THREAT: DATA CORRUPTION

This attack is really a nonattack. We do not know how well protected the student records are for the university involved, although most universities' records are safe.

Nevertheless, people often receive incorrect or corrupted data: a minor misspelling of a name, an obvious typographic error, a mistaken entry on a list. If you watch real-time closed-captioning on television, sometimes you see normal text degenerate to gibberish and then return to normal after a couple of minutes. We have become accustomed to errors such as the programming situations described in Chapter 3. Mistakes like this happen, and we either contact someone for a correction if the issue is serious, or ignore it otherwise. Errors occur so frequently that we sometimes fail even to notice them.

In Figure 16-1 we remind you of some of the sources of data corruption; we have previously described most of these causes. You should keep in mind that data corruption can be intentional or unintentional, from a malicious or nonmalicious source, and directed or accidental. Data corruption can occur during data entry, in storage, during use and computation, in transit, and on output and retrieval.

Sometimes modification is blatant, making it readily apparent that a change has occurred (for example, complete deletion, which could be detected by a program, or replacement of text by binary data, which would be apparent to a human reader). Other times the alteration is subtle, such as the change of a single bit, which might allow processing to continue, although perhaps producing incorrect results.

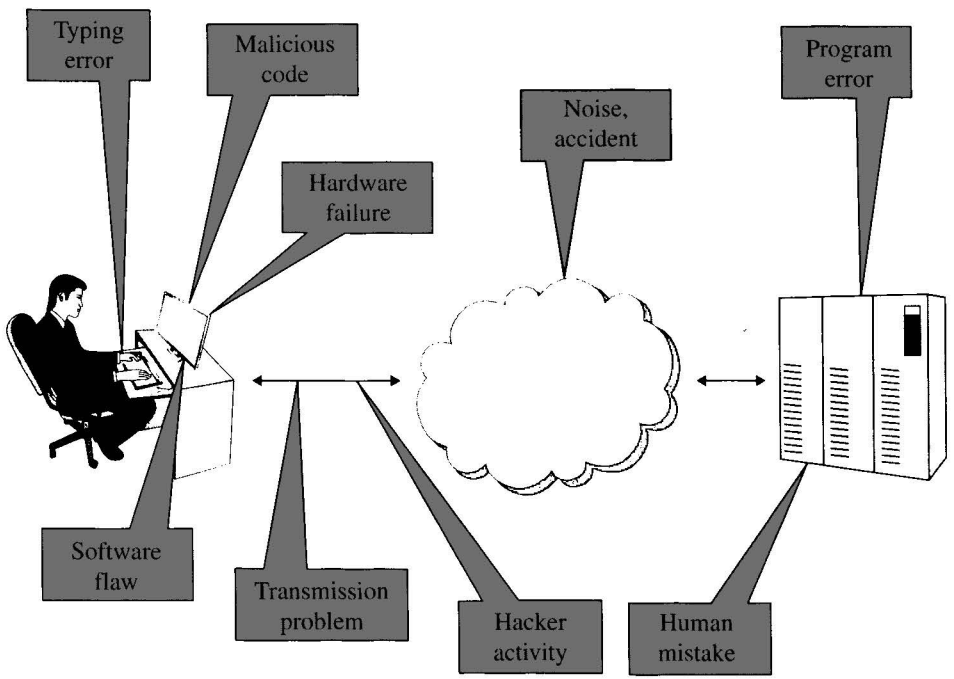

FIGURE 16-1 Data Corruption Sources

Another kind of modification attack involves the order of data: the data items are not changed, just rearranged. Such an attack can be either accidental or intentional, malicious or not. Encryption is effective for protecting against these attacks, too, although the encryption needs to be applied carefully to be effective. In this section we consider the issue of ordering of data.

Sequencing

A **sequencing** attack or problem involves permuting the order of data. Most commonly found in network communications, a sequencing error occurs when a later fragment of a data stream arrives before a previous one: packet 2 arrives before packet 1.

