you — assuming it has access to sufficient entropy to create a true random number of the size commensurate with the size of modulus for your key. [See Section 10.9 of Lecture 10 on the topic of "Software Entropy Sources". Also see Section 10.9.2 of the same lecture on EGD (Entropy Gathering Daemon) that deposits a Unix socket named '`entropy=`' in your `.gnupg` directory though which `gpg` gathers the entropy it needs for random number generation.] If the entropy found is insufficient, you will be asked to make mouse movements and random keyboard entries for increasing the entropy.

– After the keys are generated, `gpg` will output a 40-character "Key Fingerprint". Save it at a safe place. Your "KeyID" consists of the last 8 characters of the "Key Fingerprint". Save your "KeyID" also at a safe place.

– The public and private keys that are generated are deposited in the files `pubring.gpg` and `secring.gpg` of the `.gnupg` directory. There is another file created in this directory that is called `trustdb.gpg`. This is the file that keeps the trust database I talked about earlier.

– Your final step is to export your public key to one of the worldwide PGP keyservers. Exporting to one automatically broadcasts it to all other such servers. The most popular keyserver in the US appears to pgp.mit.edu. You can upload your public key to this server by

```
gpg --keyserver  pgp.mit.edu  --send-keys  your_8_char_KeyID
```

– If you have questions about the uploading of the keys to the PGP keyserver mentioned above or, perhaps, about possibly deleting of the keys you have uploaded there, visit the FAQ at `http://pgp.mit.edu/faq.html`.

20.3: IPSec – PROVIDING SECURITY AT THE PACKET LAYER

- A more broad-based approach to security consists of providing authentication, confidentiality, and key management at the level of IP packets (the Packet Layer or the Network Layer).

- When security is implemented at the Network Layer in the TCP/IP protocol, it covers all applications running over the network. That makes it unnecessary to provide security separately for, say, email exchange, running distributed databases, file transfer, remote site administration, etc. This, one could argue, spares the application-level programs the computational overhead of having to provide for security. The largest application of IPSec is in Virtual Private Networks (VPN). A VPN is an overlay network that allows a group of hosts that may be widely scattered in the internet to act as if they were in a single LAN.

- IP-level authentication means that the source of the packet is as stated in the packet header. Additionally, it means that the packet was not altered during transmission. **IP-level authentication is provided by inserting an Authentication**

Header (AH) into the packets. Stated simply, the AH stores a hash value for those portions of a packet that are expected to stay invariant during its transmission from the source to the destination. The receiver can compute a hash from the same fields and compare his/her hash to the hash in the AH associated with the packet.

- IP-level confidentiality means that third-party packet sniffers cannot eavesdrop on the communications. **IP-level confidentiality is provided by inserting an Encapsulating Security Payload (ESP) header into the packets.** ESP can also do the job of the AH header by providing authentication in addition to confidentiality.

- **IPSec** is a specification for the IP-level security features that are built into the IPv6 internet protocol. These security features can also be used with the IPv4 internet protocol. [To briefly review again the difference between IPv4 and IPv6, in addition to the built-in security achieved with IPSec, the main features of IPv6 is its much larger address space. The older and much more widely used IPv4 supports 4.3×10^9 addresses, IPv6 supports 3.4×10^{38} addresses. (The population of the earth is only (roughly) 6×10^9.) It is interesting to note that because of the DHCP protocol, which allows IP addresses to be allocated dynamically, and NAT, which as explained in Lecture 18 allows for network address translation on the fly, the general concern about the world running out of IPv4 addresses has subsided a bit. It is also interesting to note that even though IPv6 has now been around for roughly ten years, it still accounts for only a tiny fraction of the live addresses in the internet. As mentioned in Lecture 16, DHCP stands for the Dynamic Host Configuration Protocol. And, as mentioned in Lecture 18, NAT, which stands for Network Address Translation, allows all the computers in a LAN to access the internet using a single public IP address. NAT is achieved by the router rewriting the

source and/or destination address in the IP packets as they pass through.]

- IPSec is used in two different modes: the **Transport Mode** and the **Tunnel Mode**:

  - The **Transport Mode** is the regular mode for packets to travel from a source to its destination in a network — except for the fact that the two endpoints must carry out the security checks on the packets on the basis of the information contained in the authentication header.

  - With regard to the **Tunnel Mode**, the main point here is that the source and the destination endpoints for a given packet stream may not have the ability or the resources to carry out the security checks on the packets. So a source must route the packets to a designated location — let's call it P — in the network for inserting the authentication and/or ESP headers. If the originally intended destination also is not able to carry out the security checks on the packets, P may need to send the packets to another designated location — let's call it Q — that is in the "vicinity" of the actual destination for the packet stream. The host at Q can then carry out the security verification on the basis of the information in the security headers inserted by P and send the packets thus verified to their true destination. P is sometimes referred to as the encapsulator and Q as the decapsulator. The points P and Q define the two endpoints of what's referred to as a tunnel.

- Here is a good question regarding the tunnel mode: How does the source of an IP stream send its packets to the designated point P mentioned above? For the answer, the source can use the IP-

in-IP protocol (RFC 2003) for that purpose. More on that in the red note that follows. . [Encapsulating the original IP header inside a new IP header finds applications even outside the security context. For example, a networking app on a mobile device may want to send the packets to a billing host before they are actually sent to their real destination. Since the IP protocols do not make it easy to specify the routing at the source, an alternative is to use the notion of IP-in-IP, meaning encapsulating the IP header that has the actual source and destination fields with another IP header that first sends the packet to a designated location. The outter IP header is ripped off at that location and the original IP packet sent onwards to its originally intended destination. You can read more on IP-in-IP in the standards document RFC 2003. The protocol number for IP-in-IP is 4.]

- IPSec includes **filtering capability** so that only specified traffic need be subject to security processing. In other words, only those packets that are deemed to be security-sensitive need to be further processed for authentication, confidentiality, etc.

- To summarize, if you want to use IPSec for just authentication of the sender/receiver information that is placed in the IP headers, and if the two endpoints of a communication link are able to their own authentication processing, you will use IPSec in the Transport Mode with just the additional AH headers. On the other hand, if the endpoints cannot do their own authentication, you will have to use IPSec in the Tunnel Mode.

- And if you want to use IPSec for confidentiality (as provided by encryption), you'll need to the ESP headers (with or without the AH headers since the ESP headers can also carry out authenti-

cation). Again, if the two endpoints can do their own security processing, you will use IPSec in the Transport Mode. Otherwise, you'll use IPSec in the Tunnel Mode.

20.3.1: IPv4 and IPv6 Packet Headers

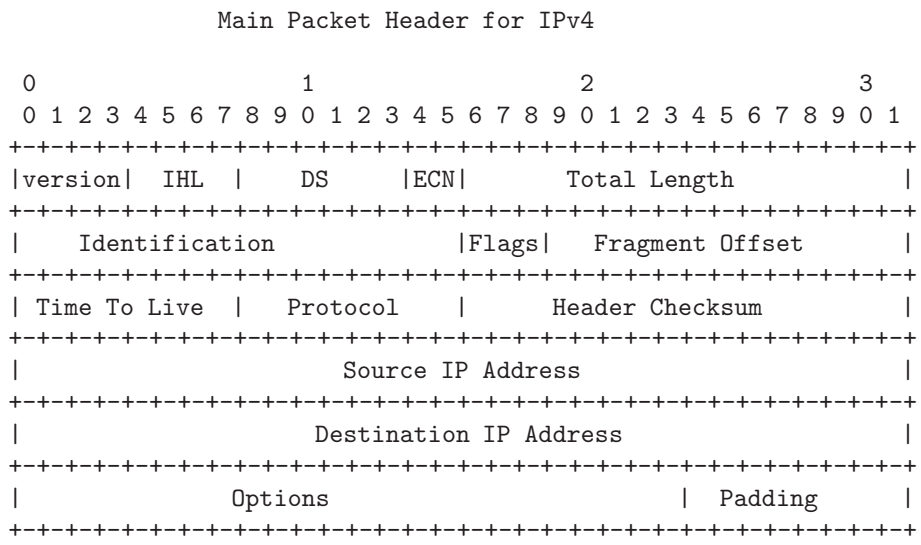
- Before we can talk about the extension headers used for IPSec, it's good to review the IPv4 and IPv6 headers. Although you have already seen these headers in Lecture 16, they are included here again for your reading convenience. IPSec security features are implemented **as extension headers** that follow the main IP header in an IP packet.

- With regard to the IPv4 header shown in Figure 5, the `Total Length` field is a 16-bit word, designates the total length of the overall packet (including the data payload) in bytes. (Therefore, the maximum size of an IPv4 packet is 65,536 bytes.) The `Identification`, `flags`, and the `Fragment Offset` fields hold values that are assigned by the sender to help the receiver with the re-assembly of the IP fragments back into an IP datagram. The `Time to Live` field, specified by 8 bits, is subtracted by 1 for each pass through a router. The `Source Address` and the `Destination Address` are each represented by 32 bits.

- The `Protocol` field of the IPv4 header **plays an important role in grafting IPSec onto IPv4.** Ordinarily this field indicates the next higher level protocol in the TCP/IP stack that is responsible for the contents of the data field of the IP packet. [Each protocol (such as the TCP protocol) has a number assigned to it. It is this number that is

stored in the `Protocol` field. For example, the number 6 represents the TCP protocol.] When IPSec is used with IPv4, this field contains the integer value that represents the security header to follow the main header. For example, the integer 50 represents the ESP header that is used for encryption services in IPSec. Therefore, if the next header is the ESP header, number 50 will be stored in the `Protocol` field. Along the same lines, the number 51 represents the AH protocol that is used for authentication services. **We will talk shortly about AH and ESP protocols.**. [Lecture 16 provides additional information on the IPv4 header.]

- For the IPv6 header shown in Figure 5, it has a fixed length of 40 bytes. What makes IPv6 headers more versatile is that the header of a packet can be followed by **extension headers**. The main header and any extension headers are linked by the `Next Header` field consisting of 8 bits. The extension headers of interest to us are the **Authentication Header** and the **Encapsulating Security Payload Header**. The `Source Address` and the `Destination Address` are each represented by 128 bits.



