CHAPTER 7

Guards and Walls

At first glance, guards and walls seem to have paradoxical functions. Walls are designed to *block* access to the fortress. Guards are designed to *allow* access to the fortress. These two functions, however, are actually complementary. The efficacy of the walls is what makes the guard's job necessary. After all, if outsiders could send requests into the fortress any which way they wanted, there wouldn't be much incentive to use approved and guarded drawbridges.

Although charged with allowing access to the fortress, the guard is very selective. The fortress architect is responsible for deciding just how selective the guard will be and which technologies can be used to implement this selectivity.

In this chapter I discuss some of the issues involved with designing and implementing walls and guards. I will focus on three main fortress types: presentation fortresses, Web service fortresses, and business application fortresses. These three fortress types represent a good cross section of available guard and wall design and implementation strategies.

For the purposes of this discussion, I will assume a simple configuration of two fortresses wanting to communicate over a drawbridge and a bad guy who is up to no good. Shown in Figure 7.1, this setup consists of the following characters:

- Ed, the envoy in the donor fortress
- Gwen, the guard in the receiving fortress
- Bart, the bad guy

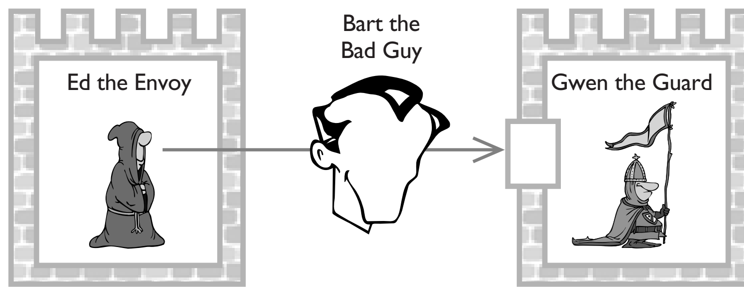

Figure 7.1 Two-Fortress Configuration

There are eight security issues we generally worry about when planning our fortress walls and guards:

1. Fortification
2. Validation
3. Auditing
4. Authentication
5. Privacy
6. Integrity
7. Nonrepudiation
8. Authorization

I'll go through these one by one, discussing each problem more fully and offering some likely solutions.

7.1 Fortification

Fortification refers to the ability of the fortress walls to prevent entry into the fortress (except, of course, through the drawbridge). If the fortress is well fortified, then the only way Bart the bad guy is going to get into it is by somehow tricking Gwen the guard into letting him in. Figure 7.2 illustrates the concept of fortification.

In the case of an Internet fortress (Web service or presentation), Bart will attempt to break into the fortress using an unexpected entry point, such as FTP or remote login. Once Bart has penetrated the

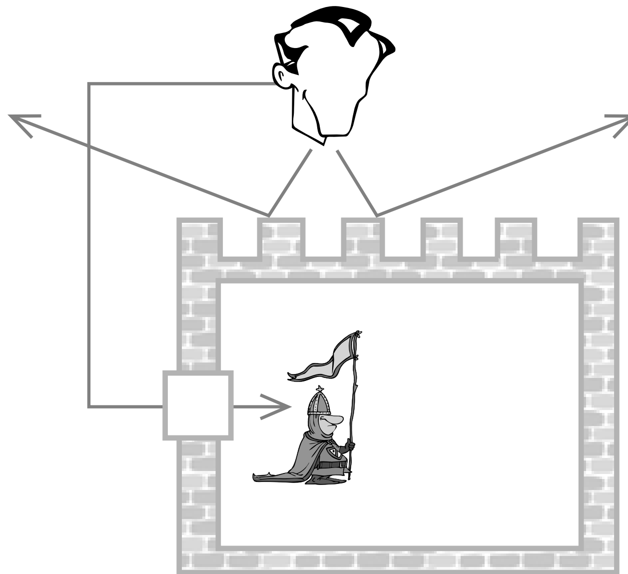

Figure 7.2 Fortress Fortification

Internet fortress, he may be able to erase or modify binaries, reboot the system, copy sensitive files, or set up his own privileged accounts, depending on the authorization level he manages to gain.

The most common way of fortifying the Internet fortress walls is with firewalls. Firewalls are a common fortification technology for many fortresses, but especially those connected to the Internet. Firewalls can restrict all but a select group of users from remote access. This small, select group is part of the fortress community (remember, a fortress is a group of people as well as a group of systems). Bart, needless to say, is not part of this select group.