Sequencing errors are actually quite common in network traffic. Because data units are routed according to available routing information, when packet 1 is sent, the best route, which is the route chosen, goes via node C. Subsequently the router learns node C is no longer optimal, so when packet 2 is to be sent, the routing goes via node D. The second route is indeed superior, so much so that packet 2 arrives before packet 1. Congestion, network interference, faulty or failed equipment, and performance problems can easily cause these kinds of speed difficulties.

Network protocols such as the TCP suite ensure the proper ordering of traffic. However, application programs do not always detect or correct sequencing problems within the data stream. For example, if an application handles input from many concurrent clients at a shopping site, the application must ensure that individual orders are constructed correctly, regardless of the order in which the pieces of orders arrived.

Substitution

A **substitution** attack is the replacement of one piece of a data stream with another. Nonmalicious substitution can occur if a hardware or software malfunction causes two data streams to become tangled, such that a piece of one stream is exchanged with the other stream.

Substitution errors can occur with adjacent cables or multiplexed parallel communications in a network; occasionally, interference called crosstalk allows data to flow into an adjacent path. Metallic cable is more subject to crosstalk from adjacent cables than is optical fiber. Crossover in a multiplexed communication occurs if the separation between subchannels is inadequate. Such hardware-induced substitution is uncommon.

A malicious attacker can perform a substitution attack, similar to the replay attack described in Chapter 14, by splicing a piece from one communication into another. Thus, Amy might obtain two communications, one to transfer $100 to Amy, and a second to transfer $100,000 to Bill, and swap either the two amounts or the two destinations. Substitution attacks of this sort are easiest to carry out with formatted communications. If Amy knows, for example, which bytes represent the account number, she knows she need only exchange bytes 24–31 between these two messages.

Not all substitution attacks are malicious, as the example of Sidebar 16-1 describes.

Insertion

An **insertion** attack, which is almost a form of substitution, is one in which data values are inserted into a stream. As in Chapter 14, an attacker does not even need to break an

Substitute Donors **Sidebar 16-1**

The British National Health Service (NHS) maintains a database of potential organ donors in the United Kingdom. According to an article in *The Register* on April 12, 2010, the organ donor status field was incorrectly entered for people who registered their organ donation preferences while applying for a driver's license. Some 400,000 data fields were corrected by the NHS and another 300,000 people had to be contacted to determine the correct value.

According to a subsequent review [DUF10], the error arose in 1999 and went unnoticed from then until 2010. The NHS receives data from three sources: hospitals, doctors, and the driver's license office. When applying for a driver's license or registering with a doctor or hospital, an applicant can mark boxes specifying which organs, if any, the applicant wishes to donate after death. The record transmitted to NHS from any source contains identification data and a seven-digit number coded as 1 for no and 2 for yes. However, the order of the organs listed on the license application is different from the order the other two sources use, which was properly handled by software before 1999. As part of a software upgrade in 1999, all inputs were erroneously processed as if they were arranged in the same order.

The review after discovery of the error recommended enhanced testing procedures, notification of all affected parties whenever a programming change was to be implemented, and periodic auditing of the system, including sample record validation.

encryption scheme in order to insert authentic-seeming data, as long as the attacker knows precisely where to slip in the data.

Salami

With an interesting and apt name, a salami attack involves a different kind of substitution. In a financial institution, a criminal can perform what is known as a **salami attack**: The crook shaves a little from many accounts and puts these shavings together to form a valuable result, like the meat scraps joined in a salami. Suppose an account generates $10.32 in interest; would the account-holder notice if there were only $10.31? Or $10.21? Or $9.31? Highly unlikely for a difference of $0.01, and not very likely for $1.01. If the thief accumulates all the shaved-off interest "scraps" into a single account (of an accomplice), the total interest amount is right, and the attack is unlikely to cause any alarm.

Similarity

As we described in Chapter 2, in spite of continuing pleas to do otherwise, people often choose common passwords. Some organizations recognize the vulnerability of storing passwords in a table that an attacker might find and download, so they encrypt them. Then the attacker has an only slightly more difficult challenge: Create several accounts, each with one of the most common passwords; then look for matches in the password table. If the attacker creates NewUser1 with a password of asdfgh and if its encrypted form 3a$woQ is the same as the encrypted password for Joe Jonas, the attacker can infer Joe's password without doing any hard cryptanalysis. In this example, forcing an entry into the password database discloses other data in the table.