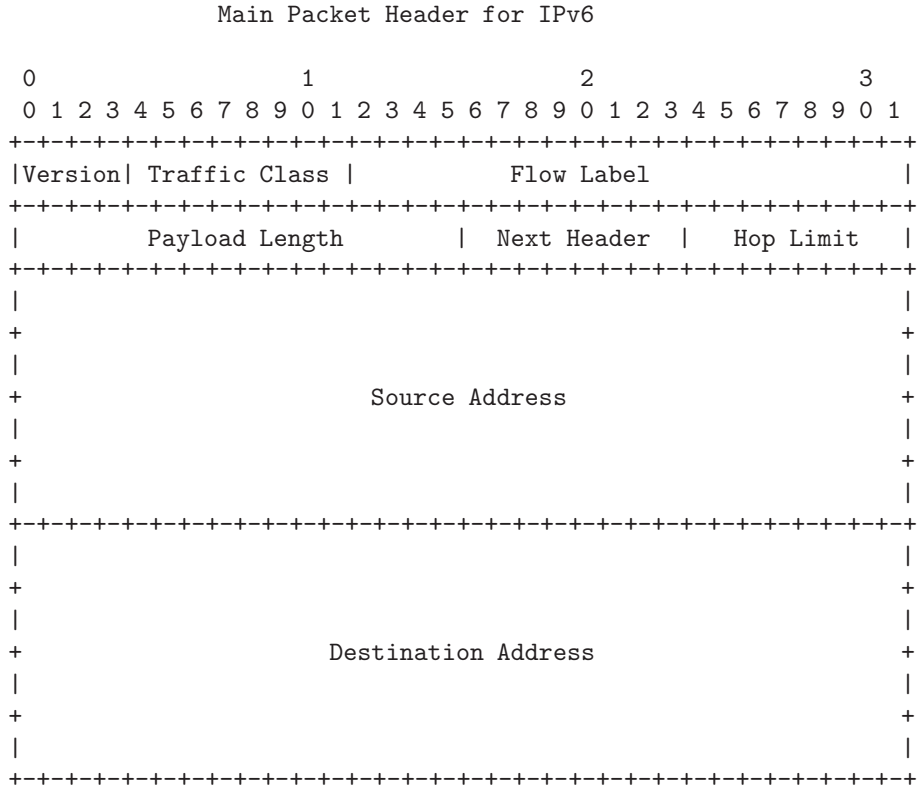


Figure 5: *The IP Headers for the IPv4 and the IPv6 protocols.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

20.3.2: IPSec: Authentication Header

- Figure 6 shows the **Authentication Header** (AH).

- In the Transport Mode of IPSec, the AH header is inserted right after the IP header in both the IPv4 and the IPv6 protocols. The second packet layout in Figure 7 illustrates the position of the AH header for IPv4 in the transport mode. And the second packet layout in Figure 8 illustrates the position of the AH header for IPv6 in the transport mode. The regular packet layouts in IPv4 and IPv6 are shown in the topmost packet layouts in the two figures.

- To elaborate, when no AH header is used, an IPv4 packet may look like

```
original IP  |   TCP header   |  Data
  header     |                |
```

- When the AH header is included, an IPv4 packet looks like

```
original IP  |  AH  |  TCP header   |  Data
  header     |      |               |
```

- With IPv6, since it allows for various sorts of extension headers, under ordinary circumstances a packet is likely to look like:

```
original IP    |   extension hdrs   | TCP header   | Data
  header       |     if present     |              |
```

- However, when the AH header is included in the Transport Mode, an IPv6 packet will looks like

```
original IP    | AH  | other extension  |   TCP header   | Data
  header       |     |      headers     |                |
```

- Referring to Figure 6, the `Payload Length` field specifies the length of the AH in 32-bit word, minus the integer 2.

- Again referring to Figure 6, the `Security Parameter Index` (SPI) field, a 32-bit value, establishes the **Security Association** (SA) for this packet. The Security Association for a packet is a grouping of the security parameters needed for authentication. These parameters may involve a public key identifier, an initialization vector identifier, an identifier for the hashing algorithm used, etc., used for authentication. The Security Parameter Index along with the source IP address is used to establish the Security Association of the sending party.

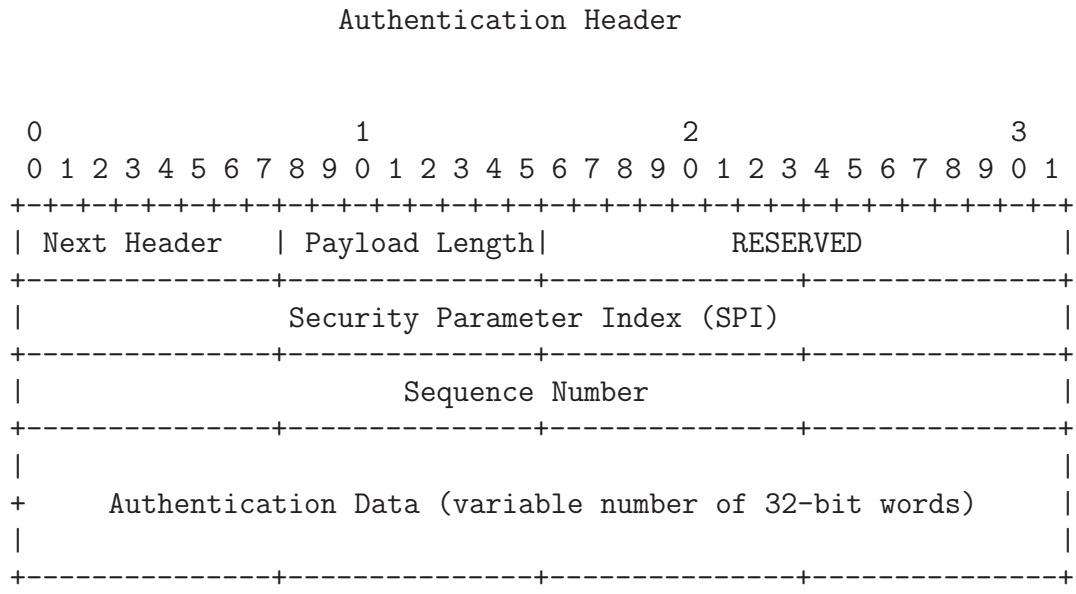


Figure 6: *The IPSec Authentication Header* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

- The `Sequence Number` field, a 32-bit integer, is a monotonically increasing number for each packet sent to prevent replay attacks. [The important point here is that for each SPI as defined above, only one packet can have a given sequence number. So if an adversary were to capture some of the IP packets and re-transmit them (say, repeatedly) to the destination (for, say, mounting a DoS attack), the destination IP engine would detect that there was a problem when it starts receiving multiple packets with the same sequence number for the same value of SPI. Since the `Sequence Number` field is only 32 bits wide, obviously the largest value permissible for this field is $2^{32} - 1$. If the sender needs to go past this number for a given transmission, the sender must zero out the `Sequence Number` field and, at the same time, change the value of SPI.]

- The variable length `Authentication Data Field` holds the MAC (Message Authentication Code) of the packet calculated with either the SHA-1 hash function or the HMAC algorithm. See Lecture 15 for what the acronyms MAC, HMAC, and SHA stand for.

- The MAC is calculated over the IP header fields that do not change in transit, obviously including the source and the destination IP addresses, the AH header (but without the Authentication Data since it will be the output of the MAC algorithm), and the **inner IP packet** for establishing authentication in the tunnel mode.

- The receiver calculates the MAC value over the appropriate fields of the packet and compares it with the value that is stored in the

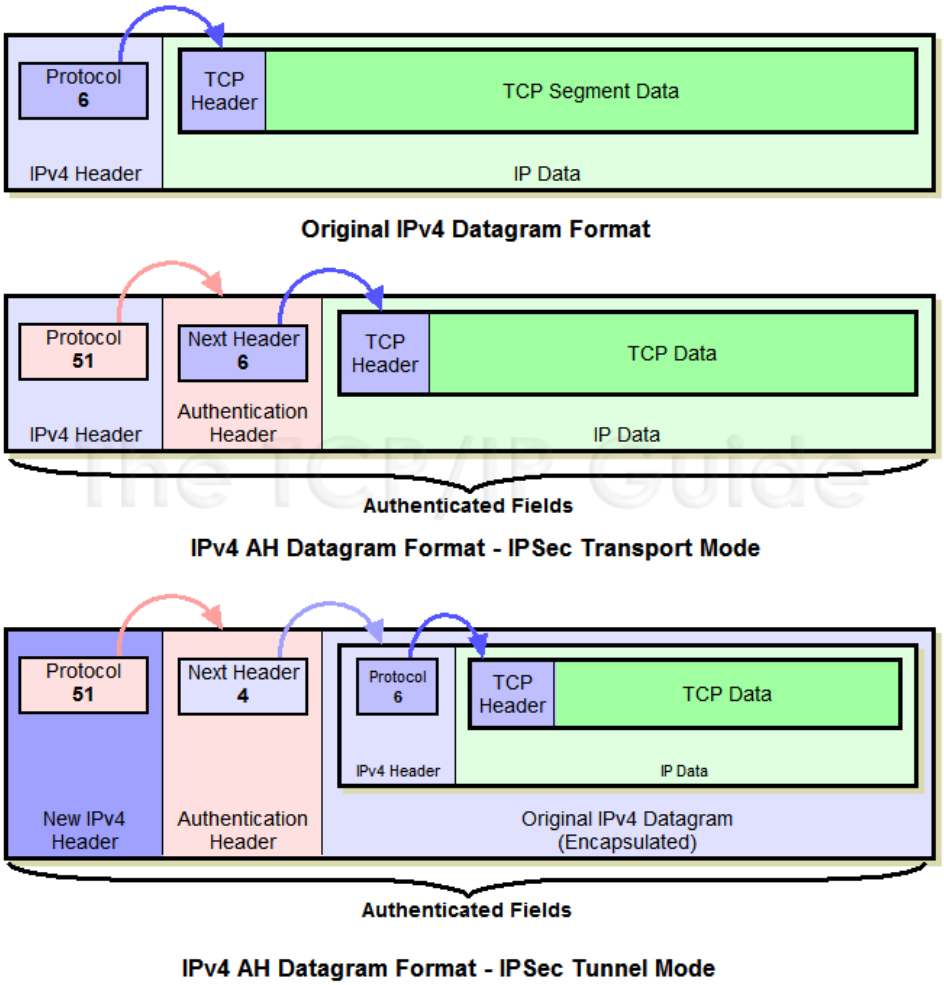
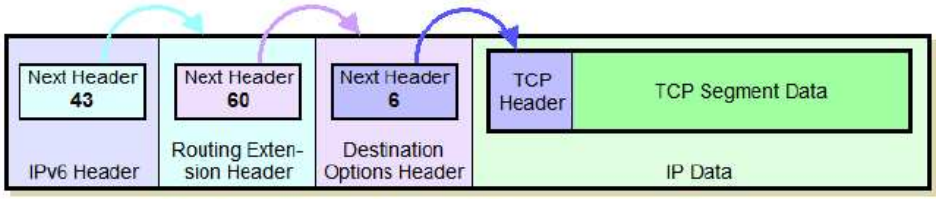
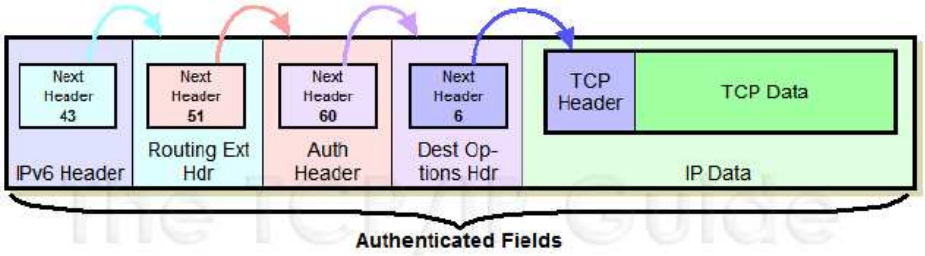


