8

NETWORK SECURITY

For the first few decades of their existence, computer networks were primarily used by university researchers for sending email and by corporate employees for sharing printers. Under these conditions, security did not get a lot of attention. But now, as millions of ordinary citizens are using networks for banking, shopping, and filing their tax returns, and weakness after weakness has been found, network security has become a problem of massive proportions. In this chapter, we will study network security from several angles, point out numerous pitfalls, and discuss many algorithms and protocols for making networks more secure.

On a historical note, network hacking already existed long before there was an Internet. Instead, the telephone network was the target and messing around with the signaling protocol was known as **phone phreaking**. Phone phreaking started in the late 1950s, and really took off in the 1960s and 1970s. In those days, the control signals used to authorize and route calls, were still ''in band'': the phone company used sounds at specific frequencies in the same channel as the voice communication to tell the switches what to do.

One of the best-known phone phreakers is **John Draper**, a controversial figure who found that the toy whistle included in the boxes of Cap'n Crunch cereals in the late 1960s emitted a tone of exactly 2600 Hz which happened to be the frequency that AT&T used to authorize long-distance calls. Using the whistle, Draper was able to make long distance calls for free. Draper became known as **Captain Crunch** and used the whistles to build so-called blue boxes to hack the telephone

system. In 1974, Draper was arrested for toll fraud and went to jail, but not before he had inspired two other pioneers in the Bay area, **Steve Wozniak** and **Steve Jobs**, to also engage in phone phreaking and build their own blue boxes, as well as, at a later stage, a computer that they decided to call *Apple*. According to Wozniak, there would have been no Apple without Captain Crunch.

Security is a broad topic and covers a multitude of sins. In its simplest form, it is concerned with making sure that nosy people cannot read, or worse yet, secretly modify messages intended for other recipients. It is also concerned with attackers who try to subvert essential network services such as BGP or DNS, render links or network services unavailable, or access remote services that they are not authorized to use. Another topic of interest is how to tell whether that message purportedly from the IRS ''Pay by Friday, or else'' is really from the IRS and not from the Mafia. Security additionally deals with the problems of legitimate messages being captured and replayed, and with people later trying to deny that they sent certain messages.

Most security problems are intentionally caused by malicious people trying to gain some benefit, get attention, or harm someone. A few of the most common perpetrators are listed in Fig. 8-1. It should be clear from this list that making a network secure involves a lot more than just keeping it free of programming errors. It involves outsmarting often intelligent, dedicated, and sometimes well-funded adversaries. Measures that will thwart casual attackers will have little impact on the serious ones.

In an article in *USENIX ;Login:*, James Mickens of Microsoft (and now a professor at Harvard University) argued that you should distinguish between everyday attackers and, say, sophisticated intelligence services. If you are worried about garden-variety adversaries, you will be fine with common sense and basic security measures. Mickens eloquently explains the distinction:

''If your adversary is the Mossad, you're gonna die and there's nothing that you can do about it. The Mossad is not intimidated by the fact that you employ https://. *If the Mossad wants your data, they're going to use a drone to replace your cellphone with a piece of uranium that's shaped like a cellphone, and when you die of tumors filled with tumors, they're going to hold a press conference and say ''It wasn't us'' as they wear t-shirts that say ''IT WAS DEFINITELY US'' and then they're going to buy all of your stuff at your estate sale so that they can directly look at the photos of your vacation instead of reading your insipid emails about them.''*

Mickens' point is that sophisticated attackers have advanced means to compromise your systems and stopping them is very hard. In addition, police records show that the most damaging attacks are often perpetrated by insiders bearing a grudge. Security systems should be designed accordingly.

Adversary	Goal
Student	To have fun snooping on people's email
Cracker	To test someone's security system; steal data
Sales rep	To claim to represent all of Europe, not just Andorra
Corporation	To discover a competitor's strategic marketing plan
Ex-employee	To get revenge for being fired
Accountant	To embezzle money from a company
Stockbroker	To deny a promise made to a customer by email
Identity thief	To steal credit card numbers for sale
Government	To learn an enemy's military or industrial secrets
Terrorist	To steal biological warfare secrets

Figure 8-1. Some people who may cause security problems, and why.

8.1 FUNDAMENTALS OF NETWORK SECURITY

The classic way to deal with network security problems is to distinguish three essential security properties: confidentiality, integrity, and availability. The common abbreviation, **CIA**, is perhaps a bit unfortunate, given that the other common expansion of that acronym has not been shy in violating those properties in the past. **Confidentiality** has to do with keeping information out of the grubby little hands of unauthorized users. This is what often comes to mind when people think about network security. **Integrity** is all about ensuring that the information you received was really the information sent and not something that an adversary modified. **Availability** deals with preventing systems and services from becoming unusable due to crashes, overload situations, or deliberate misconfigurations. Good examples of attempts to compromise availability are the denial-of-service attacks that frequently wreak havoc on high-value targets such as banks, airlines and the local high school during exam time. In addition to the classic triumvirate of confidentiality, integrity, and availability that dominates the security domain, there are other issues that play important roles also. In particular, **authentication** deals with determining whom you are talking to before revealing sensitive information or entering into a business deal. Finally, **nonrepudiation** deals with signatures: how do you prove that your customer really placed an electronic order for 10 million left-handed doohickeys at 89 cents each when he later claims the price was 69 cents? Or maybe he claims he never placed any order after seeing that a Chinese firm is flooding the market with left-handed doohickeys for 49 cents.

All these issues occur in traditional systems, too, but with some significant differences. Integrity and secrecy are achieved by using registered mail and locking documents up. Robbing the mail train is harder now than it was in Jesse James' day. Also, people can usually tell the difference between an original paper

document and a photocopy, and it often matters to them. As a test, make a photocopy of a valid check. Try cashing the original check at your bank on Monday. Now try cashing the photocopy of the check on Tuesday. Observe the difference in the bank's behavior.

As for authentication, people authenticate other people by various means, including recognizing their faces, voices, and handwriting. Proof of signing is handled by signatures on letterhead paper, raised seals, and so on. Tampering can usually be detected by handwriting, ink, and paper experts. None of these options are available electronically. Clearly, other solutions are needed.

Before getting into the solutions themselves, it is worth spending a few moments considering where in the protocol stack network security belongs. There is probably no one single place. Every layer has something to contribute. In the physical layer, wiretapping can be foiled by enclosing transmission lines (or better yet, optical fibers) in sealed metal tubes containing an inert gas at high pressure. Any attempt to drill into a tube will release some gas, reducing the pressure and triggering an alarm. Some military systems use this technique.

In the data link layer, packets on a point-to-point link can be encrypted as they leave one machine and decrypted as they enter another. All the details can be handled in the data link layer, with higher layers oblivious to what is going on. This solution breaks down when packets have to traverse multiple routers, however, because packets have to be decrypted at each router, leaving them vulnerable to attacks from within the router. Also, it does not allow some sessions to be protected (e.g., those involving online purchases by credit card) and others not. Nevertheless, **link encryption**, as this method is called, can be added to any network easily and is often useful.

In the network layer, firewalls can be deployed to prevent attack traffic from entering or leaving networks. IPsec, a protocol for IP security that encrypts packet payloads, also functions at this layer. At the transport layer, entire connections can be encrypted end-to-end, that is, process to process. Problems such as user authentication and nonrepudiation are often handled at the application layer, although occasionally (e.g., in the case of wireless networks), user authentication can take place at lower layers. Since security applies to all layers of the network protocol stack, we dedicate an entire chapter of the book to this topic.

8.1.1 Fundamental Security Principles

While addressing security concerns in all layers of the network stack is certainly necessary, it is very difficult to determine when you have addressed them sufficiently and if you have addressed them all. In other words, *guaranteeing* security is hard. Instead, we try to improve security as much as we can by consistently applying a set of security principles. Classic security principles were formulated as early as 1975 by Jerome Saltzer and Michael Schroeder:

1. **Principle of economy of mechanism**. This principle is sometimes paraphrased as the principle of simplicity. Complex systems tend to have more bugs than simple systems. Moreover, users may not understand them well and use them in a wrong or insecure way. Simple systems are good systems. For instance, PGP (Pretty Good Privacy, see Sec. 8.11), offers powerful protection for email. However, many users find it cumbersome in practice and so far it has not yet gained very widespread adoption. Simplicity also helps to minimize the **attack surface** (all the points where an attacker may interact with the system to try to compromise it). A system that offers a large set of functions to untrusted users, each implemented by many lines of code, has a large attack surface. If a function is not really needed, leave it out.
2. **Principle of fail-safe defaults**. Say you need to organize the access to a resource. It is better to make explicit rules about when one can access the resource than trying to identify the condition under which access to the resource should be denied. Phrased differently: a default of lack of permission is safer.
3. **Principle of complete mediation**. Every access to every resource should be checked for authority. It implies that we must have a way to determine the source of a request (the requester).
4. **Principle of least authority**. This principle, often known as **POLA**, states that any (sub) system should have just enough authority (privilege) to perform its task and no more. Thus, if attackers compromise such a system, they elevate their privilege by only the bare minimum.
5. **Principle of privilege separation**. Closely related to the previous point: it is better to split up the system into multiple POLA-compliant components than a single component with all the privileges combined. Again, if one component is compromised, the attackers will be limited in what they can do.
6. **Principle of least common mechanism**. This principle is a little trickier and states that we should minimize the amount of mechanism common to more than one user and depended on by all users. Think of it this way: if we have a choice between implementing a network routine in the operating system where its global variables are shared by all users, or in a user space library which, to all intents and purposes, is private to the user process, we should opt for the latter. The shared data in the operating system may well serve as an information path between different users. We shall see an example of this in the section on TCP connection hijacking.

7. **Principle of open design**. This states plain and simple that the design should not be secret and generalizes what is known as Kerckhoffs' principle in cryptography. In 1883, the Dutch-born Auguste Kerckhoffs published two journal articles on military cryptography which stated that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge. In other words, do not rely on ''security by obscurity,'' but assume that the adversary immediately gains familiarity with your system and knows the encryption and decryption algorithms.
8. **Principle of psychological acceptability**. The final principle is not a technical one at all. Security rules and mechanisms should be easy to use and understand. Again, many implementations of PGP protection for email fail this principle. However, acceptability entails more. Besides the usability of the mechanism, it should also be clear why the rules and mechanisms are necessary in the first place.

An important factor in ensuring security is also the concept of **isolation**. Isolation guarantees the separation of components (programs, computer systems, or even entire networks) that belong to different security domains or have different privileges. All interaction that takes place between the different components is mediated with proper privilege checks. Isolation, POLA, and a tight control of the flow of information between components allow the design of strongly compartmentalized systems.

Network security comprises concerns in the domain of systems and engineering as well as concerns rooted in theory, math, and cryptography. A good example of the former is the classic **ping of death**, which allowed attackers to crash hosts all over the Internet by using fragmentation options in IP to craft ICMP echo request packets larger than the maximum allowed IP packet size. Since the receiving side never expected such large packets, it reserved insufficient buffer memory for all the data and the excess bytes would overwrite other data that followed the buffer in memory. Clearly, this was a bug, commonly known as a buffer overflow. An example of a cryptography problem is the 40-bit key used in the original WEP encryption for WiFi networks which could be easily brute-forced by attackers with sufficient computational power.

8.1.2 Fundamental Attack Principles

The easiest way to structure a discussion about systems aspects of security is to put ourselves in the shoes of the adversary. So, having introduced fundamental aspects of security above, let us now consider the **fundamentals of attacks**.

From an attacker perspective, the security of a system presents itself as a set of challenges that attackers must solve to reach their objectives. There are multiple ways to violate confidentiality, integrity, availability, or any of the other security

properties. For instance, to break confidentiality of network traffic, an attacker may break into a system to read the data directly, trick the communicating parties to send data without encryption and capture it, or, in a more ambitious scenario, break the encryption. All of these are used in practice and all of them consist of multiple steps. We will deep dive into the fundamentals of attacks in Sec. 8.2. As a preview, let us consider the various steps and approaches attackers may use.

1. **Reconnaissance**. Alexander Graham Bell once said: ''Preparation is the key to success.'' and thus it is for attackers also. The first thing you do as an attacker is to get to know as much about your target as you can. In case you plan to attack by means of spam or social engineering, you may want to spend some time sifting through the online profiles of the people you want to trick into giving up information, or even engage in some old-fashioned dumpster diving. In this chapter, however, we limit ourselves to technical aspects of attacks and defenses. Reconnaissance in network security is about discovering information that helps the attacker. Which machines can we reach from the outside? Using which protocols? What is the topology of the network? What services run on which machines? Et cetera. We will discuss reconnaissance in Sec. 8.2.1

2. **Sniffing and Snooping**. An important step in many network attacks concerns the interception of network packets. Certainly if sensitive information is sent ''in the clear'' (without encryption), the ability to intercept network traffic is very useful for the attacker, but even encrypted traffic can be useful—to find out the MAC addresses of communicating parties, who talks to whom and when, etc. Moreover, an attacker needs to intercept the encrypted traffic to break the encryption. Since an attacker has access to other people's network traffic, the ability to sniff indicates that at least the principles of least authority and complete mediation are not sufficiently enforced. Sniffing is easy on a broadcast medium such as WiFi, but how to intercept traffic if it does not even travel over the link to which your computer is connected? Sniffing is the topic of Sec. 8.2.2.

3. **Spoofing**. Another basic weapon in the hands of attackers is masquerading as someone else. Spoofed network traffic pretends to originate from some other machine. For instance, we can easily transmit an Ethernet frame or IP packet with a different source address, as a means to bypass a defense or launch denial-of-service attacks, because these protocols are very simple. However, can we also do so for complicated protocols such as TCP? After all, if you send a TCP SYN segment to set up a connection to a server with a spoofed IP address, the server will reply with its SYN/ACk segment (the second phase of the connection setup) to that IP address, so unless the

attackers are on the same network segment, they will not see the reply. Without that reply, they will not know the sequence number used by the server, and hence, they will not be able to communicate. Spoofing circumvents the principle of complete mediation: if we cannot determine who sent a request, we cannot properly mediate it. In Sec. 8.2.3, we discuss spoofing in detail.

4. **Disruption**. The third component of our CIA triad, availability, has grown in importance also for attackers, with devastating **DoS** (**Denial of Service**) attacks on all sorts of organizations. Moreover, in response to new defenses, these attacks have grown ever more sophisticated. One can argue that DoS attacks abuse the fact that the principle of least common mechanism is not rigorously enforced—there is insufficient isolation. In Sec. 8.2.4, we will look at the evolution of such attacks.

Using these fundamental building blocks, attackers can craft a wide range of attacks. For instance, using reconnaissance and sniffing, attackers may find the address of a potential victim computer and discover that it trusts a server so that any request coming from that server is automatically accepted. By means of a denial-of-service (disruption) attack they can bring down the real server to make sure it does not respond to the victim any more and then send spoofed requests that appear to originate from the server. In fact, this is exactly how one of the most famous attacks in the history of the Internet (on the San Diego Supercomputer Center) happened. We will discuss the attack later.

8.1.3 From Threats to Solutions

After discussing the attacker's moves, we will consider what we can do about them. Since most attacks arrive over the network, the security community quickly realized that the network may also be a good place to monitor for attacks. In Sec. 8.3, we will look at firewalls, intrusion detection systems and similar defenses.

Where Secs. 8.2 and 8.3 address the systems-related issues of attackers getting their grubby little hands on sensitive information or systems, we devote Secs. 8.4–8.9 to the more formal aspects of network security, when we discuss **cryptography** and **authentication**. Rooted in mathematics and implemented in computer systems, a variety of cryptographic primitives help ensure that even if network traffic falls in the wrong hands, nothing too bad can happen. For instance, attackers will still not be able to break confidentiality, tamper with the content, or successfully replay a network conversation. There is a lot to say about cryptography, as there are different types of primitives for different purposes (proving authenticity, encryption using public keys, encryption using symmetric keys, etc.) and each type tends to have different implementations. In Sec. 8.4, we introduce the key concepts of cryptography, and Sections 8.5 and 8.6 discuss symmetric and public

key cryptography, respectively. We explore digital signatures in Sec. 8.7 and key management in Sec. 8.8.

Sec. 8.9 discusses the fundamental problem of secure authentication . Authentication is that which prevents spoofing altogether: the technique by which a process verifies that its communication partner is who it is supposed to be and not an imposter. As security became increasingly important, the community developed a variety of authentication protocols. As we shall see, they tend to build on cryptography.

In the sections following authentication, we survey concrete examples of (often crypto-based) network security solutions. In Sec. 8.10, we discuss network technologies that provide communication security, such as IPsec, VPNs, and Wireless security. Section 8.11 looks at the problem of email security, including explanations of PGP (Pretty Good Privacy) and S/MIME (Secure Multipurpose Internet Mail Extension). Section 8.12 discusses security in the wider Web domain, with descriptions of secure DNS (DNSSEC), scripting code that runs in browsers, and the Secure Sockets Layer (SSL). As we shall see, these technologies use many of the ideas discussed in the preceding sections.

Finally, we discuss social issues in Sec. 8.13. What are the implications for important rights, such as privacy and freedom of speech? What about copyright and protection of intellectual property? Security is an important topic so looking at it closely is worthwhile.

Before diving in, we should reiterate that security is an entire field of study in its own right. In this chapter, we focus only on networks and communication, rather than issues related to hardware, operating systems, applications, or users. This means that we will not spend much time looking at bugs and there is nothing here about user authentication using biometrics, password security, buffer overflow attacks, Trojan horses, login spoofing, process isolation, or viruses. All of these topics are covered at length in Chap. 9 of *Modern Operating Systems* (Tanenbaum and Bos, 2015). The interested reader is referred to that book for the systems aspects of security. Now let us begin our journey.

8.2 THE CORE INGREDIENTS OF AN ATTACK

As a first step, let us consider the fundamental ingredients that make up an attack. Virtually all network attacks follow a recipe that mixes some variants of these ingredients in a clever manner.

8.2.1 Reconnaissance

Say you are an attacker and one fine morning you decide that you will hack organization X, where do you start? You do not have much information about the organization and, physically, you are an Internet away from the nearest office, so

dumpster diving or shoulder surfing are not options. You can always use **social engineering**, to try and extract sensitive information from employees by sending them emails (spam), or phoning them, or befriending them on social networks, but in this book, we are interested in more technical issues, related to computer networks. For instance, can you find out what computers exist in the organization, how they are connected, and what services they run?

As a starting point, we assume that an attacker has a few IP addresses of machines in the organization: Web servers, name servers, login servers, or any other machines that communicate with the outside world. The first thing the attacker will want to do is explore that server. Which TCP and UDP ports are open? An easy way to find out is simply to try and set up a TCP connection to each and every port number. If the connection is successful, there was a service listening. For instance, if the server replies on port 25, it suggests an SMTP server is present, if the connection succeeds on port 80, there will likely be a Web server, etc. We can use a similar technique for UDP (e.g., if the target replies on UDP port 53, we know it runs a domain name service because that is the port reserved for DNS).

Port Scanning

Probing a machine to see which ports are active is known as **port scanning** and may get fairly sophisticated. The technique we described earlier, where an attacker sets up a full TCP connection to the target (a so-called **connect scan**) is not sophisticated at all. While effective, its major drawback is that it is very visible to the target's security team. Many servers tend to log successful TCP connections, and showing up in logs during the **reconnaissance** phase is not what an attacker wants. To avoid this, she can make the connections deliberately unsuccessful by means of a **half-open scan**. A half-open scan only pretends to set up connections: it sends TCP packets with the SYN flag set to all port numbers of interest and waits for the server to send the corresponding SYN/ACKs for the ports that are open, but it never completes the three-way handshake. Most servers will not log these unsuccessful connection attempts.

If half-open scans are better than connect scans, why do we still discuss the latter? The reason is that half-open scans require more advanced attackers. A full connection to a TCP port is typically possible from most machines using simple tools such as telnet, that are often available to unprivileged users. For a half-open scan, however, attackers need to determine exactly which packets should and should not be transmitted. Most systems do not have standard tools for nonprivileged users to do this and only users with administrator privileges can perform a half-open scan.

Connect scans (sometimes referred to as **open scans**) and half-open scans both assume that it is possible to initiate a TCP connection from an arbitrary machine outside the victim's network. However, perhaps the firewall does not allow connections to be set up from the attacker's machine. For instance, it may block all

SYN segments. In that case, the attacker may have to resort to more esoteric scanning techniques. For instance, rather than a SYN segment, a **FIN scan** will send a TCP FIN segment, which is normally used to close a connection. At first sight, this does not make sense because there is no connection to terminate. However, the response to the FIN packet is often different for open ports (with listening services behind them) and closed ports. In particular, many TCP implementations send a TCP RST packet if the port is closed, and nothing at all if it is open. Fig. 8-2 illustrates these three basic scanning techniques.

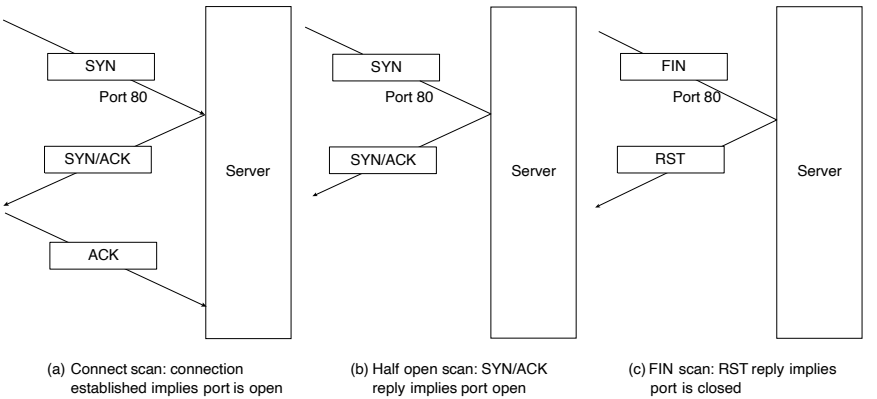

Figure 8-2. Basic port scanning techniques. (a) Connect scan. (b) Half-open scan. (c) FIN scan.

By this time, you are probably thinking: ''If we can do this with the SYN flags and the FIN flags, can we try some of the other flags?'' You would be right. Any configuration that leads to different responses for open and closed ports works. A well-known other option is to set many flags at once (FIN, PSH, URG), something known as **Xmas scan** (because your packet is lit up like a Christmas tree).

Consider Fig. 8-2(a). If a connection can be established, it means the port is open. Now look at Fig. 8-2(b). A SYN/ACK reply implies the port is open. Finally, we have Fig. 8-2(c). An RST reply means the port is open.

Probing for open ports is a first step. The next thing the attacker wants to know is exactly what server runs on this port, what software, what version of the software, and on what operating system. For instance, suppose we find that port 8080 is open. This is probably a Web server, although this is not certain. Even if it is a Web server, which one is it: Nginx, Lighttpd, Apache? Suppose an attacker only has an exploit for Apache version 2.4.37 and only on Windows, finding out all these details, known as **fingerprinting** is important. Just like in our port scans, we do so by making use of (sometimes subtle) differences in the way these servers and operating systems reply. If all of this sounds complicated, do not worry. Like many complicated things in computer networks, some helpful soul has sat down and

implemented all these scanning and fingerprinting techniques for you in friendly and versatile programs such as **netmap** and **zmap**.

Traceroute

Knowing which services are active on one machine is fine and dandy, but what about the rest of the machines in the network? Given knowledge of that first IP address, attackers may try to ''poke around'' to see what else is available. For instance, if the first machine has IP address 130.37.193.191, they might also try 130.37.193.192, 130.37.193.193, and all other possible addresses on the local network. Moreover, they can use programs such as **traceroute** to find the path toward the original IP address. Traceroute first sends a small batch of UDP packets to the target with the time-to-live (TTL) value set to one, then another batch with the TTL set to two, then a batch with a TTL of three, and so on. The first router lowers the TTL and immediately drops the first packets (because the TTL has now reached zero), and sends back an ICMP error message indicating that the packets have outlived their allocated life span. The second router does the same for the second batch of packets, the third for the third batch, until eventually some UDP packets reach the target. By collecting the ICMP error packets and their source IP addresses, traceroute is able to stitch together the overall route. Attackers can use the results to scan even more targets by probing address ranges of routers close to the target, thus obtaining a rudimentary knowledge of the network topology.

8.2.2 Sniffing and Snooping (with a Dash of Spoofing)

Many network attacks start with the interception of network traffic. For this attack ingredient, we assume that the attacker has a presence in the victim's network. For instance, the attacker brings a laptop in range of the victim's WiFi network, or obtains access to a PC in the wired network. Sniffing on a broadcast medium, such as WiFi or the original Ethernet implementation is easy: you just tune into the channel at a convenient location, and listen for the bits come thundering by. To do so, attackers set their network interfaces in **promiscuous mode**, to make it accept all packets on the channel, even those destined for another host, and use tools such as **tcpdump** or **Wireshark** to capture the traffic.

Sniffing in Switched Networks

However, in many networks, things are not so easy. Take modern Ethernet as an example. Unlike its original incarnations, Ethernet today is no longer a proper shared-medium network technology. All communication is switched and attackers, even if they are connected to the same network segment, will never receive any of the Ethernet frames destined for the other hosts on the segment. Specifically, recall

that Ethernet switches are self-learning and quickly build up a forwarding table. The self-learning is simple and effective: as soon as an Ethernet frame from host *A* arrives at port 1, the switch records that traffic for host *A* should be sent on port 1. Now it knows that all traffic with host *A*'s MAC address in the destination field of the Ethernet header should be forwarded on port 1. Likewise, it will send the traffic for host *B* on port 2, and so on. Once the forwarding table is complete, the switch will no longer send any traffic explicitly addressed to host *B* on any port other than 2. To sniff traffic, attackers must find a way to make exactly that happen.

There are several ways for an attacker to overcome the switching problem. They all use **spoofing**. Nevertheless, we will discuss them in this section, since the sole goal here is to sniff traffic.

The first is **MAC cloning**, duplicating the MAC address of the host of which you want to sniff the traffic. If you claim to have this MAC address (by sending out Ethernet frames with that address), the switch will duly record this in its table and henceforth send all traffic bound for the victim to your machine instead. Of course, this assumes that you know this address, but you should be able to obtain it from the ARP requests sent by the target that are, after all, broadcast to all hosts in the network segment. Another complicating factor is that your mapping will be removed from the switch as soon as the original owner of the MAC address starts communicating again, so you will have to repeat this **switch table poisoning** constantly.

As an alternative, but in the same vein, attackers can use the fact that the switch table has a limited size and flood the switch with Ethernet frames with fake source addresses. The switch does not know the MAC addresses are fake and simply records them until the table is full, evicting older entries to include the new ones if need be. Since the switch now no longer has an entry for the target host, it reverts to broadcast for all traffic towards it. **MAC flooding** makes your Ethernet behave like a broadcast medium again and party like it is 1979.

Instead of confusing the switch, attackers can also target hosts directly in a so-called **ARP spoofing** or **ARP poisoning** attack. Recall from Chap. 5 that the ARP protocol helps a computer find the MAC address corresponding to an IP address. For this purpose, the ARP implementation on a machine maintains a table with mappings from IP to MAC addresses for all hosts that have communicated with this machine (the **ARP table**). Each entry has a time-to-live (TTL) of, typically, a few tens of minutes. After that, the MAC address of the remote party is silently forgotten, assuming there is no further communication between these parties (in which case the TTL is reset), and all subsequent communication requires an ARP lookup first. The ARP lookup is simply a broadcast message that says something like: ''Folks, I am looking for the MAC address of the host with IP address 192.168.2.24. If this is you, please let me know.'' The lookup request contains the requester's MAC address, so host 192.168.2.24 knows where to send the reply, and also the requester's IP address, so 192.168.2.24 can add the IP to MAC address of the requester to its own ARP table.

Whenever the attacker sees such an ARP request for host 192.168.2.24, she can race to supply the requester with her own MAC address. In that case, all communication for 192.168.2.24 will be sent to the attacker's machine. In fact, since ARP implementations tend to be simple and stateless, the attacker can often just send ARP replies even if there was no request at all: the ARP implementation will accept the replies at face value and store the mappings in its ARP table.

By using this same trick on both communicating parties, the attacker receives all the traffic between them. By subsequently forwarding the frames to the right MAC addresses again, the attacker has installed a stealthy **MITM** (**Man-in-the-Middle**) gateway, capable of intercepting all traffic between the two hosts.

8.2.3 Spoofing (beyond ARP)

In general, spoofing means sending bytes over the network with a falsified source address. Besides ARP packets, attackers may spoof any other type of network traffic. For instance, SMTP (Simple Mail Transfer Protocol) is a friendly, text-based protocol that is used everywhere for sending email. It uses the Mail From: header as an indication of the source of an email, but by default it does not check this for correctness of the email address. In other words, you can put anything you want in this header. All replies will be sent to this address. Incidentally, the content of the Mail From: header is not even shown to the recipient of the email message. Instead, your mail client shows the content of a separate From: header. However, there is no check on this field either, and SMTP allows you to falsify it, so that the email that you send to your fellow students informing them that they failed the course appears to have been sent by the course instructor. If you additionally set the Mail From: header to your own email address, all replies sent by panicking students will end up in your mailbox. What fun you will have! Less innocently, criminals frequently spoof email to send phishing emails from seemingly trusted sources. That email from ''your doctor'' telling you to click on the link below to get urgent information about your medical test may lead to a site that says everything is normal, but fails to mention that it just downloaded a virus to your computer. The one from ''your bank'' can be bad for your financial health.

ARP spoofing occurs at the link layer, and SMTP spoofing at the application layer, but spoofing may happen at any layer in the protocol stack. Sometimes, spoofing is easy. For instance, anyone with the ability to craft custom packets can create fake Ethernet frames, IP datagrams, or UDP packets. You only need to change the source address and that is it: these protocols do not have any way to detect the tampering. Other protocols are much more challenging. For instance, in TCP connections the endpoints maintain state, such as the sequence and acknowledgement numbers, that make spoofing much trickier. Unless the attacker can sniff or guess the appropriate sequence numbers, the spoofed TCP segments will be rejected by the receiver as ''out-of-window.'' As we shall see later, there are substantial other difficulties as well.

Even the simple protocols allow attackers to cause a lot of damage. Shortly, we will see how spoofed UDP packets may lead to devastating **DoS** denial-of-Service attacks. First, however, we consider how spoofing permits attackers to intercept what clients send to a server by spoofing UDP datagrams in DNS.

DNS Spoofing

Since DNS uses UDP for its requests and replies, spoofing should be easy. For instance, just like in the ARP spoofing attack, we could wait for a client to send a lookup request for domain *trusted-services.com* and then race with the legitimate domain name system to provide a false reply that informs the client that *trusted-services.com* is located at an IP address owned by us. Doing so is easy if we can sniff the traffic coming from the client (and, thus, see the DNS lookup request to which to respond), but what if we cannot see the request? After all, if we can already sniff the communication, intercepting it via DNS spoofing is not that useful. Also, what if we want to intercept the traffic of many people instead of just one?

The simplest solution, if attackers share the local name server of the victim, is that they send their own request for, say, *trusted-services.com*, which in turn will trigger the local name server to do a lookup for this IP address on their behalf by contacting the next name server in the lookup process. The attackers immediately ''reply'' to this request by the local name server with a spoofed reply that appears to come from the next name server. The result is that the local name server stores the falsified mapping in its cache and serves it to the victim when it finally does the lookup for *trusted-services.com* (and anyone else who may be looking up the same name). Note that even if the attackers do not share the local name, the attack may still work, if the attacker can trick the victim into doing a lookup request with the attacker-provided domain name. For instance, the attacker could send an email that urges the victim to click on a link, so that the browser will do the name lookup for the attacker. After poisoning the mapping for *trusted-services.com*, all subsequent lookups for this domain will return the false mapping.

The astute reader will object that this is not so easy at all. After all, each DNS request carries a 16-bit query ID and a reply is accepted only if the ID in the reply matches. But if the attackers cannot see the request, they have to guess the identifier. For a single reply, the odds of getting it right is one in 65,536. On average, an attacker would have to send tens of thousands of DNS replies in a very short time, to falsify a single mapping at the local name server, and do so without being noticed. Not easy.

Birthday Attack

There is an easier way that is sometimes referred to as a birthday attack (or **birthday paradox**, even though strictly speaking it is not a paradox at all). The idea for this attack comes from a technique that math professors often use in their

probability courses. The question is: how many students do you need in a class before the probability of having two people with the same birthday exceeds 50%? Most of us expect the answer to be way over 100. In fact, probability theory says it is just 23. With 23 people, the probability of *none* of them having the same birthday is:

$$\frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} = 0.497203$$

In other words, the probability of two students celebrating their birthday on the same day is over 50%.

More generally, if there is some mapping between inputs and outputs with n inputs (people, identifiers, etc.) and k possible outputs (birthdays, identifiers, etc.), there are $n(n-1)/2$ input pairs. If $n(n-1)/2 > k$, the chance of having at least one match is pretty good. Thus, approximately, a match is likely for $n > \sqrt{2k}$. The key is that rather than look for a match for one particular student's birthday, we compare everyone to everyone else and any match counts.

Using this insight, the attackers first send a few hundred DNS requests for the domain mapping they want to falsify. The local name server will try to resolve each of these requests individually by asking the next-level name server. This is perhaps not very smart, because why would you send multiple queries for the same domain, but few people have argued that name servers are smart, and this is how the popular BIND name server operated for a long time. Anyway, immediately after sending the requests, the attackers also send hundreds of spoofed ''replies'' for the lookup, each pretending to come from the next-level name server and carrying a different guess for the query ID. The local name server implicitly performs the many-to-many comparison for us because if any reply ID matches that of a request sent by the local name server, the reply will be accepted. Note how this scenario resembles that of the students' birthdays: the name server compares all requests sent by the local name server with all spoofed replies.

By poisoning the local name server for a particular Web site, say, the attackers obtain access to the traffic sent to this site for all clients of the name server. By setting up their own connections to the Web site and then relaying all communication from the clients and all communication from the server, they now serve as a stealthy man-in-the-middle.

Kaminsky Attack

Things may get even worse when attackers poison the mapping not just for a single Web site, but for an entire zone. The attack is known as Dan Kaminsky's DNS attack and it caused a huge panic among information security officers and network administrators the world over. To see why everybody got their knickers in a twist, we should go into DNS lookups in a little more detail.

Consider a DNS lookup request for the IP address of *www.cs.vu.nl*. Upon reception of this request, the local name server, in turn, sends a request either to the root name server or, more commonly, to the TLD (top-level domain) name server for the *.nl* domain. The latter is more common because the IP address of the TLD name server is often already in the local name server's cache. Figure 8-3 shows this request by the local name server (asking for an "A record" for the domain) in a recursive lookup with query 1337.

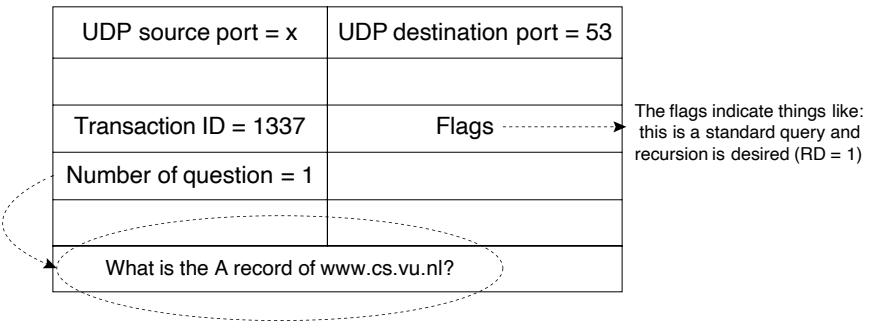

Figure 8-3. A DNS request for *www.cs.vu.nl*.

The TLD server does not know the exact mapping, but does know the names of the DNS servers of Vrije Universiteit which it sends back in a reply, since it does not do recursive lookups, thank you very much. The reply, shown in Fig. 8-4 has a few interesting fields to discuss. First, we observe, without going into details, that the flags indicate explicitly that the server does not want to do recursive lookups, so the remainder of the lookup will be iterative. Second, the query ID of the reply is also 1337, matching that of the lookup. Third, the reply provides the symbolic names of the name servers of the university *ns1.vu.nl* and *ns2.vu.nl* as NS records. These answers are authoritative and, in principle, suffice for the local name server to complete the query: by first performing a lookup for the A record of one of the name servers and subsequently contacting it, it can ask for the IP address of *www.cs.vu.nl*. However, doing so means that it will first contact the same TLD name server again, this time to ask for the IP address of the university's name server, and as this incurs an extra round trip time, it is not very efficient. To avoid this extra lookup, the TLD name server helpfully provides the IP addresses of the two university name servers as additional records in its reply, each with a short TTL. These additional records are known as **DNS glue records** and are the key to the Kaminsky attack.

Here is what the attackers will do. First, they send lookup requests for a non-existing subdomain of the university domain like:: *ohdeardankaminsky.vu.nl*. Since the subdomain does not exist, no name server can provide the mapping from its

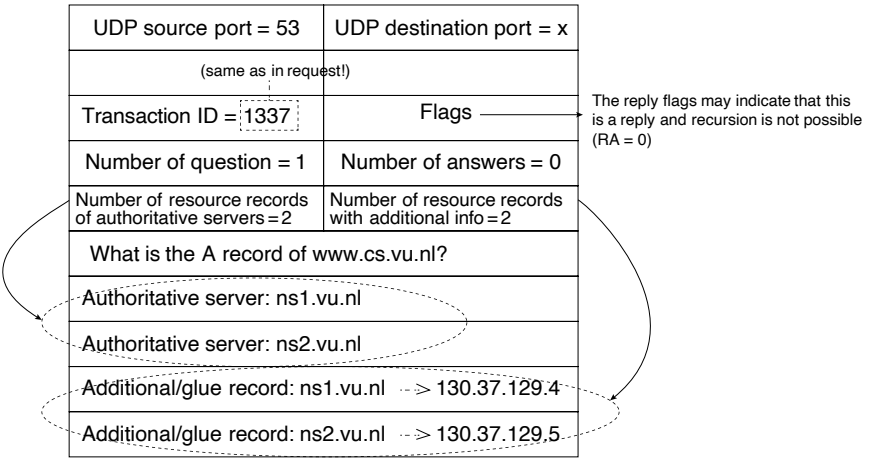

Figure 8-4. A DNS reply sent by the TLD name server.

cache. The local name server will instead contact the TLD name server. Immediately after sending the requests, the attackers also send many spoofed replies, pretending to be from the TLD name server, just like in a regular DNS spoofing request, except this time, the reply indicates that the TLD name server does not know the answer (i.e., it does not provide the A record), does not do recursive lookups, and advises the local name server to complete the lookup by contacting one of the university name servers. It may even provide the real names of these name servers. The only things they falsify are the glue records, for which they supply IP addresses that they control. As a result, every lookup for any subdomain of *.vu.nl* will contact the attackers' name server which can provide a mapping to any IP address it wants. In other words, the attackers are able to operate as man-in-the-middle for any site in the university domain!

