Chapter 7

Authentication

Guard: *Halt! Who goes there?*
Arthur: *It is I, Arthur, son of Uther Pendragon, from the castle of Camelot. King of the Britons, defeater of the Saxons, sovereign of all England!*
— Monty Python and the Holy Grail

Then said they unto him, Say now Shibboleth: and he said Sibboleth: for he could not frame to pronounce it right. Then they took him, and slew him at the passages of Jordan: and there fell at that time of the Ephraimites forty and two thousand.
— Judges 12:6

7.1 Introduction

We'll use the term *access control* as an umbrella for any security issues related to access of system resources. Within this broad definition, there are two areas of primary interest, namely, authentication and authorization.

Authentication is the process of determining whether a user (or other entity) should be allowed access to a system. In this chapter, our focus is on the methods used by humans to authenticate to local machines. Another type of authentication problem arises when the authentication information must pass through a network. While it might seem that these two authentication problems are closely related, in fact, they are almost completely different. When networks are involved, authentication is almost entirely an issue of security protocols. We'll defer our discussion of protocols to Chapters 9 and 10.

By definition, authenticated users are allowed access to system resources. However, an authenticated user is generally not given carte blanche access to all system resources. For example, we might only allow a privileged user—

such as an administrator—to install software on a system. How do we restrict the actions of authenticated users? This is the field of authorization, which is covered in the next chapter. Note that authentication is a binary decision—access is granted or it is not—while authorization is all about a more fine-grained set of restrictions on access to various system resources.

In security, terminology is far from standardized. In particular, the term access control is often used as a synonym for authorization. However, in our usage, access control is more broadly defined, with both authentication and authorization falling under the heading of access control. These two parts of access control can be summarized as follows.

- Authentication: Are you who you say you are?[1]
- Authorization: Are you allowed to do that?

7.2 Authentication Methods

In this chapter we address various methods that are commonly used to authenticate a human to a machine. That is, we want to convince a dumb machine that someone or something claiming to be Alice is indeed Alice and not, say, Trudy. That is, we want to answer the question, "Are you who you say you are?" Of course, we'd like to do this in as secure manner as possible.

A human can be authenticated to a machine based on any of the following[2] "somethings" [14].

- Something you know
- Something you have
- Something you are

A password is an example of "something you know." We'll spend some time discussing passwords, and in the process show that passwords represent a weak link in many modern information security systems.

An example of "something you have" is an ATM card or a smartcard. The "something you are" category is synonymous with the rapidly expanding field of biometrics. For example, today you can purchase a laptop that scans your thumbprint and uses the result for authentication. We'll discuss a few biometric methods later in this chapter. But first up are passwords.

[1] Try saying that three times, fast.

[2] Additional "somethings" are sometimes proposed. For example, one wireless access point authenticates a user by the fact that the user pushes a button on the device. This shows that the user has physical access to the device, and could be viewed as authentication by "something you do."

7.3 Passwords

Your password must be at least 18770 characters
and cannot repeat any of your previous 30689 passwords.
— Microsoft Knowledge Base Article 276304

An ideal password is something that you know, something that a computer can verify that you know, and something nobody else can guess—even with access to unlimited computing resources. We'll see that in practice it's difficult to even come close to this ideal.

Undoubtedly you are familiar with passwords. It's virtually impossible to use a computer today without accumulating a significant number of passwords. You probably log into your computer by entering a username and password, in which case you have obviously used a password. In addition, many other things that we don't call "password" act as passwords. For example, the PIN number used with an ATM card is, in effect, a password. And if you forget your password, a user-friendly website might authenticate you based on your social security number, your mother's maiden name, or your date of birth, in which case, these things are acting as passwords. A problem with such passwords is that they are often not secret.

If left to their own devices, users tend to select bad passwords, which makes password cracking surprisingly easy. In fact, we'll provide some basic mathematical arguments to show that it's inherently difficult to achieve security via passwords.

From a security perspective, a solution to the password problem would be to instead use randomly generated cryptographic keys. The work of cracking such a "password" would be equivalent to an exhaustive key search, in which case our passwords could be made at least as strong as our cryptography. The problem with such an approach is that humans must remember their passwords and we're not good at remembering randomly selected bits.

We're getting ahead of ourselves. Before discussing the numerous problems with passwords, we consider why passwords are so popular. Why is authentication based on "something you know" so much more popular than the more secure "somethings" (i.e., "something you have" and "something you are")? The answer, as always, is cost[3] and, secondarily, convenience. Passwords are free, while smartcards and biometric devices cost money. Also, it's more convenient for an overworked system administrator to reset a password than to provide a new smartcard or issue a user a new thumb.

[3]Students claim that when your Socratic author asks a question in his security class, the correct answer is invariably either "money" or "it depends."

7.3.1 Keys Versus Passwords

We've already claimed that cryptographic keys would solve the password problem. To see why this is so, let's compare keys to passwords. On the one hand, suppose our generic attacker, Trudy, is confronted with a 64-bit cryptographic key. Then there are 2^{64} possible keys, and, if the key was chosen at random (and assuming there is no shortcut attack), Trudy must on average try 2^{63} keys before she expects to find the correct one.

On the other hand, suppose Trudy is confronted with a password that is known to be eight characters long, with 256 possible choices for each character. Then there are $256^8 = 2^{64}$ possible passwords. At first glance, cracking such passwords might appear to be equivalent to the key search problem. Unfortunately (or, from Trudy's perspective, fortunately) users don't select passwords at random, because users must remember their passwords. As a result, a user is far more likely to choose an 8-character dictionary word such as

```
password
```

than, say,

```
kf&Yw!a[
```

So, in this case Trudy can make far fewer than 2^{63} guesses and have a high probability of successfully cracking a password. For example, a carefully selected dictionary of $2^{20} \approx 1,000,000$ passwords would likely give Trudy a reasonable probability of cracking a given password. On the other hand, if Trudy attempted to find a randomly generated 64-bit key by trying only 2^{20} possible keys, her chance of success would be a mere $2^{20}/2^{64} = 1/2^{44}$, or less than 1 in 17 trillion. The bottom line is that the non-randomness of password selection is at the root of the problems with passwords.

7.3.2 Choosing Passwords

Not all passwords are created equal. For example, everyone would probably agree that the following passwords are weak:

- `Frank`
- `Pikachu`
- `10251960`
- `AustinStamp`

especially if your name happens to be Frank, or Austin Stamp, or your birthday is on 10/25/1960.

Security often rests on passwords and, consequently, users should have passwords that are difficult to guess. However, users must be able to remember their passwords. With that in mind, are the following passwords better than the weak passwords above?

- `jfIej(43j-EmmL+y`
- `09864376537263`
- `POkemON`
- `FSa7Yago`

The first password, `jfIej(43j-EmmL+y`, would certainly be difficult for Trudy to guess, but it would also be difficult for Alice to remember. Such a password is likely to end up on the proverbial post-it note stuck to the front of Alice's computer. This could make Trudy's job much easier than if Alice had selected a "less secure" password.

The second password on the list above is also probably too much for most users to remember. Even the highly trained U.S. military personal responsible for launching nuclear missiles are only required to remember 12-digit firing codes [14].

The password `POkemON` might be difficult to guess, since it's not a standard dictionary word due to the digits and the upper case letters. However, if the user were known to be a fan of Pokémon, this password might be relatively easy prey.

The final password, `FSa7Yago`, might appear to reside in the difficult to guess, but too difficult to remember category. However, there is a trick to help the user remember it—it's based on a *passphrase*. That is, `FSa7Yago` is derived from the phrase "four score and seven years ago." Consequently, this password should be relatively easy for Alice to remember, and yet relatively difficult for Trudy to guess.

An interesting password experiment is described in [14]. Users were divided into three groups, and given the following advice regarding password selection:

- Group A — Select passwords consisting of at least six characters, with at least one non-letter. This is fairly typical password selection advice.
- Group B — Select passwords based on passphrases.
- Group C — Select passwords consisting of eight randomly selected characters.

The experimenters tried to crack the resulting passwords for each of the three groups. The results were as follows:

- Group A — About 30% of passwords were easy to crack. Users in this group found their passwords easy to remember.
- Group B — About 10% of the passwords were cracked, and, as with users in Group A, users in this group found their passwords easy to remember.
- Group C — About 10% of the passwords were cracked. Not surprisingly, the users in this group found their passwords difficult to remember.

These results clearly indicate that passphrases provide the best option for password selection, since the resulting passwords are relatively difficult to crack yet easy to remember.

This password experiment also demonstrated that user compliance is hard to achieve. In each of groups A, B, and C, about one-third of the users did not comply with the instructions. Assuming that non-compliant users tend to select passwords similar to Group A, about one-third of these passwords would be easy to crack. The bottom line is that nearly 10% of passwords are likely to be easy to crack, regardless of the advice given.

In some situations, it makes sense to assign passwords, and if this is the case, noncompliance with the password policy is a non-issue. The trade-off here is that users are likely to have a harder time remembering assigned passwords as compared to passwords they select themselves.

Again, if users are allowed to choose passwords, then the best advice is to choose passwords based on passphrases. In addition, system administrators should use a password-cracking tool to test for weak passwords, since attackers certainly will.

It is also sometimes suggested that periodic password changes should be required. However, users can be very clever at avoiding such requirements, invariably to the detriment of security. For example, Alice might simply "change" her password without changing it. In response to such users, the system could remember, say, five previous passwords. But a clever user like Alice will soon learn that she can cycle through five password changes and then reset her password to its original value. Or, if Alice is required to choose a new password each month she might select, say, `frank01` in January, `frank02` in February, and so on. Forcing reluctant users to choose reasonably strong passwords is not as simple as it might seem.

7.3.3 Attacking Systems via Passwords

Suppose that Trudy is an outsider, that is, she has no access to a particular system. A common attack path for Trudy would be

$$\text{outsider} \longrightarrow \text{normal user} \longrightarrow \text{administrator}.$$

In other words, Trudy will initially seek access to any account on the system and then attempt to upgrade her level of privilege. In this scenario, one weak password on a system—or in the extreme, one weak password on an entire network—could be enough for the first stage of the attack to succeed. The bottom line is that one weak password may be one too many.

Another interesting issue concerns the proper response when attempted password cracking is detected. For example, systems often lock users out after three bad passwords attempts. If this is the case, how long should the system lock? Five seconds? Five minutes? Until the administrator manually resets the service? Five seconds might be insufficient to deter an automated attack. If it takes more than five seconds for Trudy to make three password guesses for every user on the system, then she could simply cycle through all accounts, making three guesses on each. By the time she returns to a particular user's account, more than five seconds will have elapsed and she will be able to make three more guesses without any delay. On the other hand, five minutes might open the door to a denial of service attack, where Trudy is able to lock accounts indefinitely by periodically making three password guesses on an account. The correct answer to this dilemma is not readily apparent.

7.3.4 Password Verification

Next, we consider the important issue of verifying that an entered password is correct. For a computer to determine the validity of a password, it must have something to compare against. That is, the computer must have access to the correct password in some form. But it's probably a bad idea to simply store the actual passwords in a file, since this would be a prime target for Trudy. Here, as in many other areas in information security, cryptography provides a sound solution.

It might be tempting to encrypt the password file with a symmetric key. However, to verify passwords, the file must be decrypted, so the decryption key must be as accessible as the file itself. Consequently, if Trudy can steal the password file, she can probably steal the key as well. Consequently, encryption is of little value here.

So, instead of storing raw passwords in a file or encrypting the password file, it's more secure to store hashed passwords. For example, if Alice's password is `FSa7Yago`, we could store

$$y = h(\texttt{FSa7Yago})$$

in a file, where h is a secure cryptographic hash function. Then when someone claiming to be Alice enters a password x, it is hashed and compared to y, and if $y = h(x)$ then the entered password is assumed to be correct and the user is authenticated.

The advantage of hashing passwords is that if Trudy obtains the password file, she does not obtain the actual passwords—instead she only has the hashed passwords. Note that we are relying on the one-way property of cryptographic hash functions to protect the passwords. Of course, if Trudy knows the hash value y, she can conduct a forward search attack by guessing likely passwords x until she finds an x for which $y = h(x)$, at which point she will have cracked the password. But at least Trudy has work to do after she has obtained the password file.

Suppose Trudy has a dictionary containing N common passwords, say,

$$d_0, d_1, d_2, \ldots, d_{N-1}.$$

Then she could precompute the hash of each password in the dictionary,

$$y_0 = h(d_0), y_1 = h(d_1), \ldots, y_{N-1} = h(d_{N-1}).$$

Now if Trudy gets access to a password file containing hashed passwords, she only needs to compare the entries in the password file to the entries in her precomputed dictionary of hashes. Furthermore, the precomputed dictionary could be reused for each password file, thereby saving Trudy the work of recomputing the hashes. And if Trudy is feeling particularly generous, she could post her dictionary of common passwords and their corresponding hashes online, saving all other attackers the work of computing these hashes. From the good guy's point of view, this is a bad thing, since the work of computing the hashes has been largely negated. Can we prevent this attack, or at least make Trudy's job more difficult?

Recall that to prevent a forward search attack on public key encryption, we append random bits to the message before encrypting. We can accomplish a similar effect with passwords by appending a non-secret random value, known as a *salt*, to each password before hashing. A password salt is analogous to the initialization vector, or IV, in, say, cipher block chaining (CBC) mode encryption. Whereas an IV is a non-secret value that causes identical plaintext blocks to encrypt to different ciphertext values, a salt is a non-secret value that causes identical password to hash to different values.

Let p be a newly entered password. We generate a random salt value s and compute $y = h(p, s)$ and store the pair (s, y) in the password file. Note that the salt s is no more secret than the hash value. Now to verify an entered password x, we retrieve (s, y) from the password file, compute $h(x, s)$, and compare this result with the stored value y. Note that salted password verification is just as easy as it was in the unsalted case. But Trudy's job has become much more difficult. Suppose Alice's password is hashed with salt value s_a and Bob's password is hashed with salt value s_b. Then, to test Alice's password using her dictionary of common passwords, Trudy must compute the hash of each word in her dictionary with salt value s_a, but to attack

Bob's password, Trudy must recompute the hashes using salt value s_b. For a password file with N users, Trudy's work has just increased by a factor of N. Consequently, a precomputed file of hashed passwords is no longer useful for Trudy. She can't be pleased with this turn of events.[4]

7.3.5 Math of Password Cracking

Now we'll take a look at the math behind password cracking. Throughout this section, we'll assume that all passwords are eight characters in length and that there are 128 choices for each character, which implies there are

$$128^8 = 2^{56}$$

possible passwords. We'll also assume that passwords are stored in a password file that contains 2^{10} hashed passwords, and that Trudy has a dictionary of 2^{20} common passwords. From experience, Trudy expects that any given password will appear in her dictionary with a probability of about 1/4. Also, work is measured by the number of hashes computed. Note that comparisons are free—only hash calculations count as work.

Under these assumptions, we'll determine the probability of successfully cracking a password in each of the following four cases.

I. Trudy wants to determine Alice's password (perhaps Alice is the administrator). Trudy does not use her dictionary of likely passwords.

II. Trudy wants to determine Alice's password. Trudy does use her dictionary of common passwords.

III. Trudy will be satisfied to crack any password in the password file, without using her dictionary.

IV. Trudy wants to find any password in the hashed password file, using her dictionary.

In each case, we'll consider both salted and unsalted passwords.

Case I: Trudy has decided that she wants to crack Alice's password. Trudy, who is somewhat absent-minded, has forgotten that she has a password dictionary available. Without a dictionary of common passwords, Trudy has no choice other than a brute force approach. This is precisely equivalent to an exhaustive key search and hence the expected work is

$$2^{56}/2 = 2^{55}.$$

[4]Salting password hashes is as close to a free lunch as you'll come in information security. Maybe the connection with a free lunch is why it's called a salt?

The result here is the same whether the passwords are salted or not, unless someone has precomputed, sorted, and stored the hashes of all possible passwords. If the hashes of all passwords are already known, then in the unsalted case, there is no work at all—Trudy simply looks up the hash value and finds the corresponding password. But, if the passwords are salted, there is no benefit to having the password hashes. In any case, precomputing all possible password hashes is a great deal of work, so for the remainder of this discussion, we'll assume this is infeasible.

Case II: Trudy again wants to recover Alice's password, and she is going to use her dictionary of common passwords. With probability 1/4, Alice's password is in Trudy's dictionary. Suppose the passwords are salted. Furthermore, suppose Alice's password is in Trudy's dictionary. Then Trudy would expect to find Alice's password after hashing half of the words in the dictionary, that is, after 2^{19} tries. With probability 3/4 the password is not in the dictionary, in which case Trudy would expect to find it after about 2^{55} tries. Combining these cases gives Trudy an expected work of

$$\frac{1}{4}\left(2^{19}\right) + \frac{3}{4}\left(2^{55}\right) \approx 2^{54.6}.$$

Note that the expected work here is almost the same as in Case I, where Trudy did not use her dictionary. However, in practice, Trudy would simply try all the words in her dictionary and quit if she did not find Alice's password. Then the work would be at most 2^{20} and the probability of success would be 1/4.

If the passwords are unsalted, Trudy could precompute the hashes of all 2^{20} passwords in her dictionary. Then this small one-time work could be amortized over the number of times that Trudy uses this attack. That is, the larger the number of attacks, the smaller the average work per attack.

Case III: In this case, Trudy will be satisfied to determine any of the 1024 passwords in the hashed password file. Trudy has again forgotten about her password dictionary.

Let $y_0, y_1, \ldots, y_{1023}$ be the password hashes. We'll assume that all 2^{10} passwords in the file are distinct. Let $p_0, p_1, \ldots, p_{2^{56}-1}$ be a list of all 2^{56} possible passwords. As in the brute force case, Trudy needs to make 2^{55} distinct comparisons before she expects to find a match.

If the passwords are not salted, then Trudy can compute $h(p_0)$ and compare it with each y_i, for $i = 0, 1, 2, \ldots, 1023$. Next she computes $h(p_1)$ and compares it with all y_i and so on. The point here is that each hash computation provides Trudy with 2^{10} comparisons. Since work is measured in terms of hashes, not comparisons, and 2^{55} comparisons are needed, the expected work is

$$2^{55}/2^{10} = 2^{45}.$$

Now suppose the passwords are salted. Let s_i denote the salt value corresponding to hash password y_i. Then Trudy computes $h(p_0, s_0)$ and compares it with y_0. Next, she computes $h(p_0, s_1)$ and compares it with y_1, she computes $h(p_0, s_2)$ and compares it with y_2, and she continues in this manner up to $h(p_0, s_{1023})$. Then Trudy must repeat this entire process with password p_1 in place of p_0, and then with password p_2 and so on. The bottom line is that each hash computation only yields one comparison and consequently the expected work is 2^{55}, which is the same as in Case I above.

This case illustrates the benefit of salting passwords. However, Trudy has not made use of her password dictionary, which is unrealistic.

Case IV: Finally, suppose that Trudy will be satisfied to recover any one of the 1024 passwords in the hashed password file, and she will make use of her password dictionary. First, note that the probability that at least one of the 1024 passwords in the file appears in Trudy's dictionary is

$$1 - \left(\frac{3}{4}\right)^{1024} \approx 1.$$

Therefore, we can safely ignore the case where no password from the file is in Trudy's dictionary.

If the passwords are not salted, then Trudy could simply hash all password in her dictionary and compare the results to all 1024 hashes in the password file. Since we are certain that at least one of these passwords is in the dictionary, Trudy's work is 2^{20} and she is assured of finding at least one password. However, if Trudy is a little more clever, she can greatly reduce this meager work factor. Again, we can safely assume that at least one of the passwords is in Trudy's dictionary. Consequently, Trudy only needs to make about 2^{19} comparisons—half the size of her dictionary—before she expects to find a password. As in Case III, each hash computation yields 2^{10} comparisons, so the expected work is only

$$2^{19}/2^{10} = 2^9.$$

Finally, note that in this unsalted case, if the hashes of the dictionary passwords have been precomputed, no additional work is required to recover one (or more) passwords. That is, Trudy simply compares the hashes in the file to the hashes of her dictionary passwords and, in the process, she recovers any passwords that appear in her dictionary.

Now we consider the most realistic case—Trudy has a dictionary of common passwords, she will be happy to recover any password from the password file, and the passwords in the file are salted. For this case, we let $y_0, y_1, \ldots, y_{1023}$ be the password hashes and $s_0, s_1, \ldots, s_{1023}$ be the corresponding salt values. Also, let $d_0, d_1, \ldots, d_{2^{20}-1}$ be the dictionary words. Suppose that Trudy first computes $h(d_0, s_0)$ and compares it to y_0, then she

compute $h(d_1, s_0)$ and compares it to y_0, then she compute $h(d_2, s_0)$ and compares it to y_0, and so on. That is, Trudy first compares y_0 to all of her (hashed) dictionary words. Of course, she must use salt s_0 for these hashes. If she does not recover the password corresponding to y_0, then she repeats the process using y_1 and s_1, and so on.

Note that if y_0 is in the dictionary (which has probability 1/4), Trudy expects to find it after about 2^{19} hashes, while if it is not in the dictionary (probability 3/4) Trudy will compute 2^{20} hashes. If Trudy finds y_0 in the dictionary, then she's done. If not, Trudy will have computed 2^{20} hashes before she moves on to consider y_1. Continuing in this manner, we find that the expected work is about

$$\frac{1}{4}\left(2^{19}\right) + \frac{3}{4}\cdot\frac{1}{4}\left(2^{20}+2^{19}\right) + \left(\frac{3}{4}\right)^2\frac{1}{4}\left(2\cdot 2^{20}+2^{19}\right) + \cdots + \left(\frac{3}{4}\right)^{1023}\frac{1}{4}\left(1023\cdot 2^{20}+2^{19}\right) < 2^{22}.$$

This is somewhat disappointing, since it shows that, for very little work, Trudy can expect to crack at least one password.

It can be shown (see Problems 24 and 25) that, under reasonable assumptions, the work needed to crack a (salted) password is approximately equal to size of the dictionary divided by the probability that a given password is in the dictionary. In our example here, the size of the dictionary is 2^{20} while the probability of finding a password is 1/4. So, the expected work should be about

$$\frac{2^{20}}{1/4} = 2^{22}$$

which is consistent with the calculation above. Note that this approximation implies that we can increase Trudy's work by forcing her to have a larger dictionary or by decreasing her probability of success (or both), which makes intuitive sense. Of course, the obvious way to accomplish this is to choose passwords that are harder to guess.

The inescapable conclusion is that password cracking is too easy, particularly in situations where one weak password is sufficient to break the security of an entire system. Unfortunately, when it comes to passwords, the numbers strongly favor the bad guys.

7.3.6 Other Password Issues

As bad as it is, password cracking is only the tip of the iceberg when it comes to problems with passwords. Today, most users need multiple passwords, but users can't (or won't) remember a large number of passwords. This results in a significant amount of password reuse, and any password is only as secure

as the least secure place it's used. If Trudy finds one of your passwords, she would be wise to try it (and slight variations of it) in other places where you use a password.

Social engineering is also a major concern with passwords.[5] For example, if someone calls you, claiming to be a system administrator who needs your password to correct a problem with your account, would you give away your password? According to a recent survey, 34% of users will give away their password if you ask for it, and the number increases to 70% if you offer a candy bar as incentive [232].

Keystroke logging software and similar spyware are also serious threats to password-based security [22]. The failure to change default passwords is a major source of attacks as well [339].

An interesting question is, who suffers from bad passwords? The answer is that it depends. If you choose your birthday for your ATM PIN number, only you stand to lose.[6] On the other hand, if you choose a weak password at work, the entire company stands to lose. This explains why banks usually let users choose any PIN number they desire for their ATM cards, but companies generally try to force users to select reasonably strong passwords.

There are many popular password cracking tools including L0phtCrack [2] (for Windows) and John the Ripper [157] (for Unix). These tools come with preconfigured dictionaries, and it is easy to produce customized dictionaries. These are good examples of the types of tools that are available to hackers.[7] Since virtually no skill is required to leverage these powerful tools, the door to password cracking is open to all, regardless of ability.

Passwords are one of the most severe real-world security problems today, and this is unlikely to change any time soon. The bad guys clearly have the advantages when it comes to passwords. In the next section, we'll look at biometrics, which—together with smartcards ad similar devices—are often touted as the best way to escape from the multitude of problems inherent with passwords.

[5]Actually, social engineering is a major concern in all aspects of information security where humans play a role. Your all-too-human author heard a talk about penetration testing, where the tester was paid to probe the security of a major corporation. The tester lied and forged a (non-digital) signature to obtain entry into corporate headquarters, where he posed as a system administrator trainee. Secretaries and other employees were more than happy to accept "help" from this fake SA trainee. As a result, the tester claimed to have obtained almost all of the company's intellectual property (including such sensitive information as the design of nuclear power plants) within two days. This attack consisted almost entirely of social engineering.

[6]Perhaps the bank will lose too, but only if you live in the United States and you have a very good lawyer.

[7]Of course, almost every hacker tool has legitimate uses. For example, password cracking tools are valuable for system administrators, since they can use these tools to test the strength of the passwords on their system.

7.4 Biometrics

You have all the characteristics of a popular politician:
a horrible voice, bad breeding, and a vulgar manner.
— Aristophanes

Biometrics represent the "something you are" method of authentication or, as Schneier so aptly puts it, "you are your key" [260]. There are many different types of biometrics, including such long-established methods as fingerprints. Recently, biometrics based on speech recognition, gait (walking) recognition, and even a digital doggie (odor recognition) have been developed. Biometrics are currently a very active topic for research [151, 176].

In the information security arena, biometrics are seen as a more secure alternative to passwords. For biometrics to be a practical replacement for passwords, cheap and reliable systems are needed. Today, usable biometric systems exist, including laptops using thumbprint authentication, palm print systems for secure entry into restricted facilities, the use of fingerprints to unlock car doors, and so on. But given the potential of biometrics—and the well-known weaknesses of password-based authentication—it's perhaps surprising that biometrics are not more widely used.

An ideal biometric would satisfy all of the following:

- Universal — A biometric should apply to virtually everyone. In reality, no biometric applies to everyone. For example, a small percentage of people do not have readable fingerprints.
- Distinguishing — A biometric should distinguish with virtual certainty. In reality, we can't hope for 100% certainty, although, in theory, some methods can distinguish with very low error rates.
- Permanent — Ideally, the physical characteristic being measured should never change. In practice, it's sufficient if the characteristic remains stable over a reasonably long period of time.
- Collectable — The physical characteristic should be easy to collect without any potential to cause harm to the subject. In practice, collectability often depends heavily on whether the subject is cooperative or not.
- Reliable, robust, and user-friendly — These are just some of the additional real-world considerations for a practical biometric system. Some biometrics that have shown promise in laboratory conditions have subsequently failed to deliver similar performance in practice.

Biometrics are also applied in various *identification* problems. In the identification problem we are trying to answer the question "Who are you?," while

for the authorization problem, we want to answer the question, "Are you who you say you are?" That is, in identification, the goal is to identify the subject from a list of many possible subjects. This occurs, for example, when a suspicious fingerprint from a crime scene is sent to the FBI fingerprint database for comparison with all of the millions of fingerprint records currently on file.

In the identification problem, the comparison is one-to-many whereas for authentication, the comparison is one-to-one. For example, if someone claiming to be Alice uses a thumbprint mouse biometric, the captured thumbprint image is only compared with the stored thumbprint of Alice. The identification problem is inherently more difficult and subject to a much higher error rate due to the larger number of comparisons that must be made. That is, each comparison carries with it a probability of an error, so the more comparisons required, the higher the error rate.

There are two phases to a biometric system. First, there is an *enrollment phase*, where subjects have their biometric information gathered and entered into a database. Typically, during this phase very careful measurement of the pertinent physical information is required. Since this is one-time work (per subject), it's acceptable if the process is slow and multiple measurements are required. In some fielded systems, enrollment has proven to be a weak point since it may be difficult to obtain results that are comparable to those obtained under laboratory conditions.

The second phase in a biometric system is the *recognition phase*. This occurs when the biometric detection system is used in practice to determine whether (for the authentication problem) to authenticate the user or not. This phase must be quick, simple, and accurate.

We'll assume that subjects are cooperative, that is, they're willing to have the appropriate physical characteristic measured. This is a reasonable assumption in the authentication case, since authentication is generally required for access to certain information resources or for entry into an otherwise restricted area.

For the identification problem, it is often the case that subjects are uncooperative. For example, consider a facial recognition system used for identification. Las Vegas casinos use such systems to detect known cheaters as they attempt to enter a casino [300]. Another fanciful proposed use of facial recognition is to spot terrorists in airports.[8] In such cases, the enrollment conditions may be far from ideal, and in the recognition phase, the subjects are certainly uncooperative as they likely do everything possible to avoid detection. Of course, uncooperative subjects can only serve to make the underlying biometric problem more difficult. For the remainder of this discussion we'll focus on the authentication problem and we'll assume that the subjects are cooperative.

[8] Apparently, terrorists are welcome in casinos, as long as they don't cheat.

7.4.1 Types of Errors

There are two types of errors that can occur in biometric recognition. Suppose Bob poses as Alice and the system mistakenly authenticates Bob as Alice. The rate at which such misauthentication occurs is the *fraud rate*. Now suppose that Alice tries to authenticate as herself, but the system fails to authenticate her. The rate at which this type of error occurs is the *insult rate* [14].

For any biometric, we can decrease the fraud or insult rate at the expense of the other. For example, if we require a 99% voiceprint match, then we can obtain a low fraud rate, but the insult rate will be high, since a speaker's voice will naturally change slightly from time to time. On the other hand, if we set the threshold at a 30% voiceprint match, the the fraud rate will likely be high, but the system will have a low insult rate.

The *equal error rate* is the rate for which the fraud and insult rates are the same. That is, the parameters of the system are adjusted until the fraud rate and insult rate are precisely in balance. This is a useful measure for comparing different biometric systems.

7.4.2 Biometric Examples

In this section, we'll briefly discuss three common biometrics. First, we'll consider fingerprints, which, in spite of their long history, are relative newcomers in computing applications. Then we'll discuss palm prints and iris scans.

7.4.2.1 Fingerprints

Fingerprints were used in ancient China as a form of signature, and they have served a similar purpose at other times in history. But the use of fingerprints as a scientific form of identification is a much more recent phenomenon.

A significant analysis of fingerprints occurred in 1798 when J. C. Mayer suggested that fingerprints might be unique. In 1823, Johannes Evangelist Purkinje discussed nine fingerprint patterns, but this work was a biological treatise and did not suggest using fingerprints as a form of identification. The first modern use of fingerprints for identification occurred in 1858 in India, when Sir William Hershel used palm prints and fingerprints as forms of signatures on contracts.

In 1880, Dr. Henry Faulds published an article in *Nature* that discussed the use of fingerprints for identification purposes. In Mark Twain's *Life on the Mississippi*, which was published in 1883, a murderer is identified by a fingerprint. However, the widespread use of fingerprinting only became possible in 1892 when Sir Francis Galton developed a classification system based on "minutia" that enabled efficient searching, and he verified that fingerprints do not change over time [188].

Examples of the different types of minutia in Galton's classification system appear in Figure 7.1. Galton's system allowed for an efficient solution to the identification problem in the pre-computer era.[9]

Figure 7.1: Examples of Galton's Minutia

Today, fingerprints are routinely used for identification, particularly in criminal cases. It is interesting to note that the standard for determining a match varies widely. For example, in Britain fingerprints must match in 16 points, whereas in the United States, no fixed number of points are required to match.[10]

A fingerprint biometric works by first capturing an image of the fingerprint. The image is then enhanced using various image-processing techniques, and various points are identified and extracted from the enhanced image. This process is illustrated in Figure 7.2.

Figure 7.2: Automatic Extraction of Minutia

The points extracted by the biometric system are compared in a manner that is somewhat analogous to the manual analysis of fingerprints. For authentication, the extracted points are compared with the claimed user's stored

[9]Fingerprints were classified into one of 1024 "bins." Then, given a fingerprint from an unknown subject, a binary search based on the minutia quickly focused the effort of matching the print on one of these bins. Consequently, only a very small subset of recorded fingerprints needed to be carefully compared to the unknown fingerprint.

[10]This is a fine example of the way that the U.S. generously ensures full employment for lawyers—they can always argue about whether fingerprint evidence is admissible or not.

information, which was previously captured during the enrollment phase. The system then determines whether a statistical match occurs, with some predetermined level of confidence. This fingerprint comparison process is illustrated in Figure 7.3.

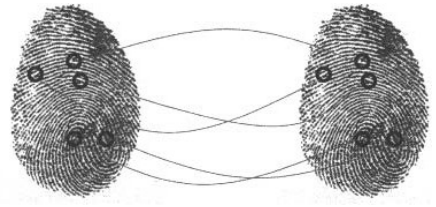


Figure 7.3: Minutia Comparison

7.4.2.2 Hand Geometry

Another popular biometric is hand geometry, which is particularly popular for entry into secure facilities [138, 256]. In this system, the shape of the hand is carefully measured, including the width and length of the hand and fingers.[11] The paper [152] describes 16 such measurements, of which 14 are illustrated in Figure 7.4 (the other two measure the thickness of the hand). Human hands are not nearly as unique as fingerprints, but hand geometry is easy and quick to measure, while being sufficiently robust for many authentication uses. However, hand geometry would probably not be suitable for identification, since the number of false matches would be high.

One advantage of hand geometry systems is that they are fast, taking less than one minute in the enrollment phase and less than five seconds in the recognition phase. Another advantage is that human hands are symmetric, so if the enrolled hand is, say, in a cast, the other hand can be used by placing it palm side up. Some disadvantages of hand geometry include that it cannot be used on the young or the very old, and, as we'll discuss in a moment, the system has a relatively high equal error rate.

7.4.2.3 Iris Scan

A biometric that is, in theory, one of the best for authentication is the iris scan. The development of the iris (the colored part of the eye) is chaotic, which implies that minor variations lead to large differences. There is little or no genetic influence on the iris pattern, so that the measured pattern

[11]Note that palm print systems do not read your palm. For that, you'll have to see your local chiromancer.

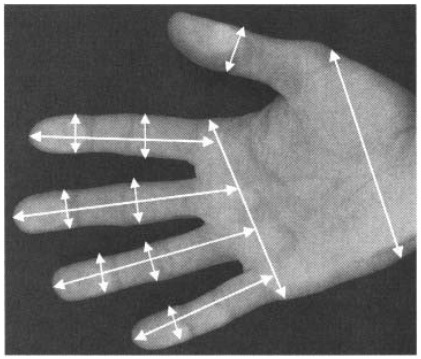


Figure 7.4: Hand Geometry Measurements

is uncorrelated for identical twins and even for the two eyes of one individual. Another desirable property is that the pattern is stable throughout a lifetime [149].

The development of iris scan technology is relatively new. In 1936, the idea of using the human iris for identification was suggested by Frank Burch. In the 1980s, the idea resurfaced in James Bond films, but it was not until 1986 that the first patents appeared—a sure sign that people foresaw money to be made on the technology. In 1994, John Daugman, a researcher at Cambridge University, patented what is generally accepted as the best approach currently available [76].

Iris scan systems require sophisticated equipment and software. First, an automated iris scanner locates the iris. Then a black and white photo of the eye is taken. The resulting image is processed using a two-dimensional wavelet transform, the result of which is a 256-byte (that is, 2048-bit) iris code.

Two iris codes are compared based on the Hamming distance between the codes. Suppose that Alice is trying to authenticate using an iris scan. Let x be the iris code computed from Alice's iris in the recognition phase, while y is Alice's iris code stored in the scanner's database, which was gathered during the enrollment phase. Then x and y are compared by computing the distance $d(x, y)$ defined by

$$d(x,y) = \frac{\text{number of non-match bits}}{\text{number of bits compared}}. \tag{7.1}$$

For example, $d(0010, 0101) = 3/4$ and $d(101111, 101001) = 1/3$.

For an iris scan, $d(x, y)$ is computed on the 2048-bit iris code. A perfect match yields $d(x, y) = 0$, but we can't expect perfection in practice. Under

laboratory conditions, for the same iris the expected distance is 0.08, and for different irises the expect distance is 0.50. The usual thresholding scheme is to accept the comparison as a match if the distance is less than 0.32 and otherwise consider it a non-match [76]. An image of an iris appears in Figure 7.5.

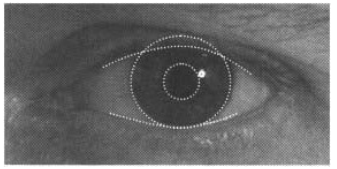


Figure 7.5: An Iris Scan

Define the *match* cases to be those where, for example, Alice's data from the enrollment phase is compared to her scan data from the recognition phase. Define the *no-match* cases to be when, for example, Alice's enrollment data is compared to Bob's recognition phase data (or vice versa). Then the left histogram in Figure 7.6 represents match data, while the right histogram represents no-match data. Note that the match data provides information relevant to the insult rate, whereas the no-match data provides information relevant to the fraud rate.

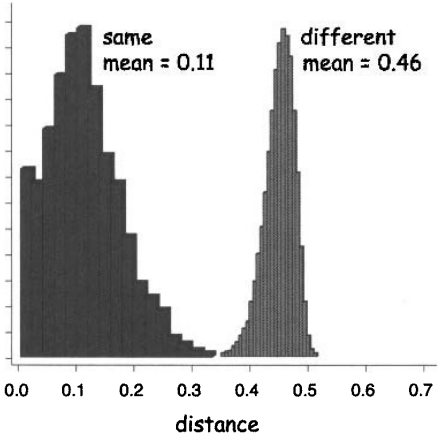


Figure 7.6: Histogram of Iris Scan Results [149]

The iris scan is often cited as the ultimate biometric for authentication. The histogram in Figure 7.6, which is based on 2.3 million comparisons, tends

to support this view, since the overlapping region between the "same" (match) and "different" (no-match) cases appears to be virtually nonexistent. Note that the overlap represents the region where an error can occur. In reality, there is some overlap between the histograms in in Figure 7.6, but the overlap is extremely small.

The iris scan distances for the match data in Table 7.1 provide a more detailed view of the same histogram marked as "same" in Figure 7.6. From Figure 7.6, we see that the equal error rate (which corresponds to the crossover point between the two graphs) occurs somewhere near distance 0.34. From Table 7.1, this implies an equal error rate of about 10^{-5}. For this biometric, we would certainly be willing to tolerate a slightly higher insult rate since that would further reduce the fraud rate. Hence, the typical threshold used is 0.32, as mentioned above.

Table 7.1: Iris Scan Match Scores and Error Rates [149]

Score	Probability
0.29	1 in 1.3×10^{10}
0.30	1 in 1.5×10^{9}
0.31	1 in 1.8×10^{8}
0.32	1 in 2.6×10^{7}
0.33	1 in 4.0×10^{6}
0.34	1 in 6.9×10^{5}
0.35	1 in 1.3×10^{5}

Is it possible to attack an iris-scanning system? Suppose Bob has a good photo of Alice's eye. Then he can claim to be Alice and try to use the photo to trick the system into authenticating him as Alice. This attack is not at all far fetched. In fact, an Afghan woman whose photo appeared on a famous *National Geographic* magazine cover in 1984 was positively identified 17 years later by comparing her then-current iris scan with an iris scan taken from the 1984 photo. The woman had never seen the magazine, but she did recall being photographed. The magazine cover with the woman's photo and the fascinating story of finding this person after years of war and chaos in Afghanistan can be found at [28].

To prevent attacks based on a photo, an iris-scanning system could first shine a light on the "eye" and verify that the pupil contracts before proceeding with the iris scan. While this eliminates an attack that relies on a static photo, it also might significantly increase the cost of the system. Given that biometrics are in competition with passwords, and passwords are free, cost is always an issue.

7.4.3 Biometric Error Rates

Recall that the equal error rate—the point at which the fraud rate equals the insult rate—is generally considered the best measure for comparing different biometric systems. The equal error rates for several popular biometrics are given in Table 7.2.

Table 7.2: Biometric Equal Error Rates [32]

Biometric	Equal error rate
fingerprint	2.0×10^{-3}
hand geometry	2.0×10^{-3}
voice recognition	2.0×10^{-2}
iris scan	7.6×10^{-6}
retina scan	1.0×10^{-7}
signature recognition	2.0×10^{-2}

For fingerprint biometric systems, the equal error rate may seem high. However, most fingerprint biometrics are relatively cheap devices that do not achieve anything near the theoretical potential for fingerprint matching. On the other hand, hand geometry systems are relatively expensive and sophisticated devices, so they probably do achieve something close to the theoretical potential.

In theory, iris scanning has an equal error rate of about 10^{-5}. But to achieve such spectacular results, the enrollment phase must be extremely accurate. If the real-world enrollment environment is not up to laboratory standards, then the results might not be so impressive.

Undoubtedly many inexpensive biometrics systems fare worse than the results given in Table 7.2. And biometrics in general have a very poor record with respect to the inherently difficult identification problem.

7.4.4 Biometric Conclusions

Biometrics clearly have many potential advantages over passwords. In particular, biometrics are difficult, although not impossible, to forge. In the case of fingerprints, Trudy could steal Alice's thumb, or, in a less gruesome attack, Trudy might be able to use a copy of Alice's fingerprint. Of course, a more sophisticated system might be able to detect such an attack, but then the system will be more costly, thereby reducing its desirability as a replacement for passwords.[12]

[12]Unfortunately for security, passwords are likely to remain free for the foreseeable future.

There are also many potential software-based attacks on authentication. For example, it may be possible to subvert the software that does the comparison or to manipulate the database that contains the enrollment data. Such attacks apply to most authentication systems, regardless of whether they are based on biometrics, passwords, or other techniques.

While a broken cryptographic key or password can be revoked and replaced, it's not clear how to revoke a broken biometric. This and other biometric pitfalls are discussed by Schneier [260].

Biometrics have a great deal of potential as a substitute for passwords, but biometrics are not foolproof. And given the enormous problems with passwords and the vast potential of biometrics, it's perhaps surprising that biometrics are not more widely used today. This should change in the future as biometrics become more robust and inexpensive.

7.5 Something You Have

Smartcards or other hardware tokens can be used for authentication. Such authentication is based on the "something you have" principle. A smartcard is a credit card sized device that includes a small amount of memory and computing resources, so that it is able to store cryptographic keys or other secrets, and perhaps even do some computations on the card. A special-purpose smartcard reader, as shown in Figure 7.7, is used to read the key stored on the card. Then the key can be used to authenticate the user. Since a key is used, and keys are selected at random, password guessing attacks can be eliminated.[13]

Figure 7.7: A Smartcard Reader (Courtesy of Athena, Inc.)

There are several other examples of authentication based on "something you have," including a laptop computer (or its MAC address), an ATM card, or a password generator. Here, we give an example of a password generator.

[13]Well, a PIN might be required to access the key, so password issues might still arise.

A password generator is a small device that the user must have (and use) to log in to a system. Suppose that Alice has a password generator, and she wants to authenticate herself to Bob. Bob sends a random "challenge" R to Alice, which Alice then inputs into the password generator along with her PIN number. The password generator then produces a response, which Alice transmits to Bob. If the response is correct, Bob is convinced that he's indeed talking to Alice, since only Alice is supposed to have the password generator. This process is illustrated in Figure 7.8.

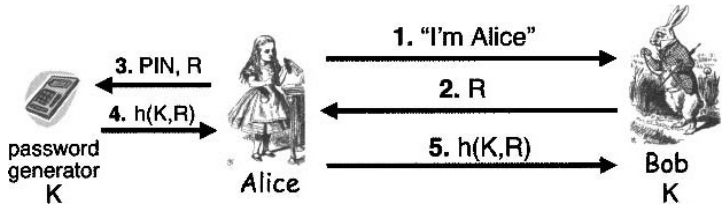


Figure 7.8: Password Generator

For a challenge-response authentication scheme to work, Bob must be able to verify that Alice's response is correct. For the example in Figure 7.8, Bob and the password generator must both have access to the key K, since the password generator needs the key to compute the hash, and Bob needs the key to verify Alice's response. Alice accesses the key K only indirectly—by entering her PIN into the key generator. We'll see more examples of the use of challenge-response mechanisms in the upcoming chapters on security protocols.

7.6 Two-Factor Authentication

In fact, the password generator scheme in Figure 7.8 requires both "something you have" (the password generator) and "something you know" (the PIN). Any authentication method that requires two out of the three "somethings" is known as *two-factor authentication*. Another example of a two-factor authentication is an ATM card, where the user must have the card and know the PIN number. Other examples of two-factor authentication include a credit card together with a signature, a biometric thumbprint system that also requires a password, and a cell phone that requires a PIN.

7.7 Single Sign-On and Web Cookies

Before concluding this chapter, we briefly mention two additional authentication topics. First, we discuss *single sign-on*, which is a topic of considerable

practical importance. We'll also briefly mention *Web cookies*, which are often used as a weak form of authentication.

Users find it troublesome to enter their authentication information (typically, passwords) repeatedly. For example, when browsing the Web, it is not uncommon that many different websites require passwords. While this is sensible from a security perspective, it places a burden on users who must either remember different passwords for many different websites or compromise their security by reusing passwords.

A more convenient solution would be to have Alice authenticate once and then have a successful result automatically "follow" her wherever she goes on the Internet. That is, the initial authentication would require Alice's participation, but subsequent authentications would happen behind the scenes. This is known as single sign-on, and single sign-on for the Internet has been a topic of some interest for several years.

As with many computing topics, there are competing and incompatible approaches to single sign-on for the Internet. As is often the case, there is the Microsoft way, and the "everybody else" way. The approach favored by Microsoft goes by the name of Passport [171, 203], while the method preferred by (nearly) everybody else is the Liberty Alliance [100, 192]. The latter approach is based on the Security Assertion Markup Language, or SAML [78].

Certainly, a secure single sign-on for the Internet would be a major convenience. However, it does not appear that any such method is likely to gain widespread acceptance any time soon. It is worth noting that we will see a single sign-on architecture in Chapter 10 when we discuss the Kerberos security protocol.

Finally, we mention Web cookies, which have some interesting security implications. When Alice is surfing the Web, websites often provide Alice's browser with a Web cookie, which is simply a numerical value that is stored and managed by Alice's browser. The website also stores the cookie, which is used to index a database that retains information about Alice.