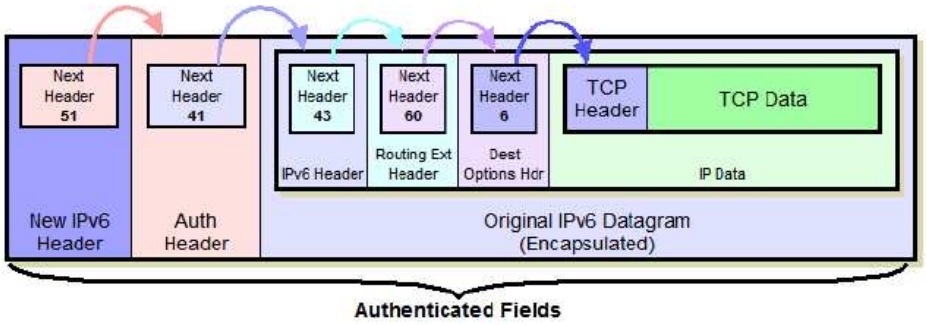
Figure 7: *The relationship between how an IPv4 packet is laid out without and with the Authentication Header, in the Transport Mode and in the Tunnel Mode.* (*This figure is from http://www.tcpguide.com*)



Original IPv6 Datagram Format (Including Routing Extension Header and Destination-Specific Destination Options Extension Header)



IPv6 AH Datagram Format - IPSec Transport Mode



IPv6 AH Datagram Format - IPSec Tunnel Mode

Figure 8: *The relationship between how an IPv6 packet is laid out without and with the Authentication Header, in the Transport Mode and in the Tunnel Mode.* (*This figure is from http://www.tcpguide.com*)

`Authentication Data` field. If the two values do not match, the packet is discarded.

- The bottom-most packet layouts in Figures 7 and 8 are for the case when AH is used in the Tunnel Mode, the former for IPv4 and the latter for IPv6. Note the word "encapsulated" in these packet layout diagrams means IP-in-IP sort of encapsulation — similar to what is described in RFC 2003. Recall what was mentioned earlier about the need for the tunnel mode: This mode is used when the source and the destination endpoints of a communication link are not able to do their own authentication processing.

20.3.3: IPSec: Encapsulating Security Payload (ESP) and Its Header

- The ESP (Encapsulating Security Payload) protocol (RFC 4303) is used for providing encryption services in IPSec.

- Figure 9 shows the layout of the header for the ESP protocol and the payload that follows the header. The header itself is just the first eight bytes. That is followed by the payload that consists of the encrypted information that needs to be transmitted. Finally, you have the optional authentication data. The whole thing is commonly referred to by the acronym ESP. [The word "encapsulation" in ESP is not be confused with our use of the same word when describing the use of AH in the tunnel mode. The word encapsulation there is more in the sense of the IP-in-IP protocol as described in RFC 2003.]

- Note that when IPSec uses the ESP header, its payload swallows up the TCP segment in the original IP packet. The encrypted version of the TCP segment is in the "Encrypted Payload Data" portion of the ESP payload. The receiving endpoint must obviously decrypt this payload in order to extract the original TCP segment.

- While ESP may be used to provide the same services as the AH header, **its main purpose is to provide confidentiality**

through encryption. ESP may be applied alone or in conjunction with the AH header. [More generally, though, ESP can be used to provide confidentiality, data origin authentication, limited traffic flow confidentiality, and so on, depending on the options selected through the value stored in the `Security Parameter Index` (SPI) field. This value must be between 1 and 255.]

- In the Transport Mode, as shown in the second packet layout in Figures 10 for IPv4 and in the second packet layout in Figure 11 for IPv6, the `Encrypted Payload Data` field, of variable length, is the encrypted version of the TCP segment (meaning the TCP header plus the data payload of the TCP segment) that would ordinarily follow the IPv4 header. So that the value of the `Next Header` field that you see at the bottom would contain number 6 and point backwards to the main content of the `Encrypted Payload Data`. It is interesting to note that an adversary would not be able see even the `Next Header` field since it is a part of what stays encrypted in an ESP packet.

- Note the role played by the fields `Padding` and `Pad Length`. Padding is meant to take care of the fact that the length of the encrypted segment would ordinarily be a multiple of the block size used for encryption with symmetric key cryptography. Let's say the block size is 1024 bits (128 bytes), then the entire encrypted portion, meaning the ESP payload, would be a multiple of 128 bytes. As to how much padding is used is stored in the field `Pad Length`. Padding must ensure that the ciphertext ends on a 4-byte boundary.

- Before encryption, an **ESP Trailer** is appended to the data to be encrypted. As shown in Figure 9, the payload (meaning the TCP/UDP message in the transport mode or the encapsulated IP datagram in the tunnel mode) and the ESP Trailer are both encrypted, but the eight-byte ESP Header is not.

- Whereas in the Transport Mode, ESP achieves confidentiality by placing in its `Encrypted Payload` an encrypted version of the entire TCP segment, in the Tunnel Mode (see the bottom-most packet layouts in Figures 10 and 11), the payload contains an encryption of the entire IP packet.

- In the Tunnel Mode, we still have the same 8-byte ESP header that you see in Figure 9. But now the `Encrypted Payload` is obtained by encrypting the entire IP packet along with the padding and the ESP trailer as before. Obviously, now you would need a new IP header for the destination of the tunnel transmission.

- The `Authentication Data` field attached at the very end of what you see in Figure 9 consists of the MAC value of the ESP packet. In the context of IPSec, this value is known as the `Integrity Check Value`.

- ESP's authentication scheme can be used either independently of the AH header or in conjunction with it.

- If the optional ESP authentication is used, the authenticator is calculated over the entire ESP datagram. This includes the ESP Header, the payload, and the trailer.

- ESP's authentication service is similar to what is provided by AH.

ESP Protocol Header and the ESP Payload

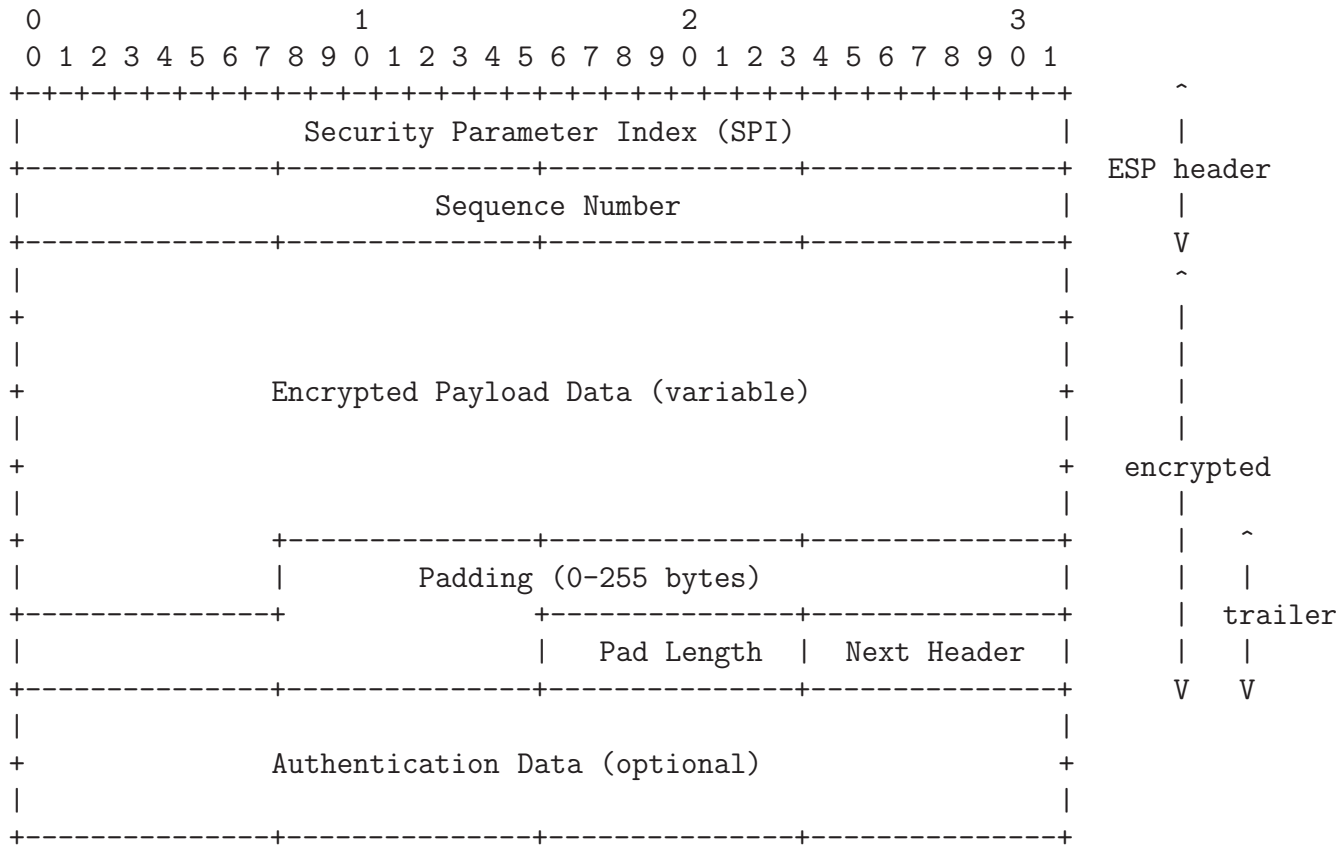


Figure 9: *ESP Protocol Header and the ESP Payload* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