All these attacks have involved some aspect of integrity. Remember the range of properties covered by the general concept of integrity; we repeat them from Chapter 1 for reference:

- precise
- accurate
- unmodified
- modified only in acceptable ways
- modified only by authorized people
- modified only by authorized processes
- consistent
- internally consistent
- meaningful and usable

Protecting these different properties requires different countermeasures, including tools, protocols, and cryptography. In previous chapters we presented some of these approaches, and now we build upon those earlier methods.

COUNTERMEASURE: CODES

One way to protect data is by using techniques to detect change. In Chapter 4 we introduced codes to detect likely changes, using programs such as Tripwire. We expanded that concept in Chapter 13 to include message digest and secure hash codes, which are really just other specializations of the general concept; these are all types of **error detection codes**. Here we explore the strengths and weaknesses of such codes and present a different class that can be used to correct, not just detect, errors.

Error Detection Codes

Checksums, hash codes, and message digests all work the same way: They are functions that can detect *some*, not necessarily all, changes, regardless of whether the change was accidental or intentionally malicious. These functions all produce a result whose value depends on each individual bit of the input data: Change just one bit in the input and usually a change in the output happens.

We hedge this description with the words some and usually for a mathematical reason: These functions reduce an n-bit message to a k-bit digest, where $n>k$, and n is usually much larger than k. Mathematically, this means not every input can have a unique output, equivalently, that there must be two inputs that produce the same output (and generally many pairs of inputs produce the same output).

To see why, consider an 8-bit input and a 4-bit output. There are 256 (2^8) inputs and only 16 (2^4) outputs, so many of the 256 inputs will have to double up on a single one of the 16 outputs. Each instance of two inputs producing the same output is called a **collision**. We say a hash function has a uniform distribution if the collisions are evenly distributed across the output values.

Each collision is a case in which a modification is undetectable. That is, if x and y are two inputs that collide so $h(x) = h(y)$, changing x to y cannot be detected under h because of the collision. Knowing that, we want these functions to be sensitive to small changes, meaning that if x and y differ in only one bit, $h(x) \neq h(y)$.

Error-detecting codes are used with credit card numbers and other account numbers to quickly detect some errors of transcription or entry; they also mean you cannot just guess any 16-digit number to use as a credit card number. (Other characteristics of credit card numbers prevent guessing attacks, as well.) But, as their name implies, error detection codes can detect that a change has occurred but usually cannot show precisely which bits were changed.

Error Correction Codes

By contrast, **error correction codes** can show not only that a change occurred but also help correct it. Richard Hamming [HAM50] first devised these codes at Bell Laboratories by inventing a code that could correct single-bit errors and detect two-bit errors; that code has subsequently been called the **Hamming code**. The original form, called a Hamming (7,3) code, appends three check bits to each group of seven data bits; the three check bits are the parity of the first three data bits, the parity of the last three data bits, and the parity of the first, second, and fourth data bits, respectively. Although a breakthrough, the Hamming code has the disadvantages that it is applied bit-by-bit and that it increases the size of the data stream by over 40 percent.

Later, Reed and Soloman [REE60] invented a series of codes, known as **Reed Solomon codes**, that apply to blocks of data, such as bytes. These codes are especially suited for correcting bursts of errors, several bits in a row, among the more common computer communications errors. In fact, Reed Solomon codes are used to correct data errors on CDs, and the coding scheme currently in use can correct 2,000 consecutive errors. Other correction codes have been developed in the branch of mathematics known as coding theory.

Codes protect the integrity of individual data items, but we need procedures to act upon these detections and corrections to produce a high-integrity data stream in storage or in transit. We discuss such procedures next.

COUNTERMEASURE: PROTOCOLS

Procedures and protocols operate internally so the user sees only data. The most common use of protocols is in networked communications.

As we described in Chapter 14, the IP protocol routes data from source to destination. On top of IP are two protocols, TCP and UDP, to decompose a data stream into smaller units for transmission and reassemble it on receipt. (A third protocol, ICMP, is used mostly for machine-to-machine error communication and is not relevant to this discussion.)