When Alice returns to a website for which she has a cookie, the cookie is automatically passed by her browser to the website. The website can then access its database to remember important information about Alice. In this way, cookies maintain state across sessions. Since the Web uses HTTP, which is a stateless protocol, cookies are also used to maintain state within a session.

In a sense, cookies can act as a single sign-on method for a website. That is, a website can authenticate "Alice" based on the possession of Alice's Web cookie. Or, in a slightly stronger version, a password is used to initially authenticate Alice, after which the cookie is considered sufficient. Either way, this is a fairly weak form of authentication, but it illustrates the often irresistible temptation to use whatever is available and convenient as a security mechanism, whether it is secure or not.

7.8 Summary

You can authenticate to a machine based on "something you know," "something you have," or "something you are." Passwords are synonymous with the "something you know" method of authentication. In this chapter, we discussed passwords at length. The bottom line is that passwords are far from an ideal method of authentication, but they are likely to remain popular for the foreseeable future, since passwords are the lowest cost option.

We also discussed authentication based on "something you are," i.e., biometrics. It is clear that biometrics offer the potential for much higher security than passwords. However, biometrics cost money, and they are not entirely without problems.

We briefly mentioned "something you have" methods of authentication, as well as two-factor authentication, which combines any two of the three methods. Finally, we briefly discussed single sign-on and Web cookies.

In the next chapter, we'll discuss authorization, which deals with restrictions placed on authenticated users. The authentication problem returns to the fore in Chapters 9 and 10, where we cover security protocols. We'll see that authentication over a network is a whole nother can of worms.

7.9 Problems

1. As discussed in this chapter, relatively strong passwords can be derived from passphrases.

   a. Give two passwords derived from the passphrase "Gentlemen do not read other gentlemen's mail."

   b. Give two passwords derived from the passphrase "Are you who you say you are?"

2. For each of the following passwords, give a passphrase that the password could have been derived from.

   a. `PokeGCTall`

   b. `4s&7yrsa`

   c. `gimmeliborD`

   d. `IcntgetNOsat`

3. In the context of biometrics, define the terms *fraud rate* and *insult rate*. In statistics, which of these is a Type I error and which is a Type II error?

4. In some applications, a passcode consisting of some number of digits is required (for example, a PIN). Using the number-to-letter conversion on a telephone,

   a. What passcode corresponds to the password "hello"?

   b. Find as many passwords as you can that correspond to the passcode 5465, where each password is an English dictionary word.

5. Suppose that on a particular system, all passwords are 10 characters, there are 64 choices for each character, and the system has a password file containing 512 hashed passwords. Furthermore, Trudy has a dictionary of 2^{20} common passwords. Provide pseudo-code for an efficient attack on the password file in the following cases.

   a. The password hashes are not salted.

   b. The password hashes are salted.

6. This problem deals with storing passwords in a file.

   a. Why is it a good idea to hash passwords that are stored in a file?

   b. Why is it a much better idea to hash passwords stored in a file than to encrypt the password file?

   c. What is a salt and why should a salt be used whenever passwords are hashed?

7. On a particular system, all passwords are 8 characters, there are 128 choices for each character, and there is a password file containing the hashes of 2^{10} passwords. Trudy has a dictionary of 2^{30} passwords, and the probability that a randomly selected password is in her dictionary is 1/4. Work is measured in terms of the number of hashes computed.

   a. Suppose that Trudy wants to recover Alice's password. Using her dictionary, what is the expected work for Trudy to crack Alice's password, assuming the passwords are not salted?

   b. Repeat part a, assuming the passwords are salted.

   c. What is the probability that at least one of the passwords in the password file appears in Trudy's dictionary?

8. Suppose you are a merchant and you decide to use a biometric fingerprint device to authenticate people who make credit card purchases at your store. You can choose between two different systems: System A has a fraud rate of 1% and an insult rate of 5%, while System B has a fraud rate of 5% and an insult rate of 1%.

   a. Which system is more secure and why?

   b. Which system is more user-friendly and why?

   c. Which system would you choose and why?

9. Research has shown that most people cannot accurately identify an individual from a photo. For example, one study found that most people will accept an ID with any photo that has a picture of a person of the same gender and race as the presenter.

   a. It has also been demonstrated that when photos are included on credit cards, the fraud rate drops significantly. Explain this apparent contradiction.

   b. Your easily amused author frequents an amusement park that provides each season passholder with a plastic card similar to a credit card. The park takes a photo of each season passholder, but the photo does not appear on the card. Instead, when the card is presented for admission to the park, the photo appears on a screen that is visible to the park attendant. Why might this approach be better than putting the photo on the card?

10. Suppose all passwords on a given system are 8 characters and that each character can be any one of 64 different values. The passwords are hashed (with a salt) and stored in a password file. Now suppose Trudy has a password cracking program that can test 64 passwords per second. Trudy also has a dictionary of 2^{30} common passwords and the probability that any given password is in her dictionary is 1/4. The password file on this system contains 256 password hashes.

    a. How many different passwords are possible?

    b. How long, on average, will it take Trudy to crack the administrator's password?

    c. What is the probability that at least one of the 256 passwords in the password file is in the dictionary?

    d. What is the expected work for Trudy to recover any one of the passwords in the password file?

11. Let h be a secure cryptographic hash function. For this problem, a password consists of a maximum of 14-characters and there are 32 possible choices for each character. If a password is less than 14 characters, it's padded with nulls until it is exactly 14 characters. Let P be the resulting 14 character password. Consider the following two password hashing schemes.

    (i) The password P is split into two parts, with X equal to the first 7 characters and Y equal to the last 7 characters. The password is stored as $(h(X), h(Y))$. No salt is used.

(ii) The password is stored as $h(P)$. Again, no salt is used.

Note that the method in scheme (i) is used in Windows to store the so-called LANMAN password.

a. Assuming a brute force attack, how much easier is it to crack the password if scheme (i) is used as compared with scheme (ii)?

b. If scheme (i) is used, why might a 10-character password be *less* secure than a 7-character password?[14]

12. Suppose that passwords are stored as follows, where there are 128 possible choices for each character: If a password exceeds 16 characters, it is truncated to 16 characters. If a password is less than 16 characters, it is padded with "A" until it is exactly 16 characters. The resulting 16-character password is split into two parts, X_0 and X_1, where X_0 consists of the first six characters and X_1 consists of the last 10 characters. The password is hashed as $Y_0 = h(X_0, S_0)$ and $Y_1 = h(X_1, S_1)$, where S_0 and S_1 are each 64-bit salt values. The values (Y_0, S_0) and (Y_1, S_1) are stored for use in password verification.

a. Precisely how are (Y_0, S_0) and (Y_1, S_1) used to verify an entered password?

b. What is the expected work for an exhaustive search to recover one particular password (for example, the administrator's password)?

c. How would you attack a password in a way that could provide a significant shortcut over an exhaustive search or a standard dictionary attack? Explain.

13. Many websites require users to register before they can access information or services. Suppose that you register at such a website, but when you return later you've forgotten your password. The website then asks you to enter your email address, which you do. Later, you receive your original password via email.

a. Discuss several security concerns with this approach to dealing with forgotten passwords.

b. The correct way to deal with passwords is to store salted hashes of passwords. Does this website use the correct approach? Justify your answer.

[14] In fact, the standard advice for LANMAN passwords is that users should choose either a 7-character password, or a 14-character password, since anything in between these two lengths is less secure.

14. Alice forgets her password. She goes to the system administrator's office, and the admin resets her password and gives Alice the new password.

a. Why does the SA reset the password instead of giving Alice her previous (forgotten) password?

b. Why should Alice re-reset her password immediately after the SA has reset it?

c. Suppose that after the SA resets Alice's password, she remembers her previous password. Alice likes her old password, so she resets it to its previous value. Would it be possible for the SA to determine that Alice has chosen the same password as before? Why or why not?

15. Consider the password generator in Figure 7.8.

a. If R is repeated, is the protocol secure?

b. If R is predictable, is the protocol secure?

16. Describe attacks on an authentication scheme based on Web cookies.

17. Briefly outline the most significant technical differences between Passport and Liberty Alliance.

18. MAC address are globally unique and they don't change except in rare instances where hardware changes.

a. Explain how the MAC address on your computer could be used as a "something you have" form of authentication.

b. How could you use the MAC address as part of a two-factor authentication scheme?

c. How secure is your authentication scheme in part a? How much more secure is your authentication scheme in part b?

19. Suppose you have six accounts, each of which requires a password, and you choose distinct passwords for each account.

a. If the probability that any given password is in Trudy's password dictionary is 1/4, what is the probability that at least one of your passwords is in Trudy's dictionary?

b. If the probability that any one of your passwords is in Trudy's dictionary is reduced to 1/10, what is the probability that at least one of your passwords is in Trudy's dictionary?

20. Suppose that you have n accounts, each of which requires a password. Trudy has a dictionary and the probability that a password appears in Trudy's dictionary is p.

    a. If you use the same password for all n accounts, what is the probability that your password appears in Trudy's dictionary?

    b. If you use distinct passwords for each of your n accounts, what is the probability that at least one of your passwords appears in Trudy's dictionary? Show that if $n = 1$, your answer agrees with your answer to part a.

    c. Which is more secure, choosing the same password for all accounts, or choosing different passwords for each account? Why? See also Problem 21.

21. Suppose that Alice uses two distinct passwords—one strong password for sites where she believes security is important (e.g., her online bank), and one weak password for sites where she does not care much about security (e.g., social networking sites).

    a. Alice believes this is a reasonable compromise between security and convenience. What do you think?

    b. What are some practical difficulties that might arise with such an approach?

22. Suppose Alice requires passwords for eight different accounts. She could choose the same password for all of these accounts. With just a single password to remember, Alice might be more likely to choose a strong password. On the other hand, Alice could choose different passwords for each account. With distinct passwords, she might be tempted to choose weaker passwords since this might make it easier for her to remember all of her passwords.

    a. What are the trade-offs between one well-chosen password versus several weaker passwords?

    b. Is there a third approach that is more secure than either of these options?

23. Consider Case I from Section 7.3.5.

    a. If the passwords are unsalted, how much work is it for Trudy to precompute all possible hash values?

    b. If each password is salted with a 16-bit value, how much work is it for Trudy to precompute all possible hash values?

    c. If each password is salted with a 64-bit value, how much work is it for Trudy to precompute all possible hash values?

24. Suppose that Trudy has a dictionary of 2^n passwords and the probability that a given password is in her dictionary is p. If Trudy obtains a file containing a large number of salted password hashes, show that the expected work to recover a password is bounded by $2^{n-1}(1+2(1-p)/p)$. Hint: As in Section 7.3.5, Case IV, ignore the highly improbable case where none of the passwords in the file appears in Trudy's dictionary. Then make use of the fact that $\sum_{k=0}^{\infty} x^k = 1/(1-x)$ and also $\sum_{k=1}^{\infty} kx^k = x/(1-x)^2$, provided $|x| < 1$.

25. For password cracking, generally the most realistic situation is Case IV of Section 7.3.5. In this case, the amount of work that Trudy must do to determine a password depends on the size of the dictionary, the probability that a given password is in the dictionary, and the size of the password file. Suppose Trudy's dictionary is of size 2^n, the probability that a password is in the dictionary is p, and the password file is of size M. Show that if p is small and M is sufficiently large, then Trudy's expected work is about $2^n/p$. Hint: Use the result of Problem 24.

26. Suppose that when a fingerprint is compared with one other (non-matching) fingerprint, the chance of a false match is 1 in 10^{10}, which is approximately the error rate when 16 points are required to determine a match (the British legal standard). Suppose that the FBI fingerprint database contains 10^7 fingerprints.

    a. How many false matches will occur when 100,000 suspect fingerprints are each compared with the entire database?

    b. For any individual suspect, what is the chance of a false match?

27. Suppose DNA matching could be done in real time.

    a. Describe a biometric for secure entry into a restricted facility based on this technique.

    b. Discuss one security concern and one privacy concern with your proposed system in part a.

28. This problem deals with biometrics.

    a. What is the difference between the authentication problem and the identification problem?

    b. Which is the inherently easier problem, authentication or identification? Why?

29. This problem deals with biometrics.

    a. Define fraud rate.

    b. Define insult rate.

    c. What is the equal error rate, how is it determined, and why is it useful?

30. Gait recognition is a biometric that distinguishes based on the way a person walks, whereas a digital doggie is a biometric that distinguishes based on odor.

    a. Describe an attack on gait recognition when it's used for identification.

    b. Describe an attack on a digital doggie when it's used for identification.

31. Recently, facial recognition has been touted as a possible method for, say, identifying terrorists in airports. As mentioned in the text, facial recognition is used by Las Vegas casinos in an attempt to detect cheaters. Note that in both of these cases the biometric is being used for identification (not authentication), presumably with uncooperative subjects.

    a. Discuss an attack on facial recognition when used by a casino to detect cheaters.

    b. Discuss a countermeasure that casinos might employ to reduce the effectiveness of your attack in part a.

    c. Discuss a counter-countermeasure that attackers might employ to reduce the effectiveness of your countermeasure in b.

32. In one episode of the television show *MythBusters*, three successful attacks on fingerprint biometrics are demonstrated [213].

    a. Briefly discuss each of these attacks.

    b. Discuss possible countermeasures for each of the attacks in part a. That is, discuss ways that the biometric systems could be made more robust against the specific attacks.

33. This problem deals with possible attacks on a hand geometry biometric system.

    a. Discuss analogous attacks to those in Problem 32 but for a hand geometry biometric system.

    b. In your judgment, which would be more difficult to break, the fingerprint door lock in Problem 32, or an analogous system based on hand geometry? Justify your answer.

34. A retina scan is an example of a well-known biometric that was not discussed in this chapter.

    a. Briefly outline the history and development of the retina scan biometric. How does a modern retina scan system work?

    b. Why, in principle, can a retina scan be extremely effective?

    c. List several pros and cons of retina scanning as compared to a fingerprint biometric.

    d. Suppose that your company is considering installing a biometric system that every employee will use every time they enter their office building. Your company will install either a retina scan or an iris scan system. Which would you prefer that they choose? Why?

35. A sonogram is a visual representation of sound. Obtain and install a speech analysis tool that can generate sonograms.[15]

    a. Examine several sonograms of your voice, each time saying "open sesame." Qualitatively, how similar are the sonograms?

    b. Examine several sonograms of someone else saying "open sesame." How similar are these sonograms to each other?

    c. In what ways do your sonograms from part a differ from those in part b?

    d. How would you go about trying to develop a reliable biometric based on voice recognition? What characteristics of the sonograms might be useful for distinguishing speakers?

36. This problem deals with possible attacks on an iris scan biometric system.

    a. Discuss analogous attacks to those in Problem 32 on an iris scan biometric system.

    b. Why would it be significantly more difficult to break an iris scan system than the fingerprint door lock in Problem 32?

    c. Given that an iris scan biometric is inherently stronger than a fingerprint-based biometric system, why are fingerprint biometrics far more popular?

[15] Your audacious author uses Audacity [20] to record speech and Sonogram [272] to generate sonograms and analyze the resulting audio files. Both of these are freeware.

37. Suppose that a particular iris scan systems generates 64-bit iris codes instead of the standard 2048-bit iris codes mentioned in this chapter. During the enrollment phase, the following iris codes (in hex) are determined.

User	Iris code
Alice	BE439AD598EF5147
Bob	9C8B7A1425369584
Charlie	885522336699CCBB

During the recognition phase, the following iris codes are obtained.

User	Iris code
U	C975A2132E89CEAF
V	DB9A8675342FEC15
W	A6039AD5F8CFD965
X	1DCA7A54273497CC
Y	AF8B6C7D5E3F0F9A

Use the iris codes above to answer the following questions.

a. Use the formula in equation (7.1) to compute the following distances:

$$d(\text{Alice}, \text{Bob}),\ d(\text{Alice}, \text{Charlie}),\ d(\text{Bob}, \text{Charlie}).$$

b. Assuming that the same statistics apply to these iris codes as the iris codes discussed in Section 7.4.2.3, which of the users, U,V,W,X,Y, is most likely Alice? Bob? Charlie? None of the above?

38. A popular "something you have" method of authentication is the RSA SecurID [252]. The SecureID system is often deployed as a USB key. The algorithm used by SecurID is similar to that given for the password generator illustrated in Figure 7.8. However, no challenge R is sent from Bob to Alice; instead, the current time T (typically, to a resolution of one minute) is used. That is, Alice's password generator computes $h(K, T)$ and this is sent directly to Bob, provided Alice has entered the correct PIN (or password).

a. Draw a diagram analogous to that in Figure 7.8 illustrating the SecurID algorithm.

b. Why do we need T? That is, why is the protocol insecure if we remove T?

c. What are the advantages and disadvantages of using the time T as compared to using a random challenge R?

d. Which is more secure, using a random challenge R or the time T? Why?

39. A password generator is illustrated in Figure 7.8.

a. Discuss possible cryptanalytic attacks on the password generator scheme in Figure 7.8.

b. Discuss network-based attacks on the password generator scheme in Figure 7.8.

c. Discuss possible non-technical attacks on the password generator scheme in Figure 7.8.

40. In addition to the holy trinity of "somethings" discussed in this chapter (something you know, are, or have), it is also possible to base authentication on "something you do." For example, you might need to press a button on your wireless access point to reset it, proving that you have physical access to the device.

a. Give another real-world example where authentication could be based on "something you do."

b. Give an example of two-factor authentication that includes "something you do" as one of the factors.

Chapter 8

Authorization

It is easier to exclude harmful passions than to rule them, and to deny them admittance than to control them after they have been admitted.
— Seneca

You can always trust the information given to you by people who are crazy; they have an access to truth not available through regular channels.
— Sheila Ballantyne

8.1 Introduction

Authorization is the part of access control concerned with restrictions on the actions of authenticated users. In our terminology, authorization is one aspect of access control and authentication is another. Unfortunately, some authors use the term "access control" as a synonym for authorization.

In the previous chapter we discussed authentication, where the issue is one of establishing identity. In its most basic form, authorization deals with the situation where we've already authenticated Alice and we want to enforce restrictions on what she is allowed to do. Note that while authentication is binary (either a user is authenticated or not), authorization can be a much more fine grained process.

In this chapter, we'll extend the traditional notion of authorization to include a few non-traditional topics. We'll discuss CAPTCHAs, which are designed to restrict access to humans (as opposed to computers), and we'll consider firewalls, which can be viewed as a form of access control for networks. We'll follow up the section on firewalls with a discussion of intrusion detection systems, which come into play when firewalls fail to keep the bad guys out.

8.2 A Brief History of Authorization

History is ... bunk.
— Henry Ford

Back in the computing dark ages,[1] authorization was often considered the heart of information security. Today, that seems like a rather quaint notion. In any case, it is worth briefly considering the historical context from which modern information security has arisen.

While cryptography has a long and storied history, other aspects of modern information security are relative newcomers. Here, we take a brief look at the history of system certification, which, in some sense, represents the modern history of authorization. The goal of such certification regimes is to give users some degree of confidence that the systems they use actually provide a specified level of security. While this is a laudable goal, in practice, system certification is often laughable. Consequently, certification has never really become a significant piece of the security puzzle—as a rule, only those products that absolutely must be certified are. And why would any product need to be certified? Governments, which created the certification regimes, require certification for certain products that they purchase. So, as a practical matter, certification is generally only an issue if you are trying to sell your product to the government.[2]

8.2.1 The Orange Book

The Trusted Computing System Evaluation Criteria (TCSEC), or "orange book" [309] (so called because of the color of its cover) was published in 1983. The orange book was one of a series of related books developed under the auspices of the National Security Agency. Each book had a different colored cover and collectively they are known as the "rainbow series." The orange book primarily deals with system evaluation and certification and, to some extent, multilevel security—a topic discussed later in this chapter.

Today, the orange book is of little, if any, practical relevance. Moreover, in your opinionated author's opinion, the orange book served to stunt the growth of information security by focusing vast amounts of time and resources on some of the most esoteric and impractical aspects of security.[3]

Of course, not everyone is as enlightened as your humble author, and, in some circles, there is still something of a religious fervor for the orange book

[1] That is, before the Apple Macintosh was invented.

[2] It's tempting to argue that certification is an obvious failure simply because there is no evidence that the government is any more secure than anybody else, in spite of its use of certified security products. However, your certifiable author will, for once, refrain from making such a smug and unsubstantiated (but oddly satisfying) claim.

[3] Other than that, the orange book was a smashing success.

and its view of the security universe. In fact, the faithful tend to believe that if only the orange book way of thinking had prevailed, we'd all be much more secure today.

So, is it worth knowing something about the orange book? Well, it's always good to have some historical perspective on any subject. Also, as previously mentioned, there are some people who still take it seriously (although fewer and fewer each day), and you may need to deal with such a person at some point. In addition, it is possible that you may need to worry about system certification.

The stated purpose of the orange book is to provide criteria for assessing the effectiveness of the security provided by "automatic data processing system products." The overriding goals, as given in [309], are:

a. To provide users with a yardstick with which to assess the degree of trust that can be placed in computer systems for the secure processing of classified or other sensitive information.

b. To provide guidance to manufacturers as to what to build into their new, widely available trusted commercial products in order to satisfy trust requirements for sensitive applications.

c. To provide a basis for specifying security requirements in acquisition specifications.

In short, the orange book intended to provide a way to assess the security of existing products and to provide guidance on how to build more secure products. The practical effect was that the orange book provided the basis for a certification regime that could be used to provide a security rating to a security product. In typical governmental fashion, the certification was to be determined by navigating through a complex and ill-defined maze of rules and requirements.

The orange book proposes four broad divisions, labeled as D through A, with D being the lowest and A the highest. Most of the divisions are split into classes. For example, under the C division, we have classes C1 and C2. The four divisions and their corresponding classes are as follows.

D. Minimal protection — This division contains only one class which is reserved for those systems that can't meet the requirements for any higher class. That is, these are the losers that couldn't make it into any "real" class.

C. Discretionary protection — There are two classes here, both of which provide some level of "discretionary" protection. That is, they don't necessarily force security on users, but instead they provide some means of detecting security breaches—specifically, there must be an audit capability. The two classes in this division are the following.

C1. Discretionary security protection — In this class, a system must provide "credible controls capable of enforcing access limitations on an individual basis."[4]

C2. Controlled access protection —- Systems in this class "enforce a more finely grained discretionary access control than (C1) systems."[5]

B. Mandatory protection — This a big step up from C. The idea of the C division is that users can break the security, but they might get caught. However, for division B, the protection is mandatory, in the sense that users cannot break the security even if they try. The B classes are the following.

B1. Labeled security protection — Mandatory access control is based on specified labels. That is, all data carries some sort of label, which determines which users can do what with the data. Also, the access control is enforced in a way so that users cannot violate it (i.e., the access control is mandatory).

B2. Structured protection — This adds covert channel protection (discussed later in this chapter) and a few other technical issues on top of B1.

B3. Security domains — On top of B2 requirements, this class adds that the code that enforces security must "be tamperproof, and be small enough to be subjected to analysis and tests." We'll have much more to say about software issues in later chapters. For now, it is worth mentioning that making software tamperproof is, at best, difficult and expensive, and is seldom attempted in any serious way.

A. Verified protection — This is the same as B3, except that so-called formal methods must be used to, in effect, prove that the system does what is claimed. In this division there is a class A1 and a brief discussion of what might lie beyond A1.

The A division was certainly very optimistic for a document published in the 1980s, since the formal proofs that it envisions are not yet feasible for systems of moderate or greater complexity. As a practical matter, satisfying the C level requirements should be, in principle, almost trivial, but even today, achieving any of the B (or higher) classes would be a challenging task, except, perhaps, for certain straightforward applications (e.g., digital signature software).

[4] Hmm...

[5] Yes, of course, it's all so clear now...

There is a second part to the orange book that covers the "rationale and criteria," that is, it gives the reasoning behind the various requirements outlined above, and it attempts to provide specific guidance on how to meet the requirements. The rationale section includes a brief discussion of such topics as a reference monitor and a trusted computing base—topics that we will mention in our final chapter. There is also a brief discussion of the Bell-LaPadula security model, which we cover later in this chapter.

The criteria (i.e., the guidelines) section is certainly much more specific than the general discussion of the classes, but it is not clear that the guidelines are really all that useful or sensible. For example, under the title of "testing for division C" we have the following guidance, where "team" refers to the security testing team [309].

> The team shall independently design and implement at least five system-specific tests in an attempt to circumvent the security mechanisms of the system. The elapsed time devoted to testing shall be at least one month and need not exceed three months. There shall be no fewer than twenty hands-on hours spent carrying out system developer-defined tests and test team-defined tests.

While this is specific, it's not difficult to imagine a scenario where one team could accomplish more in a few hours of automated testing than another team could accomplish in three months of manual testing.[6]

8.2.2 The Common Criteria

Formally, the orange book has been superseded by the cleverly named Common Criteria [65], which is an international government-sponsored standard for certifying security products. The Common Criteria is similar to the orange book in the sense that, as much as is humanly possible, it is ignored in practice. However, if you want to sell your security product to the government, it may be necessary to obtain some specified level of Common Criteria certification. Even the lower-level Common Criteria certifications can be costly to obtain (on the order of six figures, in U.S. dollars), and the higher-level certifications are prohibitively expensive due to many fanciful requirements.

A Common Criteria certification yields a so-called Evaluation Assurance Level (EAL) with a numerical rating from 1 to 7, that is, EAL1 through EAL7, where the higher the number, the better. Note that a product with a higher EAL is not necessarily more secure than a product with a lower (or no) EAL. For example, suppose that product A is certified EAL4, while product B

[6] As an aside, your easily annoyed author finds it highly ironic and somewhat disturbing that the same people who gave us the dubious orange book now want to set educational standards in information security; see [216].

carries an EAL5 rating. All this means is that product A was evaluated for EAL4 (and passed), while product B was actually evaluated for EAL5 (and passed). It is possible that product A could actually have achieved EAL5 or higher, but the developers simply felt it was not worth the cost and effort to try for a higher EAL. The different EALs are listed below [106].

- EAL1 — Functionally Tested
- EAL2 — Structurally Tested
- EAL3 — Methodically Tested and Checked
- EAL4 — Methodically Designed, Tested, and Reviewed
- EAL5 — Semiformally Designed and Tested
- EAL6 — Semiformally Verified Design and Tested
- EAL7 — Formally Verified Design and Tested

To obtain an EAL7 rating, formal proofs of security must be provided, and security experts carefully analyze the product. In contrast, at the lowest EALs, the documentation is all that is analyzed. Of course, at an intermediate level, something between these two extremes is required.

Certainly the most commonly sought-after Common Criteria certification is EAL4, since it is generally the minimum required to sell to the government. Interestingly, your hard-working author could find a grand total of precisely two products certified at the highest Common Criteria level, EAL7. This is not an impressive number considering that this certification regime has been around for more than a decade and it is an international standard.

And who are these security "experts" who perform Common Criteria evaluations? The security experts work for government-accredited Common Criteria Testing Laboratories—in the U.S. the accrediting agency is NIST.

We won't go into the details of Common Criteria certification here.[7] In any case, the Common Criteria will never evoke the same sort of passions (pro or con) as the orange book. Whereas the orange book is, in a sense, a philosophical statement claiming to provide the answers about how to do security, the Common Criteria is little more than a mind-numbing bureaucratic hurdle that must be overcome if you want to sell your product to the government. It is also worth noting that whereas the orange book is only about 115 pages long, due to inflation, the Common Criteria documentation exceeds 1000 pages. Consequently, few mortals will ever read the Common

[7] During your tireless author's two years at a small startup company, he spent an inordinate amount of time studying the Common Criteria documentation—his company was hoping to sell its product to the U.S. government. Because of this experience, mere mention of the Common Criteria causes your usually hypoallergenic author to break out in hives.

Criteria, which is another reason why it will never evoke more than a yawn from the masses.

Next, we consider the classic view of authorization. Then we look at multilevel security (and related topics) before considering a few cutting-edge topics, including firewalls, IDS, and CAPTCHAs.

8.3 Access Control Matrix

The classic view of authorization begins with Lampson's access control matrix [5]. This matrix contains all of the relevant information needed by an operating system to make decisions about which users are allowed to do what with the various system resources.

We'll define a *subject* as a user of a system (not necessarily a human user) and an *object* as a system resource. Two fundamental constructs in the field of authorization are *access control lists*, or ACLs, and *capabilities*, or C-lists. Both ACLs and C-lists are derived from Lampson's *access control matrix*, which has a row for every subject and a column for every object. Sensibly enough, the access allowed by subject S to object O is stored at the intersection of the row indexed by S and the column indexed by O. An example of an access control matrix appears in Table 8.1, where we use UNIX-style notation, that is, `x`, `r`, and `w` stand for execute, read, and write privileges, respectively.

Table 8.1: Access Control Matrix

	OS	Accounting program	Accounting data	Insurance data	Payroll data
Bob	`rx`	`rx`	`r`	—	—
Alice	`rx`	`rx`	`r`	`rw`	`rw`
Sam	`rwx`	`rwx`	`r`	`rw`	`rw`
Accounting program	`rx`	`rx`	`rw`	`rw`	`r`

Notice that in Table 8.1, the accounting program is treated as both an object and a subject. This is a useful fiction, since we can enforce the restriction that the accounting data is only modified by the accounting program. As discussed in [14], the intent here is to make corruption of the accounting data more difficult, since any changes to the accounting data must be done by software that, presumably, includes standard accounting checks and balances. However, this does not prevent all possible attacks, since the system administrator, Sam, could replace the accounting program with a faulty (or fraudulent) version and thereby break the protection. But this trick does

allow Alice and Bob to access the accounting data without allowing them to corrupt it—either intentionally or unintentionally.

8.3.1 ACLs and Capabilities

Since all subjects and all objects appear in the access control matrix, it contains all of the relevant information on which authorization decisions can be based. However, there is a practical issue in managing a large access control matrix. A system could have hundreds of subjects (or more) and tens of thousands of objects (or more), in which case an access control matrix with millions of entries (or more) would need to be consulted before any operation by any subject on any object. Dealing with such a large matrix could impose a significant burden on the system.

To obtain acceptable performance for authorization operations, the access control matrix can be partitioned into more manageable pieces. There are two obvious ways to split the access control matrix. First, we could split the matrix into its columns and store each column with its corresponding object. Then, whenever an object is accessed, its column of the access control matrix would be consulted to see whether the operation is allowed. These columns are known as access control lists, or ACLs. For example, the ACL corresponding to insurance data in Table 8.1 is

$$(\text{Bob}, —), (\text{Alice}, \texttt{rw}), (\text{Sam}, \texttt{rw}), (\text{accounting program}, \texttt{rw}).$$

Alternatively, we could store the access control matrix by row, where each row is stored with its corresponding subject. Then, whenever a subject tries to perform an operation, we can consult its row of the access control matrix to see if the operation is allowed. This approach is know as capabilities, or C-lists. For example, Alice's C-list in Table 8.1 is

$$(\text{OS}, \texttt{rx}), (\text{accounting program}, \texttt{rx}), (\text{accounting data}, \texttt{r}), (\text{insurance data}, \texttt{rw}), (\text{payroll data}, \texttt{rw}).$$

It might seem that ACLs and C-lists are equivalent, since they simply provide different ways of storing the same information. However, there are some subtle differences between the two approaches. Consider the comparison of ACLs and capabilities illustrated in Figure 8.1.

Note that the arrows in Figure 8.1 point in opposite directions, that is, for ACLs, the arrows point from the resources to the users, while for capabilities, the arrows point from the users to the resources. This seemingly trivial difference has real significance. In particular, with capabilities, the association between users and files is built into the system, while for an ACL-based system, a separate method for associating users to files is required. This illustrates one of the inherent advantages of capabilities. In fact, capabilities have several security advantages over ACLs and, for this reason, C-lists are

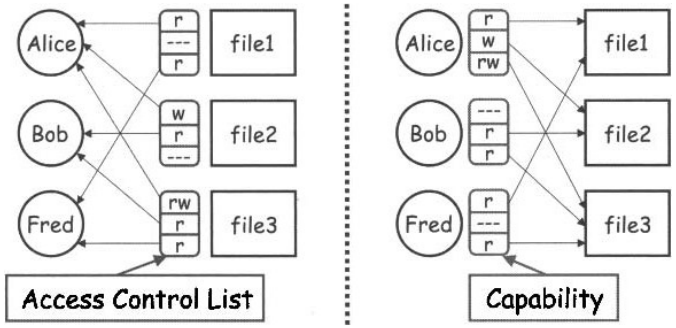


Figure 8.1: ACLs versus Capabilities

much beloved within the academic research community [206]. In the next section, we discuss one potential security advantage of capabilities over ACLs. Then we move on to the topic of multilevel security.

8.3.2 Confused Deputy

The *confused deputy* is a classic security problem that arises in many contexts [139]. For our illustration of this problem, we consider a system with two resources, a compiler and a file named BILL that contains critical billing information, and one user, Alice. The compiler can write to any file, while Alice can invoke the compiler and she can provide a filename where debugging information will be written. However, Alice is not allowed to write to the file BILL, since she might corrupt the billing information. The access control matrix for this scenario appears in Table 8.2.

Table 8.2: Access Control Matrix for Confused Deputy Example

	Compiler	BILL
Alice	x	—
Compiler	rx	rw

Now suppose that Alice invokes the compiler, and she provides BILL as the debug filename. Alice does not have the privilege to access the file BILL, so this command should fail. However, the compiler, which is acting on Alice's behalf, does have the privilege to overwrite BILL. If the compiler acts with its privilege, then a side effect of Alice's command will be the trashing of the BILL file, as illustrated in Figure 8.2.

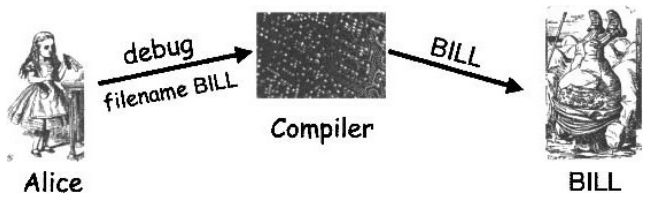


Figure 8.2: Confused Deputy

Why is this problem known as the confused deputy? The compiler is acting on Alice's behalf, so it is her deputy. The compiler is confused since it is acting based on its own privileges when it should be acting based on Alice's privileges.

With ACLs, it's more difficult (but not impossible) to avoid the confused deputy. In contrast, with capabilities it's relatively easy to prevent this problem, since capabilities are easily delegated, while ACLs are not. In a capabilities-based system, when Alice invokes the compiler, she can simply give her C-list to the compiler. The compiler then consults Alice's C-list when checking privileges before attempting to create the debug file. Since Alice does not have the privilege to overwrite BILL, the situation in Figure 8.2 can be avoided.

A comparison of the relative advantages of ACLs and capabilities is instructive. ACLs are preferable when users manage their own files and when protection is data oriented. With ACLs, it's also easy to change rights to a particular resource. On the other hand, with capabilities it's easy to delegate (and sub-delegate and sub-sub-delegate, and so on), and it's easier to add or delete users. Due to the ability to delegate, it's easy to avoid the confused deputy when using capabilities. However, capabilities are more complex to implement and they have somewhat higher overhead—although it may not be obvious, many of the difficult issues inherent in distributed systems arise in the context of capabilities. For these reasons, ACLs are used in practice far more often than capabilities.

8.4 Multilevel Security Models

In this section we briefly discuss security modeling in the context of multilevel security. Security models are often presented at great length in information security textbooks, but here we'll only mention two of the best-known models, and we only present an overview of these models. For a more thorough introduction to MLS and related security models, see [283] or Gollmann's book [125].

In general, security models are descriptive, not proscriptive. That is, these models tell us what needs to be protected, but they don't answer the real question, that is, how to provide such protection. This is not a flaw in the models, as they are designed to set a framework for protection, but it is an inherent limitation on the practical utility of security modeling.

Multilevel security, or MLS, is familiar to all fans of spy novels, where classified information often figures prominently. In MLS, the subjects are the users (generally, human) and the objects are the data to be protected (for example, documents). Furthermore, *classifications* apply to objects while *clearances* apply to subjects.

The U.S. Department of Defense, or DoD, employs four levels of classifications and clearances, which can be ordered as

$$\text{TOP SECRET} > \text{SECRET} > \text{CONFIDENTIAL} > \text{UNCLASSIFIED}. \tag{8.1}$$

For example, a subject with a SECRET clearance is allowed access to objects classified SECRET or lower but not to objects classified TOP SECRET. Apparently to make them more visible, security levels are generally rendered in upper case.

Let O be an object and S a subject. Then O has a classification and S has a clearance. The security *level* of O is denoted $L(O)$, and the security level of S is similarly denoted $L(S)$. In the DoD system, the four levels shown above in (8.1) are used for both clearances and classifications. Also, for a person to obtain a SECRET clearance, a more-or-less routine background check is required, while a TOP SECRET clearance requires an extensive background check, a polygraph exam, a psychological profile, etc.

There are many practical problems related to the classification of information. For example, the proper classification is not always clear, and two experienced users might have widely differing views. Also, the level of granularity at which to apply classifications can be an issue. It's entirely possible to construct a document where each paragraph, when taken individually, is UNCLASSIFIED, yet the overall document is TOP SECRET. This problem is even worse when source code must be classified, which is sometimes the case within the DoD. The flip side of granularity is aggregation—an adversary might be able to glean TOP SECRET information from a careful analysis of UNCLASSIFIED documents.

Multilevel security is needed when subjects and objects at different levels use the same system resources. The purpose of an MLS system is to enforce a form of access control by restricting subjects so that they only access objects for which they have the necessary clearance.

Military and government have long had an interest in MLS. The U.S. government, in particular, has funded a great deal of research into MLS and, as a consequence, the strengths and weaknesses of MLS are relatively well understood.

Today, there are many potential uses for MLS outside of its traditional classified government setting. For example, most businesses have information that is restricted to, say, senior management, and other information that is available to all management, while still other proprietary information is available to everyone within the company and, finally, some information is available to everyone, including the general public. If this information is stored on a single system, the company must deal with MLS issues, even if they don't realize it. Note that these categories correspond directly to the TOP SECRET, SECRET, CONFIDENTIAL, and UNCLASSIFIED classifications discussed above.

There is also interest in MLS in such applications as network firewalls. The goal in such a case is to keep an intruder, Trudy, at a low level to limit the damage that she can inflict after she breaches the firewall. Another MLS application that we'll examine in more detail below deals with private medical information.

Again, our emphasis here is on MLS models, which explain what needs to be done but do not tell us how to implement such protection. In other words, we should view these models as high-level descriptions, not as security algorithms or protocols. There are many MLS models—we'll only discuss the most elementary. Other models can be more realistic, but they are also more complex and harder to analyze and verify.

Ideally, we would like to prove results about security models. Then any system that satisfies the assumptions of the model automatically inherits all of the results that have been proved about the model. However, we will not delve so deeply into security models in this book.

8.4.1 Bell-LaPadula

The first security model that we'll consider is Bell-LaPadula, or BLP, which, believe it or not, was named after its inventors, Bell and LaPadula. The purpose of BLP is to capture the minimal requirements, with respect to confidentiality, that any MLS system must satisfy. BLP consists of the following two statements:

> **Simple Security Condition**: Subject S can read object O if and only if $L(O) \leq L(S)$.
>
> ***-Property** (Star Property): Subject S can write object O if and only if $L(S) \leq L(O)$.

The simple security condition merely states that Alice, for example, cannot read a document for which she lacks the appropriate clearance. This condition is clearly required of any MLS system.

The star property is somewhat less obvious. This property is designed to prevent, say, TOP SECRET information from being written to, say, a SECRET document. This would break MLS security since a user with a SECRET clearance could then read TOP SECRET information. The writing could occur intentionally or, for example, as the result of a computer virus. In his groundbreaking work on viruses, Cohen mentions that viruses could be used to break MLS security [60], and such attacks remain a very real threat to MLS systems today.

The simple security condition can be summarized as "no read up," while the star property implies "no write down." Consequently, BLP is sometimes succinctly stated as "no read up, no write down." It's difficult to imagine a security model that's any simpler.

Although simplicity in security is a good thing, BLP may be too simple. At least that is the conclusion of McLean, who states that BLP is "so trivial that it is hard to imagine a realistic security model for which it does not hold" [198]. In an attempt to poke holes in BLP, McLean defined a "system Z" in which an administrator is allowed to temporarily reclassify objects, at which point they can be "written down" without violating BLP. System Z clearly violates the spirit of BLP, but, since it is not expressly forbidden, it is apparently allowed.

In response to McLean's criticisms, Bell and LaPadula fortified BLP with a *tranquility property*. Actually, there are two versions of this property. The strong tranquility property states that security labels can never change. This removes McLean's system Z from the BLP realm, but it's also impractical in the real world, since security labels must sometimes change. For example, the DoD regularly declassifies documents, which would be impossible under strict adherence to the strong tranquility property. For another example, it is often desirable to enforce *least privilege*. If a user has, say, a TOP SECRET clearance but is only browsing UNCLASSIFIED Web pages, it is desirable to only give the user an UNCLASSIFIED clearance, so as to avoid accidentally divulging classified information. If the user later needs a higher clearance, his active clearance can be upgraded. This is known as the *high water mark principle*, and we'll see it again when we discuss Biba's model, below.

Bell and Lapadula also offered a *weak tranquility property* in which a security label can change, provided such a change does not violate an "established security policy." Weak tranquility can defeat system Z, and it can allow for least privilege, but the property is so vague as to be nearly meaningless for analytic purposes.

The debate concerning BLP and system Z is discussed thoroughly in [34], where the author points out that BLP proponents and McLean are each making fundamentally different assumptions about modeling. This debate gives rise to some interesting issues concerning the nature—and limits—of modeling.

The bottom line regarding BLP is that it's very simple, and as a result it's one of the few models for which it's possible to prove things about systems. Unfortunately, BLP may be too simple to be of any practical benefit.

BLP has inspired many other security models, most of which strive to be more realistic. The price that these systems pay for more reality is more complexity. This makes most other models more difficult to analyze and more difficult to apply, that is, it's more difficult to show that a real-world system satisfies the requirements of the model.

8.4.2 Biba's Model

In this section, we'll look briefly at Biba's model. Whereas BLP deals with confidentiality, Biba's model deals with integrity. In fact, Biba's model is essentially an integrity version of BLP.

If we trust the integrity of object O_1 but not that of object O_2, then if object O is composed of O_1 and O_2, we cannot trust the integrity of object O. In other words, the integrity level of O is the minimum of the integrity of any object contained in O. Another way to say this is that for integrity, a low water mark principle holds. In contrast, for confidentiality, a high water mark principle applies.

To state Biba's model formally, let $I(O)$ denote the integrity of object O and $I(S)$ the integrity of subject S. Biba's model is defined by the two statements:

> **Write Access Rule**: Subject S can write object O if and only if $I(O) \leq I(S)$.
>
> **Biba's Model**: A subject S can read the object O if and only if $I(S) \leq I(O)$.

The write access rule states that we don't trust anything that S writes any more than we trust S. Biba's model states that we can't trust S any more than the lowest integrity object that S has read. In essence, we are concerned that S will be "contaminated" by lower integrity objects, so S is forbidden from viewing such objects.

Biba's model is actually very restrictive, since it prevents S from ever viewing an object at a lower integrity level. It's possible—and, in many cases, perhaps desirable—to replace Biba's model with the following:

> **Low Water Mark Policy**: If subject S reads object O, then $I(S) = \min\big(I(S), I(O)\big)$.

Under the low water mark principle, subject S can read anything, under the condition that the integrity of subject S is downgraded after accessing an object at a lower level.

Figure 8.3 illustrates the difference between BLP and Biba's model. Of course the fundamental difference is that BLP is for confidentiality, which implies a high water mark principle, while Biba is for integrity, which implies a low water mark principle.

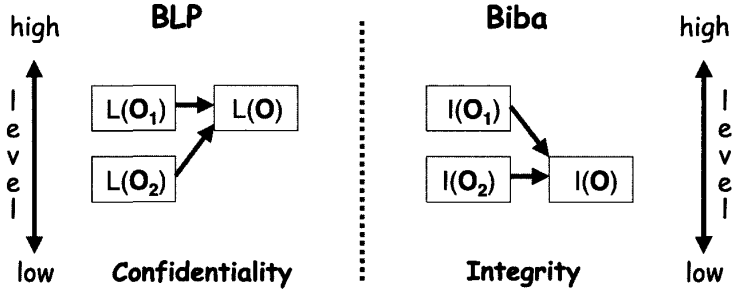


Figure 8.3: BLP versus Biba

8.5 Compartments

Multilevel security systems enforce access control (or information flow) "up and down," where the security levels are ordered in a hierarchy, such as (8.1). Usually, a simple hierarchy of security labels is not flexible enough to deal with a realistic situation. In practice, it is usually necessary to also use *compartments* to further restrict information flow "across" security levels.

We use the notation

SECURITY LEVEL {COMPARTMENT}

to denote a security level and its associated compartment or compartments. For example, suppose that we have compartments CAT and DOG within the TOP SECRET level. Then we would denote the resulting compartments as TOP SECRET {CAT} and TOP SECRET {DOG}. Note that there is also a TOP SECRET {CAT,DOG} compartment. While each of these compartments is TOP SECRET, a subject with a TOP SECRET clearance can only access a compartment if he or she is specifically allowed to do so. As a result, compartments have the effect of restricting information flow across security levels.

Compartments serve to enforce the *need to know* principle, that is, subjects are only allowed access to the information that they must know for their work. If a subject does not have a legitimate need to know everything at, say,

the TOP SECRET level, then compartments can be used to limit the TOP SECRET information that the subject can access.

Why create compartments instead of simply creating a new classification level? It may be the case that, for example, TOP SECRET {CAT} and TOP SECRET {DOG} are not comparable, that is, neither

$$\text{TOP SECRET } \{\text{CAT}\} \leq \text{TOP SECRET } \{\text{DOG}\}$$

nor

$$\text{TOP SECRET } \{\text{CAT}\} \geq \text{TOP SECRET } \{\text{DOG}\}$$

holds. Using a strict MLS hierarchy, one of these two conditions must hold true.

Consider the compartments in Figure 8.4, where the arrows represent "$\geq$" relationships. In this example, a subject with a TOP SECRET {CAT} clearance does not have access to information in the TOP SECRET {DOG} compartment. In addition, a subject with a TOP SECRET {CAT} clearance has access to the SECRET {CAT} compartment but not to the compartment SECRET {CAT,DOG}, even though the subject has a TOP SECRET clearance. Again, compartments provide a means to enforce the need to know principle.