While not all name server implementations were vulnerable to this attack, most of them were. Clearly, the Internet had a problem. An emergency meeting was hastily organized in Microsoft's headquarters in Redmond. Kaminsky later stated that all of this was shrouded in such secrecy that ''there were people on jets to Microsoft who didn't even know what the bug was.''

So how did these clever people solve the problem? The answer is, they didn't, not really. What they did do is make it harder. Recall that a core problem of these DNS spoofing attacks is that the query ID is only 16 bits, making it possible to guess it, either directly or by means of a birthday attack. A larger query ID makes the attack much less likely to succeed. However, simply changing the format of the DNS protocol message is not so easy and would also break many existing systems.

The solution was to extend the length of the random ID without really extending the query ID, by instead introducing randomness also in the UDP source port. When sending out a DNS request to, say, the TLD name server, a patched name server would pick a random port out of thousands of possible port numbers and use that as the UDP source port. Now the attacker must guess not just the query ID, but also the port number and do so before the legitimate reply arrives. The 0x20 encoding that we described in Chap. 7 exploits the case-insensitive nature of DNS queries to add even more bits to the transaction ID.

Fortunately, **DNSSEC**, provides a more solid defense against DNS spoofing. DNSSEC consists of a collection of extensions to DNS that offer both integrity and origin authentication of DNS data to DNS clients. However, DNSSEC deployment has been extremely slow. The initial work on DNSSEC was conducted in the early 1990s and the first RFC was published by the IETF in 1997; DNSSEC is now starting to see more widespread deployment, as we will discuss later in this chapter.

TCP Spoofing

Compared to the protocols discussed so far, spoofing in TCP is infinitely more complicated. When attackers want to pretend that a TCP segment came from another computer on the Internet, they not only have to guess the port number, but also the correct sequence numbers. Moreover, keeping a TCP connection in good shape, while injecting spoofed TCP segments is very complicated. We distinguish between two cases:

1. **Connection spoofing.** The attacker sets up a *new* connection, pretending to be someone at a different computer.
2. **Connection hijacking.** The attacker injects data in a connection that already exists between two parties, pretending to be either of these two parties.

The best-known example of **TCP connection spoofing** was the attack by **Kevin Mitnick** against the San Diego Supercomputing Center (SDSC) on Christmas day 1994. It is one of the most famous hacks in history, and the subject of several books and movies. Incidentally, one of them is a fairly big-budget flick called ''Takedown,'' that is based on a book that was written by the system administrator of the Supercomputing Center. (Perhaps not surprisingly, the administrator in the movie is portrayed as a very cool guy). We discuss it here because it illustrates the difficulties in TCP spoofing quite well.

Kevin Mitnick had a long history of being an Internet bad boy before he set his sights on SDSC. Incidentally, attacking on Christmas day is generally a good idea because on public holidays there are fewer users and administrators around. After some initial reconnaissance, Mitnick discovered that an (X-terminal) computer in SDSC had a trust relationship with another (server) machine in the same center.

Fig. 8-5(a) shows the configuration. Specifically, the server was implicitly trusted and anyone on the server could log in on the X-terminal as administrator using remote shell (*rsh*) without the need to enter a password. His plan was to set up a TCP connection to the X-terminal, pretending to be the server and use it to turn off password protection altogether—in those days, this could be done by writing "+ +" in the *.rhosts* file.

Doing so, however, was not easy. If Mitnick had sent a spoofed TCP connection setup request (a SYN segment) to the X-terminal with the IP address of the server (step 1 in Fig. 8-5(b)), the X-terminal would have sent its SYN/ACK reply to the actual server, and this reply would have been invisible to Mitnick (step 2 in Fig. 8-5(b)). As a result, he would not know the X-terminal's initial sequence number (ISN), a more-or-less random number that he would need for the third phase of the TCP handshake (which as we saw earlier, is the first segment that may contain data). What is worse, upon reception of the SYN/ACK, the server would have immediately responded with an RST segment to terminate the connection setup (step 3 in Fig. 8-5(c)). After all, there must have been a problem, as it never sent a SYN segment.

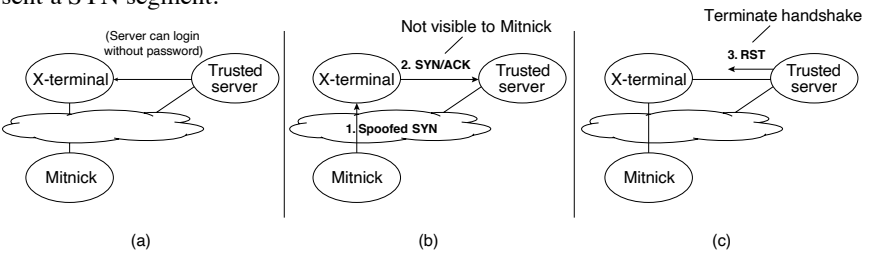

Figure 8-5. Challenges faced by Kevin Mitnick during the attack on SDSC.

Note that the problem of the invisible SYN/ACK, and hence the missing initial sequence number (ISN), would not be a problem at all if the ISN would have been predictable. For instance, if it would start at 0 for every new connection. However, since the ISN was chosen more or less random for every connection, Mitnick needed to find out how it was generated in order to *predict* the number that the X-terminal would use in its invisible SYN/ACK to the server.

To overcome these challenges, Mitnick launched his attack in several steps. First, he interacted extensively with the X-terminal using nonspoofed SYN messages (step 1 in Fig. 8-6(a)). While these TCP connection attempts did not get him access to the machine, they did give him a sequence of ISNs. Fortunately for Kevin, the ISNs were not *that* random. He stared at the numbers for a while until he found a pattern and was confident that given one ISN, he would be able to predict the next one. Next, he made sure that the trusted server would not be able to reset his connection attempts by launching a DoS attack that made the server unresponsive (step 2 in Fig. 8-6(b)). Now the path was clear to launch his real attack.

After sending the spoofed SYN packet (step 3 in Fig. 8-6(b)), he predicted the sequence number that the X-terminal would be using in its SYN/ACK reply to the server (step 4 in Fig. 8-6(b)) and used this in the third and final step, where he sent the command *echo ''+ +'' >> .rhosts* as data to the port used by the remote shell daemon (step 5 in Fig. 8-6(c)). After that, he could log in from any machine without a password.

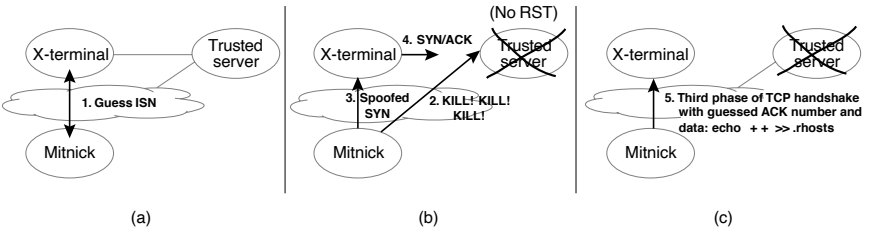

Figure 8-6. Mitnick's attack

Since one of the main weaknesses exploited by Mitnick was the predictability of TCP's initial sequence numbers, the developers of network stacks have since spent much effort on improving the randomness of TCP's choice for these security-sensitive numbers. As a result, the Mitnick attack is no longer practical. Modern attackers need to find a different way to guess the initial sequence numbers, for instance, the one employed in the connection hijacking attack we describe no

TCP Connection Hijacking

Compared to connection spoofing, connection hijacking adds even more hurdles to overcome. For now, let us assume that the attackers are able to eavesdrop on an existing connection between two communicating parties (because they are on the same network segment) and therefore know the exact sequence numbers and all other relevant information related to this communication. In a hijacking attack, the aim is to take over an existing connection, by injecting data into the stream.

To make this concrete, let us assume that the attacker wants to inject some data into the TCP connection that exists between a client who is logged in to a Web application at a server with the aim of making either the client or server receive attacker-injected bytes. In our example, the sequence numbers of the last bytes sent by the client and server are 1000 and 12,500, respectively. Assume that all data received so far have been acknowledged and the client and server are not currently sending any data. Now the attacker injects, say, 100 bytes into the TCP stream to the server, by sending a spoofed packet with the client's IP address and source port, as well as the server's IP address and source port. This 4-tuple is enough to make the network stack demultiplex the data to the right socket. In addition, the attacker provides the appropriate sequence number (1001) and acknowledgement number (12501), so TCP will pass the 100-byte payload to the Web server.

However, there is a problem. After passing the injected bytes to the application, the server will acknowledge them to the client: ''Thank you for the bytes, I am now ready to receive byte number 1101.'' This message comes as a surprise to the client, who thinks the server is confused. After all, it never sent any data, and still intends to send byte 1001. It promptly tells the server so, by sending an empty segment with sequence number 1001 and acknowledgement number 12501. ''Wow'' says the server, ''thanks, but this looks like an old ACK. By now, I already received the next 100 bytes. Best tell the remote party about this.'' It resends the ACK (seq = 1101, ack = 12501), which leads to another ACK by the client, and so on. This phenomenon is known as an **ACK storm**. It will never stop until one of the ACKs gets lost (because TCP does not retransmit dataless ACKs).

How does the attacker quell the ACK storm? There are several tricks and we will discuss all of them. The simplest one is to tear down the connection explicitly by sending an RST segment to the communicating parties. Alternatively, the attacker may be able to use ARP poisoning to cause one of the ACKs to be sent to a nonexisting address, forcing it to get lost. An alternative strategy is to desynchronize the two sides of the connection so much that all data sent by the client will be ignored by the server and vice versa. Doing so by sending lots of data is quite involved, but an attacker can easily accomplish this at the connection setup phase. The idea is as follows. The attacker waits until the client sets up a connection to the server. As soon as the server replies with a SYN/ACK, the attacker sends it an RST packet to terminate the connection, immediately followed by a SYN packet, with the same IP address and TCP source port as the ones originally used by the client, but a different client-side sequence number. After the subsequent SYN/ACK by the server, the server and client are both in the established state, but they cannot communicate with each other, because their sequence numbers are so far apart that they are always out-of-window. Instead, the attacker plays the role of man-in-the-middle and relays data between the two parties, able to inject data at will.

Off-Path TCP Exploits

Some of the attacks are very complex and hard to even understand, let alone defend against. In this section we will look at one of the more complicated ones. In most cases, attackers are not on the same network segment and cannot sniff the traffic between the parties. Attacks in such a scenario are known as off-path TCP exploits and are very tricky to pull off. Even if we ignore the ACK storm, the attacker needs a lot of information to inject data into an existing connection:

1. Even before the actual attack, the attackers should discover that there is a connection between two parties on the Internet to begin with.
2. Then they should determine the port numbers to use.
3. Finally, they need the sequence numbers.

Quite a tall order, if you are on the other side of the Internet, but not necessarily impossible, though. Decades after the Mitnick attack on SDSC, security researchers discovered a new vulnerability that permitted them to perform an off-path TCP exploit on widely deployed Linux systems. They described their attack in a paper titled ''Off-Path TCP Exploits: Global Rate Limit Considered Dangerous,'' which is a very apt title, as we shall see. We discuss it here because it illustrates that secret information can sometimes leak in an indirect way.

Ironically, the attack was made possible by a novel feature that was supposed to make the system more secure, not less secure. Recall that we said off-path data injections were very difficult because the attacker had to guess the port numbers and the sequence numbers and getting this right in a brute force attack is unlikely. Still, you just might get it right. Especially since you do not even have to get the sequence number exactly right, as long as the data you send is ''in-window.'' This means that with some (small) probability, attackers may reset, or inject data into existing connections. In August 2010, a new TCP extension appeared in the form of RFC 5961 to remedy this problem.

RFC 5961 changed how TCP handled the reception of SYN segments, RST segments, and regular data segments. The reason that the vulnerability existed only in Linux is that only Linux implemented the RFC correctly. To explain what it did, we should consider first how TCP worked before the extension. Let us consider the reception of SYN segments first. Before RFC 5961, whenever TCP received a SYN segment for an already existing connection, it would discard the packet if it was out-of-window, but it would reset the connection if it was in-window. The reason is that upon receiving a SYN segment, TCP would assume that the other side had restarted and thus that the existing connection was no longer valid. This is not good, as an attacker only needs to get one SYN segment with a sequence number somewhere in the receiver window to reset a connection. What RFC 5961 proposed instead was to not reset the connection immediately, but first send a **challenge ACK** to the apparent sender of the SYN. If the packet did come from the legitimate remote peer, it means that it really did lose the previous connection and is now setting up a new one. Upon receiving the challenge ACK, it will therefore send an RST packet with the correct sequence number. The attackers cannot do this since they never received the challenge ACK.

The same story holds for RST segments. In traditional TCP, hosts would drop the RST packets if they are out-of-window, and reset the connection if they are in-window. To make it harder to reset someone else's connection, RFC 5961 proposed to reset the connection immediately only if the sequence number in the RST segment was exactly the one at the start of the receiver window (i.e., next expected sequence number). If the sequence number is not an exact match, but still in-window, the host does not drop the connection, but sends a challenge ACK. If the sender is legitimate, it will send a RST packet with the right sequence number.

Finally, for data segments, old-style TCP conducts two checks. First, it checks the sequence number. If that was in-window, it also checks the acknowledgement

number. It considers acknowledgement numbers valid as long as they fall in an (enormous) interval. Let us denote the sequence numbers of the first unacknowledged byte by *FUB* and the sequence number of the next byte to be sent by *NEXT*. All packets with acknowledgement numbers in $[FUB - 2GB, NEXT]$ are valid, or half the ACK number space. This is easy to get right for an attacker! Moreover, if the acknowledgement number also happens to be in-window, it would process the data and advance the window in the usual way. Instead, RFC 5961 says that while we should accept packets with acknowledgement numbers that are (roughly) in-window, we should send challenge ACKs for the ones that are in the window $[FUB - 2GB, FUB - MAXWIN]$, where MAXWIN is the largest window ever advertised by the peer.

The designers of the protocol extension quickly recognized that it may lead to a huge number of challenge ACKs, and proposed ACK throttling as a solution. In the implementation of Linux, this meant that it would send at most 100 challenge ACKs per second, across all connections. In other words, a global variable shared by all connections kept track of how many challenge ACKs were sent and if the counter reached 100, it would send no more challenge ACKs for that one-second interval, whatever happened.

All this sounds good, but there is a problem. A single global variable represents shared state that can serve as a side channel for clever attacks. Let us take the first obstacle the attackers must overcome: are the two parties communicating? Recall that a challenge ACK is sent in three scenarios:

1. A SYN segment has the right source and destination IP addresses and port numbers, regardless of the sequence number.
2. A RST segment where the sequence number is in-window.
3. A data segment where additionally the acknowledgement number is in the challenge window.

Let us say that the attackers want to know whether a user at 130.37.20.7 is talking to a Web server (destination port 80) at 37.60.194.64. Since the attackers need not get the sequence number right, they only need to guess the source port number. To do so, they set up their own connection to the Web server and send 100 RST packets in quick succession, in response to which the server sends 100 challenge ACKs, unless it has already sent some challenge ACKs, in which case it would send fewer. However, this is quite unlikely. In addition to the 100 RSTs, the attackers therefore send a spoofed SYN segment, pretending to be the client at 130.37.20.7, with a guessed port number. If the guess is wrong, nothing happens and the attackers will still receive the 100 challenge ACKs. However, if they guessed the port number correctly, we end up in scenario (1), where the server sends a challenge ACK to the legitimate client. But since the server can only send 100 challenge ACKs per second, this means that the attackers receive only 99. In other words, by counting the number of challenge ACKs, the attackers can determine not just that the two hosts

are communicating, but even the (hidden) source port number of the client. Of course, you need quite a few tries to get it right, but this is definitely doable. Also, there are various techniques to make this more efficient.

Once the attackers have the port number they can move to the next phase of the attack: guessing the sequence and acknowledgement numbers. The idea is quite similar. For the sequence number the attackers again send 100 legitimate RST packets (spurring the server into sending challenge ACKs) and an additional spoofed RST packet with the right IP addresses and now known port numbers, as well as a guessed sequence number. If the guess is in-window, we are in scenario 2. Thus, by counting the challenge ACKs the attackers receive, they can determine whether the guess was correct.

Finally, for the acknowledgement number they send, in addition to the 100 RST packets, a data packet with all fields filled in correctly, but with a guess for the acknowledgement number, and apply the same trick. Now the attackers have all the information they need to reset the connection, or inject data.

The off-path TCP attack is a good illustration of three things. First, it shows how crazy complicated network attacks may get. Second, it is an excellent example of a network-based **side-channel attack**. Such attacks leak important information in an indirect way. In this case, the attackers learned all the connection details by counting something that appears very unrelated. Third, the attack shows that global shared state is the core problem of such side-channel attacks. Side-channel vulnerabilities appear everywhere, in both software and hardware, and in all cases, the root cause is the sharing of some important resource. Of course, we knew this already, as it is a violation of Saltzer and Schroeder's general principle of least common mechanism which we discussed in the beginning of this chapter. From a security perspective, it is good to remember that often sharing is not caring!

Before we move to the next topic (disruption and denial of service), it is good to know that data injection is not just nice in theory, it is actively used in practice. After the revelations by Edward Snowden in 2013, it became clear that the NSA (National Security Agency) ran a mass surveillance operation. One of its activities was Quantum, a sophisticated network attack that used packet injection to redirect targeted users connecting to popular services (such as *Twitter, Gmail*, or *Facebook*) to special servers that would then hack the victims' computers to give the NSA complete control. NSA denies everything, of course. It almost even denies its own existence. An industry joke goes:

Q: What does NSA stand for?
A: No Such Agency

8.2.4 Disruption

Attacks on availability are known as denial-of-service" attacks. They occur when a victim receives data it cannot handle, and as a result, becomes unresponsive. There are various reasons why a machine may stop responding:

1. **Crashes**. The attacker sends content that causes the victim to crash or hang. An example of such an attack was the ping of death we discussed earlier.
2. **Algorithmic complexity**. The attacker sends data that is crafted specifically to create a lot of (algorithmic) overhead. Suppose a server allows clients to send rich search queries. In that case, an algorithmic complexity attack may consist of a number of complicated regular expressions that incur the worst-case search time for the server.
3. **Flooding/swamping**. The attacker bombards the victim with such a massive flood of requests or replies that the poor system cannot keep up. Often, but not always, the victim eventually crashes.

Flooding attacks have become a major headache for organizations because these days it is very easy and cheap to carry out large-scale DoS attacks. For a few dollars or euros, you can rent a botnet consisting of many thousands of machines to attack any address you like. If the attack data is sent from a large number of distributed machines, we refer to the attack as a **DDoS**, (**Distributed Denial-of-Service**) attack. Specialized services on the Internet, known as **booters** or **stressers**, offer user-friendly interfaces to help even nontechnical users to launch them.

SYN Flooding

In the old days, DDoS attacks were quite simple. For instance, you would use a large number of hacked machines to launch a SYN flooding attack. All of these machines would send TCP SYN segments to the server, often spoofed to make it appear as if they came from different machines. While the server responded with a SYN/ACK, nobody would complete the TCP handshake, leaving the server dangling. That is quite expensive. A host can only keep a limited number of connections in the half-open state. After that, it no longer accepts new connections.

There are many solutions for SYN flooding attacks. For instance, we may simply drop half-open connections when we reach a limit to give preference to new connections or reduce the SYN-received timeout. An elegant and very simple solution, supported by many systems today goes by the name of **SYN cookies**, also briefly discussed in Chap. 6. Systems protected with SYN cookies use a special algorithm to determine the initial sequence number in such a way that the server does not need to remember *anything* about a connection until it receives the third packet in the three-way handshake. Recall that a sequence number is 32 bits wide. With SYN cookies, the server chooses the initial sequence number as follows:

1. The top 5 bits are the value of t *modulo* 32, where t is a slowly incrementing timer (e.g., a timer that increases every 64 seconds).
2. The next 3 bits are an encoding of the MSS (maximum segment size), giving eight possible values for the MSS.

3. The remaining 24 bits are the value of a cryptographic hash over the timestamp t and the source and destination IP addresses and port numbers.

The advantage of this sequence number is that the server can just stick it in a SYN/ACK and forget about it. If the handshake never completes, it is no skin off its back (or off whatever it is the server has on its back). If the handshake does complete, containing its own sequence number plus one in the acknowledgement, the server is able to reconstruct all the state it requires to establish the connection. First, it checks that the cryptographic hash matches a recent value of t and then quickly rebuilds the SYN queue entry using the MSS encoded in the 3 bits. While SYN Cookies allow only eight different segment sizes and make the sequence number grow faster than usual, the impact is minimal in practice. What is particularly nice is that the scheme is compatible with normal TCP and does not require the client to support the same extension.

Of course, it is still possible to launch a DDoS attack even in the presence of SYN cookies by completing the handshake, but this is more expensive for the attackers (as their own machines have limits on open TCP connections also), and more importantly, prevents TCP attacks with spoofed IP addresses.

Reflection and Amplification in DDoS Attacks

However, TCP-based DDoS attacks are not the only game in town. In recent years, more and more of the large-scale DDoS attacks have used UDP as the transport protocol. Spoofing UDP packets is typically easy. Moreover, with UDP it is possible to trick legitimate servers on the Internet to launch so-called **reflection attacks** on a victim. In a reflection attack, the attacker sends a request with a spoofed source address to a legitimate UDP service, for instance, a name server. The server will then reply to the spoofed address. If we do this from a large number of servers, the deluge of UDP reply packets is more than likely to take down the victim. Reflection attacks have two main advantages.

1. By adding the extra level of indirection, the attacker makes it difficult for the victim to block the senders somewhere in the network (after all, the senders are all legitimate servers).
2. Many services can *amplify* the attack by sending large replies to small requests.

These amplification-based DDoS attacks have been responsible for some of the largest volumes of DDoS attack traffic in history, easily reaching into the Terabit-per-second range. What the attacker must do for a successful amplification attack is to look for publicly accessible services with a large amplification factor. For instance, where one small request packet becomes a large reply packet, or

better still, multiple large reply packets. The byte amplification factor represents the relative gain in bytes, while the packet amplification factor represents the relative gain packets. Figure 8-7 shows the amplification factors for several popular protocols. While these numbers may look impressive, it is good to remember that these are averages and individual servers may have even higher ones. Interestingly, DNSSEC, the protocol that was intended to fix the security problems of DNS, has a much higher amplification factor than plain old DNS, exceeding 100 for some servers. Not to be outdone, misconfigured *memcached* servers (fast in-memory databases), clocked an amplification factor well exceeding 50,000 during a massive amplification attack of 1.7 Tbps in 2018.

Protocol	Byte amplification	Packet amplification
NTP	556.9	3.8
DNS	54.6	2.1
Bittorrent	3.8	1.6

Figure 8-7. Amplification factors for popular protocols

Defending against DDoS Attacks

Defending against such enormous streams of traffic is not easy, but several defenses exist. One, fairly straightforward technique is to block traffic close to the source. The most common way to do so is using a technique called **egress filtering**, whereby a network device such as a firewall blocks all outgoing packets whose source IP addresses do not correspond to those inside the network where it is attached. This, of course, requires the firewall to know what packets could possibly arrive with a particular source IP address, which is typically only possible at the edge of the network; for example, a university network might know all IP address ranges on its campus network and could thus block outgoing traffic from any IP address that it did not own. The dual to egress filtering is **ingress filtering**, whereby a network device filters all incoming traffic with internal IP addresses.

Another measure we can take is to try and absorb the DDoS attack with spare capacity. Doing so is expensive and may be unaffordable on an individual basis, for all but the biggest players. Fortunately, there is no reason to do this individually. By pooling resources that can be used by many parties, even smaller players can afford DDoS protection. Like insurance, the assumption is that not everybody will be attacked at the same time.

So what insurance will you get? Several organizations offer to protect your Web site by means of **cloud-based DDoS protection** which uses the strength of the cloud, to scale up capacity as and when needed, to fend off DoS attacks. At its core, the defense consists of the cloud shielding and even hiding the IP address of the real server. All requests are sent to proxies in the cloud that filter out the

malicious traffic the best they can (although doing so may not be so easy for advanced attacks), and forward the benign requests to the real server. If the number of requests or the amount of traffic for a specific server increases, the cloud will allocate more resources to handling these packets. In other words, the cloud ''absorbs'' the flood of data. Typically, it may also operate as a **scrubber** to sanitize the data as well. For instance, it may remove overlapping TCP segments or weird combinations of TCP flags, and serve in general as a **WAF** (**Web Application Firewall**).

To relay the traffic via the cloud-based proxies Web site owners can choose between several options with different price tags. If they can afford it, they can opt for **BGP blackholing**. In this case, the assumption is that the Web site owner controls an entire /24 block of (16,777,216) addresses. The idea is that the owner simply withdraws the BGP announcements for that block from its own routers. Instead, the cloud-based security provider starts announcing this IP from *its* network, so that all traffic for the server will go to the cloud first. However, not everybody has entire network blocks to play around with, or can afford the cost of BGP rerouting. For them, there is the more economical option to use **DNS rerouting**. In this case, the Web site's administrators change the DNS mappings in their name servers to point to servers in the cloud, rather than the real server. In either case, visitors will send their packets to the proxies owned by the cloud-based security provider first and these cloud-based proxies subsequently forward the packets to the real server.

DNS rerouting is easier to implement, but the security guarantees of the cloud-based security provider are only strong if the real IP address of the server remains hidden. If the attackers obtain this address, they can bypass the cloud and attack the server directly. Unfortunately, there are many ways in which the IP address may leak. Like FTP, some Web applications send the IP address to the remote party in-band, so there is not a lot one could do in those cases. Alternatively, attackers could look at historical DNS data to see what IP addresses were registered for the server in the past. Several companies collect and sell such historical DNS data.

8.3 FIREWALLS AND INTRUSION DETECTION SYSTEMS

The ability to connect any computer, anywhere, to any other computer, anywhere, is a mixed blessing. For individuals at home, wandering around the Internet is lots of fun. For corporate security managers, it is a nightmare. Most companies have large amounts of confidential information online—trade secrets, product development plans, marketing strategies, financial analyses, tax records, etc. Disclosure of this information to a competitor could have dire consequences.

In addition to the danger of information leaking out, there is also a danger of information leaking in. In particular, viruses, worms, and other digital pests can breach security, destroy valuable data, and waste large amounts of administrators'

time trying to clean up the mess they leave. Often they are imported by careless employees who want to play some nifty new game.

Consequently, mechanisms are needed to keep ''good'' bits in and ''bad'' bits out. One method is to use encryption, which protects data in transit between secure sites. However, it does nothing to keep digital pests and intruders from getting onto the company's LAN. To see how to accomplish this goal, we need to look at firewalls.

8.3.1 Firewalls

Firewalls are just a modern adaptation of that old medieval security standby: digging a wide and deep moat around your castle. This design forced everyone entering or leaving the castle to pass over a single drawbridge, where they could be inspected by the I/O police. With networks, the same trick is possible: a company can have many LANs connected in arbitrary ways, but all traffic to or from the company is forced through an electronic drawbridge (firewall), as shown in Fig. 8-8. No other route exists.

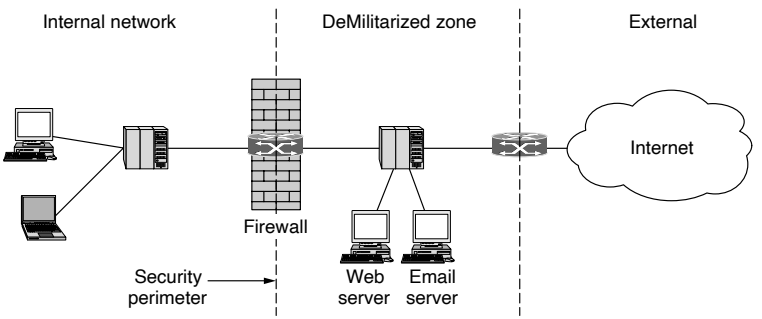

Figure 8-8. A firewall protecting an internal network.

The firewall acts as a **packet filter**. It inspects each and every incoming and outgoing packet. Packets meeting some criterion described in rules formulated by the network administrator are forwarded normally. Those that fail the test are unceremoniously dropped.

The filtering criterion is typically given as rules or tables that list sources and destinations that are acceptable, sources and destinations that are blocked, and default rules about what to do with packets coming from or going to other machines. In the common case of a TCP/IP setting, a source or destination might consist of an IP address and a port. Ports indicate which service is desired. For example, TCP port 25 is for mail, and TCP port 80 is for HTTP. Some ports can simply be blocked outright. For example, a company could block incoming packets for all IP addresses combined with TCP port 79. It was once popular for the Finger service

to look up people's email addresses but is barely used today due to its role in a now-infamous (accidental) attack on the Internet in 1988.

Other ports are not so easily blocked. The difficulty is that network administrators want security but cannot cut off communication with the outside world. That arrangement would be much simpler and better for security, but there would be no end to user complaints about it. This is where arrangements such as the **DMZ** (**DeMilitarized Zone**) shown in Fig. 8-8 come in handy. The DMZ is the part of the company network that lies outside of the security perimeter. Anything goes here. By placing a machine such as a Web server in the DMZ, computers on the Internet can contact it to browse the company Web site. Now the firewall can be configured to block incoming TCP traffic to port 80 so that computers on the Internet cannot use this port to attack computers on the internal network. To allow the Web server to be managed, the firewall can have a rule to permit connections between internal machines and the Web server.

Firewalls have become much more sophisticated over time in an arms race with attackers. Originally, firewalls applied a rule set independently for each packet, but it proved difficult to write rules that allowed useful functionality but blocked all unwanted traffic. **Stateful firewalls** map packets to connections and use TCP/IP header fields to keep track of connections. This allows for rules that, for example, allow an external Web server to send packets to an internal host, but only if the internal host first establishes a connection with the external Web server. Such a rule is not possible with stateless designs that must either pass or drop all packets from the external Web server.

Another level of sophistication up from stateful processing is for the firewall to implement **application-level gateways**. This processing involves the firewall looking inside packets, beyond even the TCP header, to see what the application is doing. With this capability, it is possible to distinguish HTTP traffic used for Web browsing from HTTP traffic used for peer-to-peer file sharing. Administrators can write rules to spare the company from peer-to-peer file sharing but allow Web browsing that is vital for business. For all of these methods, outgoing traffic can be inspected as well as incoming traffic, for example, to prevent sensitive documents from being emailed outside of the company.

As the above discussion should make abundantly clear, firewalls violate the standard layering of protocols. They are network layer devices, but they peek at the transport and applications layers to do their filtering. This makes them fragile. For instance, firewalls tend to rely on standard port numbering conventions to determine what kind of traffic is carried in a packet. Standard ports are often used, but not by all computers, and not by all applications either. Some peer-to-peer applications select ports dynamically to avoid being easily spotted (and blocked). Moreover, encryption hides higher-layer information from the firewall. Finally, a firewall cannot readily talk to the computers that communicate through it to tell them what policies are being applied and why their connection is being dropped. It must simply pretend that it is a broken wire. For these reasons, networking purists

consider firewalls to be a blemish on the architecture of the Internet. However, the Internet can be a dangerous place if you are a computer. Firewalls help with that problem, so they are likely to stay.

Even if the firewall is perfectly configured, plenty of security problems still exist. For example, if a firewall is configured to allow in packets from only specific networks (e.g., the company's other plants), an intruder outside the firewall can spoof the source addresses to bypass this check. If an insider wants to ship out secret documents, he can encrypt them or even photograph them and ship the photos as JPEG files, which bypasses any email filters. And we have not even discussed the fact that, although three-quarters of all attacks come from outside the firewall, the attacks that come from inside the firewall, for example, from disgruntled employees, may be the most damaging (Verizon, 2009).

A different problem with firewalls is that they provide a single perimeter of defense. If that defense is breached, all bets are off. For this reason, firewalls are often used in a layered defense. For example, a firewall may guard the entrance to the internal network and each computer may also run its own firewall, too. Readers who think that one security checkpoint is enough clearly have not made an international flight on a scheduled airline recently. As a result, many networks now have multiple levels of firewall, all the way down to per-host firewalls—a simple example of **defense in depth**. Suffice it to say that in both airports and computer networks if attackers have to compromise multiple independent defenses, it is much harder for them to breach the entire system.

8.3.2 Intrusion Detection and Prevention

Besides firewalls and scrubbers, network administrators may deploy a variety of other defensive measures, such as intrusion detection systems and intrusion prevention systems, to be described shortly. As the name implies, the role of an **IDS** (**Intrusion Detection System**) is to detect attacks—ideally before they can do any damage. For instance, they may generate warnings early on, at the onset of an attack, when it observes port scanning or a brute force **ssh password attack** (where an attacker simply tries many popular passwords to try and log in), or when the IDS finds the signature of the latest and greatest exploit in a TCP connection. However, it may also detect attacks only at a later stage, when a system has already been compromised and now exhibits unusual behavior.

We can categorize intrusion detection systems by considering *where* they work and *how* they work. A **HIDS** (**Host-based IDS**) works on the end-point itself, say a laptop or server, and scans, for instance, the behavior of the software or the network traffic to and from a Web server only on that machine. In contrast, a **NIDS** (**Network IDS**) checks the traffic for a set of machines on the network. Both have advantages and disadvantages.

A NIDS is attractive because it protects many machines, with the ability to correlate events associated with different hosts, and does not use up resources on the

machines it protects. In other words, the IDS has no impact on the performance of the machines in its protection domain. On the other hand, it is difficult to handle issues that are system specific. As an example, suppose that a TCP connection contains overlapping TCP segments: packet A contains bytes 1–200 while packet B contains bytes 100–300. Clearly, there is overlap between the bytes in the payloads. Let us also assume that the bytes in the overlapping region are different. What is the IDS to do?

The real question is: which bytes will be used by the receiving host? If the host uses the bytes of packet A, the IDS should check these bytes for malicious content and ignore the ones in packet B. However, what if the host instead uses the bytes in packet B? And what if some hosts in the network take the bytes in packet A and some take the bytes in packet B? Even if the hosts are all the same and the IDS knows how they reassemble the TCP streams there may still be difficulties. Suppose all hosts will normally take the bytes in packet A. If the IDS looks at that packet, it is still wrong if the destination of the packet is two or three network hops away, and the TTL value in packet A is 1, so it never even reaches its destination. Tricks that attackers play with TTL, or with overlapping byte ranges in IP fragments or TCP segments, are called **IDS evasion** techniques.

Another problem with NIDS is encryption. If the network bytes are no longer decipherable, it becomes much harder for the IDS to determine if they are malicious. This is another example of one security measure (encryption) reducing the protection offered by another (IDS). As a work-around, administrators may give the IDS the encryption keys to the NIDS. This works, but is not ideal, as it creates additional key management headaches. Also, observe that the IDS sees *all* the network traffic and tends to contain a great many lines of code itself. In other words, it may form a very juicy target for attackers. Break the IDS and you get access to all network traffic!

A host-based IDS' drawbacks are that it uses resources at each machine on which it runs and that it sees only a small fraction of the events in the network. On the other hand, it does not suffer as much from evasion problems as it can check the traffic after it has been reassembled by the very network stack of the machine it is trying to protect. Also, in cases such as IPsec, where packets encrypted and decrypted in the network layer, the IDS may check the data after decryption.

Beside the different locations of an IDS, we also have some choice in *how* an IDS determines whether something poses a threat. There are two main categories. **Signature-based intrusion detection systems** use patterns in terms of bytes or sequences of packets that are symptoms of known attacks. If you know that a UDP packet to port 53 with 10 specific bytes at the start of the payload are part of an exploit E, an IDS can easily scan the network traffic for this pattern and raise an alert when it detects it. The alert is specific: (''I have detected E'') and has a high confidence (''I know that it is E''). However, with a signature-based IDS, you only detect threats that are known and for which a signature is available. Alternatively, an IDS may raise an alert if it sees *unusual* behavior. For instance, a computer that

normally only exchanges SMTP and DNS traffic with a few IP addresses, suddenly starts sending HTTP traffic to many completely unknown IP addresses outside the local network. An IDS may classify this as fishy. Since such **anomaly-based intrusion detection systems**, or anomaly detection systems for short, trigger on any abnormal behavior, they are capable of detecting new attacks as well as old ones. The disadvantage is that the alerts do not carry a lot of explanation. Hearing that ''something unusual happened in the network'' is much less specific and much less useful than learning that ''the security camera at the gate is now attacked being by the Hajime malware.''

An **IPS** (**Intrusion Prevention System**) should not only detect the attack, but also stop it. In that sense, it is a glorified **firewall**. For instance, when the IPS sees a packet with the Hajime signature it can drop it on the floor rather than allowing it to reach the security camera. To do so, the IPS should sit on the path towards the target and take decisions about accepting or dropping traffic ''on the fly.'' In contrast, an IDS may reside elsewhere in the network, as long as we mirror all the traffic so it sees it. Now you may ask: why bother? Why not simply deploy an IPS and be done with the threats entirely? Part of the answer is the performance: the processing at the IDS determines the speed of the data transfer. If you have very little time, you may not be able to analyze the data very deeply. More importantly, what if you get it wrong? Specifically, what if your IPS decides a connection contains an attack and drops it, even though it is benign? That is really bad if the connection is important, for example, when your business depends on it. It may be better to raise an alert and let someone look into it, to decide if it really was malicious.

In fact, it is important to know how often your IDS or IPS gets it right. If it raises too many false alerts (**false positives**) you may end up spending a lot of time and money chasing those. If, on the other hand, it plays conservative and often does not raise alerts when attacks do take place (**false negatives**), attackers may still easily compromise your system. The number of false positives (FPs) and false negatives (FNs) with respect to the true positives (TPs) and true negatives (TNs) determines the usefulness of your protection. We commonly express these properties in terms of **precision** and **recall**. Precision represents a metric that indicates how many of the alarms that you generated were justified. In mathematical terms: $P = TP/(TP + FP)$. Recall indicates how many of the actual attacks you detected: $R = TP/(TP + FN)$. Sometimes, we combine the two values in what is known as the **F-measure**: $F = 2PR/(P + R)$. Finally, we are sometimes simply interested in how often an IDS or IPS got things right. In that case, we use the **accuracy** as a metric: $A = (TP + TN)/total$.

While it is always true that high values for recall and high precision are better than low ones, the number of false negatives and false positives are typically somewhat inversely correlated: if one goes down, the other goes up. However, the trade-off for what acceptable ranges are varies from situation to situation. If you are the Pentagon, you care deeply about not getting compromised. In that case, you may be willing to chase down a few more false positives, as long as you do not have

many false negatives. If, on the other hand, you are a school, things may be less critical and you may choose to not spend your money on an administrator who spends most of his working days analyzing false alarms.