The TCP and UDP protocols have different objectives and hence different designs. TCP is a robust protocol, delivering a high-integrity data stream, but it does so with a

performance overhead. UDP is smaller and simpler, but without the integrity. For data communications such as email, web pages, and file transfers, where size is small and speed is not critical, high-integrity—correct and unmodified—data lead to TCP. For fast, large-sized communications—such as streaming audio or video or bursts of communications from satellites—in which there may already be embedded error detection and correction codes, UDP is more appropriate.

TCP achieves data integrity through a combination of a 16-bit checksum error detection code and a 32-bit sequence number, as well as a series of sending and receiving acknowledgments to ensure that nothing is lost. The receiving protocol handler inspects and verifies all the data check fields. Also, using the sequence number, it buffers packets that arrive out of order and waits for delayed data. If anything does not match properly, it calls for a retransmission of damaged or missing packets. Only when the data stream passes all integrity checks is it passed to the user.

Protocol design with several integrity-checking features improves the quality of data, especially in network communications. People, too have a role in protecting data, as we describe now.

COUNTERMEASURE: PROCEDURES

Procedures come at a level above the communications stream to which protocols apply. Once correct data have been received and stored for future reference, we want to be sure they remain intact. Error detection and correction codes are used in storage systems to ensure that what was written is what is returned later when reread. Storage systems do not have the robustness options of the TCP protocol to ensure integrity. Thus, other procedures are necessary to ensure integrity.

Backup

In Chapter 7 we explained how backups are an effective control against data loss; they are also a way to recover from integrity failures. Although we would prefer to prevent modifications, we may not always be able to do so after a failure. In such situations, a backup becomes very valuable.

Redundancy

In Chapter 15 we described redundancy as a countermeasure to denial of service: If a process or data at one location is attacked, having a second copy elsewhere reduces the risk of denied access. With the low prices of hardware and especially storage, having several redundant copies becomes a reasonable option.

One form of redundancy involves a procedure by which whenever data are written at one location, a second copy at another location is also created. The original must be written quickly enough that it does not impair usability, but the second copy can be written more slowly, as time allows. This form of redundancy is called **mirroring**. By using different locations, mirroring reduces the risk of data harm from physical threat, such as fire, flood, or vandalism.

COUNTERMEASURE: CRYPTOGRAPHY

As we have shown in several previous chapters, cryptography is a valuable tool for ensuring the integrity of data in storage or in transit. In this section we show one more use of block ciphers to guard against certain modification and sequencing threats.

Block Chaining

Cryptography is useful in part because it is consistent: The originator encrypts something and the retriever who decrypts it will always receive the same result with the proper key. But that consistency can become a disadvantage if the attacker tries to perform a substitution or sequencing attack.

Figure 16-2 shows conventional encryption, performed in what is often called the **electronic code book** mode. In this mode, each block is separately encrypted and stored or transmitted as a distinct unit. If a block is lost, two blocks are reordered or one block is substituted for another; as long as each block is properly encrypted, nothing in the encrypted result indicates an error. ABC is always encrypted as qw3, regardless of whether it is the first, second, or last block. (In these figures, plaintext is shown in UPPERCASE and ciphertext in lowercase.)

Figure 16-3 shows that an attacker can take a block, for example, YZQ/4ok from one data stream and substitute it for a different block in another data stream, with no detectable harm to the result (other than the fact that the data value has changed).

In Figure 16-4 we introduce a mode called **cipher block chaining**. In this mode, the separate encrypted blocks are chained together to prevent substitution or reordering without detection. With cipher block chaining, the first block is encrypted as normal. It is recorded as the first output, but it is also combined with the second input block (using an exclusive OR function) to produce the second output block. This second output block is then combined with the third input block to produce both the third output and a block to feed into the next encryption step, and so forth. In this way, each encrypted block depends on both the content of that input data block and the encryptions of all preceding blocks. Therefore, it is impossible to delete or reorder any of the output blocks without detection. During decryption, applying the same exclusive OR function removes the prior blocks because the exclusive OR function (represented by the symbol $\oplus$) is self-cancelling, that is, for any blocks a and b, $a \oplus b \oplus b = a$ and $a \oplus b \oplus a = b$.

