THE EXPERT'S VOICE® IN SECUR

Foundations

Secu

Foundations of Security: What Every Programmer Needs to Know

Copyright © 2007 by Neil Daswani, Christoph Kern, and Anita Kesavan

ISBN-13 (pbk): 978-1-59059-784-2

ISBN-10 (pbk): 1-59059-784-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewer: Dan Pilone
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole Flores
Copy Editor: Damon Larson
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreader: Liz Welch
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code/ Download section.

assume he is communicating with Alice. Passwords are so prevalently used that we dedicate Chapter 9 to studying how to properly build a password management system.

There are advantages and disadvantages to using passwords. One advantage is that password schemes are simple to implement compared to other authentication mechanisms, such as biometrics, which we will discuss later in this chapter. Another advantage of password security systems is that they are simple for users to understand.

There are, however, disadvantages to using password security systems. First, most users do not choose strong passwords, which are hard for attackers to guess. Users usually choose passwords that are simple concatenations of common names, common dictionary words, common street names, or other easy-to-guess terms or phrases. Attackers interested in hacking into somebody's account can use password-cracking programs to try many common login names and concatenations of common words as passwords. Such password cracking programs can easily determine 10 to 20 percent of the usernames and passwords in a system. Of course, to gain access to a system, an attacker typically needs only one valid username and password. Passwords are relatively easy to crack, unless users are somehow forced to choose passwords that are hard for such password-cracking programs to guess. A second disadvantage of password security systems is that a user needs to reuse a password each time she logs into a system—that gives an attacker numerous opportunities to "listen in" (see Section 1.4) on that password. If the attacker can successfully listen in on a password just once, the attacker can then log in as the user.

A one-time password (OTP) system, which forces the user to enter a new password each time she logs in, eliminates the risks of using a password multiple times. With this system, the user is given a list of passwords—the first time she logs in, she is asked for the first password; the second time she logs in, she is asked the second password; and so on. The major problem with this system is that no user will be able to remember all these passwords. However, a device could be used that keeps track of all the different passwords the user would need to use each time she logs in. This basic idea of such a device naturally leads us from the topic of "something you know" to the topic of "something you have."

1.2.2. Something You Have

A second general method of authenticating a user is based on something that the user has.

OTP Cards

OTP products generate a new password each time a user needs to log in. One such product, offered by RSA Security, is the SecurID card (other companies have different names for such cards). The SecurID card is a device that flashes a new password to the user periodically (every 60 seconds or so). When the user wants to log into a computer system, he enters the number displayed on the card when prompted by the server. The server knows the algorithm that the SecurID card uses to generate passwords, and can verify the password that the user enters. There are many other variations of OTP systems as well. For instance, some OTP systems generate passwords for their users only when a personal identification number (PIN) is entered. Also, while OTP systems traditionally required users to carry additional devices, they are sometimes now integrated into personal digital assistants (PDAs) and cell phones.

Smart Cards

Another mechanism that can authenticate users based on something that they have is a *smart card*. A smart card is tamper-resistant, which means that if a bad guy tries to open the card or gain access to the information stored on it, the card will self-destruct. The card will not self-destruct in a manner similar to what comes to mind when you think of *Mission Impossible*. Rather, the microprocessor, memory, and other components that make up the "smart" part of the smart card are epoxied (or glued) together such that there is no easy way to take the card apart. The only feasible way to communicate with the microprocessor is through its electronic interface. Smart cards were designed with the idea that the information stored in the card's memory would only be accessible through the microprocessor. A smart card's microprocessor runs software that can authenticate a user while guarding any secret information stored on the card. In a typical scenario, a user enters a smart card into a smart card reader, which contains a numeric keypad. The smart card issues a "challenge" to the reader. The user is required to enter a PIN into the reader, and the reader computes a response to the challenge. If the smart card receives a correct response, the user is considered authenticated, and access to use the secret information stored on the smart card is granted.

One problem with using smart cards for authentication is that the smart card reader (into which the PIN is entered) must be trusted. A rogue smart card reader that is installed by a bad guy can record a user's PIN, and if the bad guy can then gain possession of the smart card itself, he can authenticate himself to the smart card as if he were the user. While such an attack sounds as if it requires quite a bit of control on the part of the attacker, it is very feasible. For example, an attacker could set up a kiosk that contains a rogue smart card reader in a public location, such as a shopping mall. The kiosk could encourage users to enter their smart cards and PINs by displaying an attractive message such as "Enter your smart card to receive a 50 percent discount on all products in this shopping mall!" Such types of attacks have occurred in practice. Attacks against smart cards have also been engineered by experts such as Paul Kocher, who runs a security company called Cryptography Research (`www.cryptography.com`). By studying a smart card's power consumption as it conducted various operations, Kocher was able to determine the contents stored on the card. While such attacks are possible, they require a reasonable amount of expertise on the part of the attacker. However, over time, such attacks may become easier to carry out by an average attacker.

ATM Cards

The ATM (automatic teller machine) card is another example of a security mechanism based on some secret the user has. On the back of an ATM card is a magnetic stripe that stores data—namely the user's account number. This data is used as part of the authentication process when a user wants to use the ATM. However, ATM cards, unlike smart cards, are not tamper-resistant—anyone who has a magnetic stripe reader can access the information stored on the card, without any additional information, such as a PIN. In addition, it is not very difficult to make a copy of an ATM card onto a blank magnetic stripe card. Since the magnetic stripe on an ATM card is so easy to copy, credit card companies also sometimes incorporate holograms or other hard-to-copy elements on the cards themselves. However, it's unlikely that a cashier or point-of-sale device will actually check the authenticity of the hologram or other elements of the card.

In general, the harder it is for an attacker to copy the artifact that the user has, the stronger this type of authentication is. Magnetic stripe cards are fairly easy to copy. Smart cards, however, are harder to copy because of their tamper-resistance features.

1.2.3. Something You Are

The third general method of authenticating a user is based on something that the user *is*. Most of the authentication techniques that fall into this category are biometric techniques, in which something about the user's biology is measured. When considering a biometric authentication technique as part of your system, it is important to consider its effectiveness and social acceptability.

The first biometric authentication technique that we consider is a palm scan in which a reader measures the size of a person's hand and fingers, and the curves that exist on their palm and fingers. It also incorporates fingerprint scans on each of the fingers. In this way, the palm scan technique is much more effective than simply taking a single fingerprint of the user.

A second technique used to biometrically authenticate someone is to scan their iris. In this technique, a camera takes a picture of a person's iris and stores certain features about it in the system. Studies have been conducted to measure how comfortable people are with such scans, and the iris scan appears to be more socially acceptable than the palm scan. In the palm scan technique, the user is required to actually put her hand on the reader for a few seconds, while in the iris scan, a camera just takes a quick picture of the user's iris. The iris scan is less intrusive since the user does not have to do anything except look in a particular direction.

Another biometric technique is a retinal scan, in which infrared light is shot into a user's eyes, and the pattern of retinal blood vessels is read to create a signature that is stored by a computer system. In a retinal scan, the user puts his head in front of a device, and then the device blows a puff of air and shoots a laser into the user's eye. As you can imagine, a retinal scan is more intrusive than an iris scan or a palm scan.

Another biometric authentication technique is fingerprinting. In fingerprinting, the user places her finger onto a reader that scans the set of curves that makes up her fingerprint. Fingerprinting is not as socially accepted as other biometric identification techniques since people generally associate taking fingerprints with criminal activity. In addition, fingerprinting provides less information than a palm scan.

Voice identification is a mechanism in which a computer asks a user to say a particular phrase. The computer system then takes the electrically coded signals of the user's voice, compares them to a databank of previous signals, and determines whether there is close enough of a match.

Facial recognition involves a camera taking a picture of a person's face and a computer system trying to recognize its features.

Another technique, signature dynamics, records not only a user's signature, but also the pressure and timing at which the user makes various curves and motions while writing. The advantage of signature dynamics over simple signature matching is that it is far more difficult to replicate.

The key disadvantages to these biometric authentication techniques are the number of false positives and negatives generated, their varying social acceptance, and key management issues.

A *false positive* occurs when a user is indeed an authentic user of the system, but the biometric authentication device rejects the user. A *false negative*, on the other hand, occurs when an impersonator successfully impersonates a user.

Social acceptance is another issue to take into account when considering biometric authentication techniques. All the biometric authentication techniques discussed here are less socially accepted than entering a password.

The final disadvantage for biometric authentication techniques is the key management issue. In each of these biometric authentication techniques, measurements of the user's biology are used to construct a key, a supposedly unique sequence of zeros and ones that corresponds only to a particular user. If an attacker is able to obtain a user's biological measurements, however, the attacker will be able to impersonate the user. For example, a criminal may able to "copy" a user's fingerprint by re-creating it with a wax imprint that the criminal puts on top of his finger. If you think of the user's fingerprint as a "key," then the key management issue in this case is that we cannot revoke the user's key because the user cannot get a new fingerprint—even though her original fingerprint has been stolen. By contrast, the keys in password systems are generated from passwords, and users can easily have their passwords changed if they are ever stolen or compromised. Biometric authentication becomes ineffective once attackers are able to impersonate biometric measurements.