Bart might also try to break into the fortress through the data strongbox. Data strongboxes are usually implemented with databases. If Bart can gain direct access to the database, he has effectively circumvented Gwen and can access all of the critical fortress data.

The database should be configured so that only user IDs associated with the owning software fortress are allowed to read, write, or modify data. Those user IDs, as trusted members of the fortress, should be allowed to do pretty much whatever they want.

Although proper configuration of the database is an important part of the fortress fortification, that configuration is much less complex for the software fortress architecture than for most N-tier architectures. N-tier architectures often require sophisticated database security configurations, with specific users allowed to access specific tables in specific ways that change as implementations of the business logic change. For a software fortress, database configuration boils down to a simple rule: If you are a member of the fortress, you can do what you want; if not, you can't. The configuration is very simple, it is easy to set up, and it doesn't change as implementations inside the fortress change.

Although all fortresses can avail themselves of firewalls and database security, business application fortresses have another trick up their sleeves: a mechanism called *role-based security*.

As I will discuss in Chapter 11 (Business Application Fortresses), business application fortress infrastructures are based on a technology that I refer to as *component-oriented middleware* (*COMWare*). COMWare is a technology closely associated with components, which I have discussed in the context of synchronous drawbridges.

Component systems transmit requests from a caller (the client) to a callee (the component instance) as a remote method invocation. The client lives in one process (the client process) and the component instance lives in another process (the component process).

Don't confuse the idea of a caller and client with a drawbridge. When we're looking at components as used within a business application fortress, the caller and the callee are part of the same fortress (although probably not part of the same process). As part of the same fortress, they do not go through drawbridges to get to each other, even though the technology is similar to some of the drawbridge technology.

Both the caller and callee processes have a specific associated user ID. In role-based security, each of these user IDs is assigned one or more roles by a component administrator. Examples of roles could be company managers or bank tellers.

Once the component administrator has given each client user ID one or more roles, that same component administrator sets up component access rights. The administrator decides which roles can access which methods in which components. For example, it might be appropriate to give either tellers or managers access to the updateAccounts method but only managers access to the addNewAccount method.

Role-based security was originally designed to deal with two security issues: authorization and authentication (I will discuss these issues in more detail later in the chapter). When role-based security was first introduced, it received a flurry of interest. It seemed attractive because it was an administrative model (one that can be totally managed by an administrator) rather than a code-based model (one that requires a programmer). All things being equal, administrative models are always preferable over coding models because they are easier to maintain. Besides, nobody believes that coders care all that much about security anyway.

However, role-based security turned out not to live up to its promise. The reason is that relatively few problems can be dealt with through role-based authorization. Let me give an example of a common problem that is not amenable to a role-based solution.

Consider a bank wanting to let clients check their balances online. The bank wants to make sure the clients are authorized to access the requested balances. This is a two-part problem. First, the clients must be authenticated (we know who they really are). Second, the bank is going to let clients access only their own bank balances. Using role-based security, we can say that only users in the client role can access the checkBalance method, but there is no way to specify that they can do so only when requesting to check their own account balances.

The problem we run into here is that limiting a client to only a specific account means that authorization is based not only on a specific method, but also on specific parameters to that method (e.g., accountID). Even the parameters don't give the full story. To

really verify the user's access right, we probably need to check information in the database to be sure this particular account ID is associated with this particular client. Such a task is way beyond the capabilities of role-based security.

Why, you might ask, am I spending so much time on role-based security if it is so limited? It turns out that fortress architectures may be the redemption of role-based security. Although role-based security is relatively useless in run-of-the-mill authorization problems, it can provide one important capability for software fortresses: fortification.

In the same way that we can fortify the fortress by locking foreign users, such as Bart the bad guy, out of the database, we can also fortify the fortress by locking foreign users out of the components. We do this by using role-based security. We define each process in the fortress as being a member of the role of, say, this-fortress. All components that are part of the fortress are then configured to allow access by members of the this-fortress role. If your user ID is associated with this-fortress, then you can ask any of your fortressmates to do whatever you want. If your user ID is not associated with this-fortress, then don't even bother asking for the time of day. And that includes you, Bart!