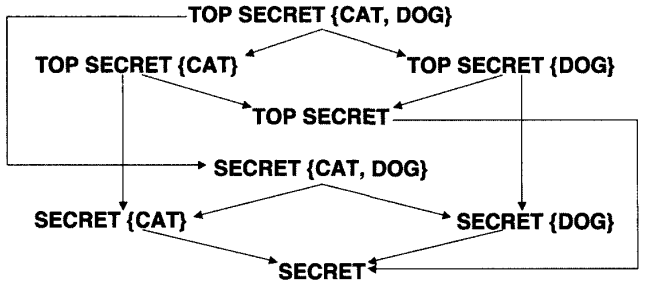


Figure 8.4: Compartments Example

Multilevel security can be used without compartments and vice versa, but the two are usually used together. An interesting example described in [14] concerns the protection of personal medical records by the British Medical Association, or BMA. The law that required protection of medical records mandated a multilevel security system—apparently because lawmakers were familiar with MLS. Certain medical conditions, such as AIDS, were considered to be the equivalent of TOP SECRET, while other less sensitive information, such as drug prescriptions, was considered SECRET. But if a subject had been prescribed AIDS drugs, anyone with a SECRET clearance could easily deduce TOP SECRET information. As a result, all information tended

to be classified at the highest level, and consequently all users required the highest level of clearance, which defeated the purpose of the system. Eventually, the BMA system was changed to a system using only compartments, which effectively solved the problem. Then, for example, AIDS prescription information could be compartmented from general prescription information, thereby enforcing the desired need to know principle.

In the next two sections we'll discuss covert channels and inference control. Both of these topics are related to MLS, but covert channels, in particular, arise in many different contexts.

8.6 Covert Channel

A *covert channel* is a communication path not intended as such by the system's designers. Covert channels exist in many situations, but they are particularly prevalent in networks. Covert channels are virtually impossible to eliminate, so the emphasis is instead on limiting the capacity of such channels.

MLS systems are designed to restrict legitimate channels of communication. But a covert channel provides another way for information to flow. It is not difficult to give an example where resources shared by subjects at different security levels can be used to pass information, and thereby violate the security of an MLS system.

For example, suppose Alice has a TOP SECRET clearance while Bob only has a CONFIDENTIAL clearance. If the file space is shared by all users, then Alice and Bob can agree that if Alice wants to send a 1 to Bob, she will create a file named, say, `FileXYzW`, and if she wants to send a 0 she will not create such a file. Bob can check to see whether file `FileXYzW` exists, and if it does, he knows Alice has sent him a 1, while if it does not, Alice has sent him a 0. In this way, a single bit of information has been passed through a covert channel, that is, through a means that was not intended for communication by the designers of the system. Note that Bob cannot look inside the file `FileXYzW` since he does not have the required clearance, but we are assuming that he can query the file system to see if such a file exists.

A single bit leaking from Alice to Bob is not a concern, but Alice could leak any amount of information by synchronizing with Bob. That is, Alice and Bob could agree that Bob will check for the file `FileXYzW` once each minute. As before, if the file does not exist, Alice has sent 0, and if it does exists, Alice has sent a 1. In this way Alice can (slowly) leak TOP SECRET information to Bob. This process is illustrated in Figure 8.5.

Covert channels are everywhere. For example, the print queue could be used to signal information in much the same way as in the example above. Networks are a rich source of covert channels, and several hacking tools exist that exploit these covert channels—we'll mention one later in this section.

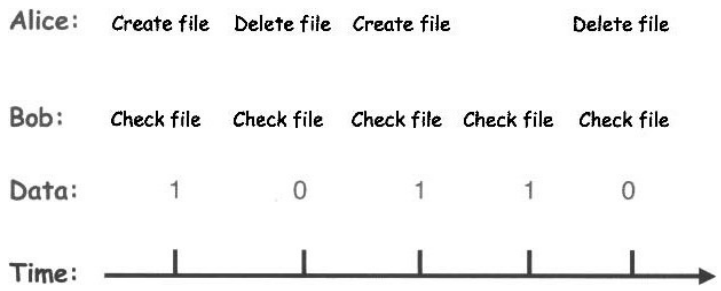


Figure 8.5: Covert Channel Example

Three things are required for a covert channel to exist. First, the sender and receiver must have access to a shared resource. Second, the sender must be able to vary some property of the shared resource that the receiver can observe. Finally, the sender and receiver must be able to synchronize their communication. From this description, it's apparent that potential covert channels really are everywhere. Of course, we can eliminate all covert channels—we just need to eliminate all shared resources and all communication. Obviously such a system would generally be of little use.

The conclusion here is that it's virtually impossible to eliminate all covert channels in any useful system. The DoD apparently agrees, since their guidelines merely call for reducing covert channel capacity to no more than one bit per second [131]. The implication is that DoD has given up trying to eliminate covert channels.

Is a limit of one bit per second sufficient to prevent damage from covert channels? Consider a TOP SECRET file that is 100 MB in size. Suppose the plaintext version of this file is stored in a TOP SECRET file system, while an encrypted version of the file—encrypted with, say, AES using a 256-bit key—is stored in an UNCLASSIFIED location. Following the DoD guidelines, suppose that we have reduced the covert channel capacity of this system to 1 bit per second. Then it would take more than 25 years to leak the entire 100 MB TOP SECRET document through a covert channel. However, it would take less than 5 minutes to leak the 256-bit AES key through the same covert channel. The conclusion is that reducing covert channel capacity might be useful, but it will not be sufficient in all cases.

Next, we consider a real-world example of a covert channel. The Transmission Control Protocol (TCP) is widely used on the Internet. The TCP header, which appears in the Appendix in Figure A-3, includes a "reserved" field which is reserved for future use, that is, it is not used for anything. This field can easily be used to pass information covertly.

It's also easy to hide information in the TCP sequence number or ACK field and thereby create a more subtle covert channel. Figure 8.6 illustrates the method used by the tool `Covert_TCP` to pass information in the sequence number. The sender hides the information in the sequence number X and the packet—with its source address forged to be the address of the intended recipient—is sent to an innocent server. When the server acknowledges the packet, it unwittingly completes the covert channel by passing the information contained in X to the intended recipient. Such stealthy covert channels are often employed in network attacks [270].

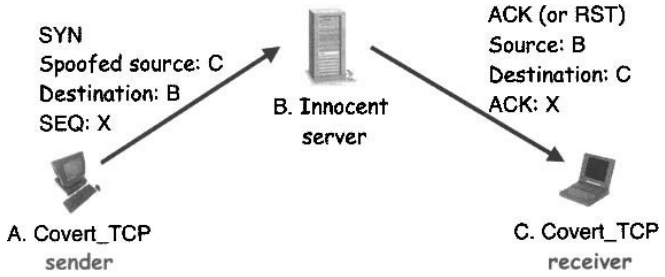


Figure 8.6: Covert Channel Using TCP Sequence Number

8.7 Inference Control

Consider a database that includes information on college faculty in California. Suppose we query the database and ask for the average salary of female computer science professors at San Jose State University (SJSU) and we find the answer is $100,000. We then query the database and ask for the number of female computer science professors at SJSU, and the answer is one. Then we could go to the SJSU computer science department website and determine the identity of this person.[8] In this example, specific information has leaked from responses to general questions. The goal of inference control is to prevent such leaks from happening, or at least minimize the leakage.

A database containing medical records would be of considerable interest to researchers. For example, by searching for statistical correlations, it may be possible to determine causes or risk factors for certain diseases. But patients want to keep their medical information private. How can we allow access to the statistically significant data while protecting privacy?

[8]In this case, no harm was done, since state employee salaries are public information in California.

An obvious first step is to remove names and addresses from the medical records. But this is not sufficient to ensure privacy as the college professor example above clearly demonstrates. What more can be done to provide stronger inference control while leaving the data accessible for legitimate research uses?

Several techniques used in inference control are discussed in [14]. One such technique is *query set size control*, in which no response is returned if the size of the set it too small. This approach would make it more difficult to determine the college professor's salary in the example above. However, if medical research is focused on a rare disease, query set size control could also prevent or distort important research.

Another technique is known as the *N-respondent, k% dominance rule*, whereby data is not released if k% or more of the result is contributed by N or fewer subjects. For example, we might query the census database and ask for the average net worth of individuals in Bill Gates' neighborhood. With any reasonable setting for N and k no results would be returned. In fact, this technique is actually applied to information collected by the United States Census Bureau.

Another approach to inference control is randomization, that is, a small amount of random noise is added to the data. This is problematic in situations such as research into rare medical conditions, where the noise might swamp legitimate data.

Many other methods of inference control have been proposed, but none are completely satisfactory. It appears that strong inference control may be impossible to achieve in practice, yet it seems obvious that employing some inference control, even if it's weak, is better than no inference control at all. Inference control will make Trudy's job more difficult, and it will almost certainly reduce the amount of information that leaks, thereby limiting the damage.

Does this same logic hold for crypto? That is, is it better to use weak encryption or no encryption at all? Surprisingly, for crypto, the answer is that, in most cases, you'd be better off not encrypting rather than using a weak cipher. Today, most information is not encrypted, and encryption tends to indicate important data. If there is a lot of data being sent and most of it is plaintext (e.g., email sent over the Internet), then Trudy faces an enormous challenge in attempting to filter interesting messages from this mass of uninteresting data. However, if your data is encrypted, it would be much easier to filter, since encrypted data looks random, whereas unencrypted data tends to be highly structured.[9] That is, if your encryption is weak, you may have just solved Trudy's difficult filtering problem for her, while providing no significant protection from a cryptanalytic attack [14].

[9]For one way around this problem, see [287].

8.8 CAPTCHA

The Turing test was proposed by computing pioneer (and breaker of the Enigma) Alan Turing in 1950. The test has a human ask questions to a human and a computer. The questioner, who can't see either the human or the computer, can only submit questions by typing on a keyboard, and responses are received on a computer screen. The questioner does not know which is the computer and which is the human, and the goal is to distinguish the human from the computer, based solely on the questions and answers. If the human questioner can't solve this puzzle with a probability better than guessing, the computer passes the Turing test. This test is the gold standard in artificial intelligence, and no computer has yet passed the Turing test, but occasionally some claim to be getting close.

A "completely automated public Turing test to tell computers and humans apart," or *CAPTCHA*,[10] is a test that a human can pass, but a computer can't pass with a probability better than guessing [319]. This could be considered as a sort of inverse Turing test. The assumptions here are that the test is generated by a computer program and graded by a computer program, yet no computer can pass the test, even if that computer has access to the source code used to generate the test. In other words, a "CAPTCHA is a program that can generate and grade tests that it itself cannot pass, much like some professors" [319].

At first blush, it seems paradoxical that a computer can create and score a test that it cannot pass. However, this becomes less of a paradox when we look more closely the details of the process.

Since CAPTCHAs are designed to prevent non-humans from accessing resources, a CAPTCHA can be viewed as a form of access control. According to folklore, the original motivation for CAPTCHAs was an online poll that asked users to vote for the best computer science graduate school. In this version of reality, it quickly become obvious that automated responses from MIT and Carnegie-Mellon were skewing the results [320] and researchers developed the idea of a CAPTCHA to prevent automated "bots" from stuffing the ballot box. Today, CAPTCHAs are used in a wide variety of applications. For example, free email services use CAPTCHAs to prevent spammers from automatically signing up for large numbers of email accounts.

The requirements for a CAPTCHA include that it must be easy for most humans to pass and it must be difficult or impossible for a machines to pass, even if the machine has access to the CAPTCHA software. From the attacker's perspective, the only unknown is some randomness that is used to generate the specific CAPTCHA. It is also desirable to have different types

[10]CAPTCHAs are also known as "human interactive proofs," or HIPs. While CAPTCHA may well rank as the worst acronym in the history of the universe, HIP is, well, just not hip.

of CAPTCHAs in case some person cannot pass one particular type. For example, many websites allow users to choose an audio CAPTCHA as an alternative to the usual visual CAPTCHA.

An example of a CAPTCHA from [320] appears in Figure 8.7. In this case, a human might be asked to find three words that appear in the image. This is a relatively easy problem for humans and today it is also a fairly easy problem for computers to solve—much stronger CAPTCHAs exist.

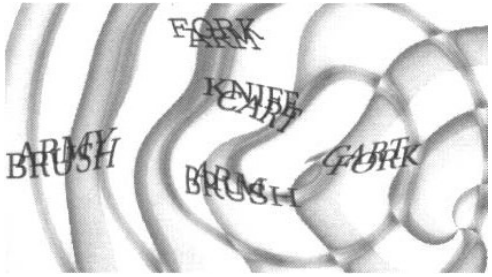


Figure 8.7: CAPTCHA (Courtesy of Luis von Ahn [320])

Perhaps surprisingly, in [56] it is shown that computers are actually better than humans at solving all of the fundamental visual CAPTCHA problems, with one exception—the so-called segmentation problem, i.e., the problem of separating the letters from each other. Consequently, strong CAPTCHAs tend to look more like Figure 8.8 than Figure 8.7.

Figure 8.8: A Strong CAPTCHA [47]

For a word-based visual CAPTCHA, we assume that Trudy knows the set of possible words that could appear and she knows the general format of the image, as well as the types of distortions that can be applied. From Trudy's perspective, the only unknown is a random number that is used to select the word or words and to distort the resulting image.

There are several types of visual CAPTCHAs of which Figures 8.7 and 8.8 are representative examples. There are also audio CAPTCHAs in which the audio is distorted in some way. The human ear is very good at removing such distortion, while automated methods are not so good. Currently, there are no text-based CAPTCHAs.

The computing problems that must be solved to break CAPTCHAs can be viewed as difficult problems from the domain of artificial intelligence, or AI.

For example, automatic recognition of distorted text is an AI problem, and the same is true of problems related to distorted audio. If attackers are able to break such CAPTCHAs, they have, in effect, solved a hard AI problem. As a result, attacker's efforts are being put to good use.

Of course, the attackers may not play by the rules—so-called CAPTCHA farming is possible, where humans are paid to solve CAPTCHAs. For example, it has been widely reported that the lure of free pornography has been successfully used to get humans to solve vast numbers of CAPTCHAs at minimal cost to the attacker [172].

8.9 Firewalls

Suppose you want to meet with the chairperson of your local computer science department. First, you will probably need to contact the computer science department secretary. If the secretary deems that a meeting is warranted, she will schedule it; otherwise, she will not. In this way, the secretary filters out many requests that would otherwise occupy the chair's time.

A *firewall* acts a lot like a secretary for your network. The firewall examines requests for access to your network, and it decides whether they pass a reasonableness test. If so, they are allowed through, and, if not, they are refused.

If you want to meet the chair of the computer science department, the secretary does a certain level of filtering; however, if you want to meet the President of the United States,[11] his secretary will perform a much different level of filtering. This is somewhat analogous to firewalls, where some simple firewalls only filter out obviously bogus requests and other types of firewalls make a much greater effort to filter anything suspicious.

A network firewall, as illustrated in Figure 8.9, is placed between the internal network, which might be considered relatively safe,[12] and the external network (the Internet), which is known to be unsafe. The job of the firewall is to determine what to let into and out of the internal network. In this way, a firewall provides access control for the network.

As with most of information security, for firewalls there is no standard terminology. But whatever you choose to call them, there are essentially three types of firewalls—marketing hype from firewall vendors not withstanding. Each type of firewall filters packets by examining the data up to a particular layer of the network protocol stack. If you are not familiar with networking (and even if you are), now would be a good time to review the networking material in the Appendix.

[11] POTUS, that is.

[12] This is almost certainly not a valid assumption. It's estimated that about 80% of all significant computer attacks are due to insiders [49].

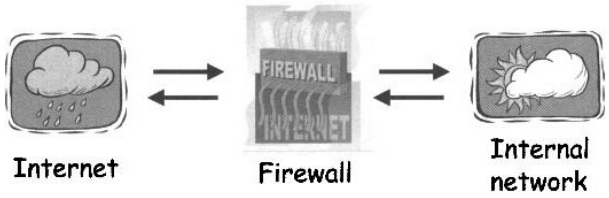


Figure 8.9: Firewall

We'll adopt the following terminology for the classification of firewalls.

- A *packet filter* is a firewall that operates at the network layer.
- A *stateful packet filter* is a firewall that lives at the transport layer.
- An *application proxy* is, as the name suggests, a firewall that operates at the application layer where it functions as a proxy.

8.9.1 Packet Filter

A packet filter firewall examines packets up to the network layer, as indicated in Figure 8.10. As a result, this type of firewall can only filter packets based on the information that is available at the network layer. The information at this layer includes the source IP address, the destination IP address, the source port, the destination port, and the TCP flag bits (SYN, ACK, RST, etc.).[13] Such a firewall can filter packets based on ingress or egress, that is, it can have different filtering rules for incoming and outgoing packets.

The primary advantage of a packet filter is efficiency. Since packets only need to be processed up to the network layer and only header information is examined, the entire operation is inherently efficient. However, there are several disadvantages to the simple approach employed by a packet filter. First, the firewall has no concept of state, so each packet is treated independently of all others. In particular, a packet filter can't examine a TCP connection. We'll see in a moment that this is a serious limitation. In addition, a packet filter firewall is blind to application data, which is where viruses and other malware resides.

Packet filters are configured using access control lists, or ACLs. In this context, "ACL" has a completely different meaning than in Section 8.3.1. An example of a packet filter ACL appears in Table 8.3. Note that the purpose of the ACL in Table 8.3 is to restrict incoming packets to Web responses,

[13] Yes, we're cheating. TCP is part of the transport layer, so the TCP flag bits are not visible if we follow a strict definition of network layer. Nevertheless, it's OK to cheat sometimes, especially in a security class.

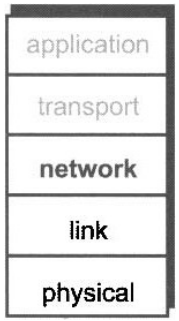


Figure 8.10: Packet Filter

Table 8.3: Example ACL

Action	Source IP	Dest IP	Source Port	Dest Port	Protocol	Flag Bits
Allow	Inside	Outside	Any	80	HTTP	Any
Allow	Outside	Inside	80	> 1023	HTTP	ACK
Deny	All	All	All	All	All	All

which should have source port 80. The ACL allows all outbound Web traffic, which should be destined to port 80. All other traffic is forbidden.

How might Trudy take advantage of the inherent limitations of a packet filter firewall? Before we can answer this question, we need a couple of fun facts. Usually, a firewall (of any type) drops packets sent to most incoming ports. That is, the firewall filters out and drops packets that are trying to access services that should not be accessed. Because of this, the attacker, Trudy, wants to know which ports are open through the firewall. These open ports are where Trudy will concentrate her attack. So, the first step in any attack on a firewall is usually a *port scan*, where Trudy tries to determine which ports are open through the firewall.

Now suppose Trudy wants to attack a network that is protected by a packet filter. How can Trudy conduct a port scan of the firewall? She could, for example, send a packet that has the ACK bit set, without the prior two steps of the TCP three-way handshake. Such a packet violates the TCP protocol, since the initial packet in any connection must have the SYN bit set. Since the packet filter has no concept of state, it will assume that this packet is part of an established connection and let it through—provided that

it is sent to an open port. Then when this forged packet reaches a host on the internal network, the host will realize that there is a problem (since the packet is not part of an established connection) and respond with a RST packet, which is supposed to tell the sender to terminate the connection. While this process may seem harmless, it allows Trudy to scan for open ports through the firewall. That is, Trudy can send an initial packet with the ACK flag set to a particular port p. If no response is received, then the firewall is not forwarding packets sent to port p. However, if a RST packet is received, then the packet was allowed through port p into the internal network. This technique, which is known as a TCP ACK scan, is illustrated in Figure 8.11.

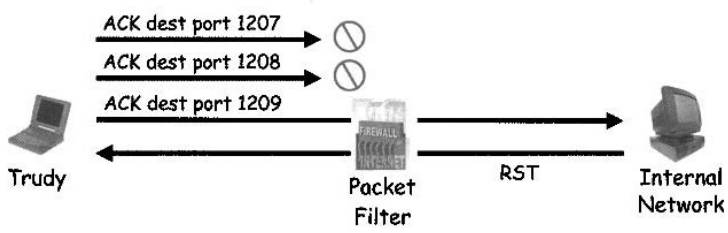


Figure 8.11: TCP ACK Scan

From the ACK scan in Figure 8.11, Trudy has learned that port 1209 is open through the firewall. To prevent this attack, the firewall would need to remember existing TCP connections, so that it will know that the ACK scan packets are not part of any legitimate connection. Next, we'll discuss stateful packet filters, which keep track of connections and are therefore able to prevent this ACK scan attack.

8.9.2 Stateful Packet Filter

As the name implies, a stateful packet filter adds state to a packet filter firewall. This means that the firewall keeps track of TCP connections, and it can remember UDP "connections" as well. Conceptually, a stateful packet filter operates at the transport layer, since it is maintaining information about connections. This is illustrated in Figure 8.12.

The primary advantage of a stateful packet filter is that, in addition to all of the features of a packet filter, it also keeps track of ongoing connection. This prevents many attacks, such as the TCP ACK scan discussed in the previous section. The disadvantages of a stateful packet filter are that it cannot examine application data, and, all else being equal, it's slower than a packet filtering firewall since more processing is required.

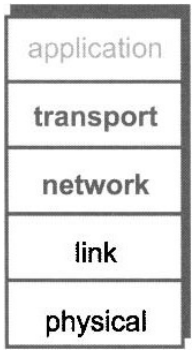


Figure 8.12: Stateful Packet Filter

8.9.3 Application Proxy

A proxy is something that acts on your behalf. An application proxy firewall processes incoming packets all the way up to the application layer, as indicated in Figure 8.13. The firewall, acting on your behalf, is then able to verify that the packet appears to be legitimate (as with a stateful packet filter) and, in addition, that the actual data inside the packet is safe.

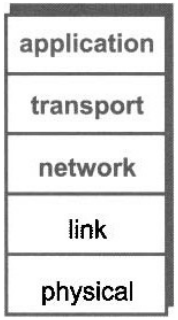


Figure 8.13: Application Proxy

The primary advantage of an application proxy is that it has a complete view of connections and application data. Consequently, it can have as comprehensive of a view as the host itself could have. As a result, the application proxy is able to filter bad data at the application layer (such as viruses) while also filtering bad packets at the transport layer. The disadvantage of an application proxy is speed or, more precisely, the potential lack thereof. Since the firewall is processing packets to the application layer, examining the resulting data, maintaining state, etc., it is doing a great deal more work than packet filtering firewalls.

One interesting feature of an application proxy is that the incoming packet is destroyed and a new packet is created in its place when the data passes through the firewall. Although this might seem like a minor and insignificant point, it's actually a security feature. To see why creating a new packet is beneficial, we'll consider the tool known as `Firewalk`, which is designed to scan for open ports through a firewall. While the purpose of `Firewalk` is the same as the TCP ACK scan discussed above, the implementation is completely different.

The time to live, or TTL, field in an IP packet header contains the number of hops that the packet will travel before it is terminated. When a packet is terminated due to the TTL field, an ICMP "time exceeded" error message is sent back to the source.[14]

Suppose Trudy knows the IP address of the firewall, the IP address of one system on the inside network, and the number of hops to the firewall. Then she can send a packet to the IP address of the known host inside the firewall, with the TTL field set to one more than the number of hops to the firewall. Suppose Trudy sets the destination port of such a packet to p. If the firewall does not let data through on port p, there will be no response. If, on the other hand, the firewall does let data through on port p, Trudy will receive a time exceeded error message from the first router inside the firewall that receives the packet. Trudy can then repeat this process for different ports p to determine open ports through the firewall. This port scan is illustrated in Figure 8.14. `Firewalk` will succeed if the firewall is a packet filter or a stateful packet filter. However, `Firewalk` won't succeed if the firewall is an application proxy (see Problem 29).

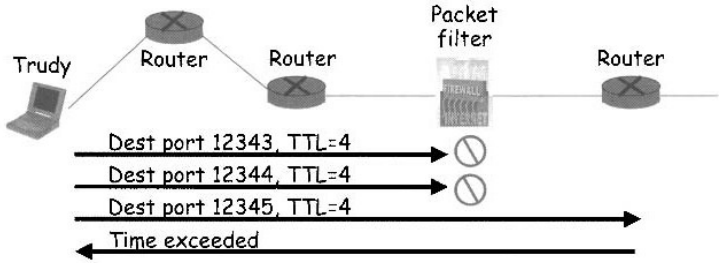


Figure 8.14: Firewalk

[14] And what happens to terminated packets? Of course, they die and go to packet heaven.

The net effect of an application proxy is that it forces Trudy to talk to the proxy and convince it to forward her messages. Since the proxy is likely to be well configured and carefully managed—compared with a typical host—this may prove difficult.

8.9.4 Personal Firewall

A personal firewall is used to protect a single host or a small network, such as a home network. Any of the three methods discussed above (packet filter, stateful packet filter, or application proxy) could be used, but generally such firewalls are relatively simple for the sake of efficiency and ease of configuration.

8.9.5 Defense in Depth

Finally, we consider a network configuration that includes several layers of protection. Figure 8.15 gives a schematic for a network that includes a packet filter firewall, an application proxy, and personal firewalls, as well as a demilitarized zone, or DMZ.

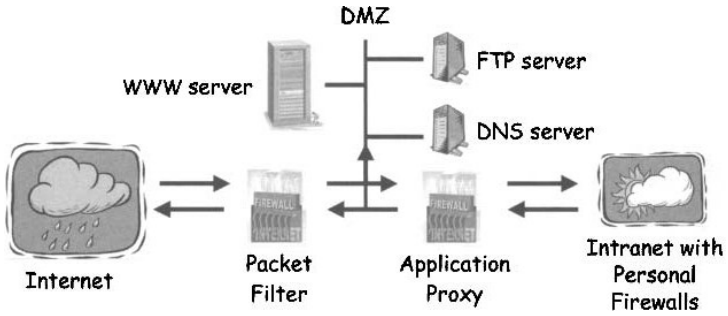


Figure 8.15: Defense in Depth

The packet filter in Figure 8.15 is used to prevent common attacks on the systems in the DMZ. The systems in the DMZ are those that must be exposed to the outside world. These systems receive most of the outside traffic, so a simple packet filter is used for the sake of efficiency. The systems in the DMZ must be carefully maintained by the administrator since they are the most exposed to attack. However, if an attack succeeds on a system in the DMZ, the consequences for the company are annoying, but they will probably not be life threatening, since the internal network is largely unaffected.

In Figure 8.15, an application proxy firewall sits between the internal network and the DMZ. This provides the strongest possible firewall protection

for the internal network. The amount of traffic into the internal network is likely to be relatively small, so an application proxy in this position will not create a bottleneck. As a final layer of protection, personal firewalls could be deployed on the individual hosts inside the corporate network.

The architecture in Figure 8.15 is an example of *defense in depth*, which is a good security strategy in general—if one layer of the defense is breached, there are more layers that the attacker must overcome. If Trudy is skilled enough to break through one level, then she may have the necessary skills to penetrate other levels. But it's likely to take her some time to do so and the longer it takes, the more time an administrator has to detect Trudy's attack in progress.

Regardless of the strength of the firewall (or firewalls), some attacks by outsiders will succeed. In addition, attacks by insiders are a serious threat and firewalls are of limited value against such attacks. In any case, when an attack succeeds, we would like to detect it as soon as possible. In the next section we'll discuss this intrusion detection problem.

8.10 Intrusion Detection Systems

The primary focus of computer security tends to be *intrusion prevention*, where the goal is to keep the Trudys of the world out of your system or network. Authentication can be viewed as a means to prevent intrusions, and firewalls are certainly a form of intrusion prevention, as are most types of virus protection. Intrusion prevention is the information security analog of locking the doors on your car.

But even if you lock the doors on your car, it might still get stolen. In information security, no matter how much effort you put into intrusion prevention, occasionally the bad guys will be successful and an intrusion will occur.

What should we do when intrusion prevention fails? *Intrusion detection systems*, or IDSs, are a relatively recent development in information security. The purpose of such a system is to detect attacks before, during, and after they occurr.

The basic approach employed by IDSs is to look for "unusual" activity. In the past, an administrator would scan through log files looking for signs of unusual activity—automated intrusion detection is a natural outgrowth of manual log file analysis.

It is also worth noting that intrusion detection is currently an active research topic. As with any relatively new technology, there are many claims in the field that have yet to be substantiated. At this point, it's far from clear how successful or useful some of these techniques will prove, particularly in the face of increasingly sophisticated attacks.

Before discussing the main threads in IDS, we mention in passing that *intrusion response* is a related topic of practical importance. That is, once an intrusion is detected, we want to respond to it. In some cases we obtain specific information and a reasonable response is fairly obvious. For example, we might detect a password guessing attack aimed at a specific account, in which case we could respond by locking the account. However, it's not always so straightforward. We'll see below that in some cases IDSs provide little specific information on the nature of an attack. In such cases, determining the proper response is not easy, since we may not be sure of the specifics of the attack. In any case, we won't deal further with intrusion response here.

Who are the intruders that an IDS is trying to detect? An intruder could be a hacker who got through your network defenses and is now launching an attack on the internal network. Or, even more insidious, the intrusion could be due to an evil insider, such as a disgruntled employee.

What sorts of attacks might an intruder launch? An intruder with limited skills (i.e., a "script kiddie") would likely attempt a well-known attack or a slight variation on such an attack. A more skilled attacker might be capable of launching a significant variation on a well-known attack, or a little-known attack or an entirely new attack. Often, the attacker will simply use the breached system as a base from which to launch attacks on other systems.

Broadly speaking, there are two approaches to intrusion detection.

- *Signature-based IDSs* detect attacks based on specific known signatures or patterns. This is analogous to signature-based virus detection, which we'll discuss in Chapter 11.

- *Anomaly-based IDSs* attempt to define a baseline of normal behavior and provide a warning whenever the system strays too far from this baseline.

We'll have more to say about signature-based and anomaly-based intrusion detection below.

There are also two basic architectures for IDSs.

- *Host-based IDSs* apply their detection method or methods to activity that occurs on hosts. These systems have the potential to detect attacks that are visible at hosts (e.g., buffer overflows or escalation of privilege). However, host-based systems have little or no view of network activities.

- *Network-based IDSs* apply their detection methods to network traffic. These systems are designed to detect attacks such as denial of service, port scans, probes involving malformed packets, etc. Such systems have some obvious overlap with firewalls. Network-based systems have little or no direct view of host-based attacks.

Of course, various combinations of these categories of IDSs are possible. For example a host-based system could use both signature-based and anomaly-based techniques, or a signature-based system might employ aspects of both host-based and network-based detection.

8.10.1 Signature-Based IDS

Failed login attempts may be indicative of a password cracking attack, so an IDS might consider "N failed login attempts in M seconds" an indication, or *signature*, of an attack. Then anytime that N or more failed login attempts occur within M seconds, the IDS would issue a warning that a password cracking attack is suspected to be in progress.

If Trudy happens to know that Alice's IDS issues a warning whenever N or more failed logins occur within M seconds, then Trudy can safely guess $N-1$ passwords every M seconds. In this case, the signature detection would slow Trudy's password guessing attack, but it would not completely prevent the attack. Another concern with such a scheme is that N and M must be set so that the number of false alarms is not excessive.

Many techniques are used to make signature-based detection more robust, where the usual approach is to detect "almost" signatures. For example, if about N login attempts occur in about M seconds, then the system could warn of a possible password cracking attack, perhaps with a degree of confidence based on the number of attempts and the time interval. But it's not always easy to determine reasonable values for "about." Statistical analysis and heuristics are useful, but much care must be taken to minimize the false alarm rate. False alarms will quickly undermine confidence in any security system—like the boy who cried wolf, the security system that screams "attack" when none is present, will soon be ignored.

The advantages of signature-based detection include simplicity, efficiency (provided the number of signatures is not excessive), and an excellent ability to detect known attacks. Another major benefit is that the warning that is issued is specific, since the signature matches a specific attack pattern. With a specific warning, an administrator can quickly determine whether the suspected attack is real or a false alarm and, if it is real, the admin can usually respond appropriately.

The disadvantages of signature detection include the fact that the signature file must be current, the number of signatures may become large thereby reducing efficiency, and most importantly, the system can only detect known attacks. Even slight variations on known attack will likely be missed by signature-based systems.

Anomaly-based IDSs attempt to overcome the shortcomings of signature-based schemes. But no anomaly-based scheme available today could reasonably claim to be a replacement for signature-based detection. That is, an

anomaly-based system can supplement the performance of a signature-based system, but it is not a replacement for signature detection.

8.10.2 Anomaly-Based IDS

Anomaly-based IDSs look for unusual or abnormal behavior. There are several major challenges inherent in such an approach. First, we must determine what constitutes normal behavior for a system, and this must occur when the system is behaving normally. Second, the definition of normal must adapt as system usage changes and evolves, otherwise the number of false alarms will grow. Third, there are difficult statistical thresholding issues involved. For example, we must have a good idea of how far abnormal is away from normal.

Statistics are obviously necessary in the development of an anomaly-based IDS. Recall that the *mean* defines the statistical norm while the *variance* gives us a way to measure the distribution of the data about the mean. The mean and variance together gives us a way to determine abnormal behavior.

How can we measure normal system behavior? Whatever characteristics we decide to measure, we must take the measurements during times of representative behavior. In particular, we must not set the baseline measurements during an attack or else an attack will be considered normal. Measuring abnormal or, more precisely, determining how to separate normal variations in behavior from an attack, is an equally challenging problem. Abnormal must be measured relative to some specific value of normal. We'll consider abnormal as synonymous with attack, although in reality there are other possible causes of abnormal behavior, which further complicates the situation.

Statistical discrimination techniques are used to separate normal from abnormal. Examples of such techniques include Bayesian analysis, linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), neural nets, and hidden Markov models (HMM), among others. In addition, some anomaly detection researchers employ advanced modeling techniques from the fields of artificial intelligence and artificial immune systems. Such approaches are beyond the scope of our discussion here.

Next, we'll consider two simplified examples of anomaly detection. The first example is simple, but not very realistic, whereas the second is slightly less simple and correspondingly more realistic.

Suppose that we monitor the use of the three commands

```
open, read, close.
```

We find that under normal use, Alice uses the series of commands

```
open, read, close, open, open, read, close.
```

For our statistic, we'll consider pairs of consecutive commands and try to devise a measure of normal behavior for Alice. From Alice's series of commands, we observe that, of the six possible ordered pairs or commands, four pairs appear to be normal for Alice, namely,

(`open,read`), (`read,close`), (`close,open`), (`open,open`),

while the other two pairs,

(`read,open`), (`close,read`),

are not normally used by Alice. We can use this observation to identify potentially unusual behavior by "Alice" that might indicate an intruder is posing as Alice. We can then monitor the use of these three commands by Alice. If the ratio of abnormal to normal pairs is "too high," we would warn the administrator that an attack may be in progress.

This simple anomaly detection scheme can be improved. For example, we could include the expected frequency of each normal pair in the calculation, and if the observed pairs differ significantly from the expected distribution, we would warn of a possible attack. We might also try to improve the anomaly detection by using more than two consecutive commands, or by including more commands, or by including other user behavior in the model, or by using a more sophisticated statistical discrimination technique.

For a slightly more plausible anomaly detection scheme, let's focus on file access. Suppose that, over an extended period of time, Alice has accessed four files, F_0, F_1, F_2, F_3, at the rates H_0, H_1, H_2, H_3, respectively, where the observed values of the H_i are given in Table 8.4.

Table 8.4: Alice's Initial File Access Rates

H_0	H_1	H_2	H_3
0.10	0.40	0.40	0.10

Now suppose that, over a recent time interval, Alice has accessed file F_i at the rate A_i, for $i = 0, 1, 2, 3$, as given in Table 8.5. Do Alice's recent file access rates represent normal use? To decide, we need some way to compare her long-term access rates to the current rates. To answer this question, we'll employ the statistic

$$S = (H_0 - A_0)^2 + (H_1 - A_1)^2 + (H_2 - A_2)^2 + (H_3 - A_3)^2, \tag{8.2}$$

where we define $S < 0.1$ as normal. In this example, we have

$$S = (0.1 - 0.1)^2 + (0.4 - 0.4)^2 + (0.4 - 0.3)^2 + (0.1 - 0.2)^2 = 0.02,$$

and we conclude that Alice's recent use is normal—at least according to this one statistic.

Table 8.5: Alice's Recent File Access Rates

A_0	A_1	A_2	A_3
0.10	0.40	0.30	0.20

Alice's file access rates can be expected to vary over time, and we need to account for this in our IDS. We'll do so by updating Alice's long-term history values H_i according to the formula

$$H_i = 0.2 \cdot A_i + 0.8 \cdot H_i \quad \text{for } i = 0, 1, 2, 3. \tag{8.3}$$

That is, we update the historical access rates based on a moving average that combines the previous values with the recently observed rates—the previous values are weighted at 80%, while the current values are weighted 20%. Using the data in Tables 8.4 and 8.5, we find that the updated values of H_0 and H_1 are unchanged, whereas

$$H_2 = 0.2 \cdot 0.3 + 0.8 \cdot 0.4 = 0.38 \quad \text{and} \quad H_3 = 0.2 \cdot 0.2 + 0.8 \cdot 0.1 = 0.12.$$

These updated values appear in Table 8.6.

Table 8.6: Alice's Updated File Access Rates

H_0	H_1	H_2	H_3
0.10	0.40	0.38	0.12

Suppose that over the next time interval Alice's measured access rates are those given in Table 8.7. Then we compute the statistic S using the values in Tables 8.6 and 8.7 and the formula in equation (8.2) to find

$$S = (0.1 - 0.1)^2 + (0.4 - 0.3)^2 + (0.38 - 0.3)^2 + (0.12 - 0.3)^2 = 0.0488.$$

Since $S = 0.0488 < 0.1$ we again conclude that this is normal use for Alice. Again, we update Alice's long-term averages using the formula in (8.3) and

Table 8.7: Alice's More Recent File Access Rates

A_0	A_1	A_2	A_3
0.10	0.30	0.30	0.30

Table 8.8: Alice's Second Updated Access Rates

H_0	H_1	H_2	H_3
0.10	0.38	0.364	0.156

the data in Tables 8.6 and 8.7. In this case, we obtain the results that appear in Table 8.8.

Comparing Alice's long-term file access rates in Table 8.4 with her long-term averages after two updates, as given in Table 8.8, we see that the rates have changed significantly over time. Again, it is necessary that an anomaly-based IDS adapts over time, otherwise we will have a large number of false alarms (and a very annoyed system administrator) as Alice's actual behavior changes. However, this also presents an opportunity for the attacker, Trudy.

Since the H_i values slowly evolve to match Alice's behavior, Trudy can pose as Alice and remain undetected, provided she doesn't stray too far from Alice's usual behavior. But even more worrisome is the fact that Trudy can eventually convince the anomaly detection algorithm that her evil behavior is normal for Alice, provided Trudy has enough patience. For example, suppose that Trudy, posing as Alice, wants to always access file F_3. Then, initially, she can access file F_3 at a slightly higher rate than is normal for Alice. After the next update of the H_i values, Trudy will be able to access file F_3 at an even higher rate without triggering a warning from the anomaly detection software, and so on. By going slowly, Trudy will eventually convince the anomaly detector that it's normal for "Alice" to only access file F_3.

Note that $H_3 = 0.1$ in Table 8.4 and, two iterations later, $H_3 = 0.156$ in Table 8.8. These changes did not trigger a warning by the anomaly detector. Does this change represent a new usage pattern by Alice, or does it indicate an attempt by Trudy to trick the anomaly detector by going slow?

To make this anomaly detection scheme more robust, we should also incorporate the variance. In addition, we would certainly need to measure more than one statistic. If we measured N different statistics, $S_1, S_2, \ldots S_N$, we might combine them according to a formula such as

$$T = (S_1 + S_2 + S_3 + \ldots + S_N)/N$$

and make the determination of normal or abnormal based on the statistic T. This would provide a more comprehensive view of normal behavior and make it more difficult for Trudy, as she would need to approximate more of Alice's normal behavior. A similar—although much more sophisticated—approach is used in a popular IDS known as NIDES [9, 155]. NIDES incorporates both anomaly-based and signature-based IDSs. A good elementary introduction to NIDES, as well as several other IDSs, can be found in [304].

Robust anomaly detection is a difficult problem for a number of reasons. For one, system usage and user behavior constantly evolves and, therefore, so must the anomaly detector. Without allowing for such changes in behavior, false alarms would soon overwhelm the administrator, who would quickly lose confidence in the system. But an evolving anomaly detector means that it's possible for Trudy to slowly convince the anomaly detector that an attack is normal.

Another fundamental issue with anomaly detection is that a warning of abnormal behavior may not provide any useful specific information to the administrator. A vague warning that the system may be under attack could make it difficult to take concrete action. In contrast, a signature-based IDS will provide the administrator with precise information about the nature of the suspected attack.

The primary potential advantage of anomaly detection is that there is a chance of detecting previously unknown attacks. It's also sometimes argued that anomaly detection can be more efficient than signature detection, particularly if the signature file is large. In any case, the current generation of anomaly detectors must be used in combination with a signature-based IDS since they are not sufficiently robust to act as standalone systems.

Anomaly-based intrusion detection is an active research topic, and many security professionals have high hopes for its ultimate success. Anomaly detection is often cited as key future security technology [120]. But it appears that the hackers are not convinced, at least based on the title of a talk presented at a recent Defcon[15] conference: "Why anomaly-based intrusion detection systems are a hacker's best friend" [79].

The bottom line is that anomaly detection is a difficult and tricky problem. It also appears to have parallels with the field of artificial intelligence. Nearly a third of a century has passed since we were promised "robots on your doorstep" [327] and such predictions appear no more plausible today than at the time they were originally made. If anomaly-based intrusion detection proves to be anywhere near as challenging as AI, it may never live up to its claimed potential.

8.11 Summary

In this chapter we reviewed some of the history of authorization, with the focus on certification regimes. Then we covered the basics of traditional authorization, namely, Lampson's access control matrix, ACLs, and capabilities. The confused deputy problem was used to highlight the differences between ACLs and capabilities. We then presented some of the security issues related to multilevel security (MLS) and compartments, as well as the topics of covert channels and inference control. MLS naturally led us into the rarified air of security modeling, where we briefly considered Bell-LaPadula and Biba's Model.

[15]Defcon is the oldest, largest, and best-known hackers convention. It's held in Las Vegas each August, and it's inexpensive, totally chaotic, lots of fun, and hot (literally).

After covering the basics of security modeling, we pulled our heads out of the clouds, put our feet back on *terra firma*, and proceeded to discuss a few important non-traditional access control topics, including CAPTCHAs and firewalls. We concluded the chapter by stretching the definition of access control to cover intrusion detection systems (IDS). Many of the issues we discussed with respect to IDSs will resurface when we cover virus detection in Chapter 11.

8.12 Problems

1. On page 269 there is an example of orange book guidelines for testing at the so-called C division. Your skeptical author implies that these guidelines are somewhat dubious.

   a. Why might the guidelines that appear on page 269 not be particularly sensible or useful?

   b. Find three more examples of useless guidelines that appear in Part II of the orange book [309]. For each of these, summarize the guideline and give reasons why you feel it is not particularly sensible or useful.

2. The seven Common Criteria EALs are listed in Section 8.2.2. For each of these seven levels, summarize the testing required to achieve that level of certification.

3. In this chapter we discussed access control lists (ACLs) and capabilities (aka C-lists).

   a. Give two advantaged of capabilities over ACLs.

   b. Give two advantages of ACLs over capabilities.

4. In the text, we argued that it's easy to delegate using capabilities.

   a. It is also possible to delegate using ACLs. Explain how this would work.

   b. Suppose Alice delegates to Bill who then delegates to Charlie who, in turn, delegates to Dave. How would this be accomplished using capabilities? How would this be accomplished using ACLs? Which is easier and why?

   c. Which is better for delegation, ACLs or capabilities? Why?

5. Suppose Alice wants to temporarily delegate her C-list (capabilities) to Bob. Alice decides that she will digitally sign her C-list before giving it to Bob.
   a. What are the advantages, if any, of such an approach?
   b. What are the disadvantages, if any, of such an approach?
6. Briefly discuss one real-world application not mentioned in the text where multilevel security (MLS) would be useful.
7. What is the "need to know" principle and how can compartments be used to enforce this principle?
8. Suppose that you work in a classified environment where MLS is employed and you have a TOP SECRET clearance.
   a. Describe a potential covert channel involving the User Datagram Protocol (UDP).
   b. How could you minimize your covert channel in part a, while still allowing network access and communication by users with different clearances?
9. The *high water mark principle* and *low water mark principle* both apply in the realm of multilevel security.
   a. Define the high water mark principle and the low water mark principle in the context of MLS.
   b. Is BLP consistent with a high water mark principle, a low water mark principle, both, or neither? Justify your answer.
   c. Is Biba's Model consistent with a high water mark principle, a low water mark principle, both, or neither? Justify your answer.
10. This problem deals with covert channels.
    a. Describe a covert channel involving the print queue and estimate the realistic capacity of your covert channel.
    b. Describe a subtle covert channel involving the TCP network protocol.
11. We briefly discussed the following methods of inference control: query set size control; N-respondent, $k\%$ dominance rule; and randomization.
    a. Explain each of these three methods of inference control.
    b. Briefly discuss the relative strengths and weaknesses of each of these methods.

12. Inference control is used to reduce the amount of private information that can leak as a result of database queries.
    a. Discuss one practical method of inference control not mentioned in the book.
    b. How could you attack the method of inference control given in your solution to part a?
13. A *botnet* consists of a number of compromised machines that are all controlled by an evil botmaster [39, 146].
    a. Most botnets are controlled using the Internet Relay Chat (IRC) protocol. What is IRC and why is it particularly useful for controlling a botnet?
    b. Why might a covert channel be useful for controlling a botnet?
    c. Design a covert channel that could provide a reasonable means for a botmaster to control a botnet.
14. Read and briefly summarize each of the following sections from the article on covert channels at [131]: 2.2, 3.2, 3.3, 4.1, 4.2, 5.2, 5.3, 5.4.
15. Ross Anderson claims that "Some kinds of security mechanisms may be worse than useless if they can be compromised" [14].
    a. Does this statement hold true for inference control? Why or why not?
    b. Does this hold true for encryption? Why or why not?
    c. Does this hold true for methods that are used to reduce the capacity of covert channels? Why or why not?
16. Combine BLP and Biba's Model into a single MLS security model that covers both confidentiality and integrity.
17. BLP can be stated as "no read up, no write down." What is the analogous statement for Biba's Model?
18. Consider the visual CAPTCHA known as Gimpy [249].
    a. Explain how EZ Gimpy and Hard Gimpy work.
    b. How secure is EZ Gimpy compared to Hard Gimpy?
    c. Discuss the most successful known attack on each type of Gimpy.
19. This problem deals with visual CAPTCHAs.

a. Describe an example of a real-world visual CAPTCHA not discussed in the text and explain how this CAPTCHA works, that is, explain how a program would generate the CAPTCHA and score the result, and what a human would need to do to pass the test.

b. For the CAPTCHA in part a, what information is available to an attacker?