1.2.4. Final Notes on Authentication

Combining various authentication techniques can be more effective than using a single authentication technique. For example, in the previous section, we discussed some of the disadvantages of using biometric authentication alone. However, if you combine biometric authentication with another technique, such as a password or a token, then the authentication process becomes more effective.

The term *two-factor authentication* is used to describe the case in which a user is to be authenticated based upon two methods. ATM cards are an example of two-factor authentication at work. ATM cards have magnetic stripes that have the user's name and account number. When the card is used, the user is required to enter not only the card into the teller machine, but also a PIN, which can basically be thought of as a password. In such an example of *two-factor authentication*, the bank requires the user to be authenticated based upon two methods—in this case, something that the user has and something that the user knows.

There are other factors that can be taken into account when conducting authentication. For instance, Alice's location can be considered a factor. Alice may carry around a cell phone that has a GPS (Global Positioning System) chip inside of it. When Alice is standing in front of an ATM requesting to withdraw money, Alice's bank could ask her cell phone company's computer system where she currently is. If the cell phone company's computer responds with a latitude and longitude that corresponds to the expected location of the ATM, the bank can approve the withdrawal request. However, if Alice's ATM card and PIN were stolen by a bad guy who is trying to withdraw money, then taking Alice's location (or specifically, the location of her cell phone) into account could help thwart such a fraudulent withdrawal request. If Alice's cell phone is still in her possession, when an attacker attempts to use her card at an ATM, the location of the ATM will not correspond to the location of Alice's cell phone, and the bank will deny the withdrawal request (unless, of course, Alice and her cell phone are being held captive in front of the ATM). In this example, it is advantageous for Alice to keep her cell phone and her ATM card in different places; she should not, say, keep both of them in her purse.

In all the examples discussed so far, we have talked about people authenticating people or people authenticating themselves to computers. In a large distributed system, however, computers are also interacting with other computers. The computers may have to authenticate themselves to each other because all computers cannot be trusted equally. There are many protocols that can be used to allow computer-to-computer authentication, and these protocols will, in general, support three types of authentication: client authentication, server authentication, and mutual authentication.

Client authentication involves the server verifying the client's identity, *server authentication* involves the client verifying the server's identity, and *mutual authentication* involves the client and server verifying each other's identity. When we discuss protocols, such as Secure Sockets Layer (SSL) in Chapter 15, we will discuss the different modes they use to support client, server, and mutual authentication.

Whether client, server, or mutual authentication is done often depends upon the nature of the application and the expected threats. Many e-commerce web sites provide server authentication once a user is ready to make a purchase because they do not want the client to submit a credit card number to a spoofed or impostor web site. Spoofed web sites are a significant security threat because they do not cost much to set up.

On the other hand, in older cell phone networks, only client authentication was required. Cell phone towers (servers) would only check that a phone (client) that attempted to communicate with it was owned by an authentic customer. The phones did not authenticate the cell phone towers because cell phone towers were costly to set up, and an attacker would require significant capital to spoof a cell phone tower. On the other hand, the cell phones themselves were much cheaper, and hence wireless carriers only required phones to be authenticated. Today, the cost of cell phone base stations is significantly cheaper, and modern-day cell phone networks use mutual authentication.

Now that we have completed our discussion of authentication, we are going to explore our next security concept: authorization.

1.3. Authorization

Authorization is the act of checking whether a user has permission to conduct some action. Whereas authentication is about verifying identity, authorization is about verifying a user's authority. To give a concrete example, let us examine the case in which Alice authenticates herself at an ATM by putting in her ATM card and entering her PIN. Alice may want to deduct \$500, but may only be authorized to deduct a maximum of \$300 per day. If Alice enters \$500 as the amount that she is requesting to deduct, the system will not authorize her transaction even if she successfully authenticates herself.

In the previous example, an authorization check questions whether Alice has the authority to deduct a certain amount of money. Operating systems such as Windows and Linux do authorization checks all the time. For example, when Alice attempts to delete a file, the operating system checks whether Alice is allowed to do so. A general mechanism called an *access control list* (ACL) is used by many operating systems to determine whether users are authorized to conduct different actions.

1.3.1. Access Control Lists (ACLs)

Minimally, an ACL is a set of users and a corresponding set of resources they are allowed to access. For example, Alice may have access to all the files in her home directory,[1] but may not have access to Bob's files. Suppose Alice's home directory is `/home/Alice`, and Bob's home directory is `/home/Bob`. An ACL that models this would list Alice as the principal,[2] and it would also list the set of files in her home directory that she is allowed to access, as shown in Table 1-1. In the table, an asterisk (*) is used as a wildcard to indicate all files and subdirectories within a particular home directory. An ACL may optionally include privileges that are associated with resources. The Privilege column indicates that Alice and Bob are allowed to read, write, and execute files in their respective home directories.

Table 1-1. *A Simple ACL*

User	Resource	Privilege
Alice	`/home/Alice/*`	Read, write, execute
Bob	`/home/Bob/*`	Read, write, execute

In some more sophisticated ACL schemes, another piece of information called a *role* is added, which enables a user or principal to access particular resources. Table 1-2 shows an example mapping of users to roles, and Table 1-3 shows a role-based ACL. In Table 1-2, Alice is both a programmer and an administrator, and Bob is both a programmer and a backup operator.[3]

Table 1-2. *A User-Role Mapping*

User	Role
Alice	Administrator, Programmer
Bob	Backup Operator, Programmer

Table 1-3. *A Role-Based ACL*

Role	Resource	Privilege
Backup Operator	`/home/*`	Read
Administrator	`/*`	Read, write, execute

1. A user's *home directory* is the location on a file system where her files are stored.
2. An entity (or a process) that is capable of being authenticated is often referred to as a principal.
3. A backup operator is responsible for backing up all user files on a periodic basis.

1.3.2. Access Control Models

ACLs can be used to implement one of three access control models—the mandatory access control (MAC) model, the discretionary access control (DAC) model, and the role-based access control (RBAC) model—sometimes called the non-discretionary access model.

Mandatory Access Control (MAC)

In the MAC model, the computer system decides exactly who has access to which resources in the system. In the MAC model, if Alice creates a new document, the system can decide that no one but Alice is allowed to access that document. Alice herself does not have the right to decide who else is allowed to access the file that she authored. Even if she wants to share the document she authored with her friend Bob, she is not authorized to make that decision. For instance, if Alice creates a file `/home/Alice/product_specs.txt` in a system with a MAC model, there would be no way for Alice to decide on her own to allow Bob to see that file. In a MAC model, only the computer system determines who is authorized to access documents that Alice creates.

Discretionary Access Control (DAC)

The DAC model is different from the MAC model in that users are authorized to determine which other users can access files or other resources that they create, use, or own. In a discretionary access system, Alice could let Bob access a file at her discretion by issuing a command to the system, and then Bob would be given access to that file. For instance, in UNIX, which uses a DAC model, Alice could issue the command `chmod a+r /home/Alice/product_specs.txt` to allow all users on the system to read the file. The ACL that results from such a command is shown in Table 1-4, in which the third row specifies that every user (denoted by *) has read privileges for the file `/home/Alice/product_specs.txt`.

Table 1-4. *The Resulting ACL*

User	Resource	Privilege
Alice	`/home/Alice/*`	Read, write, execute
Bob	`/home/Bob/*`	Read, write, execute
*	`/home/Alice/product_specs.txt`	Read

Role-Based Access Control (RBAC)

The third access control model is the RBAC model, which is similar to the MAC model in the sense that the system decides exactly which users are allowed to access which resources—but the system does this in a special way. A RBAC system will incorporate the user's role into its access decision. For instance, the system may know about the user's position (or role) within a company (e.g., administrative assistant, manager, or CEO) and give the user different privileges based on that role. For instance, the CEO may be allowed to access salary information about any employee in the company, whereas a manager may only be able to access salary information about his or her subordinates.

As per the role-based ACL shown in Table 1-3, a backup operator is allowed to read data from all user home directories (`/home/*`) so that the data can be archived. However, a principal with an administrator role, such as Alice, may be able to read, write, and execute files anywhere on the file system. Users that have multiple roles would declare their role just prior to conducting an action, such as doing a backup or modifying a file. While a user such as Bob may have both read and write privileges to some files (such as those in his home directory), the purpose of the role would be to ensure that he could not inadvertently modify a file while doing a backup.

Another example might use the concept of a group in the UNIX operating system to implement RBAC. All users with a particular role would be placed in a group with the same name as their role (e.g., Alice and Bob would be members of the group `programmer`). To make the file `/home/Alice/product_specs.txt` available to all programmers, one could use the command `chgrp programmer /home/Alice/product_specs.txt`. As long as the file has group read privileges, all users within the `programmer` group will have read privileges for the file. The results of such a command are shown in Table 1-5, which contains a third row that specifies that any user with the programmer role can read the file `/home/Alice/product_specs.txt`.