7.2 Validation

Validation refers to the checking and rechecking of user input. One of the tricks that Bart learns in bad-guy school is how to exploit weaknesses in the guard, especially guards in Internet fortresses. In this scenario, Bart doesn't try to bypass Gwen the guard; he tries to trick her into doing something she shouldn't do. There are two ruses Bart will use to try to trick Gwen into becoming his willing accomplice.

In the first approach, Bart tries to overwhelm Gwen with large amounts of data, hoping to overwrite her allocated memory buffers. Bart sends much more data to Gwen than she is expecting. If Gwen doesn't check for this data excess, Bart can turn Gwen into his personal zombie. Gwen will now do anything Bart asks! And since Gwen

is a trusted member of the inner sanctum, whatever Gwen asks, the rest of the fortress, having no idea that Gwen is now under the hypnotic spell of Bart, will do. Effectively, Bart has control of the entire fortress.

The technical term for Gwen's zombielike condition is a *buffer overflow*. Buffer overflows are probably the most common approach used by hackers to attack both presentation and Web service fortresses. The attacker enters large amounts of data into form fields, hoping to overwrite the memory addresses that control program flow. This approach allows a hacker to hijack the presentation (or Web service) fortress processes.

If Bart is thwarted in his attempt to cause Gwen's memory buffers to overflow, his next approach will be to send in scam data. Scam data is data that Bart hopes Gwen will pass to the inner fortress workers as real data, and that he further hopes will convince those workers to do something they normally wouldn't do.

For example, let's say that one of the parameters Gwen is expecting is an account ID. Gwen will pass that account ID on to an inner worker—say, Walt—who will subtract an amount from that account. Assume that Walt will find the account in the database using the SQL statement "Select * from accounts where accountID = account", where *account* is the parameter that Gwen passed through.

Bart is expected to pass in a string like "12345", which then will be passed through Gwen. But we know what Bart is like. Suppose that instead of passing in "12345", Bart passes in the string "1@". Now the SQL statement becomes "Select * from accounts where accountID = 1@". If the database interprets the @ character as a wildcard, then Bart will be able to subtract money from every account in the database. This is not exactly something that Walt would willingly do.

It is Gwen's responsibility to worry about unexpected characters in input fields. Unexpected characters are the sign of scam data attacks. If Gwen finds, for example, a quotation mark within a field that is expected to contain a user name, she must assume that she is under scam data attack. If user names should contain only letters, numbers, and underscores, then Gwen must treat *any* character that is not one of these as signaling a scam data attack.

Gwen needs to be highly suspicious of all user input. She must check each string character by character. She must examine the length of every string to be sure it meets length constraints. Gwen can never know what input is coming from a bona fide user and what input is coming from Bart the Bad.

If Gwen finds herself under attack, either by attempted buffer overflow or by scam data, she should take appropriate actions. She should *not* try to fix the data and then send it on to Walt the worker. She should reject the infogram containing the scam data in its entirety. If possible, she might want to take evasive actions to guard against further attacks. In an extreme case, she might even want to shut herself and her fortress down, sacrificing her own life to protect the greater good of the enterprise.

7.3 Auditing

Auditing refers to the ability to track all changes to the internal state of a fortress. Because the state of the fortress is effectively the content of the data strongbox, we can say that auditing is the ability to track all changes to the data strongbox.

There are four things we want tracked in an auditing system:

1. The fortress that made the request resulting in the change.
2. The exact request the fortress made.
3. The data that was sent with the request.
4. The time the request was made.

Other bits of information may be included in the audit trail, but these are the main points.

Hypothetically, auditing could be done in several places. It could be done at the worker level, with each worker in the fortress, when asked to do something, logging that request. It could be done within the database itself, which is where the database vendors, with their egocentric view of the world, recommend it be done. It could be done by Gwen. Where should it be done?

In my view, auditing at either the worker or the database level makes no sense. The workers have no way of knowing which fortress originated the request, what the original request looked like, what data was sent in, or even when the request was made. The database is even further removed from this information. So neither the worker nor the database is in a good position to take on serious audit responsibility.

The only one who has access to the exact information we want audited is good old reliable Gwen. Gwen is not only the obvious candidate for auditing; she is, in my view, the *only* candidate. Not every fortress may need to have requests audited. For those that do, Gwen is our girl.