20. Design and implement your own visual CAPTCHA. Outline possible attacks on your CAPTCHA. How secure is your CAPTCHA?

21. This problem deals with audio CAPTCHAs.

a. Describe an example of a real-world audio CAPTCHA and explain how this CAPTCHA works, that is, explain how a program would generate the CAPTCHA and score the result, and what a human would need to do to pass the test.

b. For the CAPTCHA in part a, what information is available to an attacker?

22. Design and implement your own audio CAPTCHA. Outline possible attacks on your CAPTCHA. How secure is your CAPTCHA?

23. In [56] it is shown that computers are better than humans at solving all of the fundamental visual CAPTCHA problems, with the exception of the segmentation problem.

a. What are the fundamental visual CAPTCHA problems?

b. With the exception of the segmentation problem, how can computers solve each of these fundamental problems?

c. Intuitively, why is the segmentation problem more difficult for computers to solve?

24. The reCAPTCHA project is an attempt to make good use of the effort humans put into solving CAPTCHAs [322]. In reCAPTCHA, a user is shown two distorted words, where one of the words is an actual CAPTCHA, but the other is a word—distorted to look like a CAPTCHA—that an optical character recognition (OCR) program was unable to recognize. If the real CAPTCHA is solved correctly, then the reCAPTCHA program assumes that the other word was also solved correctly. Since humans are good at correcting OCR errors, reCAPTCHA can be used, for example, to improve the accuracy of digitized books.

a. It is estimated that about 200,000,000 CAPTCHAs are solved daily. Suppose that each of these is a reCAPTCHA and each requires about 10 seconds to solve. Then, in total, about how much time would be spent by users solving OCR problems each day? Note that we assume two CAPTCHAs are solved for one reCAPTCHA, so 200,000,000 CAPTCHAs represents 100,000,000 reCAPTCHAs.

b. Suppose that when digitizing a book, on average, about 10 hours of human effort is required to fix OCR problems. Under the assumptions in part a, how long would it take to correct all of the OCR problems created when digitizing all books in the Library of Congress? The Library of Congress has about 32,000,000 books, and we assume that every CAPTCHA in the world is a reCAPTCHA focused on this specific problem.

c. How could Trudy attack a reCAPTCHA system? That is, what could Trudy do to make the results obtained from a reCAPTCHA less reliable?

d. What could the reCAPTCHA developer do to minimize the effect of attacks on the system?

25. It has been widely reported that spammers sometimes pay humans to solve CAPTCHAs [293].

a. Why would spammers want to solve lots of CAPTCHAs?

b. What is the current cost, per CAPTCHA solved (in U.S. dollars), to have humans solve CAPTCHAs?

c. How might you entice humans to solve CAPTCHAs for you without paying them any money?

26. In this chapter, we discussed three types of firewalls: packet filter, stateful packet filter, and application proxy.

a. At which layer of the Internet protocol stack does each of these firewalls operate?

b. What information is available to each of these firewalls?

c. Briefly discuss one practical attack on each of these firewalls.

27. Commercial firewalls do not generally use the terminology packet filter, stateful packet filter, or application proxy. However, any firewall must be one of these three types, or a combination thereof. Find information on a commercial firewall product and explain (using the terminology of this chapter) which type of firewall it really is.

28. If a packet filter firewall does not allow reset (RST) packets out, then the TCP ACK scan described in the text will not succeed.

a. What are some drawbacks to this approach?

b. Could the TCP ACK scan attack be modified to work against such a system?

29. In this chapter it's stated that `Firewalk`, a port scanning tool, will succeed if the firewall is a packet filter or a stateful packet filter, but it will fail if the firewall is an application proxy.

    a. Why is this the case? That is, why does `Firewalk` succeed when the firewall is a packet filter or stateful packet filter, but fail when the firewall is an application proxy?

    b. Can `Firewalk` be modified to work against an application proxy?

30. Suppose that a packet filter firewall resets the TTL field to 255 for each packet that it allows through the firewall. Then the `Firewalk` port scanning tool described in the this chapter will fail.

    a. Why does `Firewalk` fail in this case?

    b. Does this proposed solution create any problems?

    c. Could `Firewalk` be modified to work against such a firewall?

31. An application proxy firewall is able to scan all incoming application data for viruses. It would be more efficient to have each host scan the application data it receives for viruses, since this would effectively distribute the workload among the hosts. Why might it still be preferable to have the application proxy perform this function?

32. Suppose incoming packets are encrypted with a symmetric key that is known only to the sender and the intended recipient. Which types of firewall (packet filter, stateful packet filter, application proxy) will work with such packets and which will not? Justify your answers.

33. Suppose that packets sent between Alice and Bob are encrypted and integrity protected by Alice and Bob with a symmetric key known only to Alice and Bob.

    a. Which fields of the IP header can be encrypted and which cannot?

    b. Which fields of the IP header can be integrity protected and which cannot?

    c. Which of the firewalls—packet filter, stateful packet filter, application proxy—will work in this case, assuming all IP header fields that can be integrity protected are integrity protected, and all IP header fields that can be encrypted are encrypted? Justify your answer.

34. Suppose that packets sent between Alice and Bob are encrypted and integrity protected by Alice's firewall and Bob's firewall with a symmetric key known only to Alice's firewall and Bob's firewall.

    a. Which fields of the IP header can be encrypted and which cannot?

    b. Which fields of the IP header can be integrity protected and which cannot?

    c. Which of the firewalls—packet filter, stateful packet filter, application proxy—will work in this case, assuming all IP header fields that can be integrity protected are integrity protected, and all IP header fields that can be encrypted are encrypted? Justify your answer.

35. Defense in depth using firewalls is illustrated in Figure 8.15. List other security applications where defense in depth is a sensible strategy.

36. Broadly speaking, there are two distinct types of intrusion detection systems, namely, signature-based and anomaly-based.

    a. List the advantages of signature-based intrusion detection, as compared to anomaly-based intrusion detection.

    b. List the advantages of an anomaly-based IDS, in contrast to a signature-based IDS.

    c. Why is effective anomaly-based IDS inherently more challenging than signature-based detection?

37. A particular vendor uses the following approach to intrusion detection.[16] The company maintains a large number of honeypots distributed across the Internet. To a potential attacker, these honeypots look like vulnerable systems. Consequently, the honeypots attract many attacks and, in particular, new attacks tend to show up on the honeypots soon after—sometimes even during—their development. Whenever a new attack is detected at one of the honeypots, the vendor immediately develops a signature and distributes the resulting signature to all systems using its product. The actual derivation of the signature is generally a manual process.

    a. What are the advantages, if any, of this approach as compared to a standard signature-based system?

    b. What are the advantages, if any, of this approach as compared to a standard anomaly-based system?

[16] This problem is based on a true story, just like many Hollywood movies...

c. Using the terminology given in this chapter, the system outlined in this problem would be classified as a signature-based IDS, not an anomaly-based IDS. Why?

d. The definition of signature-based and anomaly-based IDS are not standardized.[17] The vendor of the system outlined in this problem refers to it as an anomaly-based IDS. Why might they insist on calling it an anomaly-based IDS, when your well-nigh infallible author would classify it as a signature-based system?

38. The anomaly-based intrusion detection example presented in this chapter is based on file-use statistics.

a. Many other statistics could be used as part of an anomaly-based IDS. For example, network usage would be a sensible statistic to consider. List five other statistics that could reasonably be used in an anomaly-based IDS.

b. Why might it be a good idea to combine several statistics rather than relying on just a few?

c. Why might it not be a good idea to combine several statistics rather than relying on just a few?

39. Recall that the anomaly-based IDS example presented in this chapter is based on file-use statistics. The expected file use percentages (the H_i values in Table 8.4) are periodically updated using equation (8.3), which can be viewed as a moving average.

a. Why is it necessary to update the expected file use percentages?

b. When we update the expected file use percentages, it creates a potential avenue of attack for Trudy. How and why is this the case?

c. Discuss a different generic approach to constructing and updating an anomaly-based IDS.

40. Suppose that at the time interval following the results in Table 8.8, Alice's file-use statistics are given by $A_0 = 0.05$, $A_1 = 0.25$, $A_2 = 0.25$, and $A_3 = 0.45$.

a. Is this normal for Alice?

[17] Lack of standard terminology is a problem throughout most of the fields in information security (crypto being one of the few exceptions). It's important to be aware of this situation, since differing definitions is a common source of confusion. Of course, this problem is not unique to information security—differing definitions also cause confusion in many other fields of human endeavor. For proof, ask any two randomly selected economists about the current state of the economy.

b. Compute the updated values of H_0 through H_3.

41. Suppose that we begin with the values of H_0 through H_3 that appear in Table 8.4.

a. What is the minimum number of iterations required until it is possible to have $H_2 > 0.9$ without the IDS triggering a warning at any step?

b. What is the minimum number of iterations required until it is possible to have $H_3 > 0.9$ without the IDS triggering a warning at any step?

42. Consider the results given in Table 8.6.

a. For the subsequent time interval, what is the largest possible value for A_3 that will not trigger a warning from the IDS?

b. Give values for A_0, A_1, and A_2 that are compatible with the solution to part a.

c. Compute the statistic S, using the solutions from parts a and b, and the H_i values in Table 8.6.

Part III

Protocols

This page intentionally left blank

Chapter 9

Simple Authentication Protocols

"I quite agree with you," said the Duchess; "and the moral of that is—
'Be what you would seem to be'—or,
if you'd like it put more simply—'Never imagine yourself not to be
otherwise than what it might appear to others that what you were
or might have been was not otherwise than what you
had been would have appeared to them to be otherwise.' "
— Lewis Carroll, *Alice in Wonderland*

Seek simplicity, and distrust it.
— Alfred North Whitehead

9.1 Introduction

Protocols are the rules that are followed in some particular interaction. For example, there is a protocol that you follow if you want to ask a question in class, and it goes something like this:

1. You raise your hand.
2. The teacher calls on you.
3. You ask your question.
4. The teacher says, "I don't know."[1]

There are a vast number of human protocols, some of which can be very intricate, with numerous special cases to consider.

[1]Well, at least that's the way it works in your oblivious author's classes.

In the context of networking, protocols are the rules followed in networked communication systems. Examples of formal networking protocols include HTTP, FTP, TCP, UDP, PPP, and there are many, many more. In fact, the study of networks is largely the study of networking protocols.

Security protocols are the communication rules followed in security applications. In Chapter 10 we'll look closely at several real-world security protocols including SSH, SSL, IPSec, WEP, and Kerberos. In this chapter, we'll consider simplified authentication protocols so that we can better understand the fundamental security issues involved in the design of such protocols. If you want to delve a little deeper than the material presented in this chapter, the paper [3] has a discussion of some security protocol design principles.

In Chapter 7, we discussed methods that are used, primarily, to authenticate humans to a machines. In this chapter, we'll discuss authentication protocols. Although it might seem that these two authentication topics must be closely related, in fact, they are almost completely different. Here, we'll deal with the security issues related to the messages that are sent over a network to authenticate the participants. We'll see examples of well-known types of attacks on protocols and we'll show how to prevent these attacks. Note that our examples and analysis are informal and intuitive. The advantage of this approach is that we can cover all of the basic concepts quickly and with minimal background, but the price we pay is that some rigor is sacrificed.

Protocols can be subtle—often, a seemingly innocuous change makes a significant difference. Security protocols are particularly subtle, since the attacker can actively intervene in the process in a variety of ways. As an indication of the challenges inherent in security protocols, many well-known security protocols—including WEP, GSM, and even IPSec—have significant security issues. And even if the protocol itself is not flawed, a particular implementation can be.

Obviously, a security protocol must meet some specified security requirements. But we also want protocols to be efficient, both in computational cost and bandwidth usage. An ideal security protocol would not be too fragile, that is, the protocol would function correctly even when an attacker actively tries to break it. In addition, a security protocol should continue to work even if the environment in which it's deployed changes. Of course, it's impossible to protect against every possible eventuality, but protocol developers can try to anticipate likely changes in the environment and build in protections. Some of the most serious security challenges today are due to the fact that protocols are being used in environments for which they were not initially designed. For example, many Internet protocols were designed for a friendly, academic environment, which is about as far from the reality of the modern Internet as possible. Ease of use and ease of implementation are also desirable features of security protocols. Obviously, it's going to be difficult to design an ideal protocol.

9.2 Simple Security Protocols

The first security protocol that we consider is a protocol that could be used for entry into a secure facility, such as the National Security Agency. Employees are given a badge that they must wear at all times when in the secure facility. To enter the building, the badge is inserted into a card reader and the employee must provide a PIN number. The secure entry protocol can be described as follows.

1. Insert badge into reader.
2. Enter PIN.
3. Is the PIN correct?
   - Yes: Enter the building.
   - No: Get shot by a security guard.[2]

When you withdraw money from an ATM machine, the protocol is virtually identical to the secure entry protocol above, but without the violent ending:

1. Insert ATM card into reader
2. Enter PIN
3. Is the PIN correct?
   - Yes: Conduct your transactions
   - No: Machine eats your ATM card

The military has a need for many specialized security protocols. One example is an *identify friend or foe* (IFF) protocol. These protocols are designed to help prevent friendly-fire incidents—where soldiers accidentally attack soldiers on their own side—while not seriously hampering the fight against the enemy.

A simple example of an IFF protocol appears in Figure 9.1. This protocol was reportedly used by the South African Air Force, or SAAF, when fighting in Angola in the mid-1970s [14]. The South Africans were fighting Angola for control of Namibia (known as Southwest Africa at the time). The Angolan side was flying Soviet MiG aircraft, piloted by Cubans.[3]

[2]Of course, this is an exaggeration—you get three tries before being shot by the security guard.

[3]This was one of the hot wars that erupted during the Cold War. Early in the war, the South Africans were amazed by the skill of the "Angolan" pilots. They eventually realized the pilots were actually Cuban when satellite photos revealed baseball diamonds.

The IFF protocol in Figure 9.1 works as follows. When the SAAF radar detects an aircraft approaching its base, a random number, or *challenge*, N is sent to the aircraft. All SAAF aircraft have access to a key K that they use to encrypt the challenge, $E(N, K)$, which is computed and sent back to the radar station. Time is of the essence, so all of this happens automatically, without human intervention. Since enemy aircraft do not know K, they cannot send back the required response. It would seem that this protocol gives the radar station a simple way to determine whether an approaching aircraft is a friend (let it land) or foe (shoot it down).

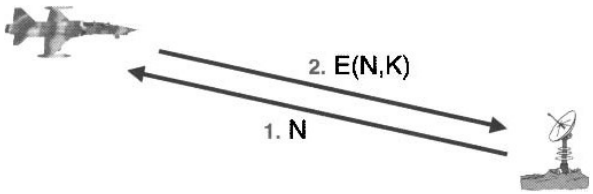


Figure 9.1: Identify Friend or Foe (IFF)

Unfortunately for those manning the radar station, there is a clever attack on the IFF system in Figure 9.1. Anderson has dubbed this attack the MiG-in-the-middle [14], which is a pun on man-in-the-middle. The scenario for the attack, which is illustrated in Figure 9.2, is as follows. While an SAAF Impala fighter is flying a mission over Angola, a Cuban-piloted MiG aircraft (the foe of the SAAF) loiters just outside of the range of the SAAF radar. When the Impala fighter is within range of a Cuban radar station in Angola, the MiG is told to move within range of the SAAF radar. As specified by the protocol, the SAAF radar sends the challenge N to the MiG. To avoid being shot down, the MiG needs to respond with $E(N, K)$, and quickly. Because the MiG does not know the key K, its situation appears hopeless. However, the MiG can forward the challenge N to its radar station in Angola, which, in turn, forwards the challenge to the SAAF Impala. The Impala fighter—not realizing that it has received the challenge from an enemy radar site—responds with $E(N, K)$. At this point, the Cuban radar relays the response $E(N, K)$ to the MiG, which can then provide it to the SAAF radar. Assuming this all happens fast enough, the SAAF radar will signal that the MiG is a friend, with disastrous consequences for the SAAF radar station and its operators.

Although it nicely illustrates an interesting security failure, it seems that this MiG-in-the-middle attack never actually occurred [15]. In any case, this is our first illustration of a security protocol failure, but it certainly won't be the last.

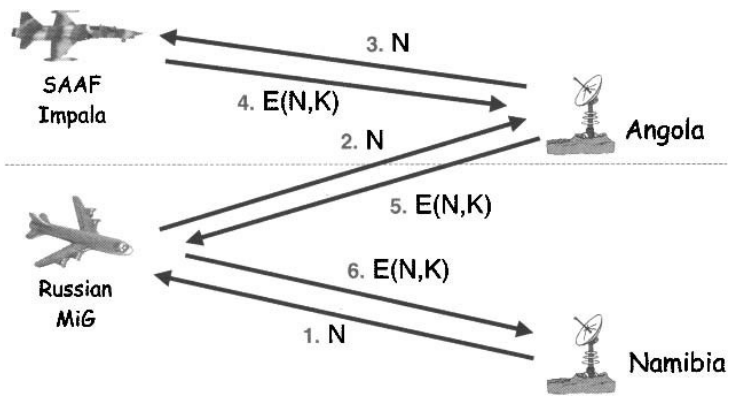


Figure 9.2: MiG-in-the-Middle

9.3 Authentication Protocols

"I can't explain myself, I'm afraid, Sir," said Alice,
"because I'm not myself you see."
— Lewis Carroll, *Alice in Wonderland*

Suppose that Alice must prove to Bob that she's Alice, where Alice and Bob are communicating over a network. Keep in mind that Alice can be a human or a machine, and ditto for Bob. In fact, in this networked scenario, Alice and Bob will almost invariably be machines, which has important implications that we'll consider in a moment.

In many cases, it's sufficient for Alice to prove her identity to Bob, without Bob proving his identity to Alice. But sometimes *mutual authentication* is necessary, that is, Bob must also prove his identity to Alice. It seems obvious that if Alice can prove her identity to Bob, then precisely the same protocol can be used in the other direction for Bob to prove his identity to Alice. We'll see that, in security protocols, the obvious approach is not always secure.

In addition to authentication, a *session key* is inevitably required. A session key is a symmetric key that will be used to protect the confidentiality and/or integrity of the current session, provided the authentication succeeds. Initially, we'll ignore the session key so that we can concentrate on authentication.

In certain situations, there may be other requirements placed on a security protocol. For example, we might require that the protocol use public keys, or symmetric keys, or hash functions. In addition, some situations might call for

a protocol that provides anonymity or plausible deniability (discussed below) or other not-so-obvious features.

We've previously considered the security issues associated with authentication on standalone computer systems. While such authentication presents its own set of challenges (hashing, salting, etc.), from the protocol perspective, it's straightforward. In contrast, authentication over a network requires very careful attention to protocol issues. When a network is involved, numerous attacks are available to Trudy that are generally not a concern on a standalone computer. When messages are sent over a network, Trudy can passively observe the messages and she can conduct various active attacks such as replaying old messages, inserting, deleting, or changing messages. In this book, we haven't previously encountered anything comparable to these types of attacks.

Our first attempt at authentication over a network is the protocol in Figure 9.3. This three-message protocol requires that Alice (the client) first initiate contact with Bob (the server) and state her identity. Then Bob asks for proof of Alice's identity, and Alice responds with her password. Finally, Bob uses Alice's password to authenticate Alice.

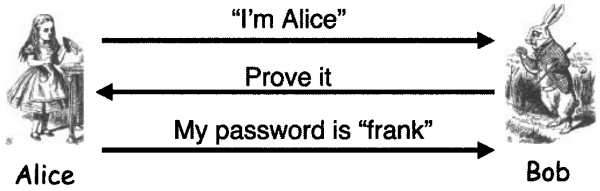


Figure 9.3: Simple Authentication

Although the protocol in Figure 9.3 is certainly simple, it has some major flaws. For one thing, if Trudy is able to observe the messages that are sent, she can later replay the messages to convince Bob that she is Alice, as illustrated in Figure 9.4. Since we are assuming these messages are sent over a network, this replay attack is a serious threat.

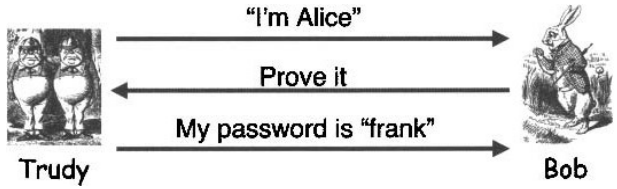


Figure 9.4: Replay Attack

Another issue with the too-simple authentication in Figure 9.3 is that Alice's password is sent in the clear. If Trudy observes the password when it is sent from Alice's computer, then Trudy knows Alice's password. This is even worse than a replay attack since Trudy can then pose as Alice on any site where Alice has reused this particular password. Another password issue with this protocol is that Bob must know Alice's password before he can authenticate her.

This simple authentication protocol is also inefficient, since the same effect could be accomplished in a single message from Alice to Bob. So, this protocol is a loser in every respect. Finally, note that the protocol in Figure 9.3 does not attempt to provide mutual authentication, which may be required in some cases.

For our next attempt at an authentication protocol, consider Figure 9.5. This protocol solves some of the problems of our previous simple authentication protocol. In this new-and-improved version, a passive observer, Trudy, will not learn Alice's password and Bob no longer needs to know Alice's password—although he must know the hash of Alice's password.

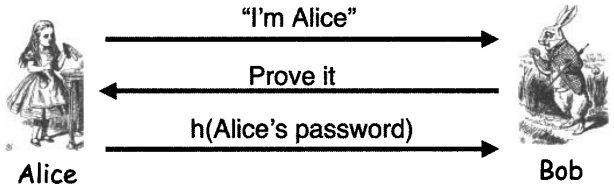


Figure 9.5: Simple Authentication with a Hash

The major flaw in the protocol of Figure 9.5 is that it's still subject to a replay attack, where Trudy records Alice's messages and later replays them to Bob. In this way, Trudy could be authenticated as Alice, without knowledge of Alice's password.

To authenticate Alice, Bob will need to employ a *challenge-response* mechanism. That is, Bob will send a challenge to Alice, and the response from Alice must be something that only Alice can provide and that Bob can verify. To prevent a replay attack, Bob can incorporate a "number used once," or *nonce*, in the challenge. That is, Bob will send a unique challenge each time, and the challenge will be used to compute the appropriate response. Bob can thereby distinguish the current response from a replay of a previous response. In other words, the nonce is used to ensure the freshness of the response. This approach to authentication with replay prevention is illustrated in Figure 9.6.

First, we'll design an authentication protocol using Alice's password. A password is something only Alice should know and Bob can verify—assuming that Bob knows Alice's password, that is.

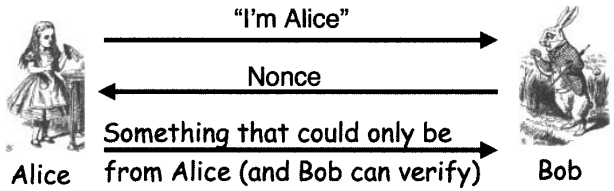


Figure 9.6: Generic Authentication

Our first serious attempt at an authentication protocol that is resistant to replay appears is Figure 9.7. In this protocol, the nonce sent from Bob to Alice is the challenge. Alice must respond with the hash of her password together with the nonce, which, assuming Alice's password is secure, serves to prove that the response was generated by Alice. Note that the nonce proves that the response is fresh and not a replay.

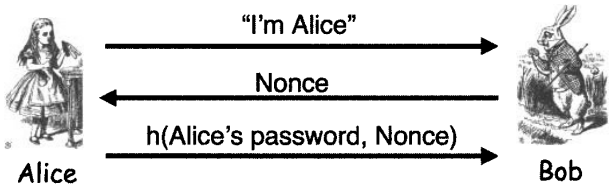


Figure 9.7: Challenge-Response

One problem with the protocol in Figure 9.7 is that Bob must know Alice's password. Furthermore, Alice and Bob typically represent machines rather than users, so it makes no sense to use passwords. After all, passwords are little more than a crutch used by humans because we are incapable of remembering keys. That is, passwords are about the closest thing to a key that humans can remember. So, if Alice and Bob are actually machines, they should be using keys instead of passwords.

9.3.1 Authentication Using Symmetric Keys

Having liberated ourselves from passwords, let's design a secure authentication protocol based on symmetric key cryptography. Recall that our notation for encrypting is $C = E(P, K)$ where P is plaintext, K is the key, and C is the ciphertext, while the notation for decrypting is $P = D(C, K)$. When discussing protocols, we are primarily concerned with attacks on protocols, not attacks on the cryptography used in protocols. Consequently, in this chapter we'll assume that the underlying cryptography is secure.

Suppose that Alice and Bob share the symmetric key K_{AB}. As in symmetric cryptography, we assume that nobody else has access to K_{AB}. Alice will authenticate herself to Bob by proving that she knows the key, without revealing the key to Trudy. In addition, the protocol must provide protection against a replay attack.

Our first symmetric key authentication protocol appears in Figure 9.8. This protocol is analogous to our previous password-based challenge-response protocol, but instead of hashing a nonce with a password, we've encrypted the nonce R with the shared symmetric key K_{AB}.

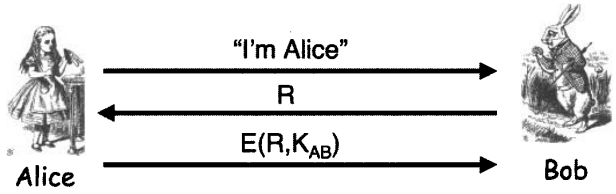


Figure 9.8: Symmetric Key Authentication Protocol

The symmetric key authentication protocol in Figure 9.8 allows Bob to authenticate Alice, since Alice can encrypt R with K_{AB}, Trudy cannot, and Bob can verify that the encryption was done correctly—Bob knows K_{AB}. This protocol prevents a replay attack, thanks to the nonce R, which ensures that each response is fresh. The protocol lacks mutual authentication, so our next task will be to develop a mutual authentication protocol based on symmetric keys.

Our first attempt at mutual authentication appears in Figure 9.9. This protocol is certainly efficient, and it does use symmetric key cryptography, but it has an obvious flaw. The third message in this protocol is simply a replay of the second, and consequently it proves nothing about the sender, be it Alice or Trudy.

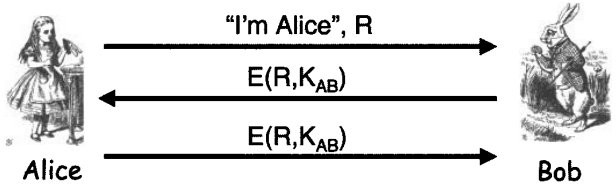


Figure 9.9: Mutual Authentication?

A more plausible approach to mutual authentication would be to use the secure authentication protocol in Figure 9.8 and repeat the process twice,

once for Bob to authenticate Alice and once more for Alice to authenticate Bob. We've illustrated this approach in Figure 9.10, where we've combined some messages for the sake of efficiency.

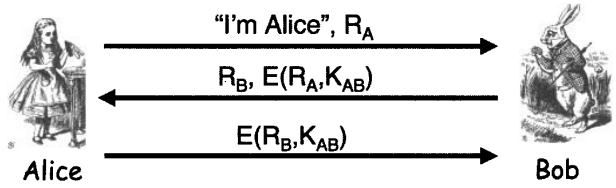


Figure 9.10: Secure Mutual Authentication?

Perhaps surprisingly, the protocol in Figure 9.10 is insecure—it is subject to an attack that is analogous to the MiG-in-the-middle attack discussed previously. In this attack, which is illustrated in Figure 9.11, Trudy initiates a conversation with Bob by claiming to be Alice and sends a challenge R_A to Bob. Following the protocol, Bob encrypts the challenge R_A and sends it, along with his challenge R_B, to Trudy. At this point Trudy appears to be stuck, since she doesn't know the key K_{AB}, and therefore she can't respond appropriately to Bob's challenge. However, Trudy cleverly opens a new connection to Bob where she again claims to be Alice and this time sends Bob his own "random" challenge R_B. Bob, following the protocol, responds with $E(R_B, K_{AB})$, which Trudy can now use to complete the first connection. Trudy can leave the second connection to time out, since she has—in the first connection—convinced Bob that she is Alice.

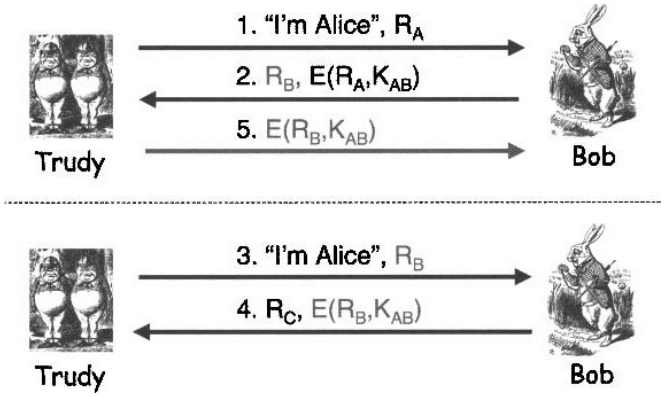


Figure 9.11: Trudy's Attack

The conclusion is that a non-mutual authentication protocol may not be secure for mutual authentication. Another conclusion is that protocols (and attacks on protocols) can be subtle. Yet another conclusion is that "obvious" changes to protocols can cause unexpected security problems.

In Figure 9.12, we've made a couple of minor changes to the insecure mutual authentication protocol of Figure 9.10. In particular, we've encrypted the user's identity together with the nonce. This change is sufficient to prevent Trudy's previous attack since she cannot use a response from Bob for the third message—Bob will realize that he encrypted it himself.

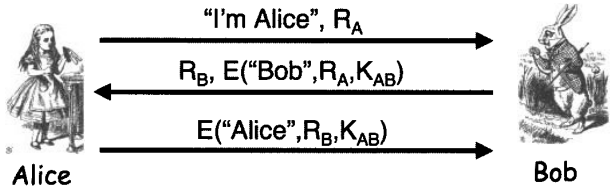


Figure 9.12: Strong Mutual Authentication Protocol

One lesson here is that it's a bad idea to have the two sides in a protocol do exactly the same thing, since this might open the door to an attack. Another lesson is that small changes to a protocol can result in big changes in its security.

9.3.2 Authentication Using Public Keys

In the previous section we devised a secure mutual authentication protocol using symmetric keys. Can we accomplish the same thing using public key cryptography? First, recall our public key notation. Encrypting a message M with Alice's public key is denoted $C = \{M\}_{\text{Alice}}$ while decrypt C with Alice's private key, and thereby recovering the plaintext M, is denoted $M = [C]_{\text{Alice}}$. Signing is also a private key operation. Of course, encryption and decryption are inverse operation, as are signing and signature verification, that is

$$[\{M\}_{\text{Alice}}]_{\text{Alice}} = M \quad \text{and} \quad \{[M]_{\text{Alice}}\}_{\text{Alice}} = M.$$

It's always important to remember that in public key cryptography, anybody can do public key operations, while only Alice can use her private key.[4]

Our first attempt at authentication using public key cryptography appears in Figure 9.13. This protocol allows Bob to authenticate Alice, since only Alice can do the private key operation that is necessary to reply with R in the third message. Also, assuming that the nonce R is chosen (by Bob) at

[4]Repeat to yourself 100 times: The public key is public.

random, a replay attack is not feasible. That is, Trudy cannot replay R from a previous iteration of the protocol, since the random challenge will almost certainly not be the same in a subsequent iteration.

However, if Alice uses the same key pair to encrypt as she uses for authentication, then there is a potential problem with the protocol in Figure 9.13. Suppose Trudy has previously intercepted a message encrypted with Alice's public key, say, $C = \{M\}_{\text{Alice}}$. Then Trudy can pose as Bob and send C to Alice in message two, and Alice will decrypt it and send the plaintext back to Trudy. From Trudy's perspective, it doesn't get any better than that. The moral of the story is that you should not use the same key pair for signing as you use for encryption.

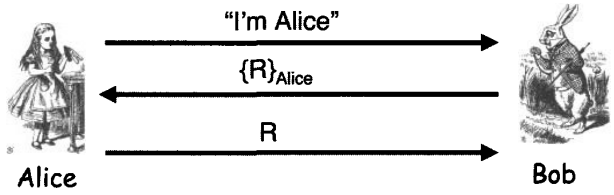


Figure 9.13: Authentication with Public Key Encryption

The authentication protocol in Figure 9.13 uses public key encryption. Is it possible to accomplish the same feat using digital signatures? In fact, it is, as illustrated in Figure 9.14.

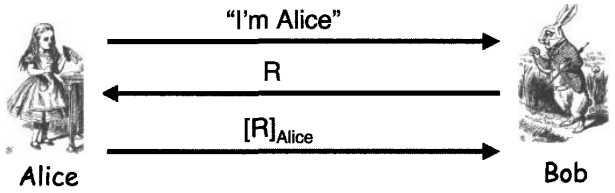


Figure 9.14: Authentication via Digital Signature

The protocol in Figure 9.14 has similar security issues as the public key encryption protocol in Figure 9.13. In Figure 9.14, if Trudy can pose as Bob, she can get Alice to sign anything. Again, the solution to this problem is to always use different key pairs for signing and encryption. Finally, note that, from Alice's perspective, the protocols in Figures 9.13 and 9.14 are identical, since in both cases she applies her private key to whatever shows up in message two.

9.3.3 Session Keys

Along with authentication, we invariably require a session key. Even when a symmetric key is used for authentication, we want to use a distinct session keys to encrypt data within each connection. The purpose of a session key is to limit the amount of data encrypted with any one particular key, and it also serves to limit the damage if one session key is compromised. A session key is used to provide confidentiality or integrity protection (or both) to the messages.

We want to establish the session key as part of the authentication protocol. That is, when the authentication is complete, we will also have securely established a shared symmetric key. Therefore, when analyzing an authentication protocol, we need to consider attacks on the authentication itself, as well as attacks on the session key.

Our next goal is to design an authentication protocol that also provides a shared symmetric key. It looks to be straightforward to include a session key in our secure public key authentication protocol. Such a protocol appears in Figure 9.15.

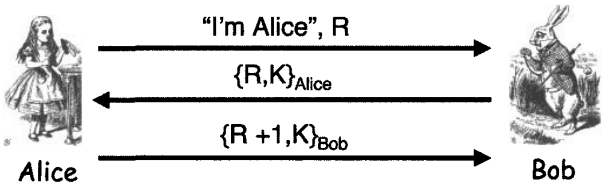


Figure 9.15: Authentication and a Session Key

One possible concern with the protocol of Figure 9.15 is that it does not provide for mutual authentication—only Alice is authenticated.[5] But before we tackle that issue, can we modify the protocol in Figure 9.15 so that it uses digital signatures instead of public key encryption? This also seems straightforward, and the result appears in Figure 9.16.

However, there is a fatal flaw in the protocol of Figure 9.16. Since the key is signed, anybody can use Bob's (or Alice's) public key and find the session key K. A session key that is public knowledge is definitely not secure. But before we dismiss this protocol entirely, note that it does provide mutual authentication, whereas the public key encryption protocol in Figure 9.15 does not. Can we combine these protocols so as to achieve both mutual

[5]One strange thing about this protocol is that the key K acts as Bob's challenge to Alice and the nonce R is useless. But there is a method to the madness, which will become clear shortly.

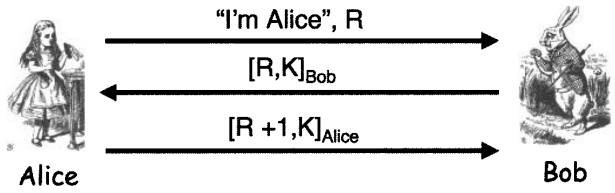


Figure 9.16: Signature-Based Authentication and Session Key

authentication and a secure session key? The answer is yes, and there are a couple of ways to do so.

Suppose that, instead of signing or encrypting the messages, we sign and encrypt the messages. Figure 9.17 illustrates such a sign and encrypt protocol. This appears to provide the desired secure mutual authentication and a secure session key.

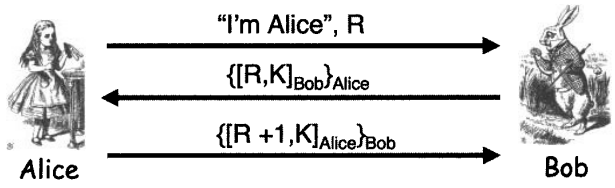


Figure 9.17: Mutual Authentication and Session Key

Since the protocol in Figure 9.17 provides mutual authentication and a session key using sign and encrypt, surely encrypt and sign must work, too. An encrypt and sign protocol appears in Figure 9.18.

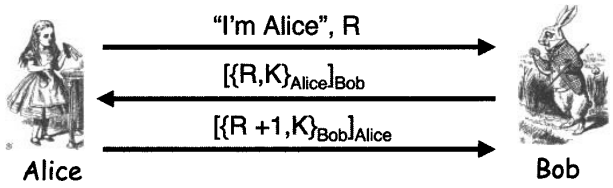


Figure 9.18: Encrypt and Sign Mutual Authentication

Note that the values $\{R, K\}_{\text{Alice}}$ and $\{R+1, K\}_{\text{Bob}}$ in Figure 9.18 are available to anyone who has access to Alice's or Bob's public keys (which, by assumption, is anybody who wants them). Since this is not the case in Figure 9.17, it might seem that sign and encrypt somehow reveals less

information than encrypt and sign. However, it appears that an attacker must break the public key encryption to recover K in either case and, if so, there is no security difference between the two. Recall that when analyzing protocols, we assume all crypto is strong, so breaking the encryption is not an option for Trudy.

9.3.4 Perfect Forward Secrecy

Now that we have conquered mutual authentication and session key establishment (using public keys), we turn our attention to *perfect forward secrecy*, or PFS. What is PFS? Rather than answer directly, we'll look at an example that illustrates what PFS is not. Suppose that Alice encrypts a message with a shared symmetric key K_{AB} and sends the resulting ciphertext to Bob. Trudy can't break the cipher to recover the key, so out of desperation she simply records all of the messages encrypted with the key K_{AB}. Now suppose that at some point in the future Trudy manages to get access to Alice's computer, where she finds the key K_{AB}. Then Trudy can decrypt the recorded ciphertext messages. While such an attack may seem unlikely, the problem is potentially significant since, once Trudy has recorded the ciphertext, the encryption key remains a vulnerability into the future. To avoid this problem, Alice and Bob must both destroy all traces of K_{AB} once they have finished using it. This might not be as easy as it seems, particularly if K_{AB} is a long-term key that Alice and Bob will need to use in the future. Furthermore, even if Alice is careful and properly manages her keys, she would have to rely on Bob to do the same (and vice versa).

PFS makes such an attack impossible. That is, even if Trudy records all ciphertext messages and she later recovers all long-term secrets (symmetric keys and/or private keys), she cannot decrypt the recorded messages. While it might seem that this is an impossibility, it is not only possible, but actually fairly easy to achieve in practice.

Suppose Bob and Alice share a long-term symmetric key K_{AB}. Then if they want PFS, they definitely can't use K_{AB} as their encryption key. Instead, Alice and Bob must agree on a session key K_S and forget K_S after it's no longer needed, i.e., after the current session ends. So, as in our previous protocols, Alice and Bob must find a way to agree on a session key K_S, by using their long-term symmetric key K_{AB}. However, for PFS we have the added condition that if Trudy later finds K_{AB}, she cannot determine K_S, even if she recorded all of the messages exchanged by Alice and Bob.

Suppose that Alice generates a session key K_S and sends $E(K_S, K_{AB})$ to Bob, that is, Alice simply encrypts the session key and sends it to Bob. If we are not concerned with PFS, this would be a sensible way to establish a session key in conjunction with an authentication protocol. However, this approach, which is illustrated in Figure 9.19, does not provide PFS. If Trudy records

all of the messages and later recovers K_{AB}, she can decrypt $E(K_S, K_{AB})$ to recover the session key K_S, which she can then use to decrypt the recorded ciphertext messages. This is precisely the attack that PFS is supposed to prevent.

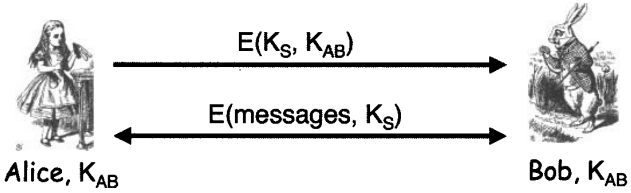


Figure 9.19: Naïve Attempt at PFS

There are actually several ways to achieve PFS, but the most elegant approach is to use an *ephemeral Diffie-Hellman* key exchange. As a reminder, the standard Diffie-Hellman key exchange protocol appears in Figure 9.20. In this protocol, g and p are public, Alice chooses her secret exponent a and Bob chooses his secret exponent b. Then Alice sends $g^a \bmod p$ to Bob and Bob sends $g^b \bmod p$ to Alice. Alice and Bob can each compute the shared secret $g^{ab} \bmod p$. Recall that the crucial weakness with Diffie-Hellman is that it is subject to a man-in-the-middle attack, as discussed in Section 4.4 of Chapter 4.

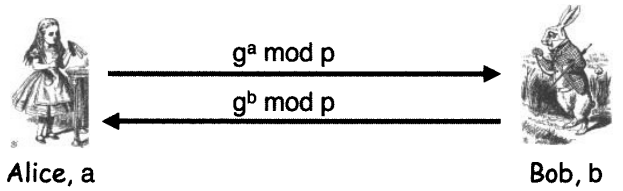


Figure 9.20: Diffie-Hellman

If we are to use Diffie-Hellman for PFS,[6] we must prevent the man-in-the-middle attack, and, of course, we must somehow assure PFS. The aforementioned ephemeral Diffie-Hellman can accomplish both. To prevent the MiM attack, Alice and Bob can use their shared symmetric key K_{AB} to encrypt the Diffie-Hellman exchange. Then to get PFS, all that is required is that, once Alice has computed the shared session key $K_S = g^{ab} \bmod p$, she must forget

[6]Your acronym-loving author was tempted to call this protocol DH4PFS or maybe EDH4PFS but, for once, he showed restraint.

her secret exponent a and, similarly, Bob must forget his secret exponent b. This protocol is illustrated in Figure 9.21.

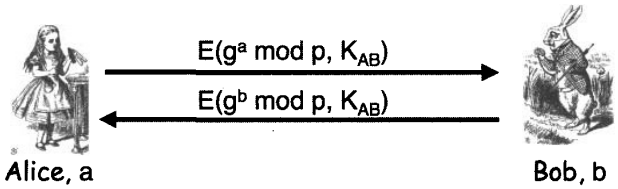


Figure 9.21: Ephemeral Diffie-Hellman for PFS

One interesting feature of the PFS protocol in Figure 9.21 is that once Alice and Bob have forgotten their respective secret exponents, even they can't reconstruct the session key K_S. If Alice and Bob can't recover the session key, certainly Trudy can be no better off. If Trudy records the conversation in Figure 9.21 and later is able to find K_{AB}, she will not be able to recover the session key K_S unless she can break Diffie-Hellman. Assuming the underlying crypto is strong, we have satisfied our requirements for PFS.

9.3.5 Mutual Authentication, Session Key, and PFS

Now let's put it all together and design a mutual authentication protocol that establishes a session key with PFS. The protocol in Figure 9.22, which is a slightly modified form of the encrypt and sign protocol from Figure 9.18, appears to fill the bill. It is a good exercise to give convincing arguments that Alice is actually authenticated (explaining exactly where and how that happens and why Bob is convinced he's talking to Alice), that Bob is authenticated, that the session key is secure, that PFS is provided, and that there are no obvious attacks.

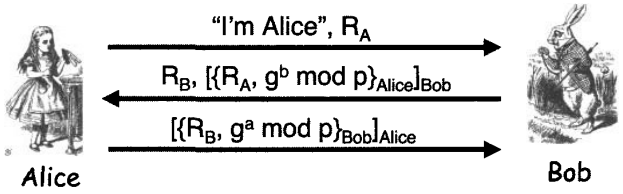


Figure 9.22: Mutual Authentication, Session Key and PFS

Now that we've developed a protocol that satisfies all of our security requirements, we can turn our attention to questions of efficiency. That is, we'll try to reduce the number of messages in the protocol or increase the efficiency in some other way, such as by reducing the number of public key operations.

9.3.6 Timestamps

A *timestamp* T is a time value, typically expressed in milliseconds. With some care, a timestamp can be used in place of a nonce, since a current timestamp ensures freshness. The benefit of a timestamp is that we don't need to waste any messages exchanging nonces, assuming that the current time is known to both Alice and Bob. Timestamps are used in many real-world security protocols, such as Kerberos, which we discuss in the next chapter.

Along with the potential benefit of increased efficiency, timestamps create some security issues as well.[7] For one thing, the use of timestamps implies that time is a security-critical parameter. For example, if Trudy can attack Alice's system clock (or whatever Alice relies on for the current time), she may cause Alice's authentication to fail. A related problem is that we can't rely on clocks to be perfectly synchronized, so we must allow for some *clock skew*, that is, we must accept any timestamp that is close to the current time. In general, this can open a small window of opportunity for Trudy to conduct a replay attack—if she acts within the allowed clock skew a replay will be accepted. It is possible to close this window completely, but the solution puts an additional burden on the server (see Problem 27). In any case, we would like to minimize the clock skew without causing excessive failures due to time inconsistencies between Alice and Bob.

To illustrate the benefit of a timestamp, consider the authentication protocol in Figure 9.23. This protocol is essentially the timestamp version of the sign and encrypt protocol in Figure 9.17. Note that by using a timestamp, we're able to reduce the number of messages by a third.