Table 1-5. *The ACL Based on the RBAC Model*

Role	Resource	Privilege
Backup Operator	`/home/*`	Read
Administrator	`/*`	Read, write, execute
Programmer	`/home/Alice/product_specs.txt`	Read

■ **Note** Our illustrations of various types of access control models using UNIX have been shown for conceptual clarity only. Various implementations of UNIX may implement ACLs using different data structures than in the tables we have used.

Now that we have summarized the three different types of access control models, we will examine an access control model called the Bell-LaPadula model. The Bell-LaPadula model can be used to implement either a mandatory or discretionary access model, depending upon the particular details of the implementation.

1.3.3. The Bell-LaPadula Model

The *Bell-LaPadula* model is a popular access control model used by many government and military organizations. In this model, all resources within the system are classified with a certain level of access. The classifications are, in order of increasing privilege: unclassified, confidential, secret, and top secret, as shown in Figure 1-2. In addition to associating a classification with resources, all users are also given a classification (unclassified, confidential, secret, or top secret).

The key innovation in the Bell-LaPadula model is not the idea of adding classifications to users and resources, it is the use of various rules used to guide the decisions about who is allowed to access the resources. There are three rules that guide the decisions about which users are allowed to access which files: the simple property, the star property, and the tranquility property.

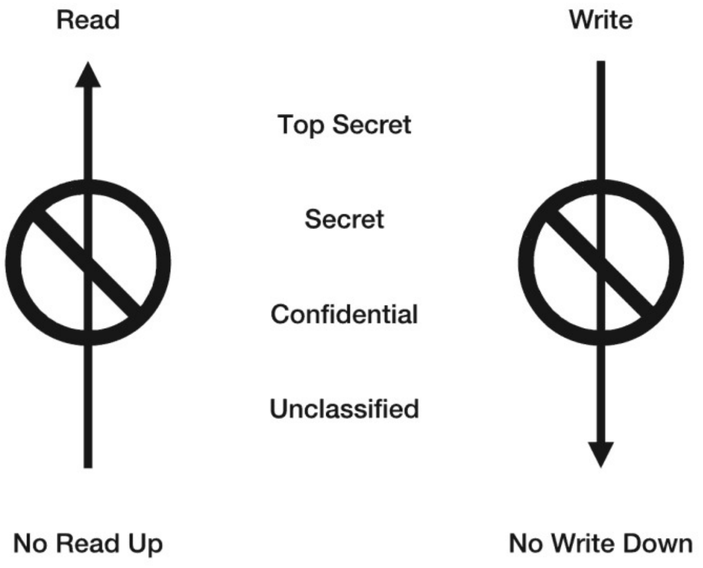

Figure 1-2. *The Bell-LaPadula model*

The first rule, the *simple property*, states that if a user has a particular level of access, then that user is not allowed to access any information resources that have a higher classification than the user does. In essence, a user that has only unclassified access will only be able to access unclassified files. A user with confidential access will be able to access confidential and unclassified files, but not secret or top secret files. The simple property is an intuitive rule that is very often called *no read up*.

The *star property*, also called the confinement property, is the second rule. If a user has secret level access, then the user is not allowed to write any files or create any resources that have a lower level of access. For example, if a user logs into a system and has secret level access, that user is not allowed to write any files that would be accessible by someone with only confidential or unclassified access. The idea behind this *no write down* strategy is that we would not want any information to leak from a higher level to a lower level. With this strategy, it would be impossible for someone with secret level access to write out any file in a system that could be read by a user that has only unclassified or confidential access. The goal of the star property is to restrict secret-level information only to the appropriate level of classification or above.

The third property of the Bell-LaPadula model is the *tranquility property*. The tranquility property states that the classification of a file cannot be changed while that file is in use by any user of the system. (The file is not considered to be tranquil while it is being edited or written.) For example, if the status of a confidential file is to be changed to unclassified, one has to wait until all users currently using (and potentially writing to) that file to stop using it. The reason to wait for tranquility is that it may be possible that some document could get declassified while some user with confidential access is still writing confidential information into the document. The tranquility property is a synchronization constraint placed upon all the resources in the system that uses the Bell-LaPadula model.

1.4. Confidentiality

The goal of *confidentiality* is to keep the contents of a transient communication or data on temporary or persistent storage secret.

If Alice and Bob want to exchange some information that they do not want Eve to see, the challenge is to make sure that Eve is not able to understand that information, even if Eve can see the bits that are being transferred over the network.

Suppose Eve is an eavesdropper who may be able to listen in on the contents of Alice and Bob's secret conversations. If Alice and Bob are communicating over a network, then Eve is able to see the bits—the zeros and ones—that make up Alice and Bob's conversation go back and forth over the wires (or over the air, in the case Alice and Bob are using a wireless network).

A real-world Eve might employ various existing software tools to eavesdrop. On an Ethernet network that uses a hub (as opposed to a switch), for instance, each computer is capable of actually seeing all the network traffic that is generated and received by any other computer. A computer's operating system is typically responsible for only allowing applications running on that computer to access traffic that is directed to or from that computer, and filtering out traffic that originates or is destined for other computers on the same network. However, if a user has root or administrator privileges on a computer, that user can use a software package such as Ethereal, tcpdump, or dsniff to access network traffic. These software packages are run in a "promiscuous mode," in which the operating system provides the software access to *all* traffic on the network instead of providing filtered traffic that is just directed to or from the computer on which it is running. While such packages exist to help network administrators and engineers debug problems, they can be used for eavesdropping. Attackers may not have administrator privileges, but can obtain them by first getting access to *some* account, and then exploiting software vulnerabilities in the operating system to gain such privileges.

Usually, some kind of encryption technology is used to achieve confidentiality. Most encryption technologies use a key to encrypt the communication between Alice and Bob. A *key* is a secret sequence of bits that Alice and Bob know (or share) that is not known to potential attackers.[4] A key may be derived from a password that is known to both Alice and Bob. An encryption algorithm will take the key as input, in addition to the message that Alice wants to transfer to Bob, and will scramble the message in a way that is mathematically dependent on the key. The message is scrambled such that when Eve sees the scrambled communication, she will not be able to understand its contents. Bob can use the key to unscramble the message by computing the mathematical inverse of the encryption algorithm. If Alice and Bob use good encryption technology and keep the key secret, then Eve will not be able to understand their communication.

4. In this chapter, we use the term *key* to refer to a secret key. In some encryption schemes (covered in Chapter 13), some keys can be made public.

1.5. Message/Data Integrity

When Alice and Bob exchange messages, they do not want a third party such as Mallory to be able to modify the contents of their messages.

Mallory has capabilities similar to Eve, but Eve is a passive eavesdropper while Mallory is an active eavesdropper. Though Eve is able to see the zeros and ones go by, she is unable to modify them. Eve therefore cannot modify any part of the conversation. On the other hand, Mallory has the ability to modify, inject, or delete the zeros and ones, and thus change the contents of the conversation—a potentially more significant kind of attack. Mallory is sometimes referred to as a *man in the middle*.

Alice and Bob can use an *integrity check* to detect if an active eavesdropper like Mallory has modified the messages in an attempt to corrupt or disrupt their conversation. That is, Alice and Bob want to protect the *message integrity* of their conversation. One approach that they can take to ensure message integrity is to add redundancy to their messages.

Consider a hypothetical scenario in which Alice wants to send an "I owe you" (IOU) message such as "`I, Alice, owe you, Bob, $1.00`," and Mallory has the ability to change only one character in the message. If Mallory wants Alice to be in more debt to Bob, she could change the message to "`I, Alice, owe you, Bob, $1000`" by changing the dot to a zero. On the other hand, if Mallory wants to cheat Bob out of his dollar, she could change the message to "`I, Alice, owe you, Bob, $0.00`." Assuming Mallory can only change a single character in a message, Alice could add redundancy to her message by repeating the dollar amount twice so that Bob could detect tampering. For example, if Alice sends the message "`I, Alice, owe you, Bob, $1.00. Confirm, $1.00`," then Mallory would not be able to change both of the dollar values in the message, and Bob would be able to detect tampering by Mallory. If Mallory changes one of the amounts in the message, Bob will see a mismatch between the two dollar amounts and discard the message. In this manner, redundancy can be used to provide message integrity.

While Mallory may not be able to tamper with Alice's IOU if she uses redundancy, she may still be able to conduct a denial-of-service attack. If Mallory changes one of the dollar amounts in the IOU each time Alice tries to send it to Bob, and Bob is forced to discard the message each time because of the mismatched dollar amounts, Bob will never receive the IOU he rightly deserves! (Denial-of-service attacks are discussed further in Section 1.7.)

Unfortunately, a real-world active eavesdropper will typically have the power to change much more than a single character in a message, and the simple approach of repeating the dollar amount will not work. In addition, repeating information more than once requires extra communications bandwidth and is not terribly efficient.