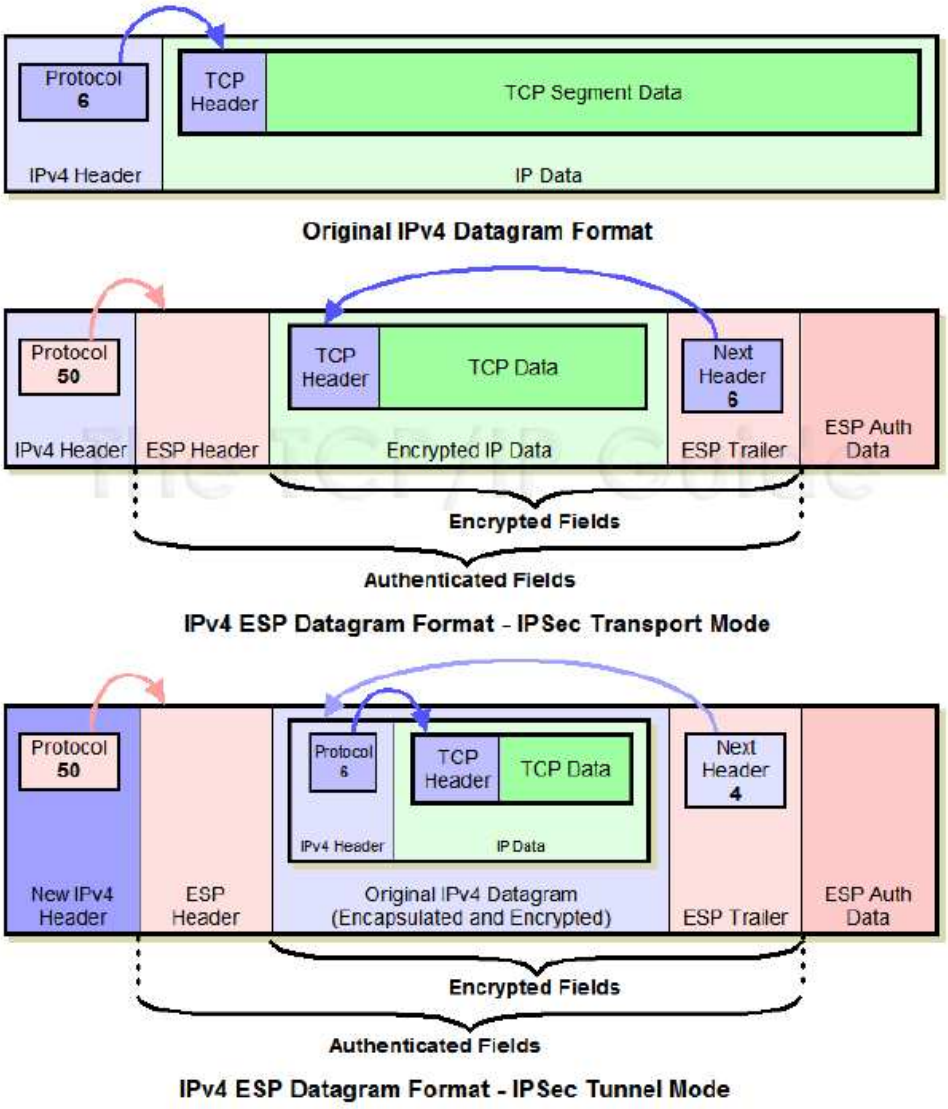


Figure 10: *The relationship between how an IPv4 packet is laid out without and with the ESP Header, in the transport mode and in the tunnel mode.* (*This figure is from http://www.tcpguide.com*)

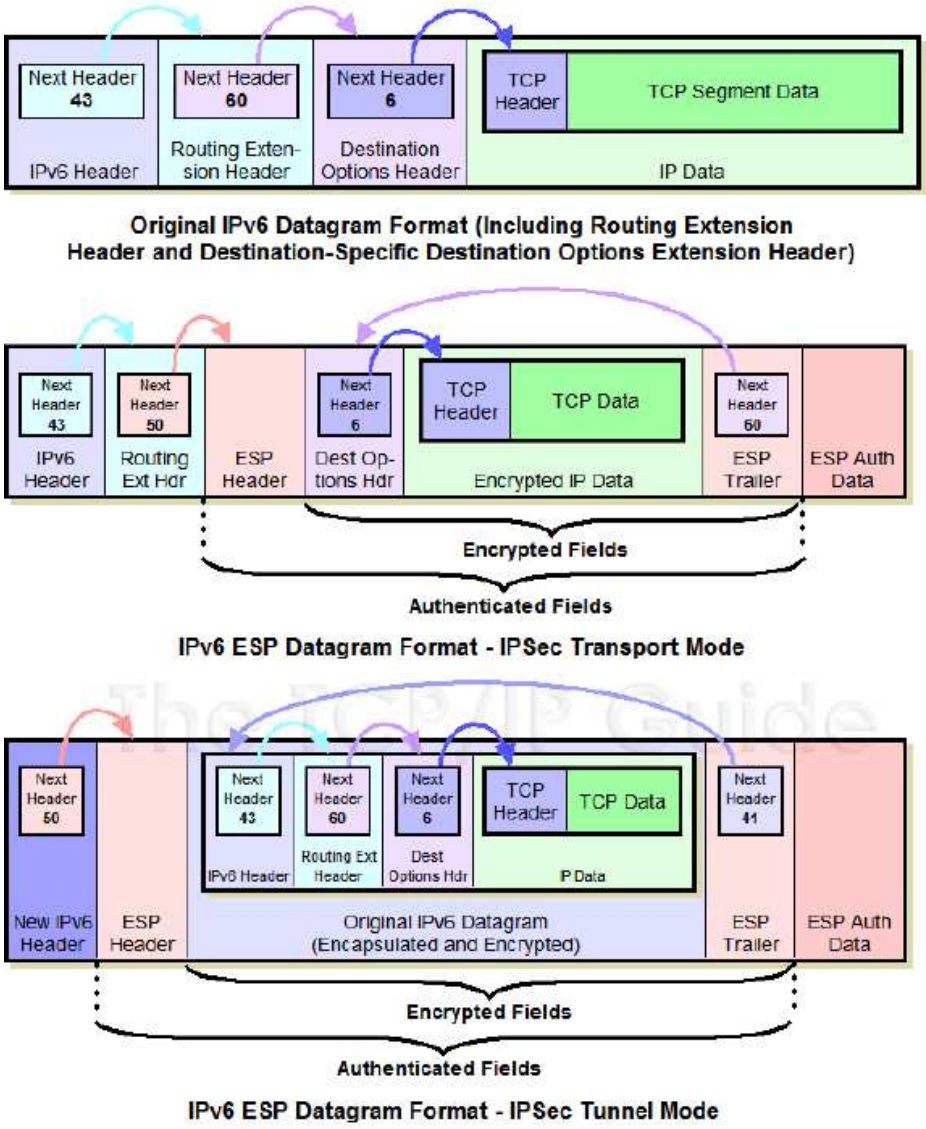


Figure 11: *The relationship between how an IPv6 packet is laid out without and with the ESP Header, in the transport mode and in the tunnel mode.* (*This figure is from http://www.tcpguide.com*)

20.3.4: IPSec Key Exchange

- Before ESP can be used, it is necessary for the two ends of a communication link to exchange the secret key that will be used for encryption. Similarly, AH needs an authentication key. [This is exactly what is achieved by the **Security Association (SA)** that was previously mentioned in Section 20.3.2. With IPSec, in general, the two endpoints must first establish an SA that declares what authentication and encryption algorithms will be used between the two endpoints.] The Security Association is established and the keys are exchanged with the **Internet Key Exchange** (IKE) protocol, whose latest version is described in RFC 5996. This version is also known as IKEv2.

- IKE combines the functions of three other protocols:

  – The Internet Security Association and Key Management Protocol (ISAKMP) that provides a generic framework for exchanging encryption keys and security association information. ISAKMP supports many different key exchange methods.

  – The Oakley Key-Exchange Protocol. it is based on Diffie-Hellman algorithm but provides additional security. This is the default method used by ISAKMP for creating a packet content encryption key.

  - The SKEME protocol for key exchange. ISAKMP uses the re-keying feature of this protocol.

- Diffie-Hellman's computationally expensive modular exponentiation makes it vulnerable to a **clogging attack** in which a communication node spends an inordinate amount of time generating session keys if too many of them are requested all at once. [An adversary may forge the source address of a legitimate party and send a public Diffie-Hellman key to an unsuspecting host, which then has to carry out modular exponentiation to compute the secret session key. But repeated receipts of the same request could clog up the host by causing it to spend all its time in modular exponentiation.] Diffie-Hellman is also vulnerable to the man-in-the-middle attack, as was mentioned in Lecture 13.

- Oakley thwarts the clogging attack by using a cookie-exchange between the two parties. A request for a secret session key must be accompanied with a cookie that is nothing but a pseudorandom number.

- Cookie exchange consists of each side sending a pseudorandom number to the other that must be acknowledged by the receiving party to the sending party. If the original requester for a secret session key was masquerading as someone else, they would never receive the cookie.

- A cookie is generated by hashing the IP source and destination

addresses, the UDP source and destination ports, and a locally generated secret value.

- Finally, as stated earlier in Section 20.3, the largest application of IPSec is in VPN. With regard to how IPSec security associations are used in VPN, each SA is for just one communication link. In other words, a typical VPN implementation provides you with a secure point-to-point tunnel between two specific endpoints in the VPN overlay network. These days there is considerable interest in extending the idea to **Group VPN** in which the same SA is shared by a large collection of communication endpoints.

- At the moment, there are several companies in the Bay Area working on implementing Group VPN with the GDOI protocol. GDOI stands for "Group Domain of Interpretation." It is specified by the IETF standard RFC 6407. The GDOI protocol runs on port 848.

20.4: SSL/TLS FOR TRANSPORT LAYER SECURITY

- SSL (Secure Socket Layer) was developed originally by Netscape in 1995 to provide secure and authenticated connections between browsers and servers. [Until recently, the title of this section was "SSL/TLS for Secure Web Services." That made sense because SSL/TLS was designed originally for secure exchange of information between web servers and browsers. More recently, though, SSL/TLS has become critically important to several other forms of information exchange in the internet. These include the exchange of information between routers, between routers and servers, between email exchange servers, between hosts and the internet-accessible printers, and so on. When the two endpoints involved in all these forms of information exchange have a need to authenticate each other and to create session keys for content encryption, they are likely to use the SSL/TLS protocol. Considering this widespread application of the protocol, the present section title is more appropriate.]