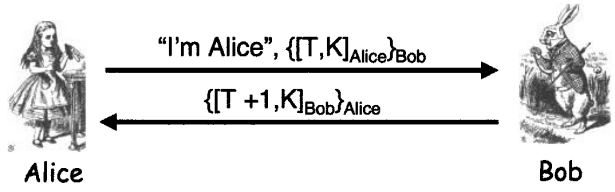


Figure 9.23: Authentication Using a Timestamp

The authentication protocol in Figure 9.23 uses a timestamp together with sign and encrypt and it appears to be secure. So it would seem obvious that the timestamp version of encrypt and sign must also be secure. This protocol is illustrated in Figure 9.24.

[7] This is yet another example of the "no free lunch" principle.

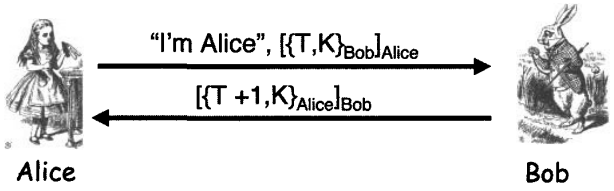


Figure 9.24: Encrypt and Sign Using a Timestamp

Unfortunately, with protocols, the obvious is not always correct. In fact, the protocol in Figure 9.24 is subject to attack. Trudy can recover $\{T, K\}_{\text{Bob}}$ by applying Alice's public key. Then Trudy can open a connection to Bob and send $\{T, K\}_{\text{Bob}}$ in message one, as illustrated in Figure 9.25. Following the protocol, Bob will then send the key K to Trudy in a form that Trudy can decrypt. This is not good, since K is the session key shared by Alice and Bob.

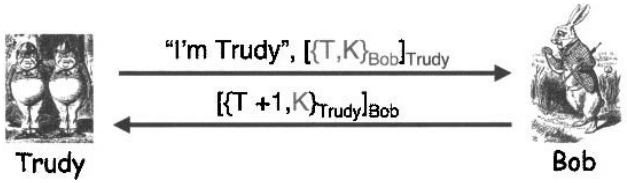


Figure 9.25: Trudy's Attack on Encrypt and Sign

The attack in Figure 9.25 shows that our encrypt and sign protocol is not secure when we use a timestamp. But our sign and encrypt protocol is secure when a timestamp is used. In addition, the nonce versions of both sign and encrypt as well as encrypt and sign are secure (see Figures 9.17 and 9.18). These examples nicely illustrate that, when it comes to security protocols, we should never take anything for granted.

Is the flawed protocol in Figure 9.24 fixable? In fact, there are several minor modifications that will make this protocol secure. For example, there's no reason to return the key K in the second message, since Alice already knows K and the only purpose of this message is to authenticate Bob. The timestamp in message two is sufficient to authenticate Bob. This secure version of the protocol is illustrated in Figure 9.26 (see also Problem 21).

In the next chapter, we'll discuss several well-known, real-world security protocols. These protocols use the concepts that we've presented in this chapter. But before moving on to the real world of Chapter 10, we briefly look at a couple of additional protocol topics. First, we'll consider a weak

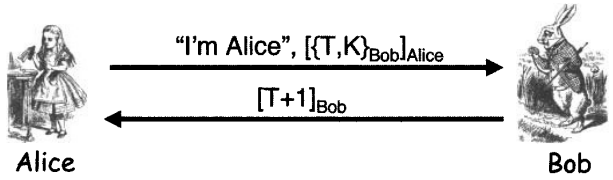


Figure 9.26: Secure Encrypt and Sign with a Timestamp

form of authentication that relies on TCP which, unfortunately, is sometimes used in practice. Finally, we discuss the Fiat-Shamir zero knowledge protocol. We'll encounter Fiat-Shamir again in the final chapter.

9.4 Authentication and TCP

In this section we'll take a quick look at how TCP is sometimes used for authentication. TCP was not designed to be used in this manner and, not surprisingly, this authentication method is not secure. But it does illustrate some interesting network security issues.

There is an undeniable temptation to use the IP address in a TCP connection for authentication.[8] If we could make this work, then we wouldn't need any of those troublesome keys or pesky authentication protocols.

Below, we'll give an example of TCP-based authentication and we illustrate an attack on the scheme. But first we briefly review the TCP three-way handshake, which is illustrated in Figure 9.27. The first message is a synchronization request, or SYN, whereas the second message, which acknowledges the synchronization request, is a SYN-ACK, and the third message—which can also contain data—acknowledges the previous message, and is simply known as an ACK.

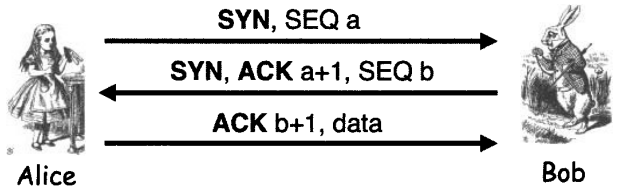


Figure 9.27: TCP 3-Way Handshake

[8] As we'll see in the next chapter, the IPSec protocol relies on the IP address for user identity in one of its modes. So, even people who should know better cannot always resist the temptation.

Suppose that Bob decides to rely on the completed three-way handshake to verify that he is connected to a specific IP address, which he knows belongs to Alice. Then, in effect, he is using the TCP connection to authenticate Alice. Since Bob sends the SYN-ACK to Alice's IP address, it's tempting to assume that the corresponding ACK must have come from Alice. In particular, if Bob verifies that ACK $b + 1$ appears in message three, he has some reason to believe that Alice, at her known IP address, has received message two and responded, since message two contains SEQ b and nobody else should know b. An underlying assumption here is that Trudy can't see the SYN-ACK packet—otherwise she would know b and she could easily forge the ACK. Clearly, this is not a strong form of authentication. However, as a practical matter, it might actually be difficult for Trudy to intercept the message containing b. So, if Trudy cannot see b, is the protocol secure?

Even if Trudy cannot see the initial SEQ number b, she might be able to make a reasonable guess. If so, the attack scenario illustrated in Figure 9.28 may be feasible. In this attack, Trudy first sends an ordinary SYN packet to Bob, who responds with a SYN-ACK. Trudy examines the SEQ value b_1 in this SYN-ACK packet. Suppose that Trudy can use b_1 to predict Bob's next initial SEQ value b_2.[9] Then Trudy can send a packet to Bob with the source IP address forged to be Alice's IP address. Bob will send the SYN-ACK to Alice's IP address which, by assumption, Trudy can't see. But, if Trudy can guess b_2, she can complete the three-way handshake by sending ACK $b_2 + 1$ to Bob. As a result, Bob will believe that data received from Trudy on this particular TCP connection actually came from Alice.

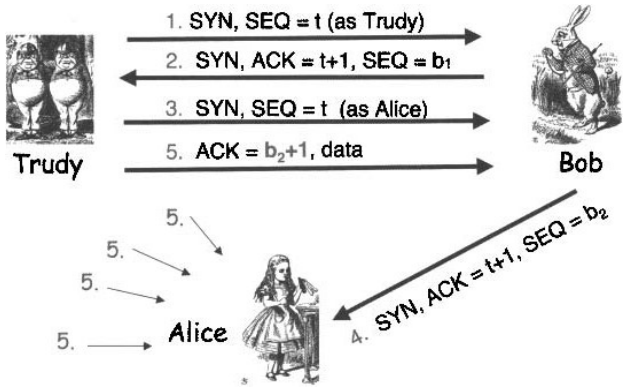


Figure 9.28: TCP "Authentication" Attack

[9]In practice, Trudy could send many SYN packets to Bob, trying to diagnose his initial sequence number generation scheme before actually attempting to guess a value.

Note that Bob always responds to Alice's IP address and, by assumption, Trudy cannot see his responses. But Bob will accept data from Trudy, thinking it came from Alice, as long as the connection remains active. However, when the data sent by Bob to Alice's IP address reaches Alice, Alice will terminate the connection since she has not completed the three-way handshake. To prevent this from happening, Trudy could mount a denial of service attack on Alice by sending enough messages so that Bob's messages can't get through—or, even if they do get through, Alice can't respond. This denial of service is illustrated Figure 9.28. Of course, if Alice happens to be offline, Trudy could conduct the attack without having to do this denial of service on Alice.

This attack is well known, and as a result initial SEQ numbers are supposed to be generated at random. So, how random are initial SEQ numbers? Surprisingly, they're often not very random at all. For example, Figure 9.29 provides a visual comparison of random initial SEQ numbers versus the highly biased initial SEQ numbers generated under an early version of Mac OS X. The Mac OS X numbers are biased enough that the attack in Figure 9.28 would have a reasonable chance of success. Many other vendors fail to generate random initial SEQ numbers, as can be seen from the fascinating pictures at [335].

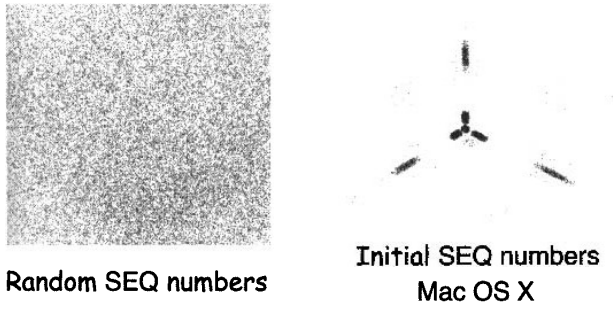


Figure 9.29: Plots of Initial SEQ Numbers (Courtesy of Michal Zalewski [335])

Even if initial SEQ numbers are random, it's a bad idea to rely on a TCP connection for authentication. A much better approach would be to employ a secure authentication protocol after the three-way handshake has completed. Even a simple password scheme would be far superior to relying on TCP. But, as often occurs in security, the TCP authentication method is sometimes used in practice simply because it's there, it's convenient, and it doesn't annoy users—not because it's secure.

9.5 Zero Knowledge Proofs

In this section we'll discuss a fascinating authentication scheme developed by Fiege, Fiat, and Shamir [111] (yes, that Shamir), but usually known simply as Fiat-Shamir. We'll mention this method again in Chapter 13 when we discuss Microsoft's trusted operating system.

In a *zero knowledge proof*,[10] or ZKP, Alice wants to prove to Bob that she knows a secret without revealing any information about the secret—neither Trudy nor Bob can learn anything about the secret. Bob must be able to verify that Alice knows the secret, even though he gains no information about the secret. On the face of it, this sounds impossible. However, there is an interactive probabilistic process whereby Bob can verify that Alice knows a secret to an arbitrarily high probability. This is an example of an interactive proof system.

Before describing such a protocol, we first consider Bob's Cave,[11] which appears in Figure 9.30. Suppose that Alice claims to know the secret phrase ("open sarsaparilla"[12]) that opens the door between R and S in Figure 9.30. Can Alice convince Bob that she knows the secret phrase without revealing any information about it?

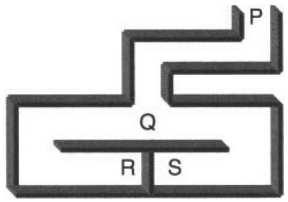


Figure 9.30: Bob's Cave

Consider the following protocol. Alice enters Bob's Cave and flips a coin to decide whether to position herself at point R or S. Bob then enters the cave and proceeds to point Q. Suppose that Alice happens to be positioned at point R. This situation is illustrated in Figure 9.31.

Then Bob flips a coin to randomly select one side or the other and asks Alice to appear from that side. With the situation as in Figure 9.31, if Bob happens to select side R, then Alice would appear at side R whether she knows the secret phrase or not. But if Bob happens to choose side S, then Alice can only appear on side S if she knows the secret phrase that opens

[10] Not to be confused with a "zero knowledge Prof."

[11] Traditionally, Ali Baba's Cave is used here.

[12] Traditionally, the secret phrase is "open says me," which sounds a lot like "open sesame." In the cartoon world, "open sesame" somehow became "open sarsaparilla" [242].

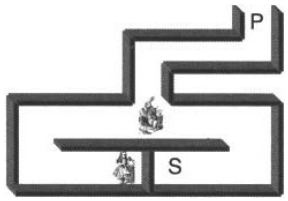


Figure 9.31: Bob's Cave Protocol

the door between R and S. In other words, if Alice doesn't know the secret phrase, the probability that she can trick Bob into believing that she does is $\frac{1}{2}$. This does not seem particularly useful, but if the protocol is repeated n times, then the probability that Alice can trick Bob every time is only $(\frac{1}{2})^n$. So, Alice and Bob will repeat the protocol n times and Alice must pass every time before Bob will believe she knows the secret phrase.

Note that if Alice (or Trudy) does not know the secret phrase, there is always a chance that she can trick Bob into believing that she does. However, Bob can make this probability as small as he desires by choosing n appropriately. For example, with $n = 20$ there is less than a 1 in 1,000,000 chance that "Alice" would convince Bob that she knows the phrase when she does not. Also, Bob learns nothing about the secret phrase in this protocol. Finally, it is critical that Bob randomly chooses the side where he asks Alice to appear—if Bob's choice is predictable, then Alice (or Trudy) would have a better chance of tricking Bob and thereby breaking the protocol.

While Bob's Cave indicates that zero knowledge proofs are possible in principle, cave-based protocols are not particularly popular. Can we achieve the same effect without the cave? The answer is yes, thanks to the Fiat-Shamir protocol.

Fiat-Shamir relies on the fact that finding a square root modulo N is as difficult as factoring. Suppose $N = pq$, where p and q are prime. Alice knows a secret S, which, of course, she must keep secret. The numbers N and $v = S^2 \bmod N$ are public. Alice must convince Bob that she knows S without revealing any information about S.

The Fiat-Shamir protocol, which is illustrated in Figure 9.32, works as follows. Alice randomly selects a value r, and she computes $x = r^2 \bmod N$. In message one, Alice sends x to Bob. In message two, Bob chooses a random value $e \in \{0, 1\}$, which he sends to Alice who, in turn, then computes the quantity $y = rS^e \bmod N$. In the third message Alice sends y to Bob. Finally, Bob needs to verify that

$$y^2 = xv^e \bmod N,$$

which, if everyone has followed the protocol, holds true since

$$y^2 = r^2 S^{2e} = r^2 (S^2)^e = xv^e \bmod N. \tag{9.1}$$

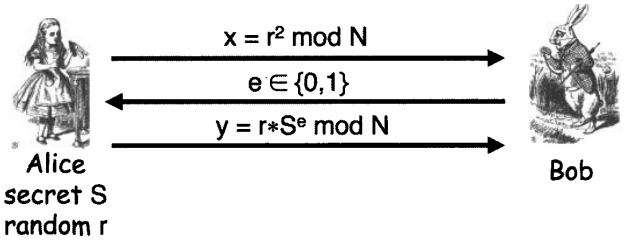


Figure 9.32: Fiat-Shamir Protocol

In message two, Bob sends either $e = 0$ or $e = 1$. Let's consider these cases separately. If Bob sends $e = 1$, then Alice responds with $y = r \cdot S \bmod N$ in the third message, and equation (9.1) becomes

$$y^2 = r^2 \cdot S^2 = r^2 \cdot (S^2) = x \cdot v \bmod N.$$

Note that in this case, Alice must know the secret S.

On the other hand, if Bob sends $e = 0$ in message two, then Alice responds in the third message with $y = r \bmod N$ and equation (9.1) becomes

$$y^2 = r^2 = x \bmod N.$$

Note that in this case, Alice does not need to know the secret S. This may seem strange, but it's roughly equivalent to the situation in Bob's Cave where Alice did not need to open the secret passage to come out on the correct side. Regardless, it is tempting to have Bob always send $e = 1$. However, we'll see in a moment that that this would not be wise.

The first message in the Fiat-Shamir protocol is the *commitment* phase, since Alice commits to her choice of r by sending $x = r^2 \bmod N$ to Bob. That is, Alice cannot change her mind (she is committed to r), but she has not revealed r, since finding modular square roots is hard. The second message is the *challenge* phase—Bob is challenging Alice to provide the correct response. The third message is the *response* phase, since Alice must respond with the correct value. Bob then verifies the response using equation (9.1). These phases correspond to the three steps in Bob's Cave protocol in Figure 9.31, above.

The mathematics behind the Fiat-Shamir protocol works, that is, assuming everyone follows the protocol, Bob can verify $y^2 = xv^e \bmod N$ from the

information he receives. But this does not establish the security of the protocol. To do so, we must determine whether an attacker, Trudy, can convince Bob that she knows Alice's secret S and thereby convince Bob that she is Alice.

Suppose Trudy expects Bob to send the challenge $e = 0$ in message two. Then Trudy can send $x = r^2 \bmod N$ in message one and $y = r \bmod N$ in message three. That is, Trudy simply follows the protocol in this case, since she does not need to know the secret S.

On the other hand, if Trudy expects Bob to send $e = 1$, then she can send $x = r^2 v^{-1} \bmod N$ in message one and $y = r \bmod N$ in message three. Following the protocol, Bob will compute $y^2 = r^2$ and $xv^e = r^2 v^{-1} v = r^2$ and he will find that equation (9.1) holds. Bob therefore accepts the result as valid.

The conclusion here is that Bob must choose $e \in \{0, 1\}$ at random (as specified by the protocol). If so, then Trudy can only trick Bob with probability $\frac{1}{2}$, and, as with Bob's Cave, after n iterations, the probability that Trudy can fool Bob is only $(\frac{1}{2})^n$.

So, Fiat-Shamir requires that Bob's challenge $e \in \{0, 1\}$ be unpredictable. In addition, Alice must generate a random r at each iteration of the protocol or her secret S will be revealed (see Problem 40 at the end of this chapter).

Is the Fiat-Shamir protocol really zero knowledge? That is, can Bob—or anyone else—learn anything about Alice's secret S? Recall that v and N are public, where $v = S^2 \bmod N$. In addition, Bob sees $r^2 \bmod N$ in message one, and, assuming $e = 1$, Bob sees $rS \bmod N$ in message three. If Bob can find r from $r^2 \bmod N$, then he can find S. But finding modular square roots is computationally infeasible. If Bob were somehow able to find such square roots, he could obtain S directly from the public value v without bothering with the protocol at all. While this is not a rigorous proof that Fiat-Shamir is zero knowledge, it does indicate that there is nothing obvious in the protocol itself that helps Bob (or anyone else) to determine Alice's secret S.

Is there an security benefit of Fiat-Shamir, or is it just fun and games for mathematicians? If public keys are used for authentication, then each side must know the other side's public key. At the start of the protocol, typically Alice would not know Bob's public key, and vice versa. So, in many public key-based protocols Bob sends his certificate to Alice. But the certificate identifies Bob, and consequently this exchange would tell Trudy that Bob is a party to the transaction. In other words, public keys make it hard for the participants to remain anonymous.

A potential advantage of zero knowledge proofs is that they allow for authentication with anonymity. In Fiat-Shamir, both sides must know the public value v, but there is nothing in v that identifies Alice, and there is nothing in the messages that are passed that must identify Alice. This is an advantage that has led Microsoft to include support for zero knowledge

proofs in its Next Generation Secure Computing Base, or NGSCB, which we'll discuss in Chapter 13. The bottom line is that Fiat-Shamir does have some potential practical utility.

9.6 The Best Authentication Protocol?

In general there is no "best" authentication protocol. What is best for a particular situation will depend on many factors. At a minimum, we need to consider the following questions.

- What is the sensitivity of the application?
- How much delay is tolerable?
- Do we want to deal with time as a security critical parameter?
- What type of crypto is supported—public key, symmetric key, or hash functions?
- Is mutual authentication required?
- Is a session key required?
- Is perfect forward secrecy desired?
- Is anonymity a concern?

In the next chapter, we'll see that there are additional issues that can influence our choice of protocol.

9.7 Summary

In this chapter we discussed several different ways to authenticate and establish a session key over an insecure network. We can accomplish these feats using symmetric keys, public keys, or hash functions (with symmetric keys). We also learned how to achieve perfect forward secrecy, and we considered the benefits (and potential drawbacks) of using timestamps.

Along the way, we came across many security pitfalls. You should now have some appreciation for the subtle issues that can arise with security protocols. This will be useful in the next chapter where we look closely at several real-world security protocols. We'll see that, despite extensive development effort by lots of smart people, such protocols are not immune to some of the security flaws highlighted in this chapter.

9.8 Problems

1. Modify the authentication protocol in Figure 9.12 so that it uses a hash function instead of symmetric key encryption. The resulting protocol must be secure.

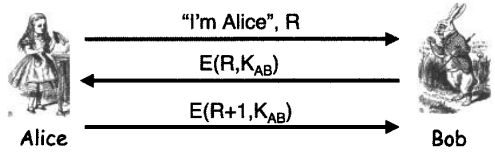
2. The insecure protocol in Figure 9.24 was modified in Figure 9.26 to be secure. Find two other distinct ways to slightly modify the protocol in Figure 9.24 so that the resulting protocol is secure. Your protocols must use a timestamp and "encrypt and sign."

3. We want to design a secure mutual authentication protocol based on a shared symmetric key. We also want to establish a session key, and we want perfect forward secrecy.

   a. Design such a protocol that uses three messages.

   b. Design such a protocol that uses two messages.

4. Consider the following mutual authentication protocol, where K_{AB} is a shared symmetric key.



   Give two different attacks that Trudy can use to convince Bob that she is Alice.

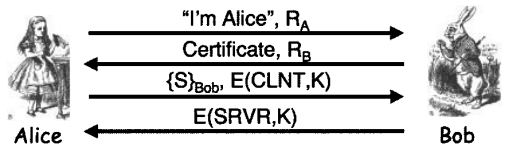
5. Consider the attack on TCP authentication illustrated in Figure 9.28. Suppose that Trudy cannot guess the initial sequence number b_2 exactly. Instead, Trudy can only narrow b_2 down to one of, say, 1,000 possible values. How can Trudy conduct an attack so that she is likely to succeed?

6. Timestamps can be used in place of nonces in security protocols.

   a. What is the primary advantage of using timestamps?

   b. What is the primary disadvantage of using timestamps?

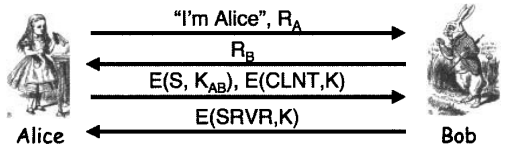
7. Consider the following protocol, where CLNT and SRVR are constants, and the session key is $K = h(S, R_A, R_B)$.



a. Does Alice authenticate Bob? Justify your answer.

b. Does Bob authenticate Alice? Justify your answer.

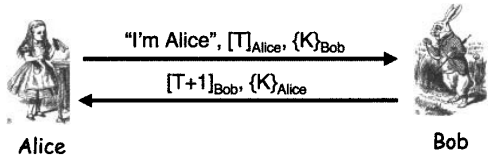
8. Consider the following protocol, where K_{AB} is a shared symmetric key, CLNT and SRVR are constants, and $K = h(S, R_A, R_B)$ is the session key.



a. Does Alice authenticate Bob? Justify your answer.

b. Does Bob authenticate Alice? Justify your answer.

9. The following two-message protocol is designed for mutual authentication and to establish a session key K. Here, T is a timestamp.



This protocol is insecure. Illustrate a successful attack by Trudy.

10. Suppose R is a random challenge sent in the clear from Alice to Bob and K is a symmetric key known only to Alice and Bob. Which of the following are secure session keys and which are not? Justify your answers.

a. $R \oplus K$

b. $E(R, K)$

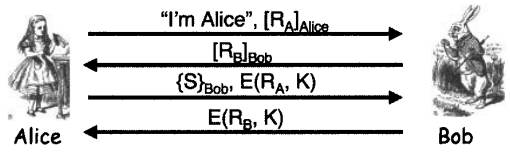
c. $E(K, R)$

d. $h(K, R)$

e. $h(R, K)$

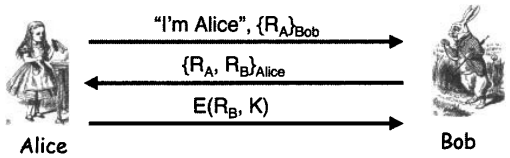
11. Design a secure two-message authentication protocol that provides mutual authentication and establishes a session key K. Assume that Alice and Bob know each other's public keys beforehand. Does your protocol protect the anonymity of Alice and Bob from a passive attacker (i.e., an attacker who can only observe messages sent between Alice and Bob)? If not, modify your protocol so that it does provide anonymity.

12. For some particular security protocol, suppose that Trudy can construct messages that appear to any observer (including Alice and/or Bob) to be valid messages between Alice and Bob. Then the protocol is said to provide *plausible deniability*. The idea here is that Alice and Bob can (plausibly) argue that any conversation they had using the protocol never actually occurred—it could have been faked by Trudy. Consider the following protocol, where $K = h(R_A, R_B, S)$.



Does this protocol provide plausible deniability? If so, why? If not, slightly modify the protocol so that it does, while still providing mutual authentication and a secure session key.

13. Consider the following protocol where $K = h(R_A, R_B)$.

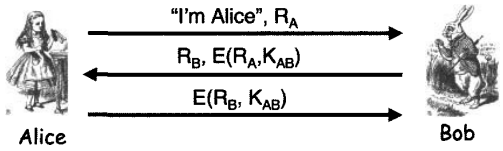


Does this protocol provide for plausible deniability (see Problem 12)? If so, why? If not, slightly modify the protocol so that it does, while still providing mutual authentication and a secure session key.

14. Design a mutual authentication protocol that employs digital signatures for authentication and provides plausible deniability (see Problem 12).

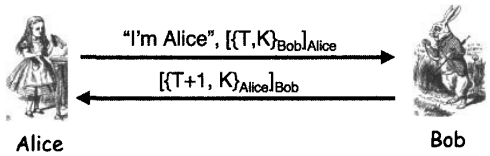
15. Is plausible deniability (see Problem 12) a feature or a security flaw? Explain.

16. The following mutual authentication protocol is based on a shared symmetric key K_{AB}.



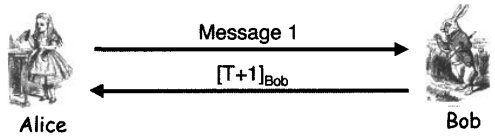
Show that Trudy can attack the protocol to convince Bob that she is Alice, where, as usual, we assume that the cryptography is secure. Modify the protocol to prevent such an attack by Trudy.

17. Consider the following mutual authentication and key establishment protocol, which employs a timestamp T and public key cryptography.



Show that Trudy can attack the protocol to discover the key K where, as usual, we assume that the cryptography is secure. Modify the protocol to prevent such an attack by Trudy.

18. Consider the following mutual authentication and key establishment protocol, which uses a timestamp T and public key cryptography.



For each of the following cases, explain whether or not the resulting protocol provides an effective means for secure mutual authentication and a secure session key K. Ignore replay attacks based solely on the clock skew.

a. Message 1: $\{[T, K]_{\text{Alice}}\}_{\text{Bob}}$

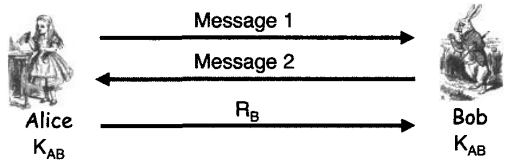
b. Message 1: $\{\text{“Alice”}, [T, K]_{\text{Alice}}\}_{\text{Bob}}$

c. Message 1: $\text{“Alice”}, \{[T, K]_{\text{Alice}}\}_{\text{Bob}}$

d. Message 1: T, $\text{“Alice”}, \{[K]_{\text{Alice}}\}_{\text{Bob}}$

e. Message 1: $\text{“Alice”}, \{[T]_{\text{Alice}}\}_{\text{Bob}}$ and let $K = h(T)$

19. Consider the following three-message mutual authentication and key establishment protocol, which is based on a shared symmetric key K_{AB}.



For each of the following cases, briefly explain whether or not the resulting protocol provides an effective means for secure mutual authentication and a secure session key K.

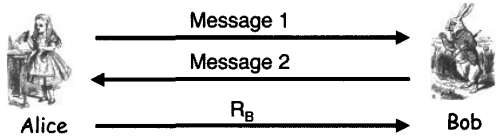
a. Message 1: $E(\text{“Alice”}, K, R_A, K_{AB})$, Message 2: R_A, $E(R_B, K_{AB})$

b. Message 1: “Alice”, $E(K, R_A, K_{AB})$, Message 2: R_A, $E(R_B, K)$

c. Message 1: “Alice”, $E(K, R_A, K_{AB})$, Message 2: R_A, $E(R_B, K_{AB})$

d. Message 1: “Alice”, R_A, Message 2: $E(K, R_A, R_B, K_{AB})$

20. Consider the following three-message mutual authentication and key establishment protocol, which is based on public key cryptography.



For each of the following cases, briefly explain whether or not the resulting protocol provides an effective means for secure mutual authentication and a secure session key K.

a. Message 1: $\{\text{“Alice”}, K, R_A\}_{\text{Bob}}$, Message 2: R_A, R_B

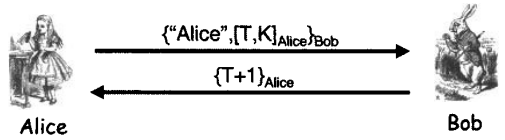
b. Message 1: “Alice”, $\{K, R_A\}_{\text{Bob}}$, Message 2: R_A, $\{R_B\}_{\text{Alice}}$

c. Message 1: “Alice”, $\{K\}_{\text{Bob}}$, $[R_A]_{\text{Alice}}$, Message 2: R_A, $[R_B]_{\text{Bob}}$

d. Message 1: R_A, $\{\text{“Alice”}, K\}_{\text{Bob}}$, Message 2: $[R_A]_{\text{Bob}}$, $\{R_B\}_{\text{Alice}}$

e. Message 1: $\{\text{“Alice”}, K, R_A, R_B\}_{\text{Bob}}$, Message 2: R_A, $\{R_B\}_{\text{Alice}}$

21. Consider the following mutual authentication and key establishment protocol (it may be instructive to compare this protocol to the protocol in Figure 9.26).

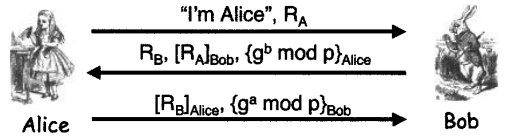


Suppose that Trudy pretends to be Bob. Further, suppose that Trudy can guess the value of T to within 5 minutes, and the resolution of T is to the nearest millisecond.

a. What is the probability that Trudy can send a correct response in message two, causing Alice to erroneously authenticate Trudy as Bob?

b. Give two distinct modifications to the protocol, each of which make Trudy's attack more difficult, if not impossible.

22. Consider the following mutual authentication and key establishment protocol, where the session key is given by $K = g^{ab} \bmod p$.



Suppose that Alice attempts to initiate a connection with Bob using this protocol.

a. Show that Trudy can attack the protocol so that both of the following will occur.

   i. Alice and Bob authenticate each other.

   ii. Trudy knows Alice's session key.

   Hint: Consider a man-in-the-middle attack.

b. Is this attack of any use to Trudy?

23. For each of the following cases, design a mutual authentication and key establishment protocol that uses public key cryptography and minimizes the number of messages.

a. Use a timestamp to authenticate Alice and a nonce to authenticate Bob.

b. Use a nonce to authenticate Alice and a timestamp to authenticate Bob.

24. Suppose we replace the third message of the protocol in Figure 9.22 with

$$\{R_B\}_{\text{Bob}}, \quad g^a \bmod p.$$

a. How can Trudy convince Bob that she is Alice, that is, how can Trudy break the authentication?

b. Can Trudy convince Bob that she is Alice and also determine the session key that Bob will use?

25. Suppose we replace the second message of the protocol in Figure 9.22 with

$$R_B, \quad [R_A]_{\text{Bob}}, \quad g^b \bmod p,$$

and we replace the third message with

$$[R_B]_{\text{Alice}}, \quad g^a \bmod p.$$

a. Can Trudy convince Bob that she is Alice, that is, can Trudy break the authentication?

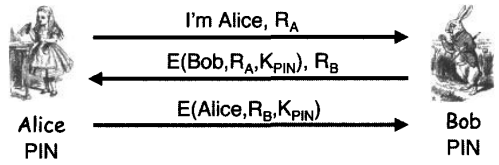
b. Can Trudy determine the session key that Alice and Bob will use?

26. In the text, it is claimed that the protocol in Figure 9.18 is secure, while the similar protocol in Figure 9.24 is not. Why does the attack on the latter protocol not succeed against the former?

27. A timestamp-based protocol may be subject to a replay attack, provided that Trudy can act within the clock skew. Reducing the acceptable clock skew might make the attack more difficult, but it will not prevent the attack unless the skew is zero, which is impractical. Assuming a non-zero clock skew, what can Bob, the server, do to prevent attacks based on the clock skew?

28. Modify the identify friend or foe (IFF) protocol discussed at the beginning of the chapter so that it's no longer susceptible to the MiG-in-the-middle attack.

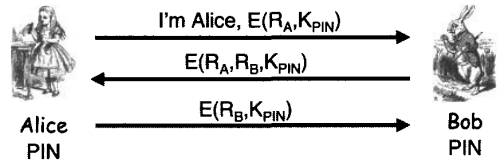
29. Consider the authentication protocol below, which is based on knowledge of a shared 4-digit PIN number. Here, $K_{\text{PIN}} = h(\text{PIN}, R_A, R_B)$.



a. Suppose that Trudy passively observes one iteration of the protocol. Can she determine the 4-digit PIN number? Justify your answer.

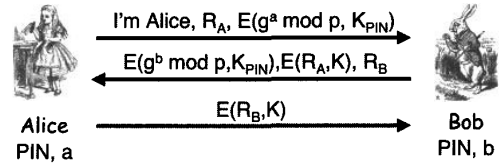
b. Suppose that the PIN number is replaced by a 256-bit shared symmetric key. Is the protocol secure? Why or why not?

30. Consider the authentication protocol below, which is based on knowledge of a shared 4-digit PIN number. Here, $K_{\text{PIN}} = h(\text{PIN})$.



Suppose that Trudy passively observes one iteration of the protocol. Can she then determine the 4-digit PIN? Justify your answer.

31. Consider the authentication protocol below, which is based on knowledge of a shared 4-digit PIN number and uses Diffie-Hellman. Here, $K_{\text{PIN}} = h(\text{PIN})$ and $K = g^{ab} \bmod p$.



a. Suppose that Trudy passively observes one iteration of the protocol. Can she then determine the 4-digit PIN number? Justify your answer.

b. Suppose that Trudy can actively attack the protocol. Can she determine the 4-digit PIN? Explain.

32. Describe a way to provide perfect forward secrecy that does not use Diffie-Hellman.

33. Can you achieve an effect similar to perfect forward secrecy (as described in this chapter) using only symmetric key cryptography? If so, give such a protocol and, if not, why not?

34. Design a zero knowledge protocol analogy that uses Bob's Cave and only requires one iteration for Bob to determine with certainty whether or not Alice knows the secret phrase.

35. The analogy between Bob's Cave and the Fiat-Shamir protocol is not entirely accurate. In the Fiat-Shamir protocol, Bob knows which value of e will force Alice to use the secret value S, assuming Alice follows the protocol. That is, if Bob chooses $e = 1$, then Alice must use the secret value S to construct the correct response in message three, but if Bob chooses $e = 0$, then Alice does not use S. As noted in the text, Bob must choose e at random to prevent Trudy from breaking the protocol. In the Bob's Cave analogy, Bob does not know whether Alice was required to use the secret phrase or not (again, assuming that Alice follows the protocol).

a. Modify the cave analogy so that Bob knows whether Alice used the secret phrase or not, assuming that Bob is not allowed to see which side Alice actually chooses. Bob's New-and-Improved Cave protocol must still resist an attack by someone who does not know the secret phrase.

b. Does your new cave analogy differ from the Fiat-Shamir protocol in any significant way?

36. Suppose that in the Fiat-Shamir protocol in Figure 9.32 we have $N = 63$ and $v = 43$. Recall that Bob accepts an iteration of the protocol if he verifies that $y^2 = x \cdot v^e \bmod N$.

a. In the first iteration of the protocol, Alice sends $x = 37$ in message one, Bob sends $e = 1$ in message two, and Alice sends $y = 4$ in message three. Does Bob accept this iteration of the protocol? Why or why not?

b. In the second iteration of the protocol, Alice sends $x = 37$, Bob sends $e = 0$, and Alice sends $y = 10$. Does Bob accept this iteration of the protocol? Why or why not?

c. Find Alice's secret value S. Hint: $10^{-1} = 19 \bmod 63$.

37. Suppose that in the Fiat-Shamir protocol in Figure 9.32 we have $N = 77$ and $v = 53$.

a. Suppose that Alice sends $x = 15$ in message one, Bob sends $e = 1$ in message two, and Alice sends $y = 5$ in message three. Show that Bob accepts this iteration of the protocol.

b. Suppose Trudy knows in advance that Bob will select $e = 1$ in message two. If Trudy selects $r = 10$, what can she send for x in message one and y in message three so that Bob accepts this iteration of the protocol? Using your answer, show that Bob actually accepts this iteration. Hint: $53^{-1} = 16 \bmod 77$.

38. Suppose that in the Fiat-Shamir protocol in Figure 9.32 we have $N = 55$ and Alice's secret is $S = 9$.

a. What is v?

b. If Alice chooses $r = 10$, what does Alice send in the first message?

c. Suppose Alice chooses $r = 10$ and Bob sends $e = 0$ in message two. What does Alice send in the third message?

d. Suppose Alice chooses $r = 10$ and Bob sends $e = 1$ in message two. What does Alice send in the third message?

39. Consider the Fiat-Shamir protocol in Figure 9.32. Suppose that the public values are $N = 55$ and $v = 5$. Suppose Alice sends $x = 4$ in the first message, Bob sends $e = 1$ in the second message, and Alice sends $y = 30$ in message three. Show that Bob will verify Alice's response in this case. Can you find Alice's secret S?

40. In the Fiat-Shamir protocol in Figure 9.32, suppose that Alice gets lazy and she decides to use the same "random" r for each iteration.

a. Show that Bob can determine Alice's secret S.

b. Why is this a security concern?

41. Suppose that in the Fiat-Shamir protocol, as illustrated in Figure 9.32, we have $N = 27{,}331$ and $v = 7339$.

a. In the first iteration, Alice sends $x = 21{,}684$ in message one, Bob sends $e = 0$ in message two, and Alice sends $y = 657$ in the third message. Show that Bob verifies Alice's response in this case.

b. At the next iteration, Alice again sends $x = 21{,}684$ in message one, but Bob sends $e = 1$ in message two, and Alice responds with $y = 26{,}938$ in message three. Show that Bob again verifies Alice's response.

c. Determine Alice's secret S. Hint: $657^{-1} = 208 \bmod 27{,}331$.

Chapter 10

Real-World Security Protocols

The wire protocol guys don't worry about security because that's really a network protocol problem. The network protocol guys don't worry about it because, really, it's an application problem. The application guys don't worry about it because, after all, they can just use the IP address and trust the network.
— Marcus J. Ranum

In the real world, nothing happens at the right place at the right time. It is the job of journalists and historians to correct that.
— Mark Twain

10.1 Introduction

In this chapter, we'll discuss several widely used real-world security protocols. First on the agenda is the Secure Shell, or SSH, which is used for a variety of purposes. Next, we consider the Secure Socket Layer, or SSL, which is currently the most widely used security protocol for Internet transactions. The third protocol that we'll consider in detail is IPSec, which is a complex protocol with some significant security issues. Then we will discuss Kerberos, a popular authentication protocol based on symmetric key cryptography and timestamps.

We conclude the chapter with two wireless protocols, WEP and GSM. WEP is a seriously flawed security protocols, and we'll consider several well-known attacks. The final protocol we'll cover is GSM, which is used to secure mobile communications. The GSM protocol is provides an interesting case study due to the large number and wide variety of known attacks.

10.2 SSH

The Secure Shell, SSH, creates a secure tunnel which can be used to secure otherwise insecure commands. For example, in UNIX, the `rlogin` command is used for a remote login, that is, to log into a remote machine over a network. Such a login typically requires a password and `rlogin` simply sends the password in the clear, which might be observed by a snooping Trudy. By first establishing an SSH session, any inherently insecure command such as `rlogin` will be secure. That is, an SSH session provides confidentiality and integrity protection, thereby eliminating Trudy's ability to obtain passwords and other confidential information that would otherwise be sent unprotected.

SSH authentication can be based on public keys, digital certificates, or passwords. Here, we give a slightly simplified version of SSH using digital certificates.[1] The other authentication options are covered in various homework problems at the end of this chapter.

SSH is illustrated in Figure 10.1, using the following notation:

$$\begin{aligned}
\text{certificate}_A &= \text{Alice's certificate}\\
\text{certificate}_B &= \text{Bob's certificate}\\
\text{CP} &= \text{crypto proposed}\\
\text{CS} &= \text{crypto selected}\\
H &= h(\text{Alice}, \text{Bob}, \text{CP}, \text{CS}, R_A, R_B, g^a \bmod p, g^b \bmod p, g^{ab} \bmod p)\\
S_B &= [H]_{\text{Bob}}\\
K &= g^{ab} \bmod p\\
S_A &= [H, \text{Alice}, \text{certificate}_A]_{\text{Alice}}
\end{aligned}$$

As usual, h is a cryptographic hash function.

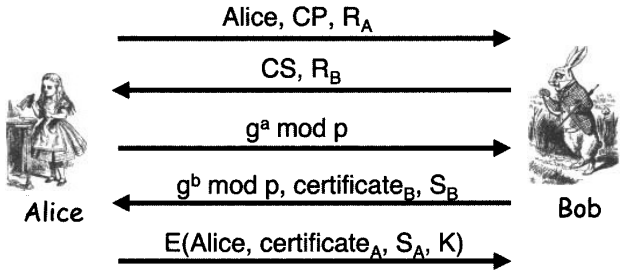


Figure 10.1: Simplified SSH

[1] In our simplified version, a few parameters have been omitted and a couple of bookkeeping messages have been eliminated.

In the first message in Figure 10.1, Alice identifies herself and she sends information regarding the crypto parameters that she prefers (crypto algorithms, key lengths, etc.), along with her nonce, R_A. In message two, Bob selects from Alice's crypto parameters and returns his selections, along with his nonce, R_B. In message three, Alice sends her Diffie-Hellman value, and in message four, Bob responds with his Diffie-Hellman value, his certificate, and S_B, which consists of a signed hash value. At this point, Alice is able to compute the key K, and in the final message, she sends an encrypted block that contains her identity, her certificate, and her signed value S_A.

In Figure 10.1, the signatures are intended to provide mutual authentication. Note that the nonce R_A is Alice's challenge to Bob, and S_B is Bob's response. That is, the nonce R_A provides replay protection, and only Bob can give the correct response since a signature is required (assuming, of course, that his private key has not been compromised). A similar argument shows that Alice is authenticated in the final message. So, SSH provides mutual authentication. The security of SSH authentication, the security of the key K, and some other quirks of SSH are considered further in the homework problems at the end of this chapter.

10.3 SSL

The mythical "socket layer" lives between the application layer and the transport layer in the Internet protocol stack, as illustrated in Figure 10.2. In practice, SSL most often deals with Web browsing, in which case the application layer protocol is HTTP and the transport layer protocol is TCP.

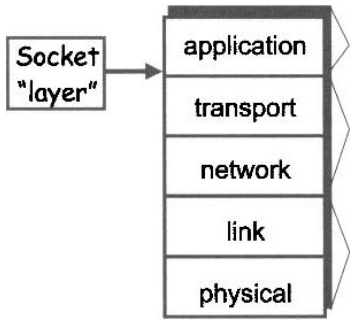


Figure 10.2: Socket Layer

SSL is the protocol of choice for the vast majority of secure transactions over the Internet. For example, suppose that you want to buy a book at

`amazon.com`. Before you provide your credit card information, you want to be sure you are dealing with Amazon, that is, you must authenticate Amazon. Generally, Amazon doesn't care who you are, as long as you have money. As a result, the authentication need not be mutual.

After you are satisfied that you are dealing with Amazon, you will provide private information, such as your credit card number, your address, and so on. You probably want this information protected in transit—in most cases, you want both confidentiality (to protect your privacy) and integrity protection (to assure the transaction is received correctly).

The general idea behind SSL is illustrated in Figure 10.3. In this protocol, Alice (the client) informs Bob (the server) that she wants to conduct a secure transaction. Bob responds with his certificate. Alice then needs to verify the signature on the certificate. Assuming the signature verifies, Alice will be confident that she has Bob's certificate, although she cannot yet be certain that she's talking to Bob. Then Alice will encrypt a symmetric key K_{AB} with Bob's public key and send the encrypted key to Bob. This symmetric key is used to encrypt and integrity protect subsequent communications.

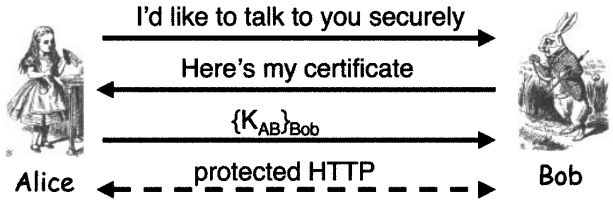


Figure 10.3: Too-Simple Protocol

The protocol in Figure 10.3 is not useful as it stands. For one thing, Bob is not explicitly authenticated and the only way Alice could possibly know she is talking to Bob is by checking to see that the encrypted data decrypts correctly. This is not a desirable situation in any security protocol. Also note that Alice is not authenticated to Bob at all, but in most cases, this is reasonable for transactions on the Internet.

In Figure 10.4, we've given a reasonably complete view of the basic SSL protocol. In this protocol,

$$
\begin{aligned}
S &= \text{the pre-master secret} \\
K &= h(S, R_A, R_B) \\
\text{msgs} &= \text{shorthand for "all previous messages"} \\
\text{CLNT} &= \text{literal string} \\
\text{SRVR} &= \text{literal string}
\end{aligned}
$$

where h is a secure hash function. The actual SSL protocol is more complex than what appears in Figure 10.4 but this simplified version is sufficient for our purposes. The complete SSL specification can be found at [271].