In networking communications protocols, approaches such as CRCs (cyclic redundancy checks) can be used to achieve integrity and detect when bits in a message have been lost or altered due to inadvertent communications failures. These techniques compute short codes that are functions of the message being sent. Alice can attach a short code to the message such that if the message or code are modified, Bob can determine whether they were tampered with.

However, while CRCs are sufficient to detect inadvertent communications failures, they are typically not good enough to deal with adversaries such as Mallory. If Mallory knows that a CRC is being used, and she has no restrictions on how many bytes she can modify, she can also change the short code to match her modified message.

Instead, message authentication codes (MACs) are typically used to achieve message integrity in real-world security protocols. A MAC is not only a function of the message itself, but is also a function of a key known only to Alice and Bob, such that even if Mallory is able to modify the bytes of a message, she will not be able to appropriately modify the corresponding MAC. (MACs are covered in more detail in Chapter 15.)

While the goal in confidentiality is to make sure that the contents of Alice and Bob's communication cannot be understood by a third party like Eve or Mallory, there is no such requirement for message integrity. For message integrity to be achieved, it does not matter whether the eavesdropper can see the data in the message so long as she is unable to change it undetected. The goal of message integrity is to make sure that even if Mallory can "look," she cannot "touch" the contents of the message.

1.6. Accountability

While authentication and authorization are important, *accountability* is another key security goal (especially for a company's internal systems). The goal of accountability is to ensure that you are able to determine who the attacker or principal is in the case that something goes wrong or an erroneous transaction is identified. In the case of a malicious incident, you want to be able to prosecute and prove that the attacker conducted illegitimate actions. In the case of an erroneous transaction, you want to identify which principal made the mistake. Most computer systems achieve accountability through authentication and the use of logging and audit trails. To obtain accountability, you can have a system write log entries every time a user authenticates, and use the log to keep a list of all the actions that the user conducted.

The chief financial officer (CFO) of a company may have the authority to transfer money from the company's bank account to any another, but you want to hold the CFO accountable for any actions that could be carried out under her authority. The CFO should have the ability to transfer money from the company account to other accounts because the company may have certain financial commitments to creditors, vendors, or investors, and part of the CFO's job may involve satisfying those commitments. Yet, the CFO could abuse that capability. Suppose the CFO, after logging into the system, decides to transfer some money from the company's bank account to her own personal account, and then leave the country. When the missing funds are discovered, the system log can help you ascertain whether or not it was the CFO who abused her privileges. Such a system log could even potentially be used as evidence in a court of law.

It is also crucial to make sure that when the logging is done and audit trails are kept, the logs cannot be deleted or modified after the fact. For example, you would not want the CFO to be able to transfer money into her own personal account and then delete or change the audit trail so that transaction no longer appears, or is covered up in any way to appear as if the transaction had a different recipient. To prevent logs from being deleted or altered, they could immediately be transferred to another system that hopefully an attacker would not be able to access as easily. Also, Chapter 15 discusses how MACs (message authentication codes) can be used to construct integrity check tokens that can either be added to each entry of a log or associated with an entire log file to allow you to detect any potential modifications to the system log. You can also use *write once, read many* (WORM) media to store system logs, since once written, these logs may be hard (or even physically impossible) to modify—short of destroying the media completely.

A good logging or audit trail facility also provides for accurate timestamping. When actions are written to an entry in a log, the part of the entry that contains the time and date at which the action occurred is called a *timestamp*. You need to ensure that no user can modify timestamps recorded in the log. The operating system, together with all the other computers on the network, must be in agreement on the current time. Otherwise, an attacker can log into a computer whose clock is ahead or behind the real time to cause confusion about when certain actions actually occurred. A protocol such as Network Time Protocol (NTP) can be used to keep the clocks of multiple computers synchronized.

One problem with many of today's systems is that logging facilities do not have secure timestamping and integrity checking facilities. As a result, after attackers hack into a system, they can change the logs such that no one can detect that they hacked in. Therefore, it is especially important to think carefully about a secure audit trail facility when you design secure systems. If existing or third-party software tools are used when constructing systems, they may have to be instrumented or modified to satisfy accountability goals.

1.7. Availability

An *available* system is one that can respond to its users' requests in a reasonable timeframe. While availability is typically thought of as a performance goal, it can also be thought of as a security goal. If an attacker is able to make a system unavailable, a company may lose its ability to earn revenue. For example, if an online bookstore's web site is attacked, and legitimate customers are unable to make purchases, the company will lose revenue. An attacker that is interested in reducing the availability of a system typically launches a denial-of-service (DoS) attack. If the online bookstore web site were run on a single web server, and an attacker transmitted data to the web server to cause it to crash, it would result in a DoS attack in which legitimate customers would be unable to make purchases until the web server was started again. Most web sites are not run using just a single web server, but even multiple web servers running a web site can be vulnerable to an attack against availability.

In a *distributed denial-of-service* (DDoS) attack, perpetrators commandeer weakly protected personal computers and install malicious software (malware) on them that sends excessive amounts of network traffic to the victim web sites.[5] The servers running the victim web sites are then overwhelmed with the large number of packets arriving from the commandeered computers, and are unable to respond to legitimate users.

In February 2000, the eBay, E*TRADE, Amazon, CNN, and Yahoo web sites were victims of DDoS attacks, and some were disabled for almost an entire business day. This meant lost revenues and interruption of service for legitimate users. One study by the Yankee Group estimated the damage due to lost capitalization, lost revenues, and cost of security upgrades to be $1.2 billion (Kovar 2000); this cost figure was also cited in a FBI congressional statement on cybercrime (Gonzalez 2000).

We include availability as a security goal because it is sometimes difficult to provide a system that is both highly secure and available all the time. There is sometimes an interesting trade-off between availability and security. For example, if a computer is disconnected from

5. Such attacks are called network-layer denial-of-service attacks. Application-layer denial-of-service attacks are also possible, in which vulnerabilities in applications are exploited to make systems unavailable.

the Internet and stored in a physically secure location where no one is allowed to access it, the computer will be very secure. The problem is that such a computer is not readily available to anyone for use.

You want to design systems whose functionality is available to the largest possible intended audience while being as secure as possible. A service like PayPal (`www.paypal.com`), which supports person-to-person payments, is an example of a system that generates more revenue the more users take advantage of it, and as such, its availability is critical—users may get very upset if they cannot access their funds at a moment's notice.

How does one achieve availability in a system? One method is to add redundancy to eliminate any single point of failure. For example, consider a telephone network. In such a network, phones connect to a switch (central office) that directs calls. If someone wants to attack your ability to place phone calls, he might cut the telephone line that connects to that particular central office, and as a result you would not be able to make calls. Attackers sometimes cut off a victim's ability to communicate prior to launching an attack.

One potential way to avoid single points of failure is to add redundancy. (Note that we are referring to a different type of redundancy than the redundancy we referred to in our discussion of message integrity.) A second switch can be added to the network so that if an attacker disables the first switch, the system will automatically connect you to the second.

Another potential DoS attack can be conducted by filling up a system's disk. Suppose users are sharing a disk on a server that is used to store their photos. That server may be running critical processes that need some disk space themselves. If an attacker can sign up as a user (or compromise an existing account) and fill up the shared disk with his own photos (or garbage data), then the critical processes may not be able to properly function, and system failure may ensue.

If you impose limits on the amount of disk space that each user can use, then even if the attacker is able to compromise one user's account, he will only be able to use up a certain amount of disk space. The attacker would need to compromise additional accounts to use up more disk space. In such a system, even if a user is a legitimate, paying customer, that user should not be trusted with more than her fair share of disk space because her account could be compromised.

Now that we have covered availability, let us move on to the last key security goal we consider in this chapter: non-repudiation.

1.8. Non-repudiation

The goal of *non-repudiation* is to ensure undeniability of a transaction by any of the parties involved. A trusted third party, such as Trent, can be used to accomplish this.

For example, let us say Alice interacted with Bob at some point, and she does not want Bob to deny that she interacted with him. Alice wants to prove to some trusted third party (i.e., Trent) that she did communicate with Bob. If, for instance, Alice sent a payment for a bill to Bob over the Web, she may want her payment to be non-repudiable. That is, she does not want Bob to be able to deny that he received the payment at some later point for any reason.

Alice, for example, may feel comfortable sending money to Trent, but not directly to Bob. Bob also trusts Trent. Trent may say to Bob, "Yes, Alice gave me the $500, so you can ship her the goods, and then I will pay you." In such an example, Trent is playing the role of an escrow agent, but trusted third parties may be able to serve in many other types of trusted roles beyond being escrow agents. Because Alice and Bob trust Trent, they may be able to conduct certain types of transactions that they could not have accomplished otherwise.