There is one final thing we need to explain about these metrics to make you appreciate the importance of false positives. We will use an analogy similar to the one introduced by Stefan Axelsson in an influential paper that explained why intrusion detection is difficult (Axelsson, 1999). Suppose that there is a disease that affects 1 in 100,000 people in practice. Anyone diagnosed with the disease dies within a month. Fortunately, there is a great test to see if someone is infected. The test has 99% accuracy: if a patient is sick (S) the test will be positive (in the medical world a positive test is a bad thing!) in 99% of the cases, while for healthy patients (H), the test will be negative (Neg) in 99% of the cases. One day you take the test and, blow me down, the test is positive (i.e., indicates Pos). The million dollar question: how bad is this? Phrased differently: should you say goodbye to friends and family, sell everything you own in a yard sale, and live a (short) life of debauchery for the remaining 30-odd days? Or not?

To answer this question we should look at the math. What we are interested in is the probability that you have the disease given that you tested positive: $P(S|Pos)$. What we know is:

$$P(Pos|S) = 0.99$$

$$P(Neg|H) = 0.99$$

$$P(S) = 0.00001$$

To calculate $P(S|Pos)$, we use the famous Bayes theorem:

$$P(S|Pos) = \frac{P(S)P(Pos|S)}{P(Pos)}$$

In our case, there are only two possible outcomes for the test and two possible outcomes for you having the disease. In other words

$$\begin{aligned} P(Pos) &= P(S)P(Pos|S) + P(H)P(Pos|H) \\ &\quad \textit{where}\ \ P(H) = 1 - P(S), \\ &\quad \text{and}\ \ P(Pos|H) = 1 - P(Neg|H),\ \textit{so that}: \\ P(Pos) &= P(S)P(Pos|S) + (1 - P(S))(1 - P(Neg|H)) \\ &= 0.00001 * 0.99 + 0.99999 * 0.01 \end{aligned}$$

so that

$$\begin{aligned} P(S|Pos) &= \frac{0.00001 * 0.99}{0.00001 * 0.99 + 0.99999 * 0.01} \\ &= 0.00098 \end{aligned}$$

In other words, the probability of you having the disease is less than 0.1%. No need to panic yet. (Unless of course you *did* prematurely sell all your belongings in an estate sale.)

What we see here is that the final probability is strongly dominated by the false positive rate $P(Pos|H) = 1 - P(Neg|H) = 0.01$. The reason is that the number of incidents is so small (0.00001) that all the other terms in the equation hardly count. The problem is referred to as the **Base Rate Fallacy**. If we substitute ''under attack'' for ''sick,'' and ''alert'' for ''positive test,'' we see that the base rate fallacy is extremely important for any IDS or IPS solution. It motivates the need for keeping the number of false positives low.

Besides the fundamental security principles by Saltzer and Schroeder, many people have offered additional, often very practical principles. One that is particularly useful to mention here is the pragmatic **principle of defense in depth**. Often it is a good idea to use multiple complementary techniques to protect a system. For instance, to stop attacks, we may use a firewall *and* an intrusion detection system *and* a virus scanner. While no single measure may be foolproof by itself, the idea is that it is much harder to bypass all of them at the same time.

8.4 CRYPTOGRAPHY

Cryptography comes from the Greek words for ''secret writing.'' It has a long and colorful history going back thousands of years. In this section, we will just sketch some of the highlights, as background information for what follows. For a complete history of cryptography, Kahn's (1995) book is recommended reading. For a comprehensive treatment of modern security and cryptographic algorithms, protocols, and applications, and related material, see Kaufman et al. (2002). For a more mathematical approach, see Kraft and Washington (2018). For a less mathematical approach, see Esposito (2018).

Professionals make a distinction between ciphers and codes. A **cipher** is a character-for-character or bit-for-bit transformation, without regard to the linguistic structure of the message. In contrast, a **code** replaces one word with another word or symbol. Codes are not used any more, although they have a glorious history.

The most successful code ever devised was used by the United States Marine Corps during World War II in the Pacific. They simply had Navajo Marines talking to each other in their native language using specific Navajo words for military terms, for example, *chay-da-gahi-nail-tsaidi* (literally: tortoise killer) for antitank weapon. The Navajo language is highly tonal, exceedingly complex, and has no written form. And not a single person in Japan knew anything about it. In September 1945, the *San Diego Union* published an article describing the previously secret use of the Navajos to foil the Japanese, telling how effective it was. The Japanese never broke the code and many Navajo code talkers were awarded

high military honors for extraordinary service and bravery. The fact that the U.S. broke the Japanese code but the Japanese never broke the Navajo code played a crucial role in the American victories in the Pacific.

8.4.1 Introduction to Cryptography

Historically, four groups of people have used and contributed to the art of cryptography: the military, the diplomatic corps, diarists, and lovers. Of these, the military has had the most important role and has shaped the field over the centuries. Within military organizations, the messages to be encrypted have traditionally been given to poorly paid, low-level code clerks for encryption and transmission. The sheer volume of messages prevented this work from being done by a few elite specialists.

Until the advent of computers, one of the main constraints on cryptography had been the ability of the code clerk to perform the necessary transformations, often on a battlefield with little equipment. An additional constraint has been the difficulty in switching over quickly from one cryptographic method to another, since this entails retraining a large number of people. However, the danger of a code clerk being captured by the enemy has made it essential to be able to change the cryptographic method instantly if need be. These conflicting requirements have given rise to the model of Fig. 8-9.

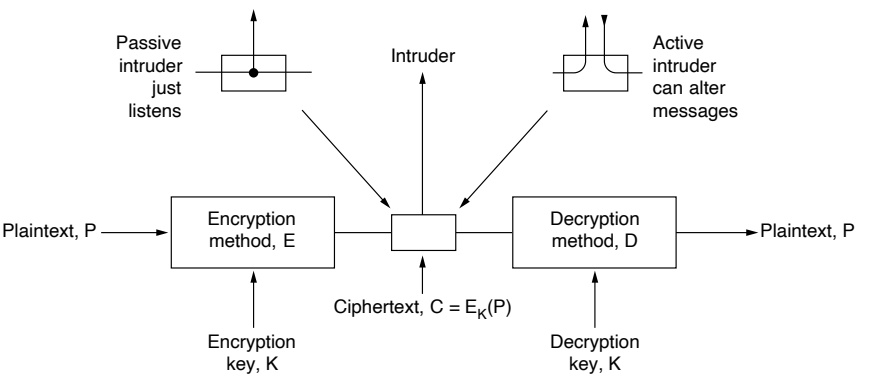

Figure 8-9. The encryption model (for a symmetric-key cipher).

The messages to be encrypted, known as the **plaintext**, are transformed by a function that is parametrized by a **key**. The output of the encryption process, known as the **ciphertext**, is then transmitted, often by messenger or radio. We assume that the enemy, or **intruder**, hears and accurately copies down the complete ciphertext. However, unlike the intended recipient, he does not know what the

decryption key is and so cannot decrypt the ciphertext easily. Sometimes the intruder can not only listen to the communication channel (passive intruder) but can also record messages and play them back later, inject his own messages, or modify legitimate messages before they get to the receiver (active intruder). The art of breaking ciphers, known as **cryptanalysis**, and the art of devising them (cryptography) are collectively known as **cryptology**.

It will often be useful to have a notation for relating plaintext, ciphertext, and keys. We will use $C = E_K(P)$ to mean that the encryption of the plaintext P using key K gives the ciphertext C. Similarly, $P = D_K(C)$ represents the decryption of C to get the plaintext again. It then follows that

$$D_K(E_K(P)) = P$$

This notation suggests that E and D are just mathematical functions, which they are. The only tricky part is that both are functions of two parameters, and we have written one of the parameters (the key) as a subscript, rather than as an argument, to distinguish it from the message.

A fundamental rule of cryptography is that one must assume that the cryptanalyst knows the methods used for encryption and decryption. In other words, the cryptanalyst knows how the encryption method, E, and decryption, D, of Fig. 8-9 work in detail. The amount of effort necessary to invent, test, and install a new algorithm every time the old method is compromised (or thought to be compromised) has always made it impractical to keep the encryption algorithm secret. Thinking it is secret when it is not does more harm than good.

This is where the key enters. The key consists of a (relatively) short string that selects one of many potential encryptions. In contrast to the general method, which may only be changed every few years, the key can be changed as often as required. Thus, our basic model is a stable and publicly known general method parametrized by a secret and easily changed key. The idea that the cryptanalyst knows the algorithms and that the secrecy lies exclusively in the keys is called **Kerckhoffs' principle**, named after the Dutch-born military cryptographer Auguste Kerckhoffs who first published it in a military journal 1883 (Kerckhoffs, 1883). Thus, we have

Kerckhoffs' principle: all algorithms must be public; only the keys are secret.

The nonsecrecy of the algorithm cannot be emphasized enough. Trying to keep the algorithm secret, known in the trade as **security by obscurity**, never works. Also, by publicizing the algorithm, the cryptographer gets free consulting from a large number of academic cryptologists eager to break the system so they can publish papers demonstrating how smart they are. If many experts have tried to break the algorithm for a long time after its publication and no one has succeeded, it is probably pretty solid. (On the other hand, researchers have found bugs in open source security solutions such as OpenSSL that were over a decade

old, so the common belief that ''given enough eyeballs, all bugs are shallow'' argument does not always work in practice.)

Since the real secrecy is in the key, its length is a major design issue. Consider a simple combination lock. The general principle is that you enter digits in sequence. Everyone knows this, but the key is secret. A key length of two digits means that there are 100 possibilities. A key length of three digits means 1000 possibilities, and a key length of six digits means a million. The longer the key, the higher the **work factor** the cryptanalyst has to deal with. The work factor for breaking the system by exhaustive search of the key space is exponential in the key length. Secrecy comes from having a strong (but public) algorithm and a long key. To prevent your kid brother from reading your email, perhaps even 64-bit keys will do. For routine commercial use, perhaps 256 bits should be used. To keep major governments at bay, much larger keys of at least 256 bits, and preferably more are needed. Incidentally, these numbers are for symmetric encryption, where the encryption and the decryption key are the same. We will discuss the differences between symmetric and asymmetric encryption later.

From the cryptanalyst's point of view, the cryptanalysis problem has three principal variations. When he has a quantity of ciphertext and no plaintext, he is confronted with the **ciphertext-only** problem. The cryptograms that appear in the puzzle section of newspapers pose this kind of problem. When the cryptanalyst has some matched ciphertext and plaintext, the problem is called the **known plaintext** problem. Finally, when the cryptanalyst has the ability to encrypt pieces of plaintext of his own choosing, we have the **chosen plaintext** problem. Newspaper cryptograms could be broken trivially if the cryptanalyst were allowed to ask such questions as ''What is the encryption of ABCDEFGHIJKL?''

Novices in the cryptography business often assume that if a cipher can withstand a ciphertext-only attack, it is secure. This assumption is very naive. In many cases, the cryptanalyst can make a good guess at parts of the plaintext. For example, the first thing many computers say when you boot them up is ''login:''. Equipped with some matched plaintext-ciphertext pairs, the cryptanalyst's job becomes much easier. To achieve security, the cryptographer should be conservative and make sure that the system is unbreakable even if his opponent can encrypt arbitrary amounts of chosen plaintext.

Encryption methods have historically been divided into two categories: substitution ciphers and transposition ciphers. We will now deal with each of these briefly as background information for modern cryptography.

8.4.2 Two Fundamental Cryptographic Principles

Although we will study many different cryptographic systems in the pages ahead, two principles underlying all of them are important to understand. Pay attention. You violate them at your peril.

Redundancy

The first principle is that all encrypted messages must contain some redundancy, that is, information not needed to understand the message. An example may make it clear why this is needed. Consider a mail-order company, The Couch Potato (TCP), with 60,000 products. Thinking they are being very efficient, TCP's programmers decide that ordering messages should consist of a 16-byte customer name followed by a 3-byte data field (1 byte for the quantity and 2 bytes for the product number). The last 3 bytes are to be encrypted using a very long key known only by the customer and TCP.

At first, this might seem secure, and in a sense it is because passive intruders cannot decrypt the messages. Unfortunately, it also has a fatal flaw that renders it useless. Suppose that a recently fired employee wants to punish TCP for firing her. Just before leaving, she takes the customer list with her. She works through the night writing a program to generate fictitious orders using real customer names. Since she does not have the list of keys, she just puts random numbers in the last 3 bytes, and sends hundreds of orders off to TCP.

When these messages arrive, TCP's computer uses the customers' name to locate the key and decrypt the message. Unfortunately for TCP, almost every 3-byte message is valid, so the computer begins printing out shipping instructions. While it might seem a bit odd for a customer to order 837 sets of children's swings or 540 sandboxes, for all the computer knows, the customer might be planning to open a chain of franchised playgrounds. In this way, an active intruder (the ex-employee) can cause a massive amount of trouble, even though she cannot understand the messages her computer is generating.

This problem can be solved by the addition of redundancy to all messages. For example, if order messages are extended to 12 bytes, the first 9 of which must be zeros, this attack no longer works because the ex-employee can no longer generate a large stream of valid messages. The moral of the story is that all messages must contain considerable redundancy so that active intruders cannot send random junk and have it be interpreted as a valid message. Thus we have:

Cryptographic principle 1: Messages must contain some redundancy

However, adding redundancy makes it easier for cryptanalysts to break messages. Suppose that the mail-order business is highly competitive, and The Couch Potato's main competitor, The Sofa Tuber, would dearly love to know how many sandboxes TCP is selling, so it taps TCP's phone line. In the original scheme with 3-byte messages, cryptanalysis was nearly impossible because after guessing a key, the cryptanalyst had no way of telling whether it was right because almost every message was technically legal. With the new 12-byte scheme, it is easy for the cryptanalyst to tell a valid message from an invalid one.

In other words, upon decrypting a message, the recipient must be able to tell whether it is valid by simply inspecting the message and perhaps performing a

simple computation. This redundancy is needed to prevent active intruders from sending garbage and tricking the receiver into decrypting the garbage and acting on the ''plaintext.''

However, this same redundancy makes it much easier for passive intruders to break the system, so there is some tension here. Furthermore, the redundancy should never be in the form of *n* 0s at the start or end of a message, since running such messages through some cryptographic algorithms gives more predictable results, making the cryptanalysts' job easier. A CRC polynomial (see Chapter 3) is much better than a run of 0s since the receiver can easily verify it, but it generates more work for the cryptanalyst. Even better is to use a cryptographic hash, a concept we will explore later. For the moment, think of it as a better CRC.

Freshness

The second cryptographic principle is that measures must be taken to ensure that each message received can be verified as being fresh, that is, sent very recently. This measure is needed to prevent active intruders from playing back old messages. If no such measures were taken, our ex-employee could tap TCP's phone line and just keep repeating previously sent valid messages. Thus,

Cryptographic principle 2: Some method is needed to foil replay attacks

One such measure is including in every message a timestamp valid only for, say, 60 seconds. The receiver can then just keep messages around for 60 seconds and compare newly arrived messages to previous ones to filter out duplicates. Messages older than 60 seconds can be thrown out, since any replays sent more than 60 seconds later will be rejected as too old. The interval should not be too short (e.g., 5 seconds) because the sender's and receiver's clocks may be slightly out of sync. Measures other than timestamps will be discussed later.

8.4.3 Substitution Ciphers

In a **substitution cipher**, each letter or group of letters is replaced by another letter or group of letters to disguise it. One of the oldest known ciphers is the **Caesar cipher**, attributed to Julius Caesar. With this method, *a* becomes *D*, *b* becomes *E*, *c* becomes *F*, . . . , and *z* becomes *C*. For example, *attack* becomes *DWWDFN*. In our examples, plaintext will be given in lowercase letters, and ciphertext in uppercase letters.

A slight generalization of the Caesar cipher allows the ciphertext alphabet to be shifted by *k* letters, instead of always three. In this case, *k* becomes a key to the general method of circularly shifted alphabets. The Caesar cipher may have fooled Pompey, but it has not fooled anyone since.

The next improvement is to have each of the symbols in the plaintext, say, the 26 letters for simplicity, map onto some other letter. For example,

plaintext: a b c d e f g h i j k l m n o p q r s t u v w x y z
ciphertext: Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

The general system of symbol-for-symbol substitution is called a **monoalphabetic substitution cipher**, with the key being the 26-letter string corresponding to the full alphabet. For the key just given, the plaintext *attack* would be transformed into the ciphertext *QZZQEA*.

At first glance, this might appear to be a safe system because although the cryptanalyst knows the general system (letter-for-letter substitution), he does not know which of the 26! $\approx 4 \times 10^{26}$ possible keys is in use. In contrast with the Caesar cipher, trying all of them is not a promising approach. Even at 1 nsec per solution, a million cores working in parallel would take 10,000 years to try all the keys.

Nevertheless, given a surprisingly small amount of ciphertext, the cipher can be broken easily. The basic attack takes advantage of the statistical properties of natural languages. In English, for example, *e* is the most common letter, followed by *t*, *o*, *a*, *n*, *i*, etc. The most common two-letter combinations, or **digrams**, are *th*, *in*, *er*, *re*, and *an*. The most common three-letter combinations, or **trigrams**, are *the*, *ing*, *and*, and *ion*.

A cryptanalyst trying to break a monoalphabetic cipher would start out by counting the relative frequencies of all letters in the ciphertext. Then he might tentatively assign the most common one to *e* and the next most common one to *t*. He would then look at trigrams to find a common one of the form *tXe*, which strongly suggests that *X* is *h*. Similarly, if the pattern *thYt* occurs frequently, the *Y* probably stands for *a*. With this information, he can look for a frequently occurring trigram of the form *aZW*, which is most likely *and*. By making guesses at common letters, digrams, and trigrams and knowing about likely patterns of vowels and consonants, the cryptanalyst builds up a tentative plaintext, letter by letter.

Another approach is to guess a probable word or phrase. For example, consider the following ciphertext from an accounting firm (blocked into groups of five characters):

```
CTBMN BYCTC BTJDS QXBNS GSTJC BTSWX CTQTZ CQVUJ
QJSGS TJQZZ MNQJS VLNSX VSZJU JDSTS JQUUS JUBXJ
DSKSU JSNTK BGAQJ ZBGYQ TLCTZ BNYBN QJSW
```

A likely word in a message from an accounting firm is *financial*. Using our knowledge that *financial* has a repeated letter (*i*), with four other letters between their occurrences, we look for repeated letters in the ciphertext at this spacing. We find 12 hits, at positions 6, 15, 27, 31, 42, 48, 56, 66, 70, 71, 76, and 82. However, only two of these, 31 and 42, have the next letter (corresponding to *n* in the plaintext) repeated in the proper place. Of these two, only 31 also has the *a* correctly positioned, so we know that *financial* begins at position 30. From this point on, deducing the key is easy by using the frequency statistics for English text and looking for nearly complete words to finish off.

8.4.4 Transposition Ciphers

Substitution ciphers preserve the order of the plaintext symbols but disguise them. **Transposition ciphers**, in contrast, reorder the letters but do not disguise them. Figure 8-10 depicts a common transposition cipher, the columnar transposition. The cipher is keyed by a word or phrase not containing any repeated letters. In this example, MEGABUCK is the key. The purpose of the key is to order the columns, with column 1 being under the key letter closest to the start of the alphabet, and so on. The plaintext is written horizontally, in rows, padded to fill the matrix if need be. The ciphertext is read out by columns, starting with the column whose key letter is the lowest.

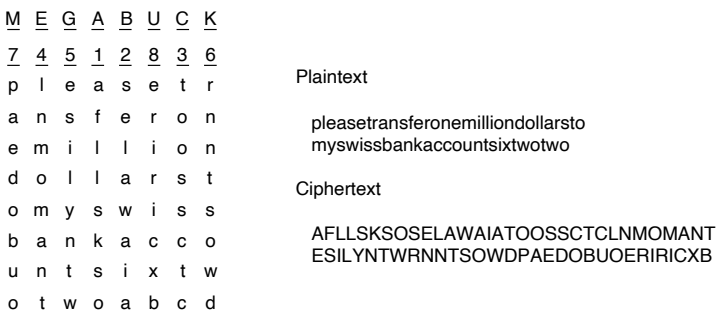

Figure 8-10. A transposition cipher.

To break a transposition cipher, the cryptanalyst must first be aware that he is dealing with a transposition cipher. By looking at the frequency of *E*, *T*, *A*, *O*, *I*, *N*, etc., it is easy to see if they fit the normal pattern for plaintext. If so, the cipher is clearly a transposition cipher because in such a cipher every letter represents itself, keeping the frequency distribution intact.

The next step is to make a guess at the number of columns. In many cases, a probable word or phrase may be guessed at from the context. For example, suppose that our cryptanalyst suspects that the plaintext phrase *milliondollars* occurs somewhere in the message. Observe that digrams *MO*, *IL*, *LL*, *LA*, *IR*, and *OS* occur in the ciphertext as a result of this phrase wrapping around. The ciphertext letter *O* follows the ciphertext letter *M* (i.e., they are vertically adjacent in column 4) because they are separated in the probable phrase by a distance equal to the key length. If a key of length seven had been used, the digrams *MD*, *IO*, *LL*, *LL*, *IA*, *OR*, and *NS* would have occurred instead. In fact, for each key length, a different set of digrams is produced in the ciphertext. By hunting for the various possibilities, the cryptanalyst can often easily determine the key length.

The remaining step is to order the columns. When the number of columns, k, is small, each of the $k(k-1)$ column pairs can be examined in turn to see if its digram frequencies match those for English plaintext. The pair with the best match is assumed to be correctly positioned. Now each of the remaining columns is tentatively tried as the successor to this pair. The column whose digram and trigram frequencies give the best match is tentatively assumed to be correct. The next column is found in the same way. The entire process is continued until a potential ordering is found. Chances are that the plaintext will be recognizable at this point (e.g., if *milloin* occurs, it is clear what the error is).

Some transposition ciphers accept a fixed-length block of input and produce a fixed-length block of output. These ciphers can be completely described by giving a list telling the order in which the characters are to be output. For example, the cipher of Fig. 8-10 can be seen as a 64 character block cipher. Its output is 4, 12, 20, 28, 36, 44, 52, 60, 5, 13, . . . , 62. In other words, the fourth input character, a, is the first to be output, followed by the twelfth, f, and so on.

8.4.5 One-Time Pads

Constructing an unbreakable cipher is actually quite easy; the technique has been known for decades. First, choose a random bit string as the key. Then convert the plaintext into a bit string, for example, by using its ASCII representation. Finally, compute the XOR (eXclusive OR) of these two strings, bit by bit. The resulting ciphertext cannot be broken because in a sufficiently large sample of ciphertext, each letter will occur equally often, as will every digram, every trigram, and so on. This method, known as the **one-time pad**, is immune to all present and future attacks, no matter how much computational power the intruder has. The reason derives from information theory: there is simply no information in the message because all possible plaintexts of the given length are equally likely.

An example of how one-time pads are used is given in Fig. 8-11. First, message 1, ''I love you.'' is converted to 7-bit ASCII. Then a one-time pad, pad 1, is chosen and XORed with the message to get the ciphertext. A cryptanalyst could try all possible one-time pads to see what plaintext came out for each one. For example, the one-time pad listed as pad 2 in the figure could be tried, resulting in plaintext 2, ''Elvis lives,'' which may or may not be plausible (a subject beyond the scope of this book). In fact, for every 11-character ASCII plaintext, there is a one-time pad that generates it. That is what we mean by saying there is no information in the ciphertext: you can get any message of the correct length out of it.

One-time pads are great in theory, but have a number of disadvantages in practice. To start with, the key cannot be memorized, so both sender and receiver must carry a written copy with them. If either one is subject to capture, written keys are clearly undesirable. Additionally, the total amount of data that can be transmitted is limited by the amount of key available. If the spy strikes it rich and discovers a wealth of data, he may find himself unable to transmit them back to headquarters

Message 1:	1001001	0100000	1101100	1101111	1110110	1100101	0100000	1111001	1101111	1110101	0101110
Pad 1:	1010010	1001011	1110010	1010101	1010010	1100011	0001011	0101010	1010111	1100110	0101011
Ciphertext:	0011011	1101011	0011110	0111010	0100100	0000110	0101011	1010011	0111000	0010011	0000101
Pad 2:	1011110	0000111	1101000	1010011	1010111	0100110	1000111	0111010	1001110	1110110	1110110
Plaintext 2:	1000101	1101100	1110110	1101001	1110011	0100000	1101100	1101001	1110110	1100101	1110011

Figure 8-11. The use of a one-time pad for encryption and the possibility of getting any possible plaintext from the ciphertext by the use of some other pad.

because the key has been used up. Another problem is the sensitivity of the method to lost or inserted characters. If the sender and receiver get out of synchronization, all data from then on will appear garbled.

With the advent of computers, the one-time pad might potentially become practical for some applications. The source of the key could be a special DVD that contains several gigabytes of information and, if transported in a DVD movie box and prefixed by a few minutes of video, would not even be suspicious. Of course, at gigabit network speeds, having to insert a new DVD every 30 sec could become tedious. And the DVDs must be personally carried from the sender to the receiver before any messages can be sent, which greatly reduces their practical utility. Also, given that very soon nobody will use DVD or Blu-Ray discs any more, anyone caught carrying around a box of them should perhaps be regarded with suspicion.

Quantum Cryptography

Interestingly, there may be a solution to the problem of how to transmit the one-time pad over the network, and it comes from a very unlikely source: quantum mechanics. This area is still experimental, but initial tests are promising. If it can be perfected and be made efficient, virtually all cryptography will eventually be done using one-time pads since they are provably secure. Below we will briefly explain how this method, **quantum cryptography**, works. In particular, we will describe a protocol called **BB84** after its authors and publication year (Bennet and Brassard, 1984).

Suppose that a user, Alice, wants to establish a one-time pad with a second user, Bob. Alice and Bob are called **principals**, the main characters in our story. For example, Bob is a banker with whom Alice would like to do business. The names ''Alice'' and ''Bob'' have been used for the principals in virtually every paper and book on cryptography since Ron Rivest introduced them many years ago (Rivest et al., 1978). Cryptographers love tradition. If we were to use ''Andy'' and ''Barbara'' as the principals, no one would believe anything in this chapter. So be it.

If Alice and Bob could establish a one-time pad, they could use it to communicate securely. The obvious question is: how can they establish it without having

previously exchanging them physically (using DVDs, books, or USB sticks)? We can assume that Alice and Bob are at the opposite ends of an optical fiber over which they can send and receive light pulses. However, an intrepid intruder, Trudy, can cut the fiber to splice in an active tap. Trudy can read all the bits sent in both directions. She can also send false messages in both directions. The situation might seem hopeless for Alice and Bob, but quantum cryptography can shed some new light on the subject.

Quantum cryptography is based on the fact that light comes in microscopic little packets called **photons**, which have some peculiar properties. Furthermore, light can be polarized by being passed through a polarizing filter, a fact well known to both sunglasses wearers and photographers. If a beam of light (i.e., a stream of photons) is passed through a polarizing filter, all the photons emerging from it will be polarized in the direction of the filter's axis (e.g., vertically). If the beam is now passed through a second polarizing filter, the intensity of the light emerging from the second filter is proportional to the square of the cosine of the angle between the axes. If the two axes are perpendicular, no photons get through. The absolute orientation of the two filters does not matter; only the angle between their axes counts.

To generate a one-time pad, Alice needs two sets of polarizing filters. Set one consists of a vertical filter and a horizontal filter. This choice is called a **rectilinear basis**. A basis (plural: bases) is just a coordinate system. The second set of filters is the same, except rotated 45 degrees, so one filter runs from the lower left to the upper right and the other filter runs from the upper left to the lower right. This choice is called a **diagonal basis**. Thus, Alice has two bases, which she can rapidly insert into her beam at will. In reality, Alice does not have four separate filters, but a crystal whose polarization can be switched electrically to any of the four allowed directions at great speed. Bob has the same equipment as Alice. The fact that Alice and Bob each have two bases available is essential to quantum cryptography.

For each basis, Alice now assigns one direction as 0 and the other as 1. In the example presented below, we assume she chooses vertical to be 0 and horizontal to be 1. Independently, she also chooses lower left to upper right as 0 and upper left to lower right as 1. She sends these choices to Bob as plaintext, fully aware that Trudy will be able to read her message.

Now Alice picks a one-time pad, for example, based on a random number generator (a complex subject all by itself). She transfers it bit by bit to Bob, choosing one of her two bases at random for each bit. To send a bit, her photon gun emits one photon polarized appropriately for the basis she is using for that bit. For example, she might choose bases of diagonal, rectilinear, rectilinear, diagonal, rectilinear, etc. To send her one-time pad of 1001110010100110 with these bases, she would send the photons shown in Fig. 8-12(a). Given the one-time pad and the sequence of bases, the polarization to use for each bit is uniquely determined. Bits sent one photon at a time are called **qubits**.

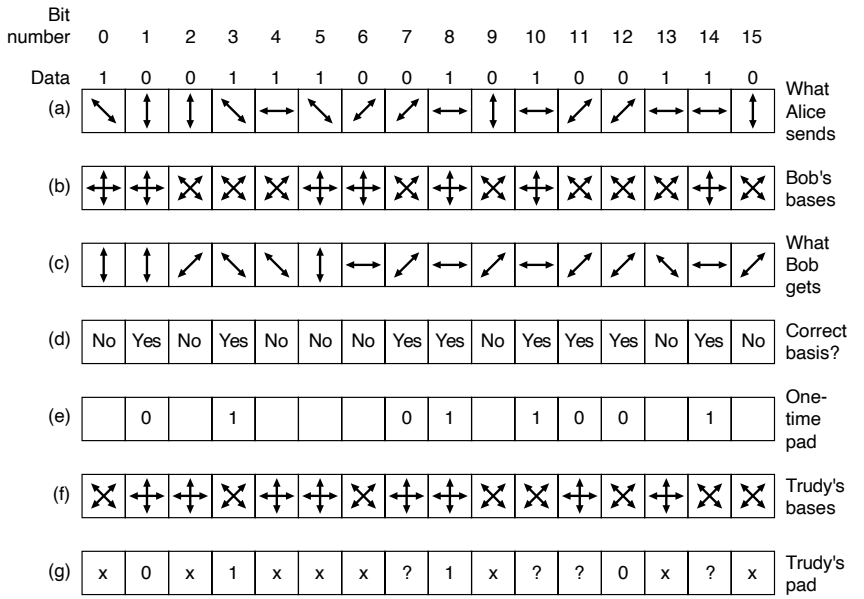

Figure 8-12. An example of quantum cryptography.

Bob does not know which bases to use, so he picks one at random for each arriving photon and just uses it, as shown in Fig. 8-12(b). If he picks the correct basis, he gets the correct bit. If he picks the incorrect basis, he gets a random bit because if a photon hits a filter polarized at 45 degrees to its own polarization, it randomly jumps to the polarization of the filter or to a polarization perpendicular to the filter, with equal probability. This property of photons is fundamental to quantum mechanics. Thus, some of the bits are correct and some are random, but Bob does not know which are which. Bob's results are depicted in Fig. 8-12(c).

How does Bob find out which bases he got right and which he got wrong? He simply tells Alice (in plaintext) which basis he used for each bit in plaintext and she tells him which are right and which are wrong in plaintext, as shown in Fig. 8-12(d). From this information, both of them can build a bit string from the correct guesses, as shown in Fig. 8-12(e). On the average, this bit string will be half the length of the original bit string, but since both parties know it, they can use it as a one-time pad. All Alice has to do is transmit a bit string slightly more than twice the desired length, and she and Bob will have a one-time pad of the desired length. Done.

But wait a minute. We forgot Trudy for the moment. Suppose that she is curious about what Alice has to say and cuts the fiber, inserting her own detector and

transmitter. Unfortunately for her, she does not know which basis to use for each photon either. The best she can do is pick one at random for each photon, just as Bob does. An example of her choices is shown in Fig. 8-12(f). When Bob later reports (in plaintext) which bases he used and Alice tells him (in plaintext) which ones are correct, Trudy now knows when she got it right and when she got it wrong. In Fig. 8-12, she got it right for bits 0, 1, 2, 3, 4, 6, 8, 12, and 13. But she knows from Alice's reply in Fig. 8-12(d) that only bits 1, 3, 7, 8, 10, 11, 12, and 14 are part of the one-time pad. For four of these bits (1, 3, 8, and 12), she guessed right and captured the correct bit. For the other four (7, 10, 11, and 14), she guessed wrong and does not know the bit transmitted. Thus, Bob knows the one-time pad starts with 01011001, from Fig. 8-12(e) but all Trudy has is 01?1??0?, from Fig. 8-12(g).

Of course, Alice and Bob are aware that Trudy may have captured part of their one-time pad, so they would like to reduce the information Trudy has. They can do this by performing a transformation on it. For example, they could divide the one-time pad into blocks of 1024 bits, square each one to form a 2048-bit number, and use the concatenation of these 2048-bit numbers as the one-time pad. With her partial knowledge of the bit string transmitted, Trudy has no way to generate its square and so has nothing. The transformation from the original one-time pad to a different one that reduces Trudy's knowledge is called **privacy amplification**. In practice, complex transformations in which every output bit depends on every input bit are used instead of squaring.

Poor Trudy. Not only does she have no idea what the one-time pad is, but her presence is not a secret either. After all, she must relay each received bit to Bob to trick him into thinking he is talking to Alice. The trouble is, the best she can do is transmit the qubit she received, using the polarization she used to receive it, and about half the time she will be wrong, causing many errors in Bob's one-time pad.

When Alice finally starts sending data, she encodes it using a heavy forward-error-correcting code. From Bob's point of view, a 1-bit error in the one-time pad is the same as a 1-bit transmission error. Either way, he gets the wrong bit. If there is enough forward error correction, he can recover the original message despite all the errors, but he can easily count how many errors were corrected. If this number is far more than the expected error rate of the equipment, he knows that Trudy has tapped the line and can act accordingly (e.g., tell Alice to switch to a radio channel, call the police, etc.). If Trudy had a way to clone a photon so she had one photon to inspect and an identical photon to send to Bob, she could avoid detection, but at present no way to clone a photon perfectly is known. And even if Trudy could clone photons, the value of quantum cryptography to establish one-time pads would not be reduced.

Although quantum cryptography has been shown to operate over distances of 60 km of fiber, the equipment is complex and expensive. Still, the idea has promise if it can be made to scale up and become cheaper. For more information about quantum cryptography, see Clancy et al. (2019).

8.5 SYMMETRIC-KEY ALGORITHMS

Modern cryptography uses the same basic ideas as traditional cryptography (transposition and substitution), but its emphasis is different. Traditionally, cryptographers have used simple algorithms. Nowadays, the reverse is true: the object is to make the encryption algorithm so complex and involuted that even if the cryptanalyst acquires vast mounds of enciphered text of his own choosing, he will not be able to make any sense of it at all without the key.

The first class of encryption algorithms we will study in this chapter are called **symmetric-key algorithms** because they use the same key for encryption and decryption. Fig. 8-9 illustrates the use of a symmetric-key algorithm. In particular, we will focus on **block ciphers**, which take an n-bit block of plaintext as input and transform it using the key into an n-bit block of ciphertext.

Cryptographic algorithms can be implemented in either hardware (for speed) or software (for flexibility). Although most of our treatment concerns the algorithms and protocols, which are independent of the actual implementation, a few words about building cryptographic hardware may be of interest. Transpositions and substitutions can be implemented with simple electrical circuits. Figure 8-13(a) shows a device, known as a **P-box** (P stands for permutation), used to effect a transposition on an 8-bit input. If the 8 bits are designated from top to bottom as 01234567, the output of this particular P-box is 36071245. By appropriate internal wiring, a P-box can be made to perform any transposition and do it at practically the speed of light since no computation is involved, just signal propagation. This design follows Kerckhoffs' principle: the attacker knows that the general method is permuting the bits. What he does not know is which bit goes where.

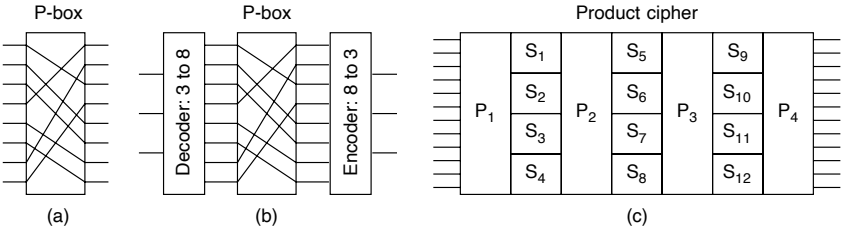

Figure 8-13. Basic elements of product ciphers. (a) P-box. (b) S-box. (c) Product.

Substitutions are performed by **S-boxes**, as shown in Fig. 8-13(b). In this example, a 3-bit plaintext is entered and a 3-bit ciphertext is output. The 3-bit input selects one of the eight lines exiting from the first stage and sets it to 1; all the other lines are 0. The second stage is a P-box. The third stage encodes the selected input line in binary again. With the wiring shown, if the eight octal numbers 01234567 were input one after another, the output sequence would be 24506713.

In other words, 0 has been replaced by 2, 1 has been replaced by 4, etc. Again, by appropriate wiring of the P-box inside the S-box, any substitution can be accomplished. Furthermore, such a device can be built in hardware to achieve great speed, since encoders and decoders have only one or two (subnanosecond) gate delays and the propagation time across the P-box may well be less than 1 picosec.

The real power of these basic elements only becomes apparent when we cascade a whole series of boxes to form a **product cipher**, as shown in Fig. 8-13(c). In this example, 12 input lines are transposed (i.e., permuted) by the first stage (P_1). In the second stage, the input is broken up into four groups of 3 bits, each of which is substituted independently of the others (S_1 to S_4). This arrangement shows a method of approximating a larger S-box from multiple, smaller S-boxes. It is useful because small S-boxes are practical for a hardware implementation (e.g., an 8-bit S-box can be realized as a 256-entry lookup table), but large S-boxes become quite unwieldy to build (e.g., a 12-bit S-box would at a minimum need $2^{12} = 4096$ crossed wires in its middle stage). Although this method is less general, it is still powerful. By including a sufficiently large number of stages in the product cipher, the output can be made to be an exceedingly complicated function of the input.

Product ciphers that operate on k-bit inputs to produce k-bit outputs are common. One common value for k is 256. A hardware implementation usually has at least 20 physical stages, instead of just 7 as in Fig. 8-13(c). A software implementation has a loop with at least eight iterations, each one performing S-box-type substitutions on subblocks of the 64- to 256-bit data block, followed by a permutation that mixes the outputs of the S-boxes. Often there is a special initial permutation and one at the end as well. In the literature, the iterations are called **rounds**.

8.5.1 The Data Encryption Standard

In January 1977 the U.S. Government adopted a product cipher developed by IBM as its official standard for unclassified information. This cipher, **DES** (**Data Encryption Standard**), was widely adopted by the industry for use in security products. It is no longer secure in its original form, but in a modified form it is still used here and there. The original version was controversial because IBM specified a 128-bit key but after discussions with NSA, IBM ''voluntarily'' decided to reduce the key length to 56 bits, which cryptographers at the time said was too small.

DES operates essentially as shown in Fig. 8-13(c), but on bigger units. The plaintext (in binary) is broken up into 64-bit units, and each one is encrypted separately by doing permutations and substitutions parametrized by the 56-bit key on each of 16 consecutive rounds. In effect, it is a gigantic monoalphabetic substitution cipher on an alphabet with 64-bit characters (about which more shortly).