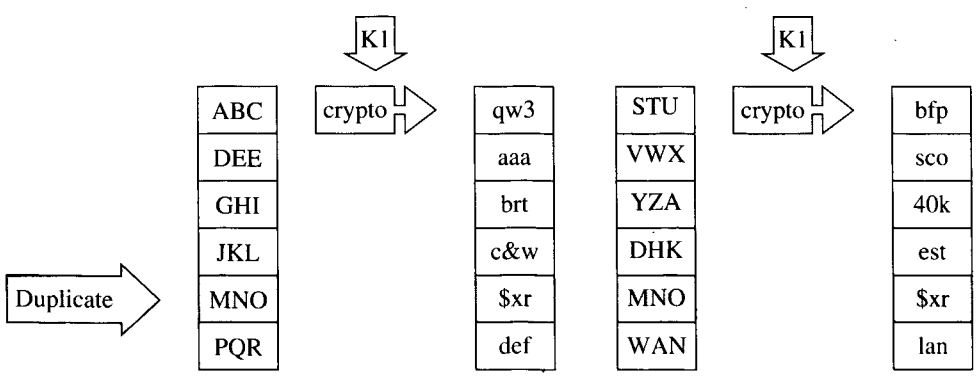

FIGURE 16-2 Electronic Code Book Encryption

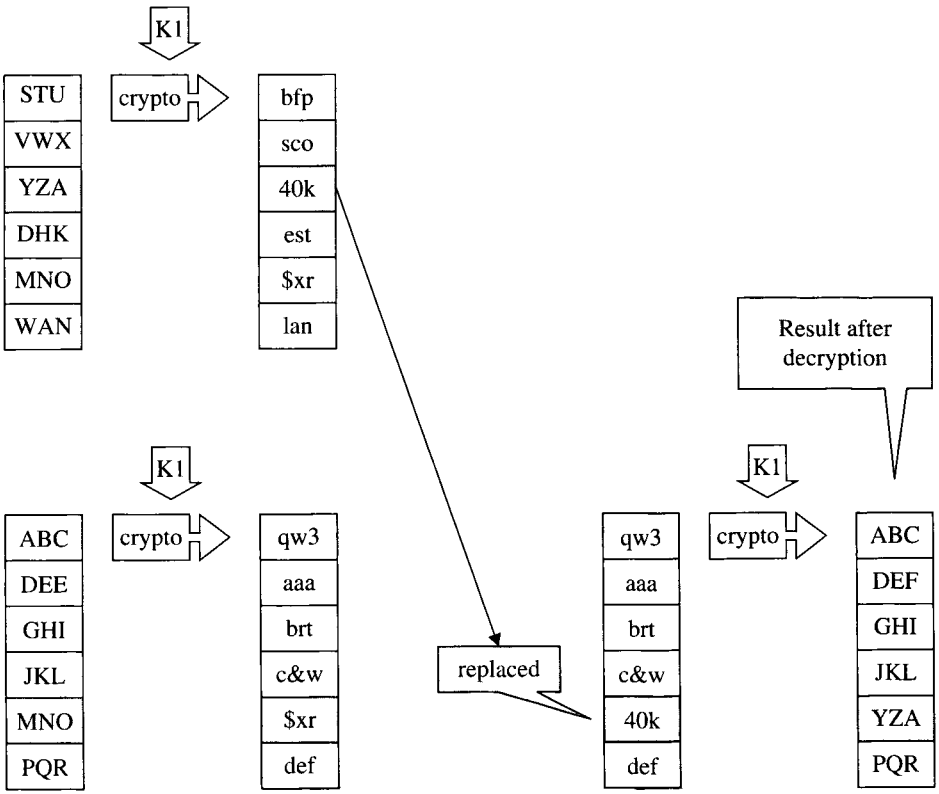

FIGURE 16-3 Undetected Substitution in Electronic Code Book Encryption

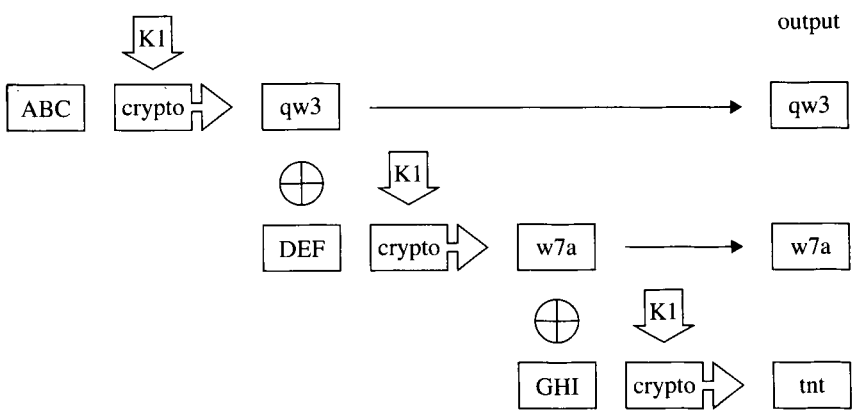

FIGURE 16-4 Block Chaining

Finally, in Figure 16-5 we add one more piece to guard against substitutions. We start each encryption with a random block, called the **initialization vector**. This first block is meaningless. But using block chaining with an initialization vector ensures that identical

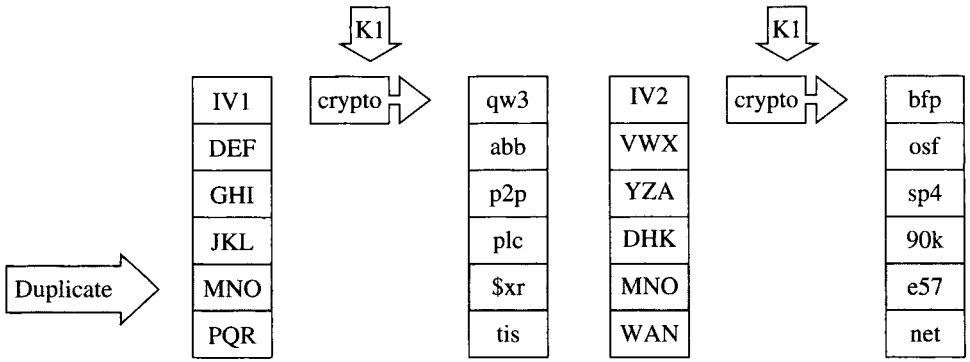

FIGURE 16-5 Block Chaining with Use of Initialization Vectors

blocks in two input streams produce different blocks in the outputs, because the initialization vectors, on which all succeeding outputs depend, will be different.

These simple cryptographic procedures guard against integrity failures.

Password Salt

Sometimes we store passwords using a technique similar to encryption with an initialization vector. Certain systems store the password table in plaintext, but in protected memory, accessible only to the routines to check and manage passwords. Assuming memory protection is strong enough, no attacker can get these passwords. On the other hand, compromise of memory leads to compromise of all users' passwords, which is a serious exposure. Even if memory protection is solid during execution, the attack might still be able to find the password table in a dump of memory, on a virtual memory swap device, or in an unguarded backup file. Because of that risk, a more secure system design, found especially in Unix systems, uses an encrypted password table.

When a user creates a password, it is immediately encrypted and stored in the table as ciphertext. Anytime the user wants to access the system, the user enters the password, which is also immediately encrypted, and this encrypted version is compared to the value stored in the password table. If the two values match, the user entered the correct password and is granted access. Because of the strength of encryption, even if an attacker gains access to the password table, the attacker cannot decrypt any of the passwords to determine what to enter as a user's password, and entering the encrypted form at the password prompt will not produce a matching result. Early Unix system designers then thought it was safe to store the password table in open system memory, reasoning that nobody could use the data stored there to obtain fraudulent access.

Or so they thought. If an attacker creates some accounts with known, common password values and the encrypted version of one of those passwords matches another password in the table, the attacker can infer the plaintext version of the matching password. Here is how this attack would work. The attacker creates user accounts with popular passwords, for example, DUMMY1 with password "password," DUMMY2 with "asdfghjk," and so forth. Suppose the encryption of password is 1zc*3jjeq8. If user Martin also uses "password" as a password, the table will have two matching

entries: passwords for both DUMMY1 and Martin will be 1zc*3jjeq8, which lets the attacker deduce Martin's password.