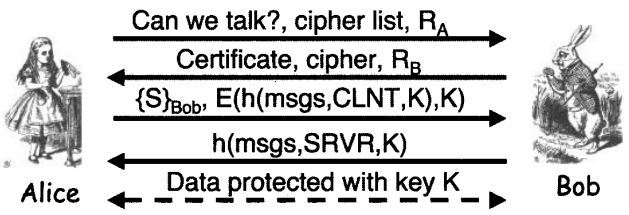


Figure 10.4: Simplified SSL

Next, we briefly discuss each message in the simplified SSL protocol given in Figure 10.4. In the first message, Alice informs Bob that she would like to establish an SSL connection, and she passes a list of ciphers that she supports, along with a nonce R_A. In the second message, Bob responds with his certificate, he selects one of the ciphers from the cipher list that Alice sent in message one, and he sends a nonce R_B.

In the third message, Alice sends the so-called pre-master secret S, which she randomly generated, along with a hash that is encrypted with the key K. In this hash, "msgs" includes all previous messages and CLNT is a literal string.[2] The hash is used as an integrity check to verify that the previous messages have been received correctly.

In the fourth message, Bob responds with a similar hash. By computing this hash herself, Alice can verify that Bob received the messages correctly, and she can authenticate Bob, since only Bob could have decrypted S, which is required to generate the key K. At this point, Alice has authenticated Bob, and Alice and Bob have established a shared session key K, which they can use to encrypt and integrity protect subsequent messages.

In reality, more than one key is derived from the hash $h(S, R_A, R_B)$. In fact, the following six quantities are generated from this hash.

- Two encryption keys, one for messages sent from the client to server, and one for messages sent from the server to the client.
- Two integrity keys, used in the same way as the encryption keys.
- Two initialization vectors (IVs), one for the client and one for the server.

[2]In this context, "msg" has nothing to do with the list of ingredients at a Chinese restaurant.

In short, different keys are used in each direction. This could help to prevent certain types of attacks where Trudy tricks Bob into doing something that Alice should have done, or vice versa.

The attentive reader may wonder why $h(\text{msgs}, \text{CLNT}, K)$ is encrypted in messages three and four. In fact, this adds no security, although it does add extra work, so it could be considered a minor flaw in the protocol.

In the SSL protocol of Figure 10.4, Alice, the client, authenticates Bob, the server, but not vice versa. With SSL, it is possible for the server to authenticate the client. If this is desired, Bob sends a "certificate request" in message two. However, this feature is generally not used, particularly in e-commerce situations, since it requires users to have valid certificates. If the server wants to authenticate the client, the server could instead require that the client enter a valid password, in which case the resulting authentication is outside the scope of the SSL protocol.

10.3.1 SSL and the Man-in-the-Middle

Hopefully, SSL prevents the man-in-the-middle, or MiM, attack illustrated in Figure 10.5. But what mechanism in SSL prevents this attack? Recall that Bob's certificate must be signed by a certificate authority. If Trudy sends her own certificate instead of Bob's, the attack will fail when Alice attempts to verify the signature on the certificate. Or, Trudy could make a bogus certificate that says "Bob," keep the private key for herself, and sign the certificate herself. Again, this will not pass muster when Alice tries to verify the signature on "Bob's" certificate (which is really Trudy's certificate). Finally, Trudy could simply send Bob's certificate to Alice, and Alice would verify the signature on this certificate. However, this is not an attack since it would not break the protocol—Alice would authenticate Bob, and Trudy would be left out in the cold.

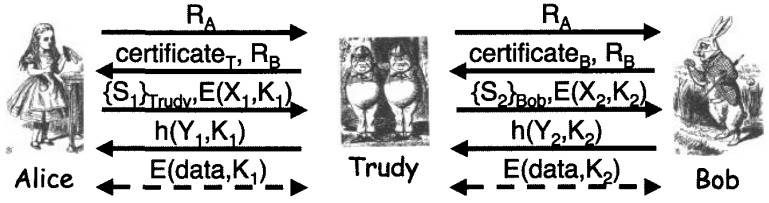


Figure 10.5: Man-in-the-Middle Attack on SSL

However, the real world is not so kind to poor Alice. Typically, SSL is used in a Web browsing session. Then, what happens when Trudy attempts a MiM attack by sending a bogus certificate to Alice? The signature on the

certificate is not valid so the attack should fail. However, Alice does not personally check the signature on the certificate—her browser does. And what does Alice's browser do when it detects a problem with a certificate? As you probably know from experience, the browser provides Alice with a warning. Does Alice heed the warning? If she's like most users, Alice ignores the warning and allows the connection to proceed.[3] Note that when Alice ignores this warning, she's opened the door to the MiM attack in Figure 10.5. Finally, it's important to realize that while this attack is a very real threat, it's not due to a flaw in the SSL protocol. Instead, it's caused by a flaw in human nature, making a patch much more problematic.

10.3.2 SSL Connections

An SSL *session* is established as shown in Figure 10.4. This session establishment protocol is relatively expensive, since public key operations are involved.

SSL was originally developed by Netscape, specifically for use in Web browsing. The application layer protocol for the Web is HTTP, and two versions of it are in common usage, HTTP 1.0 and HTTP 1.1. With version 1.0, it is not uncommon for a Web browser to open multiple parallel connections so as to improve performance. Due to the public key operations, there would be significant overhead if a new SSL session was established for each of these HTTP connections. The designers of SSL were aware of this issue, so they included an efficient protocol for opening new SSL *connections* provided that an SSL session already exists. The idea is simple—after establishing one SSL session, Alice and Bob share a session key K, which can then be used to establish new connections, thereby avoiding expensive public key operations.

The SSL connection protocol appears in Figure 10.6. The protocol is similar to the SSL session establishment protocol, except that the previously established session key K is used instead of the public key operation that are used in the session protocol.

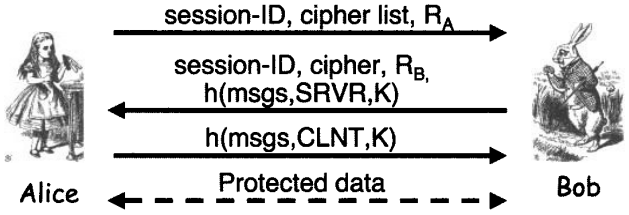


Figure 10.6: SSL Connection Protocol

[3]If possible, Alice would probably disable the warning so that she'd never get this annoying "error" message again.

The bottom line here is that in SSL, one (expensive) session is required, but then we can create any number of (cheap) connections. This is a useful feature that was designed to improve the performance of the protocol when used with HTTP 1.1.

10.3.3 SSL Versus IPSec

In the next section, we'll discuss IPSec, which is short for Internet Protocol Security. The purpose of IPSec is similar to that of SSL, namely, security over the network. However, the implementation of the two protocols is very different. For one thing, SSL is relatively simple, while IPSec is relatively complex.

It might seem logical to discuss IPSec in detail before contrasting it with SSL. However, we might get so lost in the weeds with IPSec that we'd completely lose sight of SSL. So instead of waiting until after we discuss IPSec to contrast the two protocols, we'll do so beforehand. You might consider this a partial preview of IPSec.

The most obvious difference between SSL and IPSec is that the two protocols operate at different layers of the protocol stack. SSL (and its twin,[4] the IEEE standard known as TLS), both live at the socket layer. As a result, SSL resides in user space. IPSec, on the other hand, lives at the network layer and is therefore not directly accessible from user space—it's in the domain of the operating system. When viewed from a high level, this is the fundamental distinction between SSL and IPSec.

Both SSL and IPSec provide encryption, integrity protection, and authentication. SSL is relatively simple and well designed, whereas IPSec is complex and, as a result, includes some significant flaws.

Since IPSec is part of the OS, it must be built-in at that level. In contrast, SSL is part of user space, so it requires nothing special of the OS. IPSec also requires no changes to applications, since all of the security magically happens at the network layer. On the other hand, developers have to make a conscious decision to use SSL.

SSL was built for Web application early on, and its primary use remains secure Web transactions. IPSec is often used to secure a virtual private network, or VPN, an application that creates a secure tunnel between the endpoints. Also, IPSec is required in IP version 6 (IPv6), so if IPv6 ever takes over the world, IPSec will be ubiquitous.

There is, understandably, a reluctance to retrofit applications for SSL. There is, also understandably, a reluctance to use IPSec due to its complexity (which creates some challenging implementation issues). The net result is that the Net is less secure than it should be.

[4]They are fraternal twins, not identical twins.

10.4 IPSec

Figure 10.7 illustrates the primary logical difference between SSL and IPSec, that is, one lives at the socket layer (SSL), while the other resides at the network layer (IPSec). As mentioned above, the major advantage of IPSec is that it's essentially transparent to applications. However, IPSec is a complex protocol, which can perhaps best be described as over-engineered.

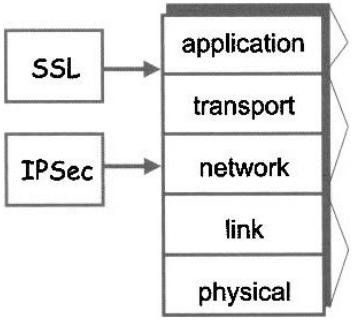


Figure 10.7: IPSec

IPSec has many dubious features, which makes implementation difficult. Also, IPSec has some flaws, probably as a direct result of its complexity. In addition, there are interoperability issues, due to the complexity of the IPSec specification, which seems to run contrary to the point of having a standard. Another complicating factor is that the IPSec specification is split into three pieces, to be found in RFC 2407 [237], RFC 2408 [197], and RFC 2409 [140], and these RFCs were written by disjoint sets of authors using different terminology.

The two main parts to IPSec are

- The Internet Key Exchange, or IKE, which provides for mutual authentication and a session key. There are two phases of IKE, which are analogous to SSL sessions and connections.
- The Encapsulating Security Payload and Authentication Header, or ESP/AH, which together make up the second part of IPSec. ESP[5] provides encryption and integrity protection to IP packets, whereas AH provides integrity only.

Technically, IKE is a standalone protocol that could live a life separate from ESP/AH. However, since IKE's only application in the real world seems to

[5]Contrary to what you are thinking, this protocol cannot read your mind.

be in conjunction with IPSec, we lump them together under the name IPSec. The comment about IPSec being over-engineered applies primarily to IKE. The developers of IKE apparently thought they were creating the Swiss army knife of security protocols—a protocol that would be used to solve every conceivable authentication problem. This explains the multitude of options and features built into IKE. However, since IKE is only used with IPSec, any features or options that are not directly relevant to IPSec are simply extraneous.

First, we'll consider IKE, then ESP/AH. IKE, the more complex of the two, consists of two phases—cleverly called Phase 1 and Phase 2. Phase 1 is the more complex of the two. In Phase 1, a so-called IKE security association, or IKE-SA, is established, while in Phase 2, an IPSec security association, IPSec-SA, is established. Phase 1 corresponds to an SSL session, whereas Phase 2 is comparable to an SSL connection. In IKE, both Phase 1 and Phase 2 must occur before we can do ESP/AH.

Recall that SSL connections serve a specific and useful purpose—they make SSL more efficient when HTTP 1.0 is used. But, unlike SSL, in IPSec there is no obvious need for two phases. And if multiple Phase 2s do not occur (and they typically do not), then it would be more efficient to just require Phase 1 with no Phase 2. However, this is not an option. Apparently, the developers of IKE believed that their protocol was so self-evidently wonderful that users would want to do multiple Phase 2s (one for IPSec, another for something else, another for some other something else, and so on). This is our first example of over-engineering in IPSec, and it won't be the last.

In IKE Phase 1, there are four different key options:

- Public key encryption (original version)
- Public key encryption (improved version)
- Digital signature
- Symmetric key

For each of these key options there is a main mode and an aggressive mode. As a result, there are a staggering eight different versions of IKE Phase 1. Do you need any more evidence that IPSec is over-engineered?

You may be wondering why there are public key encryption and digital signature options in Phase 1. Surprisingly, the answer is not over-engineering. Alice always knows her own private key, but she may not know Bob's public key. With the signature version of IKE Phase 1, Alice does not need to have Bob's public key in hand to start the protocol. In any protocol that uses public key crypto, Alice will need Bob's public key to complete the protocol, but in the signature mode, she can simultaneously begin the protocol and search for Bob's public key. In contrast, in the public key encryption modes,

Alice needs Bob's public key immediately, so she must first find Bob's key before she can begin the protocol. So, there could be an efficiency gain with the signature option.

We'll discuss six of the eight Phase 1 variants, namely, digital signatures (main and aggressive modes), symmetric key (main and aggressive modes), and public key encryption (main and aggressive). We'll consider the original version of public key encryption, since it's slightly simpler, although less efficient, than the improved version.

Each of the Phase 1 variants use an ephemeral Diffie-Hellman key exchange to establish a session key. The benefit of this approach is that it provides perfect forward secrecy (PFS). For each of the variants we discuss, we'll use the following Diffie-Hellman notation. Let a be Alice's (ephemeral) Diffie-Hellman exponent and let b be Bob's (ephemeral) Diffie-Hellman exponent. Let g be the generator and p the prime. Recall that p and g are public.

10.4.1 IKE Phase 1: Digital Signature

The first Phase 1 variant that we'll consider is digital signature, main mode. This six message protocol is illustrated in Figure 10.8, where

$$\begin{aligned}
\text{CP} &= \text{crypto proposed}\\
\text{CS} &= \text{crypto selected}\\
\text{IC} &= \text{initiator cookie}\\
\text{RC} &= \text{responder cookie}\\
K &= h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B)\\
\text{SKEYID} &= h(R_A, R_B, g^{ab} \bmod p)\\
\text{proof}_A &= [h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{“Alice”})]_{\text{Alice}}
\end{aligned}$$

Here, h is a hash function and proof_B is analogous to proof_A.

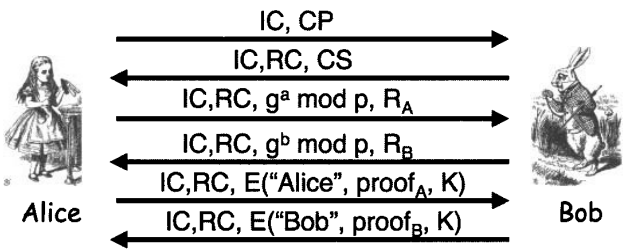


Figure 10.8: Digital Signature, Main Mode

Let's briefly consider each of the six messages that appear in Figure 10.8. In the first message, Alice provides information on the ciphers that she supports and other crypto related information, along with a so-called cookie.[6] In message two, Bob selects from Alice's crypto proposal and sends the cookies, which serve as an identifier for the remainder of the messages in the protocol. The third message includes a nonce and Alice's Diffie-Hellman value. Bob responds similarly in message four, providing a nonce and his Diffie-Hellman value. In the final two messages, Alice and Bob authenticate each other using digital signatures.

Recall that an attacker, Trudy, is said to be passive if she can only observe messages sent between Alice and Bob. In contrast, if Trudy is an active attacker, she can also insert, delete, alter, and replay messages. For the protocol in Figure 10.8, a passive attacker cannot discern Alice or Bob's identity. So this protocol provides anonymity, at least with respect to passive attacks. Does this protocol also provide anonymity in the case of an active attack? This question is considered in Problem 27, which means that the answer is not to be found here.

Each key option has a main mode and an aggressive mode. The main modes are supposed to provide anonymity, while the aggressive modes are not. Anonymity comes at a price—aggressive mode only requires three messages, as opposed to six messages for main mode.

The aggressive mode version of the digital signature key option appears in Figure 10.9. Note that there is no attempt to hide the identities of Alice or Bob, which simplifies the protocol considerably. The notation in Figure 10.9 is the same as that used in Figure 10.8.

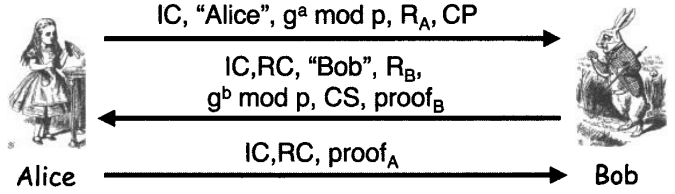


Figure 10.9: Digital Signature, Aggressive Mode

One subtle difference between digital signature main and aggressive modes is that in main mode it is possible to negotiate the values of g and p as part of the "crypto proposed" and "crypto accepted" messages. However, this is not the case in aggressive mode, since the Diffie-Hellman value $g^a \bmod p$ is sent in the first message.

[6]Not to be confused with Web cookies or chocolate chip cookies. We have more to say about these IPSec cookies in Section 10.4.4, below.

As per the appropriate RFCs, for each key option main mode MUST be implemented, while aggressive mode SHOULD be implemented. In [162], the authors interpret this to mean that if aggressive mode is not implemented, "you should feel guilty about it."

10.4.2 IKE Phase 1: Symmetric Key

The next version of Phase 1 that we'll consider is the symmetric key option—both main mode and aggressive mode. As above, the main mode is a six-message protocol, where the format is formally the same as in Figure 10.8, above, except that the notation is interpreted as follows.

$$\begin{aligned} K_{AB} &= \text{symmetric key shared in advance} \\ K &= h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B, K_{AB}) \\ \text{SKEYID} &= h(K, g^{ab} \bmod p) \\ \text{proof}_A &= h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{Alice}) \end{aligned}$$

Again, the purported advantage of the complex six-message main mode over the corresponding aggressive mode is that main mode is supposed to provide anonymity. But there is a Catch-22 in this main mode. Note that in message five Alice sends her identity, encrypted with key K. But Bob has to use the key K_{AB} to determine K. So Bob has to know to use the key K_{AB} *before* he knows that he's talking to Alice. However, Bob is a busy server who deals with lots of users (Alice, Charlie, Dave, ...). How can Bob possibly know that he is supposed to use the key he shares with Alice before he knows he's talking to Alice? The answer is that he cannot, at least not based on any information available within the protocol itself.

The developers of IPSec recognized this snafu. And their solution? Bob is to rely on the IP address to determine which key to use. So, Bob must use the IP address of incoming packets to determine who he's talking to before he knows who he's talking to (or something like that...). The bottom line is that Alice's IP address acts as her identity.

There are a couple of problems with this approach. First, Alice must have a static IP address—this mode fails if Alice's IP address changes. A more fundamental issue is that the protocol is complex and uses six messages, presumably to hide identities. But the protocol fails to hide identities, unless you consider a static IP address to be secret. So it would seem pointless to use symmetric key main mode instead of the simpler and more efficient aggressive mode, which we describe next.[7]

IPSec symmetric key aggressive mode follows the same format as the digital signature aggressive mode in Figure 10.9, with the key and signature

[7]Of course, main mode MUST be implemented, while aggressive mode SHOULD be implemented. Go figure.

computed as in symmetric key main mode. As with the digital signature variant, the main difference from main mode is that aggressive mode does not attempt to hide identities. Since symmetric key main mode also fails to effectively hide Alice's identity, this is not a serious limitation of aggressive mode in this case.

10.4.3 IKE Phase 1: Public Key Encryption

Next, we'll consider the public key encryption version of IKE Phase 1, both main and aggressive modes. We've already seen the digital signature versions. In the main mode of the encryption version, Alice must know Bob's public key in advance and vice versa. Although it would be possible to exchange certificates, that would reveal the identities of Alice and Bob, defeating the primary advantage of main mode. So an assumption here is that Alice and Bob have access to each other's certificates, without sending them over the network.

The public key encryption main mode protocol is given in Figure 10.10, where the notation is as in the previous modes, except

$$\begin{aligned} K &= h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B) \\ \text{SKEYID} &= h(R_A, R_B, g^{ab} \bmod p) \\ \text{proof}_A &= h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{“Alice”}) \end{aligned}$$

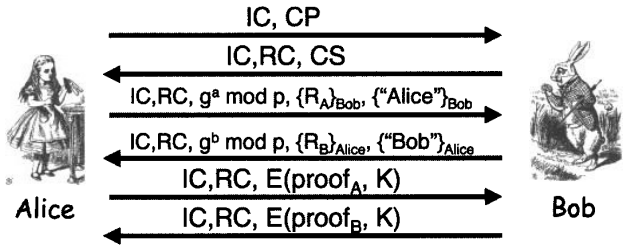


Figure 10.10: Public Key Encryption, Main Mode

Public key encryption, aggressive mode, appears in Figure 10.11, where the notation is similar to main mode. Interestingly, unlike the other aggressive modes, public key encryption aggressive mode allows Alice and Bob to remain anonymous. Since this is the case, is there any possible advantage of main mode over aggressive mode? The answer is yes, but it's a minor issue (see Problem 25 at the end of the chapter).

There is an interesting security quirk that arises in the public key encryption versions—both main and aggressive modes. For simplicity, let's consider

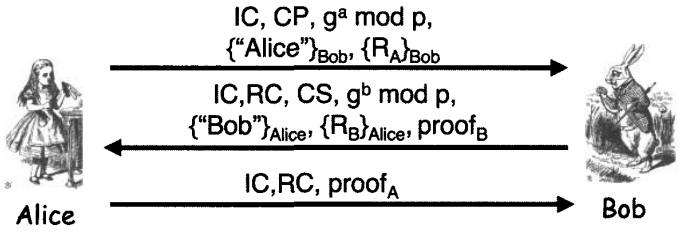


Figure 10.11: Public Key Encryption, Aggressive Mode

aggressive mode. Suppose Trudy generates Diffie-Hellman exponents a and b and random nonces R_A and R_B. Then Trudy can compute all of the remaining quantities that appear in the protocol in Figure 10.11, namely, $g^{ab} \bmod p$, K, SKEYID, proof_A, and proof_B. The reason that Trudy can do this is because the public keys of Alice and Bob are public.

Why would Trudy go to the trouble of generating all of these values? Once Trudy has done so, she can create an entire conversation that appears to be a valid IPSec transaction between Alice and Bob, as indicated in Figure 10.12. Amazingly, this conversation appears to be valid to any observer, including Alice and/or Bob!

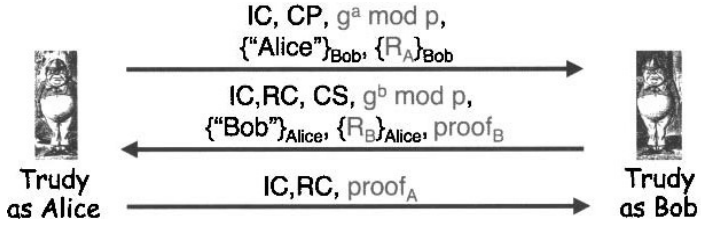


Figure 10.12: Trudy Making Mischief

Note that in Figure 10.12, Trudy is playing the roles of both Alice and Bob. Here, Trudy does not convince Bob that she's Alice, she does not convince Alice that she's Bob, nor does she determine a session key used by Alice and Bob. So, this is a very different kind of attack than we have previously seen. Or maybe it's not an attack at all.

But surely, the fact that Trudy can create a fake conversation that appears to be a legitimate connection between Alice and Bob is a security flaw. Surprisingly, in this mode of IPSec it is considered a security feature, which goes by the name of *plausible deniability*. A protocol that includes plausible deniability allows Alice and Bob to deny that a conversation ever took place,

since anyone could have faked the whole thing. In some situations, this could be a desirable feature. On the other hand, in some situations it might be a problem. For example, if Alice makes a purchase from Bob, she could later repudiate it, unless Bob also required a digital signature from Alice.

10.4.4 IPSec Cookies

The cookies IC and RC that appear in the IPSec protocols above are officially known as "anti-clogging tokens" in the relevant RFCs. These IPSec cookies have no relation to Web cookies, which are used to maintain state across HTTP sessions. Instead, the stated purpose of IPSec cookies is to make denial of service, or DoS, attacks more difficult.

Consider TCP SYN flooding, which is a prototypical DoS attack. Each TCP SYN request causes the server to do a little work (create a SEQ number, for example) and to keep some amount of state. That is, the server must remember the so-called half-open connection so that it can complete the connection when the corresponding ACK arrives in the third step of the three-way handshake. It is this keeping of state that an attacker can exploit to create a DoS. If the attacker bombards a server with a large number of SYN packets and never completes the resulting half-open connections, the server will eventually deplete its resources. When this occurs, the server cannot handle legitimate SYN requests and a DoS results.

To reduce the threat of DoS in IPSec, the server Bob would like to remain stateless as much as possible. The IPSec cookies are supposed to help Bob remain stateless. However, they clearly fail to achieve their design goal. In each of the main mode protocols, Bob must remember the crypto proposal, CP, from message one, since it is required in message six when Bob computes proof_B. Consequently, Bob must keep state beginning with the first message. The IPSec cookies therefore offer no significant DoS protection.

10.4.5 IKE Phase 1 Summary

Regardless of which of the eight versions is used, successful completion of IKE Phase 1 results in mutual authentication and a shared session key. This is known an an IKE Security Association (IKE-SA).

IKE Phase 1 is computationally expensive in any of the public key modes, and the main modes also require six messages. Developers of IKE assumed that it would be used for lots of things, not just IPSec (which explains the over-engineering). So they included an inexpensive Phase 2, which must be used after the IKE-SA has been established in Phase 1. That is, a separate Phase 2 is required for each different application that will make use of the IKE-SA. However, if IKE is only used for IPSec (as is the case in practice), the potential efficiency provided by multiple Phase 2s is not realized.

IKE Phase 2 is used to establish a so-called IPSec Security Association, or IPSec-SA. The IKE Phase 1 is more or less equivalent to establishing an SSL session, whereas IKE Phase 2 is more or less equivalent to establishing an SSL connection. Again, the designers of IPSec wanted to make it as flexible as possible, since they assumed it would be used for lots of things other than IPSec. In fact, IKE could conceivably be used for lots of things other than IPSec, however, in practice, it's not.

10.4.6 IKE Phase 2

IKE Phase 2 is mercifully simple—at least in comparison to Phase 1. Before IKE Phase 2 can occur, IKE Phase 1 must be completed, in which case a shared session key K, the IPSec cookies, IC, RC, and the IKE-SA have all been established and are known to Alice and Bob. Given that this is the case, the IKE Phase 2 protocol appears in Figure 10.13, where the following holds true.

- The crypto proposal includes ESP or AH (discussed below). This is where Alice and Bob decide whether to use ESP or AH.
- SA is an identifier for the IKE-SA established in Phase 1.
- The hashes numbered 1, 2, and 3 depend on SKEYID, R_A, R_B, and the IKE SA from Phase 1.
- The keys are derived from KEYMAT = $h(\text{SKEYID}, R_A, R_B, \text{junk})$, where the "junk" is known to all (including an attacker).
- The value of SKEYID depends on the Phase 1 key method.
- Optionally, PFS can be employed, using an ephemeral Diffie-Hellman exchange.

Note that R_A and R_B in Figure 10.13 are not the same as those from IKE Phase 1. As a result, the keys generated in each Phase 2 differ from the Phase 1 key and from each other.

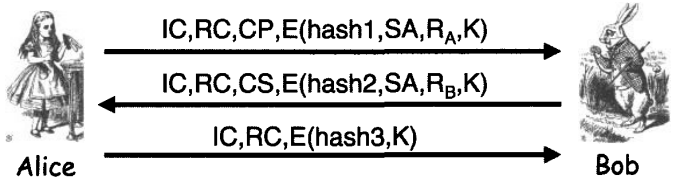


Figure 10.13: IKE Phase 2

After completing IKE Phase 1, we have established an IKE-SA, and after completing IKE Phase 2, we have established an IPSec-SA. After Phase 2, both Alice and Bob have been authenticated and they have a shared session key for use in the current connection.

Recall that in SSL, once we completed mutual authentication and had established a session key, we were done. Since SSL deals with application layer data, we simply encrypt and integrity protect in a standard way. In SSL, the network is transparent to Alice and Bob because SSL lives at the socket layer—which is really part of the application layer. This is one advantage to dealing with application layer data.

In IPSec, protecting the data is not so straightforward. Assuming IPSec authentication succeeds and we establish a session key, then we need to protect IP datagrams. The complication here is that protection must occur at the network layer. But before we discuss this issue in detail, we need to consider IP datagrams from the perspective of IPSec.

10.4.7 IPSec and IP Datagrams

An IP datagram consists of a header and data. The IP header is illustrated in the Appendix in Figure A-5. If the `options` field is empty (as it usually is), then the IP header consists of 20 bytes. For the purposes of IPSec, one important point is that routers must see the destination address in the IP header so that they can route the packet. Most other header fields are also used in conjunction with routing the packet. Since the routers do not have access to the session key, we cannot encrypt the IP header.

A second crucial point is that some of the fields in the IP header change as the packet is forwarded. For example, the TTL field—which contains the number of hops remaining before the packet dies—is decremented by each router that handles the packet. Since the session key is not known to the routers, any header fields that change cannot be integrity protected. In IPSec-speak, the header fields that can change are known as mutable fields.

Next, we look inside an IP datagram. Consider, for example, a Web browsing session. The application layer protocol for such traffic is HTTP, and the transport layer protocol is TCP. In this case, IP encapsulates a TCP packet, which encapsulates an HTTP packet as is illustrated in Figure 10.14. The point here is that, from the perspective of IP (and hence, IPSec), the data includes more than application layer data. In this example, the "data" includes the TCP and HTTP headers, as well as the application layer data. We'll see why this is relevant below.

As previously mentioned, IPSec uses either ESP or AH to protect an IP datagram. Depending on which is selected, an ESP header or an AH header is included in an IPSec-protected datagram. This header tells the recipient to treat this as an ESP or AH packet, not as a standard IP datagram.

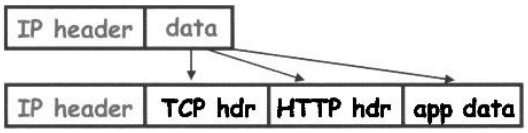


Figure 10.14: IP Datagram

10.4.8 Transport and Tunnel Modes

Independent of whether ESP or AH is used, IPSec employs either *transport mode* or *tunnel mode*. In transport mode, as illustrated in Figure 10.15, the new ESP/AH header is sandwiched between the IP header and the data. Transport mode is more efficient since it adds a minimal amount of additional header information. Note that in transport mode the original IP header remains intact. The downside of transport mode is that a passive attacker can see the headers. So, if Trudy observes an IPSec protected conversation between Alice and Bob where transport mode is used, the headers will reveal that Alice and Bob are communicating.[8]

Transport mode is designed for host-to-host communication, that is, when Alice and Bob are communicating directly with each other using IPSec. This is illustrated in Figure 10.16.

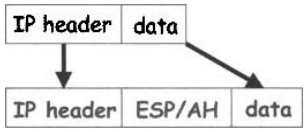


Figure 10.15: IPSec Transport Mode

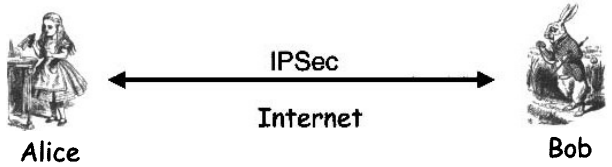


Figure 10.16: IPSec from Host-to-Host

In tunnel mode, as illustrated in Figure 10.17, the entire IP packet is encapsulated in a new IP packet. One advantage of this approach is that the

[8]Recall that we cannot encrypt the header.

original IP header is no longer visible to an attacker—assuming the packet is encrypted. However, if Alice and Bob are communicating directly with each other, the new IP header will be the same as the encapsulated IP header, so hiding the original header would be pointless. However, IPSec is often used from firewall to firewall, not from host to host. That is, Alice's firewall and Bob's firewall communicate using IPSec, not Alice and Bob directly. Suppose IPSec is being used from firewall to firewall. Using tunnel mode, the new IP header will only reveal that the packet is being sent between Alice's firewall and Bob's firewall. So, if the packet is encrypted, Trudy would know that Alice's and Bob's firewalls are communicating, but she would not know which specific hosts behind the firewalls are communicating.

Tunnel mode was designed for firewall-to-firewall communication. Again, when tunnel mode is used from firewall to firewall—as illustrated in Figure 10.18—Trudy does not know which hosts are communicating. The disadvantage of tunnel mode is the overhead of an additional IP header.

Figure 10.17: IPSec Tunnel Mode

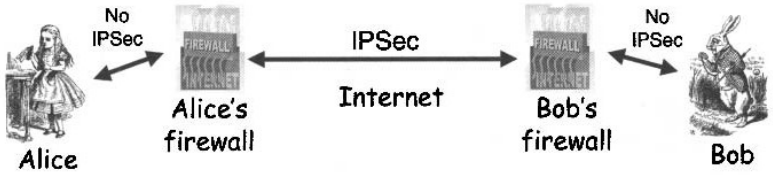


Figure 10.18: IPSec from Firewall to Firewall

Technically, transport mode is not necessary, since we could encapsulate the original IP packet in a new IPSec packet, even in the host-to-host case. For firewall-to-firewall protected traffic, tunnel mode is necessary, as we must preserve the original IP header so that the destination firewall can route the packet to the destination host. But transport mode is more efficient, which makes it preferable when traffic is protected from host to host.

10.4.9 ESP and AH

Once we've decided whether to use transport mode or tunnel mode, then we must (finally) consider the type of protection we actually want to apply to

the IP datagrams. The choices are confidentiality, integrity, or both. But we also must consider the protection, if any, to apply to the header. In IPSec, the only choices are AH and ESP. So, what protection options do each of these provide?

AH, the Authentication Header, provides integrity only, that is, AH provides no encryption. The AH integrity protection applies to everything beyond the IP header and some fields of the header. As previously mentioned, not all fields of the IP header can be integrity protected (TTL, for example). AH classifies IP header fields as mutable or immutable, and it applies its integrity protection to all of the immutable fields.

In ESP, the Encapsulating Security Payload, both integrity and confidentiality are required. Both the confidentiality and integrity protection are applied to everything beyond the IP header, that is, the "data" from the perspective of IP. No protection is applied to the IP header

Encryption is required in ESP. However, there is a trick whereby ESP can be used for integrity only. In ESP, Alice and Bob negotiate the cipher that they will use. One of the ciphers that MUST be supported is the NULL cipher, described in RFC 2410 [123]. Here are some excerpts from this unusual RFC.

- NULL encryption is a block cipher, the origins of which appear to be lost in antiquity.
- Despite rumors, there is no evidence that NSA suppressed publication of this algorithm.
- Evidence suggests it was developed in Roman times as an exportable version of Caesar's cipher.
- NULL encryption can make use of keys of varying length.
- No IV is required.
- NULL encryption is defined by $\text{Null}(P, K) = P$ for any plaintext P and any key K.

This RFC proves that security people are strange.[9]

In ESP, if the NULL cipher is selected then no encryption is applied, but the data is integrity protected. This case looks suspiciously similar to AH. So, why does AH exist?

There are three reasons given to justify the existence of AH. As previously noted, the IP header can't be encrypted since routers must see the header to route packets. But AH does provide integrity protection to the immutable

[9] As if you didn't already know that.

fields in the IP header, whereas ESP provides no protection to the header. That is, AH provides slightly more integrity protection than ESP/NULL.

A second reason for the existence of AH is that ESP encrypts everything beyond the IP header, provided a non-NULL cipher is selected. If ESP is used and the packet is encrypted, a firewall can't look inside the packet to, for example, examine the TCP header. Perhaps surprisingly, ESP with NULL encryption doesn't solve this problem. When the firewall sees the ESP header, it will know that ESP is being used. However, the header does not tell the firewall that the NULL cipher is used—that was negotiated between Alice and Bob and is not included in the header. So, when a firewall sees that ESP is used, it has no way to know whether the TCP header is encrypted or not. In contrast, when a firewall sees that AH is used, it knows that nothing is encrypted.

Neither of these reasons for the existence of AH is particularly persuasive. The designers of AH/ESP could have made minor modifications to the protocol so that ESP alone could overcome these drawbacks. But there is a more convincing reason given for the existence of AH. At one meeting where the IPSec standard was being developed, "someone from Microsoft gave an impassioned speech about how AH was useless ..." and "... everyone in the room looked around and said, Hmm. He's right, and we hate AH also, but if it annoys Microsoft let's leave it in since we hate Microsoft more than we hate AH" [162]. So now you know the rest of the story.

10.5 Kerberos

In Greek mythology, Kerberos is a three-headed dog that guards the entrance to Hades.[10] In security, Kerberos is a popular authentication protocol that uses symmetric key cryptography and timestamps. Kerberos originated at MIT and is based on work by Needham and Schroeder [217]. Whereas SSL and IPSec are designed for the Internet, Kerberos is designed for a smaller scale, such as on a local area network (LAN) or within a corporation.

Suppose we have N users, where each pair needs to be able to authenticate each other. If our authentication protocol is based on public key cryptography, then each user requires a public-private key pair and, consequently, N key pairs are needed. On the other hand, if our authentication protocol is based on symmetric keys, it would appear that each pair of users must share a symmetric key, in which case $N(N-1)/2 \approx N^2$ keys are required. Consequently, authentication based on symmetric keys doesn't scale. However, by relying on a Trusted Third Party (TTP), Kerberos only requires N symmetric keys for N users. Users do not share keys with each other. Instead each user shares one key with the KDC, that is, Alice and the KDC share K_A, Bob

[10] The authors of [162] ask, "Wouldn't it make more sense to guard the exit?"

and the KDC share K_B, Carol and the KDC share K_C, and so on. Then, the KDC acts as a go-between that enables any pair of users to communicate securely with each other. The bottom line is that Kerberos uses symmetric keys in a way that does scale.

The Kerberos TTP is a security critical component that must be protected from attack. This is certainly a security issue, but in contrast to a system that uses public keys, no public key infrastructure (PKI) is required.[11] In essence, the Kerberos TTP plays a similar role as a certificate authority in a public key system.

The Kerberos TTP is known as the *key distribution center*, or KDC.[12] Since the KDC acts as a TTP, if it's compromised, the security of the entire system is compromised.

As noted above, the KDC shares a symmetric key K_A with user Alice, and it shares a symmetric key K_B with Bob, and so on. The KDC also has a master key K_{KDC}, which is known only to the KDC. Although it might seem senseless to have a key that only the KDC knows, we'll see that this key plays a critical role. In particular, the key K_{KDC} allows the KDC to remain stateless, which eliminates most denial of service attacks. A stateless KDC is a major security feature of Kerberos.

Kerberos is used for authentication and to establish a session key that can subsequently be used for confidentiality and integrity. In principle, any symmetric cipher can be used with Kerberos. However, in practice, it seems the crypto algorithm of choice is the Data Encryption Standard (DES).

In Kerboros-speak, the KDC issues various types of *tickets*. Understanding these tickets is critical to understanding Kerberos. A ticket contains the keys and other information required to access network resource. One special ticket that the KDC issues is the all-important *ticket-granting ticket*, or TGT. A TGT, which is issued when a user initially logs into the system, acts as the user's credentials. The TGT is then used to obtain (ordinary) tickets that enable access to network resources. The use of TGTs is crucial to the statelessness of Kerberos.

Each TGT contains a session key, the user ID of the user to whom the TGT is issued, and an expiration time. For simplicity, we'll ignore the expiration time, but it's worth noting that TGTs don't last forever. Every TGT is encrypted with the key K_{KDC}. Recall that only the KDC knows the key K_{KDC}. As a result, a TGT can only be read by the KDC.

Why does the KDC encrypt a user's TGT with a key that only the KDC knows and then send the result to the user? The alternative would be for the KDC to maintain a database of which users are logged in, their session keys, etc. That is, the TGT would have to maintain state. In effect, TGTs

[11] As we discussed in Chapter 4, PKI presents a substantial challenge in practice.

[12] The most difficult part about Kerberos is keeping track of all of the acronyms. There are a lot more acronyms to come—we're just getting warmed up.

provides a simple, effective, and secure way to distribute this database to the users. Then when, say, Alice presents her TGT to the KDC, the KDC can decrypt it and, voila, it remembers everything it needs to know about Alice.[13] The role of the TGT will become clear below. For now, just note that TGTs are a clever design feature of Kerberos.

10.5.1 Kerberized Login

To understand Kerberos, let's first consider how a "Kerberized" login works, that is, we'll examine the steps that occur when Alice logs in to a system where Kerberos is used for authentication. As on most systems, Alice first enters her username and password. In Kerberos, Alice's computer then derives the key K_A from Alice's password, where K_A is the key that Alice and the KDC share. Alice's computer uses K_A to obtain Alice's TGT from the KDC. Alice can then use her TGT (i.e., her credentials) to securely access network resources. Once Alice has logged in, all of the security is automatic and takes place behind the scenes, without any additional involvement by Alice.

A Kerberized login is illustrated in Figure 10.19, where the following notation is used.

- The key K_A is derived as $K_A = h(\text{Alice's password})$
- The KDC creates the session key S_A
- Alice's computer uses K_A to obtain S_A and the TGT; then Alice's computer forgets K_A
- $\text{TGT} = E(\text{"Alice"}, S_A; K_{\text{KDC}})$

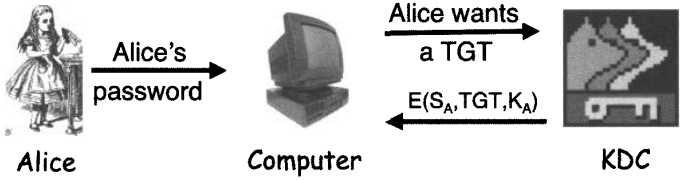


Figure 10.19: Kerberized Login

[13] Your hapless author's ill-fated startup company had a similar situation, i.e., a database of customer security-related information that had to be maintained (assuming the company had ever actually had any customers, that is). Instead of creating a security-critical database, the company chose to encrypt each user's information with a key known only to the company, then distribute this encrypted data to the appropriate user. Users then had to present this encrypted data before they could access any security-related features of the system. This is essentially the same trick used in Kerberos TGTs.

One major advantage to the Kerberized login is that the entire security process (beyond the password entry) is transparent to Alice. The major disadvantage is that the reliance on the security of the KDC is total.

10.5.2 Kerberos Ticket

Once Alice's computer receives its TGT, it can then use the TGT to request access to network resources. For example, suppose that Alice wants to talk to Bob. Then Alice's computer presents its TGT to the KDC, along with an authenticator. The authenticator is an encrypted timestamp that serves to avoid a replay. After the KDC verifies Alice's authenticator, it responds with a "ticket to Bob." Alice's computer then uses this ticket to Bob to securely communicate directly with Bob's computer. Alice's acquisition of the ticket to Bob is illustrated in Figure 10.20, where the following notation is used.

$$\begin{aligned} \text{REQUEST} &= (\text{TGT}, \text{authenticator}) \\ \text{authenticator} &= E(\text{timestamp}, S_A) \\ \text{REPLY} &= E(\text{"Bob"}, K_{AB}, \text{ticket to Bob}; S_A) \\ \text{ticket to Bob} &= E(\text{"Alice"}, K_{AB}; K_B) \end{aligned}$$

In Figure 10.20, the KDC obtains the key S_A from the TGT and uses this key to verify the timestamp. Also, the key K_{AB} is the session key that Alice and Bob will use for their session.

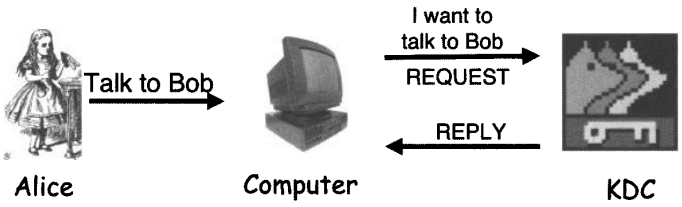


Figure 10.20: Alice Gets Ticket to Bob

Once Alice has obtained the "ticket to Bob," she can then securely communicate with Bob. This process is illustrated in Figure 10.21, where the ticket to Bob is as above and

$$\text{authenticator} = E(\text{timestamp}, K_{AB}).$$

Note that Bob decrypts "ticket to Bob" with his key K_B to obtain K_{AB}, which he then uses to verify the timestamp. The key K_{AB} is also used to protect the confidentiality and integrity of the subsequent conversation between Alice and Bob.

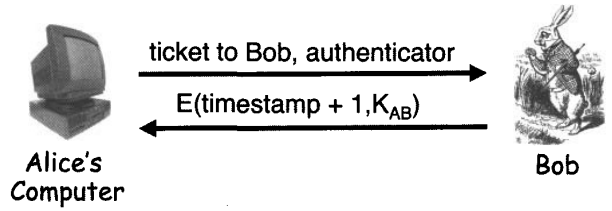


Figure 10.21: Alice Contacts Bob

Since timestamps are used for replay prevention, Kerberos minimizes the number of messages that must be sent. As we mentioned in the previous chapter, the primary drawback to using timestamps is that time becomes a security-critical parameter. Another issue with timestamps is that we can't expect all clocks to be perfectly synchronized and therefore some clock skew must be tolerated. In Kerberos, this clock skew is by default set to five minutes, which seems like an eternity in a networked world.

10.5.3 Kerberos Security

Recall that, when Alice logs in, the KDC sends $E(S_A, \text{TGT}; K_A)$ to Alice, where $\text{TGT} = E(\text{"Alice"}, S_A; K_{\text{KDC}})$. Since the TGT is encrypted with the key K_{KDC}, why is the TGT encrypted again with the key K_A? The answer is that this is a minor flaw in Kerberos, since it's extra work that provides no additional security. If the key K_{KDC} is compromised, the entire security of the system is broken, so there is no added benefit to encrypting the TGT again after it's already encrypted with K_{KDC}.

Notice that, in Figure 10.20, Alice remains anonymous in the REQUEST. This is a nice security feature that is a side benefit of the fact that the TGT is encrypted with the key K_{KDC}. That is, the KDC does not need to know who is making the REQUEST before it can decrypt the TGT, since all TGTs are encrypted with K_{KDC}. Anonymity with symmetric keys can be difficult, as we saw with the IPSec symmetric key main mode. But, in this part of Kerberos, anonymity is easy.

In the Kerberos example above, why is "ticket to Bob" sent to Alice, when Alice simply forwards it on to Bob? Apparently, it would be more efficient to have the KDC send the ticket directly to Bob, and the designers of Kerberos were certainly concerned with efficiency (e.g., they use timestamps instead of nonces). However, if the ticket to Bob arrives at Bob before Alice initiates contact, then Bob would have to remember the key K_{AB} until it's needed. That is, Bob would need to maintain state. Statelessness is an important feature of Kerberos.

Finally, how does Kerberos prevent replay attacks? Replay prevention relies on the timestamps that appear in the authenticators. But there is still an issue of replay within the clock skew. To prevent such replay attacks, the KDC would need to remember all timestamps received within the clock skew interval. However, most Kerberos implementations apparently don't bother to do this [162].