- SSL provides **transport layer** security. Recall from Figure 1 that the transport layer is where the TCP and UDP protocols reside in the TCP/IP stack. [Since SSL sits immediately above TCP in the protocol stack, a more precise way of stating this would be that SSL provides **Session Layer** security in the OSI model of the internet protocols. See Section 16.2 of Lecture 16 for the OSI model.]

- IETF (Internet Engineering Task Force, the body in charge of the core internet protocols, including the TCP/IP protocol) made

SSL Version 3 an open standard in 1999 and called it **TLS** (Transport Layer Security) **Version 1**. This first version of the TLS protocol is described in RFC 2246.

- Now it is common to refer to this protocol by the combined acronym SSL/TLS or TLS/SSL. Probably the biggest reason for why the acronym SSL continues to survive is the fact the world's most popular software library that implements this protocol is `OpenSSL`. I'll have more to say about that library later in this section.

- SSL/TLS plays a central role in the security and privacy needed for web commerce to work. As a case in point, before your laptop uploads your credit card information to, say, the Amazon.com website, your laptop must make certain that the remote host is indeed what it claims to be. That's where a protocol like SSL/TLS comes in. This protocol is also widely used to protect email servers (running under SMTP, POP, and IMAP protocols), chat servers (running under XMPP protocol), remote login security (through SSH servers), instant messaging (IM), and some virtual private networks (SSL VPNs).

- Fundamental to the security that is established with the SSL/TLS protocol are the certificates issued by the Certificate Authorities (CA). See Section 13.8 of Lecture 13 for how it has been possible for attackers to forge such certificates. These successful attempts

at creating forged certificates undermine the security that can be achieved with the SSL protocol.

- SSL/TLS allows for either server-only authentication or server-client authentication. In server-only authentication, the client receives the server's certificate. The client verifies the server's certificate and generates a secret key that it then encrypts with the server's public key. The client sends the encrypted secret key to the server; the server decrypts it with its own private key and subsequently uses the client-generated secret key to encrypt the messages meant for the client. [For a web browser to be able to engage in an SSL/TLS supported session with a web server — which is what you would want to see happen if you are exchanging, say, credit-card information with the web server — the web server must be able to provide the browser with a valid certificate signed by a recognized Certificate Authority (CA). As you know from Lecture 13, a certificate is validated by checking it with the public key of the CA, and the validation of the signing CA done in a similar manner, until you reach the Root Certificate Authority. The public keys of the root authorities are programmed into your browser. If a certificate cannot be validated by your browser in this manner — say because the CA that has signed that certificate is not known to your browser — a warning popup will be generated by the browser. If you tell your browser that you are willing to accept the certificate nonetheless, the authority that signed the certificate will be entered into the database of legitimate CAs maintained by your browser. Note that programming the keys of the root CAs into the browser code makes the root verification free of potential man-in-the-middle attacks. You can yourself check what root CAs are known to your browser by descending down the menu made available by the Preferences sub-menu under the Editor button of your browser menu bar.] Note that when a certificate received from a server is validated by your browser, most browsers will indicate the fact that you are now engaged in a secure link with the server

by showing a padlock icon usually at the right in the bottom portion of the browser frame, or by changing 'http' to 'https' in the URL window, or by changing the color of the URL window to green.

- In the server-client authentication, in addition to the secret key, the client also sends to the server its certificate that the server uses for authenticating the client.

- **OpenSSL is an *implementation* of the SSL and the TLS protocols.** [OpenSSL is used by the HTTPS and SMTPS protocols. When your browser connects with a web server to which you have to upload your credit card or banking information, your browser is most likely to be using the HTTPS protocol in its interaction with the server. SMTPS is for the secure transfer of email between hosts in the internet. Another closely related protocol that uses the `libssl` library component of the OpenSSL implementation is OpenSSH which is an *implementation* of the SSH protocol. As you surely know already, SSH, which stands for "Secure Shell," is used for logging into remote machines and for executing commands at those machines.]

- SSL (and, therefore, TLS) is actually not a single protocol, or even a single protocol layer. SSL is composed of four protocols in two layers, as shown in Figure 12. Of the four, the two most important protocols that are at the heart of SSL are the **SSL Handshake Protocol** and the **SSL Record Protocol**. The former authenticates the clients and the servers to each other and the latter then transmits the data confidentially. The other two protocols shown in the figure, the **SSL Cipher Change**

Protocol and the **SSL Alert Protocol** play relatively minor roles in how SSL works.

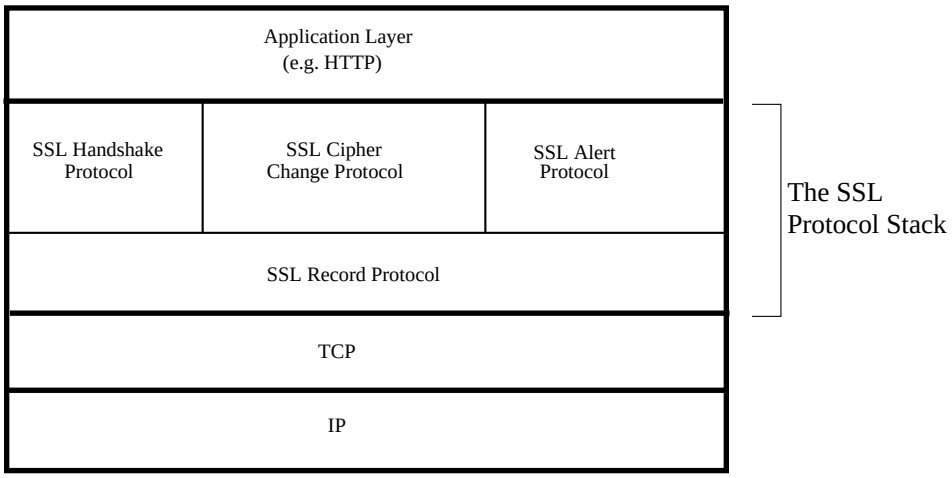


Figure 12: *SSL (and, therefore, TLS) is composed of four protocols in two layers as shown in this figure.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

20.4.1: The Twin Concepts of "SSL Connection" and "SSL Session"

- In the SSL family of protocols, a **connection** is a **one-time transport** of information between two nodes in a communication network.
  - A connection constitutes a **peer-to-peer** relationship between the two nodes.
  - Being one-time, connections are transient.
  - Every connection is associated with a **session**.
- A **session** is an enduring association between a **client** and a **server**.
  - A session is created by the **SSL Handshaking Protocol**.
  - A session can consist of multiple connections.
  - A session is characterized by a set of security parameters that apply to all the connections in the session.

- So whereas a connection takes care of transferring information securly from one endpoint to the other, the concept of a session allows for such data transfers to take place back and forth without having to renegotiate the security parameters for each separate connection. Note that this does NOT imply that a session can continue indefinitely. A session comes to an end when the exchange of data between the two endpoints has come to an end. **But what if we wanted to leave a session open in anticipitation of upcoming data exchanges between the two endpoints?** For that, you need what is known as the **Heartbeat Extension** to the SSL/TLS protocol. This extension, described in RFC 6520, will be presented briefly in Section 20.4.4. As mentioned earlier, the basic TLS protocol is described in RFC 2246.

- An SSL **connection state** is characterized by the following **parameters**:

  - `Server Write MAC Secret:` The secret key used in calculating the MAC (Message Authentication Code) value for the data sent by the server.

  - `Client Write MAC Secret:` The secret key used in calculating the MAC value for the data sent by the client.

  - `Server Write Key:` The symmetric-key encryption key for

data encrypted by the server and decrypted by the client.

- `Client Write Key:` The symmetric-key encryption key for data encrypted by the client and decrypted by the server.

- `Initialization vectors:` An initialization vector (IV) for each key used by a block cipher operating in the CBC mode is maintained. See Lecture 9 for the CBC block cipher mode. The vectors are initialized by the **SSL Handshake Protocol**. Subsequently, the final ciphertext block from each record is preserved for use as the IV with the following record. (*This will become clearer after we have discussed the SSL Record Protocol.*)

- `Sequence Numbers:` Each party maintains separate sequence numbers for the transmitted and received messages through each connection. When a party sends or receives a **change cipher spec** message, the appropriate sequence number is set to zero. Sequence numbers may not exceed $2^{64} - 1$.

- An SSL **session state** is characterized by the following **parameters**:

  - `Session Identifier:` An arbitrary byte sequence chosen by the server to identify an active or resumable session state.

– `Peer Certificate:` An X509.v3 certificate of the peer. This element of the state may be null.

– `Compression Method:` The algorithm used to compress the data prior to encryption.

– `Cipher Spec:` Specifics of the bulk data encryption algorithm and the hash algorithm used for MAC (Message Authentication Code) calculations. See Lecture 15 for further information on MAC and the related acronyms HMAC, SHA, etc.

– `Master Secret:` A 48-byte secret shared between the client and the server.

– `IsResumable:` A flag indicating whether the session is allowed to initiate new connections.

20.4.2: The SSL Record Protocol

- The **SSL Record Protocol** sits directly above the TCP protocol.

- This protocol provides two services: **Confidentiality** and **Message Integrity**.

- In a nutshell, this protocol is in charge of taking the actual data that the server wants to send to a client or that the client wants to send to a server, fragmenting the data into blocks, applying authentication and encryption primitives to each block, and handing the block to TCP for transmission over the network. On the receive side, the blocks are decrypted, verified for message integrity, reassembled, and delivered to the higher-level protocol.

- The operation of the **SSL Record Protocol** consists of the following **five** steps:

  - `Fragmentation:` The message (either from server to client, or from client to server) is fragmented into blocks whose length does not exceed 2^{14} (16384) bytes.

- `Compression:` This optional step requires lossless compression and carries the stipulation that the size of the input block will not increase by more than 1024 bytes. [As you'd expect, compression will, in most cases, reduce the length of a block produced by the fragmentation step. But for very short blocks, the length may increase.] SSLv3, the current version of SSL, does not specify compression.

- `Adding MAC:` This step computes the MAC (Message Authentication Code) for the block. The MAC is appended to the compressed message block.

- `Encryption:` The compressed message and the MAC are encrypted using symmetric-key encryption. The encryption may be carried out with a block cipher such as 3DES or with a stream cipher such as RC4-128. A number of choices are available for the encryption step depending on the level of security needed.