To illustrate another example in which Alice and Bob use the help of Trent, consider that Alice might want to sign a contract to be employed by Bob. Alice might want Trent to serve as a judge so that if Bob ever tries to pay her less than the salary specified by the contract, she can call on Trent to help enforce the contract. At the same time, Bob might not want Alice to show the employment contract to another potential employer to try to get a higher offer.

Alice and Bob can accomplish both of their goals by using Trent's help. Bob can give Trent the employment contract. Trent tells Alice the amount of the offer, and agrees not to show the employment contract to other employers. Then, Alice can decide whether to accept the contract, but will not be able to use it to negotiate higher offers with other employers. Also, if Bob ever tries to cheat Alice by not issuing payment, Trent can intervene. Note that we assume that Trent is trusted to be impartial and will not collude with either Alice or Bob. To summarize, trusted third parties can help conduct non-repudiable transactions.

In general, non-repudiation protocols in the world of security are used to ensure that two parties cannot deny that they interacted with each other. In most non-repudiation protocols, as Alice and Bob interact, various sets of evidence, such as receipts, are generated. The receipts can be digitally signed statements that can be shown to Trent to prove that a transaction took place.

Unfortunately, while non-repudiation protocols sound desirable in theory, they end up being very expensive to implement, and are not used often in practice.

1.9. Concepts at Work

Now that we have covered a number of key security concepts, let us examine how those concepts work together in a typical web client/web server interaction. Suppose Alice is an employee of a company called PCs-R-Us, and her job responsibility is to order DVD drives for the company's PCs from a company called DVD-Factory. DVD-Factory has a web site that Alice uses to procure DVDs for her company. The following points examine why DVD-Factory might want to care about the security goals discussed in this chapter when implementing its web site.

- *Authentication*: If a malicious competitor is trying to steal business from DVD-Factory, the competitor could create a web site that looks exactly like the DVD-Factory web site, but at a different web address. To combat that tactic, DVD-Factory needs to make sure that the web server can be authenticated so that when Alice goes to the DVD-Factory web site, she knows she is dealing with DVD-Factory and not DVD-Factory's look-alike competitor.

  The SSL protocol is used between web clients and web servers to do secure transactions. When Alice enters the web address `https://www.dvd-factory.biz`, Alice's browser will invoke the SSL protocol to make sure that the web site authenticates itself to her browser. (We will talk more about how the SSL protocol accomplishes this later in the book, but at this point, it is important to note that the web browser authenticates the web site to make sure that it is dealing with DVD-Factory's web site and not another web site that is trying to spoof or imitate it.)

Alice then has to log into DVD-Factory and give that web site her username and password so that DVD-Factory knows that it is Alice, an authenticated principal at PCs-R-Us, who is attempting to buy DVDs from them.

- *Authorization*: While Alice is the trusted PCs-R-Us employee to order DVDs, Bob, another employee at PCs-R-Us, might be responsible for accounting and auditing. He might also have a login and password to the DVD-Factory web site because he needs to see prices and orders placed, but he may not be allowed to place orders for DVDs himself. Before accepting an order, the DVD-Factory web site conducts an authorization check to make sure that the logged-in user is allowed to place an order. If Alice tries to order DVDs from PCs-R-Us, the web site will allow it, but if Bob attempts to order DVDs, the order will be rejected.

- *Confidentiality*: DVD-Factory doesn't want competitors to be able to see exactly how many or which DVDs Alice happens to be ordering from DVD-Factory, because that may give them competitive information. The SSL protocol encrypts all of the communication between Alice and the DVD-Factory web site with an algorithm such as Triple DES. (We cover SSL in more detail in Chapter 15, and we cover Triple DES and other encryption algorithms in Chapters 12 and 13.)

- *Message Integrity*: Suppose that Alice wants to order ten DVDs from DVD-Factory, but an attacker wants to alter her order to zero DVDs. If the attacker succeeds and DVD-Factory gets a message saying that Alice has ordered zero DVDs, her job may be affected, since no DVDs are actually going to be shipped. Alice may eventually get frustrated with DVD-Factory and might decide to go to a competitor (who may be behind this mischief). Message and data integrity are very important to prevent such mischief. The SSL protocol uses message authentication codes in the messages that are sent between Alice and the web site to make sure that no competitor or other malicious party can tamper with the data.

- *Availability*: DVD-Factory may have a competitor that launches a DoS attack against the site in order that Alice will stop buying from DVD-Factory and instead come to their competing site. As part of DVD-Factory's security strategy, its web site needs to be kept running and available 24 hours a day, 7 days a week. One simple (but potentially expensive) approach that DVD-Factory might use to mitigate a DoS attack against it would be to overprovision their bandwidth to handle the increased traffic load caused by illegitimate clients. You can read more about overprovisioning and other approaches to mitigating DoS attacks in *Internet Denial of Service Attack and Defense Mechanisms*, by Jelena Mirkovic et al.

- *Accountability*: To ensure accountability, every time Alice places an order from the DVD-Factory web site, it produces a log entry so that Alice cannot later claim to have not ordered the DVDs. This may sound a bit like non-repudiation, but it is actually accountability, since the goal is to simply keep a log of what Alice has and has not done.

- *Non-repudiation*: It is possible for DVD-Factory to cheat and report that Alice ordered more DVDs than she actually did. If the web browser and web site run a non-repudiation protocol, it is then possible for Alice to prove to a third party that she only ordered, say, 10 DVDs, and not the 12 that DVD-Factory may claim she ordered.

Unfortunately, true non-repudiation is not provided by SSL, and is not implemented on most web sites, partially due to the absence of a practical protocol for non-repudiation, and partially because there are no organizations to serve as the trusted third parties.

In practice, when customers pay for services with credit cards on web sites, Visa and Mastercard take on the role of trusted third parties, but they usually end up trusting their users much more than the merchant web sites from which their users buy products. If a user claims that he or she did not place an order with a merchant, then the credit card company favors the user and issues a *chargeback*. In the physical world, merchants can fight the chargeback if they can produce a receipt that the user signed. Of course, in the context of a web transaction, there is no good proxy or replacement for a receipt signed by the user!

While we unfortunately do not see true non-repudiation on the Web today, it is possible that the Web of the future will provide better non-repudiation capabilities.

CHAPTER 2

Secure Systems Design

This chapter examines how to architect and design systems that accomplish the security goals covered in Chapter 1. We first spend some time discussing prototypical threats to software, and then discuss how to design security into applications from the beginning. We focus on a number of high-level approaches and trade-offs, and discuss how security is sometimes perceived to be at odds with factors such as convenience and usability. We also discuss the concept of "security by obscurity" and why it is usually not sufficient. We look at security as a game of economics and risk management. Some of the approaches and design principles we cover in this chapter and the next were for the first time described in Jerome Saltzer and Michael Schroeder's paper, "The Protection of Information in Computer Systems"—we bring them to life and illustrate them with many real-world examples.

We also illustrate the approaches, trade-offs, and security design principles using a concrete, running code example throughout this chapter and the next. While most security books only talk about these principles in the abstract, we present actual code examples for a simple, small web server, and show specifically how it can be exploited by an attacker if security design principles are not followed. The code is written in the Java programming language, but we explain each line of code such that programmers of any language should be able to understand how the principles apply in their favorite programming language. The code examples can be downloaded from `www.learnsecurity.com/ntk`.

2.1. Understanding Threats

As new businesses take shape, new threats need to be identified and mitigated to allow for the continued success of those businesses. Over time, new businesses can use additional security technology to mitigate such threats. As your organization enters new businesses, it may be worthwhile to consider developing, buying, and deploying new technological solutions that help mitigate threats that did not exist prior to the organization's entry into that new business.

Different types of businesses will be more sensitive to different threats, and will have different security goals to mitigate those threats. Understanding threats is important in determining a system's security goals.

In the following section, we describe some sample threats and types of attacks to give you a flavor of some prototypical applications and threats they may face. Of course, keep in mind that there are many more types of computer security threats and attack types than those we list here.

2.1.1. Defacement

Consider what might be the most significant types of threats to a civil liberties web site or the White House web site. Since these web sites are created by organizations that advocate a particular political stance, an attacker is probably interested in making some kind of political statement against these organizations. Therefore, the most significant threat against such sites may be defacement.

Defacement is a form of online vandalism in which attackers replace legitimate pages of an organization's web site with illegitimate ones. In the years 1999 and 2001, for example, the White House web site was defaced by supposed anti-NATO activists (Dennis and Gold 1999) and Chinese hackers (Anderson 2001). In such defacement attacks, the attackers usually replace the front page of a web site with one of their own choice.

Defacement is a very different type of threat than what other web sites, such as financial institutions or e-commerce vendors, might face. The attackers of these web sites may be most interested in compromising bank accounts or conducting credit card fraud. Therefore, how we design systems to be secure against attacks is dependent on the type of threats that we expect them to face.

In the case of a politically oriented web site, say, `www.whitehouse.gov`, there may be a database where all of the content for that web site is stored. The owner of the web site may not care if an attacker gains read-only access to the information in that database—however, they do not want the attacker changing the information in that database. On the other hand, a financial institution or e-commerce web site does not want the attacker to be able to even read the information in the back-end database. If this happened, the credit card or account numbers of clients might be compromised.