As early as 1979, IBM realized that 56 bits was much too short and devised a backward compatible scheme to increase the key length by having two 56-bit keys

used at once, for a total of 112 bits worth of key (Tuchman, 1979). The new scheme, called **Triple DES** is still in use and works like this.

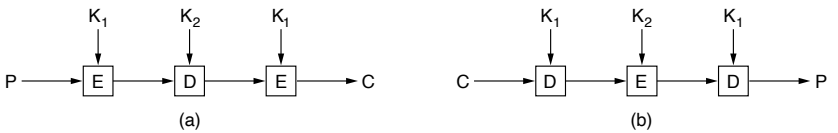

Figure 8-14. (a) Triple encryption using DES. (b) Decryption.

Obvious questions are: (1) Why two keys instead of three? and (2) Why encryption-decryption-encryption? The answer to both is that if a computer that uses triple DES has to talk to one that uses only single DES, it can set both keys to the same value and then apply triple DES to give the same result as single DES. This design made it easier to phase in triple DES. It is basically obsolete now, but still in use in some change-resistant applications.

8.5.2 The Advanced Encryption Standard

As DES began approaching the end of its useful life, even with triple DES, **NIST** (**National Institute of Standards and Technology**), the agency of the U.S. Dept. of Commerce charged with approving standards for the U.S. Federal Government, decided that the government needed a new cryptographic standard for unclassified use. NIST was keenly aware of all the controversy surrounding DES and well knew that if it just announced a new standard, everyone knowing anything about cryptography would automatically assume that NSA had built a back door into it so NSA could read everything encrypted with it. Under these conditions, probably no one would use the standard and it would have died quietly.

So, NIST took a surprisingly different approach for a government bureaucracy: it sponsored a cryptographic bake-off (contest). In January 1997, researchers from all over the world were invited to submit proposals for a new standard, to be called **AES** (**Advanced Encryption Standard**). The bake-off rules were:

1. The algorithm must be a symmetric block cipher.
2. The full design must be public.
3. Key lengths of 128, 192, and 256 bits must be supported.
4. Both software and hardware implementations must be possible.
5. The algorithm must be public or licensed on nondiscriminatory terms.

Fifteen serious proposals were made, and public conferences were organized in which they were presented and attendees were actively encouraged to find flaws in

all of them. In August 1998, NIST selected five finalists, primarily on the basis of their security, efficiency, simplicity, flexibility, and memory requirements (important for embedded systems). More conferences were held and more potshots taken at the contestants.

In October 2000, NIST announced that it had selected Rijndael, invented by Joan Daemen and Vincent Rijmen. The name Rijndael, pronounced Rhine-doll (more or less), is derived from the last names of the authors: Rijmen + Daemen. In November 2001, Rijndael became the AES U.S. Government standard, published as FIPS (Federal Information Processing Standard) 197. Owing to the extraordinary openness of the competition, the technical properties of Rijndael, and the fact that the winning team consisted of two young Belgian cryptographers (who were unlikely to have built in a back door just to please NSA), Rijndael has become the world's dominant cryptographic cipher. AES encryption and decryption is now part of the instruction set for some CPUs.

Rijndael supports key lengths and block sizes from 128 bits to 256 bits in steps of 32 bits. The key length and block length may be chosen independently. However, AES specifies that the block size must be 128 bits and the key length must be 128, 192, or 256 bits. It is doubtful that anyone will ever use 192-bit keys, so de facto, AES has two variants: a 128-bit block with a 128-bit key and a 128-bit block with a 256-bit key.

In our treatment of the algorithm, we will examine only the 128/128 case because this is the commercial norm. A 128-bit key gives a key space of $2^{128} \approx 3 \times 10^{38}$ keys. Even if NSA manages to build a machine with 1 billion parallel processors, each being able to evaluate one key per picosecond, it would take such a machine about 10^{10} years to search the key space. By then the sun will have burned out, so the folks then present will have to read the results by candlelight.

Rijndael

From a mathematical perspective, Rijndael is based on Galois field theory, which gives it some provable security properties. However, it can also be viewed as C code, without getting into the mathematics.

Like DES, Rijndael uses both substitution and permutations, and it also uses multiple rounds. The number of rounds depends on the key size and block size, being 10 for 128-bit keys with 128-bit blocks and moving up to 14 for the largest key or the largest block. However, unlike DES, all operations involve an integral number of bytes, to allow for efficient implementations in both hardware and software. DES is bit oriented and software implementations are slow as a result.

The algorithm has been designed not only for great security, but also for great speed. A good software implementation on a 2-GHz machine should be able to achieve an encryption rate of 700 Mbps, which is fast enough to encrypt over a dozen 4K videos in real time. Hardware implementations are faster still.

8.5.3 Cipher Modes

Despite all this complexity, AES (or DES, or any block cipher for that matter) is basically a monoalphabetic substitution cipher using big characters (128-bit characters for AES and 64-bit characters for DES). Whenever the same plaintext block goes in the front end, the same ciphertext block comes out the back end. If you encrypt the plaintext *abcdefgh* 100 times with the same DES or AES key, you get the same ciphertext 100 times. An intruder can exploit this property to help subvert the cipher.

Electronic Code Book Mode

To see how this monoalphabetic substitution cipher property can be used to partially defeat the cipher, we will use (triple) DES because it is easier to depict 64-bit blocks than 128-bit blocks, but AES has exactly the same problem. The straightforward way to use DES to encrypt a long piece of plaintext is to break it up into consecutive 8-byte (64-bit) blocks and encrypt them one after another with the same key. The last piece of plaintext is padded out to 64 bits, if need be. This technique is known as **ECB mode** (**Electronic Code Book mode**) in analogy with old-fashioned code books where each plaintext word was listed, followed by its ciphertext (usually a five-digit decimal number).

In Fig. 8-15, we have the start of a computer file listing the annual bonuses a company has decided to award to its employees. This file consists of consecutive 32-byte records, one per employee, in the format shown: 16 bytes for the name, 8 bytes for the position, and 8 bytes for the bonus. Each of the sixteen 8-byte blocks (numbered from 0 to 15) is encrypted by (triple) DES.

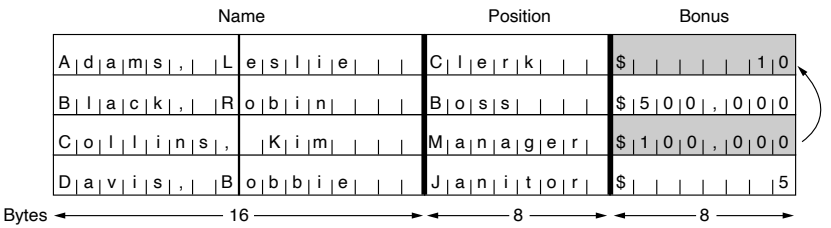

Figure 8-15. The plaintext of a file encrypted as 16 DES blocks.

Leslie just had a fight with the boss and is not expecting much of a bonus. Kim, in contrast, is the boss' favorite, and everyone knows this. Leslie can get access to the file after it is encrypted but before it is sent to the bank. Can Leslie rectify this unfair situation, given only the encrypted file?

No problem at all. All Leslie has to do is make a copy of the 12th ciphertext block (which contains Kim's bonus) and use it to replace the fourth ciphertext

block (which contains Leslie's bonus). Even without knowing what the 12th block says, Leslie can expect to have a much merrier Christmas this year. (Copying the eighth ciphertext block is also a possibility, but is more likely to be detected; besides, Leslie is not a greedy person.)

Cipher Block Chaining Mode

To thwart this type of attack, all block ciphers can be chained in various ways so that replacing a block the way Leslie did will cause the plaintext decrypted starting at the replaced block to be garbage. One method to do so is **cipher block chaining**. In this method, shown in Fig. 8-16, each plaintext block is XORed with the previous ciphertext block before being encrypted. Consequently, the same plaintext block no longer maps onto the same ciphertext block, and the encryption is no longer a big monoalphabetic substitution cipher. The first block is XORed with a randomly chosen **IV** (**Initialization Vector**), which is transmitted (in plaintext) along with the ciphertext.

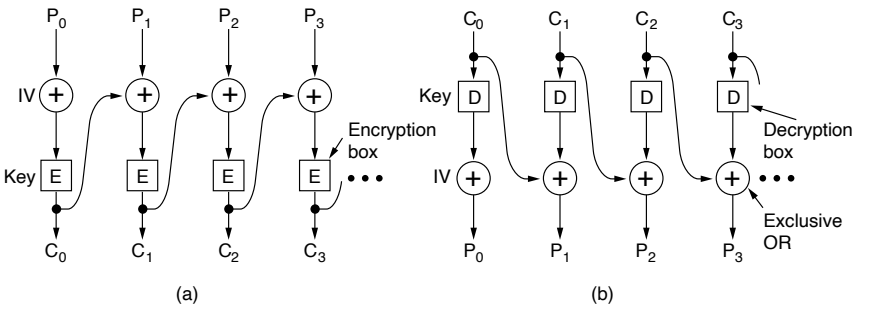

Figure 8-16. Cipher block chaining. (a) Encryption. (b) Decryption.

We can see how cipher block chaining mode works by examining the example of Fig. 8-16. We start out by computing $C_0 = E(P_0 \text{ XOR } IV)$. Then we compute $C_1 = E(P_1 \text{ XOR } C_0)$, and so on. Decryption also uses XOR to reverse the process, with $P_0 = IV \text{ XOR } D(C_0)$, and so on. Note that the encryption of block i is a function of all the plaintext in blocks 0 through $i - 1$, so the same plaintext generates different ciphertext depending on where it occurs. A transformation of the type Leslie made will result in nonsense for two blocks starting at Leslie's bonus field. To an astute security officer, this peculiarity might suggest where to start the ensuing investigation.

Cipher block chaining also has the advantage that the same plaintext block will not result in the same ciphertext block, making cryptanalysis more difficult. In fact, this is the main reason it is used.

Cipher Feedback Mode

However, cipher block chaining has the disadvantage of requiring an entire 64-bit block to arrive before decryption can begin. For byte-by-byte encryption, **cipher feedback mode** using (triple) DES is used, as shown in Fig. 8-17. For AES, the idea is exactly the same, only a 128-bit shift register is used. In this figure, the state of the encryption machine is shown after bytes 0 through 9 have been encrypted and sent. When plaintext byte 10 arrives, as illustrated in Fig. 8-17(a), the DES algorithm operates on the 64-bit shift register to generate a 64-bit ciphertext. The leftmost byte of that ciphertext is extracted and XORed with P_{10}. That byte is transmitted on the transmission line. In addition, the shift register is shifted left 8 bits, causing C_2 to fall off the left end, and C_{10} is inserted in the position just vacated at the right end by C_9.

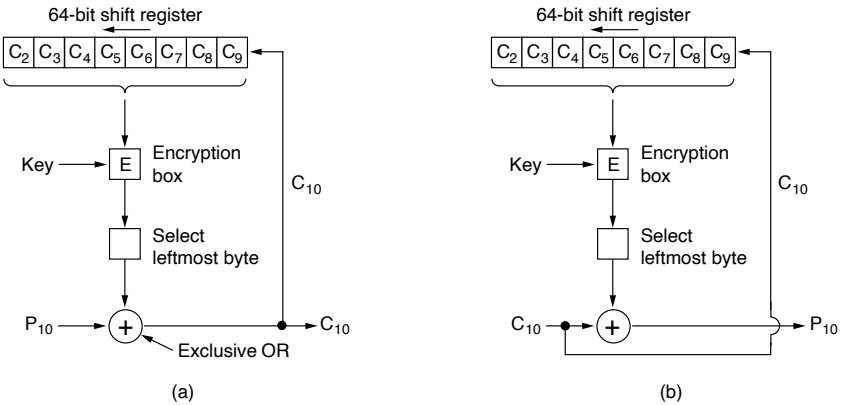

Figure 8-17. Cipher feedback mode. (a) Encryption. (b) Decryption.

Note that the contents of the shift register depend on the entire previous history of the plaintext, so a pattern that repeats multiple times in the plaintext will be encrypted differently each time in the ciphertext. As with cipher block chaining, an initialization vector is needed to start the ball rolling.

Decryption with cipher feedback mode works the same way as encryption. In particular, the content of the shift register is *encrypted*, not *decrypted*, so the selected byte that is XORed with C_{10} to get P_{10} is the same one that was XORed with P_{10} to generate C_{10} in the first place. As long as the two shift registers remain identical, decryption works correctly. This is illustrated in Fig. 8-17(b).

A problem with cipher feedback mode is that if one bit of the ciphertext is accidentally inverted during transmission, the 8 bytes that are decrypted while the bad byte is in the shift register will be corrupted. Once the bad byte is pushed out of the shift register, correct plaintext will once again be generated thereafter. Thus,

the effects of a single inverted bit are relatively localized and do not ruin the rest of the message, but they do ruin as many bits as the shift register is wide.

Stream Cipher Mode

Nevertheless, applications exist in which having a 1-bit transmission error mess up 64 bits of plaintext is too large an effect. For these applications, a fourth option, **stream cipher mode**, exists. It works by encrypting an initialization vector (IV), using a key to get an output block. The output block is then encrypted, using the key to get a second output block. This block is then encrypted to get a third block, and so on. The (arbitrarily large) sequence of output blocks, called the **keystream**, is treated like a one-time pad and XORed with the plaintext to get the ciphertext, as shown in Fig. 8-18(a). Note that the IV is used only on the first step. After that, the output is encrypted. Also, note that the keystream is independent of the data, so it can be computed in advance, if need be, and is completely insensitive to transmission errors. Decryption is shown in Fig. 8-18(b).

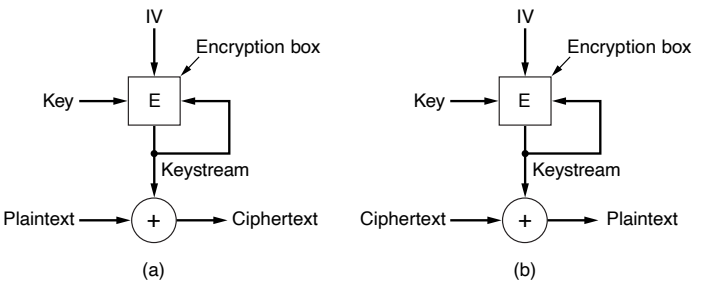

Figure 8-18. A stream cipher. (a) Encryption. (b) Decryption.

Decryption occurs by generating the same keystream at the receiving side. Since the keystream depends only on the IV and the key, it is not affected by transmission errors in the ciphertext. Thus, a 1-bit error in the transmitted ciphertext generates only a 1-bit error in the decrypted plaintext.

It is essential never to use the same (key, IV) pair twice with a stream cipher because doing so will generate the same keystream each time. Using the same keystream twice exposes the ciphertext to a **keystream reuse attack**. Imagine that the plaintext block, P_0, is encrypted with the keystream to get P_0 XOR K_0. Later, a second plaintext block, Q_0, is encrypted with the same keystream to get Q_0 XOR K_0. An intruder who captures both of these ciphertext blocks can simply XOR them together to get P_0 XOR Q_0, which eliminates the key. The intruder now has the XOR of the two plaintext blocks. If one of them is known or can be reasonably guessed, the other can also be found. In any event, the XOR of two plaintext streams can be attacked by using statistical properties of the message.

For example, for English text, the most common character in the stream will probably be the XOR of two spaces, followed by the XOR of space and the letter "e" and so on. In short, equipped with the XOR of two plaintexts, the cryptanalyst has an excellent chance of deducing both of them.

8.6 PUBLIC-KEY ALGORITHMS

Historically, distributing the keys has always been the weakest link in most cryptosystems. No matter how strong a cryptosystem was, if an intruder could steal the key, the system was worthless. Cryptologists always took for granted that the encryption key and decryption key were the same (or easily derived from one another). But the key had to be distributed to all users of the system. Thus, it seemed as if there was an inherent problem. Keys had to be protected from theft, but they also had to be distributed, so they could not be locked in a bank vault.

In 1976, two researchers at Stanford University, Diffie and Hellman (1976), proposed a radically new kind of cryptosystem, one in which the encryption and decryption keys were so different that the decryption key could not feasibly be derived from the encryption key. In their proposal, the (keyed) encryption algorithm, E, and the (keyed) decryption algorithm, D, had to meet three requirements. These requirements can be stated simply as follows:

1. $D(E(P)) = P$.
2. It is exceedingly difficult to deduce D from E.
3. E cannot be broken by a chosen plaintext attack.

The first requirement says that if we apply D to an encrypted message, $E(P)$, we get the original plaintext message, P, back. Without this property, the legitimate receiver could not decrypt the ciphertext. The second requirement speaks for itself. The third requirement is needed because, as we shall see in a moment, intruders may experiment with the algorithm to their hearts' content. Under these conditions, there is no reason that the encryption key cannot be made public.

The method works like this. A person, say, Alice, who wants to receive secret messages, first devises two algorithms meeting the above requirements. The encryption algorithm and Alice's key are then made public, hence the name **public-key cryptography**. Alice might put her public key on her home page on the Web, for example. We will use the notation E_A to mean the encryption algorithm parametrized by Alice's public key. Similarly, the (secret) decryption algorithm parameterized by Alice's private key is D_A. Bob does the same thing, publicizing E_B but keeping D_B secret.

Now let us see if we can solve the problem of establishing a secure channel between Alice and Bob, who have never had any previous contact. Both Alice's encryption key, E_A, and Bob's encryption key, E_B, are assumed to be in publicly

readable files. Now Alice takes her first message, P, computes $E_B(P)$, and sends it to Bob. Bob then decrypts it by applying his secret key D_B [i.e., he computes $D_B(E_B(P)) = P$]. No one else can read the encrypted message, $E_B(P)$, because the encryption system is assumed to be strong and because it is too difficult to derive D_B from the publicly known E_B. To send a reply, R, Bob transmits $E_A(R)$. Alice and Bob can now communicate securely.

A note on terminology is perhaps useful here. Public-key cryptography requires each user to have two keys: a public key, used by the entire world for encrypting messages to be sent to that user, and a private key, which the user needs for decrypting messages. We will consistently refer to these keys as the *public* and *private* keys, respectively, and distinguish them from the *secret* keys used for conventional symmetric-key cryptography.

8.6.1 RSA

The only catch is that we need to find algorithms that indeed satisfy all three requirements. Due to the potential advantages of public-key cryptography, many researchers are hard at work, and some algorithms have already been published. One good method was discovered by a group at M.I.T. (Rivest et al., 1978). It is known by the initials of the three discoverers (Rivest, Shamir, Adleman): **RSA**. It has survived all attempts to break it for more than 40 years and is considered very strong. Much practical security is based on it. For this reason, Rivest, Shamir, and Adleman were given the 2002 ACM Turing Award. Its major disadvantage is that it requires keys of at least 2048 bits for good security (versus 256 bits for symmetric-key algorithms), which makes it quite slow.

The RSA method is based on some principles from number theory. We will now summarize how to use the method; for details, consult their paper.

1. Choose two large primes, p and q (say, 1024 bits).
2. $n = p \times q$ and $z = (p - 1) \times (q - 1)$.
3. Choose a number relatively prime to z and call it d.
4. Find e such that $e \times d = 1 \ mod \ z$.

With these parameters computed in advance, we are ready to begin encryption. Divide the plaintext (regarded as a bit string) into blocks, so that each plaintext message, P, falls in the interval $0 \le P < n$. Do that by grouping the plaintext into blocks of k bits, where k is the largest integer for which $2^k < n$ is true.

To encrypt a message, P, compute $C = P^e \pmod{n}$. To decrypt C, compute $P = C^d \pmod{n}$. It can be proven that for all P in the specified range, the encryption and decryption functions are inverses. To perform the encryption, you need e and n. To perform the decryption, you need d and n. Therefore, the public key consists of the pair (e, n) and the private key consists of (d, n).

The security of the method is based on the difficulty of factoring large numbers. If the cryptanalyst could factor the (publicly known) n, he could then find p and q, and from these z. Equipped with knowledge of z and e, d can be found using Euclid's algorithm. Fortunately, mathematicians have been trying to factor large numbers for at least 300 years, and the accumulated evidence suggests that it is an exceedingly difficult problem.

At the time, Rivest and colleagues concluded that factoring a 500-digit number would require 10^{25} years using brute force. In both cases, they assumed the best-known algorithm and a computer with a 1-μsec instruction time. With a million chips running in parallel, each with an instruction time of 1 nsec, it would still take 10^{16} years. Even if computers continue to get faster by an order of magnitude per decade, it will be many years before factoring a 500-digit number becomes feasible, at which time our descendants can simply choose p and q still larger. However, it will probably not come as a surprise that the attacks have made progress and are now significantly faster.

A trivial pedagogical example of how the RSA algorithm works is given in Fig. 8-19. For this example, we have chosen $p = 3$ and $q = 11$, giving $n = 33$ and $z = 20$ (since$(3 - 1) \times (11 - 1) = 20$). A suitable value for d is $d = 7$, since 7 and 20 have no common factors. With these choices, e can be found by solving the equation $7e = 1 \pmod{20}$, which yields $e = 3$. The ciphertext, C, corresponding to a plaintext message, P, is given by $C = P^3 \pmod{33}$. The ciphertext is decrypted by the receiver by making use of the rule $P = C^7 \pmod{33}$. The figure shows the encryption of the plaintext ''SUZANNE'' as an example.

Plaintext (P)			Ciphertext (C)		After decryption	
Symbolic	Numeric	P^3	$P^3 \pmod{33}$	C^7	$C^7 \pmod{33}$	Symbolic
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	01	1	1	1	01	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	05	125	26	8031810176	05	E
		Sender's computation		Receiver's computation		

Figure 8-19. An example of the RSA algorithm.

Because the primes chosen for this example are so small, P must be less than 33, so each plaintext block can contain only a single character. The result is a monoalphabetic substitution cipher, not very impressive. If instead we had chosen p and $q \approx 2^{512}$, we would have $n \approx 2^{1024}$, so each block could be up to 1024 bits or 128 eight-bit characters, versus 8 characters for DES and 16 characters for AES.

It should be pointed out that using RSA as we have described is similar to using a symmetric algorithm in ECB mode—the same input block gives the same output block. Therefore, some form of chaining is needed for data encryption. However, in practice, most RSA-based systems use public-key cryptography primarily for distributing one-time 128- or 256-bit session keys for use with some symmetric-key algorithm such as AES. RSA is too slow for actually encrypting large volumes of data but is widely used for key distribution.

8.6.2 Other Public-Key Algorithms

Although RSA is still widely used, it is by no means the only public-key algorithm known. The first public-key algorithm was the knapsack algorithm (Merkle and Hellman, 1978). The idea here is that someone owns a very large number of objects, each with a different weight. The owner encodes the message by secretly selecting a subset of the objects and placing them in the knapsack. The total weight of the objects in the knapsack is made public, as is the list of all possible objects and their corresponding weights. The list of objects in the knapsack is kept secret. With certain additional restrictions, the problem of figuring out a possible list of objects with the given weight was thought to be computationally infeasible and formed the basis of the public-key algorithm.

The algorithm's inventor, Ralph Merkle, was quite sure that this algorithm could not be broken, so he offered a $100 reward to anyone who could break it. Adi Shamir (the "S" in RSA) promptly broke it and collected the reward. Undeterred, Merkle strengthened the algorithm and offered a $1000 reward to anyone who could break the new one. Ronald Rivest (the "R" in RSA) promptly broke the new one and collected the reward. Merkle did not dare offer $10,000 for the next version, so "A" (Leonard Adleman) was out of luck. Nevertheless, the knapsack algorithm is not considered secure and is not used in practice any more.

Other public-key schemes are based on the difficulty of computing discrete logarithms or on elliptic curves (Menezes and Vanstone, 1993). Algorithms that use discrete algorithms have been invented by El Gamal (1985) and Schnorr (1991). Elliptic curves, meanwhile are based on a branch of mathematics that is not so well-known except among the elliptic curve *illuminati*.

A few other schemes exist, but those based on the difficulty of factoring large numbers, computing discrete logarithms modulo a large prime, and elliptic curves, are by far the most important. These problems are thought to be genuinely difficult to solve—mathematicians have been working on them for many years without any great breakthroughs. Elliptic curves in particular enjoy a lot of interest because the elliptic curve discrete algorithm problems are even harder than those of factorization. The Dutch mathematician Arjen Lenstra proposed a way to compare cryptographic algorithms by computing how much energy you need to break them. According to this calculation, breaking a 228-bit RSA key takes the energy equivalent to that needed to boil less than a teaspoon of water. Breaking an elliptic curve

of that length would require as much energy as you would need to boil all the water on the planet. Paraphrasing Lenstra: with all water evaporated, including that in the bodies of would-be code breakers, the problem would run out of steam.

8.7 DIGITAL SIGNATURES

The authenticity of many legal, financial, and other documents is determined by the presence or absence of an authorized handwritten signature. And photocopies do not count. For computerized message systems to replace the physical transport of paper-and-ink documents, a method must be found to allow documents to be signed in an unforgeable way.

The problem of devising a replacement for handwritten signatures is a difficult one. Basically, what is needed is a system by which one party can send a signed message to another party in such a way that the following conditions hold:

1. The receiver can verify the claimed identity of the sender.
2. The sender cannot later repudiate the contents of the message.
3. The receiver cannot possibly have concocted the message himself.

The first requirement is needed, for example, in financial systems. When a customer's computer orders a bank's computer to buy a ton of gold, the bank's computer needs to be able to make sure that the computer giving the order really belongs to the customer whose account is to be debited. In other words, the bank has to authenticate the customer (and the customer has to authenticate the bank).

The second requirement is needed to protect the bank against fraud. Suppose that the bank buys the ton of gold, and immediately thereafter the price of gold drops sharply. A dishonest customer might then proceed to sue the bank, claiming that he never issued any order to buy gold. When the bank produces the message in court, the customer may deny having sent it. The property that no party to a contract can later deny having signed it is called **nonrepudiation**. The digital signature schemes that we will now study help provide it.

The third requirement is needed to protect the customer in the event that the price of gold shoots up and the bank tries to construct a signed message in which the customer asked for one bar of gold instead of one ton. In this fraud scenario, the bank just keeps the rest of the gold for itself.

8.7.1 Symmetric-Key Signatures

One approach to digital signatures is to have a central authority that knows everything and whom everyone trusts, say, Big Brother (*BB*). Each user then chooses a secret key and carries it by hand to *BB*'s office. Thus, only Alice and *BB*

know Alice's secret key, K_A, and so on. In case you get lost with all notations, with symbols and subscripts, have a look at Fig. 8-20 which summarizes the most important notations for this and subsequent sections.

Term	Description
A	Alice (sender)
B	Bob the Banker (recipient)
P	Plaintext message Alice wants to send
BB	Big Brother (a trusted central authority)
t	Timestamp (to ensure freshness)
R_A	Random number chosen by Alice
Symmetric key	
K_A	Alice's secret key (analogous for K_B, K_{BB}, etc.)
$K_A(M)$	Message M encrypted/decrypted with Alice's secret key
Asymmetric keys	
D_A	Alice's private key (analogous for D_B, etc.)
E_A	Alice's public key (analogous for E_B, etc.)
$D_A(M)$	Message M encrypted/decrypted with Alice's private key
$E_A(M)$	Message M encrypted/decrypted with Alice's public key
Digest	
$MD(P)$	Message Digest of plaintext P)

Figure 8-20. Alice wants to send a message to her banker: a legend to keys and symbols

When Alice wants to send a signed plaintext message, P, to her banker, Bob, she generates $K_A(B, R_A, t, P)$, where B is Bob's identity, R_A is a random number chosen by Alice, t is a timestamp to ensure freshness, and $K_A(B, R_A, t, P)$ is the message encrypted with her key, K_A. Then she sends it as depicted in Fig. 8-21. *BB* sees that the message is from Alice, decrypts it, and sends a message to Bob as shown. The message to Bob contains the plaintext of Alice's message and also the signed message $K_{BB}(A, t, P)$. Bob now carries out Alice's request.

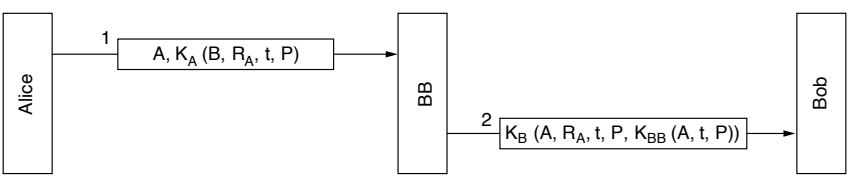

Figure 8-21. Digital signatures with Big Brother.

What happens if Alice later denies sending the message? Step 1 is that everyone sues everyone (at least, in the United States). Finally, when the case comes to court and Alice vigorously denies sending Bob the disputed message, the judge

will ask Bob how he can be sure that the disputed message came from Alice and not from Trudy. Bob first points out that *BB* will not accept a message from Alice unless it is encrypted with K_A, so there is no possibility of Trudy sending *BB* a false message from Alice without *BB* detecting it immediately.

Bob then dramatically produces Exhibit A: $K_{BB}(A, t, P)$. Bob says that this is a message signed by *BB* that proves Alice sent P to Bob. The judge then asks *BB* (whom everyone trusts) to decrypt Exhibit A. When *BB* testifies that Bob is telling the truth, the judge decides in favor of Bob. Case dismissed.

One potential problem with the signature protocol of Fig. 8-21 is Trudy replaying either message. To minimize this problem, timestamps are used throughout. Furthermore, Bob can check all recent messages to see if R_A was used in any of them. If so, the message is discarded as a replay. Note that based on the timestamp, Bob will reject very old messages. To guard against instant replay attacks, Bob just checks the R_A of every incoming message to see if such a message has been received from Alice in the past hour. If not, Bob can safely assume this is a new request.

8.7.2 Public-Key Signatures

A structural problem with using symmetric-key cryptography for digital signatures is that everyone has to agree to trust Big Brother. Furthermore, Big Brother gets to read all signed messages. The most logical candidates for running the Big Brother server are the government, the banks, the accountants, and the lawyers. Unfortunately, none of these inspire total confidence in all citizens. Hence, it would be nice if signing documents did not require a trusted authority.

Fortunately, public-key cryptography can make an important contribution in this area. Let us assume that the public-key encryption and decryption algorithms have the property that $E(D(P)) = P$, in addition, of course, to the usual property that $D(E(P)) = P$. (RSA has this property, so the assumption is not unreasonable.) Assuming that this is the case, Alice can send a signed plaintext message, P, to Bob by transmitting $E_B(D_A(P))$. Note carefully that Alice knows her own (private) key, D_A, as well as Bob's public key, E_B, so constructing this message is something Alice can do.

When Bob receives the message, he transforms it using his private key, as usual, yielding $D_A(P)$, as shown in Fig. 8-22. He stores this text in a safe place and then applies E_A to get the original plaintext.

To see how the signature property works, suppose that Alice subsequently denies having sent the message P to Bob. When the case comes up in court, Bob can produce both P and $D_A(P)$. The judge can easily verify that Bob indeed has a valid message encrypted by D_A by simply applying E_A to it. Since Bob does not know what Alice's private key is, the only way Bob could have acquired a message encrypted by it is if Alice did indeed send it. While in jail for perjury and fraud, Alice will have much time to devise interesting new public-key algorithms.

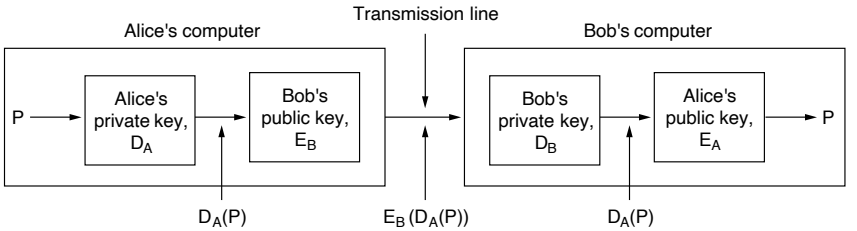

Figure 8-22. Digital signatures using public-key cryptography.

Although using public-key cryptography for digital signatures is an elegant scheme, there are problems that are related to the environment in which they operate rather than to the basic algorithm. For one thing, Bob can prove that a message was sent by Alice only as long as D_A remains secret. If Alice discloses her secret key, the argument no longer holds because anyone could have sent the message, including Bob himself.

The problem might arise, for example, if Bob is Alice's stockbroker. Suppose that Alice tells Bob to buy a certain stock or bond. Immediately thereafter, the price drops sharply. To repudiate her message to Bob, Alice runs to the police claiming that her home was burglarized and the computer holding her key was stolen. Depending on the laws in her state or country, she may or may not be legally liable, especially if she claims not to have discovered the break-in until getting home from work, several hours after it allegedly happened.

Another problem with the signature scheme is what happens if Alice decides to change her key. Doing so is clearly legal, and it is probably a good idea to do so periodically. If a court case later arises, as described above, the judge will apply the *current* E_A to $D_A(P)$ and discover that it does not produce P. Bob will look pretty stupid at this point.

In principle, any public-key algorithm can be used for digital signatures. The de facto industry standard is the RSA algorithm. Many security products use it. However, in 1991, NIST proposed using a variant of the El Gamal public-key algorithm for its new **Digital Signature Standard** (**DSS**). El Gamal gets its security from the difficulty of computing discrete logarithms, rather than from the difficulty of factoring large numbers.

As usual when the government tries to dictate cryptographic standards, there was an uproar. DSS was criticized for being

1. Too secret (NSA designed the protocol for using El Gamal).
2. Too slow (10 to 40 times slower than RSA for checking signatures).
3. Too new (El Gamal had not yet been thoroughly analyzed).
4. Too insecure (fixed 512-bit key).

In a subsequent revision, the fourth point was rendered moot when keys up to 1024 bits were allowed. Nevertheless, the first two points remain valid.

8.7.3 Message Digests

One criticism of signature methods is that they often couple two distinct functions: authentication and secrecy. Often, authentication is needed but secrecy is not always needed. Also, getting an export license is often easier if the system in question provides only authentication but not secrecy. Below we will describe an authentication scheme that does not require encrypting the entire message.

This scheme is based on the idea of a one-way hash function that takes an arbitrarily long piece of plaintext and from it computes a fixed-length bit string. This hash function, MD, often called a **message digest**, has four important properties:

1. Given P, it is easy to compute $MD(P)$.
2. Given $MD(P)$, it is effectively impossible to find P.
3. Given P, no one can find P' such that $MD(P') = MD(P)$.
4. A change to the input of even 1 bit produces a very different output.

To meet criterion 3, the hash should be at least 128 bits long, preferably more. To meet criterion 4, the hash must mangle the bits very thoroughly, not unlike the symmetric-key encryption algorithms we have seen.

Computing a message digest from a piece of plaintext is much faster than encrypting that plaintext with a public-key algorithm, so message digests can be used to speed up digital signature algorithms. To see how this works, consider the signature protocol of Fig. 8-21 again. Instead, of signing P with $K_{BB}(A, t, P)$, BB now computes the message digest by applying MD to P, yielding $MD(P)$. BB then encloses $K_{BB}(A, t, MD(P))$ as the fifth item in the list encrypted with K_B that is sent to Bob, instead of $K_{BB}(A, t, P)$.

If a dispute arises, Bob can produce both P and $K_{BB}(A, t, MD(P))$. After Big Brother has decrypted it for the judge, Bob has $MD(P)$, which is guaranteed to be genuine, and the alleged P. However, since it is effectively impossible for Bob to find any other message that gives this hash, the judge will easily be convinced that Bob is telling the truth. Using message digests in this way saves both encryption time and message transport costs.

Message digests work in public-key cryptosystems, too, as shown in Fig. 8-23. Here, Alice first computes the message digest of her plaintext. She then signs the message digest and sends both the signed digest and the plaintext to Bob. If Trudy replaces P along the way, Bob will see this when he computes $MD(P)$.

SHA-1, SHA-2 and SHA-3

A variety of message digest functions have been proposed. For a long time, one of the most widely used functions was **SHA-1** (**Secure Hash Algorithm 1**) (NIST, 1993). Before we commence our explanation, it is important to realize that

Figure 8-23. Digital signatures using message digests.

SHA-1 has been broken since 2017 and is now being phased out by many systems, but more about this later. Like all message digests, SHA-1 operates by mangling bits in a sufficiently complicated way that every output bit is affected by every input bit. SHA-1 was developed by NSA and blessed by NIST in FIPS 180-1. It processes input data in 512-bit blocks, and it generates a 160-bit message digest. A typical way for Alice to send a nonsecret but signed message to Bob is illustrated in Fig. 8-24. Here, her plaintext message is fed into the SHA-1 algorithm to get a 160-bit SHA-1 hash. Alice then signs the hash with her RSA private key and sends both the plaintext message and the signed hash to Bob.

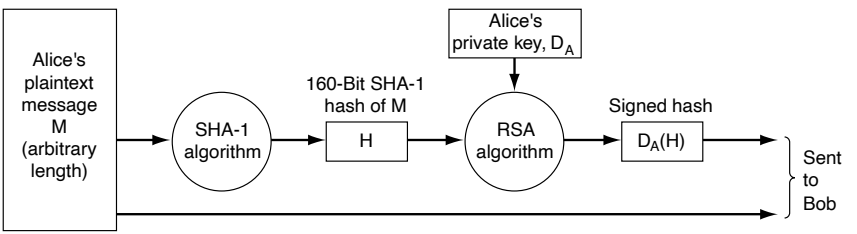

Figure 8-24. Use of SHA-1 and RSA for signing nonsecret messages.

After receiving the message, Bob computes the SHA-1 hash himself and also applies Alice's public key to the signed hash to get the original hash, *H*. If the two agree, the message is considered valid. Since there is no way for Trudy to modify the (plaintext) message while it is in transit and produce a new one that hashes to *H*, Bob can easily detect any changes Trudy has made to the message. For messages whose integrity is important but whose contents are not secret, the scheme of Fig. 8-24 is widely used. For a relatively small cost in computation, it guarantees that any modifications made to the plaintext message in transit can be detected with very high probability.

New versions of SHA-1 have been developed that produce hashes of 224, 256, 384, and 512 bits, respectively. Collectively, these versions are called SHA-2. Not only are these hashes longer than SHA-1 hashes, but the digest function has been changed to combat some potential weaknesses of SHA-1. The weaknesses are serious. In 2017, SHA-1 was broken by a team of researchers from Google and the

CWI research center in Amsterdam. Specifically, the researchers were able to generate **hash collisions**, essentially killing the security of SHA-1. Not surprisingly, the attack led to an increased interest in SHA-2.

In 2006, the National Institute of Standards and Technology (NIST) started organizing a competition for a new hash standard, which is now known as SHA-3. The competition closed in 2012. Three years later, the new SHA-3 standard (''Keccak'') was officially published. Interestingly, NIST does not suggest that we all dump SHA-2 in the trash and switch to SHA-3 because there are no successful attacks on SHA-2 yet. Even so, it is good to have a drop-in replacement lying around, just in case.