- `Append SSL Record Header:` Finally, an SSL header is is **prepended** to the encrypted block. The header consists of 8 bits for declaring the content type, 8 bits for declaring the major version used for SSL, 8 bits for declaring the minor version used, and 16 bits for declaring the length of the compressed plaintext (or the plaintext if no compression was used).

• Each output block produced by the **SSL Record Protocol** is

referred to as an **SSL record**. The length of a record is not to exceed $32,767$ bytes.

20.4.3: The SSL Handshake Protocol

- Before the **SSL Record Protocol** can do its thing, it must become aware of what algorithms to use for compression, authentication, and encryption. All of that information is generated by the **SSL Handshake Protocol**.

- The **SSL Handshake Protocol** is also responsible for the server and the client to authenticate each other.

- This protocol must also come up with the cryptographic keys to be used for the encryption and the authentication of each SSL record.

- As shown by Figure 13, the **SSL Handshake** protocol works in **four phases**.

- **Phase 1** handshaking, initiated by the client, is used to establish the security capabilities present at the two ends of a connection. The client sends to the server a `client_hello message` with the following parameters:

  – `Version` (the highest SSL version understood by the client)

  - `Random` (a 32-bit timestamp and a 28-byte random field that together serve as `nonces` during key exchange to prevent replay attacks)

  - `Session ID` (a variable length session identifier);

  - `Cipher Suite` (a list of cryptographic algorithms supported by the client, in decreasing order of preference); and

  - `Compression Method` (a list of compression methods the client supports).

- The server responds with its `server_hello message` that has a similar set of parameters. Server's response, as you'd expect, includes the specific algorithms selected by the server from the client's lists for compression, authentication, and encryption.

- The `Cipher Suite` parameter in the `server hello message` consists of two elements. The first element declares the `key exchange method` selected. (The choice is between `RSA`, three different types of `Diffie-Hellman`, etc.) The second element of the `Cipher Suite` parameter is called `CipherSpec`; it has a number of fields that indicate the authentication algorithm selected, the length of MAC, the encryption algorithm, etc.

- **Phase 2** handshaking is initiated by the server by sending the server certificate to the client. The server sends to the client the

message labeled `certificate` containing its one or more certificates for the validation of the server public key. [From the perspective of a user who wants his browser to upload his credit-card information to a website like `www.amazon.com`, this is probably the most critical part of the the handshake between the browser and the server at Amazon. Your browser must make sure that the server at the other end is the real thing and not someone else masquerading as Amazon. The browser establishes its trust in the server by validating the certificate downloaded from the Amazon server. See Section 13.8 of Lecture 13 regarding the integrity of such certificates.] This could be followed by a `server_key_exchange` message, and a `certificate_request` message if the server also wants to validate the client. The `server_key_exchange` message could, for example, consist of the global Diffie-Hellman values (a prime number and a primitive root of that number) and the server's Diffie-Hellman public key. **Phase 2** handshaking ends when the server sends the client a `server_hello_done` message.

- **Phase 3** handshaking is initiating by the client by sending to the server the client's certificate (but only if the server made a request for such a certificate in Phase 2). [In most routine applications of SSL, the client will NOT send a certificate to the server. As mentioned above, if are ordering stuff from a website like `www.amazon.com`, your browser has a need to authenticate the server and therefore needs the server's certificate. But the server has no real need to authenticate the client. In a business transaction when you are, say, ordering stuff, the server will authenticate you by, say, seeking validation for your credit-card number.] This is the message labeled `certificate` in Figure 13. Next, the client sends to the server a mandatory `client_key_exchange` message that could, for example, consist of a secret session key encrypted with the server's public key. This phase ends when the client sends to the server

a `certificate_verify` message to provide a verification of its certificates if they are signed by a certificate authority.

- **Phase 4** handshaking completes the setting up of a secure connection between the client and the server. The client sends to the server a `change_cipher_spec` message indicating that it is copying the pending CipherSpec into the current CipherSpec. (See Phase 1 handshaking for CipherSpec.) Next, the client sends to the server the `finished` message. As shown in Figure 13, the server does the same vis-a-vis the client.

- The `change_cipher_spec` message format must correspond to the **Change Cipher Spec Protocol**. This protocol says that the message must consist of a single byte with a value of 1 indicating the change.

- The last of the SSL protocols, **Alert Protocol**, is used to convey SSL-related alerts to the peer entity.

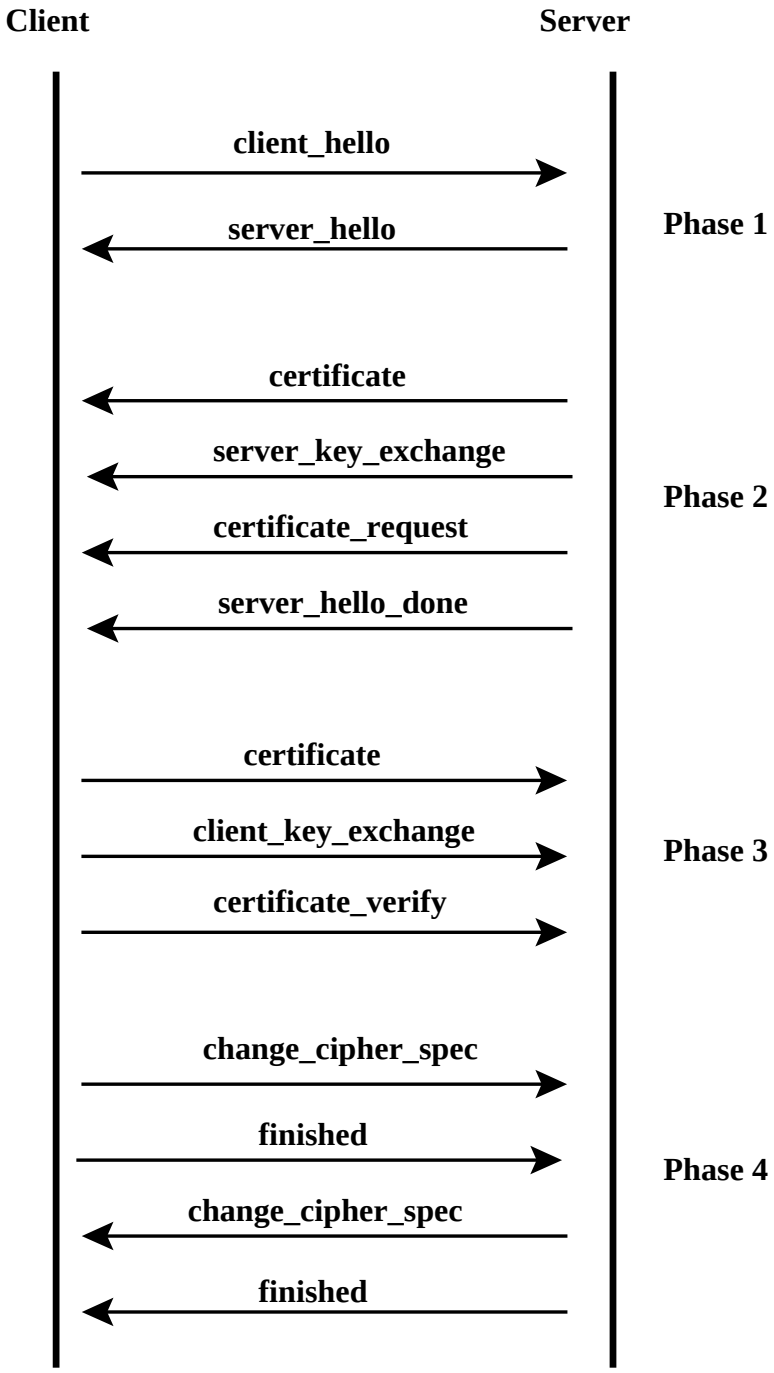


Figure 13: *The four phases of the SSL Handshake protocol*
(This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak)

20.4.4: The Heartbeat Extension to the SSL/TLS Protocol (RFC 6520)

- As mentioned earlier in Section 20.4.1, the SSL/TLS protocol has the notion of a connection and a session. Whereas a connection takes care of transferring data from one endpoint to the other, a session allows for multiple connections so that data can be exchanged back and forth between two endpoints.

- However, what a session does not allow for is to keep a session alive in anticipation of upcoming data exchanges between the two endpoints. That is, as soon as the data exchange between two endpoints terminates, the session will also terminate.

- Since there is significant overhead associated with the negotiation of the security parameters for establishing a secure session, some applications may require that once the security parameters have been agreed upon through the SSL/TLS Handshake protocol, they should continue to hold good even through lulls in data exchange between the two endpoints. So the question is how does one do that? How does either of the endpoints distinguish between a temporary lull in the data exchange and the final termination of a secure connection? These questions are answered by the SSL/TLS Heartbeat Extension Protocol as described in RFC 6520.

- The Heartbeat Extension Protocol sits on top of the SSL/TLS Record Protocol we presented in Section 20.4.2.

- Central to the Heartbeat Extension Protocol are two messages, `HeartbeatRequest` and `HeartbeatResponse`. When one endpoint sends a `HeartbeatRequest` message to the other endpoint, the former expects a `HeartbeatResponse` from the latter. A `HeartbeatRequest` message may arrive at any time during the lifetime of a session.

- When one endpoint sends a `HeartbeatRequest` message to the other endpoints, the former also starts what is known as the *retransmit timer*. During the time interval of the retransmit timer, the sending endpoint will not send another `HeartbeatRequest` message. An SSL/TLS session is considered to have terminated in the absence of a `HeartbeatResponse` packet within a time interval.

- The Heartbeat Extension protocol also includes "Heartbeat Hello Extension" that an endpoint can use to inform the other endpoint whether its implementation supports Heartbeats. In addition to declaring its support for Heartbeats, an endpoint can also indicate whether it is only willing to send `HeartbeatRequest` messages, or only willing to accept `HeartbeatResponse` messages, or both.

- As a protection against a replay attack, a `HeartbeatRequest` packet

must include a payload that must be returned without change by the receiver in its `HeartbeatResponse` packet. The payload is allowed to be arbitrary (and could potentially be a random sequence of bytes). More precisely, the Heartbeat protocol specifies that a request packet include values for the following two fields: an arbitrary payload and an integer that specifies the length of the payload. The protocol also specifies that the payload must be followed by padding (again an arbitrary sequence of bytes) whose length must be at least 16 bytes. The padding bytes are ignored by the receiving endpoint.