Auditing is not only useful for security purposes; it can also be useful for debugging. At the enterprise level, debugging often requires careful workflow tracking, as one tries to determine when an error first occurred. The ability to dynamically turn on auditing at the fortress level can be a big help in these unpleasant situations.

7.4 Authentication

Authentication refers to the procedure we go through to convince ourselves that somebody is who he or she claims to be. We have two fortresses in a trust relationship: Ed the envoy's fortress and Gwen the guard's fortress. Neither fortress wants to talk to Bart the bad guy. But how does Gwen know that she is talking to Ed, and not to Bart pretending to be Ed? For that matter, how does Ed know that he is really talking to Gwen, and not to Bart pretending to be Gwen? It is just one more task we're going to add to Gwen's ever expanding list of responsibilities: to be sure that when she thinks she is talking to Ed, she really is talking to Ed. Depending on how paranoid Ed is, he can also take on the problem of being sure that Gwen is really Gwen.

There are generally two approaches to authentication: (1) through a shared key (symmetrical encryption) or (2) through a public/private key (asymmetrical encryption). Both are based on encryption/decryption algorithms; let's look at this class of algorithms.

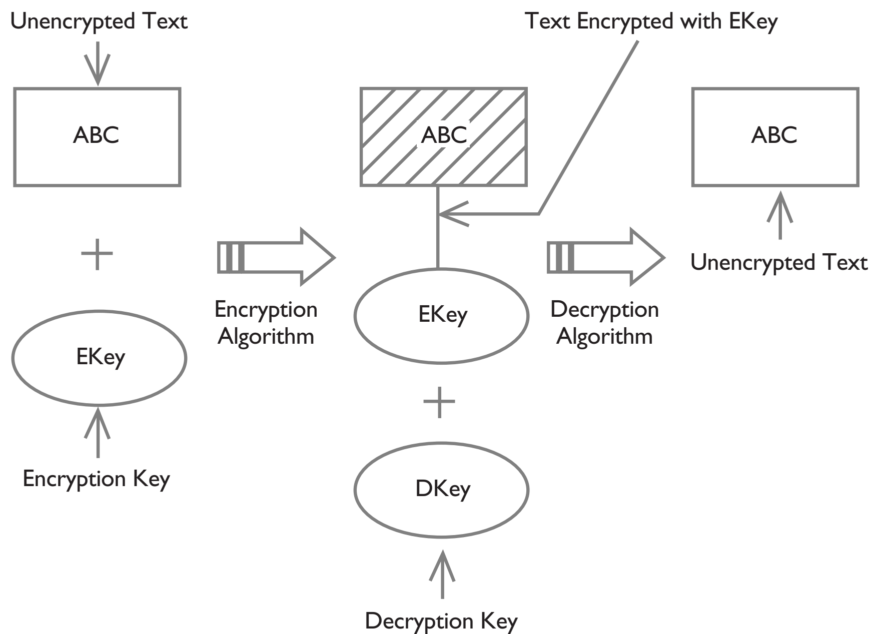

Figure 7.3 Encryption and Decryption Phases

Encryption and decryption are mirror images of each other. Encryption means taking some text and an encryption key and returning garbled text. Decryption means taking that same garbled text along with a decryption key and returning the original text. The two processes are illustrated in Figure 7.3.

The difference between a shared-key system, like Kerberos, and public/private–key systems, like secure sockets, is whether or not the decryption key is the same as the encryption key. For shared-key systems, the two keys are the same. For public/private–key systems, the two keys are different, although algorithmically related.

For shared-key systems, both parties (in this case Ed and Gwen) store a secret key (often corresponding to a password) with a trusted third-party authenticator, whom I will call Al. Both Ed and Gwen trust Al to keep each of their secret keys a secret.

When Ed wants to talk to Gwen, he asks Al the authenticator for a "ticket" to use Gwen. In response, Al gives Ed a ticket to use Gwen.

The ticket contains information about Ed and a temporary key called a *session key*. That ticket is encrypted with Gwen's secret key, which only Gwen and the authenticator (Al) know. Notice that the ticket is useless to Ed; Ed can't read it. Only Gwen, who knows the decryption key (which is, after all, her secret key), can read the ticket. About the only thing that Ed can do with the ticket is send it on to Gwen.