8.7.4 The Birthday Attack

In the world of crypto, nothing is ever what it seems to be. One might think that it would take on the order of 2^m operations to subvert an m-bit message digest. In fact, $2^{m/2}$ operations will often do using a **birthday attack**, in an approach published by Yuval (1979) in his now-classic paper ''How to Swindle Rabin.''

Remember, from our earlier discussion of the DNS birthday attack that if there is some mapping between inputs and outputs with n inputs (people, messages, etc.) and k possible outputs (birthdays, message digests, etc.), there are $n(n-1)/2$ input pairs. If $n(n-1)/2 > k$, the chance of having at least one match is pretty good. Thus, approximately, a match is likely for $n > \sqrt{k}$. This result means that a 64-bit message digest can probably be broken by generating about 2^{32} messages and looking for two with the same message digest.

Let us look at a practical example. The Department of Computer Science at State University has one position for a tenured faculty member and two candidates, Tom and Dick. Tom was hired two years before Dick, so he goes up for review first. If he gets it, Dick is out of luck. Tom knows that the department chairperson, Marilyn, thinks highly of his work, so he asks her to write him a letter of recommendation to the Dean, who will decide on Tom's case. Once sent, all letters become confidential.

Marilyn tells her secretary, Ellen, to write the Dean a letter, outlining what she wants in it. When it is ready, Marilyn will review it, compute and sign the 64-bit digest, and send it to the Dean. Ellen can send the letter later by email.

Unfortunately for Tom, Ellen is romantically involved with Dick and would like to do Tom in, so she writes the following letter with the 32 bracketed options:

Dear Dean Smith,

This [*letter* / *message*] is to give my [*honest* / *frank*] opinion of Prof. Tom Wilson, who is [*a candidate* / *up*] for tenure [*now* / *this year*]. I have [*known* / *worked with*] Prof. Wilson for [*about* / *almost*] six years. He is an [*outstanding* / *excellent*] researcher of great [*talent* / *ability*] known [*worldwide* / *internationally*] for his [*brilliant* / *creative*] insights into [*many* / *a wide variety of*] [*difficult* / *challenging*] problems.

He is also a [*highly* / *greatly*] [*respected* / *admired*] [*teacher* / *educator*]. His students give his [*classes* / *courses*] [*rave* / *spectacular*] reviews. He is [*our* / *the Department's*] [*most popular* / *best-loved*] [*teacher* / *instructor*].

[*In addition* / *Additionally*] Prof. Wilson is a [*gifted* / *effective*] fund raiser. His [*grants* / *contracts*] have brought a [*large* / *substantial*] amount of money into [*the* / *our*] Department. [*This money has* / *These funds have*] [*enabled* / *permitted*] us to [*pursue* / *carry out*] many [*special* / *important*] programs, [*such as* / *for example*] your State 2025 program. Without these funds we would [*be unable* / *not be able*] to continue this program, which is so [*important* / *essential*] to both of us. I strongly urge you to grant him tenure.

Unfortunately for Tom, as soon as Ellen finishes composing and typing in this letter, she also writes a second one:

Dear Dean Smith,

This [*letter* | *message*] is to give my [*honest* | *frank*] opinion of Prof. Tom Wilson, who is [*a candidate* | *up*] for tenure [*now* | *this year*]. I have [*known* | *worked with*] Tom for [*about* | *almost*] six years. He is a [*poor* | *weak*] researcher not well known in his [*field* | *area*]. His research [*hardly ever* | *rarely*] shows [*insight in* | *understanding of*] the [*key* | *major*] problems of [*the* | *our*] day.

Furthermore, he is not a [*respected* | *admired*] [*teacher* | *educator*]. His students give his [*classes* | *courses*] [*poor* | *bad*] reviews. He is [*our* | *the Department's*] least popular [*teacher* | *instructor*], known [*mostly* | *primarily*] within [*the* | *our*] Department for his [*tendency* | *propensity*] to [*ridicule* | *embarrass*] students [*foolish* | *imprudent*] enough to ask questions in his classes.

[*In addition* | *Additionally*] Tom is a [*poor* | *marginal*] fund raiser. His [*grants* | *contracts*] have brought only a [*meager* | *insignificant*] amount of money into [*the* | *our*] Department. Unless new [*money is* | *funds are*] quickly located, we may have to cancel some essential programs, such as your State 2025 program. Unfortunately, under these [*conditions* | *circumstances*] I cannot in good [*conscience* | *faith*] recommend him to you for [*tenure* | *a permanent position*].

Now Ellen programs her computer to compute the 2^{32} message digests of each letter overnight. Chances are, one digest of the first letter will match one digest of the second. If not, she can add a few more options and try again tonight. Suppose that she finds a match. Call the ''good'' letter *A* and the ''bad'' one *B*.

Ellen now emails letter *A* to Marilyn for approval. Letter *B* she keeps secret, showing it to no one. Marilyn, of course, approves it, computes her 64-bit message digest, signs the digest, and emails the signed digest off to Dean Smith. Independently, Ellen emails letter *B* to the Dean (not letter *A*, as she is supposed to).

After getting the letter and signed message digest, the Dean runs the message digest algorithm on letter *B*, sees that it agrees with what Marilyn sent him, and fires Tom. The Dean does not realize that Ellen managed to generate two letters with the same message digest and sent her a different one than the one Marilyn saw and approved. (Optional ending: Ellen tells Dick what she did. Dick is appalled

and breaks off the affair. Ellen is furious and confesses to Marilyn. Marilyn calls the Dean. Tom gets tenure after all.) With SHA-2, the birthday attack is difficult because even at the ridiculous speed of 1 trillion digests per second, it would take over 32,000 years to compute all 2^{80} digests of two letters with 80 variants each, and even then a match is not guaranteed. However, with a cloud of 1,000,000 chips working in parallel, 32,000 years becomes 2 weeks.

8.8 MANAGEMENT OF PUBLIC KEYS

Public-key cryptography makes it possible for people who do not share a common key in advance to nevertheless communicate securely. It also makes signing messages possible without the existence of a trusted third party. Finally, signed message digests make it possible for the recipient to verify the integrity of received messages easily and securely.

However, there is one problem that we have glossed over a bit too quickly: if Alice and Bob do not know each other, how do they get each other's public keys to start the communication process? The obvious solution—put your public key on your Web site—does not work, for the following reason. Suppose that Alice wants to look up Bob's public key on his Web site. How does she do it? She starts by typing in Bob's URL. Her browser then looks up the DNS address of Bob's home page and sends it a *GET* request, as shown in Fig. 8-25. Unfortunately, Trudy intercepts the request and replies with a fake home page, probably a copy of Bob's home page except for the replacement of Bob's public key with Trudy's public key. When Alice now encrypts her first message with E_T, Trudy decrypts it, reads it, re-encrypts it with Bob's public key, and sends it to Bob, who is none the wiser that Trudy is reading his incoming messages. Worse yet, Trudy could modify the messages before reencrypting them for Bob. Clearly, some mechanism is needed to make sure that public keys can be exchanged securely.

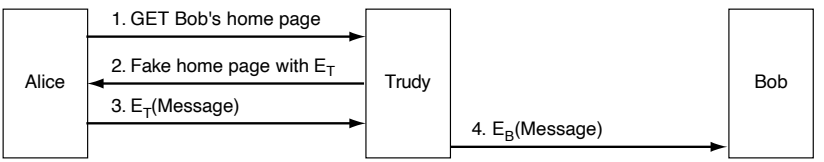

Figure 8-25. A way for Trudy to subvert public-key encryption.

8.8.1 Certificates

As a first attempt at distributing public keys securely, we could imagine a **KDC** (**Key Distribution Center**) available online 24 hours a day to provide public keys on demand. One of the many problems with this solution is that it is not

scalable, and the key distribution center would rapidly become a bottleneck. Also, if it ever went down, Internet security would suddenly grind to a halt.

For these reasons, people have developed a different solution, one that does not require the key distribution center to be online all the time. In fact, it does not have to be online at all. Instead, what it does is certify the public keys belonging to people, companies, and other organizations. An organization that certifies public keys is now called a **CA** (**Certification Authority**).

As an example, suppose that Bob wants to allow Alice and other people he does not know to communicate with him securely. He can go to the CA with his public key along with his passport or driver's license and ask to be certified. The CA then issues a certificate similar to the one in Fig. 8-26 and signs its SHA-2 hash with the CA's private key. Bob then pays the CA's fee and gets a document containing the certificate and its signed hash (ideally not sent over unreliable channels).

Figure 8-26. A possible certificate and its signed hash.

The fundamental job of a certificate is to bind a public key to the name of a principal (individual, company, etc.). Certificates themselves are not secret or protected. Bob might, for example, decide to put his new certificate on his Web site, with a link on the main page saying: click here for my public-key certificate. The resulting click would return both the certificate and the signature block (the signed SHA-2 hash of the certificate).

Now let us run through the scenario of Fig. 8-25 again. When Trudy intercepts Alice's request for Bob's home page, what can she do? She can put her own certificate and signature block on the fake page, but when Alice reads the contents of the certificate she will immediately see that she is not talking to Bob because Bob's name is not in it. Trudy can modify Bob's home page on the fly, replacing Bob's public key with her own. However, when Alice runs the SHA-2 algorithm on the certificate, she will get a hash that does not agree with the one she gets when she applies the CA's well-known public key to the signature block. Since Trudy does not have the CA's private key, she has no way of generating a signature block that contains the hash of the modified Web page with her public key on it. In this way, Alice can be sure she has Bob's public key and not Trudy's or someone else's.

And as we promised, this scheme does not require the CA to be online for verification, thus eliminating a potential bottleneck.

While the standard function of a certificate is to bind a public key to a principal, a certificate can also be used to bind a public key to an **attribute**. For example, a certificate could say: ''This public key belongs to someone over 18.'' It could be used to prove that the owner of the private key was not a minor and thus allowed to access material not suitable for children, and so on, but without disclosing the owner's identity. Typically, the person holding the certificate would send it to the Web site, principal, or process that cared about age. That site, principal, or process would then generate a random number and encrypt it with the public key in the certificate. If the owner were able to decrypt it and send it back, that would be proof that the owner indeed had the attribute stated in the certificate. Alternatively, the random number could be used to generate a session key for the ensuing conversation.

Another example of where a certificate might contain an attribute is in an object-oriented distributed system. Each object normally has multiple methods. The owner of the object could provide each customer with a certificate giving a bit map of which methods the customer is allowed to invoke and binding the bit map to a public key using a signed certificate. Again, if the certificate holder can prove possession of the corresponding private key, he will be allowed to perform the methods in the bit map. This approach has the property that the owner's identity need not be known, a property useful in situations where privacy is important.

8.8.2 X.509

If everybody who wanted something signed went to the CA with a different kind of certificate, managing all the different formats would soon become a problem. To solve this problem, a standard for certificates has been devised and approved by the International Telecommunication Union (ITU). The standard is called **X.509** and is in widespread use on the Internet. It has gone through three versions since the initial standardization in 1988. We will discuss version 3.

X.509 has been heavily influenced by the OSI world, borrowing some of its worst features (e.g., naming and encoding). Surprisingly, IETF went along with X.509, even though in nearly every other area, from machine addresses to transport protocols to email formats, IETF generally ignored OSI and tried to do it right. The IETF version of X.509 is described in RFC 5280.

At its core, X.509 is a way to describe certificates. The primary fields in a certificate are listed in Fig. 8-27. The descriptions given there should provide a general idea of what the fields do. For additional information, please consult the standard itself or RFC 2459.

For example, if Bob works in the loan department of the Money Bank, his X.500 address might be

```
/C=US/O=MoneyBank/OU=Loan/CN=Bob/
```

Field	Meaning
Version	Which version of X.509
Serial number	This number plus the CA's name uniquely identifies the certificate
Signature algorithm	The algorithm used to sign the certificate
Issuer	X.500 name of the CA
Validity period	The starting and ending times of the validity period
Subject name	The entity whose key is being certified
Public key	The subject's public key and the ID of the algorithm using it
Issuer ID	An optional ID uniquely identifying the certificate's issuer
Subject ID	An optional ID uniquely identifying the certificate's subject
Extensions	Many extensions have been defined
Signature	The certificate's signature (signed by the CA's private key)

Figure 8-27. The basic fields of an X.509 certificate.

where *C* is for country, *O* is for organization, *OU* is for organizational unit, and *CN* is for common name. CAs and other entities are named in a similar way. A substantial problem with X.500 names is that if Alice is trying to contact *bob@moneybank.com* and is given a certificate with an X.500 name, it may not be obvious to her that the certificate refers to the Bob she wants. Fortunately, starting with version 3, DNS names are now permitted instead of X.500 names, so this problem may eventually vanish.

Certificates are encoded using OSI **ASN.1** (**Abstract Syntax Notation 1**), which is sort of like a struct in C, except with an extremely peculiar and verbose notation. More information about X.509 is given by Ford and Baum (2000).

8.8.3 Public Key Infrastructures

Having a single CA to issue all the world's certificates obviously would not work. It would collapse under the load and be a central point of failure as well. A possible solution might be to have multiple CAs, all run by the same organization and all using the same private key to sign certificates. While this would solve the load and failure problems, it introduces a new problem: key leakage. If there were dozens of servers spread around the world, all holding the CA's private key, the chance of the private key being stolen or otherwise leaking out would be greatly increased. Since the compromise of this key would ruin the world's electronic security infrastructure, having a single central CA is very risky.

In addition, which organization would operate the CA? It is hard to imagine any authority that would be accepted worldwide as legitimate and trustworthy. In some countries, people would insist that it be a government, while in other countries they would insist that it not be a government.

For these reasons, a different way for certifying public keys has evolved. It goes under the general name of **PKI** (**Public Key Infrastructure**). In this section, we will summarize how it works in general, although there have been many proposals, so the details will probably evolve in time.

A PKI has multiple components, including users, CAs, certificates, and directories. What the PKI does is provide a way of structuring these components and define standards for the various documents and protocols. A particularly simple form of PKI is a hierarchy of CAs, as depicted in Fig. 8-28. In this example, we have shown three levels, but in practice, there might be fewer or more. The top-level CA, the root, certifies second-level CAs, which we here call **RA**s (**Regional Authorities**) because they might cover some geographic region, such as a country or continent. This term is not standard, though; in fact, no term is really standard for the different levels of the tree. These, in turn, certify the real CAs, which issue the X.509 certificates to organizations and individuals. When the root authorizes a new RA, it generates an X.509 certificate stating that it has approved the RA, includes the new RA's public key in it, signs it, and hands it to the RA. Similarly, when an RA approves a new CA, it produces and signs a certificate stating its approval and containing the CA's public key.

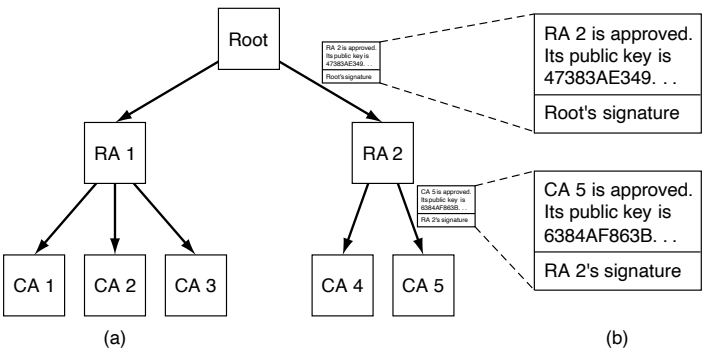

Figure 8-28. (a) A hierarchical PKI. (b) A chain of certificates.

Our PKI works like this. Suppose that Alice needs Bob's public key in order to communicate with him, so she looks for and finds a certificate containing it, signed by CA 5. But Alice has never heard of CA 5. For all she knows, CA 5 might be Bob's 10-year-old daughter. She could go to CA 5 and say: ''Prove your legitimacy.'' CA 5 will respond with the certificate it got from RA 2, which contains CA 5's public key. Now armed with CA 5's public key, she can verify that Bob's certificate was indeed signed by CA 5 and is thus legal.

Unless RA 2 is Bob's 12-year-old son. So, the next step is for her to ask RA 2 to prove it is legitimate. The response to her query is a certificate signed by the root and containing RA 2's public key. Now Alice is sure she has Bob's public key.

But how does Alice find the root's public key? Magic. It is assumed that everyone knows the root's public key. For example, her browser might have been shipped with the root's public key built in.

Bob is a friendly sort of guy and does not want to cause Alice a lot of work. He knows that she will have to check out CA 5 and RA 2, so to save her some trouble, he collects the two needed certificates and gives her the two certificates along with his. Now she can use her own knowledge of the root's public key to verify the top-level certificate and the public key contained therein to verify the second one. Alice does not need to contact anyone to do the verification. Because the certificates are all signed, she can easily detect any attempts to tamper with their contents. A chain of certificates going back to the root like this is sometimes called a **chain of trust** or a **certification path**. The technique is widely used in practice.

Of course, we still have the problem of who is going to run the root. The solution is not to have a single root, but to have many roots, each with its own RAs and CAs. In fact, modern browsers come preloaded with the public keys for over 100 roots, sometimes referred to as **trust anchors**. In this way, having a single worldwide trusted authority can be avoided.

But there is now the issue of how the browser vendor decides which purported trust anchors are reliable and which are sleazy. It all comes down to the user trusting the browser vendor to make wise choices and not simply approve all trust anchors willing to pay its inclusion fee. Most browsers allow users to inspect the root keys (usually in the form of certificates signed by the root) and delete any that seem shady. For more information on PKIs, see Stapleton and Epstein (2016).

Directories

Another issue for any PKI is where certificates (and their chains back to some known trust anchor) are stored. One possibility is to have each user store his or her own certificates. While doing this is safe (i.e., there is no way for users to tamper with signed certificates without detection), it is also inconvenient. One alternative that has been proposed is to use DNS as a certificate directory. Before contacting Bob, Alice probably has to look up his IP address using DNS, so why not have DNS return Bob's entire certificate chain along with his IP address?

Some people think this is the way to go, but others would prefer dedicated directory servers whose only job is managing X.509 certificates. Such directories could provide lookup services by using properties of the X.500 names. For example, in theory, such a directory service could answer queries like ''Give me a list of all people named Alice who work in sales departments anywhere in the U.S.''

Revocation

The real world is full of certificates, too, such as passports and drivers' licenses. Sometimes these certificates can be revoked, for example, drivers' licenses can be revoked for drunken driving and other driving offenses. The same

problem occurs in the digital world: the grantor of a certificate may decide to revoke it because the person or organization holding it has abused it in some way. It can also be revoked if the subject's private key has been exposed or, worse yet, the CA's private key has been compromised. Thus, a PKI needs to deal with the issue of revocation. The possibility of revocation complicates matters.

A first step in this direction is to have each CA periodically issue a **CRL** (**Certificate Revocation List**) giving the serial numbers of all certificates that it has revoked. Since certificates contain expiry times, the CRL need only contain the serial numbers of certificates that have not yet expired. Once its expiry time has passed, a certificate is automatically invalid, so no distinction is needed between those that just timed out and those that were actually revoked. In both cases, they cannot be used any more.

Unfortunately, introducing CRLs means that a user who is about to use a certificate must now acquire the CRL to see if the certificate has been revoked. If it has been, it should not be used. However, even if the certificate is not on the list, it might have been revoked just after the list was published. Thus, the only way to really be sure is to ask the CA. And on the next use of the same certificate, the CA has to be asked again, since the certificate might have been revoked a few seconds ago.

Another complication is that a revoked certificate could conceivably be reinstated, for example, if it was revoked for nonpayment of some fee that has since been paid. Having to deal with revocation (and possibly reinstatement) eliminates one of the best properties of certificates, namely, that they can be used without having to contact a CA.

Where should CRLs be stored? A good place would be the same place the certificates themselves are stored. One strategy is for the CA to actively push out CRLs periodically and have the directories process them by simply removing the revoked certificates. If directories are not used for storing certificates, the CRLs can be cached at various places around the network. Since a CRL is itself a signed document, if it is tampered with, that tampering can be easily detected.

If certificates have long lifetimes, the CRLs will be long, too. For example, if credit cards are valid for 5 years, the number of revocations outstanding will be much longer than if new cards are issued every 3 months. A standard way to deal with long CRLs is to issue a master list infrequently, but issue updates to it more often. Doing this reduces the bandwidth needed for distributing the CRLs.

8.9 AUTHENTICATION PROTOCOLS

Authentication is the technique by which a process verifies that its communication partner is who it is supposed to be and not an imposter. Verifying the identity of a remote process in the face of a malicious, active intruder is surprisingly difficult and requires complex protocols based on cryptography. In this section, we

will study some of the many authentication protocols that are used on insecure computer networks.

As an aside, some people confuse authorization with authentication. Authentication deals with the question of whether you are actually communicating with a specific process. Authorization is concerned with what that process is permitted to do. For example, say a client process contacts a file server and says: ''I am Mirte's process and I want to delete the file *cookbook.old*.'' From the file server's point of view, two questions must be answered:

1. Is this actually Mirte's process (authentication)?
2. Is Mirte allowed to delete *cookbook.old* (authorization)?

Only after both of these questions have been unambiguously answered in the affirmative can the requested action take place. The former question is really the key one. Once the file server knows to whom it is talking, checking authorization is just a matter of looking up entries in local tables or databases. For this reason, we will concentrate on authentication in this section.

The general model that essentially all authentication protocols use is this. Alice starts out by sending a message either to Bob or to a trusted KDC, which is expected to be honest. Several other message exchanges follow in various directions. As these messages are being sent, Trudy may intercept, modify, or replay them in order to trick Alice and Bob or just to gum up the works.

Nevertheless, when the protocol has been completed, Alice is sure she is talking to Bob and Bob is sure he is talking to Alice. Furthermore, in most of the protocols, the two of them will also have established a secret **session key** for use in the upcoming conversation. In practice, for performance reasons, all data traffic is encrypted using symmetric-key cryptography (typically AES), although public-key cryptography is widely used for the authentication protocols themselves and for establishing the session key.

The point of using a new, randomly chosen session key for each new connection is to minimize the amount of traffic that gets sent with the users' secret keys or public keys, to reduce the amount of ciphertext an intruder can obtain, and to minimize the damage done if a process crashes and its core dump (memory printout after a crash) falls into the wrong hands. Hopefully, the only key present then will be the session key. All the permanent keys should have been carefully zeroed out after the session was established.

8.9.1 Authentication Based on a Shared Secret Key

For our first authentication protocol, we will assume that Alice and Bob already share a secret key, K_{AB}. This shared key might have been agreed upon on the telephone or in person, but, in any event, not on the (insecure) network.

This protocol is based on a principle found in many authentication protocols: one party sends a random number to the other, who then transforms it in a special way and returns the result. Such protocols are called **challenge-response** protocols. In this and subsequent authentication protocols, the following notation will be used:

A, B are the identities of Alice and Bob.
R_i's are the challenges, where i identifies the challenger.
K_i's are keys, where i indicates the owner.
K_S is the session key.

The message sequence for our first shared-key authentication protocol is illustrated in Fig. 8-29. In message 1, Alice sends her identity, A, to Bob in a way that Bob understands. Bob, of course, has no way of knowing whether this message came from Alice or from Trudy, so he chooses a challenge, a large random number, R_B, and sends it back to ''Alice'' as message 2, in plaintext. Alice then encrypts the message with the key she shares with Bob and sends the ciphertext, $K_{AB}(R_B)$, back in message 3. When Bob sees this message, he immediately knows that it came from Alice because Trudy does not know K_{AB} and thus could not have generated it. Furthermore, since R_B was chosen randomly from a large space (say, 128-bit random numbers), it is very unlikely that Trudy would have seen R_B and its response in an earlier session. It is equally unlikely that she could guess the correct response to any challenge.

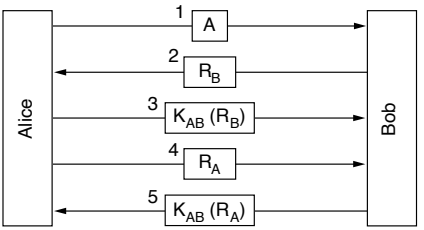

Figure 8-29. Two-way authentication using a challenge-response protocol.

At this point, Bob is sure he is talking to Alice, but Alice is not sure of anything. For all Alice knows, Trudy might have intercepted message 1 and sent back R_B in response. Maybe Bob died last night. To find out to whom she is talking, Alice picks a random number, R_A, and sends it to Bob as plaintext, in message 4. When Bob responds with $K_{AB}(R_A)$, Alice knows she is talking to Bob. If they wish to establish a session key now, Alice can pick one, K_S, and send it to Bob encrypted with K_{AB}.

The protocol of Fig. 8-29 contains five messages. Let us see if we can be clever and eliminate some of them. One approach is illustrated in Fig. 8-30. Here

Alice initiates the challenge-response protocol instead of waiting for Bob to do it. Similarly, while he is responding to Alice's challenge, Bob sends his own. The entire protocol can be reduced to three messages instead of five.

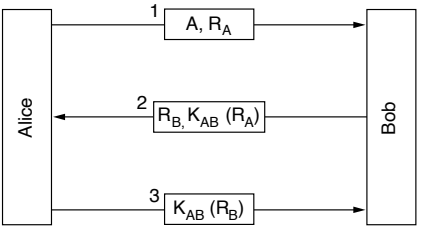

Figure 8-30. A shortened two-way authentication protocol.

Is this new protocol an improvement over the original one? In one sense it is: it is shorter. Unfortunately, it is also wrong. Under certain circumstances, Trudy can defeat this protocol by using what is known as a **reflection attack**. In particular, Trudy can break it if it is possible to open multiple sessions with Bob at once. This situation would be true, for example, if Bob is a bank and is prepared to accept many simultaneous connections from automated teller machines at once.

Trudy's reflection attack is shown in Fig. 8-31. It starts out with Trudy claiming she is Alice and sending R_T. Bob responds, as usual, with his own challenge, R_B. Now Trudy is stuck. What can she do? She does not know $K_{AB}(R_B)$.

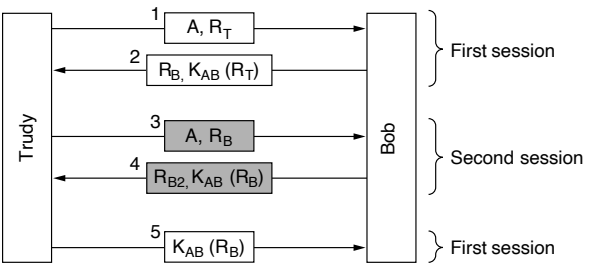

Figure 8-31. The reflection attack.

She can open a second session with message 3, supplying the R_B taken from message 2 as her challenge. Bob calmly encrypts it and sends back $K_{AB}(R_B)$ in message 4. We have shaded the messages on the second session to make them stand out. Now Trudy has the missing information, so she can complete the first session and abort the second one. Bob is now convinced that Trudy is Alice, so

when she asks for her bank account balance, he gives it to her without question. Then when she asks him to transfer it all to a secret bank account in Switzerland, he does so without a moment's hesitation.

The moral of this story is:

Designing a correct authentication protocol is much harder than it looks.

The following four general rules often help the designer avoid common pitfalls:

1. Have the initiator prove who she is before the responder has to. This avoids Bob giving away valuable information before Trudy has to give any evidence of who she is.
2. Have the initiator and responder use different keys for proof, even if this means having two shared keys, K_{AB} and K'_{AB}.
3. Have the initiator and responder draw their challenges from different sets. For example, the initiator must use even numbers and the responder must use odd numbers.
4. Make the protocol resistant to attacks involving a second parallel session in which information obtained in one session is used in a different one.

If even one of these rules is violated, the protocol can frequently be broken. Here, all four rules were violated, with disastrous consequences.

Now let us go take a closer look at Fig. 8-29. Surely that protocol is not subject to a reflection attack? Maybe. It is quite subtle. Trudy was able to defeat our protocol by using a reflection attack because it was possible to open a second session with Bob and trick him into answering his own questions. What would happen if Alice were a general-purpose computer that also accepted multiple sessions, rather than a person at a computer? Let us take a look what Trudy can do.

To see how Trudy's attack works, see Fig. 8-32. Alice starts out by announcing her identity in message 1. Trudy intercepts this message and begins her own session with message 2, claiming to be Bob. Again we have shaded the session 2 messages. Alice responds to message 2 by saying in message 3: "You claim to be Bob? Prove it." At this point, Trudy is stuck because she cannot prove she is Bob.

What does Trudy do now? She goes back to the first session, where it is her turn to send a challenge, and sends the R_A she got in message 3. Alice kindly responds to it in message 5, thus supplying Trudy with the information she needs to send in message 6 in session 2. At this point, Trudy is basically home free because she has successfully responded to Alice's challenge in session 2. She can now cancel session 1, send over any old number for the rest of session 2, and she will have an authenticated session with Alice in session 2.

But Trudy is a perfectionist, and she really wants to show off her considerable skills. Instead, of sending any old number over to complete session 2, she waits

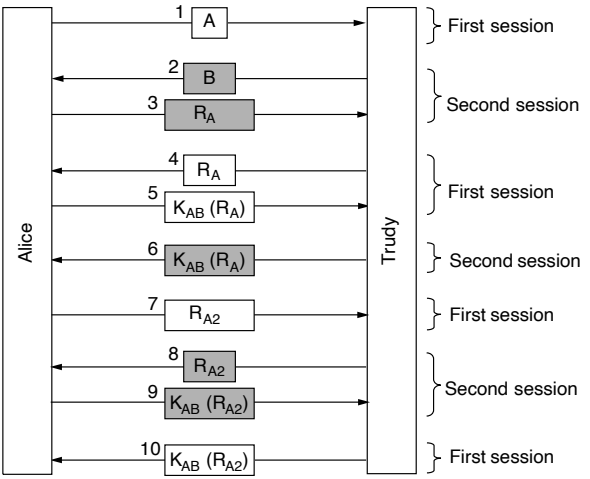

Figure 8-32. A reflection attack on the protocol of Fig. 8-29.

until Alice sends message 7, Alice's challenge for session 1. Of course, Trudy does not know how to respond, so she uses the reflection attack again, sending back R_{A2} as message 8. Alice conveniently encrypts R_{A2} in message 9. Trudy now switches back to session 1 and sends Alice the number she wants in message 10, conveniently copied from what Alice sent in message 9. At this point, Trudy has two fully authenticated sessions with Alice.

This attack has a somewhat different result than the attack on the three-message protocol that we saw in Fig. 8-31. This time, Trudy has two authenticated connections with Alice. In the previous example, she had one authenticated connection with Bob. Again here, if we had applied all the general authentication protocol rules discussed earlier, this attack could have been stopped. For a detailed discussion of these kinds of attacks and how to thwart them, see Bird et al. (1993). They also show how it is possible to systematically construct protocols that are provably correct. The simplest such protocol is nevertheless fairly complicated, so we will now show a different class of protocol that also works.

The new authentication protocol is shown in Fig. 8-33 (Bird et al., 1993). It uses a **HMAC** (**Hashed Message Authentication Code**) which guarantees the integrity and authenticity of a message. A simple, yet powerful HMAC consists of a hash over the message plus the shared key. By sending the HMAC along with the rest of the message, no attacker is able to change or spoof the message: changing any bit would lead to an incorrect hash, and generating a valid hash is not possible without the key. HMACs are attractive because they can be generated very efficiently (faster than running SHA-2 and then running RSA on the result).

Alice starts out by sending Bob a random number, R_A, as message 1. Random numbers used just once in security protocols like this one are called **nonces**, which is more-or-less a contraction of ''number used once.'' Bob responds by selecting his own nonce, R_B, and sending it back along with an HMAC. The HMAC is formed by building a data structure consisting of Alice's nonce, Bob's nonce, their identities, and the shared secret key, K_{AB}. This data structure is then hashed into the HMAC, for example, using SHA-2. When Alice receives message 2, she now has R_A (which she picked herself), R_B, which arrives as plaintext, the two identities, and the secret key, K_{AB}, which she has known all along, so she can compute the HMAC herself. If it agrees with the HMAC in the message, she knows she is talking to Bob because Trudy does not know K_{AB} and thus cannot figure out which HMAC to send. Alice responds to Bob with an HMAC containing just the two nonces.

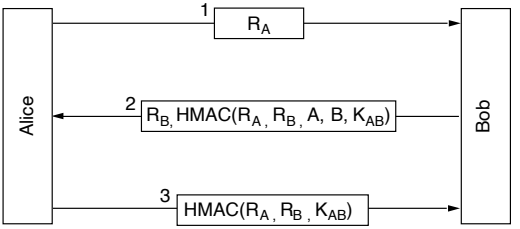

Figure 8-33. Authentication using HMACs.

Can Trudy somehow subvert this protocol? No, because she cannot force either party to encrypt or hash a value of her choice, as happened in Fig. 8-31 and Fig. 8-32. Both HMACs include values chosen by the sending party, something that Trudy cannot control.

Using HMACs is not the only way to use this idea. An alternative scheme that is often used instead of computing the HMAC over a series of items is to encrypt the items sequentially using cipher block chaining.

8.9.2 Establishing a Shared Key: The Diffie-Hellman Key Exchange

So far, we have assumed that Alice and Bob share a secret key. Suppose that they do not (because so far there is no universally accepted PKI for signing and distributing certificates). How can they establish one? One way would be for Alice to call Bob and give him her key on the phone, but he would probably start out by saying: ''How do I know you are Alice and not Trudy?'' They could try to arrange a meeting, with each one bringing a passport, a driver's license, and three major credit cards, but being busy people, they might not be able to find a mutually acceptable date for months. Fortunately, incredible as it may sound, there is a way for total strangers to establish a shared secret key in broad daylight, even with Trudy carefully recording every message.

The protocol that allows strangers to establish a shared secret key is called the **Diffie-Hellman key exchange** (Diffie and Hellman, 1976) and works as follows. Alice and Bob have to agree on two large numbers, n and g, where n is a prime, $(n - 1)/2$ is also a prime, and certain conditions apply to g. These numbers may be public, so either one of them can just pick n and g and tell the other openly. Now Alice picks a large (say, 1024-bit) number, x, and keeps it secret. Similarly, Bob picks a large secret number, y.

Alice initiates the key exchange protocol by sending Bob a (plaintext) message containing $(n, g, g^x \bmod n)$, as shown in Fig. 8-34. Bob responds by sending Alice a message containing $g^y \bmod n$. Now Alice raises the number Bob sent her to the xth power modulo n to get $(g^y \bmod n)^x \bmod n$. Bob performs a similar operation to get $(g^x \bmod n)^y \bmod n$. By the laws of modular arithmetic, both calculations yield $g^{xy} \bmod n$. Lo and behold, as if by magic, Alice and Bob suddenly share a secret key, $g^{xy} \bmod n$.

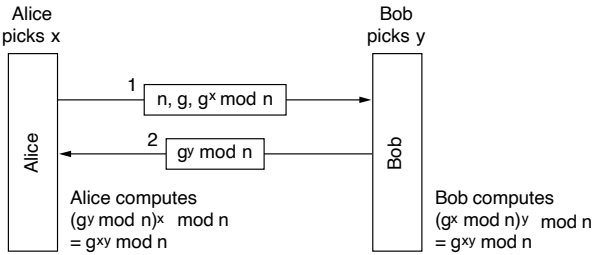

Figure 8-34. The Diffie-Hellman key exchange.

Trudy, of course, has seen both messages. She knows g and n from message 1. If she could compute x and y, she could figure out the secret key. The trouble is, given only $g^x \bmod n$, she cannot find x. No practical algorithm for computing discrete logarithms modulo a very large prime number is known.

To make this example more concrete, we will use the (completely unrealistic) values of $n = 47$ and $g = 3$. Alice picks $x = 8$ and Bob picks $y = 10$. Both of these are kept secret. Alice's message to Bob is (47, 3, 28) because $3^8 \bmod 47$ is 28. Bob's message to Alice is (17). Alice computes $17^8 \bmod 47$, which is 4. Bob computes $28^{10} \bmod 47$, which is 4. Alice and Bob have now independently determined that the secret key is now 4. To find the key, Trudy now has to solve the equation $3^x \bmod 47 = 28$, which can be done by exhaustive search for small numbers like this, but not when all the numbers are hundreds or thousands of bits long. All currently known algorithms simply take far too long, even on lightning-fast supercomputers with tens of millions of cores.

Despite the elegance of the Diffie-Hellman algorithm, there is a problem: when Bob gets the triple (47, 3, 28), how does he know it is from Alice and not from Trudy? There is no way he can know. Unfortunately, Trudy can exploit this fact to

deceive both Alice and Bob, as illustrated in Fig. 8-35. Here, while Alice and Bob are choosing x and y, respectively, Trudy picks her own random number, z. Alice sends message 1, intended for Bob. Trudy intercepts it and sends message 2 to Bob, using the correct g and n (which are public anyway) but with her own z instead of x. She also sends message 3 back to Alice. Later Bob sends message 4 to Alice, which Trudy again intercepts and keeps.

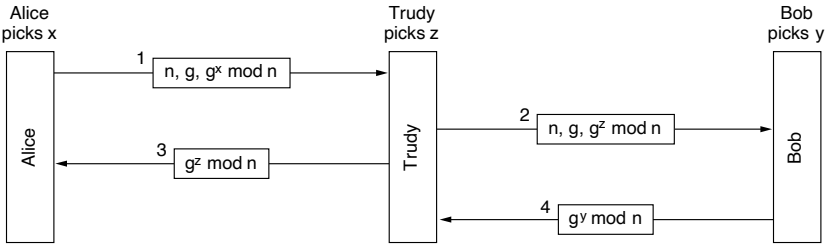

Figure 8-35. The man-in-the-middle attack.

Now everybody does the modular arithmetic. Alice computes the secret key as $g^{xz} \bmod n$, and so does Trudy (for messages to Alice). Bob computes $g^{yz} \bmod n$ and so does Trudy (for messages to Bob). Alice thinks she is talking to Bob, so she establishes a session key (with Trudy). So does Bob. Every message that Alice sends on the encrypted session is captured by Trudy, stored, modified if desired, and then (optionally) passed on to Bob. Similarly, in the other direction, Trudy sees everything and can modify all messages at will, while both Alice and Bob are under the illusion that they have a secure channel to one another. For this reason, the attack is known as the **man-in-the-middle attack**. It is also called the **bucket brigade attack**, because it vaguely resembles an old-time volunteer fire department passing buckets along the line from the fire truck to the fire.

8.9.3 Authentication Using a Key Distribution Center

Setting up a shared secret with a stranger almost worked, but not quite. On the other hand, it probably was not worth doing in the first place (sour grapes attack). To talk to n people this way, you would need n keys. For popular people, key management would become a real burden, especially if each key had to be stored on a separate plastic chip card.

A different approach is to introduce a trusted Key Distribution Center, such as a bank or government office, into the system. In this model, each user has a single key shared with the KDC. Authentication and session key management now go through the KDC. The simplest known KDC authentication protocol involving two parties and a trusted KDC is depicted in Fig. 8-36.