2.1.2. Infiltration

In general, infiltration is an attack in which an unauthorized party gains full access to the resources of a computer system (including, but not limited to, use of the CPUs, disks, and network bandwidth). In later chapters, we study how buffer overflow, command injection, and other software vulnerabilities can be used by attackers to infiltrate and "own" computers.

In some defacement attacks, an attacker may have to infiltrate a web server to conduct the defacement. But the threat from infiltration can be quite different than that of defacement, depending on the type of web site. Consider the threat from an infiltration in which an attacker is able to write to a database running behind, say, a financial web site, but not be able to read its contents. If the attacker is able to write information to the database without reading it, the situation might not be as bad as you might think. So long as you can detect that the attacker's write took place, the situation can be mitigated. You can always restore the correct account numbers and balances from a backup database, and redo all transactions that occurred after the unauthorized writes to prevent your users from being affected. (For the purposes of this example, we assume that even if an attacker is able to write the database content, the attacker would not be able to rewrite logs. In the real world, attackers can sometimes also rewrite logs, which presents greater problems.) So, in the case of the political web site, you most importantly need to defend against an attacker who attempts to gain write capability, while in the case of a financial web site, it is most important to defend against an attacker who attempts to gain read capability.

The preceding example illustrates that different types of web sites are going to have different security goals. In the case of a political web site, the integrity of the web site content is the most significant concern, while in the case of a financial web site, integrity and confidentiality of customer data are both of high importance.

Military web sites have still different security sensitivities. If a military web site is defaced, it might simply be embarrassing for them. Infiltration of a military web site, in which confidential or classified data is acquired by the attacker, however, could be a threat to national security.

2.1.3. Phishing

Phishing is an attack in which an attacker (in this case, a *phisher*) sets up a spoofed web site that looks similar to a legitimate web site. The attacker then attempts to lure victims to the spoofed web site and enter their login credentials, such as their usernames and passwords. In a phishing attack, attackers typically lure users to the spoofed web site by sending them e-mails suggesting that there is some problem with their account, and that the user should click a link within the e-mail to "verify" their account information. The link included in the e-mail, of course, is to the attacker's web site, not the legitimate site. When unsuspecting users click the link, they arrive at the spoofed site and enter their login credentials. The site simply logs the credentials, and either reports an error to the user or redirects the user to the legitimate site (or both). The attacker later uses the logged credentials to log into the user's account and transfer money from the user's account to their own.

Why do users fall for clicking such links in e-mails sent by phishers? Phishers use various techniques to hide the fact that the link is to their illegitimate, spoofed site. Following is an example.

First, in HTML documents, a link is constructed as follows:

```
<A HREF='http://www.destination-site.com/'>
Click here
</A>
```

When the e-mail is rendered by a browser, the link will look like this: `Click here`, and the destination address will not be apparent to an unsuspecting user.

An attacker can use code such as the following in an HTML e-mail sent to the victim:

```
<A HREF=http://www.evil-site.com/>
http://www.legitimate-site.com/
</A>
```

The browser displays `http://www.legitimate-site.com/`, but when the user clicks the link, the browser loads the front page of `www.evil-site.com` since that is what is specified by the hyperlink reference (`HREF`) in the anchor (`A`) tag in the HTML e-mail. In real phishing attacks, the phisher might have the browser display `www.paypal.com` or `www.google.com`, and have the hyperlink reference point to `www.paypa1.com` (with a "1" instead of a "l") or `www.gogole.com` ("google" misspelled), respectively.

Slightly more sophisticated users may position their mouse over the link prior to clicking it. Many browsers will display the address of the destination site at the bottom of the browser window or in a pop-up tool tip. Such users may decide not to click the link if the actual destination site does not match their expectation.

2.1.4. Pharming

Pharming is another attack in which a user can be fooled into entering sensitive data into a spoofed web site. It is different than phishing in that the attacker does not have to rely on the user clicking a link in an e-mail. With pharming, even if the user correctly enters a URL (uniform resource locator)—or web address—into a browser's address bar, the attacker can still redirect the user to a malicious web site.

When a user enters a URL—say, `www.google.com/index.html`—the browser needs to first figure out the IP address of the machine to which to connect. It extracts the domain name, `www.google.com`, from the URL, and sends the domain name to a domain name server (DNS). The DNS is responsible for translating the domain name to an IP address. The browser then connects to the IP address returned by the DNS and issues an HTTP request for `index.html`.

In a pharming attack, an attacker interferes with the machine name–to–IP address translation for which the DNS is responsible. The attacker can do so by, for instance, compromising the DNS server, and coaxing it into returning the attacker's IP address instead of the legitimate one. If the user is browsing via HTTP, the attack can be unnoticeable to the user. However, if a user connects to a site using SSL, a pharming attack (in most cases) will result in a dialog box from the browser complaining that it was not able to authenticate the server due to a "certificate mismatch." (We discuss certificates in Section 15.3.)

PHARMING (A.K.A. DNS CACHE POISONING)

While the term *pharming* was coined in March 2005 shortly after a significant attack, this type of attack has been known for years prior under the name *DNS cache poisoning*. However, due to the increasing use of the Internet to conduct financial transactions, DNS cache poisoning is no longer just a matter of academic interest—criminals have turned to it for financial gain.

2.1.5. Insider Threats

A surprisingly large percentage of attacks take place with the cooperation of insiders. Insiders could be, for instance, employees at a corporation who abuse their privileges to carry out malicious deeds. Employees are sometimes trusted with access to databases with customer information and employee records, copies of financial reports, or confidential information concerning product launches. Such information can be abused in the obvious ways: employee data could be sold to headhunters, customer credit card numbers could be sold on the black market, financial reports could facilitate insider trading, and product launches could be leaked to the press.

As such, it is sometimes important to defend a system against the very people that are responsible for using it on a daily basis. Database administrators, for example, have traditionally been given the "keys to the entire kingdom," and have complete access to all employee and customer data stored in a database. System administrators similarly are given "superuser" access to all resources and data under the control of an operating system. Additional features are needed in both database and operating systems to provide for *separation of privilege*, the concept that an individual should only be given the privileges that he needs, without also being given unrestricted access to all data and resources in the system.

2.2.1. Windows 98

Problems occasionally arose in Windows 98 in which a diagnostic mode is needed to deal with the issue. For example, if some device driver locks up while the computer is booting up,[1] the Windows 98 diagnostic mode can be used to help determine which device drivers may be causing the problem. The Windows 98 diagnostic mode does not load up all device drivers at boot time.

You can access the diagnostic mode in Windows 98 by pressing the F8 key while the operating system is booting up. Even if a computer is password protected, anyone can hit the F8 key at boot time, and the computer will jump into its diagnostic mode, giving the user the ability to access the hard disk and any sensitive data on it without entering a username or password.

In Windows 98, the security feature of entering a username and password was added into the operating system as an afterthought, as opposed to being part of the initial design of the operating system. If instead the operating system was designed with security in mind, then it might ask a user to enter a username and password to even enter the diagnostic mode. The design of the Windows 98 password mechanism is an example of how adding security as an afterthought does not work.

2.2.2. The Internet

Another example of how it is very difficult to add security as an afterthought is the design of the Internet itself. When the Internet was designed, all of the hosts (computers) on the network were effectively trusted because they were owned by universities or military installations that trusted each other and wanted to collaborate with one another. (The Internet grew out of a government project funded by DARPA, the Defense Advanced Research Project Agency.) In the mid-1990s, due to the mass commercialization of the Internet, just about everyone started connecting their computers to the Internet. New hosts were allowed to connect to the existing hosts regardless of whether the existing parties on the network trusted the newly connected hosts. To protect themselves, some hosts started deploying firewalls.

Firewalls and Their Limitations

A *firewall* allows hosts to specify that they trust some hosts to connect to them on some ports while they do not trust other hosts or accept traffic on other ports. However, due to the way that the Internet was designed, firewalls are not always necessarily able to enforce the trust relationships their users would like, since hosts can lie about IP addresses or communicate over ports that have been cleared to go through the firewall.

Consider two hosts, Alice (A) and Bob (B). For A to send a message to B, A needs to construct an Internet Protocol (IP) packet. You can think of an IP packet as a letter that one might send in the mail, except that it is transmitted over a wire instead of dropped into a mailbox. The data packet has two parts: an "envelope," sometimes referred to as the IP header, and the message itself. The IP header contains host B's IP address (the destination address). The message contains the data that A would like to send to B.

1. A device driver is a piece of software that allows a computer to use a hardware device that is attached to it. Device drivers are responsible for communicating between the operating system running on a computer and the hardware device that is attached to it. Device drivers are notorious for bugs that can cause an operating system to malfunction or crash altogether.