Al gives Ed one more thing. He gives him a package containing the session key, the same one contained in the ticket. That package is encrypted with Ed's secret key, which only Ed and the authenticator know.

Now Ed has two things. He has the ticket, which only Gwen can read, and he has the package containing the session key, which only he can read. So both Gwen and Ed now have the session key, albeit encoded with different secret keys.

Now Ed and Gwen are ready to prove to each other their respective identities. Ed sends the ticket to Gwen. Gwen reads it with her secret key, thereby convincing herself that only Al could have prepared the package. She opens (decrypts) the package and gets both the session key and the information about Ed. This convinces her that Al has authenticated Ed, and she trusts Al.

This algorithm is very close to the one used by Kerberos, and in fact a Kerberos authenticator could be used as the trusted third party Al.

There are at least four disadvantages to this algorithm. First, both parties need to trust the same third party with their secret keys. Kerberos relaxes this requirement slightly by providing a fourth party, but the effect is the same.

Second, the trusted third party, Al, can be a bottleneck for the algorithms because he is needed for every communication initiation. For fortresses, this probably means that Al must be consulted with every use of the drawbridge.

Third, the algorithm is susceptible to compromise. The algorithm can be compromised if Bart (the bad guy; remember him?) steals a ticket. Bart can then use brute force to guess Gwen's private key. Once he knows Gwen's private key, there is nothing to stop him

from impersonating her. Similarly, Bart can steal the package (the one encrypted with Ed's private key) and use a similar brute-force approach to guess Ed's private key, with similar consequences.

The fourth disadvantage of the algorithm is a particular problem for Web service and presentation fortresses, the very fortresses that are most likely to want to use this algorithm. To understand this problem, we need to follow the algorithm from Gwen's perspective.

Gwen gets a message from somebody who claims to be Ed. She believes it really is Ed if her secret key can be used to decrypt the data. Besides herself, only Al the authenticator knows her secret key, so Al must have sent Ed the package containing the session key. To prove that Ed is who he says he is, Gwen must decrypt the ticket. And to decrypt the ticket, she must use her secret key. This need implies that Gwen is storing her secret key someplace in her fortress, someplace where she can get to it easily.

In Chapter 10 (Internet Fortresses) I will discuss some of the security guidelines for building Web service and presentation fortresses. One of these guidelines is to be careful not to store confidential information in the fortress. A great example of confidential information that should not be stored in the presentation fortress is Gwen's secret key, the very key she needs to prove that Ed is Ed. This is a fundamental problem with the shared-key algorithm. Without her key, Gwen can't securely prove that Ed is Ed. But if anybody steals her private key, then she is really in trouble. Now Bart can impersonate her at will.

All of these disadvantages can be eliminated with public/private–key systems. In these systems, each player (Ed and Gwen, in our drama) has both a public and a private key. The public key is readily available to anybody who wants it. The private key is kept secret. The only one who knows Ed's private key is Ed.

Either the public or the private key can be used to encrypt, and whichever one is used, the other one, and only the other one, can be used to decrypt. So if the private key was used for encryption, only the public key will decrypt. If the public key was used to encrypt, only the private key will decrypt.

We still need a trusted authenticator, Al, but he plays a much smaller role here than he does in the shared-key system. First of all, Al doesn't need to know anybody's secret key (as he does with the Kerberos-like system). Second, Al is not consulted on every communications initiation. In fact, the only role Al plays is to guarantee that Ed's and Gwen's public keys are really Ed's and Gwen's.

The way that Al guarantees that Ed's public key is really Ed's is to create a package containing Ed's public key and identification information about Ed. Al then encrypts this package with his own private key. This encrypted package is called a *certificate*. Al sends this certificate to Ed. Ed can now send the certificate to Gwen.

How does Gwen know that Al certified Ed's public key, the one that is contained in the certificate? She decrypts the certificate with Al's public key. This decryption will work only if the certificate was originally encrypted with Al's private key. Gwen doesn't know Al's private key, but she does know his public key (so does everybody else). Anything that Al's public key decrypts must have been encrypted with Al's private key, and she trusts Al to keep his private key private.