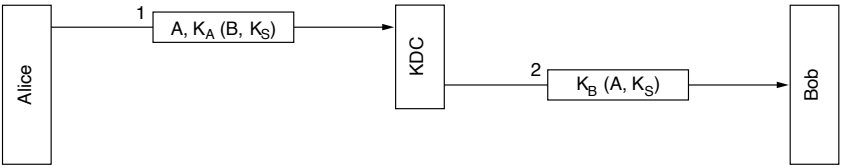

Figure 8-36. A first attempt at an authentication protocol using a KDC.

The idea behind this protocol is simple: Alice picks a session key, K_S, and tells the KDC that she wants to talk to Bob using K_S. This message is encrypted with the secret key Alice shares (only) with the KDC, K_A. The KDC decrypts this message, extracting Bob's identity and the session key. It then constructs a new message containing Alice's identity and the session key and sends this message to Bob. This encryption is done with K_B, the secret key Bob shares with the KDC. When Bob decrypts the message, he learns that Alice wants to talk to him and which key she wants to use.

The authentication here happens completely for free. The KDC knows that message 1 must have come from Alice, since no one else would have been able to encrypt it with Alice's secret key. Similarly, Bob knows that message 2 must have come from the KDC, which he trusts, since no one else knows his secret key.

Unfortunately, this protocol has a serious flaw. Trudy needs some money, so she figures out some legitimate service she can perform for Alice, makes an attractive offer, and, bingo, she gets the job. After doing the work, Trudy then politely requests Alice to pay by bank transfer. Alice then establishes a session key with her banker, Bob. Then she sends Bob a message requesting money to be transferred to Trudy's account.

Meanwhile, Trudy is back to her old ways, snooping on the network. She copies both message 2 in Fig. 8-36 and the money-transfer request that follows it. Later, she replays both of them to Bob who thinks: ''Alice must have hired Trudy again. She clearly does good work.'' Bob then transfers an equal amount of money from Alice's account to Trudy's. Sometime after the 50th message pair, Bob runs out of the office to find Trudy to offer her a big loan so she can expand her obviously successful business. This problem is called the **replay attack**.

Several solutions to the replay attack are possible. The first one is to include a timestamp in each message. Then, if anyone receives an old message, it can be discarded. The trouble with this approach is that clocks are never exactly synchronized over a network, so there has to be some interval during which a timestamp is valid. Trudy can replay the message during this interval and get away with it.

The second solution is to put a nonce in each message. Each party then has to remember all previous nonces and reject any message containing a previously used

nonce. But nonces have to be remembered forever, lest Trudy try replaying a 5-year-old message. Also, if some machine crashes and it loses its nonce list, it is again vulnerable to a replay attack. Timestamps and nonces can be combined to limit how long nonces have to be remembered, but clearly the protocol is going to get a lot more complicated.

A more sophisticated approach to mutual authentication is to use a multiway challenge-response protocol. A well-known example of such a protocol is the **Needham-Schroeder authentication** protocol (Needham and Schroeder, 1978), one variant of which is shown in Fig. 8-37.

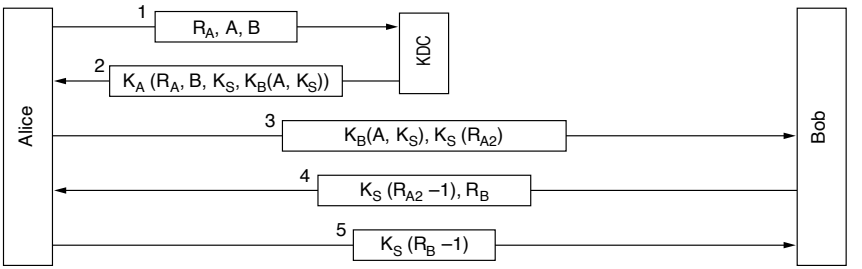

Figure 8-37. The Needham-Schroeder authentication protocol.

The protocol begins with Alice telling the KDC that she wants to talk to Bob. This message contains a large random number, R_A, as a nonce. The KDC sends back message 2 containing Alice's random number, a session key, and a ticket that she can send to Bob. The point of the random number, R_A, is to assure Alice that message 2 is fresh, and not a replay. Bob's identity is also enclosed in case Trudy gets any funny ideas about replacing B in message 1 with her own identity so the KDC will encrypt the ticket at the end of message 2 with K_T instead of K_B. The ticket encrypted with K_B is included inside the encrypted message to prevent Trudy from replacing it with something else on the way back to Alice.

Alice now sends the ticket to Bob, along with a new random number, R_{A2}, encrypted with the session key, K_S. In message 4, Bob sends back $K_S(R_{A2} - 1)$ to prove to Alice that she is talking to the real Bob. Sending back $K_S(R_{A2})$ would not have worked, since Trudy could just have stolen it from message 3.

After receiving message 4, Alice is now convinced that she is talking to Bob and that no replays could have been used so far. After all, she just generated R_{A2} a few milliseconds ago. The purpose of message 5 is to convince Bob that it is indeed Alice he is talking to, and no replays are being used here either. By having each party both generate a challenge and respond to one, the possibility of any kind of replay attack is eliminated.

Although this protocol seems pretty solid, it does have a slight weakness. If Trudy ever manages to obtain an old session key in plaintext, she can initiate a new session with Bob by replaying the message 3 that corresponds to the compromised key and convince him that she is Alice (Denning and Sacco, 1981). This time she can plunder Alice's bank account without having to perform the legitimate service even once.

Needham and Schroeder (1987) later published a protocol that corrects this problem. In the same issue of the same journal, Otway and Rees (1987) also published a protocol that solves the problem in a shorter way. Figure 8-38 shows a slightly modified Otway-Rees protocol.

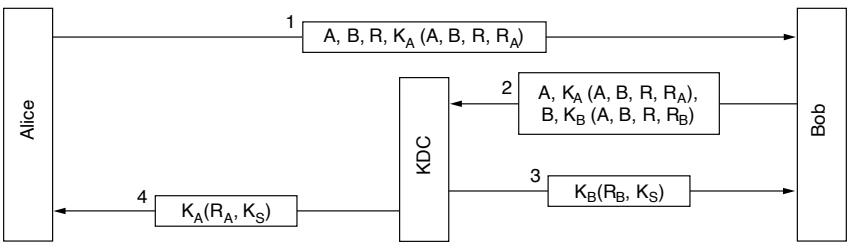

Figure 8-38. The Otway-Rees authentication protocol (slightly simplified).

In the Otway-Rees protocol, Alice starts out by generating a pair of random numbers: R, which will be used as a common identifier, and R_A, which Alice will use to challenge Bob. When Bob gets this message, he constructs a new message from the encrypted part of Alice's message and an analogous one of his own. Both the parts encrypted with K_A and K_B identify Alice and Bob, contain the common identifier, and contain a challenge.

The KDC checks to see if the R in both parts is the same. It might not be if Trudy has tampered with R in message 1 or replaced part of message 2. If the two Rs match, the KDC believes that the request message from Bob is valid. It then generates a session key and encrypts it twice, once for Alice and once for Bob. Each message contains the receiver's random number, as proof that the KDC, and not Trudy, generated the message. At this point, both Alice and Bob are in possession of the same session key and can start communicating. The first time they exchange data messages, each one can see that the other one has an identical copy of K_S, so the authentication is then complete.

8.9.4 Authentication Using Kerberos

An authentication protocol used in many real systems (including Windows) is **Kerberos**, which is based on a variant of Needham-Schroeder. It is named for a multiheaded dog in Greek mythology that used to guard the entrance to Hades

(presumably to keep undesirables out). Kerberos was designed at M.I.T. to allow workstation users to access network resources in a secure way. Its biggest difference from Needham-Schroeder is its assumption that all clocks are fairly well synchronized. The protocol has gone through several iterations. V5 is the one that is widely used in industry and defined in RFC 4120. The earlier version, V4, was finally retired after serious flaws were found (Yu et al., 2004). V5 improves on V4 with many small changes to the protocol and some improved features, such as the fact that it no longer relies on the now-dated DES. For more information, see Sood (2012).

Kerberos involves three servers in addition to Alice (a client workstation):

1. Authentication Server (AS): verifies users during login.
2. Ticket-Granting Server (TGS): issues ''proof of identity tickets.''
3. Bob the server: actually does the work Alice wants performed.

AS is similar to a KDC in that it shares a secret password with every user. The TGS's job is to issue tickets that can convince the real servers that the bearer of a TGS ticket really is who he or she claims to be.

To start a session, Alice sits down at an arbitrary public workstation and types her name. The workstation sends her name and the name of the TGS to the AS in plaintext, as shown in message 1 of Fig. 8-39. What comes back is a session key and a ticket, $K_{TGS}(A, K_S, t)$, intended for the TGS. The session key is encrypted using Alice's secret key, so that only Alice can decrypt it. Only when message 2 arrives does the workstation ask for Alice's password—not before then. The password is then used to generate K_A in order to decrypt message 2 and obtain the session key.

At this point, the workstation overwrites Alice's password to make sure that it is only inside the workstation for a few milliseconds at most. If Trudy tries logging in as Alice, the password she types will be wrong and the workstation will detect this because the standard part of message 2 will be incorrect.

After she logs in, Alice may tell the workstation that she wants to contact Bob the file server. The workstation then sends message 3 to the TGS asking for a ticket to use with Bob. The key element in this request is the ticket $K_{TGS}(A, K_S, t)$, which is encrypted with the TGS's secret key and used as proof that the sender really is Alice. The TGS responds in message 4 by creating a session key, K_{AB}, for Alice to use with Bob. Two versions of it are sent back. The first is encrypted with only K_S, so Alice can read it. The second is another ticket, encrypted with Bob's key, K_B, so Bob can read it.

Trudy can copy message 3 and try to use it again, but she will be foiled by the encrypted timestamp, t, sent along with it. Trudy cannot replace the timestamp with a more recent one because she does not know K_S, the session key Alice uses

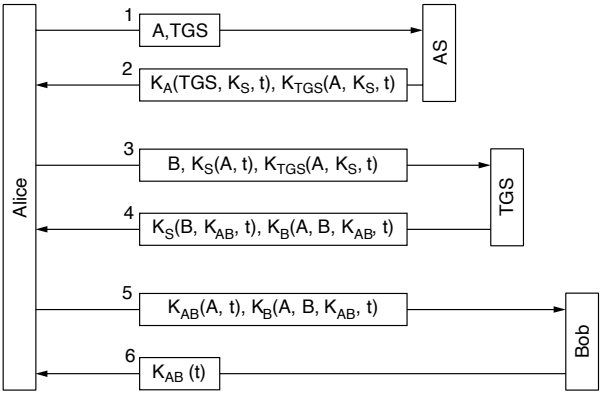

Figure 8-39. The operation of Kerberos V5.

to talk to the TGS. Even if Trudy replays message 3 quickly, all she will get is another copy of message 4, which she could not decrypt the first time and will not be able to decrypt the second time either.

Now Alice can send K_{AB} to Bob via the new ticket to establish a session with him (message 5). This exchange is also timestamped. The optional response (message 6) is proof to Alice that she is actually talking to Bob, not to Trudy.

After this series of exchanges, Alice can communicate with Bob under cover of K_{AB}. If she later decides she needs to talk to another server, Carol, she just repeats message 3 to the TGS, only now specifying C instead of B. The TGS will promptly respond with a ticket encrypted with K_C that Alice can send to Carol and that Carol will accept as proof that it came from Alice.

The point of all this work is that now Alice can access servers all over the network in a secure way and her password never has to go over the network. In fact, it only had to be in her own workstation for a few milliseconds. However, note that each server does its own authorization. When Alice presents her ticket to Bob, this merely proves to Bob who sent it. Precisely what Alice is allowed to do is up to Bob.

Since the Kerberos designers did not expect the entire world to trust a single authentication server, they made provision for having multiple **realms**, each with its own AS and TGS. To get a ticket for a server in a distant realm, Alice would ask her own TGS for a ticket accepted by the TGS in the distant realm. If the distant TGS has registered with the local TGS (the same way local servers do), the local TGS will give Alice a ticket valid at the distant TGS. She can then do business over there, such as getting tickets for servers in that realm. Note, however, that for parties in two realms to do business, each one must trust the other's TGS. Otherwise, they cannot do business.

8.9.5 Authentication Using Public-Key Cryptography

Mutual authentication can also be done using public-key cryptography. To start with, Alice needs to get Bob's public key. If a PKI exists with a directory server that hands out certificates for public keys, Alice can ask for Bob's, as shown in Fig. 8-40 as message 1. The reply, in message 2, is an X.509 certificate containing Bob's public key. When Alice verifies that the signature is correct, she sends Bob a message containing her identity and a nonce.

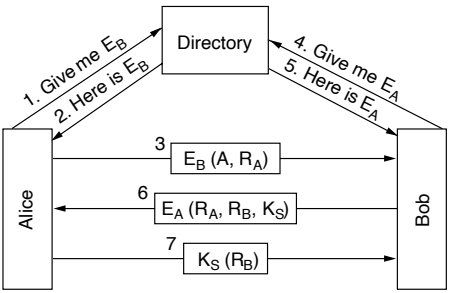

Figure 8-40. Mutual authentication using public-key cryptography.

When Bob receives this message, he has no idea whether it came from Alice or from Trudy, but he plays along and asks the directory server for Alice's public key (message 4), which he soon gets (message 5). He then sends Alice message 6, containing Alice's R_A, his own nonce, R_B, and a proposed session key, K_S.

When Alice gets message 6, she decrypts it using her private key. She sees R_A in it, which gives her a warm feeling inside. The message must have come from Bob, since Trudy has no way of determining R_A. Furthermore, it must be fresh and not a replay, since she just sent Bob R_A. Alice agrees to the session by sending back message 7. When Bob sees R_B encrypted with the session key he just generated, he knows Alice got message 6 and verified R_A. Bob is now happy.

What can Trudy do to try to subvert this protocol? She can fabricate message 3 and trick Bob into probing Alice, but Alice will see an R_A that she did not send and will not proceed further. Trudy cannot forge message 7 back to Bob because she does not know R_B or K_S and cannot determine them without Alice's private key. She is out of luck.

8.10 COMMUNICATION SECURITY

We have now finished our study of the tools of the trade. Most of the important techniques and protocols have been covered. The rest of the chapter is about how these techniques are applied in practice to provide network security, plus some thoughts about the social aspects of security at the end of the chapter.

In the following sections, we will look at communication security, that is, how to get the bits secretly and without modification from source to destination and how to keep unwanted bits outside the door. These are by no means the only security issues in networking, but they are certainly among the most important ones.

8.10.1 IPsec

IETF has known for years that security was lacking in the Internet. Adding it was not easy because a war broke out about where to put it. Most security experts believe that to be really secure, encryption and integrity checks have to be end to end (i.e., in the application layer). That is, the source process encrypts and/or integrity protects the data and sends them to the destination process where they are decrypted and/or verified. Any tampering done in between these two processes, including within either operating system, can then be detected. The trouble with this approach is that it requires changing all the applications to make them security aware. In this view, the next best approach is putting encryption in the transport layer or in a new layer between the application layer and the transport layer, making it still end to end but not requiring applications to be changed.

The opposite view is that users do not understand security and will not be capable of using it correctly and nobody wants to modify existing programs in any way, so the network layer should authenticate and/or encrypt packets without the users being involved. After years of pitched battles, this view won enough support that a network layer security standard was defined. In part, the argument was that having network layer encryption does not prevent security-aware users from doing it right and it does help security-unaware users to some extent.

The result of this war was a design called **IPsec** (**IP security**), which is described in many RFCs. Not all users want encryption (because it is computationally expensive). Rather than make it optional, it was decided to require encryption all the time but permit the use of a null algorithm. The null algorithm is described and praised for its simplicity, ease of implementation, and great speed in RFC 2410.

The complete IPsec design is a framework for multiple services, algorithms, and granularities. The reason for multiple services is that not everyone wants to pay the price for having all the services all the time, so the services are available a la carte. For example, someone streaming a movie from a remote server might not care about encryption (although the copyright owner might). The major services are secrecy, data integrity, and protection from replay attacks (where the intruder replays a conversation). All of these are based on symmetric-key cryptography because high performance is crucial.

The reason for having multiple algorithms is that an algorithm that is now thought to be secure may be broken in the future. By making IPsec algorithm-independent, the framework can survive even if some particular algorithm is later broken. Switching to algorithm #2 is a lot easier than devising a new famework.

The reason for having multiple granularities is to make it possible to protect a single TCP connection, all traffic between a pair of hosts, or all traffic between a pair of secure routers, among other possibilities.

One slightly surprising aspect of IPsec is that even though it is in the IP layer, it is connection oriented. Actually, that is not so surprising because to have any security, a key must be established and used for some period of time—in essence, a kind of connection by a different name. Also, connections amortize the setup costs over many packets. A ''connection'' in the context of IPsec is called an **SA** (**Security Association**). An SA is a simplex connection between two endpoints and has a security identifier associated with it. If secure traffic is needed in both directions, two security associations are required. Security identifiers are carried in packets traveling on these secure connections and are used to look up keys and other relevant information when a secure packet arrives.

Technically, IPsec has two principal parts. The first part describes two new headers that can be added to packets to carry the security identifier, integrity control data, and other information. The other part, **ISAKMP** (**Internet Security Association and Key Management Protocol**), deals with establishing keys. ISAKMP is a framework. The main protocol for carrying out the work is **IKE** (**Internet Key Exchange**). It has gone through multiple versions as flaws have been corrected.

IPsec can be used in either of two modes. In **transport mode**, the IPsec header is inserted just after the IP header. The *Protocol* field in the IP header is changed to indicate that an IPsec header follows the normal IP header (before the TCP header). The IPsec header contains security information, primarily the SA identifier, a new sequence number, and possibly an integrity check of the payload.

In **tunnel mode**, the entire IP packet, header and all, is encapsulated in the body of a new IP packet with a completely new IP header. Tunnel mode is useful when the tunnel ends at a location other than the final destination. In some cases, the end of the tunnel is a security gateway machine, for example, a company firewall. This is commonly the case for a VPN (Virtual Private Network). In this mode, the security gateway encapsulates and decapsulates packets as they pass through it. By terminating the tunnel at this secure machine, the machines on the company LAN do not have to be aware of IPsec. Only the security gateway has to know about it.

Tunnel mode is also useful when a bundle of TCP connections is aggregated and handled as one encrypted stream because it prevents an intruder from seeing who is sending how many packets to whom. Sometimes just knowing how much traffic is going where is valuable information. For example, if during a military crisis, the amount of traffic flowing between the Pentagon and the White House were to drop sharply, but the amount of traffic between the Pentagon and some military installation deep inside the Colorado Rocky Mountains were to increase by the same amount, an intruder might be able to deduce some useful information from these data. Studying the flow patterns of packets, even if they are encrypted,

is called **traffic analysis**. Tunnel mode provides a way to foil it to some extent. The disadvantage of tunnel mode is that it adds an extra IP header, thus increasing packet size substantially. In contrast, transport mode does not affect packet size as much.

The first new header is **AH** (**Authentication Header**). It provides integrity checking and antireplay security, but not secrecy (i.e., no data encryption). The use of AH in transport mode is illustrated in Fig. 8-41. In IPv4, it is interposed between the IP header (including any options) and the TCP header. In IPv6, it is just another extension header and is treated as such. In fact, the format is close to that of a standard IPv6 extension header. The payload may have to be padded out to some particular length for the authentication algorithm, as shown.

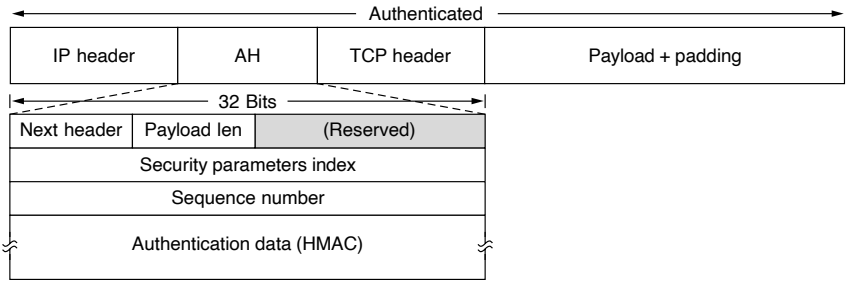

Figure 8-41. The IPsec authentication header in transport mode for IPv4.

Let us now examine the AH header. The *Next header* field is used to store the value that the IP *Protocol* field had before it was replaced with 51 to indicate that an AH header follows. In most cases, the code for TCP (6) will go here. The *Payload length* is the number of 32-bit words in the AH header minus 2.

The *Security parameters index* is the connection identifier. It is inserted by the sender to indicate a particular record in the receiver's database. This record contains the shared key used on this connection and other information about the connection. If this protocol had been invented by ITU rather than IETF, this field would have been called *Virtual circuit number*.

The *Sequence number* field is used to number all the packets sent on an SA. Every packet gets a unique number, even retransmissions. In other words, the retransmission of a packet gets a different number here than the original (even though its TCP sequence number is the same). The purpose of this field is to detect replay attacks. These sequence numbers may not wrap around. If all 2^{32} are exhausted, a new SA must be established to continue communication.

Finally, we come to *Authentication data*, which is a variable-length field that contains the payload's digital signature. When the SA is established, the two sides negotiate which signature algorithm they are going to use. Normally, public-key cryptography is not used here because packets must be processed extremely rapidly

and all known public-key algorithms are too slow. Since IPsec is based on symmetric-key cryptography and the sender and receiver negotiate a shared key before setting up an security association (SA), the shared key is used in the signature computation. In other words, IPsec uses an HMAC, much like the one we discussed in the section about authentication using shared keys. As mentioned, it is much faster to compute than first running SHA-2 and then running RSA on the result.

The AH header does not allow encryption of the data, so it is mostly useful when integrity checking is needed but secrecy is not needed. One noteworthy feature of AH is that the integrity check covers some of the fields in the IP header, namely, those that do not change as the packet moves from router to router. The *Time to live* field changes on each hop, for example, so it cannot be included in the integrity check. However, the IP source address is included in the check, making it impossible for an intruder to falsify the origin of a packet.

The alternative IPsec header is **ESP** (**Encapsulating Security Payload**). Its use for both transport mode and tunnel mode is shown in Fig. 8-42.

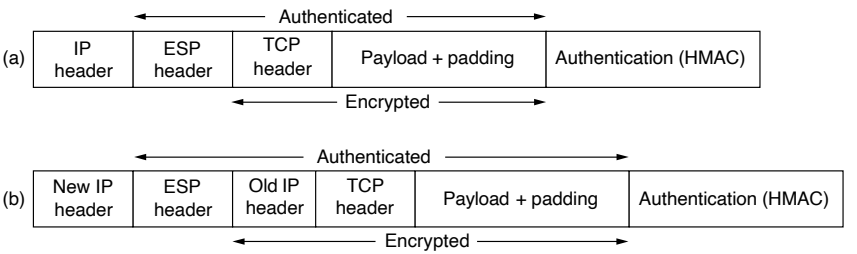

Figure 8-42. (a) ESP in transport mode. (b) ESP in tunnel mode.

The ESP header consists of two 32-bit words. They are the *Security parameters index* and *Sequence number* fields that we saw in AH. A third word that generally follows them (but is technically not part of the header) is the *Initialization vector* used for the data encryption, unless null encryption is used, in which case it is omitted.

ESP also provides for HMAC integrity checks, as does AH, but rather than being included in the header, they come after the payload, as shown in Fig. 8-42. Putting the HMAC at the end has an advantage in a hardware implementation: the HMAC can be calculated as the bits are going out over the network interface and appended to the end. This is why Ethernet and other LANs have their CRCs in a trailer, rather than in a header. With AH, the packet has to be buffered and the signature computed before the packet can be sent, potentially reducing the number of packets/sec that can be sent.

Given that ESP can do everything AH can do and more and is more efficient to boot, the question arises: why bother even having AH at all? The answer is mostly

historical. Originally, AH handled only integrity and ESP handled only secrecy. Later, integrity was added to ESP, but the people who designed AH did not want to let it die after all that work. Their only real argument is that AH checks part of the IP header, which ESP does not, but other than that it is really a weak argument. Another weak argument is that a product supporting AH but not ESP might have less trouble getting an export license because it cannot do encryption. AH is likely to be phased out in the future.

8.10.2 Virtual Private Networks

Many companies have offices and plants scattered over many cities, sometimes over multiple countries. In the olden days, before public data networks, it was common for such companies to lease lines from the telephone company between some or all pairs of locations. Some companies still do this. A network built up from company computers and leased telephone lines is called a **private network**.

Private networks work fine and are very secure. If the only lines available are the leased lines, no traffic can leak out of company locations and intruders have to physically wiretap the lines to break in, which is not easy to do. The problem with private networks is that leasing dedicated lines between two points is very expensive. When public data networks and later the Internet appeared, many companies wanted to move their data (and possibly voice) traffic to the public network, but without giving up the security of the private network.

This demand soon led to the invention of **VPN**s (**Virtual Private Networks**), which are overlay networks on top of public networks but with most of the properties of private networks. They are called ''virtual'' because they are merely an illusion, just as virtual circuits are not real circuits and virtual memory is not real memory.

One popular approach is to build VPNs directly over the Internet. A common design is to equip each office with a firewall and create tunnels through the Internet between all pairs of offices, as illustrated in Fig. 8-43(a). A further advantage of using the Internet for connectivity is that the tunnels can be set up on demand to include, for example, the computer of an employee who is at home or traveling as long as the person has an Internet connection. This flexibility is much greater than with a real private network with leased lines, yet from the perspective of the computers on the VPN, the topology looks just like it, as shown in Fig. 8-43(b). When the system is brought up, each pair of firewalls has to negotiate the parameters of its SA, including the services, modes, algorithms, and keys. If IPsec is used for the tunneling, it is possible to aggregate all traffic between any two pairs of offices onto a single authenticated, encrypted SA, thus providing integrity control, secrecy, and even considerable immunity to traffic analysis. Many firewalls have VPN capabilities built in. Some ordinary routers can do this as well, but since firewalls are primarily in the security business, it is natural to have the tunnels begin and end at the firewalls, providing a clear separation between the company and the Internet.

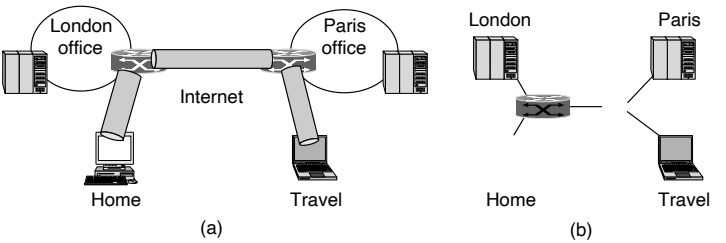

Figure 8-43. (a) A virtual private network. (b) Topology as seen from the inside.

Thus, firewalls, VPNs, and IPsec with ESP in tunnel mode are a natural combination and widely used in practice.

Once the SAs have been established, traffic can begin flowing. To a router within the Internet, a packet traveling along a VPN tunnel is just an ordinary packet. The only thing unusual about it is the presence of the IPsec header after the IP header, but since these extra headers have no effect on the forwarding process, the routers do not care about this extra header.

Another approach that is gaining popularity is to have the ISP set up the VPN. Using MPLS (as discussed in Chap. 5), paths for the VPN traffic can be set up across the ISP network between the company offices. These paths keep the VPN traffic separate from other Internet traffic and can be guaranteed a certain amount of bandwidth or other quality of service.

A key advantage of a VPN is that it is completely transparent to all user software. The firewalls set up and manage the SAs. The only person who is even aware of this setup is the system administrator who has to configure and manage the security gateways, or the ISP administrator who has to configure the MPLS paths. To everyone else, it is like having a leased-line private network again. For more about VPNs, see Ashraf (2018).

8.10.3 Wireless Security

It is surprisingly easy to design a system using VPNs and firewalls that is logically completely secure but that, in practice, leaks like a sieve. This situation can occur if some of the machines are wireless and use radio communication, which passes right over the firewall in both directions. The range of 802.11 networks can be up to 100 meters, so anyone who wants to spy on a company can simply drive into the employee parking lot in the morning, leave an 802.11-enabled notebook computer in the car to record everything it hears, and take off for the day. By late afternoon, the disk will be full of nice goodies. Theoretically, this leakage is not supposed to happen. Theoretically, people are not supposed to rob banks, either.

Much of the security problem can be traced to the manufacturers of wireless base stations (access points) trying to make their products user friendly. Usually, if the user takes the device out of the box and plugs it into the electrical power socket, it begins operating immediately—nearly always with no security at all, blurting secrets to everyone within radio range. If it is then plugged into an Ethernet, all the Ethernet traffic suddenly appears in the parking lot as well. Wireless is a snooper's dream come true: free data without having to do any work. It therefore goes without saying that security is even more important for wireless systems than for wired ones. In this section, we will look at some ways wireless networks handle security with a focus on WiFi (802.11). Some additional information is given by Osterhage (2018).

Part of the 802.11 standard, originally called **802.11i**, prescribes a data link-level security protocol for preventing a wireless node from reading or interfering with messages sent between a pair of wireless nodes. It also goes by the trade name **WPA2** (**WiFi Protected Access 2**). Plain WPA is an interim scheme that implements a subset of 802.11i. It should be avoided in favor of WPA2. The successor to WPA2, brilliantly called **WPA3**, was announced in January 2018 and uses 128-bit encryption in ''personal mode'' and 192-bit encryption in ''Enterprise mode.'' WPA3 has many improvements over WPA2, chief among which perhaps was known as ''Dragonfly,'' an overhauled handshake to thwart certain types of password guessing attacks that plague WPA2. At the time of writing, WPA3 is not yet as widely deployed as WPA2. Also, in April 2019 researchers disclosed an attack vector known as Dragonblood that removes many of WPA3's security advantages. For these reasons, we focus on WPA2 in this section.

We will describe 802.11i shortly, but will first note that it is a replacement for **WEP** (**Wired Equivalent Privacy**), the first generation of 802.11 security protocols. WEP was designed by a networking standards committee, which is a completely different process than, for example, the way NIST selected the design of AES using a worldwide public bake-off. The results were devastating. What was wrong with it? Pretty much everything from a security perspective as it turns out. For example, WEP encrypted data for confidentiality by XORing it with the output of a stream cipher. Unfortunately, weak keying arrangements meant that the output was often reused. This led to trivial ways to defeat it. As another example, the integrity check was based on a 32-bit CRC. That is an efficient code for detecting transmission errors, but it is not a cryptographically strong mechanism for defeating attackers.

These and other design flaws made WEP very easy to compromise. The first practical demonstration that WEP was broken came when Adam Stubblefield was an intern at AT&T (Stubblefield et al., 2002). He was able to code up and test an attack outlined by Fluhrer et al. (2001) in one week, of which most of the time was spent convincing management to buy him a WiFi card to use in his experiments. Software to crack WEP passwords within a minute is now freely available and the use of WEP is very strongly discouraged. While it does prevent casual access it

does not provide any real form of security. The 802.11i group was put together in a hurry when it was clear that WEP was seriously broken. It produced a formal standard by June 2004.

Now we will describe 802.11i, which does provide real security if it is set up and used properly. There are two common scenarios in which WPA2 is used. The first is a corporate setting, in which a company has a separate authentication server that has a username and password database that can be used to determine if a wireless client is allowed to access the network. In this setting, clients use standard protocols to authenticate themselves to the network. The main standards are **802.1X**, with which the access point lets the client carry on a dialogue with the authentication server and observes the result, and **EAP** (**Extensible Authentication Protocol**) (RFC 3748), which tells how the client and the authentication server interact. Actually, EAP is a framework and other standards define the protocol messages. However, we will not delve into the many details of this exchange because they do not much matter for an overview.

The second scenario is in a typical home setting in which there is no authentication server. Instead, there is a single shared password that is used by clients to access the wireless network. This setup is less complex than having an authentication server, which is why it is used at home and in small businesses, but it is less secure as well. The main difference is that with an authentication server each client gets a key for encrypting traffic that is not known by the other clients. With a single shared password, different keys are derived for each client, but all clients have the same password and can derive each others' keys if they want to.

The keys that are used to encrypt traffic are computed as part of an authentication handshake. The handshake happens right after the client associates with a wireless network and authenticates with an authentication server, if there is one. At the start of the handshake, the client has either the shared network password or its password for the authentication server. This password is used to derive a master key. However, the master key is not used directly to encrypt packets. It is standard cryptographic practice to derive a session key for each period of usage, to change the key for different sessions, and to expose the master key to observation as little as possible. It is this session key that is computed in the handshake.

The session key is computed with the four-packet handshake shown in Fig. 8-44. First, the AP (access point) sends a random number for identification. The client also picks its own nonce. It uses the nonces, its MAC address and that of the AP, and the master key to compute a session key, K_S. The session key is split into portions, each of which is used for different purposes, but we have omitted this detail. Now the client has session keys, but the AP does not. So the client sends its nonce to the AP, and the AP performs the same computation to derive the same session keys. The nonces can be sent in the clear because the keys cannot be derived from them without extra, secret information. The message from the client is protected with an integrity check called a **MIC** (**Message Integrity Check**) based on the session key. The AP can check that the MIC is correct, and so the

message indeed must have come from the client, after it computes the session keys. A MIC is just another name for a message authentication code, as in an HMAC. The term MIC is often used instead for networking protocols because of the potential for confusion with MAC (Medium Access Control) addresses.

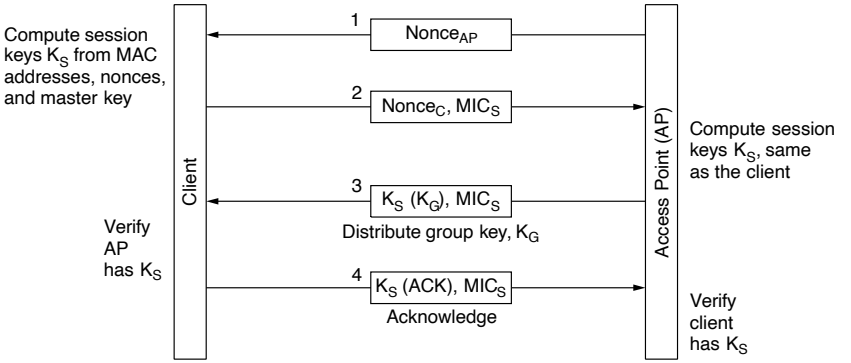

Figure 8-44. The 802.11i key setup handshake.

In the last two messages, the AP distributes a group key, K_G, to the client, and the client acknowledges the message. Receipt of these messages lets the client verify that the AP has the correct session keys, and vice versa. The group key is used for broadcast and multicast traffic on the 802.11 LAN. Because the result of the handshake is that every client has its own encryption keys, none of these keys can be used by the AP to broadcast packets to all of the wireless clients; a separate copy would need to be sent to each client using its key. Instead, a shared key is distributed so that broadcast traffic can be sent only once and received by all the clients. It must be updated as clients leave and join the network.

Finally, we get to the part where the keys are actually used to provide security. Two protocols can be used in 802.11i to provide message confidentiality, integrity, and authentication. Like WPA, one of the protocols, called **TKIP** (**Temporary Key Integrity Protocol**), was an interim solution. It was designed to improve security on old and slow 802.11 cards, so that at least some security that is better than WEP can be rolled out as a firmware upgrade. However, it, too, has now been broken so you are better off with the other, recommended protocol, **CCMP**. What does CCMP stand for? It is short for the somewhat spectacular name Counter mode with Cipher block chaining Message authentication code Protocol. We will just call it CCMP. You can call it anything you want.

CCMP works in a fairly straightforward way. It uses AES encryption with a 128-bit key and block size. The key comes from the session key. To provide

confidentiality, messages are encrypted with AES in counter mode. Recall that we discussed cipher modes in Sec. 8.2.3. These modes are what prevent the same message from being encrypted to the same set of bits each time. Counter mode mixes a counter into the encryption. To provide integrity, the message, including header fields, is encrypted with cipher block chaining mode and the last 128-bit block is kept as the MIC. Then both the message (encrypted with counter mode) and the MIC are sent. The client and the AP can each perform this encryption, or verify this encryption when a wireless packet is received. For broadcast or multicast messages, the same procedure is used with the group key.

8.11 EMAIL SECURITY

When an email message is sent between two distant sites, it will generally transit dozens of machines on the way. Any of these can read and record the message for future use. In practice, privacy is nonexistent, despite what many people think. Nevertheless, many people would like to be able to send email that can be read by the intended recipient and no one else: not their boss and not even their government. This desire has stimulated several people and groups to apply the cryptographic principles we studied earlier to email to produce secure email. In the following sections, we will study a widely used secure email system, PGP, and then briefly mention one other, S/MIME.

8.11.1 Pretty Good Privacy

Our first example, **PGP** (**Pretty Good Privacy**) is essentially the brainchild of one person, Phil Zimmermann (1995). Zimmermann is a privacy advocate whose motto is: ''If privacy is outlawed, only outlaws will have privacy.'' Released in 1991, PGP is a complete email security package that provides privacy, authentication, digital signatures, and compression, all in an easy-to-use form. Furthermore, the complete package, including all the source code, is distributed free of charge via the Internet. Owing to its quality, price (zero), and easy availability on UNIX, Linux, Windows, and Mac OS platforms, it is widely used today.

PGP originally encrypted data by using a block cipher called **IDEA** (**International Data Encryption Algorithm**), which uses 128-bit keys. It was devised in Switzerland at a time when DES was seen as tainted and AES had not yet been invented. Conceptually, IDEA is similar to DES and AES: it mixes up the bits in a series of rounds, but the details of the mixing functions are different from DES and AES. Later, AES was added as an encryption algorithm and this is now commonly used.

PGP has also been embroiled in controversy since day 1 (Levy, 1993). Because Zimmermann did nothing to stop other people from placing PGP on the Internet, where people all over the world could get it, the U.S. Government claimed

that Zimmermann had violated U.S. laws prohibiting the export of munitions. The U.S. Government's investigation of Zimmermann went on for 5 years but was eventually dropped, probably for two reasons. First, Zimmermann did not place PGP on the Internet himself, so his lawyer claimed that *he* never exported anything (and then there is the little matter of whether creating a Web site constitutes export at all). Second, the government eventually came to realize that winning a trial meant convincing a jury that a Web site containing a downloadable privacy program was covered by the arms-trafficking law prohibiting the export of war materiel such as tanks, submarines, military aircraft, and nuclear weapons. Years of negative publicity probably did not help much, either.

As an aside, the export rules are bizarre, to put it mildly. The government considered putting code on a Web site to be an illegal export and harassed and threatened Zimmermann about it for 5 years. On the other hand, when someone published the complete PGP source code, in C, as a book (in a large font with a checksum on each page to make scanning it in easy) and then exported the book, that was fine with the government because books are not classified as munitions. The sword is mightier than the pen, at least for Uncle Sam.

Another problem PGP ran into involved patent infringement. The company holding the RSA patent, RSA Security, Inc., alleged that PGP's use of the RSA algorithm infringed on its patent, but that problem was settled with releases starting at 2.6. Furthermore, PGP used another patented encryption algorithm, IDEA, whose use caused some problems at first.