- The protocol specification for a Heartbeat message is:

```
struct {
      HeartbeatMessageType type;
      uint16 payload_length;
      opaque payload[HeartbeatMessage.payload_length];
      opaque padding[padding_length];
} HeartbeatMessage;
```

where the first field, of size one byte, specifies whether it is a `HeartbeatRequest` message or a `HeartbeatResponse` message. In an implementation, the second field, `payload_length`, would be represented by two bytes. What that implies is that the maximum size of the payload is 2^{16}. The protocol, however, limits the payload to 2^{14} bytes. As already mentioned, the padding is at least 16 bytes in length. [The now well-known Heartbleed bug in OpenSSL, discovered on April 7, 2014, was caused by the fact that the receiver of a HeartbeatRequest packet did not check that the size of the payload in the packet actually equaled the value given by the sender to the `payload_length` field in the request packet. This gave the sender the freedom to use the largest possible value of 16 bytes to `payload_length`

while placing virtually no content in the actual `payload` field. For preparting the response packet, this would cause the receiver to allocate memory on the basis of the sender's value for the `payload_length` field. This memory would then be filled with 2^{16} bytes of content starting with what was at the memory address of where the payload received from the sender was stored. Consequently, the actual payload returned by the sender could potentially include objects in the memory that had nothing to do with the received payload. It would be possible for these objects to be private keys, passwords, and such.]

20.5: THE Tor PROTOCOL FOR ANONYMIZED ROUTING

- The Tor protocol for anonymized routing is described in the paper "Tor: The Second-Generation Onion Router" by Roger Dingledine, Nick Mathewson, and Paul Syverson that was presented at the 13^{th} Usenix Security Symposium in 2004.

- Tor's genesis lies in the "onion routing" research that was funded by several US Government organizations starting in 1995. The basic motivation for this research was to figure out a way to set up internet communications so that an adversary snooping on the enroute packet traffic would not be able to analyze the packet headers for the purpose of finding out who was talking to whom. Gleaning information regarding the original source of the packets and their ultimate destination is referred to as the **traffic analysis attack**. [As you already know, even when protocols based on TLS are used for establishing encrypted communication channels for the transfer of information between the web browsers and the web servers, **the packet headers are always in clear text.** Even a protocol like IPSec, or the higher level protocols like VPN that are based on IPSec, do NOT safeguard you against traffic analysis attacks since the packet headers containing the source and the destination IP addresses are visible to all, especially so to the packet sniffers at the point of origination. And that

is true even when IPSec is used in the Tunnel Mode — a packet sniffer at any point before the packets get to the encapsulator used for the Tunnel Mode would know both the source and the destination of the packets.]

- Tor is open-source and available to all from `http://www.torproject.org`

- Although originally an acronym standing for "The Onion Router," "Tor" is now used as a name unto itself.

- It is believed that the folks who like to use BitTorrent to download media content generate a significant fraction of the Tor traffic. However, Tor is also popular with folks in countries where the free flow of information is restricted and with folks who want to "leak" information anonymously. Tor is also popular for something that the internet has become such a common ground for: **anonymous defamation**. [IMPORTANT: If you are using Tor for BitTorrent downloads, you owe it to yourself to read the INRIA report "One Bad Apple Spoils the Bunch" by Stevens Le Blond, Pere Manils, Abdelberi Chaabane, Mohamed Ali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. These authors were able to reveal the source IP addresses of 10,000 users of Tor engaged in BitTorrent downloads through the data collected from six Tor exit nodes over a period of 23 days.]

- As the reader will see from the description that follows, what makes the Tor protocol work is a very clever interplay between the RSA public-key cryptography and the DH (Diffie-Hellman)

public-key cryptography. [The more recent versions of Tor use ECDH (Elliptic Curve Diffie Hellman) that is presented in Lecture 14.]

- The Tor protocol is based on the twin notions of Onion Proxies (OP) and Onion Routers (OR). A user's OP first queries a Tor directory for the IP addresses of the ORs in the Tor overlay. [The notion of an *overlay network* will become clearer in Lecture 25.] The user then selects a subset of these ORs, commonly just 3, for constructing a path to the destination resource. [As for the word "onion" in the acronyms OP and OR, it is meant to be evocative of the layers of encryption placed on the Tor messages such that, except for the user's OP, the routing knowledge at any single node on a path through the Tor overlay is limited to exactly two nodes, the immediately preceding node on the path and the immediately following node.] Figure 14 illustrates the notion of a user's OP having selected the subset $\{B,\ C,\ D\}$ of ORs for a path to the intended destination. [Note that all the ORs together constitute a fully-connected overlay, meaning that every OR can talk directly to every other OR if so needed.]

- There are two other notions that are important to understanding Tor: **circuits** and **streams**. A user's OP constructs a path through the Tor overlay. *This path constitutes a circuit.* Subsequently, the two parties at the two end of a circuit may use it for an arbitrary number of TCP streams.

- To see how a user's OP constructs a path through the Tor overlay in a way that each node on the path has only local knowledge con-

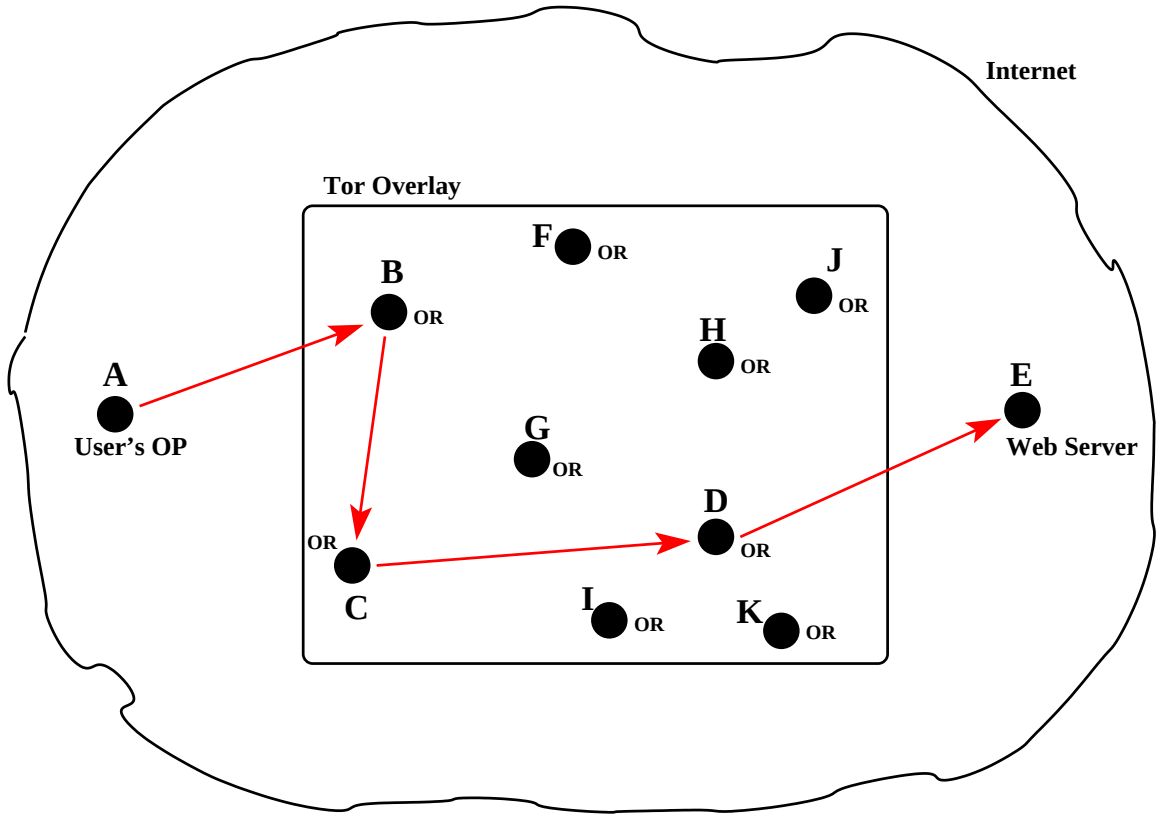
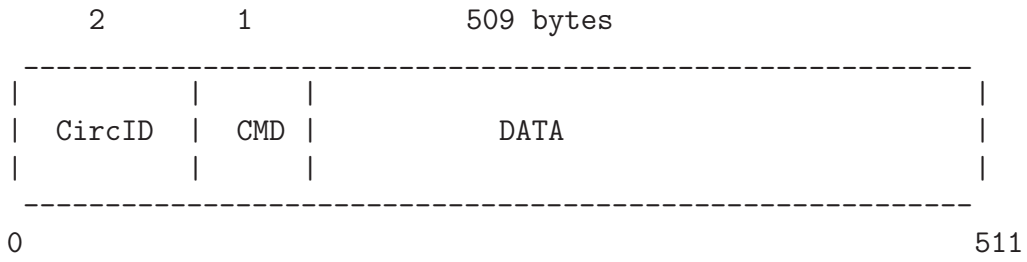


Figure 14: *B, C, and D are the ORs selected by user A for a path to the destination E.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

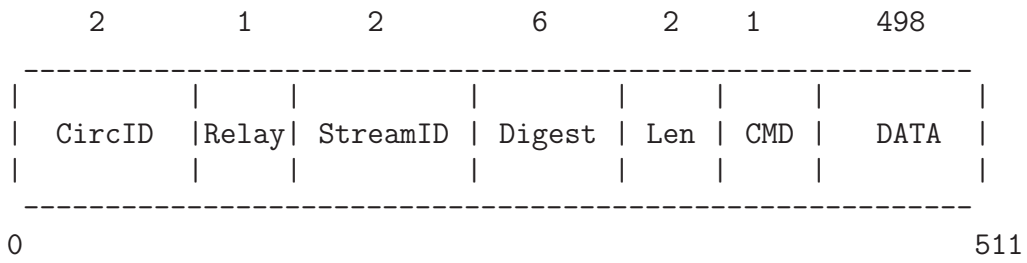
cerning the overall path, you need to understand the control and the data bearing messages that are specified by the Tor protocol.

- In the specification itself, as described in the paper by Dingledine et al., a message that is exchanged between an OP and an OR or between two ORs is called a cell. We'll refer to these messages by a more descriptive name torpacket.

- There are two types of torpackets: control torpackets and relay torpackets. Each torpacket consists of 512 bytes. Shown below is the structure of a control torpacket:



and shown below the structure of a relay torpacket:



The meanings to be associated with the various fields shown above should become clear from the discussion that follows re-

garding the different kinds of control and relay torpackets. As you will see, a control torpacket can be of the following kinds: `create`, `created`, `destroy`, and `padding`. Similarly, a relay torpacket can be of the following kinds: `relay extend`, `relay extended`, `relay truncate`, etc. [As one might guess, the role of a *control* torpacket is to alter the relationship between the sender node and the next node on the path that receives such a packet. But what about a *relay* torpacket? As paths are constructed (and torn down) incrementally by a user's OP, while the first link of the path can be constructed directly by the OP using a control torpacket, any extensions to the path are going to require that the commands for doing so be *relayed* to the currently last node on the path. Hence the need for *relay* torpackets. The discussion that follows makes this point clearer.]