Most of the disadvantages of the shared-key (Kerberos-like) algorithm are eliminated or greatly reduced with the public/private–key system. First, nobody needs to trust Al with a secret key (except Al, and if Al can't trust Al, who can?). Second, Al is not needed on an ongoing basis—only once in a while for approving certificates. The third disadvantage is still there; that is, Bart can still potentially guess Gwen's secret key by brute force, but in this case even if that happened, Gwen would just need to create a new secret key and reregister its public counterpart with Al.

The final disadvantage of the shared-key system—the fact that Gwen is forced to store her private key in her own fortress—is completely eliminated with public/private–key systems. Gwen now no longer needs her own private key, at least to prove to herself that Ed is really Ed. She may need her private key to prove to Ed that she is really Gwen, but Gwen can also provide such proof by having another fortress that she trusts do the encryption on her behalf.

One final note: None of these authentication schemes work well for the typical presentation fortresses that must deal with very large numbers of browsers. Private-key systems don't work because no third parties are widely trusted. Private/public–key systems don't work because the credentials are too difficult to manage effectively. But for presentation fortresses that deal with only a limited number of browsers, or Web service fortresses, these systems usually work fine.

This is a fairly high-level look at the main issues involved with fortress authentication, but it should start you thinking.

7.5 Privacy

Privacy refers to the ability to send information in such a way that it cannot be read by unauthorized users. When privacy is implemented, Bart can still read the infograms, but they will all appear as garbled, unintelligible text, much like the user manuals you get with your operating system.

Privacy is usually accomplished with secret-key encryption. Ed and Gwen share a secret key that only they know about. Ed uses this key to encrypt data before he sends it. Gwen uses the same key to decrypt data after she receives it. To anybody in the middle, the data appears as garbled text.

This secret key is not either of their private keys. That would be a violation of the fortress trust rule. Instead, they share a temporary session key, a key that only the two of them know and that is valid only for a limited duration (probably minutes).

The trick is to exchange this secret key in such a way that even if Bart eavesdrops, he won't be able to read the key and thereby read the transmitted data. There are two ways that Ed and Gwen can exchange this secret key. The first way to share the key is as a side effect of the private-key algorithm. Remember, I said that one of the items in the Kerberos-like ticket is a session key. This session key is exactly the kind of secret key that Ed and Gwen need.

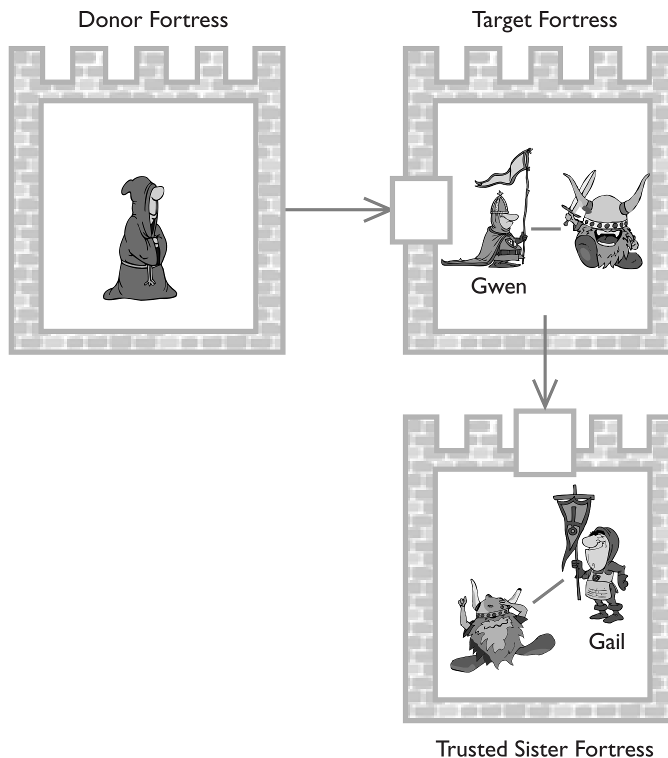

Figure 7.4 Gwen's and Gail's Fortresses

The other way to share the key is by using a public/private–key pair. In the public/private–key scheme, Ed creates a session key and encrypts it using Gwen's public key. He sends the encrypted session key to Gwen. Only Gwen's private key will decrypt the session key.