Before departing the realm of Kerberos, we consider a design alternative. Suppose we have the KDC remember session keys instead of putting these in the TGT. This design would eliminate the need for TGTs. But it would also require the KDC to maintain state, and a stateless KDC is one of the most impressive design features in Kerberos.

10.6 WEP

Wired Equivalent Privacy, or WEP, is a security protocol that was designed to make a wireless local area network (LAN) as secure as a wired LAN. By any measure, WEP is a seriously flawed protocol. As Tanenbaum so aptly puts it [298]:

> The 802.11 standard prescribes a data link-level security protocol called WEP (Wired Equivalent Privacy), which is designed to make the security of a wireless LAN as good as that of a wired LAN. Since the default for a wired LAN is no security at all, this goal is easy to achieve, and WEP achieves it as we shall see.

10.6.1 WEP Authentication

In WEP, a wireless access point shares a single symmetric key with all users. While it is not ideal to share one key among many users, it certainly does simplify things for the access point. In any case, the actual WEP authentication protocol is a simple challenge-response, as illustrated in Figure 10.22, where Bob is the access point, Alice is a user, and K is the shared symmetric key.

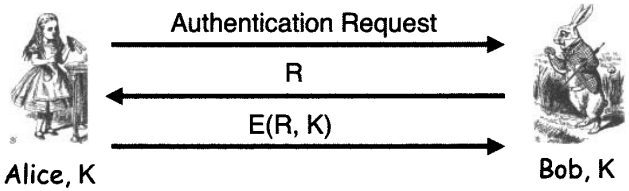


Figure 10.22: WEP Authentication

10.6.2 WEP Encryption

Once Alice has been authenticated, packets are encrypted using the RC4 stream cipher (see Section 3.2.2 for details on the RC4 algorithm), as illustrated in Figure 10.23. Each packet is encrypted with a key $K_{\text{IV}} = (\text{IV}, K)$, where IV is a 3-byte initialization vector that is sent in the clear with the packet, and K is the same key used for authentication. The goal here is to encrypt packets with distinct keys, since reuse of the key would be a bad idea (see Problem 36). Note that, for each packet, Trudy knows the 3-byte IV, but she does not know K. So the encryption key varies and it's not known by Trudy.

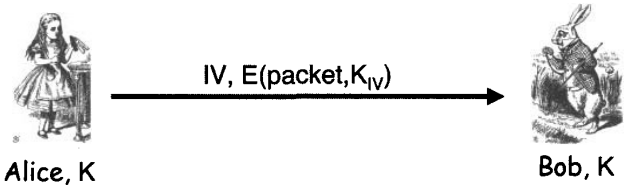


Figure 10.23: WEP Encryption

Since the IV is only three bytes long, and the key K seldom changes, the encryption key $K_{\text{IV}} = (\text{IV}, K)$ will repeat often (see Problems 37). Furthermore, whenever the key K_{IV} repeats, Trudy will know it, since the IV is visible (assuming K has not changed). RC4 is a stream cipher, so a repeated key implies reuse of the keystream, which is a serious problem. Further repeats of the same IV make Trudy's job even easier.

The number of repeated encryption keys could be reduced if K was changed regularly. Unfortunately, the long-term key K seldom changes since, in WEP, such a change is a manual process and the access point and all hosts must update their keys. That is, there is no key update procedure built into WEP.

The bottom line is that, whenever Trudy sees a repeated IV, she can safely assume the same keystream was used. Since the IV is only 24 bits, repeats will occur relatively often. And, since a stream cipher is used, a repeated keystream is at least as bad as reuse of a one-time pad.

In addition to this small-IV problem, there is another distinct cryptanalytic attack on WEP encryption. While RC4 is considered a strong cipher when used correctly, there is a practical attack that can be used to recover the RC4 key from WEP ciphertext. This clever attack, which can be considered a type of related key attack, is due to Fluhrer, Mantin, and Shamir [112]. This attack is discussed in detail in Section 6.3 of Chapter 6, or see [284] for more information.

10.6.3 WEP Non-Integrity

WEP has numerous security problems, but one of the most egregious is that it uses a cyclic redundancy check (CRC) for "integrity" protection. Recall that a cryptographic integrity check is supposed to detect malicious tampering with the data—not just transmission errors. While a CRC is a good error detection method, it is useless for cryptographic integrity, since an intelligent adversary can alter the data and, simultaneously, the CRC value so that the integrity check is passed. This is precisely the attack that a true cryptographic integrity check, such as a MAC, HMAC, or digital signature, will prevent.

This integrity problem is made worse by the fact that the data is encrypted with a stream cipher. Because a stream cipher is used, WEP encryption is linear, which allows Trudy to make changes directly to the ciphertext and change the corresponding CRC value so that the receiver will not detect the tampering. That is, Trudy does not need know the key or plaintext to make undetectable changes to the data. Under this scenario, Trudy won't know what changes she has made, but the point is that the data can be corrupted in a way that neither Alice nor Bob can detect.

The problems only get worse if Trudy should happen to know some of the plaintext. For example, suppose that Trudy knows the destination IP address of a given WEP-encrypted packet. Then without any knowledge of the key, Trudy can change the destination IP address to an IP address of her choosing (for example, her own IP address), and change the CRC integrity check so that her tampering will go undetected. Since WEP traffic is only encrypted from the host to the wireless access point (and vice versa), when the altered packet arrives at the access point, it will be decrypted and forwarded to Trudy's preferred IP address. From the perspective of a lazy cryptanalyst, it doesn't get any better than this. Again, this attack is made possible by the lack of any real integrity check. The bottom line is that the WEP "integrity check" provides no cryptographic integrity whatsoever.

10.6.4 Other WEP Issues

There are many more WEP security vulnerabilities. For example, if Trudy can send a message over the wireless link and intercept the ciphertext, then she will know the plaintext and the corresponding ciphertext, which enables her to immediately recover the keystream. This same keystream will be used to encrypt any message that uses the same IV, provided the long-term key has not changed (which, as pointed out above, it seldom does).

Would Trudy ever know the plaintext of an encrypted message sent over the wireless link? Perhaps Trudy could send an email message to Alice and ask her to forward it to another person. If Alice does so, then Trudy could intercepted the ciphertext message corresponding to the known plaintext.

Another issue is that, by default, a WEP access point broadcasts its SSID (the Service Set Identifier), which acts as its ID. The client must use the SSID when authenticating to the access point. One security feature of WEP makes it possible to configure the access point so that it does not broadcast the SSID, in which case the SSID acts something like a password that users must know to authenticate to the access point. However, users send the SSID in the clear when contacting the access point, and Trudy only needs to intercept one such packet to discover the SSID "password." Even worse, there are tools that will force WEP clients to de-authenticate, in which case the clients will then automatically attempt to re-authenticate, in the process, sending the SSID in the clear. Consequently, as long as there is at least one active user, it's a fairly simple process for Trudy to obtain the SSID.

10.6.5 WEP: The Bottom Line

It's difficult—if not impossible—to view WEP as anything but a security disaster. However, in spite of all of its multiple security problems, in some circumstances it may be possible to make WEP moderately secure in practice. Ironically, this has more to do with the inherent insecurity of WEP than with any inherent security of WEP. Suppose that you configure your WEP access points so that it encrypts the data, it does not broadcast its SSID, and you use access control (i.e., only machines with specified MAC addresses are allowed to use the access point). Then an attacker must expend some effort to gain access—at a minimum, Trudy must break the encryption, spoof her MAC address, and probably force users to de-authenticate so that she can obtain the SSID. While there are tools to help with all of these tasks, it would likely be much simpler for Trudy to find an unprotected WEP network. Like most people, Trudy generally chooses the path of least resistance. Of course, if Trudy has reason to specifically target your WEP installation (as opposed to simply wanting free network access), you will be vulnerable as long as you rely on WEP.

Finally, we note that there are more secure alternatives to WEP. For example, Wi-Fi Protected Access (WPA) is significantly stronger, but it was designed to use the same hardware as WEP, so some security compromises were necessary. A few attacks on WPA are known but, as a practical matter, it seems to be secure. There is also a WPA2 which, in principle, is somewhat stronger than WPA, but it requires more powerful hardware. As with WPA, there are some claimed attacks on WPA2, but these also appear to be of little practical significance. Today, WEP can be broken in minutes whereas the only serious threats against WPA and WPA2 are password cracking attacks. If reasonably strong passwords are chosen, WPA and WPA2 both would be considered practically secure, by any conceivable definition. In any case, both WPA and WPA2 are vast improvements over WEP [325].

10.7 GSM

To date, many wireless protocols, such as WEP, have a poor track record with respect to security [17, 38, 93]. In this section we'll discuss the security of GSM cell phones. GSM illustrates some of the unique security problems that arise in a wireless environment. It's also an excellent example of how mistakes at the design phase are extremely difficult to correct later. But before we delve into to GSM security, we need some background information on the development of cell phone technology.

Back in the computing stone age (prior to the 1980s, that is) cell phones were expensive, completely insecure, and as large as a brick. These *first-generation* cell phones were analog, not digital, and there were few standards and little or no thought was given to security.

The biggest security issue with early cell phones was their susceptibility to *cloning*. These cell phones would send their identity in the clear when a phone call was placed, and this identity was used to determine who to bill for the phone call. Since the ID was sent over a wireless media, it could easily be captured and then used to make a copy or clone, of the phone. This allowed the bad guys to make free phone calls, which did not please cellular phone companies, who ultimately had to bear the cost. Cell phone cloning became a big business, with fake base stations created simply to harvest IDs [14].

Into this chaotic environment came GSM, which began in 1982 as Groupe Spéciale Mobile, but in 1986 it was formally rechristened as Global System for Mobile Communications.[14] The founding of GSM marks the official beginning of *second-generation* cell phone technology [142]. We'll have much more to say about GSM security below.

Recently, *third-generation* cell phones have become popular. The 3rd Generation Partnership Project, or 3GPP [1], is the trade group behind 3G phones. We'll briefly mention the security architecture promoted by the 3GPP after we complete our survey of GSM security.

10.7.1 GSM Architecture

The general architecture of GSM is illustrated in Figure 10.24, where the following terminology is used.

- The *mobile* is the cell phone.
- The *air interface* is where the wireless transmission from the cell phone to a base station occurs.
- The *visited network* typically includes multiple base stations and a *base station controller*, which acts as a hub for connecting the base stations under its control to the rest of the GSM network. The base station controller includes a *visitor location registry*, or VLR, which is used to keep tabs on all mobiles currently active in the VLR's network.
- The *public switched telephone network*, or PSTN, is the ordinary (non-cellular) telephone system. The PSTN is sometimes referred to as "land lines" to distinguish it from the wireless network.
- The *home network* is the network where the mobile is registered. Each mobile is associated with a unique home network. The home network includes a *home location registry*, or HLR, which keeps track of the most recent location of all mobiles listed in the HLR. The *authentication center*, or AuC, maintains the crucial billing information for all mobiles that belong to the corresponding HLR.

We'll discuss these pieces of the GSM puzzle in more detail below.

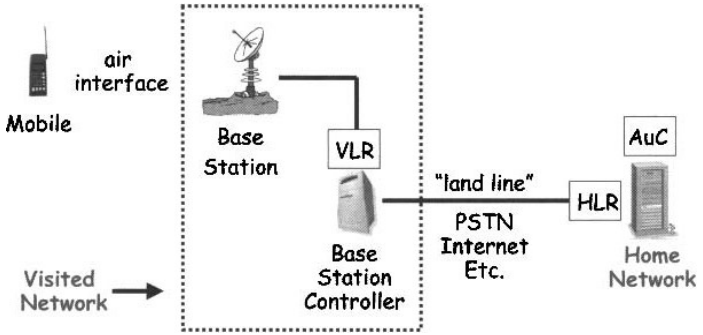


Figure 10.24: GSM Overview

Each GSM mobile phone contains a Subscriber Identity Module, or SIM, which is a tamper-resistant smartcard. The SIM contains an International Mobile Subscriber ID, or IMSI, which, not surprisingly, is used to identify the mobile. The SIM also contains a 128-bit key that is known only by the mobile and its home network. This key is universally know as Ki, so we'll follow the standard notation.

The purpose of using a smartcard for the SIM is to provide an inexpensive form of tamper-resistant hardware. The SIM card also provides two-factor authentication, relying on "something you have" (the mobile containing the SIM) and "something you know" in the form of a four-digit PIN. However, the PIN is usually treated as an annoyance, and it's often not used.

Again, the visited network is the network where the mobile is currently located. A base station is one cell in the cellular system, whereas the base

[14] This is a tribute to the universality of three-letter acronyms.

station controller manages a collection of cells. The VLR has information on all mobiles currently visiting the base station controller's territory.

The home network stores a given mobile's crucial information, namely, its IMSI and key Ki. Note that the IMSI and Ki are, in effect, the username and "password" for the mobile when it wants to access the network to make a call. The HLR keeps track of the most recent location of each of it's registered mobiles, while the AuC contains each registered mobile's IMSI and key Ki.

10.7.2 GSM Security Architecture

Now we're ready to take a close look at the GSM security architecture. The primary security goals set forth by the designers of GSM were the following.

- Make GSM as secure as ordinary telephones (the PSTN)
- Prevent cell phone cloning

Note that GSM was not designed to resist an active attack. At the time, active attacks were considered infeasible, since the necessary equipment was costly. However, today the cost of such equipment is little more than that of a good laptop computer, so neglecting active attacks was probably shortsighted. The designers of GSM considered the biggest threats to be insecure billing, corruption, and similar low-tech attacks.

GSM attempts to deal with three security issues: anonymity, authentication, and confidentiality. In GSM, the anonymity is supposed to prevent intercepted traffic from being used to identify the caller. Anonymity is not particularly important to the phone companies, except to the extent that it is important for customer confidence. Anonymity is something users might reasonably expect from non-cellular phone calls.

Authentication, on the other hand, is of paramount importance to phone companies, since correct authentication is necessary for proper billing. The first-generation cloning problems can be viewed as an authentication failure. As with anonymity, confidentiality of calls over the air interface is important to customers, and so, to that extent, it's important to phone companies.

Next, we'll look at GSM's approach to anonymity, authentication, and confidentiality in more detail. Then we'll discuss some of the many security flaws in GSM.

10.7.2.1 Anonymity

GSM provides a very limited form of anonymity. The IMSI is sent in the clear over the air interface at the start of a call. Then a random Temporary Mobile Subscriber ID, or TMSI, is assigned to the caller, and the TMSI is subsequently used to identify the caller. In addition, the TMSI changes frequently. The net effect is that, if an attacker captures the initial part

of a call, the caller's anonymity will be compromised. But if the attacker misses the initial part of the call, then the anonymity is, in a practical sense, reasonably well protected. Although this is not a strong form of anonymity, it may be sufficient for real-world situations where an attacker could have difficulty filtering the IMSIs out of a large volume of traffic. It seems that the GSM designers did not take anonymity too seriously.

10.7.2.2 Authentication

From the phone company's perspective, authentication is the most critical aspect of the GSM security architecture. Authenticating the user to the base station is necessary to ensure that the phone company will get paid for the service they provide. In GSM, the caller is authenticated to the base station, but the authentication is not mutual. That is, the GSM designers decided that it was not necessary to verify the identity of the base station. We'll see that this was a significant security oversight.

GSM authentication uses a simple challenge-response mechanism. The caller's IMSI is received by the base station, which then passes it to the caller's home network. Recall that the home network knows the caller's IMSI and key Ki. The home network generates a random challenge, RAND, and computes the "expected response," XRES = A3(RAND, Ki), where A3 is a hash function. Then the pair (RAND, XRES) is sent from the home network to the base station. The base station sends the challenge, RAND, to the mobile. The mobile's response is denoted SRES, where SRES is computed by the mobile as SRES = A3(RAND, Ki). To complete the authentication, the mobile sends SRES to the base station which verifies that SRES = XRES. Note that in this authentication protocol, the caller's key Ki never leaves its home network or the mobile. It's important that Trudy cannot obtain Ki, since that would enable her to clone the caller's phone.

10.7.2.3 Confidentiality

GSM uses a stream cipher to encrypt the data. The reason for this choice is due to the relatively high error rate in the cell phone environment, which is typically about 1 in 1000 bits. With a block cipher, each transmission error causes one or two plaintext blocks to be garbled (depending on the mode), while a stream cipher garbles only those plaintext bits corresponding to the specific ciphertext bits that are in error.[15]

The GSM encryption key is universally denoted as Kc, so we'll follow that convention. When the home network receives the IMSI from the base station controller, the home network computes Kc = A8(RAND, Ki), where A8 is

[15] It is possible to use error correcting codes to minimize the effects of transmission errors, making block ciphers feasible. However, this adds another layer of complexity to the process.

another hash function. Then Kc is sent along with the pair RAND and XRES, that is, the triple (RAND, XRES, Kc) is sent from the home network to the base station.[16]

Once the base station receives the triple (RAND, XRES, Kc), it uses the authentication protocol described above. If this succeeds, the mobile computes $Kc = \text{A8}(\text{RAND}, Ki)$. The base station already knows Kc, so the mobile and base station have a shared symmetric key with which to encrypt the conversation. As mentioned above, the data is encrypted with the A5/1 stream cipher. As with authentication, the caller's master key Ki never leaves its home network.

10.7.3 GSM Authentication Protocol

The part of the GSM protocol that occurs between the mobile and the base station is illustrated in Figure 10.25. A few security concerns with this protocol are as follows [228].

- The RAND is hashed together with Ki to produce the encryption key Kc. Also, the value of RAND is hashed with Ki to generate SRES, which a passive attacker can see. As a result, it's necessary that SRES and Kc be uncorrelated—otherwise there would have been a shortcut attack on Kc. These hash values will be uncorrelated if a secure cryptographic hash function is used.

- It must not be possible to deduce Ki from known RAND and SRES pairs, since such pairs are available to a passive attacker. This is analogous to a known plaintext attack with a hash function in place of a cipher.

- It must not be possible to deduce Ki from chosen RAND and SRES pairs, which is analogous to a chosen plaintext attack on the hash function. Although this attack might seem implausible, with possession of the SIM card, an attacker can choose the RAND values and observe the corresponding SRES values.[17]

[16]Note that the encryption key Kc is sent from the home network to the base station. Trudy may be able to obtain the encryption key by simply observing traffic sent over the network. In contrast, the authentication key Ki never leaves the home network or the mobile, so it is not subject to such an attack. This shows the relative importance the GSM architects placed on authentication as compared to confidentiality.

[17]If this attack is feasible, it is a threat even if it's slow, since the person who sells the phone would likely possess it for an extended period of time. On the other hand, if the attack is fast, then a phone that is "lost" for a few minutes would be subject to cloning.

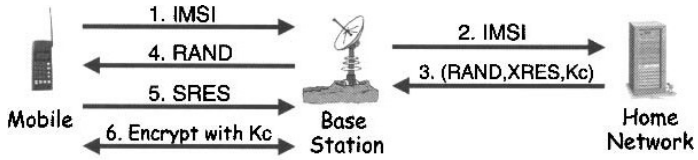


Figure 10.25: GSM Authentication and Encryption

10.7.4 GSM Security Flaws

Next, we'll discuss security flaws in GSM—there are cryptographic flaws and there are protocol flaws as well. But, arguably, the most serious problems arise from invalid security assumptions made by the designers of GSM.

10.7.4.1 Crypto Flaws

There are several cryptographic flaws in GSM. The hashes A3 and A8 both are based on a hash function known as COMP128. The hash COMP128 was developed as a secret design, in violation of Kerckhoffs' Principle. Not surprisingly, COMP128 was later found to be weak—it can be broken by 150,000 chosen "plaintexts" [130]. What this means in practice is that an attacker who has access to a SIM card can determine the key Ki in 2 to 10 hours, depending in the speed of the card. In particular, an unscrupulous seller could determine Ki before selling a phone, then create clones that would have their calls billed to the purchaser of the phone. Below, we'll mention another attack on COMP128.

There are two different forms of the encryption algorithm A5, which are known as A5/1 and A5/2. Recall that we discussed A5/1 in Chapter 3. As with COMP128, both of these ciphers were developed in violation of Kerckhoffs' Principle and both are weak. The A5/2 algorithm is the weaker of the two [26, 234] but feasible attacks on A5/1 are known [33].

10.7.4.2 Invalid Assumptions

There is a serious design flaw in the GSM protocol. A GSM phone call is encrypted between the mobile and the base station but not from the base station to the base station controller. Recall that a design goal of GSM was to develop a system as secure as the public switched telephone network (PSTN). As a result, if a GSM phone call is at some point routed over the PSTN, then from that point on, no further special protection is required. Consequently, the emphasis of GSM security is on protecting the phone call over the air interface, between the mobile and the base station.

The designers of GSM assumed that once the call reached the base station, it would be routed over the PSTN to the base station controller. This is implied by the solid line between the base station and base station controller in Figure 10.24. Due to this assumption, the GSM security protocol does not protect the conversation when it is sent from the base station to the base station controller. However, many GSM systems actually transmit calls between a base station and its base station controller over a microwave link [228]. Since microwave is a wireless media, it is possible (but not easy) for an attacker to eavesdrop on unprotected calls over this link, rendering the encryption over the air interface useless.

10.7.4.3 SIM Attacks

Several attacks have been developed on various generations of SIM cards. In one *optical fault induction* attack, an attacker could force a SIM card to divulge its Ki by using an ordinary flashbulb [269]. In another class of attacks, known as *partitioning attacks*, timing and power consumption analysis could be used to recover Ki using as few as eight adaptively chosen plaintexts [243]. As a result, an attacker who has possession of the SIM could recover Ki in seconds and, consequently, a misplaced cell phone could be cloned in seconds.

10.7.4.4 Fake Base Station

Another serious flaw with the GSM protocol is the threat posed by a fake base station. This attack, which is illustrated in Figure 10.26, exploits two flaws in the protocol. First, the authentication is not mutual. While the caller is authenticated to the base station (which is necessary for proper billing), the designers of GSM felt it was not worth the extra effort to authenticate the base station to the caller. Although they were aware of the possibility of a fake base station, apparently the protocol designers believed that the probability of such an attack was too remote to justify the (small) additional cost of mutual authentication. The second flaw that this attack exploits is that encryption over the air interface is not automatic. In fact, the base station determines whether the call is encrypted or not, and the caller does not know which is the case.

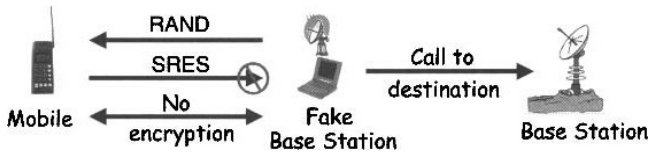


Figure 10.26: GSM Fake Base Station

In the attack illustrated in Figure 10.26, the fake base station sends a random value to the mobile, which the mobile assumes is RAND. The mobile replies with the corresponding SRES, which the fake base station discards, since it does not intend to authenticate the caller (in fact, it cannot authenticate the caller). The fake base station then tells the mobile not to encrypt the call. Unbeknownst to either the caller or the recipient, the fake base station then places a call to the the intended recipient and forwards the conversation from the caller to the recipient and vice versa. The fake base station can then eavesdrop on the entire conversation.

Note that in this fake base station attack, the fake base station would be billed for the call, not the caller. The attack might be detected if the caller complained about not being billed for the phone call. But, would anyone complain about not receiving a bill?

Also, the fake base station is in position to send any RAND it chooses and receives the corresponding SRES. Therefore, it can conduct a chosen plaintext attack on the SIM without possessing the SIM card. The SIM attack mentioned above that requires eight adaptively chosen plaintexts would be feasible with a fake base station.

Another major flaw with the GSM protocol is that it provides no replay protection. A compromised triple $(\text{RAND}, \text{XRES}, Kc)$ can be replayed forever. As a result, one compromised triple gives an attacker a key Kc that is valid indefinitely. A clever fake base station operator could even use a compromised triple to "protect" the conversation between the mobile and the fake base station so that nobody else could eavesdrop on the conversation.

Finally, it's worth noting that denial of service is always an issue in a wireless environment, since the signal can be jammed. But jamming is an issue beyond the scope of a security protocol.

10.7.5 GSM Conclusions

From our discussion of GSM security flaws, it might seem that GSM is a colossal security failure. However, GSM was certainly a commercial success, which raises some questions about the financial significance of good security. In any case, it is interesting to consider whether GSM achieved its security design goals. Recall that the two goals set forth by the designers of GSM were to eliminate the cloning that had plagued first-generation systems and to make the air interface as secure as the PSTN. Although it is possible to clone GSM phones, it never became a significant problem in practice. So it would seem that GSM did achieve its first security goal.

Did GSM make the air interface as secure as the PSTN? There are attacks on the GSM air interface (e.g., fake base station), but there are also attacks on the PSTN (tapping a line) that are at least as severe. So it could be argued that GSM achieved its second design goal, although this is debatable.

The real problem with GSM security is that the initial design goals were too limited. The major insecurities in GSM include weak crypto, SIM issues, the fake base station attack, and a total lack of replay protection. In the PSTN, the primary insecurity is tapping, though there are others threats, such as attacks on cordless phones. Overall, GSM could reasonably be considered a modest security success.

10.7.6 3GPP

The security design for third-generation cell phones was spearheaded by the 3GPP. This group clearly set their sights higher than the designers of GSM. Perhaps surprisingly, the 3GPP security model is built on the foundation of GSM. However, the 3GPP developers carefully patched all of the known GSM vulnerabilities. For example, 3GPP includes mutual authentication and integrity protection of all signaling, including the "start encryption" command for the base station to mobile communication. These improvements eliminate the GSM-style fake base station attack. Also, in 3GPP, the keys can't be reused and triples can't be replayed. The weak proprietary crypto algorithms of GSM (COMP128, A5/1, and A5/2) have been replaced by the strong encryption algorithm, KASUMI, which has undergone rigorous peer review. In addition, the encryption has been extended from the mobile all the way to the base station controller.

The history of mobile phones, from the first-generation through GSM and now 3GPP, nicely illustrates the evolution that often occurs in security. As the attackers develop new attacks, the defenders respond with new protections, which the attackers again probe for weaknesses. Ideally, this arms race approach to security could be avoided by a careful design and analysis prior to the initial development. However, it's unrealistic to believe that the designers of first-generation cell phones could have imagined the mobile world of today. Attacks such as the fake base station, which would have seemed improbable at one time, are now easily implemented. With this in mind, we should realize that, although 3GPP clearly promises more security than GSM could deliver, it's possible that attacks will eventually surface. In short, the security arms race continues.

10.8 Summary

In this chapter, we discussed several real-world security protocols in detail. We first considered SSH, which is a fairly straightforward protocol. Then we looked at SSL which is a well-designed protocol that is widely used on the Internet.

We saw that IPSec is a complex protocol with some serious security issues. The designers of IPSec over-engineered the protocol, which is the source of its complex. IPSec provides a good illustration of the maxim that complexity is the enemy of security.

Kerberos is a widely deployed authentication protocol that relies on symmetric key cryptography and timestamps. The ability of the Kerberos KDC to remain stateless is one of the many clever features of the protocol.

We finished the chapter with a discussion of two wireless protocols, WEP and GSM. WEP is a seriously flawed protocol—one of its many problems is the lack of any meaningful integrity check. You'd be hard pressed to find a better example to illustrate the pitfalls that arise when integrity is not protected.

GSM is another protocol with some major problems. The actual GSM security protocol is simple, but it has a large number of flaws. While complexity might be the enemy of security, GSM illustrates that simplicity isn't necessarily security's best friend. Arguably, the most serious problem with GSM is that its designers were not ambitious enough, since they didn't design GSM to withstand attacks that are easy today. This is perhaps excusable given that some of these attacks seemed far fetched in 1982 when GSM was developed. GSM also shows that it's difficult to overcome security flaws after the fact.

The security of third-generation cell phones is built on the GSM model, with all of the known security flaws in GSM having been patched. It will be interesting to see how 3GPP security holds up in practice.

10.9 Problems

1. Consider the SSH protocol in Figure 10.1.
   a. Explain precisely how and where Alice is authenticated. What prevents a replay attack?
   b. If Trudy is a passive attacker (i.e., she can only observe messages), she cannot determine the key K. Why?
   c. Show that if Trudy is an active attacker (i.e., she can actively send messages) and she can impersonate Bob, then she can determine the key K that Alice uses in the last message. Explain why this does not break the protocol.
   d. What is the purpose of the encrypting the final message with the key K?
2. Consider the SSH protocol in Figure 10.1. One variant of the protocol allows us to replace Alice's certificate, certificate$_A$, with Alice's password, password$_A$. Then we must also remove S_A from the final message. This modification yields a version of SSH where Alice is authenticated based on a password.

a. What does Bob need to know so that he can authenticate Alice?

b. Based on Problem 1, part b, we see that Trudy, as an active attacker, can establish a shared symmetric key K with Alice. Assuming this is the case, can Trudy then use K to determine Alice's password?

c. What are the significant advantages and disadvantages of this version of SSH, as compared to the version in Figure 10.1, which is based on certificates?

3. Consider the SSH protocol in Figure 10.1. One variant of the protocol allows us to replace certificate$_A$ with Alice's public key. In this version of the protocol, Alice must have a public/private key pair, but she is not required to have a certificate. It is also possible to replace certificate$_B$ with Bob's public key.

a. Suppose that Bob has a certificate, but Alice does not. What must Bob do so that he can authenticate Alice?

b. Suppose that Alice has a certificate, but Bob does not. What must Alice do so that she can authenticate Bob?

c. What are the significant advantages and disadvantages of this public key version of SSH, as compared to the certificate version in Figure 10.1?

4. Use Wireshark [328] to capture SSH authentication packets.

a. Identify the packets that correspond to the messages shown in Figure 10.1.

b. What other SSH packets do you observe, and what do these packets contain?

5. Consider the SSH specification, which can be found in RFC 4252 [331] and RFC 4253 [333].

a. Which message or messages in Figure 10.1 correspond to the message or messages labeled as SSH_MSG_KEXINIT in the protocol specification?

b. Which message or messages in Figure 10.1 correspond to the message or messages labeled as SSH_MSG_NEWKEYS in the protocol specification?

c. Which message or messages in Figure 10.1 correspond to the message or messages labeled as SSH_MSG_USERAUTH in the protocol specification?

d. In the actual SSH protocol, there are two additional messages that would come between the fourth and fifth messages in Figure 10.1. What are these messages and what purpose do they serve?

6. Consider the SSL protocol in Figure 10.4.

a. Suppose that the nonces R_A and R_B are removed from the protocol and we define $K = h(S)$. What effect, if any, does this have on the security of the authentication protocol?

b. Suppose that we change message four to

$$\text{HMAC}(\text{msgs}, \text{SRVR}, K).$$

What effect, if any, does this have on the security of the authentication protocol?

c. Suppose that we change message three to

$$\{S\}_{\text{Bob}},\ h(\text{msgs}, \text{CLNT}, K).$$

What effect, if any, does this have on the security of the authentication protocol?

7. Consider the SSL protocol in Figure 10.4. Modify the protocol so that the authentication is based on a digital signature. Your protocol must provide secure authentication of the server Bob, and a secure session key.

8. Consider the SSL protocol in Figure 10.4. This protocol does not allow Bob to remain anonymous, since his certificate identifies him.

a. Modify the SSL session protocol so that Bob can remain anonymous with respect to a passive attacker.

b. Can you solve part a without increasing the number of messages?

9. The SSL protocol discussed in Section 10.3 uses public key cryptography.

a. Design a variant of SSL that is based on symmetric key cryptography.

b. What is the primary disadvantage of using symmetric keys for an SSL-like protocol?

10. Use Wireshark [328] to capture SSL authentication packets.

a. Identify the packets that correspond to the messages shown in Figure 10.4.

b. What do the other SSL packets contain?

11. SSL and IPSec are both designed to provide security over the network.

    a. What are the primary advantages of SSL over IPSec?

    b. What are the primary advantages of IPSec over SSL?

12. SSL and IPSec are both designed to provide security over the network.

    a. What are the significant similarities between the two protocols?

    b. What are the significant differences between the two protocols?

13. Consider a man-in-the-middle attack on an SSL session between Alice and Bob.

    a. At what point should this attack fail?

    b. What mistake might Alice reasonably make that would allow this attack to succeed?

14. In Kerberos, Alice's key K_A, which is shared by Alice and the KDC, is computed (on Alice's computer) as $K_A = h(\text{Alice's password})$. Alternatively, this could have been implemented as follows. Initially, the key K_A is randomly generated on Alice's computer. The key is stored on Alice's computer as $E(K_A, K)$ where the key K is computed as $K = h(\text{Alice's password})$. The key K_A is also stored on the KDC.

    a. What are the advantages to this alternate approach of generating and storing K_A?

    b. Are there any disadvantages to computing and storing $E(K_A, K)$?

15. Consider the Kerberos interaction discussed in Section 10.5.2.

    a. Why is the ticket to Bob encrypted with K_B?

    b. Why is "Alice" included in the (encrypted) ticket to Bob?

    c. In the REPLY message, why is the ticket to Bob encrypted with the key S_A?

    d. Why is the ticket to Bob sent to Alice (who must then forward it to Bob) instead of being sent directly to Bob?

16. Consider the Kerberized login discussed in this chapter.

    a. What is a TGT and what is its purpose?

    b. Why is the TGT sent to Alice instead of being stored on the KDC?

    c. Why is the TGT encrypted with K_{KDC}?

d. Why is the TGT encrypted with K_A when it is sent from the KDC to Alice's computer?

17. This problem deals with Kerberos.

    a. Why can Alice remain anonymous when requesting a ticket to Bob?

    b. Why can Alice not remain anonymous when requesting a TGT from the KDC?

    c. Why can Alice remain anonymous when she sends the "ticket to Bob" to Bob?

18. Suppose we use symmetric keys for authentication and each of N users must be able to authenticate any of the other $N - 1$ users. Evidently, such a system requires one symmetric key for each pair of users, or on the order of N^2 keys. On the other hand, if we use public keys, only N key pairs are required, but we must then deal with PKI issues.

    a. Kerberos authentication uses symmetric keys, yet only N keys are required for N users. How is this accomplished?

    b. In Kerberos, no PKI is required. But, in security, there is no free lunch, so what's the tradeoff?

19. Dog race tracks often employ Automatic Betting Machines (ABMs),[18] which are somewhat analogous to ATM machines. An ABM is a terminal where Alice can place her own bets and scan her winning tickets. An ABM does not accept or dispense cash. Instead, an ABM only accepts and dispenses *vouchers*. A voucher can also be purchased from a special voucher machine for cash, but a voucher can only be redeemed for cash by a human teller.

    A voucher includes 15 hexadecimal digits, which can be read by a human or scanned by a machine—the machine reads a bar code on the voucher. When a voucher is redeemed, the information is recorded in a voucher database and a paper receipt is printed. For security reasons, the (human) teller must submit the paper receipt which serves as the physical record that the voucher was cashed.

    A voucher is valid for one year from its date of issue. However, the older that a voucher is, the more likely that it has been lost and will never be redeemed. Since vouchers are printed on cheap paper, they are often damaged to the point where they fail to scan, and they can even be difficult for human tellers to process manually.

[18] Not to be confused with anti-ballistic missiles.

A list of all outstanding vouchers is kept in a database. Any human teller can read the first 10 hex digits from this database for any outstanding voucher. But, for security reasons, the last five hex digits are not available to tellers.

If Ted, a teller, is asked to cash a valid voucher that doesn't scan, he must manually enter its hex digits. Using the database, it's generally easy for Ted to match the first 10 hex digits. However, the last five hex digits must be determined from the voucher itself. Determining these last five hex digits can be difficult, particularly if the voucher is in poor condition.

To help overworked tellers, Carl, a clever programmer, added a wildcard feature to the manual voucher entry program. Using this feature, Ted (or any other teller) can enter any of the last five hex digits that are readable and "*" for any unreadable digits. Carl's program will then inform Ted whether an outstanding voucher exists that matches in the digits that were entered, ignoring any position with a "*." Note that this program does not give Ted the missing digits, but instead, it simply returns a yes or no answer.

Suppose that Ted is given a voucher for which none of the last five hex digits can be read.

a. Without the wildcard feature, how many guesses must Ted make, on average, to recover the last five hex digits of this particular voucher?

b. Using the wildcard feature, how many guesses, on average, must Ted make to recover the last 5 hex digits of this voucher?

c. How could Dave, a dishonest teller, exploit the wildcard feature to cheat the system?

d. What is the risk for Dave? That is, how might Dave get caught under the current system?

e. Modify the current system so that it allows tellers to securely and efficiently deal with vouchers that fail to scan automatically, but also makes it impossible (or at least more difficult) for Dave to cheat the system.

20. IPSec is a much more complex protocol than SSL, which is often attributed to the fact that IPSec is over-engineered. Suppose that IPSec was not over-engineered. Then would IPSec still be more complex than SSL? In other words, is IPSec inherently more complex than SSL, or not?

21. IKE has two phases, Phase 1 and Phase 2. In IKE Phase 1, there are four key options and, for each of these, there is a main mode and an aggressive mode.

    a. What are the primary differences between main mode and aggressive mode?

    b. What is the primary advantage of the Phase 1 digital signature key option over Phase 1 public key encryption?

22. IKE has two phases, Phase 1 and Phase 2. In IKE Phase 1, there are four key options and, for each of these, there is a main mode and an aggressive mode.

    a. Explain the difference between Phase 1 and Phase 2.

    b. What is the primary advantage of Phase 1 public key encryption main mode over Phase 1 symmetric key encryption main mode?

23. IPSec cookies are also known as anti-clogging tokens.

    a. What is the intended security purpose of IPSec cookies?

    b. Why do IPSec cookies fail to fulfill their intended purpose?

    c. Redesign the IPSec Phase 1 symmetric key signing main mode so that the IPSec cookies do serve their intended purpose.

24. In IKE Phase 1 digital signature main mode, proof_A and proof_B are signed by Alice and Bob, respectively. However, in IKE Phase 1, public key encryption main mode, proof_A and proof_B are neither signed nor encrypted with public keys. Why is it necessary to sign these values in digital signature mode, yet it is not necessary to public key encrypt (or sign) them in public key encryption mode?

25. As noted in the text, IKE Phase 1 public key encryption aggressive mode[19] allows Alice and Bob to remain anonymous. Since anonymity is usually given as the primary advantage of main mode over aggressive mode, is there any reason to ever use public key encryption main mode?

26. IKE Phase 1 uses ephemeral Diffie–Hellman for perfect forward secrecy (PFS). Recall that in our example of PFS in Section 9.3.4 of Chapter 9, we encrypted the Diffie–Hellman values with a symmetric key to prevent the man-in-the-middle attack. However, the Diffie–Hellman values are not encrypted in IKE. Is this a security flaw? Explain.

[19]Don't try saying "IKE Phase 1 public key encryption aggressive mode" all at once or you might give yourself a hernia.

27. We say that Trudy is a *passive attacker* if she can only observe the messages sent between Alice and Bob. If Trudy is also able to insert, delete, or modify messages, we say that Trudy is an *active attacker*. If, in addition to being an active attacker, Trudy is able to establish a legitimate connection with Alice or Bob, then we say that Trudy is an *insider*. Consider IKE Phase 1 digital signature main mode.

    a. As a passive attacker, can Trudy determine Alice's identity?

    b. As a passive attacker, can Trudy determine Bob's identity?

    c. As an active attacker, can Trudy determine Alice's identity?

    d. As an active attacker, can Trudy determine Bob's identity?

    e. As an insider, can Trudy determine Alice's identity?

    f. As an insider, can Trudy determine Bob's identity?

28. Repeat Problem 27 for symmetric key encryption, main mode.

29. Repeat Problem 27 for public key encryption, main mode.

30. Repeat Problem 27 for public key encryption, aggressive mode.

31. Recall that IPSec transport mode was designed for host-to-host communication, while tunnel mode was designed for firewall-to-firewall communication.

    a. Why does IPSec tunnel mode fail to hide the header information when used from host to host?

    b. Does IPSec tunnel mode also fail to hide the header information when used from firewall to firewall? Why or why not?

32. Recall that IPSec transport mode was designed for host-to-host communication, while tunnel mode was designed for firewall-to-firewall communication.

    a. Can transport mode be used for firewall-to-firewall communication? Why or why not?

    b. Can tunnel mode be used for host-to-host communication? Why or why not?

33. ESP requires both encryption and integrity, yet it is possible to use ESP for integrity only. Explain this apparent contradiction.

34. What are the significant differences, if any, between AH and ESP with NULL encryption?

35. Suppose that IPSec is used from host to host as illustrated in Figure 10.16, but Alice and Bob are both behind firewalls. What problems, if any, does IPSec create for the firewalls under the following assumptions.

    a. ESP with non-NULL encryption is used.

    b. ESP with NULL encryption is used.

    c. AH is used.

36. Suppose that we modify WEP so that it encrypts each packet using RC4 with the key K, where K is the same key that is used for authentication.

    a. Is this a good idea? Why or why not?

    b. Would this approach be better or worse than $K_{\text{IV}} = (\text{IV}, K)$, as is actually done in WEP?

37. WEP is supposed to protect data sent over a wireless link. As discussed in the text, WEP has many security flaws, one of which involves its use of initialization vectors, or IVs. WEP IVs are 24 bits long. WEP uses a fixed long-term key K. For each packet, WEP sends an IV in the clear along with the encrypted packet, where the packet is encrypted with a stream cipher using the key $K_{\text{IV}} = (\text{IV}, K)$, that is, the IV is pre-pended to the long-term key K. Suppose that a particular WEP connection sends packets containing 1500 bytes over an 11 Mbps link.

    a. If the IVs are chosen at random, what is the expected amount of time until the first IV repeats? What is the expected amount of time until some IV repeats?

    b. If the IVs are not selected at random but are instead selected in sequence, say, $\text{IV}_i = i$, for $i = 0, 1, 2, \ldots, 2^{24} - 1$, what is the expected amount of time until the first IV repeats? What is the expected amount of time until some IV is repeated?

    c. Why is a repeated IV a security concern?

    d. Why is WEP "unsafe at any key length" [321]? That is, why is WEP no more secure if K is 256 bits than if K is 40 bits? Hint: See [112] for more information.

38. On page 379 it is claimed that if Trudy knows the destination IP address of a WEP-encrypted packet, she can change the IP address to any address of her choosing, and the access point will send the packet to Trudy's selected IP address.

    a. Suppose that C is the encrypted IP address, P is the plaintext IP address (which is known to Trudy), and X is the IP address where

Trudy wants the packet sent. In terms of C, P, and X, what will Trudy insert in place of C?

b. What else must Trudy do for this attack to succeed?

39. WEP also incorporates a couple of security features that were only briefly mentioned in the text. In this problem, we consider two of these features.

    a. By default, a WEP access point broadcasts its SSID, which acts as the name (or ID) of the access point. A client must send the SSID to the access point (in the clear) before it can send data to the access point. It is possible to set WEP so that it does not broadcast the SSID, in which case the SSID is supposed to act like a password. Is this a useful security feature? Why or why not?

    b. It is possible to configure the access point so that it will only accept connections from devices with specified MAC addresses. Is this a useful security feature? Why or why not?

40. After the terrorist attacks of September 11, 2001, it was widely reported that the Russian government ordered all GSM base stations in Russia to transmit all phone calls unencrypted.

    a. Why would the Russian government have given such an order?

    b. Are these news reports consistent with the technical description of the GSM security protocol given in this chapter?

41. Modify the GSM security protocol, which appears in Figure 10.25, so that it provides mutual authentication.

42. In GSM, each home network has an AuC database containing user keys Ki. Instead, a process known as *key diversification* could be used. Key diversification works as follows. Let h be a secure cryptographic hash function and let K_M be a master key known only to the AuCs. In GSM, each user has a unique ID known as an IMSI. In this key diversification scheme, a user's key Ki would be given by $Ki = h(K_M, \text{IMSI})$, and this key would be stored on the mobile. Then given any IMSI, the AuC would compute the key as $Ki = h(K_M, \text{IMSI})$.

    a. What is the primary advantage of key diversification?

    b. What is the primary disadvantage of key diversification?

    c. Why do you think the designers of GSM chose not to employ key diversification?

43. Give a secure one-message protocol that prevents cell phone cloning and establishes a shared encryption key. Mimic the GSM protocol.

44. Give a secure two-message protocol that prevents cell phone cloning, prevents a fake base station attack, and establishes a shared session key. Mimic the GSM protocol.

Part IV

Software

This page intentionally left blank

Chapter 11

Software Flaws and Malware

If automobiles had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside.
— Robert X. Cringely

My software never has bugs. It just develops random features.
— Anonymous

11.1 Introduction

Why is software an important security topic? Is it really on par with crypto, access control, and protocols? For one thing, virtually all of information security is implemented in software. If your software is subject to attack, all of your other security mechanisms are vulnerable. In effect, software is the foundation on which all other security mechanisms rest. We'll see that software provides a poor foundation on which to build security—comparable to building your house on quicksand.[1]

In this chapter, we'll discuss several software security issues. First, we consider unintentional software flaws that can cause security problems [183]. Then we consider malicious software, or malware, which is intentionally designed to be bad. We'll also discuss the future of malware, and we'll mention a few other types of software-based attacks.

Software is a big subject, so we continue with software-related security topics in the next two chapters. Even with three chapters worth of material we can, as usual, do little more than scratch the surface.

[1]Or, in an analogy that is much closer to your fearless author's heart, it's like building a house on a hillside in earthquake country.

11.2 Software Flaws

Bad software is everywhere [143]. For example, the NASA Mars Lander, which cost $165 million, crashed into Mars due to a software error related to converting between English and metric units of measure [150]. Another infamous example is the Denver airport baggage handling system. Bugs in this software delayed the airport opening by 11 months at a cost of more than $1 million per day [122].[2] Software failures also plagued the MV-22 Osprey, an advanced military aircraft—lives were lost due to this faulty software [178]. Attacks on smart electric meters, which have the potential to incapacitate the power grid, have been blamed on buggy software [127]. There are many many more examples of such problems.

In this section, we're interested in the security implications of software flaws. Since faulty software is everywhere, it shouldn't be surprising that the bad guys have found ways to take advantage of this situation.

Normal users find software bugs and flaws more or less by accident. Such users hate buggy software, but out of necessity, they've learned to live with it. Users are surprisingly good at making bad software work.

Attackers, on the other hand, look at buggy software as an opportunity, not a problem. They actively search for bugs and flaws in software, and they like bad software. Attackers try to make software misbehave, and flaws can prove very useful in this regard. We'll see that buggy software is at the core of many (if not most) attacks.