Internet *ports* are used to route messages to applications on hosts, using, for instance, Transmission Control Protocol (TCP). By convention, applications of different types communicate over different ports. For instance, web browsers typically communicate with web servers over port 80, and mail clients send outbound e-mails to mail servers on port 25. (A listing of standard port assignments can be found at `www.iana.org/assignments/port-numbers`.)

However, port assignments are by convention only, and there is no fundamental reason that a web client couldn't connect to a web server listening on port 25, or that a mail server couldn't be run on port 80. Hence, firewalls sometimes cannot effectively control or restrict traffic successfully by just imposing rules on what port numbers applications can use to communicate.

Malicious hosts can easily "lie" about source addresses in IP packets. By taking advantage of low-level networking routines, an attacker can write code that fills in the source IP address field in outgoing IP packets instead of letting the operating system do it. Let Mallory (M) be a malicious host. Host M could, for instance, put host A's IP address on the data packet that it sends to B. When host B looks at the data packet, it will look like it came from A.

One can imagine that this capability can be used for nefarious purposes. Consider a scenario in which host B is storing all of Alice's files on its hard disk, and Alice issues commands to add, remove, and delete files every now and then from host A. Host B might even be configured to only process commands in data packets that have A's address as the source address. The practice of deciding to accept communications from another host based on the host's IP address is often called *IP whitelisting.*[2] Host B is said to store a *whitelist*, which is simply a list of IP addresses from which it will accept data packets.

Unfortunately, our malicious host M may be able to coerce host B into executing commands of its choice simply by putting host A's address on the envelope. When an Internet host intentionally mislabels the source address on data packets that it sends out, it is said to be conducting an *IP spoofing attack*. IP whitelisting, as a security mechanism, is potentially susceptible to IP spoofing attacks, especially when non-connection-oriented protocols such as UDP are used.

Host B will send the results of the commands it executes to the host specified as the source address. In our example, host B sends its response to host A, not host M. However, if the command is "delete all files on the hard disk," an attacker such as M does not need to receive the results of the command in order to cause damage.

IP spoofing is possible for an attacker even if more than one round of communication is necessary. Consider a scenario in which host B from the previous example sends a message that says "Are you sure you want to delete all files?" in response to receiving the "delete all files on the hard disk" command, and waits for a response prior to actually issuing the command. Now, one might hope that since host B would require an answer back from host A confirming the deletion, the attack could be foiled.

However, our attacker on host M will still be able to delete all the files on B's hard disk. One problem the attacker would need to solve is that when host A receives the "Are you sure you want to delete all files?" message, Alice may say "No" because she never sent the delete command to begin with (and she probably does not want all her files to be deleted). Indeed, if host A receives the "Are you sure you want to delete all files?" message, it might be an

2. Similarly, the practice of denying communications from another host based on the host's IP address is called *blacklisting*.

indication that there is an attack taking place. Alternatively, host A simply might not respond to the "Are you sure?" message at all.

If host A does not respond to the "Are you sure?" message, then host M could send a second spoofed packet sometime later with the answer "Yes." Since the source IP address would be A's address, host B would allow the command because host A's address is on its whitelist, and host B would proceed to delete all files on its hard disk.

So it is actually quite critical that host M makes sure that host A does not respond to the "Are you sure?" message. To ensure that host A does not respond, host M can start a DoS attack against host A prior to sending its "delete all files" message to host B. Hence, even though host B sends an "Are you sure?" message back to the source address to confirm the "delete" command, an IP spoofing attack is still possible.

To make the attack harder, host B can include a *nonce*, a pseudo-random number intended for one-time use, in the "Are you sure?" message. When host B receives the request to delete files, it responds to host A with a confirmation request of the form "Are you sure? Please echo the random number 3957264392047453759 back if you are sure." If host A indeed wanted to delete all files, it would respond with "Confirm delete all files—confirmation number 3957264392047453759." The point of the nonce is that since host M does not know it (M does not receive the confirmation request), our adversary would not be able to issue a successful confirmation to delete the files.

IP spoofing is much easier for non-connection-oriented protocols such as UDP than for connection-oriented protocols like TCP. TCP includes a sequence number in packets that is typically used to reorder packets if they arrive from the network out of order. However, if an attacker can successfully guess TCP sequence numbers, the attacker may be able to insert packets into a TCP conversation. When a TCP connection is established, the operating system chooses a TCP sequence number for the first packet in the conversation. If it does not do a good job choosing such a number at random, an attacker may be able to predict the next sequence number that will be used, and can use that information to set up a spoofed TCP connection.

If you have further interest in IP spoofing, you can read more about it in Robert Morris's paper, "A Weakness in the 4.2BSD UNIX TCP/IP Software" (Morris 1985) or the more recent article from Phrack magazine entitled "IP Spoofing Demystified" (daemon9, route, and infinity 1996).

The Adoption of IP

There is sometimes a natural trade-off between security and convenience. We discuss this trade-off more in Section 2.3, but we'll briefly point out that the adoption of IP is an example of it here. IP was very convenient to deploy, and, partially as a result, it received wide adoption. On the other hand, if it were more secure but less convenient to deploy, it may not have been adopted as quickly or as widely.

A protocol called IPsec was developed to require hosts to authenticate each other so that they cannot lie about their IP address or identity. Unfortunately, IPsec is not widely deployed on the public Internet.[3]

3. While IPsec is not used to establish connections between two arbitrary computers on the Internet, it is used to construct virtual private networks (VPNs). A corporate VPN may use IPsec to route encrypted and integrity-protected traffic over the public Internet between two authenticated hosts on its intranet.

If IP had been first designed to have authentication built in as part of the protocol (as in IPsec), the world might be very different today. It is possible that attackers would not have as much flexibility as they do today. At the same time, if IP had been more like IPsec, and required as much overhead to deploy, it is possible that it would not have had as wide an adoption as it did.

How do you achieve both security and adoption? The challenge is to design and use security mechanisms that are not too inconvenient, and that may even help serve as a reason for increased adoption.

2.2.3. Turtle Shell Architectures

When systems are not designed with security in mind, sometimes attempts are made to secure them with turtle shell architectures after the fact. A *turtle shell architecture* is one in which an inherently insecure system is "protected" by another system that attempts to mediate accesses to the insecure system.

Firewalls are such an example. Many corporations and organizations interested in securing their systems on the Internet deploy a firewall in front of their systems. The firewall creates a turtle shell architecture that attempts to guard soft, vulnerable software inside the corporation's network perimeter.

While it is, in general, useful to construct a hard outer shell, that outer shell should not be relied upon completely for defense. If there is a way to get through that shell or "turn the turtle over," the software running on hosts underneath the shell would be very susceptible to attack.

THE DEATH STAR'S FIREWALL

If you remember the first *Star Wars* movie, *A New Hope*, originally released in 1977, you may now realize that the Death Star had a turtle shell architecture. The Death Star was a moon-sized space battlestation that went around the galaxy destroying planets with good, happy people on them. The technical specifications of the Death Star that were stored by the droid R2-D2 revealed that it had "a strong outer defense" consisting of a magnetic shield and large, powerful turbo lasers mounted on the surface of the battlestation. The Death Star's defenses were geared at mitigating the threat posed to it by large space cruisers. However, the good guys in the movie—the rebels—were able to destroy the Death Star by piloting small, one-manned stunt fighters through the magnetic shield, evading the large, relatively slow-moving turbo lasers, and exploiting its weakness—a small, thermal exhaust port connected to the battlestation's power-generation system.

Think of a firewall as the Death Star's magnetic shield. It may help mitigate large, outright, blatant attacks. However, it is fairly useless against more stealthy attacks. A firewall can prevent incoming connections from particular hosts or IP addresses based on the information in packets. Of course, if an attacker can successfully spoof IP addresses, then a firewall may not be able to tell the difference between a packet that was sent from a legitimate host and one that was not.

To summarize, when the Internet was designed, security was not one of the design parameters. The same was true when Windows 98 was designed. When you design new software features, you should think about security up front—don't add it on as an afterthought.

2.3. Convenience and Security

Security comes at a price not only to the company that is developing an information system, but to the users of that system. The system may become less convenient for the users as more security technology is deployed. For example, if you allow your users to choose whatever password they like, this may lead to security vulnerabilities since some users may choose passwords that are easy for attackers to guess. On the other hand, if you deploy a security technology that assigns complicated passwords to users, your system may *seem* more secure, but it will be less convenient to your users, since they may forget the passwords if they're too complicated. We say "seem" more secure because if the passwords are so hard to remember that users start writing them down, this introduces another vulnerability that may end up actually decreasing the security of the overall system. If those written-down passwords are stored in a user's wallet with all of his other credentials, that would involve some risk; but if they're on a Post-it note stuck to the side of a monitor in a public office space, that would involve significantly more risk!