As I mentioned earlier, if Gwen is guarding a Web service or presentation fortress, she herself will not store her private key. It will instead be stored in a closely trusted sister fortress, guarded by Gail, as shown in Figure 7.4. Gwen trusts Gail's fortress to store her private key and decrypt information on her behalf. Gwen can then either ask Gail's fortress for the session key (if that does not introduce an unacceptable security risk) or have Gail's fortress do all encrypting and decrypting on her behalf.

7.6 Integrity

Integrity means preventing Bart the bad guy from changing infograms en route. Most fortress implementations deal with integrity as a side issue of privacy. The assumption is that if Bart can't read the message, he can't change it. Even more secure algorithms exist that can ensure that Bart can't even randomly change the infogram, but I won't cover those here. These are standard security algorithms that are not specific to fortresses.

7.7 Nonrepudiation

Nonrepudiation refers to the ability to prove at a later time that an infogram came from Ed the envoy. Suppose that we get a message from Ed asking that a savings account be decremented. Gwen's fortress decrements the account. Later, Ed says he never made the request. How can Gwen's fortress prove that Ed asked for the account to be decremented?

The easiest way to do this is with public and private keys. Gwen can insist that the decrement request be encrypted with Ed's private key. Gwen can decrypt this request with Ed's public key. After arranging to have the account decremented, Gwen then permanently logs the encrypted infogram. She might even want to log Ed's public key as well, just in case Ed later decides to change it. Should Ed ever deny sending the infogram, Gwen can retrieve the logged infogram and show that she can decrypt it with Ed's public key. If she can decrypt the infogram with Ed's public key, then it must have been encrypted with Ed's private key. Because only Ed knows Ed's private key, Ed must have sent the infogram. Nice try, Ed!

If Gwen is guarding a presentation or Web service fortress, she probably won't log the encrypted infogram herself because that would involve storing valuable data (the infogram log) on a nonsecure fortress. Instead she will use a trusted service fortress to do the logging for her.

7.8 Authorization

Authorization refers to the ability to determine, *not* on the basis of the fortress making the request but on the basis of the information in the request itself, whether the request being made is allowable. As a simple example, imagine Bart the bad guy is sitting at a browser and asks that 1,000 dollars be removed from Alice's account. He knows Alice's account number but not her password. This request should be rejected. The reason for rejecting the request is not that Bart's browser is an untrusted source. Gwen, sitting in a presentation fortress, has no problem with Bart's browser. The request should be rejected because there is a problem with the data in the infogram.

Solving authorization problems generally requires deferring to the business logic. Theoretically Gwen could check the database to see if the password matched the password for Alice's account. But such checking is difficult, for different reasons, depending on what kind of a fortress Gwen is guarding.

If Gwen is guarding a presentation fortress, she has no access to the database containing account IDs and passwords. This information is obviously highly confidential, and as I have hinted at and will discuss in more detail in Chapter 10 (Internet Fortresses), we don't store confidential information in a presentation fortress.

If Gwen is guarding a business application fortress, then for performance reasons we want to minimize database access. If she has to access the database to check the password and then a business application worker has to access the same database to do the account update, we have two database accesses. If the worker does both the authorization check and the account update, we can eliminate half of the database accesses.

The bottom line is that authorization, not to be confused with authentication, is usually not a guard issue, but a business application worker issue, based on application-specific algorithms.

Summary

This chapter has shown that there is quite a bit to creating guards and walls. This is the reason I emphasize making the creation of guards and walls a specialized task within your organization.

Here are the major lessons of this chapter:

- The walls are primarily responsible for fortification.
- Three technologies are typically used to build walls:
    1. Firewalls
    2. Database security configuration
    3. Role-based security
- All data coming into the fortress must be validated. Validation includes
    - Checking for string length violations
    - Checking for unexpected characters
- Auditing is important if you need to track fortress changes.
- Authentication is needed to verify that requests are coming from approved sources. Authentication is done with encryption/decryption algorithms based on either secret keys (in the shared-key system) or public/private keys (in the public/private–key system).
- Privacy is about hiding data from prying eyes, which is accomplished by encryption/decryption.
- Integrity means keeping data from changing as it passes through the drawbridge, which also makes use of encryption/decryption.
- Nonrepudiation means being able to prove, at a later date, the source of an infogram. This is usually done with public/private–key systems.
- Authorization, although it seems like a guard/wall issue, is usually done by a business application worker.