The Unix operating system has just this situation, which it counters with a technique that is essentially the initialization vector we just described. It works as follows. When a user sets a password, the system takes that plaintext password, appends a random number, encrypts the two together (as one long string), and stores the encrypted password and the random number, called the **salt**, in two separate fields. Unix uses as the salt a string derived from the current system date and time, but really any convenient unpredictable number would work. When the user enters a password, the system fetches the salt value for the corresponding user, encrypts the two together, and compares. As with initialization vectors, different salt values yield different ciphertext results even if the input passwords happen to be the same. In the example we just gave, because Martin and DUMMY1 created their accounts at different times, they would have different salt values, and so the encryption result $E(salt_1 + password)$ would differ from $E(salt_2 + password)$ even if the passwords are the same. This relatively simple step protects against harm from accidental collisions of passwords. It is similar to the liveness indicator for passwords we described in Chapter 14.

CONCLUSION

In this chapter we examined several integrity problems in computer systems and showed how topics presented earlier in this book can counter these threats. The key points presented in this chapter are these:

- Data corruption can occur from many causes, some based in hardware, others in software, others in procedures.
- Codes can be used to detect and, in some cases, correct errors in transmission, storage, or processing. Although not perfectly effective, such codes are strong enough to protect against nonmalicious errors and moderately sophisticated intentional attacks.
- Ordinary block cipher cryptography is vulnerable to substitution and sequencing attacks. Adding an initialization vector and chaining guards against those attacks.

We summarize these threats and protections in Table 16-1.

TABLE 16-1 Threat–Vulnerability–Countermeasure Chart for Data Corruption

Threat	Consequence	Severity	Ease of Exploitation
Sequencing	Data corruption	Mild to serious	Easy (frequent)
Substitution	Data corruption, falsification	Mild to serious	Moderate
Salami	Data corruption	Mild to serious	Moderate
Similarity	Inappropriate inference	Serious	Moderate

(Continues)

TABLE 16-1 Threat–Vulnerability–Countermeasure Chart for Data Corruption *(Continued)*

Vulnerability	Exploitability		Prevalence	
Integrity failure: data corruption	Easy to difficult		Moderate	
Inappropriate disclosure	Moderate		Infrequent, because countermeasures in common use	
Countermeasure	**Issues Addressed**	**Mitigation Type**	**Mitigation Effect**	**Effort**
Error correction and detection codes	Substitution, modification	Detection, correction	Strong	Low
TCP protocol	Sequencing, substitution, corruption	Correction	Strong	Very low
Procedures: backup, redundancy	Corruption, modification, loss	Correction	Strong	Low
Cryptography: chaining, initialization vector	Sequencing, substitution	Detection	Strong	Low

In the next two chapters we turn to two popular uses of computing systems, peer-to-peer sharing and social networking, which we then explore for threats and potential countermeasures.

EXERCISES

1. Is electronic code book cryptography an adequate countermeasure against a sequencing attack? Why or why not?
2. Is electronic code book cryptography an adequate countermeasure against a salami attack? Why or why not?
3. A university uses a computer system to manage its records of students' course grades. As though you were a consultant to the university, describe three different types of attacks that might be made against such a system, and recommend countermeasures that the university should take against these attacks.
4. Does block chaining protect against a sequencing attack? That is, if two ciphertext blocks are interchanged, will the decryption fail demonstrably? Explain your answer.
5. A human keyboard operator entering data from financial records consistently enters O (uppercase oh) instead of 0 (zero). Explain how this error should be detected and corrected; that is, at what stage of processing, by what program, and when.
6. What property of the exclusive OR function allows it to be used for block chaining? That is, could the AND or OR function be used equally well? Explain your answer.
7. The password salt prevents two identical passwords from having the same encrypted value; it was originally developed because the Unix password table can be read by many processes. Would it not be more sensible simply to protect the password table against read access? Justify your answer.