Since PGP is open source and freely available, various people and groups have modified it and produced a number of versions. Some of these were designed to get around the munitions laws, others were focused on avoiding the use of patented algorithms, and still others wanted to turn it into a closed-source commercial product. Although the munitions laws have now been slightly liberalized (otherwise, products using AES would not have been exportable from the U.S.), and the RSA patent expired in September 2000, the legacy of all these problems is that several incompatible versions of PGP are in circulation, under various names. The discussion below focuses on classic PGP, which is the oldest and simplest version, except that we use AES and SHA-2 instead of IDEA and MD5 in our explanation. Another popular version, Open PGP, is described in RFC 2440. Yet another is the GNU Privacy Guard.

PGP intentionally uses existing cryptographic algorithms rather than inventing new ones. It is largely based on algorithms that have withstood extensive peer review and were not designed or influenced by any government agency trying to weaken them. For people who distrust government, this property is a big plus.

PGP supports text compression, secrecy, and digital signatures and also provides extensive key management facilities, but, oddly enough, not email facilities. It is like a preprocessor that takes plaintext as input and produces signed ciphertext in base64 as output. This output can then be emailed, of course. Some PGP implementations call a user agent as the final step to actually send the message.

To see how PGP works, let us consider the example of Fig. 8-45. Here, Alice wants to send a signed plaintext message, P, to Bob in a secure way. PGP supports different encryption schemes such as RSA and elliptic curve cryptography, but here we assume that both Alice and Bob have private (D_X) and public (E_X) RSA keys. Let us also assume that each one knows the other's public key; we will cover PGP key management shortly.

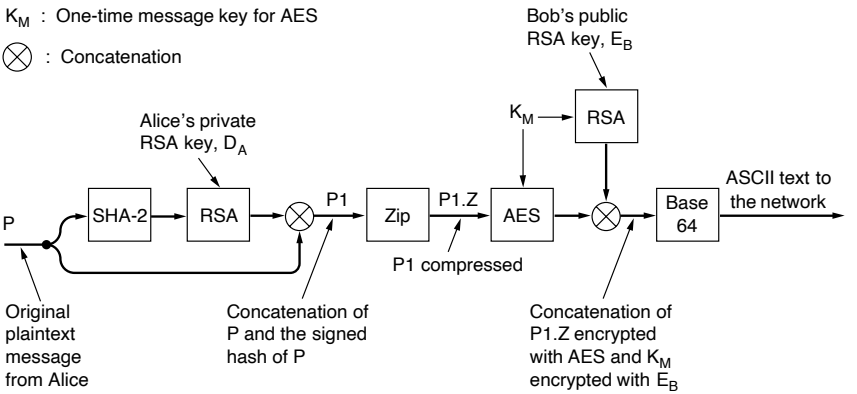

Figure 8-45. PGP in operation for sending a message.

Alice starts out by invoking the PGP program on her computer. PGP first hashes her message, P, using SHA-2, and then encrypts the resulting hash using her private RSA key, D_A. When Bob eventually gets the message, he can decrypt the hash with Alice's public key and verify that the hash is correct. Even if someone else (e.g., Trudy) could acquire the hash at this stage and decrypt it with Alice's known public key, the strength of SHA-2 guarantees that it would be computationally infeasible to produce another message with the same SHA-2 hash.

The encrypted hash and the original message are now concatenated into a single message, *P1*, and then compressed using the ZIP program, which uses the Ziv-Lempel algorithm (Ziv and Lempel, 1977). Call the output of this step *P1.Z*.

Next, PGP prompts Alice for some random input. Both the content and the typing speed are used to generate a 256-bit AES message key, K_M (called a session key in the PGP literature, but this is really a misnomer since there is no session). K_M is now used to encrypt *P1.Z* with AES. In addition, K_M is encrypted with Bob's public key, E_B. These two components are then concatenated and converted to base64, as we discussed in the section on MIME in Chap. 7. The resulting message contains only letters, digits, and the symbols +, /, and =, which means it can be put into an RFC 822 body and be expected to arrive unmodified.

When Bob gets the message, he reverses the base64 encoding and decrypts the AES key using his private RSA key. Using this key, he decrypts the message to get *P1.Z*. After decompressing it, Bob separates the plaintext from the encrypted hash

and decrypts the hash using Alice's public key. If the plaintext hash agrees with his own SHA-2 computation, he knows that *P* is the correct message and that it came from Alice.

It is worth noting that RSA is only used in two places here: to encrypt the 256-bit SHA-2 hash and to encrypt the 256-bit key. Although RSA is slow, it has to encrypt only a handful of bits, not a large volume of data. Furthermore, all 512 plaintext bits are exceedingly random, so a considerable amount of work will be required on Trudy's part just to determine if a guessed key is correct. The heavy-duty encryption is done by AES, which is orders of magnitude faster than RSA. Thus, PGP provides security, compression, and a digital signature and does so in a much more efficient way than the scheme illustrated in Fig. 8-22.

PGP supports multiple RSA key lengths. It is up to the user to select the one that is most appropriate. For instance, if you are a regular user, a key length of 1024 bits may already be sufficient. If you are worried about sophisticated government-funded three-letter organizations, perhaps 2048 bits should be the minimum. Worried about aliens whose technology is 10,000 years ahead of ours reading your emails? There is always the option to use 4096 bit keys. On the other hand, since RSA is only used for encrypting a few bits, perhaps you should always go for alien-proof.

The format of a classic PGP message is shown in Fig. 8-46. Numerous other formats are also in use. The message has three parts, containing the IDEA key, the signature, and the message, respectively. The key part contains not only the key, but also a key identifier, since users are permitted to have multiple public keys.

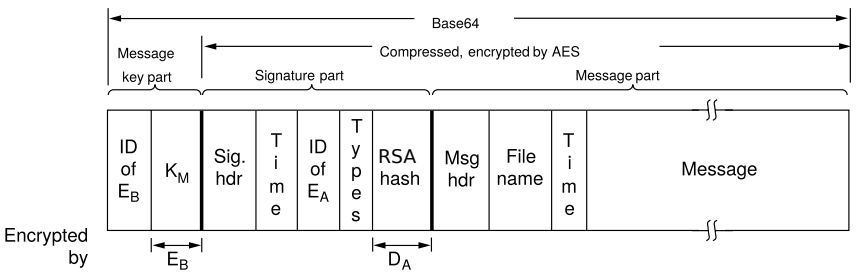

Figure 8-46. A PGP message.

The signature part contains a header, which will not concern us here. The header is followed by a timestamp, the identifier for the sender's public key that can be used to decrypt the signature hash, some type information that identifies the algorithms used (to allow SHA-4 and RSA2 to be used when they are invented), and the encrypted hash itself.

The message part also contains a header, the default name of the file to be used if the receiver writes the file to the disk, a message creation timestamp, and, finally (not surprisingly), the message itself.

Key management has received a large amount of attention in PGP as it is the Achilles' heel of all security systems. Key management works as follows. Each user maintains two data structures locally: a private key ring and a public key ring. The **private key ring** contains one or more personal private/public-key pairs. The reason for supporting multiple pairs per user is to permit users to change their public keys periodically or when one is thought to have been compromised, without invalidating messages currently in preparation or in transit. Each pair has an identifier associated with it so that a message sender can tell the recipient which public key was used to encrypt it. Message identifiers consist of the low-order 64 bits of the public key. Users are themselves responsible for avoiding conflicts in their public-key identifiers. The private keys on disk are encrypted using a special (arbitrarily long) password to protect them against sneak attacks.

The **public key ring** contains public keys of the user's correspondents. These are needed to encrypt the message keys associated with each message. Each entry on the public key ring contains not only the public key, but also its 64-bit identifier and an indication of how strongly the user trusts the key.

The problem being tackled here is the following. Suppose that public keys are maintained on Web sites. One way for Trudy to read Bob's secret email is to attack the Web site and replace Bob's public key with one of her choice. When Alice later fetches the key allegedly belonging to Bob, Trudy can mount a bucket brigade (MITM)) attack on Bob.

To prevent such attacks, or at least minimize the consequences of them, Alice needs to know how much to trust the item called ''Bob's key'' on her public key ring. If she knows that Bob personally handed her a CD-ROM (or a more modern storage device) containing the key, she can set the trust value to the highest value. It is this decentralized, user-controlled approach to public-key management that sets PGP apart from centralized PKI schemes.

Nevertheless, people do sometimes obtain public keys by querying a trusted key server. For this reason, after X.509 was standardized, PGP supported these certificates as well as the traditional PGP public key ring mechanism. All current versions of PGP have X.509 support.

8.11.2 S/MIME

IETF's venture into email security, called **S/MIME** (**Secure/MIME**), is described in RFC 2632 through RFC 2643. It provides authentication, data integrity, secrecy, and nonrepudiation. It also is quite flexible, supporting a variety of cryptographic algorithms. Not surprisingly, given the name, S/MIME integrates well with MIME, allowing all kinds of messages to be protected. A variety of new MIME headers are defined, for example, for holding digital signatures.

S/MIME does not have a rigid certificate hierarchy beginning at a single root, which had been one of the political problems that doomed an earlier system called PEM (Privacy Enhanced Mail). Instead, users can have multiple trust anchors. As

long as a certificate can be traced back to some trust anchor the user believes in, it is considered valid. S/MIME uses the standard algorithms and protocols we have been examining so far, so we will not discuss it any further here. For the details, please consult the RFCs.

8.12 WEB SECURITY

We have just studied two important areas where security is needed: communications and email. You can think of these as the soup and appetizer. Now it is time for the main course: Web security. The Web is where most of the Trudies hang out nowadays and do their dirty work. In the following sections, we will look at some of the problems and issues relating to Web security.

Web security can be roughly divided into three parts. First, how are objects and resources named securely? Second, how can secure, authenticated connections be established? Third, what happens when a Web site sends a client a piece of executable code? After looking at some threats, we will examine all these issues.

8.12.1 Threats

One reads about Web site security problems in the newspaper almost weekly. The situation is really pretty grim. Let us look at a few examples of what has already happened. First, the home pages of numerous organizations have been attacked and replaced by new home pages of the crackers' choosing. (The popular press calls people who break into computers ''hackers,'' but many programmers reserve that term for great programmers. We prefer to call these people **crackers**. Sites that have been cracked include those belonging to Yahoo!, the U.S. Army, Equifax, the CIA, NASA, and the *New York Times*. In most cases, the crackers just put up some funny text and the sites were repaired within a few hours.

Now let us look at some much more serious cases. Numerous sites have been brought down by denial-of-service attacks, in which the cracker floods the site with traffic, rendering it unable to respond to legitimate queries. Often, the attack is mounted from a large number of machines that the cracker has already broken into (DDoS attacks). These attacks are so common that they do not even make the news any more, but they can cost the attacked sites millions of dollars in lost business.

In 1999, a Swedish cracker broke into Microsoft's Hotmail Web site and created a mirror site that allowed anyone to type in the name of a Hotmail user and then read all of the person's current and archived email.

In another case, a 19-year-old Russian cracker named Maxim broke into an e-commerce Web site and stole 300,000 credit card numbers. Then he approached the site owners and told them that if they did not pay him $100,000, he would post all the credit card numbers to the Internet. They did not give in to his blackmail,

and he indeed posted the credit card numbers, inflicting great damage on many innocent victims.

In a different vein, a 23-year-old California student emailed a press release to a news agency falsely stating that the Emulex Corporation was going to post a large quarterly loss and that the CEO was resigning immediately. Within hours, the company's stock dropped by 60%, causing stockholders to lose over $2 billion. The perpetrator made a quarter of a million dollars by selling the stock short just before sending the announcement. While this event was not a Web site break-in, it is clear that putting such an announcement on the home page of any big corporation would have a similar effect.

We could (unfortunately) go on like this for many more pages. But it is now time to examine some of the technical issues related to Web security. For more information about security problems of all kinds, see Du (2019), Schneier (2004), and Stuttard and Pinto (2007). Searching the Internet will also turn up vast numbers of specific cases.

8.12.2 Secure Naming and DNSSEC

Let us revisit the problem of DNS spoofing and start with something very basic: Alice wants to visit Bob's Web site. She types Bob's URL into her browser and a few seconds later, a Web page appears. But is it Bob's? Maybe yes and maybe no. Trudy might be up to her old tricks again. For example, she might be intercepting all of Alice's outgoing packets and examining them. When she captures an HTTP *GET* request headed to Bob's Web site, she could go to Bob's Web site herself to get the page, modify it as she wishes, and return the fake page to Alice. Alice would be none the wiser. Worse yet, Trudy could slash the prices at Bob's e-store to make his goods look very attractive, thereby tricking Alice into sending her credit card number to ''Bob'' to buy some merchandise.

One disadvantage of this classic man-in-the-middle attack is that Trudy has to be in a position to intercept Alice's outgoing traffic and forge her incoming traffic. In practice, she has to tap either Alice's phone line or Bob's, since tapping the fiber backbone is fairly difficult. While active wiretapping is certainly possible, it is a fair amount of work, and while Trudy is clever, she is also lazy.

Besides, there are easier ways to trick Alice, such as DNS spoofing, which we encountered previously in Sec. 8.2.3. Briefly, attackers use DNS spoofing to store an incorrect mapping of a service in an intermediate name server, making it point to the attacker's IP address. When a user wants to communicate with the service, it looks up the address, but rather than talking to the legitimate server, ends up talking to the attacker.

The real problem is that DNS was designed at a time when the Internet was a research facility for a few hundred universities, and neither Alice, nor Bob, nor Trudy was invited to the party. Security was not an issue then; making the Internet work at all was the issue. The environment has changed radically over the years,

so in 1994 IETF set up a working group to make DNS fundamentally secure. This (ongoing) project is known as **DNSSEC** (**DNS security**); its first output was presented in RFC 2535 and later updated in RFC 4033, RFC 4034, and RFC 4035 among others. Unfortunately, DNSSEC has not been fully deployed yet, so numerous DNS servers are still vulnerable to spoofing attacks.

DNSSEC is conceptually extremely simple. It is based on public-key cryptography. Every DNS zone (as discussed in Chap. 7) has a public/private key pair. All information sent by a DNS server is signed with the originating zone's private key, so the receiver can verify its authenticity.

DNSSEC offers three fundamental services:

1. Proof of where the data originated.
2. Public key distribution.
3. Transaction and request authentication.

The main service is the first one, which verifies that the data being returned has been approved by the zone's owner. The second one is useful for storing and retrieving public keys securely. The third one is needed to guard against playback and spoofing attacks. Note that secrecy is not an offered service since all the information in DNS is considered public. Since phasing in DNSSEC was expected to take several years, the ability for security-aware servers to interwork with security-ignorant servers was essential, which implied that the protocol could not be changed. Let us now look at some of the details.

DNS records are grouped into sets called **RRSETs** (**Resource Record SETs**), with all the records having the same name, class, and type being lumped together in a set. An RRSET may contain multiple *A* records, for example, if a DNS name resolves to a primary IP address and a secondary IP address. The RRSETs are extended with several new record types (discussed below). Each RRSET is cryptographically hashed (e.g., using SHA-2). The hash is signed by the zone's private key (e.g., using RSA). The unit of transmission to clients is the signed RRSET. Upon receipt of a signed RRSET, the client can verify whether it was signed by the private key of the originating zone. If the signature agrees, the data are accepted. Since each RRSET contains its own signature, RRSETs can be cached anywhere, even at untrustworthy servers, without endangering the security.

DNSSEC introduces several new record types. The first of these is the *DNSKEY* record. This record holds the public key of a zone, user, host, or other principal, the cryptographic algorithm used for signing, the protocol used for transmission, and a few other bits. The public key is stored naked. X.509 certificates are not used due to their bulk. The algorithm field holds a 1 for MD5/RSA signatures and other values for other combinations. The protocol field can indicate the use of IPsec or other security protocols, if any.

The second new record type is the *RRSIG* record. It holds the signed hash according to the algorithm specified in the *DNSKEY* record. The signature applies

to all the records in the RRSET, including any *DNSKEY* records present, but excluding itself. It also holds the times when the signature begins its period of validity and when it expires, as well as the signer's name and a few other items.

The DNSSEC design is such that a zone's private key can be kept offline to protect it. Once or twice a day, the contents of a zone's database can be manually transported (e.g., on a secure storage device such as the old, but fairly trustworthy CD-ROM) to a disconnected machine on which the private key is located. All the RRSETs can be signed there and the *RRSIG* records thus produced can be conveyed back to the zone's primary server on a secure device. In this way, the private key can be stored on a storage device locked in a safe except when it is inserted into the disconnected machine for signing the day's new RRSETs. After signing is completed, all copies of the key are erased from memory and the disk and the storage devices are returned to the safe. This procedure reduces electronic security to physical security, something people understand how to deal with.

This method of presigning RRSETs greatly speeds up the process of answering queries since no cryptography has to be done on the fly. The trade-off is that a large amount of disk space is needed to store all the keys and signatures in the DNS databases. Some records will increase tenfold in size due to the signature.

When a client process gets a signed RRSET, it must apply the originating zone's public key to decrypt the hash, compute the hash itself, and compare the two values. If they agree, the data are considered valid. However, this procedure begs the question of how the client gets the zone's public key. One way is to acquire it from a trusted server, using a secure connection (e.g., using IPsec).

However, in practice, it is expected that clients will be preconfigured with the public keys of all the top-level domains. If Alice now wants to visit Bob's Web site, she can ask DNS for the RRSET of *bob.com*, which will contain his IP address and a *DNSKEY* record containing Bob's public key. This RRSET will be signed by the top-level *com* domain, so Alice can easily verify its validity. An example of what this RRSET might contain is shown in Fig. 8-47.

Domain name	Time to live	Class	Type	Value
bob.com.	86400	IN	A	36.1.2.3
bob.com.	86400	IN	DNSKEY	3682793A7B73F731029CE2737D...
bob.com.	86400	IN	RRSIG	86947503A8B848F5272E53930C...

Figure 8-47. An examle RRSET for *bob.com*. The *DNSKEY* record is Bob's public key. The *RRSIG* record is the top-level *com* server's signed hash of the *A* and *DNSKEY* records to verify their authenticity.

Now armed with a verified copy of Bob's public key, Alice can ask Bob's DNS server (run by Bob) for the IP address of *www.bob.com*. This RRSET will be signed by Bob's private key, so Alice can verify the signature on the RRSET Bob returns. If Trudy somehow or other manages to inject a false RRSET into any of

the caches, Alice can easily detect its lack of authenticity because the *RRSIG* record contained in it will be incorrect.

However, DNSSEC also provides a cryptographic mechanism to bind a response to a specific query, to prevent the kind of spoofing attack we discussed at the start of this chapter. This (optional) antispoofing measure adds to the response a hash of the query message signed with the respondent's private key. Since Trudy does not know the private key of the top-level *com* server, she cannot forge a response to a query Alice's ISP sent there. She can certainly get her response back first, but it will be rejected due to its invalid signature over the hashed query.

DNSSEC also supports a few other record types. For example, the *CERT* record can be used for storing (e.g., X.509) certificates. This record has been provided because some people want to turn DNS into a PKI. Whether this will actually happen remains to be seen. We will stop our discussion of DNSSEC here. For more details, please consult the RFCs.

8.12.3 Transport Layer Security

Secure naming is a good start, but there is much more to Web security. The next step is secure connections. We will now look at how secure connections can be achieved. Nothing involving security is simple and this is not either.

When the Web burst into public view, it was initially used for just distributing static pages. However, before long, some companies got the idea of using it for financial transactions, such as purchasing merchandise by credit card, online banking, and electronic stock trading. These applications created a demand for secure connections. In 1995, Netscape Communications Corp., the then-dominant browser vendor, responded by introducing a security package called **SSL** (**Secure Sockets Layer**) now called **TLS** (**Transport Layer Security**) to meet this demand. This software and its protocol are now widely used, for example, by Firefox, Brave, Safari, and Chrome, so it is worth examining in some detail.

SSL builds a secure connection between two sockets, including

1. Parameter negotiation between client and server.
2. Authentication of the server by the client.
3. Secret communication.
4. Data integrity protection.

We have seen these items before, so there is no need to elaborate on them here.

The positioning of SSL in the usual protocol stack is illustrated in Fig. 8-48. Effectively, it is a new layer interposed between the application layer and the transport layer, accepting requests from the browser and sending them down to TCP for transmission to the server. Once the secure connection has been established, SSL's main job is handling compression and encryption. When HTTP is used over

SSL, it is called **HTTPS** (**Secure HTTP**), even though it is the standard HTTP protocol. Sometimes it is available at a new port (443) instead of port 80. As an aside, SSL is not restricted to Web browsers, but that is its most common application. It can also provide mutual authentication.

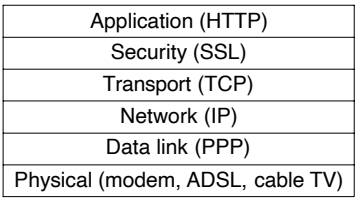

Figure 8-48. Layers (and protocols) for a home user browsing with SSL.

The SSL protocol has gone through several versions. Below we will discuss only version 3, which is the most widely used version. SSL supports a variety of different options. These options include the presence or absence of compression, the cryptographic algorithms to be used, and some matters relating to export restrictions on cryptography. The last is mainly intended to make sure that serious cryptography is used only when both ends of the connection are in the United States. In other cases, keys are limited to 40 bits, which cryptographers regard as something of a joke. Netscape was forced to put in this restriction in order to get an export license from the U.S. Government.

SSL consists of two subprotocols, one for establishing a secure connection and one for using it. Let us start out by seeing how secure connections are established. The connection establishment subprotocol is shown in Fig. 8-49. It starts out with message 1 when Alice sends a request to Bob to establish a connection. The request specifies the SSL version Alice has and her preferences with respect to compression and cryptographic algorithms. It also contains a nonce, R_A, to be used later.

Now it is Bob's turn. In message 2, Bob makes a choice among the various algorithms that Alice can support and sends his own nonce, R_B. Then, in message 3, he sends a certificate containing his public key. If this certificate is not signed by some well-known authority, he also sends a chain of certificates that can be followed back to one. All browsers, including Alice's, come preloaded with about 100 public keys, so if Bob can establish a chain anchored to one of these, Alice will be able to verify Bob's public key. At this point, Bob may send some other messages (such as a request for Alice's public-key certificate). When Bob is done, he sends message 4 to tell Alice it is her turn.

Alice responds by choosing a random 384-bit **premaster key** and sending it to Bob encrypted with his public key (message 5). The actual session key used for encrypting data is derived from the premaster key combined with both nonces in a complex way. After message 5 has been received, both Alice and Bob are able to

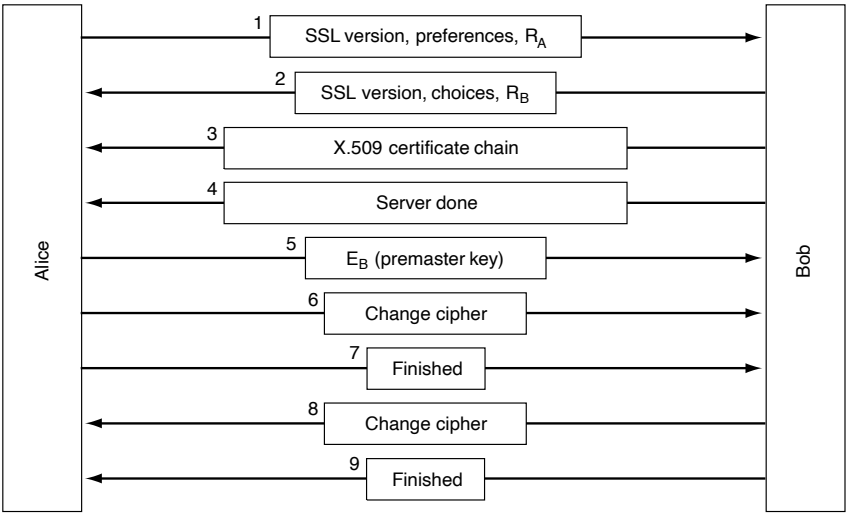

Figure 8-49. A simplified version of the SSL connection establishment subprotocol.

compute the session key. For this reason, Alice tells Bob to switch to the new cipher (message 6) and also that she is finished with the establishment subprotocol (message 7). Bob then acknowledges her (messages 8 and 9).

However, although Alice knows who Bob is, Bob does not know who Alice is (unless Alice has a public key and a corresponding certificate for it, an unlikely situation for an individual). Therefore, Bob's first message may well be a request for Alice to log in using a previously established login name and password. The login protocol, however, is outside the scope of SSL. Once it has been accomplished, by whatever means, data transport can begin.

As mentioned above, SSL supports multiple cryptographic algorithms. One of them uses triple DES with three separate keys for encryption and SHA for message integrity. This combination is relatively slow, so it was mostly used for banking and other applications in which good security is a must. For ordinary e-commerce applications, RC4 was often used with a 128-bit key for encryption and MD5 is used for message authentication. RC4 takes the 128-bit key as a seed and expands it to a much larger number for internal use. Then it uses this internal number to generate a keystream. The keystream is XORed with the plaintext to provide a classical stream cipher, as we saw in Fig. 8-18. The export versions also used RC4 with 128-bit keys, but 88 of the bits are made public to make the cipher easy to break.

For actual transport, a second subprotocol is used, as shown in Fig. 8-50. Messages from the browser are first broken into units of up to 16 KB. When data

compression is enabled, each unit is then separately compressed. After that, a secret key derived from the two nonces and premaster key is concatenated with the compressed text and the result is hashed with the agreed-on hashing algorithm (usually MD5). This hash is appended to each fragment as the MAC. The compressed fragment plus MAC is then encrypted with the agreed-on symmetric encryption algorithm (usually by XORing it with the RC4 keystream). Finally, a fragment header is attached and the fragment is transmitted over the TCP connection the usual way.

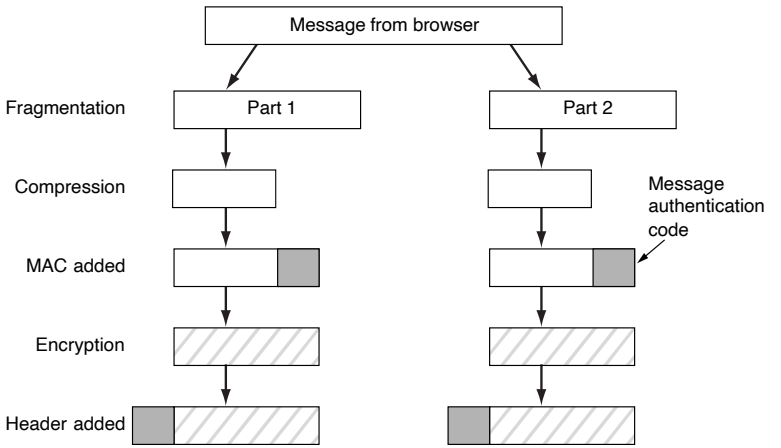

Figure 8-50. Data transmission using SSL.

A word of caution is in order, however. Since it has been shown that RC4 has some weak keys that can be easily cryptanalyzed, the security of SSL using RC4 has been on shaky ground for some time already (Fluhrer et al., 2001). Browsers that allow the user to choose the cipher suite should be configured to use, say, triple DES with 168-bit keys and SHA-2 all the time, even though this combination is slower than RC4 and MD5. Or, better yet, users should upgrade to browsers that support the successor to SSL that we describe shortly.

A problem with SSL is that Alice & Bob may not have certificates, and even if they do, they do not always verify that the keys being used match them.

In 1996, Netscape Communications Corp. turned SSL over to IETF for standardization. The result was **TLS** (**Transport Layer Security**). It is described in RFC 5246.

TLS was built on SSL version 3. The changes made to SSL were relatively small, but just enough that SSL version 3 and TLS cannot interoperate. For example, the way the session key is derived from the premaster key and nonces was changed to make the key stronger (i.e., harder to cryptanalyze). Because of this

incompatibility, most browsers implement both protocols, with TLS falling back to SSL during negotiation if necessary. This is referred to as SSL/TLS. The first TLS implementation appeared in 1999 with version 1.2 defined in August 2008, and version 1.3 in March 2018. It includes support for stronger cipher suites (notably AES), as well as encryption of the **SNI (Server Name Indication)**, which can be used to identify the Web site the user is visiting if it is transmitted in cleartext.

8.12.4 Running Untrusted Code

Naming and connections are two areas of concern related to Web security. But there are more. One particularly difficult problem is that we more and more allow foreign, untrusted code to run on our local machines. We will now take a quick peek at some of the issues raised by such untrusted code and some approaches to dealing with it.

Scripting Code in the Browser

In the early days, when Web pages were just static HTML files, they did not contain executable code. Now they often contain small programs, typically written in **JavaScript** (and sometimes compiled to the more efficient **Web Assembly**). Downloading and executing such **mobile code** is obviously a massive security risk, so various methods have been devised to minimize it.

JavaScript does not have any formal security model, but it does have a long history of leaky implementations. Each vendor handles security in a different way. The main defense is that, barring bugs, the language should not be able to do very bad things—read or write arbitrary files, access the sensitive data of other Web pages, etc. We commonly say that such code runs in a **sandboxed environment**. The problem is that bugs do exist.

The fundamental problem is that letting foreign code run on your machine is asking for trouble. From a security standpoint, it is like inviting a burglar into your house and then trying to watch him carefully so he cannot escape from the kitchen into the living room. If something unexpected occurs and you are distracted for a moment, bad things can happen. The tension here is that mobile code allows flashy graphics and fast interaction, and many Web site designers think that this is much more important than security, especially when it is somebody else's machine at risk.

For instance, imagine that a Web site containing your personal data allows you to provide feedback in the form of arbitrary text that is visible to every other user. The idea is that users can now tell the company how much they like or hate its services. However, unless that Web site very carefully sanitizes the data in the feedback form, an attacker could also place a small amount of JavaScript in the text field. Now imagine that you visit the Web site and look at the feedback provided by other users. The JavaScript will be sent to your browser which has no idea that

this is supposed to be feedback. It just sees JavaScript, just like it finds on many other Web pages, and starts executing it. The malicious JavaScript is able to steal all the privacy-sensitive data (e.g., cookies) that your browser maintains for this Web site and send it to the criminal. This is known as a **CSS** (**cross-site scripting**) attack. **CSRF** (**Cross-Site Request Forgery**) attacks, which are related, can even allow an attacker to pose as a user.

Another problem that may arise is that the JavaScript engine may not be as secure as it should be. For instance, there may be a bug in the browser that malicious JavaScript code can use to take over the browser, or perhaps even the entire system. This is known as a **drive-by download**: you visit a Web site and without realizing it, you are infected. It does not even mean that the Web site was malicious—perhaps the JavaScript was in an advertisement or in some feedback field, as we saw earlier. A particular famous attack, known as **Operation Aurora** was the attack on Google and several other tech companies, where the attackers used a drive-by download to spread through the company with an eye towards getting access to its code repositories.

Browser Extensions

As well as extending Web pages with code, there is a booming marketplace in **browser extensions**, **add-ons**, and **plug-ins**. They are computer programs that extend the functionality of Web browsers. Plug-ins often provide the capability to interpret or display a certain type of content, such as PDFs or Flash animations. Extensions and add-ons provide new browser features, such as better password management, or ways to interact with pages by, for example, marking them up or enabling easy shopping for related items.

Installing an extension, add-on, or plug-in is as simple as coming across something you want when browsing and following the link to install the program. This action will cause code to be downloaded across the Internet and installed into the browser. All of these programs are written to frameworks that differ depending on the browser that is being enhanced. However, to a first approximation, they become part of the trusted computing base of the browser. That is, if the code that is installed is buggy, the entire browser can be compromised.

There are two other obvious failure modes as well. The first is that the program may behave maliciously, for example, by gathering personal information and sending it to a remote server. For all the browser knows, the user installed the extension for precisely this purpose. The second problem is that plug-ins give the browser the ability to interpret new types of content. Often this content is a full-blown programming language itself. PDF and Flash are good examples. When users view pages with PDF and Flash content, the plug-ins in their browser are executing the PDF and Flash code. That code had better be safe; often there are vulnerabilities that it can exploit. For all of these reasons, add-ons and plug-ins should only be installed as needed and only from trusted vendors.

Trojans and Other Malware

Trojans and malicious software (malware) are another form of untrusted code. Often users install such code without realizing it because they think the code is benign, or because they opened an attachment that led to stealthy code execution, which then installed some additional malicious software. When malicious code starts executing, it usually starts out by infecting other programs (either on disk or running programs in memory). When one of these programs is run, it is running malicious code. It may spread itself to other machines, encrypt all your documents on disk (for ransom), spy on your activities, and many other unpleasant things. Some malware infects the boot sector of the hard disk, so when the machine is booted, the malware gets to run. Malware become a huge problem on the Internet and have caused billions of dollars' worth of damage. There is no obvious solution. Perhaps a whole new generation of operating systems based on secure microkernels and tight compartmentalization of users, processes, and resources might help.

8.13 SOCIAL ISSUES

The Internet and its security technology is an area where social issues, public policy, and technology meet head-on, often with huge consequences. Below we will just briefly examine three areas: privacy, freedom of speech, and copyright. Needless to say, we can only scratch the surface. For additional reading, see Anderson (2008a), Baase and Henry (2017), Bernal (2018), and Schneier (2004). The Internet is also full of material. Just type words such as ''privacy,'' ''censorship,'' and ''copyright'' into any search engine.

8.13.1 Confidential and Anonymous Communication

Do people have a right to privacy? Good question. The Fourth Amendment to the U.S. Constitution prohibits the government from searching people's houses, papers, and effects without good reason, and goes on to restrict the circumstances under which search warrants shall be issued. Thus, privacy has been on the public agenda for over 200 years, at least in the U.S.

What has changed in the past decade is both the ease with which governments can spy on their citizens and the ease with which the citizens can prevent such spying. In the 18th century, for the government to search a citizen's papers, it had to send out a policeman on a horse to go to the citizen's farm demanding to see certain documents. It was a cumbersome procedure. Nowadays, telephone companies and Internet providers readily provide wiretaps when presented with search warrants. It makes life much easier for the policeman and there is no danger of falling off a horse.

The widespread usage of smartphones adds a new dimension to government snooping. Many people carry around a smartphone that contains information about their entire life. Some smartphones can be unlocked using facial recognition software. This has the consequence that if a police officer wants to have a suspect unlock his phone and the suspect refuses, all the officer has to do is hold the phone in front of the suspect's face, and bingo, the phone unlocks. Very few people think about this scenario when enabling face recognition (or its predecessor, fingerprint recognition).

Cryptography changes all that. Anybody who goes to the trouble of downloading and installing PGP and who uses a well-guarded alien-strength key can be fairly sure that nobody in the known universe can read his email, search warrant or no search warrant. Governments well understand this and do not like it. Real privacy means it is much harder for them to spy on criminals of all stripes, but it is also much harder to spy on journalists and political opponents. Consequently, some governments restrict or forbid the use or export of cryptography. In France, for example, prior to 1999, all cryptography was banned unless the government was given the keys.

France was not alone. In April 1993, the U.S. Government announced its intention to make a hardware cryptoprocessor, the **clipper chip**, the standard for all networked communication. It was said that this would guarantee citizens' privacy. It also mentioned that the chip provided the government with the ability to decrypt all traffic via a scheme called **key escrow**, which allowed the government access to all the keys. However, the government promised only to snoop when it had a valid search warrant. Needless to say, a huge furor ensued, with privacy advocates denouncing the whole plan and law enforcement officials praising it. Eventually, the government backed down and dropped the idea.

A large amount of information about electronic privacy is available at the Electronic Frontier Foundation's Web site, *www.eff.org*.

Anonymous Remailers

PGP, SSL, and other technologies make it possible for two parties to establish secure, authenticated communication, free from third-party surveillance and interference. However, sometimes privacy is best served by *not* having authentication, in fact, by making communication anonymous. The anonymity may be desired for point-to-point messages, newsgroups, or both.

Let us consider some examples. First, political dissidents living under authoritarian regimes often wish to communicate anonymously to escape being jailed or killed. Second, wrongdoing in many corporate, educational, governmental, and other organizations has often been exposed by whistleblowers, who frequently prefer to remain anonymous to avoid retribution. Third, people with unpopular social, political, or religious views may wish to communicate with each other via email or

newsgroups without exposing themselves. Fourth, people may wish to discuss alcoholism, mental illness, sexual harassment, child abuse, or being a member of a persecuted minority in a newsgroup without having to go public. Numerous other examples exist, of course.

Let us consider a specific example. In the 1990s, some critics of a nontraditional religious group posted their views to a USENET newsgroup via an **anonymous remailer**. This server allowed users to create pseudonyms and send email to the server, which then remailed or re-posted them using the pseudonyms, so no one could tell where the messages really came from. Some postings revealed what the religious group claimed were trade secrets and copyrighted documents. The religious group responded by telling local authorities that its trade secrets had been disclosed and its copyright infringed, both of which were crimes where the server was located. A court case followed and the server operator was compelled to turn over the mapping information that revealed the true identities of the persons who had made the postings. (Incidentally, this was not the first time that a religious group was unhappy when someone leaked its trade secrets: William Tyndale was burned at the stake in 1536 for translating the Bible into English.)

A substantial segment of the Internet community was completely outraged by this breach of confidentiality. The conclusion that everyone drew is that an anonymous remailer that stores a mapping between real email addresses and pseudonyms (now called a type 1 remailer) is not worth anything at all. This case stimulated various people into designing anonymous remailers that could withstand subpoena attacks.

These new remailers, often called **cypherpunk remailers**, work as follows. The user produces an email message, complete with RFC 822 headers (except *From:*, of course), encrypts it with the remailer's public key, and sends it to the remailer. There the outer RFC 822 headers are stripped off, the content is decrypted and the message is remailed. The remailer has no accounts and maintains no logs, so even if the server is later confiscated, it retains no trace of messages that have passed through it.

Many users who wish anonymity chain their requests through multiple anonymous remailers, as shown in Fig. 8-51. Here, Alice wants to send Bob a really, really, really anonymous Valentine's Day card, so she uses three remailers. She composes the message, M, and puts a header on it containing Bob's email address. Then she encrypts the whole thing with remailer 3's public key, E_3 (indicated by horizontal hatching). To this she prepends a header with remailer 3's email address in plaintext. This is the message shown between remailers 2 and 3 in the figure.

Then she encrypts this message with remailer 2's public key, E_2 (indicated by vertical hatching) and prepends a plaintext header containing remailer 2's email address. This message is shown between 1 and 2 in Fig. 8-51. Finally, she encrypts the entire message with remailer 1's public key, E_1, and prepends a plaintext header with remailer 1's email address. This is the message shown to the right of Alice in the figure and this is the message she actually transmits.

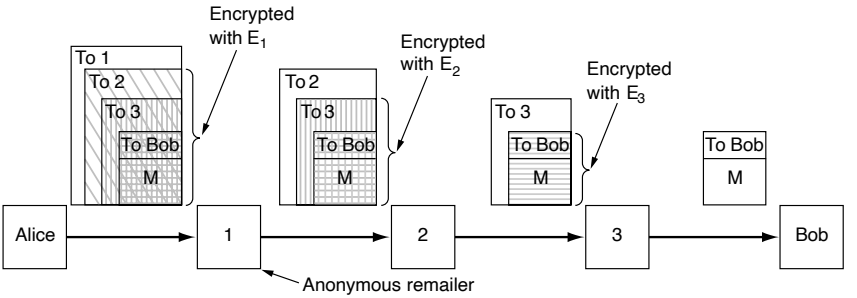

Figure 8-51. How Alice uses three remailers to send Bob a message.