- Initially, the *control* and the *relay* torpackets work together to create an end-to-end path (meaning a circuit) in the Tor overlay in such a way that each interior node on the path has only local knowledge of the path. While the basic purpose of a *relay* torpacket is to carry the data that is exchanged between the two endpoints, that can only be done after a path is fully constructed. During the process of path construction, the data carried by *relay* torpackets is for the purpose of extending the path beyond the current termination point. Such *relay* torpackets generate *contol* torpackets at the current terminal node on the path for extending the path.

- The first field in each control torpacket, circID, is a 2-byte integer circuit identifier. As you will see, a circuit identifier is unique to

each hop in a circuit — despite the fact that the circuit abstraction applies to entire end-to-end path.

- The second field, CMD, in a *control torpacket* is a one-byte integer representation of a command. A control torpacket may contain the following different commands:

  create : sent by an OP or OR to another OR to extend the path to the next node

  created : when an OR successfully extends the path to the next node in response to a *create* command from the previous node on a path, it sends back a *created* message to the previous node.

  destroy : sent by a node to another node to teardown the path

  padding : used for "keepalive" when a timeout might shut down a circuit otherwise

- The 1-byte command field (CMD) in the header of a *relay torpacket* can be used to create following kinds of such packets:

  relay extend : to extend the circuit by one hop

relay extended : to notify that *relay extend* was successful

relay truncate : to drop the last the OR on the path

relay truncated : to notify that *relay truncate* was successful

relay begin : to open a new stream

relay connected : to notify the OP that a stream was successfully opened

relay end : to close a previously opened stream

relay data : for transmission of data in stream

relay sendme : used for congestion control

relay teardown : used to close a broken stream

- What follows is a description of how a user's OP uses the control torpackets to create an end-to-end circuit incrementally, one hop at a time, in the Tor overlay. This explanation assumes that every OR node has a public RSA key that it makes available to the user's OP. These public keys will be static. So any communication sent to an OR that is encrypted with its RSA public key can only be understood by that OR. The explanation that follows

also includes another type of a public key — the Diffie-Hellman (DH) public key. Since these keys are not truly public (they are not even static), we will refer to them as the Y keys in order to remain consistent with the explanation of DH in Section 13.5 of Lecture 13. The DH Y keys are created on the fly between the user's OP and each of the ORs on the path chosen by the user. The purpose of the DH Y keys is that when the user's OP wants to send a message to a designated OR on the path, it is encrypted with the session key derived from the OP's DH Y key and that OR's DH Y key. [As a side note, AES is used for the symmetric-key encryption with such session keys.] So here we go:

- The user's OP sends a *create* control torpacket to the first node in the path chosen by the user. In Figure 14, this would be a *create* control torpacket from A to B. A's OP sets the CircID field of this torpacket to a new value, $circID_{AB}$, that was not previously used. The DATA field of this packet contains A's DH Y key $Y_{A \rightarrow B}$ that is encrypted with B's RSA public key.

- B responds back to A with the *created* control torpacket. The DATA field of this torpacket contains B's DH Y key $Y_{B \rightarrow A}$. Now both A and B can calculate the secret session key K_{AB} for their link as described in Section 13.5 of Lecture 13. [Note that all communications between any pair of nodes in the *underlying* network takes place using the TSL/SSL protocol for confidentiality. So the public DH Y key being sent by B back to A would not be visible to a packet sniffer. The RSA

public/private keys used specifically in the transmission of the control and relay torpackets are not to be confused with the RSA public/private keys that may be needed for routine but encrypted communications between any pair of nodes in the underlying network.]

– At this point we have a circuit with just one link in it. Since a circuit of any length is a legitimate circuit, the nodes A and B can now start exchanging relay torpackets, all using the identifier $circID_{AB}$ for the circID field. In order to extend the circult, A sends B a relay torpacket with the *relay extend* command. The DATA field of this *relay extend* torpacket includes a DH Y key $Y_{A\rightarrow C}$ that is meant specifically for the new terminal node on the path, that is, for the node C in Figure 14, and the identity of the new node. In order to make sure that the key $Y_{A\rightarrow C}$ is not seen by node B, it is encrypted with C's RSA public key. As you would expect, the DATA field in the *relay extend* torpacket from A to B is encrypted with the session key K_{AB}.

– When B receives the *relay extend* torpacket from A, it knows that it is the current endpoint on the path. So it generates a *control* torpacket whose DATA field contains A's DH Y key $Y_{A\rightarrow C}$ that was meant specifically for node C and that was encrypted with C's RSA public key. This DATA field is encrypted with C's RSA public key. The *control* torpacket sent by B to C uses a new randomly generated number for the circID field, $circID_{BC}$. This becomes the identifier for the segment of the circuit between the nodes B and C. There is

no need for A to know this identifier. In other words, only the node B knows both $circID_{AB}$ and $circID_{BC}$. This fact plays an important role in ensuring that each node on the path has only the local knowledge of the path.

– Node C responds back to B with a *created* control torpacket. The DATA field of this torpacket contains C's DH Y key $Y_{C \rightarrow A}$ meant for A. Node B sends this acknowledgment back to A using the *relay extended* torpacket, with its DATA field containing the key $Y_{C \rightarrow A}$. Now both A and C can calculate the secret session key K_{AC} for any messages that A may want to send to C (through B of course) that B is not allowed to see.

– The path may be extended in the same manner to the node D shown in Figure 14 by using a combination of control and relay torpackets.

- In constructing an end-to-end circuit in the manner described above, there was never a need for using A's public RSA key. In that sense, the user A remains anonymous to all the ORs in the circuit. By the same token, B will remain anonymous to D and so on. But all the ORs in a circuit are known to the user A (not surprising, since A chose them for the circuit).

- After an end-to-end circuit is created in this manner, the user A

can start pushing data into the circuit that is meant for the final destination E shown in Figure 14. However, before placing this data on the wire, A sends a *relay begin* torpacket to B, from where it is forwarded to the next node on the circuit, and so on, thus creating an end-to-end stream between A and E. The user A is allowed to create an arbitrary number of streams and they can all share the same circuit. While the different TCP streams will have different streamID values in the relay torpackets that carry the stream data, they will have the same value for the circID field (even though the value of this circID field will change from hop to hop in a circuit).

- Assuming the $A \rightarrow B \rightarrow C \rightarrow D$ path in the Tor overlay as shown in Figure 14, the stream data that the user A places on the wire is encrypted with the K_{AD} session key, followed by its encryption with K_{AC} session key, followed by its encryption by K_{AB} session key. [Hence the analogy with the onion.] As these stream data bearing *relay data* torpackets are received by B from A, the node B uses the session key K_{AB} to decrypt the top layer of encryption and forward the stream to the next code, node C, in the circuit. This process continues until the stream data reaches the final node D, from where it goes via the normal TCP transmission to the application running at the destination E.

- Here are the two most important questions that give people much anxiety when contemplating using Tor for accessing a web resource: **(1) Can the exit node operator see the source**

IP address, meaning the IP address of node A in our example? And (2) Can the exit node operator see the data payload of the source packet? The answer to the second question is easy: If node A is trying to reach an HTTPS web site, that implies end-to-end encryption of the payload in the packets. In that case, the exit node operator obviously cannot peer inside the packets that A is sending out.

- But what about the first question raised above? That is, can the exit node operator see the source IP address? **In principle, that should not be possible.** The Tor logic that keeps A's IP address shielded from the exit node D is the same as the logic that keeps B's IP address shielded from D. The packets that go out from D to the web server at E should only bear D's IP address in the source fields. When D receives replies to those packets from the web server, it simply forwards them back to C.

- Nonetheless, one should note that Le Blond et al. were able to successfully reveal the source IP addresses of 10,000 hosts that used Tor for BitTorrent downloads during a period of 23 days in 2011. (This report was cited at the start of this section.) So the question is how did Le Blond et al. manage to accomplish their feat despite the anonymity guarantees built into the Tor protocol.

- The attack by Le Blond et al. took advantage of the peculiarities of the BitTorrent protocol. Being a P2P protocol (See Lecture

25 for BitTorrent), a BitTorrent client must somehow acquire a list of the peers that are the keepers of the media content that the client wishes to download and then, subsequently, join the peers. BitTorrent gives a client three different ways to discover the peers: (1) By contacting a centralized tracker that keeps a list of all the peers currently in possession of the media content of interest to the client; (2) by contacting a DHT based tracker in accordance with the explanation in Section 25.10 of Lecture 25; and (3) through the ancillary protocol PEX as also explained in Section 25.10 of Lecture 25. [When a BitTorrent client contacts a tracker through Tor, the IP address of the client is protected since what the tracker sees is the IP address of the Tor exit node.] The Le Blond et al. attack exploited the first two methods for peer discovery. In both these methods, as things work at the moment, the peer list of IP addresses that is received by a client is without encryption. Since this list consists of the other users of BitTorrent, by simply monitoring an exit node, it is possible to figure out the identities of the BitTorrent users.

20.6: HOMEWORK PROBLEMS

1. What are the pros and cons of providing security at the different layers of the TCP/IP protocol stack?

2. How is the sender authentication carried out in PGP?

3. A truly unique feature of PGP is that it is NOT based on the notion of a Certificate Authority (CA) for authenticating the binding between a given public key and its owner. On the other hand, PGP uses the idea of "web of trust." What does it mean and what are its pros and cons vis-a-vis the more commonly used CA-based approach?

4. How is IPSec grafted onto IPv4? The "Protocol" field of the IPv4 header plays a critical role in this. How?

5. What is the difference between the server-only authentication and server-client authentication in SSL/TLS?

6. We say that SSL/TLS is not really a single protocol, but a stack of protocols. Explain. What are the different protocols in the SSL/TLS stack?

7. What is the difference between a connection and a session in SSL/TLS? Can a session include multiple connections? Explain the notions "connection state" and "session state" in SSL/TLS. What security feature apply to each?

8. What is the role of the SSL Record Protocol in SSL/TLS?

9. What is the role of the Heartbeat Extension Protocol in SSL/TLS?

10. What lesson is to be learned from the Heartbleed bug with regard to testing of C-based networking software? [See the note in red at the end of Section 20.4.4.]