A good security technology can increase both convenience and security—although that may not always be possible. For example, if you allow users to choose their own passwords, but make them choose sufficiently complicated ones (e.g., require that users enter one digit or special character into a password that's between eight and ten characters), this might significantly increase security at the cost of only a little bit of inconvenience. A good security technology will provide a relative security benefit at only a slight inconvenience to users. A good technology will increase both convenience and security, because even if it introduces a slight inconvenience, it can reduce or eliminate more significant inconveniences (and damages) that may occur as the result of a successful attack.

2.4. SimpleWebServer Code Example

Now that we have covered some basics of how the Internet works, we will introduce our code example, a simple web server that we will be using to illustrate various security design concepts later in this chapter and the next.

Before we present the code for the simple web server, we'll briefly review the basics of how web servers work. We have intentionally simplified our explanation so that we can focus on only the essential details, so if you're a web veteran, please don't be alarmed at how many details we've omitted!

2.4.1. Hypertext Transfer Protocol (HTTP)

The World Wide Web (WWW), or Web for short, is made up of a network of Internet servers ("web servers") that serve Hypertext Markup Language (HTML) documents to web browser client applications. Web servers typically listen for connections coming from web browsers on port 80. After a web browser connects to a web server on port 80, it communicates with the web server using the Hypertext Transfer Protocol (HTTP). The first HTTP message that the browser sends to a server after connecting is typically of the following form:

```
GET <filename> <http-version>
```

```
                StringTokenizer st =
                        new StringTokenizer (request, " ");

                command = st.nextToken();
                pathname = st.nextToken();

                if (command.equals("GET")) {
                        /* If the request is a GET,
                           try to respond with the file
                           the user is requesting. */
                        serveFile (osw,pathname);
                }
                else {
                        /* If the request is a NOT a GET,
                           return an error saying this server
                           does not implement the requested command. */
                        osw.write ("HTTP/1.0 501 Not Implemented\n\n");
                }

                /* Close the connection to the client. */
                osw.close();
        }

        public void serveFile (OutputStreamWriter osw,
                               String pathname) throws Exception {
                FileReader fr = null;
                int c = -1;
                StringBuffer sb = new StringBuffer();

                /* Remove the initial slash at the beginning
                   of the pathname in the request. */
                if (pathname.charAt(0) == '/')
                        pathname = pathname.substring(1);

                /* If there was no filename specified by the
                   client, serve the "index.html" file. */
                if (pathname.equals(""))
                        pathname = "index.html";

                /* Try to open file specified by pathname. */
                try {
                        fr = new FileReader (pathname);
                        c = fr.read();
                }
                catch (Exception e) {
                        /* If the file is not found, return the
                           appropriate HTTP response code. */
                        osw.write ("HTTP/1.0 404 Not Found\n\n");
                        return;
                }

                /* If the requested file can be successfully opened
                   and read, then return an OK response code and
                   send the contents of the file. */
                osw.write ("HTTP/1.0 200 OK\n\n");
                while (c != -1) {
                        sb.append((char)c);
                        c = fr.read();
                }
                osw.write (sb.toString());
        }

        /* This method is called when the program is run from
           the command line. */
        public static void main (String argv[]) throws Exception {

                /* Create a SimpleWebServer object and run it. */
                SimpleWebServer sws = new SimpleWebServer();
                sws.run();
        }
}
```

Main Program

For those readers who are familiar with Java, the preceding program should seem very straightforward. We now provide a brief explanation of how the program works for the benefit of programmers who are not familiar with Java (or object-oriented programming or networking, for that matter).[4] In our explanation, we repeat relevant parts of the code so that you do not have to keep flipping pages. We start with the program's `main()` method:

```
        /* This method is called when the program is run from
           the command line. */
        public static void main (String argv[]) throws Exception {

                /* Create a SimpleWebServer object and run it. */
                SimpleWebServer sws = new SimpleWebServer();
                sws.run();
        }
}
```

4. If you are not so familiar with object-oriented programming, the term *method* used in the following discussion is almost equivalent to *function*.

When a Java program starts running, the code in its `main()` method is executed first. The `main()` method in the program creates a new `SimpleWebServer` object and calls its `run()` method. The `SimpleWebServer` object is a data structure that contains both the code and data that make up the web server. When the line containing "`new SimpleWebServer()`" executes, it invokes the constructor method. The constructor is simply a method that creates the `SimpleWebServer` object—it allocates memory for it and initializes the data used by the object.

Once the `SimpleWebServer` object is constructed and initialized, the `main()` method calls the `run()` method, which handles all the real work done by the server. The `run()` method consists of an infinite loop—`while (true)`—that waits for a connection from a client, and then attempts to process the client's request. The call to the `ServerSocket accept()` method returns a socket object that corresponds to a unique socket on the server and allows the server to communicate with the client.

Important Data Members

The `SimpleWebServer` object has two important pieces of data (also called data members), as shown here:

```
    /* Run the HTTP server on this TCP port. */
    private static final int PORT = 8080;

    /* The socket used to process incoming connections
       from web clients. */
    private static ServerSocket dServerSocket;
```

The first is the port number that the web server should listen to for connections from clients. The `PORT` variable is simply a constant that is initialized to 8080. (Typically, only system administrators are allowed to run programs that use ports less than 1024.) Usually, clients would be able to connect to the simple web server using a URL, or web address, such as `http://machinename.yourdomain.com/`. The browser automatically assumes port 80, but you can specify a different port, such as 8080, by appending a colon followed by the desired port number in the URL, `http://machinename.yourdomain.com:8080`.

The second important data member is `dServerSocket`. The `dServerSocket` data member is a socket to which clients can connect. Think of it as being like an electrical socket. Both web browser clients and web servers have a "virtual" power strip with many sockets on them. A client can talk to a server by selecting one of its own sockets, selecting one of the server's sockets, and establishing a connection between the two by plugging a virtual wire into each end. However, since we would not want each client to have to worry about choosing a unique port number on the server so that they don't interfere with each other, the `ServerSocket` object will take connections from many clients connecting to the same port number—in our case, 8080. When a client expresses its desire to connect to the port number, the `ServerSocket` object manages assigning each client some unique port from the server's frame of reference.

Processing Requests

We now describe the `processRequest()` method that is called once the client connects:

```
        /* Reads the HTTP request from the client and
           responds with the file the user requested or
           an HTTP error code. */
        public void processRequest(Socket s) throws Exception {
                /* Used to read data from the client. */
                BufferedReader br =
                        new BufferedReader (
                                new InputStreamReader (s.getInputStream()));

                /* Used to write data to the client. */
                OutputStreamWriter osw =
                        new OutputStreamWriter (s.getOutputStream());

                /* Read the HTTP request from the client. */
                String request = br.readLine();

                String command = null;
                String pathname = null;

                /* Parse the HTTP request. */
                StringTokenizer st =
                        new StringTokenizer (request, " ");

                command = st.nextToken();
                pathname = st.nextToken();

                if (command.equals("GET")) {
                        /* If the request is a GET,
                           try to respond with the file
                           the user is requesting. */
                        serveFile (osw,pathname);
                }
                else {
                        /* If the request is a NOT a GET,
                           return an error saying this server
                           does not implement the requested command. */
                        osw.write ("HTTP/1.0 501 Not Implemented\n\n");
                }

                /* Close the connection to the client. */
                osw.close();
         }
```

The `processRequest()` method takes the client socket as input. It uses the client socket to create the `BufferedReader` and `OutputStreamWriter` objects that allow it to read data from and send data to the client, respectively. Once these communication objects have been created, the `processRequest()` method attempts to read a line of input from the client using the `BufferedReader`. We expect that the first line of data that the client sends the server is an HTTP `GET` request, as described previously. The `StringTokenizer` object, `st`, is used to break up the request into its constituent parts: the command (i.e., `GET`) and the pathname to the file that the client would like to download. If the command is a `GET` request, as expected, the `serveFile()` method is called to load the file into the server's memory and send it to the client. If the command is not a `GET` request, an appropriate HTTP error response is sent to the client.

Serving Files

Once the `GET` request is parsed for the filename, the `serveFile()` method is used to retrieve the file from disk, and serve it to the client.

```
        public void serveFile (OutputStreamWriter osw,
                               String pathname) throws Exception {
                FileReader fr = null;
                int c = -1;
                StringBuffer sb = new StringBuffer();

                /* Remove the initial slash at the beginning
                   of the pathname in the request. */
                if (pathname.charAt(0) == '/')
                        pathname = pathname.substring(1);

                /* If there was no filename specified by the
                   client, serve the "index.html" file. */
                if (pathname.equals(""))
                        pathname = "index.html";

                /* Try to open file specified by pathname. */
                try {
                        fr = new FileReader (pathname);
                        c = fr.read();
                }
                catch (Exception e) {
                        /* If the file is not found, return the
                           appropriate HTTP response code. */
                        osw.write ("HTTP/1.0 404 Not Found\n\n");
                        return;
                }

                /* If the requested file can be successfully opened
                   and read, then return an OK response code and
                   send the contents of the file. */
                osw.write ("HTTP/1.0 200 OK\n\n");
                while (c != -1) {
```