When the message hits remailer 1, the outer header is stripped off. The body is decrypted and then emailed to remailer 2. Similar steps occur at the other two remailers.

Although it is extremely difficult for anyone to trace the final message back to Alice, many remailers take additional safety precautions. For example, they may hold messages for a random time, add or remove junk at the end of a message, and reorder messages, all to make it harder for anyone to tell which message output by a remailer corresponds to which input, in order to thwart traffic analysis. For a description of this kind of remailer, see the classic paper by Mazières and Kaashoek (1998).

Anonymity is not restricted to email. Services also exist that allow anonymous Web surfing using the same form of layered path in which one node only knows the next node in the chain. This method is called **onion routing** because each node peels off another layer of the onion to determine where to forward the packet next. The user configures his browser to use the anonymizer service as a proxy. Tor is a well-known example of such a system (Bernaschi et al., 2019). Henceforth, all HTTP requests go through the anonymizer network, which requests the page and sends it back. The Web site sees an exit node of the anonymizer network as the source of the request, not the user. As long as the anonymizer network refrains from keeping a log, no one can determine who requested which page. also not in the face of a subpoena since the information simply is not there.

8.13.2 Freedom of Speech

Anonymous communication makes it harder for other people to see details about their private communications. A second key social issue is freedom of speech, and its opposite, censorship, which is about governments wanting to restrict what individuals can read and publish. With the Web containing millions

and millions of pages, it has become a censor's paradise. Depending on the nature and ideology of the regime, banned material may include Web sites containing:

1. Material inappropriate for children or teenagers.
2. Hate aimed at various ethnic, religious, sexual, or other groups.
3. Information about democracy and democratic values.
4. Accounts of historical events contradicting the government's version.
5. Manuals for picking locks, building nuclear weapons, encrypting messages, etc.

The usual response is to ban the ''bad'' sites.

Sometimes the results are unexpected. For example, some public libraries have installed Web filters on their computers to make them child friendly by blocking pornography sites. The filters veto sites on their blacklists but also check pages for dirty words before displaying them. In one case in Loudoun County, Virginia, the filter blocked a patron's search for information on breast cancer because the filter saw the word ''breast.'' The library patron sued Loudoun County. However, in Livermore, California, a parent sued the public library for *not* installing a filter after her 12-year-old son was caught viewing pornography there. What's a library to do?

It has escaped many people that the World Wide Web is a *worldwide* Web. It covers the whole world. Not all countries agree on what should be allowed on the Web. For example, in November 2000, a French court ordered Yahoo!, a corporation located in California, to block French users from viewing auctions of Nazi memorabilia on Yahoo!'s Web site because owning such material violates French law. Yahoo! appealed to a U.S. court, which sided with it, but the issue of whose laws apply where is far from settled.

Incidentally, for many years, Yahoo! was one of the darlings of the Internet companies, but nothing lasts forever and in 2017 it was announced that Verizon would buy it for close to 5 billion dollars. The price was reduced with 350 million dollars as a direct result of a series of data breaches at Yahoo! whereby the accounts of billions of users were affected. Security matters.

Going back to the court case, just imagine. What would happen if some court in Utah instructed France to block Web sites dealing with wine because they do not comply with Utah's much stricter laws about alcohol? Suppose that China demanded that all Web sites dealing with democracy be banned as not in the interest of the State. Do Iranian laws on religion apply to more liberal Sweden? Can Saudi Arabia block Web sites dealing with women's rights? The whole issue is a veritable Pandora's box.

A relevant comment from John Gilmore is: ''The net interprets censorship as damage and routes around it.'' For a concrete implementation, consider the **eternity service** (Anderson, 1996). Its goal is to make sure published information

cannot be depublished or rewritten, as was common in the Soviet Union during Josef Stalin's reign. To use the eternity service, the user specifies how long the material is to be preserved, pays a fee proportional to its duration and size, and uploads it. Thereafter, no one can remove or edit it, not even the uploader.

How could such a service be implemented? The simplest model is to use a peer-to-peer system in which stored documents would be placed on dozens of participating servers, each of which gets a fraction of the fee, and thus an incentive to join the system. The servers should be spread over many legal jurisdictions for maximum resilience. Lists of 10 randomly selected servers would be stored securely in multiple places, so that if some were compromised, others would still exist. An authority bent on destroying the document could never be sure it had found all copies. The system could also be made self-repairing in the sense that if it became known that some copies had been destroyed, the remaining sites would attempt to find new repositories to replace them.

The eternity service was the first proposal for a censorship-resistant system. Since then, others have been proposed and, in some cases, implemented. Various new features have been added, such as encryption, anonymity, and fault tolerance. Often the files to be stored are broken up into multiple fragments, with each fragment stored on many servers. Some of these systems are Freenet (Clarke et al., 2002), PASIS (Wylie et al., 2000), and Publius (Waldman et al., 2000).

Of increasing concern is not only the filtering or censorship of information, but also the spread of so-called **disinformation**, or information that is deliberately crafted to be false. Disinformation is now a tactic that attackers can use to sway political, social, and financial outcomes. In 2016, attackers famously authored disinformation sites pertaining to United States presidential candidates and disseminated them on social media. In other contexts, disinformation has been used to attempt to sway real estate prices for investors. Unfortunately, detecting disinformation is challenging, and doing so before it spreads is even more challenging.

Steganography

In countries where censorship abounds, dissidents often try to use technology to evade it. Cryptography allows secret messages to be sent (although possibly not lawfully), but if the government thinks that Alice is a Bad Person, the mere fact that she is communicating with Bob may get him put in this category, too, as repressive governments understand the concept of transitive closure, even if they are short on mathematicians. Anonymous remailers can help, but if they are banned domestically and messages to foreign ones require a government export license, they cannot help much. But the Web can.

People who want to communicate secretly often try to hide the fact that any communication at all is taking place. The science of hiding messages is called **steganography**, from the Greek words for ''covered writing.'' In fact, the ancient Greeks used it themselves. Herodotus wrote of a general who shaved the head of a

messenger, tattooed a message on his scalp, and let the hair grow back before sending him off. Modern techniques are conceptually the same, only they have a higher bandwidth, lower latency, and do not require the services of a barber.

As a case in point, consider Fig. 8-52(a). This photograph, taken by one of the authors (AST) in Kenya, contains three zebras contemplating an acacia tree. Figure 8-52(b) appears to be the same three zebras and acacia tree, but it has an extra added attraction. It contains the complete, unabridged text of five of Shakespeare's plays embedded in it: *Hamlet*, *King Lear*, *Macbeth*, *The Merchant of Venice*, and *Julius Caesar*. Together, these plays total over 700 KB of text.

(a) (b)

Figure 8-52. (a) Three zebras and a tree. (b) Three zebras, a tree, and the complete text of five plays by William Shakespeare.

How does this steganographic channel work? The original color image is 1024 × 768 pixels. Each pixel consists of three 8-bit numbers, one each for the red, green, and blue intensity of that pixel. The pixel's color is formed by the linear superposition of the three colors. The steganographic encoding method uses the low-order bit of each RGB color value as a covert channel. Thus, each pixel has room for 3 bits of secret information, 1 in the red value, 1 in the green value, and 1 in the blue value. With an image of this size, up to 1024 × 768 × 3 bits or 294,912 bytes of secret information can be stored in it.

The full text of the five plays and a short notice add up to 734,891 bytes. This text was first compressed to about 274 KB using a standard compression algorithm. The compressed output was then encrypted using IDEA and inserted into the low-order bits of each color value. As can be seen (or actually, cannot be seen), the existence of the information is completely invisible. It is equally invisible in the large, full-color version of the photo. The eye cannot easily distinguish 21-bit color from 24-bit color.

Viewing the two images in black and white with low resolution does not do justice to how powerful the technique is. To get a better feeling for how steganography works, we have prepared a demonstration, including the full-color high-

resolution image of Fig. 8-52(b) with the five plays embedded in it. The demonstration, including tools for inserting and extracting text into images, can be found at the book's Web site.

To use steganography for undetected communication, dissidents could create a Web site bursting with politically correct pictures, such as photographs of the Great Leader, local sports, movie, and television stars, etc. Of course, the pictures would be riddled with steganographic messages. If the messages were first compressed and then encrypted, even someone who suspected their presence would have immense difficulty in distinguishing the messages from white noise. Of course, the images should be fresh scans; copying a picture from the Internet and changing some of the bits is a dead giveaway. To see how you can embed an audio recording in a still image, see Chaudhary and Chaudbe (2018).

Images are by no means the only carrier for steganographic messages. Audio files also work fine. Hidden information can be carried in a voice-over-IP call by manipulating the packet delays, distorting the audio, or even in the header fields of packets (Lubacz et al., 2010). Even the layout and ordering of tags in an HTML file can carry information.

Although we have examined steganography in the context of free speech, it has numerous other uses. One common use is for the owners of images to encode secret messages in them stating their ownership rights. If such an image is stolen and placed on a Web site, the lawful owner can reveal the steganographic message in court to prove whose image it is. This technique is called **watermarking**. It is discussed in Muyco and Hernandez (2019).

Steganography is an active research area, with entire conferences devoted to the topic. Some interesting papers include Hegarty and Keane (2018), Kumar (2018), and Patil et al. (2019).

8.13.3 Copyright

Privacy and censorship are just two areas where technology meets public policy. A third one is the copyright law. **Copyright** is granting to the creators of **Intellectual Property**, including writers, poets, playwrights, artists, composers, musicians, photographers, cinematographers, choreographers, and others, the exclusive right to exploit their work for some period of time, typically the life of the author plus 50 years or 75 years in the case of corporate ownership. After the copyright of a work expires, it passes into the public domain and anyone can use or sell it as they wish. The Gutenberg Project (*www.gutenberg.org*), for example, has placed over 50,000 public-domain works (e.g., by Shakespeare, Mark Twain, and Charles Dickens) on the Web. In 1998, the U.S. Congress extended copyright in the U.S. by another 20 years at the request of Hollywood, which claimed that without an extension nobody would create anything any more. Protection of the original (1928) Mickey Mouse film was thus protected until 2024, after which time

anyone can rent a movie theater and legally show it without having to get permission from the Walt Disney Company. By way of contrast, patents last for only 20 years and people still invent things.

Copyright came to the forefront when Napster, a music-swapping service, had 50 million members. Although Napster did not actually copy any music, the courts held that its holding a central database of who had which song was contributory infringement, that is, it was helping other people infringe. While nobody seriously claims copyright is a bad idea (although many claim that the term is far too long, favoring big corporations over the public), the next generation of music sharing is already raising major ethical issues.

For example, consider a peer-to-peer network in which people share legal files (public-domain music, home videos, religious tracts that are not trade secrets, etc.) and perhaps a few that may be copyrighted. Assume that everyone is online all the time via ADSL or cable. Each machine has an index of what is on the hard disk, plus a list of other members. Someone looking for a specific item can pick a random member and see if he has it. If not, he can check out all the members in that person's list, and all the members in their lists, and so on. Computers are very good at this kind of work. Having found the item, the requester just copies it.

If the work is copyrighted, chances are the requester is infringing (although for international transfers, the question of whose law applies matters because in some countries uploading is illegal but downloading is not). But what about the supplier? Is it a crime to keep music you have paid for and legally downloaded on your hard disk where others might find it? If you have an unlocked cabin in the country and a thief sneaks in carrying a notebook computer and scanner, scans a copyrighted book to the notebook's hard disk, and sneaks out, are *you* guilty of the crime of failing to protect someone else's copyright?

But there is more trouble brewing on the copyright front. There is a huge battle going on now between Hollywood and the computer industry. The former wants stringent protection of all intellectual property but the latter does not want to be Hollywood's policeman. In October 1998, Congress passed the **DMCA** (**Digital Millennium Copyright Act**), which makes it a crime to circumvent any protection mechanism present in a copyrighted work or to tell others how to circumvent it. Similar legislation has been enacted in the European Union. While virtually no one thinks that pirates in the Far East should be allowed to duplicate copyrighted works, many people think that the DMCA completely shifts the balance between the copyright owner's interest and the public interest.

A case in point: in September 2000, a music industry consortium charged with building an unbreakable system for selling music online sponsored a contest inviting people to try to break the system (which is precisely the right thing to do with any new security system). A team of security researchers from several universities, led by Prof. Edward Felten of Princeton, took up the challenge and broke the system. They then wrote a paper about their findings and submitted it to a USENIX security conference, where it underwent peer review and was accepted. Before the

paper was presented, Felten received a letter from the Recording Industry Association of America threatening to sue under the DMCA if they published the paper.

Their response was to file a lawsuit asking a federal court to rule on whether publishing scientific papers on security research was still legal. Fearing a definitive court ruling against it, the industry reluctantly withdrew its threat and the court dismissed Felten's suit. No doubt the industry was motivated by the weakness of its case: it had invited people to try to break its system and then threatened to sue some of them for accepting its own challenge. With the threat withdrawn, the paper was published (Craver et al., 2001). A new confrontation is virtually certain.

Meanwhile, peer-to-peer networks have been used to exchange copyrighted content. In response, copyright holders have used the DMCA to send automated notices called **DMCA takedown notices** to users and ISPs. The copyright holders initially notified (and sued) individuals directly, which proved unpopular and ineffective. Now they are suing the ISPs for not terminating customers who are violating the DMCA. This is a tricky proposition, since peer-to-peer networks often have peers that lie about what they are sharing (Cuevas et al., 2014; and Santos et al., 2011) and your printer may even be mistaken for a culprit (Piatek et al., 2008), but the copyright holders are having some success with this approach: in December 2019, a federal court ordered Cox Communications to pay $1 billion to the copyright holders for not properly responding to takedown notices.

A related issue is the extent of the **fair use doctrine**, which has been established by court rulings in various countries. This doctrine says that purchasers of a copyrighted work have certain limited rights to copy the work, including the right to quote parts of it for scientific purposes, use it as teaching material in schools or colleges, and in some cases make backup copies for personal use in case the original medium fails. The tests for what constitutes fair use include (1) whether the use is commercial, (2) what percentage of the whole is being copied, and (3) the effect of the copying on sales of the work. Since the DMCA and laws within the European Union prohibit circumvention of copy protection schemes, these laws also prohibit legal fair use. In effect, the DMCA takes away historical rights from users to give content sellers more power. A showdown is inevitable.

Another development in the works that dwarfs even the DMCA in its shifting of the balance between copyright owners and users is **trusted computing** as advocated by industry bodies such as the **TCG** (**Trusted Computing Group**), led by companies like Intel and Microsoft. The idea is to provide support for carefully monitoring user behavior in various ways (e.g., playing pirated music) at a level below the operating system in order to prohibit unwanted behavior. This is accomplished with a small chip, called a **TPM** (**Trusted Platform Module**), which it is difficult to tamper with. Some PCs sold nowadays come equipped with a TPM. The system allows software written by content owners to manipulate PCs in ways that users cannot change. This raises the question of who is trusted in trusted computing. Certainly, it is not the user. Needless to say, the social consequences of this scheme are immense. It is nice that the industry is finally paying attention to

security, but it is lamentable that the driver is enforcing copyright law rather than dealing with viruses, crackers, intruders, and other security issues that most people are concerned about.

In short, the lawmakers and lawyers will be busy balancing the economic interests of copyright owners with the public interest for years to come. Cyberspace is no different from meatspace: it constantly pits one group against another, resulting in power struggles, litigation, and (hopefully) eventually some kind of resolution, at least until some new disruptive technology comes along.

8.14 SUMMARY

Security finds itself at the intersection of important properties such as confidentiality, integrity and availability (CIA). Unfortunately, it is often difficult to grasp in the sense that it is hard to specify exactly how secure a system is. What we can do is rigorously apply security principles such as those of Saltzer and Schroeder.

Meanwhile, adversaries will try to compromise a system by combining the fundamental building blocks reconnaissance (what is running where under what conditions), sniffing (eavesdropping on traffic), spoofing (pretending to be someone else), and disruption (denial-of-service). All of these building blocks can grow to be extremely advanced. To protect against these attacks and combinations thereof, network administrators install firewalls, intrusion detection systems and intrusion prevention systems. Such solutions may be deployed in the network as well as at the host and may work on the basis of signatures or anomalies. Either way, the number of false positives (false alerts) and false negatives (attacks missed) are important measures for the usefulness of such solutions. Especially if attacks are rare and there are many events, the Base Rate Fallacy dictates that false positives rate quickly reduces the power of an intrusion detection system.

Cryptography is a tool that can be used to keep information confidential and to ensure its integrity and authenticity. All modern cryptographic systems are based on Kerckhoffs' principle of having a publicly known algorithm and a secret key. Many cryptographic algorithms use complex transformations involving substitutions and permutations to transform the plaintext into the ciphertext. However, if quantum cryptography can be made practical, the use of one-time pads may provide truly unbreakable cryptosystems.

Cryptographic algorithms can be divided into symmetric-key algorithms and public-key algorithms. Symmetric-key algorithms mangle the bits in a series of rounds parametrized by the key to turn the plaintext into the ciphertext. AES (Rijndael) and triple DES are some of the most popular symmetric-key algorithms at present. These algorithms can be used in electronic code book mode, cipher block chaining mode, stream cipher mode, counter mode, and others.

Public-key algorithms have the property that different keys are used for encryption and decryption and that the decryption key cannot be derived from the encryption key. These properties make it possible to publish the public key. One of the main public-key algorithms is RSA, which derives its strength from the fact that it is difficult to factor large numbers. ECC-based algorithms are also used.

Legal, commercial, and other documents need to be signed. Accordingly, various schemes have been devised for digital signatures, using both symmetric-key and public-key algorithms. Commonly, messages to be signed are hashed using algorithms such as SHA-2 or SHA-3, and then the hashes are signed rather than the original messages.

Public-key management can be done using certificates, which are documents that bind a principal to a public key. Certificates are signed by a trusted authority or by someone (recursively) approved by a trusted authority. The root of the chain has to be obtained in advance, but browsers generally have many root certificates built into them.

These cryptographic tools can be used to secure network traffic. IPsec operates in the network layer, encrypting packet flows from host to host. Firewalls can screen traffic going into or out of an organization, often based on the protocol and port used. Virtual private networks can simulate an old leased-line network to provide certain desirable security properties. Finally, wireless networks need good security lest everyone read all the messages, and protocols like 802.11i provide it. Defense in depth, using multiple defense mechanisms, is always a good idea.

When two parties establish a session, they have to authenticate each other and, if need be, establish a shared session key. Various authentication protocols exist, including some that use a trusted third party, Diffie-Hellman, Kerberos, and public-key cryptography.

Email security can be achieved by a combination of the techniques we have studied in this chapter. PGP, for example, compresses messages, then encrypts them with a secret key and sends the secret key encrypted with the receiver's public key. In addition, it also hashes the message and sends the signed hash to verify message integrity.

Web security is also an important topic, starting with secure naming. DNSSEC provides a way to prevent DNS spoofing. Most e-commerce Web sites use TLS to establish secure, authenticated sessions between the client and server. Various techniques are used to deal with mobile code, especially sandboxing and code signing.

Finally, the Internet raises many issues in which technology interacts strongly with public policy. Some of the areas include privacy, freedom of speech, and copyright. Addressing these issues requires contribution from multiple disciplines. Given the speed at which technology evolves and the speed at which legislation and public policy evolve, we will stick out our necks and predict that these issues will not be solved by the time the next edition of this book is in print. In case we are wrong, we will buy all our readers a wheel of cheese.

PROBLEMS

1. Consider the principle of complete mediation. Which non-functional system requirement will likely be affected by adhering strictly to this principle?

2. What type of scan does the following network log represent? Complete your answer as accurately as possible, indicating which hosts you think are up and which ports you think are open or closed.

```
Time          From            To                  Flags  Other info
21:03:59.711106 brutus.net.53 > host201.caesar.org.21: F 0:0(0) win 2048 (ttl 48, id 55097)
21:04:05.738307 brutus.net.53 > host201.caesar.org.21: F 0:0(0) win 2048 (ttl 48, id 50715)
21:05:10.399065 brutus.net.53 > host202.caesar.org.21: F 0:0(0) win 3072 (ttl 49, id 32642)
21:05:16.429001 brutus.net.53 > host202.caesar.org.21: F 0:0(0) win 3072 (ttl 49, id 31501)
21:09:12.202997 brutus.net.53 > host024.caesar.org.21: F 0:0(0) win 2048 (ttl 52, id 47689)
21:09:18.215642 brutus.net.53 > host024.caesar.org.21: F 0:0(0) win 2048 (ttl 52, id 26723)
21:10:22.664153 brutus.net.53 > host003.caesar.org.21: F 0:0(0) win 3072 (ttl 53, id 24838)
21:10:28.691982 brutus.net.53 > host003.caesar.org.21: F 0:0(0) win 3072 (ttl 53, id 25257)
21:11:10.213615 brutus.net.53 > host102.caesar.org.21: F 0:0(0) win 4096 (ttl 58, id 61907)
21:11:10.227485 host102.caesar.org.21 > brutus.net.53: R 0:0(0) ack 4294947297 win 0 (ttl 25, id 38400)
```

3. What type of scan does the following network log represent? Complete your answer as accurately as possible, indicating which hosts you think are up and which ports you think are open or closed.

```
Time          From            To                  Flags  Other info
20:31:49.635055 IP 127.0.0.1.56331 > 127.0.0.1.22:    Flags [FPU], seq 149982695, win 4096, urg 0, leng
20:31:49.635123 IP 127.0.0.1.56331 > 127.0.0.1.80:    Flags [FPU], seq 149982695, win 3072, urg 0, leng
20:31:49.635162 IP 127.0.0.1.56331 > 127.0.0.1.25:    Flags [FPU], seq 149982695, win 4096, urg 0, leng
20:31:49.635200 IP 127.0.0.1.25    > 127.0.0.1.56331: Flags [R.], seq 0, ack 149982696, win 0, length 0
20:31:49.635241 IP 127.0.0.1.56331 > 127.0.0.1.10000: Flags [FPU], seq 149982695, win 3072, urg 0, leng
20:31:49.635265 IP 127.0.0.1.10000 > 127.0.0.1.56331: Flags [R.], seq 0, ack 149982696, win 0, length 0
20:31:50.736353 IP 127.0.0.1.56332 > 127.0.0.1.80:    Flags [FPU], seq 150048230, win 1024, urg 0, leng
20:31:50.736403 IP 127.0.0.1.56332 > 127.0.0.1.22:    Flags [FPU], seq 150048230, win 3072, urg 0, leng
```

4. What is an algorithmic complexity DoS attack?

5. Alice wants to communicate with the www.vu.nl Web site, but the entry for this domain in her name server was poisoned so that the packets end up at an attacker-controlled machine. To what extent is the attacker able to compromise Confidentiality, Integrity, and Authenticity in the following cases: (a) unencrypted (http) communication between Alice and www.vu.nl, (b) encrypted (https) communication between Alice and www.vu.nl when the Web site uses a self-signed certificate, (c) encrypted (https) communication between Alice and www.vu.nl when the Web site uses a certificate signed by a legitimate certificate authority?

6. A stateless firewall blocks TCP connection initiation requests from an external location to any local host. Explain why this defense is not very effective against sophisticated attackers.

7. Explain the base rate fallacy using the IDS performance of the previous question.

8. You are performing an off-path TCP hijacking attack on Herbert's machine and have already established that Herbert is logged in from his machine to the FTP server at vusec.net (recall: FTP uses destination port 21 for commands). Both machines run Linux and implement the original RFC 5961, as discussed in the text. Using the off-path TCP exploitation technique, you now also want to discover the source port of the FTP control connection (at Herbert's end). Assume all port numbers are possible in principle and that you can send an infinite number of packets per second. Show how we can use a binary search to find the correct port number quickly. Using this technique, how many *spoofed* packets do you need to send in the worst case? Explain.

9. Break the following monoalphabetic substitution cipher. The plaintext, consisting of letters only, is an excerpt from a poem by Lewis Carroll.

mvyy bek mnyx n yvjjyr snijrh invq n muvjvdt je n idnvy
jurhri n fehfevir pyeir oruvdq ki ndq uri jhrnqvdt ed zb jnvy
Irr uem rntrhyb jur yeoijrhi ndq jur jkhjyri nyy nqlndpr
Jurb nhr mnvjvdt ed jur iuvdtyr mvyy bek pezr ndq wevd jur qndpr
mvyy bek, medj bek, mvyy bek, medj bek, mvyy bek wevd jur qndpr
mvyy bek, medj bek, mvyy bek, medj bek, medj bek wevd jur qndpr

10. An affine cipher is a version of a monoalphabetic substitution cipher, in which the letters of an alphabet of size m are first mapped to the integers in the range 0 to $m - 1$. Subsequently, the integer representing each plaintext letter is transformed to an integer representing the corresponding ciphertext letter. The encryption function for a single letter is $E(x) = (ax + b) \bmod m$, where m is the size of the alphabet and a and b are the key of the cipher, and are co-prime. Trudy finds out that Bob generated a ciphertext using an affine cipher. She gets a copy of the ciphertext, and finds out that the most frequent letter of the ciphertext is ''R'', and the second most frequent letter of the ciphertext is ''K''. Show how Trudy can break the code and retrieve the plaintext.

11. Break the following columnar transposition cipher. The plaintext is taken from a popular computer networks textbook, so ''connected''' is a probable word. The plaintext consists entirely of letters (no spaces). The ciphertext is broken up into blocks of four characters for readability.

oeet nott rece rowp sabe ndea oana tmrs otne heth imnc trdi ccfa lxgo ioua iere iybe nft

12. Alice used a transposition cipher to encrypt her messages to Bob. For added security, she encrypted the transposition cipher key using a substitution cipher, and kept the encrypted cipher in her computer. Trudy managed to get hold of the encrypted transposition cipher key. Can Trudy decipher Alice's messages to Bob? Why or why not?

13. Find a 77-bit one-time pad that generates the text ''Donald Duck'' from the ciphertext of Fig. 8-11.

14. You are a spy, and, conveniently, have a library with an infinite number of books at your disposal. Your operator also has such a library at his disposal. You have initially agreed to use *Lord of the Rings* as a one-time pad. Explain how you could use these assets to generate an infinitely long one-time pad.

15. Quantum cryptography requires having a photon gun that can, on demand, fire a single photon carrying 1 bit. In this problem, calculate how many photons a bit carries on a 250-Gbps fiber link. Assume that the length of a photon is equal to its wavelength, which for purposes of this problem, is 1 micron. Also, assume that the speed of light in fiber is 20 cm/nsec.

16. If Trudy captures and regenerates photons when quantum cryptography is in use, she will get some of them wrong and cause errors to appear in Bob's one-time pad. What fraction of Bob's one-time pad bits will be in error, on average?

17. A fundamental cryptographic principle states that all messages must have redundancy. But we also know that redundancy helps an intruder tell if a guessed key is correct. Consider two forms of redundancy. First, the initial n bits of the plaintext contain a known pattern. Second, the final n bits of the message contain a hash over the message. From a security point of view, are these two equivalent? Discuss your answer.

18. Consider a banking system that uses the following format for transaction messages: two bytes for the sender ID, two bytes for the receiver ID, and four bytes for the amount to be transferred. Transactions are encrypted before sending. What could you add to these messages to make them adhere to the two cryptographic principles discussed in this chapter?

19. A group of nasty people doing nasty business do not want the police to listen in on their digital communications. To make sure this does not happen, they use an end-to-end encrypted messaging system that uses an unbreakable cipher. Think of two approaches that can still allow the police to eavesdrop on their conversations.

20. Suppose we have a cipher-breaking machine with a million processors that can analyze a key in 1 nanosecond. It would take 10^{16} years to break the 128-bit version of AES. Let us compute how long it will take for this time to get down to 1 year, still a long time, of course. To achieve this goal, we need computers to be 10^{16} times faster. If Moore's Law (computing power doubles every 18 months) continues to hold, how many years will it take before a parallel computer can get the cipher-breaking time down to a year?

21. AES supports a 256-bit key. How many keys does AES-256 have? See if you can find some number in physics, chemistry, or astronomy of about the same size. Use the Internet to help search for big numbers. Draw a conclusion from your research.

22. Consider ciphertext block chaining. Instead of a single 0 bit being transformed into a 1 bit, an extra 0 bit is inserted into the ciphertext stream after block C_i. How much plaintext will be garbled as a result?

23. Compare cipher block chaining with cipher feedback mode in terms of the number of encryption operations needed to transmit a large file. Which one is more efficient and by how much?

24. Alice and Bob are communicating using public-key cryptography. Who can retrieve the plaintext, P, from $E_B(D_A(P))$, and which steps are required to do so?

25. A few years from now, you are a teaching assistant for Computer Networks. You explain to the students that in RSA cryptography, the public and private keys consist of (e, n) and (d, n) respectively. The possible values of e and d depend on a value z,

whose possible values depend in turn on n. One of the students comments that this scheme is unnecessarily complicated, and proposes to simply it. Instead of selecting d as a relative prime to z, d is selected as a relative prime to n. Then e is found such that $e \times d = 1$ modulo n. This way, z is no longer needed. How does this change affect the effort required to break the cipher?

26. Trudy's RSA keys are as follows: $n_t = 33$, $d_t = 3$, $e_t = 7$. Trudy finds out that Bob's public key is $n_b = 33, e_b = 3$.
(a) How can Trudy use this information to read encrypted messages directed to Bob?
(b) Based on your conclusions from section (a), calculate the number of secure public key pairs for a specific pair of p and q.

27. Alice and Bob use RSA public key encryption in order to communicate between them. Trudy finds out that Alice and Bob shared one of the primes used to determine the number n of their public key pairs. In other words, Trudy found out that $n_a = p_a \times q$ and $n_b = p_b \times q$. How can Trudy use this information to break Alice's code?

28. In Fig. 8-23, we see how Alice can send Bob a signed message. If Trudy replaces P, Bob can detect it. But what happens if Trudy replaces both P and the signature?

29. Digital signatures have a potential weakness due to lazy users. In e-commerce transactions, a contract might be drawn up and the user asked to sign its SHA hash. If the user does not actually verify that the contract and hash correspond, the user may inadvertently sign a different contract. Suppose that the Mafia try to exploit this weakness to make some money. They set up a pay Web site (e.g., pornography, gambling, etc.) and ask new customers for a credit card number. Then they send over a contract saying that the customer wishes to use their service and pay by credit card and ask the customer to sign it, knowing that most of them will just sign without verifying that the contract and hash agree. Show how the Mafia can buy diamonds from a legitimate Internet jeweler and charge them to unsuspecting customers.

30. A math class has 25 students. Assuming that all of the students were born in the first half of the year—between January 1st and June 30th—what is the probability that at least two students have the same birthday? Assume that nobody was born on leap day.

31. After Ellen confessed to Marilyn about tricking her in the matter of Tom's tenure, Marilyn resolved to avoid this problem by dictating the contents of future messages into a dictating machine and having her new secretary just type them in. Marilyn then planned to examine the messages on her terminal after they had been typed in to make sure they contained her exact words. Can the new secretary still use the birthday attack to falsify a message, and if so, how? *Hint*: She can.

32. Consider the failed attempt of Alice to get Bob's public key in Fig. 8-25. Suppose that Bob and Alice already share a secret key, but Alice still wants Bob's public key. Is there now a way to get it securely? If so, how?

33. Alice wants to communicate with Bob, using public-key cryptography. She establishes a connection to someone she hopes is Bob. She asks him for his public key and he sends it to her in plaintext along with an X.509 certificate signed by the root CA. Alice already has the public key of the root CA. What steps does Alice carry out to verify that she is talking to Bob? Assume that Bob does not care who he is talking to (e.g., Bob is some kind of public service).

34. Suppose that a system uses PKI based on a tree-structured hierarchy of CAs. Alice wants to communicate with Bob, and receives a certificate from Bob signed by a CA X after establishing a communication channel with Bob. Suppose Alice has never heard of X. What steps does Alice take to verify that she is talking to Bob?

35. Can IPsec using AH be used in transport mode if one of the machines is behind a NAT box? Explain your answer.

36. Alice wants to send a message to Bob using SHA-2 hashes. She consults with you regarding the appropriate signature algorithm to be used. What would you suggest?

37. Give one advantage of HMACs over using RSA to sign SHA-2 hashes.

38. Give one reason why a firewall might be configured to inspect incoming traffic. Give one reason why it might be configured to inspect outgoing traffic. Do you think the inspections are likely to be successful?

39. Suppose an organization uses a secure VPN to securely connect its sites over the Internet. Jim, a user in the organization, uses the VPN to communicate with his boss, Mary. Describe one type of communication between Jim and Mary which would not require use of encryption or other security mechanism, and another type of communication which would require encryption or other security mechanisms. Please explain your answer.

40. Change one message in the protocol of Fig. 8-31 in a minor way to make it resistant to the reflection attack. Explain why your change works.

41. The Diffie-Hellman key exchange is being used to establish a secret key between Alice and Bob. Alice sends Bob (227, 5, 82). Bob responds with (125). Alice's secret number, x, is 12, and Bob's secret number, y, is 3. Show how Alice and Bob compute the secret key.

42. If Alice and Bob have never met, share no secrets, and have no certificates, they can nevertheless establish a shared secret key using the Diffie-Hellman algorithm. Explain why it is very hard to defend against a man-in-the-middle attack.

43. In the protocol of Fig. 8-36, why is A sent in plaintext along with the encrypted session key?

44. Are timestamps and nonces used for confidentiality, integrity, availability, authentication, or nonrepudiation? Explain your answer.

45. In the protocol of Fig. 8-36, we pointed out that starting each plaintext message with 32 zero bits is a security risk. Suppose that each message begins with a per-user random number, effectively a second secret key known only to its user and the KDC. Does this eliminate the known plaintext attack? Why?

46. Confidentiality, integrity, availability, authentication, and nonrepudiation are fundamental security properties. For each of these properties, explain if it can be provided by public-key cryptography. If yes, explain how.

47. Consider the fundamental security problems listed in the problem above. For each of these properties, explain if it can be provided by message digests. If yes, explain how.

48. In the Needham-Schroeder protocol, Alice generates two challenges, R_A and R_{A2}. This seems like overkill. Would one not have done the job?

49. Suppose an organization uses Kerberos for authentication. In terms of security and service availability, what is the effect if AS or TGS goes down?

50. Alice is using the public-key authentication protocol of Fig. 8-40 to authenticate communication with Bob. However, when sending message 7, Alice forgot to encrypt R_B. Trudy now knows the value of R_B. Do Alice and Bob need to repeat the authentication procedure with new parameters in order to ensure secure communication? Explain your answer.

51. In the public-key authentication protocol of Fig. 8-40, in message 7, R_B is encrypted with K_S. Is this encryption necessary, or would it have been adequate to send it back in plaintext? Explain your answer.

52. Point-of-sale terminals that use magnetic-stripe cards and PIN codes have a fatal flaw: a malicious merchant can modify his card reader to log all the information on the card and the PIN code in order to post additional (fake) transactions in the future. Next generation terminals will use cards with a complete CPU, keyboard, and tiny display on the card. Devise a protocol for this system that malicious merchants cannot break.

53. You get an email from your bank saying unusual behavior was detected on your account. However, when you follow the embedded link in the email and log into their Web site, it does not show any transactions. You log out again. Perhaps it was a mistake. One day later you go back to the bank's Web site and log in. This time, it shows you that all your money has been transferred to an unknown account. What happened?

54. Give *two* reasons why PGP compresses messages.

55. Is it possible to multicast a PGP message? What restrictions would apply?

56. Assuming that everyone on the Internet used PGP, could a PGP message be sent to an arbitrary Internet address and be decoded correctly by all concerned? Discuss your answer.

57. The SSL data transport protocol involves two nonces as well as a premaster key. What value, if any, does using the nonces have?

58. Consider an image of 2048 × 1536 pixels. You want to hide a file sized 2.5 MB. What fraction of the file can you steganographically hide in this image? What fraction would you be able to hide if you compressed the file to a quarter of its original size? Show your calculations.

59. The image of Fig. 8-52(b) contains the ASCII text of five plays by Shakespeare. Would it be possible to hide music among the zebras instead of text? If so, how would it work and how much could you hide in this picture? If not, why not?

60. You are given a text file of size 60 MB, which is to be hidden using steganography in the low-order bits of each color in an image file. What size image would be required in order to encrypt the entire file? What size would be needed if the file were first compressed to a third of its original size? Give your answer in pixels, and show your calculations. Assume that the images have an aspect ratio of 3:2, for example, 3000 × 2000 pixels.

61. Alice was a heavy user of a type 1 anonymous remailer. She would post many messages to her favorite newsgroup, *alt.fanclub.alice*, and everyone would know they all came from Alice because they all bore the same pseudonym. Assuming that the remailer worked correctly, Trudy could not impersonate Alice. After type 1 remailers were all shut down, Alice switched to a cypherpunk remailer and started a new thread in her newsgroup. Devise a way for her to prevent Trudy from posting new messages to the newsgroup, impersonating Alice.

62. In 2018, researchers found a pair of vulnerabilities in modern processors they called Spectre and Meltdown. Find out how the Meltdown attack works and explain which of the security principles were not sufficiently adhered to by the processor designers, causing the introduction of these vulnerabilities. Explain your answer. Give a possible motivation for not adhering strictly to these principles.

63. While traveling abroad, you connect to the WiFi network in your hotel using a unique password. Explain how an attacker may eavesdrop on your communication.

64. Search the Internet for some court case involving copyright versus fair use and write a 1-page report summarizing your findings.

65. Write a program that encrypts its input by XORing it with a keystream. Find or write as good a random number generator as you can to generate the keystream. The program should act as a filter, taking plaintext on standard input and producing ciphertext on standard output (and vice versa). The program should take one parameter, the key that seeds the random number generator.

66. Write a procedure that computes the SHA-2 hash of a block of data. The procedure should have two parameters: a pointer to the input buffer and a pointer to a 20-byte output buffer. To see the exact specification of SHA-2, search the Internet for FIPS 180-1, which is the full specification.

67. Write a function that accepts a stream of ASCII characters and encrypts this input using a substitution cipher with the Cipher Block Chaining mode. The block size should be 8 bytes. The program should take plaintext from the standard input and print the ciphertext on the standard output. For this problem, you are allowed to select any reasonable system to determine that the end of the input is reached, and/or when padding should be applied to complete the block. You may select any output format, as long as it is unambiguous. The program should receive two parameters:

1. A pointer to the initializing vector; and
2. A number, k, representing the substitution cipher shift, such that each ASCII character would be encrypted by the kth character ahead of it in the alphabet.

For example, if $x = 3$, then "A" is encoded by "D", "B" is encoded by "E" etc. Make reasonable assumptions with respect to reaching the last character in the ASCII set. Make sure to document clearly in your code any assumptions you make about the input and encryption algorithm.