It's generally accepted among computer security professionals that complexity is the enemy of security [74], and modern software is extremely complex. In fact, the complexity of software has far outstripped the abilities of humans to manage the complexity. The number of lines of code (LOC) in a piece of software is a crude measure of its complexity—the more lines of code, the more complex. The numbers in Table 11.1 highlight the extreme complexity of large-scale software projects.

Conservative estimates place the number of bugs in commercial software at about 0.5 per 1,000 LOC [317]. A typical computer might have 3,000 executable files, each of which contains the equivalent of, perhaps, 100,000 LOC, on average. Then, on average, each executable has 50 bugs, which implies about 150,000 bugs living in a single computer.

If we extend this calculation to a a medium-sized corporate network with 30,000 nodes, we'd expect to find about 4.5 billion bugs in the network. Of

[2]The automated baggage handling system proved to be an "unmitigated failure" [87] and it was ultimately abandoned in 2005. As an aside, it's interesting to note that this expensive failure was only the tip of the iceberg in terms of cost overruns and delays for the overall airport project. And, you might be wondering, what happened to the person responsible for this colossal waste of taxpayer money? He was promoted to U.S. Secretary of Transportation [170].

Table 11.1: Approximate Lines of Code

System	LOC
Netscape	17 million
Space shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

course, many of these bugs would be duplicates, but 4.5 billion is still a staggering number.

Now suppose that only 10% of bugs are security critical and that only 10% of these are remotely exploitable. Then our typical corporate network "only" has 4.5 million serious security flaws that are directly attributable to bad software!

The arithmetic of bug counting is good news for the bad guys and very bad news for the good guys. We'll return to this topic later, but the crucial point is that we are not going to eliminate software security flaws any time soon—if ever. We'll discuss ways to reduce the number and severity of flaws, but many flaws will inevitably remain. The best we can realistically hope for is to effectively manage the security risk created by buggy and complex software. In almost any real-world situation, absolute security is often unobtainable, and software is definitely no exception.[3]

In this section, we'll focus on program flaws. These are unintentional software bugs that can have security implications. We'll consider the following specific classes of flaws.

- Buffer overflow
- Race conditions
- Incomplete mediation

After covering these unintentional flaws, we'll turn our attention to malicious software, or malware. Recall that malware is designed to do bad things.

A programming mistake, or bug, is an *error*. When a program with an error is executed, the error might (or might not) cause the program to reach

[3]One possible exception is cryptography—if you use strong crypto, and use it correctly, you are as close to absolutely secure as you will ever be. However, crypto is usually only one part of a security system, so even if your crypto is perfect, many vulnerabilities will likely remain. Unfortunately, people often equate crypto with information security, which leads some to mistakenly expect absolute security.

an incorrect internal state, which is known as a *fault*. A fault might (or might not) cause the system to depart from its expected behavior, which is a *failure* [235]. In other words, an error is a human-created bug, while a fault is internal to the software, and a failure is externally observable.

For example, the C program in Table 11.2 has an error, since `buffer[20]` has not been allocated. This error might cause a fault, where the program reaches an incorrect internal state. If a fault occurs, it might lead to a failure, where the program behaves incorrectly (e.g., the program crashes). Whether a fault occurs, and whether this leads to a failure, depends on what resides in the memory location where `buffer[20]` is written. If that particular memory location is not used for anything important, the program might execute normally, which makes debugging challenging.

Table 11.2: A Flawed Program

```
int main(){
   int buffer[10];
   buffer[20] = 37;}
```

Distinguishing between errors, faults, and failures is a little too pedantic for our purposes. So, for the remainder of this section, we use the term *flaw* as a synonym for all three. The severity should be apparent from context.

One of the primary goals in software engineering is to ensure that a program does what it's supposed to do. However, for software to be secure, a much higher standard is required—secure software software must do what it's supposed to do *and nothing more* [317]. It's difficult enough just trying to ensure that a program does what it's supposed to do. Trying to ensure that a program does "nothing more" is asking for a lot more.

Next, we'll consider three specific types of program flaws that can create significant security vulnerabilities. The first of these is the infamous stack-based *buffer overflow*, also known as *smashing the stack*. Stack smashing has been called the attack of the decade for the 1990s [14] and it's likely to be the attack of the decade for the current decade, regardless of which decade happens to be current. There are several variants of the buffer overflow attack we discuss. These variants are considered in problems at the end of the chapter.

The second class of software flaws we'll consider are *race conditions*. These are common, but generally much more difficult to exploit than buffer overflows. The third major software vulnerability that we consider is *incomplete mediation*. This is the flaw that often makes buffer overflow conditions exploitable. There are other types of software flaws, but these three represent the most common sources of problems.

11.2.1 Buffer Overflow

Alice says, "My cup runneth over, what a mess."
Trudy says, "Alice's cup runneth over, what a blessing."
— Anonymous

Before we discuss buffer overflow attacks in detail, let's consider a scenario where such an attack might arise. Suppose that a Web form asks the user to enter data, such as name, age, date of birth, and so on. The entered information is then sent to a server and the server writes the data entered in the "name" field to a buffer[4] that can hold N characters. If the server software does not verify that the length of the name is at most N characters, then a buffer overflow might occur.

It's reasonably likely that any overflowing data will overwrite something important and cause the computer to crash (or thread to die). If so, Trudy might be able to use this flaw to launch a denial of service (DoS) attack. While this could be a serious issue, we'll see that a little bit of cleverness on Trudy's part can turn a buffer overflow into a much more devastating attack. Specifically, it is sometimes possible for Trudy to execute code of her choosing on the affected machine. It's remarkable that a common programming bug can lead to such an outcome.

Consider again the C source code that appears in Table 11.2. When this code is executed, a buffer overflow occurs. The severity of this particular buffer overflow depends on what resided in memory at the location corresponding to `buffer[20]` before it was overwritten. The buffer overflow might overwrite user data or code, or it could overwrite system data or code, or it might overwrite unused space.

Consider, for example, software that is used for authentication. Ultimately, the authentication decision resides in a single bit. If a buffer overflow overwrites this authentication bit, then Trudy can authenticate herself as, say, Alice. This situation is illustrated in Figure 11.1, where the "F" in the position of the boolean flag indicates failed authentication.

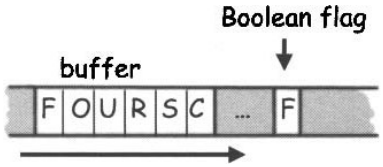


Figure 11.1: Buffer and a Boolean Flag

[4]Why is it a "buffer" and not an "array"? Obviously, it's because we're talking about buffer overflow, not array overflow...

If a buffer overflow overwrites the memory position where the boolean flag is stored, Trudy can overwrite "F" (i.e., a 0 bit) with "T" (i.e., a 1 bit), and the software will believe that Trudy has been authenticated. This attack is illustrated in Figure 11.2.

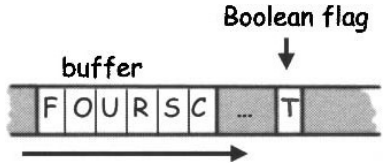


Figure 11.2: Simple Buffer Overflow

Before we can discuss the more sophisticated forms of the buffer overflow attack, we give a quick overview of memory organization for a typical modern processor. A simplified view of memory—which is sufficient for our purposes—appears in Figure 11.3. The *text* section is for code, while the *data* section holds static variables. The *heap* is for dynamic data, while the *stack* can be viewed as "scratch paper" for the processor. For example, dynamic local variables, parameters to functions, and the return address of a function call are all stored on the stack. The *stack pointer*, or SP, indicates the top of the stack. Notice that the stack grows up from the bottom in Figure 11.3, while the heap grows down.

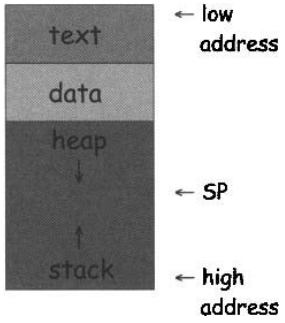


Figure 11.3: Memory Organization

11.2.1.1 Smashing the Stack

Smashing the stack refers to a particularly devastating attack that relies on a buffer overflow. For a stack smashing attack, Trudy is interested in the stack

during a function call. To see how the stack is used during a function call, consider the simple example in Table 11.3.

Table 11.3: Code Example

```
void func(int a, int b){
   char buffer[10];
}
void main(){
   func(1,2);
}
```

When the function `func` in Table 11.3 is called, the values that are pushed onto the stack appear in Figure 11.4. Here, the stack is being used to provide space for the array `buffer` while the function executes. The stack also holds the return address where control will resume after the function finishes executing. Note that `buffer` is positioned above the return address on the stack, that is, `buffer` is pushed onto the stack after the return address. As a result, if the buffer overflows, the overflowing data will overwrite the return address. This is the crucial fact that makes the buffer overflow attack so lethal.

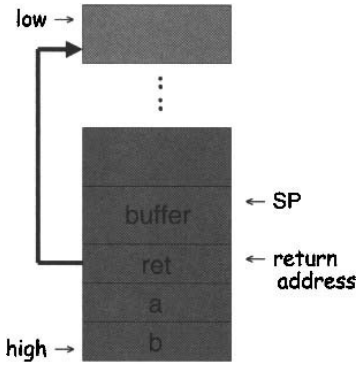


Figure 11.4: Stack Example

The `buffer` in Table 11.3 holds 10 characters. What happens if we put more than 10 characters into `buffer`? The buffer will overflow, analogous to the way that a 5-gallon gas tank will overflow if we try to add 10 gallons of gas. In both cases, the overflow will likely cause a mess. In the buffer overflow case, Figure 11.4 shows that the buffer will overflow into the space where the

return address is located, thereby "smashing" the stack. Our assumption here is that Trudy has control over the bits that go into `buffer` (e.g., the "name" field in a Web form).

If Trudy overflows `buffer` so that the return address is overwritten with random bits, the program will jump to a random memory location when the function has finished executing. In this case, which is illustrated in Figure 11.5, the most likely outcome is that the program crashes.

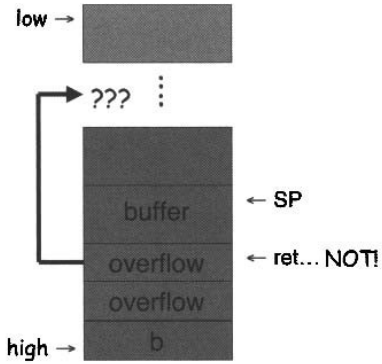


Figure 11.5: Buffer Overflow Causes a Problem

Trudy might be satisfied with simply crashing a program. But Trudy is clever enough to realize that there's much more potential to cause trouble in this situation. Since Trudy can overwrite the return address with a random address, can she also overwrite it with a specific address of her choosing? Often, the answer is yes. If so, what specific address might Trudy want to choose?

With some trial and error, Trudy can probably overwrite the return address with the address of the start of `buffer`. Then the program will try to "execute" the data stored in the buffer. Why might this be useful to Trudy? Recall that Trudy can choose the data that goes into the buffer. So, if Trudy can fill the buffer with "data" that is valid executable code, Trudy can execute this code on the victim's machine. The bottom line is that Trudy gets to execute code of her choosing on the victim's computer. This has to be bad for security. This clever version of the stack smashing attack is illustrated in Figure 11.6.

It's worth reflecting on the buffer overflow attack illustrated in Figure 11.6. Due to an unintentional programming error, Trudy can, in some cases, overwrite the return address, causing code of her choosing to execute on a remote machine. The security implications of such an attack are mind-boggling.

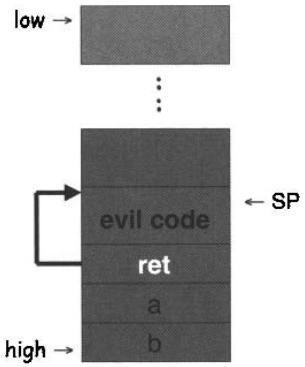


Figure 11.6: Evil Buffer Overflow

From Trudy's perspective, there are a couple of difficulties with this stack smashing attack. First, Trudy may not know the precise address of the evil code she has inserted into `buffer`, and second, she may not know the precise location of the return address on the stack. Neither of these presents an insurmountable obstacle.

Two simple tricks make a buffer overflow attack much easier to mount. For one, Trudy can precede the injected evil code with a `NOP` "landing pad" and, for another, she can insert the desired return address repeatedly. Then, if any of the multiple return addresses overwrite the actual return address, execution will jump to the specified address. And if this specified address lands on any of the inserted `NOP`s, the evil code will be executed immediately after the last `NOP` in the landing pad. This improved stack smashing attack is illustrated in Figure 11.7.

For a buffer overflow attack to succeed, obviously the program must contain a buffer overflow flaw. Not all buffer overflows are exploitable, but those that are enable Trudy to inject code into the system. That is, if Trudy finds an exploitable buffer overflow, she can execute code of her choosing on the affected system. Trudy will probably have some work to do to develop a useful attack, but it certainly can be done. And there are plenty of sources available online to help Trudy hone her skills—the standard reference is [8].

11.2.1.2 Stack Smashing Example

In this section, we'll examine code that contains an exploitable buffer overflow and we'll demonstrate an attack. Of course, we'll be working from Trudy's perspective.

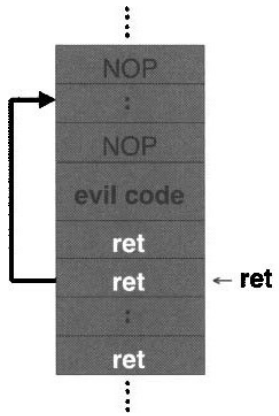


Figure 11.7: Improved Evil Buffer Overflow

Suppose that Trudy is confronted with a program that asks for a serial number—a serial number that Trudy doesn't know. Trudy wants to use the program, but she's too cheap to pay money to obtain a valid serial number.[5] Trudy does not have access to the source code, but she does possess the executable.

When Trudy runs the program and enters an incorrect serial number, the program halts without providing any further information, as indicated in Figure 11.8. Trudy proceeds to try a few different serial numbers, but, as expected, she is unable to guess the correct serial number.

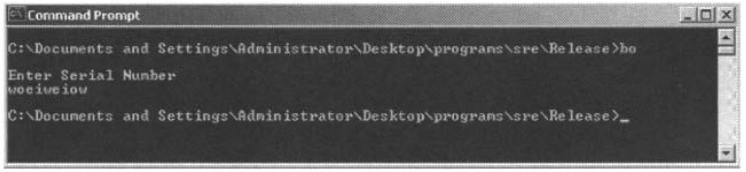


Figure 11.8: Serial Number Program

Trudy then tries entering unusual input values to see how the program reacts. She is hoping that the program will misbehave in some way and that she might have a chance of exploiting the incorrect behavior. Trudy realizes she's in luck when she observes the result in Figure 11.9. This result indicates

[5] In the real world, Trudy would be wise to Google for a serial number. But let's assume that Trudy can't find a valid serial number online.

that the program has a buffer overflow. Note that `0x41` is the ASCII code for the character "A." By carefully examining the error message, Trudy realizes that she has overwritten exactly two bytes of the return address with the character A.

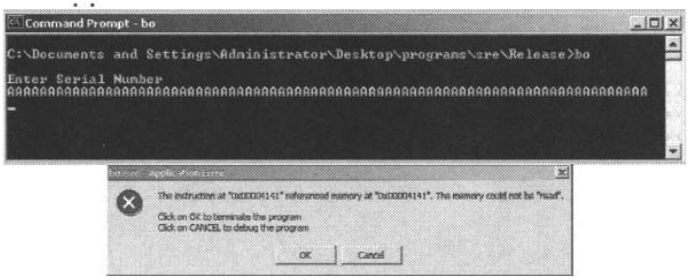


Figure 11.9: Buffer Overflow in Serial Number Program

Trudy then disassembles[6] the `exe` file and obtains the assembly code that appears in Figure 11.10. The significant information in this code is the "Serial number is correct" string, which appears at address `0x401034`. If Trudy can overwrite the return address with the address `0x401034`, then the program will jump to "Serial number is correct" and she will have obtained access to the code, without having any knowledge of the correct serial number.

```
.text:00401000
.text:00401000          sub     esp, 1Ch
.text:00401003          push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008          call    sub_40109F
.text:0040100D          lea     eax, [esp+20h+var_1C]
.text:00401011          push    eax
.text:00401012          push    offset aS       ; "%s"
.text:00401017          call    sub_401088
.text:0040101C          push    8
.text:0040101E          lea     ecx, [esp+28h+var_1C]
.text:00401022          push    offset aS123n456 ; "S123N456"
.text:00401027          push    ecx
.text:00401028          call    sub_401050
.text:0040102D          add     esp, 18h
.text:00401030          test    eax, eax
.text:00401032          jnz     short loc_401041
.text:00401034          push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039          call    sub_40109F
.text:0040103E          add     esp, 4
```

Figure 11.10: Disassembled Serial Number Program

But Trudy can't directly enter a hex address for the serial number, since the input is interpreted as ASCII text. Trudy consults an ASCII table where she finds that `0x401034` is "@^P4" in ASCII, where "^P" is control-P. Confident of success, Trudy starts the program, then enters just enough characters

[6]We'll have more to say about disassemblers in the next chapter when we cover software reverse engineering.

so that she is poised to overwrite the return address, and then she enters "@^P4." To her surprise, Trudy obtains the results in Figure 11.11.

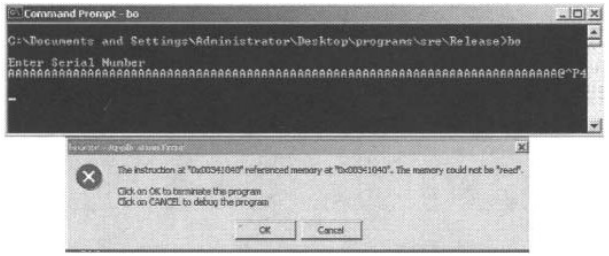


Figure 11.11: Failed Buffer Overflow Attack

A careful examination of the error message shows that the address where the error arose was `0x341040`. Apparently, Trudy caused the program to jump to this address instead of her intended address of `0x401034`. Trudy notices that the intended address and the actual address are byte-reversed. The problem here is that the machine Trudy is dealing with uses the little endian convention, so that the low-order byte is first and the high-order byte comes last. That is, the address that Trudy wants, namely, `0x401034`, is stored internally as `0x341040`. So Trudy changes her attack slightly and overwrites the return address with `0x341040`, which in ASCII is "4^P@." With this change, Trudy is successful, as shown in Figure 11.12.

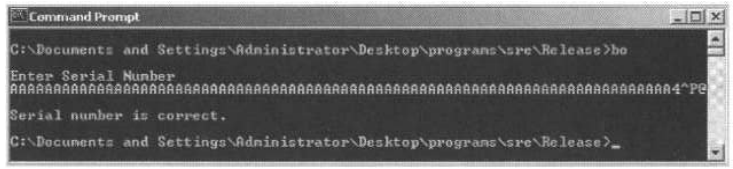


Figure 11.12: Successful Buffer Overflow Attack

The point of this example is that without knowledge of the serial number, and without access to the source code, Trudy was able to break the security of the software. The only tool she used was a disassembler to determine the address that she needed to use to overwrite the return address. In principle, this address could be found by trial and error, although that would be tedious, at best. If Trudy has the executable in her possession, she would be foolish not to employ a disassembler—and Trudy is no fool.

For the sake of completeness, we provide the C source code, `bo.c`, corresponding to the executable, `bo.exe`. This source code appears in Table 11.4.

Table 11.4: Source Code for Serial Number Example

```
main()
{
   char in[75];
   printf("\nEnter Serial Number\n");
   scanf("%s", in);
   if(!strncmp(in, "S123N456", 8))
   {
      printf("Serial number is correct.\n");
   }
}
```

Again, Trudy was able to complete her buffer overflow attack without access to the source code in Table 11.4. We provide the source code here for reference.

Finally, note that in this buffer overflow example, Trudy did not execute code on the stack. Instead, she simply overwrote the return address, which caused the program to execute code that already existed at the specified address. That is, no code injection was employed, which greatly simplfies the attack. This version of stack smashing is usually referred to as a *return-to-libc* attack.

11.2.1.3 Stack Smashing Prevention

There are several possible ways to prevent stack smashing attacks. One approach is to eliminate all buffer overflows from software. However, this is more difficult than it sounds and even if we eliminate all such bugs from new software, there is a huge base of existing software that is riddled with buffer overflows.

Another option is to detect buffer overflows as they occur and respond accordingly. Some programming languages do this automatically. Yet another option is to not allow code to execute on the stack. Finally, if we randomize the location where code is loaded into memory, then the attacker cannot know the address where the `buffer` or other code is located, which would prevent most buffer overflow attacks. In this section, we'll briefly discuss these various options.

An easy way to minimize the damage caused by many stack-based buffer overflows is to make the stack non-executable, that is, do not allow code to

execute on the stack. Some hardware (and many operating systems) support this *no execute*, or NX bit [129]. Using the NX bit, memory can be flagged so that code can't execute in specified locations. In this way the stack (as well as the heap and data sections) can be protected from many buffer overflow attacks. Recent versions of Microsoft Windows support the NX bit [311].

As the NX approach becomes more widely deployed and used, we should see a decline in the number and severity of buffer overflow attacks. However, NX will not prevent all buffer overflow attacks. For example, the return-to-libc attack discussed in the previous section would not be affected. For more information on NX and its security implications, see [173].

Using safe programming languages such as Java or C# will eliminate most buffer overflows at the source. These languages are safe because at runtime they automatically check that all memory accesses are within the declared array bounds. Of course, there is a performance penalty for such checking, and for that reason much code will continue to be written in C, particularly for applications destined for resource-constrained devices. In contrast to these safe languages, there are several C functions that are known to be unsafe and these functions are the source of the vast majority of buffer overflow attacks. There are safer alternatives to all of the unsafe C functions, so the unsafe functions should never be used—see the problems at the end of the chapter for more details.

Runtime stack checking can be used to prevent stack smashing attacks. In this approach, when the return address is popped off of the stack, it's checked to verify that it hasn't changed. This can be accomplished by pushing a special value onto the stack immediately after the return address. Then when Trudy attempts to overwrite the return address, she must first overwrite this special value, which provides a means for detecting the attack. This special value is usually known as a *canary*, in reference to the coal miner's canary.[7] The use of a canary for stack smashing detection is illustrated in Figure 11.13.

Note that if Trudy can overwrite an anti-stack-smashing canary with itself, then her attack will go undetected. Can we prevent the canary from being overwritten with itself?

A canary can be a constant, or a value that depends on the return address. A specific constant that is sometimes used is `0x000aff0d`. This constant includes `0x00` as the first byte since this is the string terminating byte. Any string that overflows a buffer and includes `0x00` will be terminated at that point and no more of the stack will be overwritten. Consequently, an attacker can't use a string input to overwrite the constant `0x000aff0d` with itself, and any other value that overwrites the canary will be detected. The other bytes in this constant serve to prevent other types of buffer overflow attacks.

[7] Coal miners would take a canary with them underground into the mine. If the canary died, the coal miners knew there was a problem with the air and they needed to get out of the mine as soon as possible.

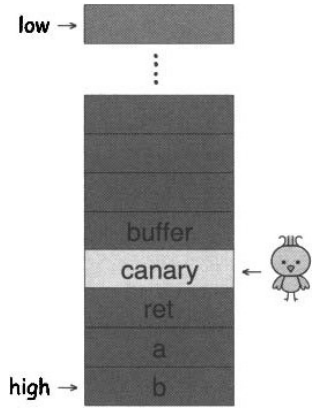


Figure 11.13: Canary

Microsoft recently added a canary feature to its C++ compiler based on the approach discussed in [246]. Any program compiled with the `/GS` compiler flag will use a canary—or, in Microsoft-speak, a "security cookie"—to detect buffer overflows at runtime. But the initial Microsoft implementation was apparently flawed. When the canary died, the program passed control to a user-supplied handler function. It was discovered that an attacker could specify this handler function, thereby executing arbitrary code on the victim machine [245], although the severity of this attack was disputed by Microsoft [187]. Assuming the claimed attack was valid, then all buffer overflows compiled under the `/GS` option were exploitable, even those that would not have been exploitable without the `/GS` option. In other words, the cure was worse than the disease.

Another option for minimizing the effectiveness of buffer overflow attacks is Address Space Layout Randomization, or ASLR [105]. This technique is used in recent Windows operating systems and several other modern OSs. ASLR relies on the fact that buffer overflow attacks are fairly delicate. That is, to execute code on the stack, Trudy usually overwrites the return address with a hard-coded specific address that causes execution to jump to the specified location. When ASLR is used, programs are loaded into more or less random locations in memory, so that any address that Trudy has hard-coded into her attack is only likely to be correct a small percentage of the time. Then Trudy's attack will only succeed a correspondingly small percentage of the time.

However, in practice, only a relatively small number of "random" layouts are used. Vista, for example, uses 256 distinct layouts and, consequently,

a given buffer overflow attacks should have a natural success probability of about 1/256. However, due to a weakness in the implementation, Vista does not choose from these 256 possible layouts uniformly, which results in a significantly greater chance of success for a clever attacker [324]. In addition, a so-called de-randomization attack on certain specific ASLR implementations is discussed in [263].

11.2.1.4 Buffer Overflow: The Last Word

Buffer overflow was unquestionably the attack of the decade for each of the past several decades. For example, buffer overflow has been the enabling vulnerability in many major malware outbreaks. This, in spite of the fact that buffer overflow attacks have been well known since the 1970s, and it's possible to prevent most such attacks by using the NX bit approach and/or safe programming languages and/or ASLR. Even with an unsafe language such as C, buffer overflow attacks can be greatly reduced by using the safer versions of the unsafe functions.

Can we hope to relegate buffer overflow attacks to the scrapheap of history? Developers must be educated, and tools for preventing and detecting buffer overflow conditions must be used. If it's available on a given platform, the NX bit should certainly be employed and ASLR is a very promising technology. Unfortunately, buffer overflows will remain a problem for the foreseeable future because of the large amount of legacy code and older machines that will continue to be in service.

11.2.2 Incomplete Mediation

The C function `strcpy(buffer, input)` copies the contents of the input string `input` to the array `buffer`. As we discovered above, a buffer overflow will occur if the length of `input` is greater than the length of `buffer`. To prevent such a buffer overflow, the program must validate the input by checking the length of `input` before attempting to write it to `buffer`. Failure to do so is an example of *incomplete mediation*.

As a somewhat more subtle example, consider data that is input to a Web form. Such data is often transferred to the server by embedding it in a URL, so that's the method we'll employ here. Suppose the input is validated on the client before constructing the required URL.

For example, consider the following URL:

```
http://www.things.com/orders/final&custID=112&
    num=55A&qty=20&price=10&shipping=5&total=205
```

On the server, this URL is interpreted to mean that the customer with ID number 112 has ordered 20 of item number 55, at a cost of $10 each, with

a $5 shipping charge, giving a total cost of $205. Since the input was checked on the client, the developer of the server software believes it would be wasted effort to check it again on the server.

However, instead of using the client software, Trudy can directly send a URL to the server. Suppose Trudy sends the following URL to the server:

```
http://www.things.com/orders/final&custID=112&
    num=55A&qty=20&price=10&shipping=5&total=25
```

If the server doesn't bother to validate the input, Trudy can obtain the same order as above, but for the bargain basement price of $25 instead of the legitimate price of $205.

Recent research [79] revealed numerous buffer overflows in the Linux kernel, and most of these were due to incomplete mediation. This is perhaps somewhat surprising since the Linux kernel is usually considered to be very good software. After all, it is open source, so anyone can look for flaws in the code (we'll have more to say about this in the next chapter) and it is the kernel, so it must have been written by experienced programmers. If these software flaws are common in such code, they are undoubtedly more common in most other code.

There are tools available to help find likely cases of incomplete mediation. These tools should be more widely used, but they are not a cure-all since this problem can be subtle, and therefore difficult to detect automatically. As with most security tools, these tools can also be useful for the bad guys.

11.2.3 Race Conditions

Ideally, security processes should be *atomic*, that is, they should occur all at once. So-called race conditions can arise when a security-critical process occurs in stages. In such cases, an attacker may be able to make a change between the stages and thereby break the security. The term race condition refers to a "race" between the attacker and the next stage of the process, although it's not so much a race as a matter of careful timing for the attacker.

The race condition that we'll consider occurs in an outdated version of the Unix command `mkdir`, which creates a new directory. With this version of `mkdir`, the directory is created in stages—there is a stage that determines authorization followed by a stage that transfers ownership. If Trudy can make a change after the authorization stage but before the transfer of ownership, then she can, for example, become the owner of some directory that she should not be able to access.

The way that this version of `mkdir` is supposed to work is illustrated in Figure 11.14. Note that `mkdir` is not atomic and that is the source of the race condition.

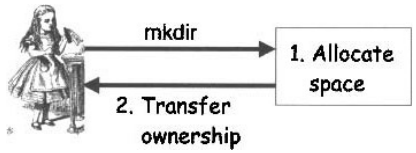


Figure 11.14: How `mkdir` is Supposed to Work

Trudy can exploit this particular `mkdir` race condition if she can somehow implement the attack that is illustrated in Figure 11.15. In this attack scenario, after the space for the new directory is allocated, a link is established from the the password file (which Trudy is not authorized to access) to this newly created space, before ownership of the new directory is transferred to Trudy. Note that this attack is not really a race, but instead it requires careful (or lucky) timing by Trudy.

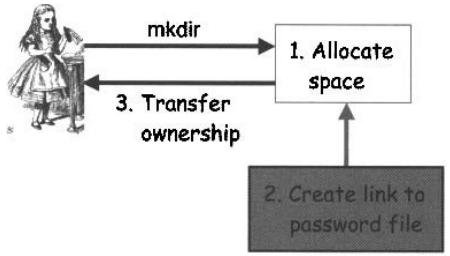


Figure 11.15: Attack on `mkdir` Race Condition

Today, race conditions are probably fairly common and with the trend towards increased parallelism, they are sure to become even more prevalent. However, real-world attacks based on race conditions are rare—attackers clearly favor buffer overflows.

Why are attacks based on race conditions a rarity? For one thing, exploiting a race condition requires careful timing. In addition, each race condition is unique, so there is no standard formula for such an attack. In comparison to, say, buffer overflow attacks, race conditions are certainly more difficult to exploit. Consequently, as of today buffer overflows are the low hanging fruit and are therefore favored by attackers. However, if the number of buffer overflows is reduced, or buffer overflows are made sufficiently difficult to exploit, it's a safe bet that we will see a corresponding increase in attacks based on race conditions. This is yet another illustration of Stamp's Principle: there is job security in security.

11.3 Malware

Solicitations malefactors!
— Plankton

In this section, we'll discuss software that is designed to break security. Since such software is malicious in its intent, it goes by the name of *malware*. Here, we mostly just cover the basics—for more details, the place to start is Aycock's fine book [21].

Malware can be subdivided into many different categories. We'll use the following classification system, although there is considerable overlap between the various types.

- A *virus* is malware that relies on someone or something else to propagate from one system to another. For example, an email virus attaches itself to an email that is sent from one user to another. Until recently, viruses were the most popular form of malware.[8]
- A *worm* is like a virus except that it propagates by itself without the need for outside assistance. This definition implies that a worm uses a network to spread its infection.
- A *trojan horse*, or trojan, is software that appears to be one thing but has some unexpected functionality. For example, an innocent-looking game could do something malicious while the victim is playing.
- A *trapdoor* or *backdoor* allows unauthorized access to a system.
- A *rabbit* is a malicious program that exhausts system resources. Rabbits could be implemented using viruses, worms, or other means.
- *Spyware* is a type of malware that monitors keystrokes, steals data or files, or performs some similar function [22].

Generally, we won't be too concerned with placing a particular piece of malware into its precise category. We'll use the term virus as shorthand for a virus, worm, or other such malware. It is worth noting that many "viruses" (in popular usage of the term) are not viruses in the technical sense.

Where do viruses live on a system? It should come as no surprise that *boot sector* viruses live in the boot sector, where they are able to take control early in the boot process. Such a virus can then take steps to mask its presence before it can be detected. From a virus writer's perspective, the boot sector is a good place to be.

[8]The term "virus" is sometimes reserved for parasitic malware, that is, malware that relies on other code to perform its intended function.

Another class of viruses are *memory resident*, meaning that they stay in memory. Rebooting the system may be necessary to flush these viruses out. Viruses also can live in applications, macros, data, library routines, compilers, debuggers, and even in virus checking software.

By computing standards, malware is ancient. The first substantive work on viruses was done by Fred Cohen in the 1980s [62], who clearly demonstrated that malware could be used to attack computer systems.[9]

Arguably, the first virus of any significance to appear in the wild was the so-called Brain virus of 1986. Brain did nothing malicious, and it was considered little more than a curiosity. As a result, it did not awaken people to the security implications of malware. That complacency was shaken in 1988 when the Morris Worm appeared. In spite of its early date, the Morris Worm remains one of the more interesting pieces of malware to date, and we'll have more to say about it below. The other examples of malware that we'll discuss in some detail are Code Red, which appeared in 2001, and SQL Slammer, which appeared in January of 2003. We'll also present a simple example of a trojan and we'll discuss the future of malware. For more details on many aspects of malware—including good historical insights—see [66].

11.3.1 Brain

The Brain virus of 1986 was more annoying than harmful. Its importance lies in the fact that it was first, and as such it became a prototype for many later viruses. But because it was not malicious, there was little reaction by users. In retrospect, Brain provided a clear warning of the potential for malware to cause damage, but at the time that warning was mostly ignored. In any case, computing systems remained extremely vulnerable to malware.

Brain placed itself in the boot sector and other places on the system. It then screened all disk access so as to avoid detection and to maintain its infection. Each time the disk was read, Brain would check the boot sector to see if it was infected. If not, it would reinstall itself in the boot sector and elsewhere. This made it difficult to completely remove the virus. For more details on Brain, see Chapter 7 of Robert Slade's excellent history of viruses [66].

11.3.2 Morris Worm

Information security changed forever when the eponymous Morris Worm attacked the Internet in 1988 [37, 229]. It's important to realize that the Internet of 1988 was nothing like the Internet of today. Back then, the Internet was populated by academics who exchanged email and used `telnet` for remote

[9]Cohen credited Len Adleman (the "A" in RSA) with coining the term "virus."

access to supercomputers. Nevertheless, the Internet had reached a critical mass that made it vulnerable to self-sustaining worm attacks.

The Morris Worm was a cleverly designed and sophisticated piece of software that was written by a lone graduate student at Cornell University.[10] Morris claimed that his worm was a test gone bad. In fact, the most serious consequence of the worm was due to a flaw (according to Morris). In other words, the worm had a bug.

The Morris Worm was apparently supposed to check whether a system was already infected before trying to infect it. But this check was not always done, and so the worm tried to re-infect already infected systems, which led to resource exhaustion. So the (unintended) malicious effect of the Morris Worm was essentially that of a so-called rabbit.

Morris' worm was designed to do the following three things.

- Determine where it could spread its infection
- Spread its infection wherever possible
- Remain undiscovered

To spread its infection, the Morris worm had to obtain remote access to machines on the network. To gain access, the worm attempted to guess user account passwords. If that failed, it tried to exploit a buffer overflow in `fingerd` (part of the Unix `finger` utility), and it also tried to exploit a trapdoor in `sendmail`. The flaws in `fingerd` and `sendmail` were well known at the time but not often patched.

Once access had been obtained to a machine, the worm sent a bootstrap loader to the victim. This loader consisted of 99 lines of C code that the victim machine compiled and executed. The bootstrap loader then fetched the rest of the worm. In this process, the victim machine even authenticated the sender.

The Morris worm went to great lengths to remain undetected. If the transmission of the worm was interrupted, all of the code that had been transmitted was deleted. The code was also encrypted when it was downloaded, and the downloaded source code was deleted after it was decrypted and compiled. When the worm was running on a system, it periodically changed its name and process identifier (PID), so that a system administrator would be less likely to notice anything unusual.

It's no exaggeration to say that the Morris Worm shocked the Internet community of 1988. The Internet was supposed to be able to survive a nuclear attack, yet it was brought to its knees by a graduate student and a few

[10] As if to add a conspiratorial overtone to the the entire affair, Morris' father worked at the super-secret National Security Agency at the time [248].

hundred lines of C code. Few, if any, had imagined that the Internet was so vulnerable to such an attack.

The results would have been much worse if Morris had chosen to have his worm do something truly malicious. In fact, it could be argued that the greatest damage was caused by the widespread panic the worm created—many users simply pulled the plug, believing it to be the only way to protect their system. Those who stayed online were able to receive some information and therefore recovered more quickly than those who chose to rely on the infallible "air gap" firewall.

As a direct result of the Morris Worm, the Computer Emergency Response Team (CERT) [51] was established, which continues to be a primary clearinghouse for timely computer security information. While the Morris Worm did result in increased awareness of the vulnerability of the Internet, curiously, only limited actions were taken to improve security. This event should have served as a wakeup call and could well have led to a complete redesign of the security architecture of the Internet. At that point in history, such a redesign effort would have been relatively easy, whereas today it is completely infeasible. In that sense, the Morris Worm can be seen as a missed opportunity.

After the Morris Worm, viruses became the mainstay of malware writers. Only relatively recently have worms reemerged in a big way. Next, we'll consider two worms that indicate some of the trends in malware.

11.3.3 Code Red

When Code Red appeared in July of 2001, it infected more than 300,000 systems in about 14 hours. Before Code Red had run its course, it infected several hundred thousand more, out of an estimated 6,000,000 susceptible systems worldwide. To gain access to a system, the Code Red worm exploited a buffer overflow in Microsoft IIS server software. It then monitored traffic on port 80, looking for other potential targets.

The action of Code Red depended on the day of the month. From day 1 to 19, it tried to spread its infection, then from day 20 to 27 it attempted a distributed denial of service (DDoS) attack on `www.whitehouse.gov`. There were many copycat versions of Code Red, one of which included a trapdoor for remote access to infected systems. After infection, this variant flush all traces of the worm, leaving only the trapdoor.

The speed at which Code Red infected the network was something new and, as a result, it generated a tremendous amount of hype [72]. For example, it was claimed that Code Red was a "beta test for information warfare" [235]. However, there was (and still is) no evidence to support such claims or any of the other general hysteria that surrounded the worm.

11.3.4 SQL Slammer

The SQL Slammer worm burst onto the scene in January of 2003, when it infected at least 75,000 systems within 10 minutes. At its peak, the number of Slammer infections doubled every 8.5 seconds [209].

The graphs in Figure 11.16 show the increase in Internet traffic as a result of Slammer. The graph on the bottom shows the increase over a period of hours (note the initial spike), while the graph on the top shows the increase over the first five minutes.

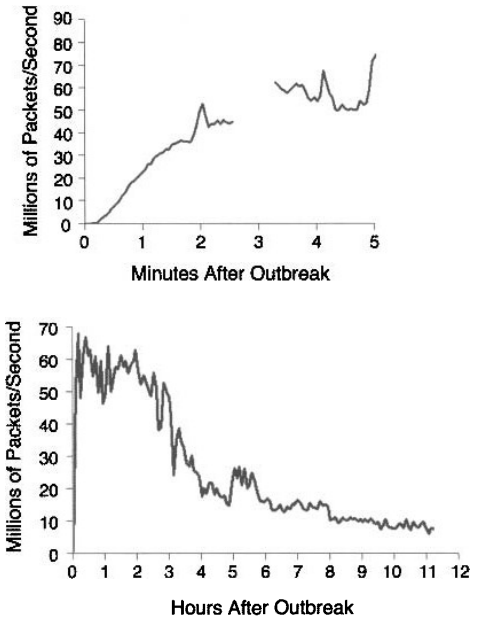


Figure 11.16: Slammer and Internet Traffic

The reason that Slammer created such a spike in Internet traffic is that each infected site searched for new susceptible sites by randomly generating IP addresses. A more efficient search strategy would have made more effective use of the available bandwidth. We'll return to this idea below when we discuss the future of malware.

It's been claimed (with good supporting evidence) that Slammer spread too fast for its own good, and effectively burned out the available bandwidth on the Internet [92]. In other words, if Slammer had been able to throttle

itself slightly, it could have ultimately infected more systems and it might have caused significantly more damage.

Why was Slammer so successful? For one thing, the entire worm fit into a single 376-byte UDP packet. Firewalls are often configured to let sporadic packets through, on the theory that a single small packet can do no harm by itself. The firewall then monitors the "connection" to see whether anything unusual occurs. Since it was generally expected that much more that 376 bytes would be required for an attack, Slammer succeeded in large part by defying the assumptions of the security experts.

11.3.5 Trojan Example

In this section, we'll present a trojan, that is, a program that has some unexpected function. This trojan comes from the Macintosh world, and it's totally harmless, but its creator could just as easily have had this program do something malicious [103]. In fact, the program could have done anything that a user who executed the program could do.

This particular trojan appears to be audio data, in the form of an mp3 file that we'll name `freeMusic.mp3`. The icon for this file appears in Figure 11.17. A user would expect that double clicking on this file would automatically launch iTunes, and play the music contained in the mp3 file.

Figure 11.17: Icon for `freeMusic.mp3`

After double-clicking on the icon in Figure 11.17, iTunes launches (as expected) and an mp3 file titled "Wild Laugh" is played (probably not expected). Simultaneously, and unexpectedly, the message window in Figure 11.18 appears.

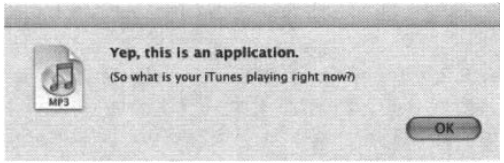


Figure 11.18: Unexpected Effect of `freeMusic.mp3` Trojan

What just happened? This "mp3" file is a wolf in sheep's clothing—the file `freeMusic.mp3` is not an mp3 file at all. Instead it's an application (that

is, an executable file) that has had its icon changed so that it appears to be an mp3 file. A careful look at `freeMusic.mp3` reveals this fact, as shown in Figure 11.19.

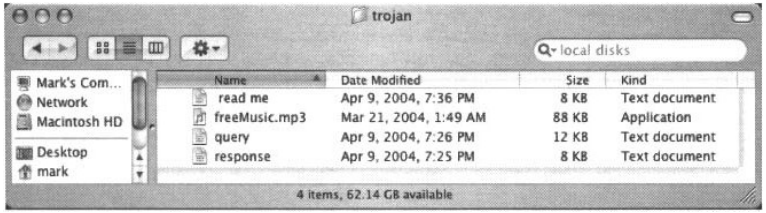


Figure 11.19: Trojan Revealed

Most users are unlikely to give a second thought to opening a file that appears to be an mp3. This trojan only issues a harmless warning, but that's because the author had no malicious intent and instead simply wanted to illustrate a point [160].

11.3.6 Malware Detection

There are three general approaches that are used to detect malware. The first, and most common, is *signature detection*, which relies on finding a pattern or signature that is present in a particular piece of malware. A second approach is *change detection*, which detects files that have changed. A file that has unexpectedly changed might indicate an infection. The third approach is *anomaly detection*, where the goal is to detect unusual or virus-like files or behavior. We'll briefly discuss each of these approaches and consider their relative advantages and disadvantages.

In Chapter 8, we discussed signature-based and anomaly-based intrusion detection systems (IDSs). There are many parallels between IDSs and the corresponding virus detection methods.

11.3.6.1 Signature Detection

A signature is generally a string of bits found in a file, which might include wildcards. A hash value could also serve as a signature, but it would be less flexible and easier for virus writers to defeat.

For example, according to [296], the signature used for the W32/Beast virus is `83EB 0274 EB0E 740A 81EB 0301 0000`. We can search for this signature in all files on a system. However, if we find the signature, we can't be certain that we've found the virus, since other innocent files could contain the same string of bits. If the bits in searched files were random, the chance of such a false match would be $1/2^{112}$, which is negligible. However, computer

software and data is far from random, so there is probably some realistic chance of a false match. This means that if a matching signature is found, further testing may be required to be certain that it actually represents the W32/Beast virus.

Signature detection is highly effective on malware that is known and for which a common signature can be extracted. Another advantage of signature detection is that it places a minimal burden on users and administrators, since all that is required is to keep signature files up to date and periodically scan for viruses.

A disadvantage of signature detection is that signature files can become large—tens or hundreds of thousands of signatures is the norm—which can make scanning slow. Also, the signature files must be kept up to date. A more fundamental problem is that we can only detect known signatures. Even a slight variant of a known virus might be missed.

Today, signature detection is by far the most popular malware detection method. As a result, virus writers have developed some sophisticated means for avoiding signature detection. We'll have more to say about this below.

11.3.6.2 Change Detection

Since malware must reside somewhere, if we detect a change somewhere on a system, then it may indicate an infection. That is, if we detect that a file has changed, it may be infected with a virus. We'll refer to this approach as change detection.

How can we detect changes? Hash functions are useful in this regard. Suppose we compute hashes of all files on a system and securely store these hash values. Then at regular intervals we can recompute the hashes and compare the new values with the stored values. If a file has changed in one or more bits—as it will in the case of a virus infection—we'll find that the computed hash does not match the previously computed hash value.

One advantage of change detection is that there are virtually no false negatives, that is, if a file has been infected, we'll detect a change. Another major advantage is that we can detect previously unknown malware (a change is a change, whether it's caused by a known or unknown virus).

However, the disadvantages to change detection are many. Files on a system often change and as a result there will be many false positives, which places a heavy burden on users and administrators. If a virus is inserted into a file that changes often, it will be more likely to slip through a change detection regimen. And what should be done when a suspicious change is detected? A careful analysis of log files might prove useful. But, in the end, it might be necessary to fall back to a signature scan, in which case the advantages of change detection have been largely negated.

11.3.6.3 Anomaly Detection

Anomaly detection is aimed at finding any unusual or virus-like or other potentially malicious activity or behavior. We discussed this idea in detail Chapter 8 when we covered intrusion detection systems (IDSs), so we only briefly discuss the concepts here.

The fundamental challenge with anomaly detection lies in determining what is normal and what is unusual, and being able to distinguish between the two. Another serious difficulty is that the definition of normal can change, and the system must adapt to such changes, or it will likely overwhelm users with false alarms.

The major advantage of anomaly detection is that there is some hope of detecting previously unknown malware. But, as with change detection, the disadvantages are many. For one, anomaly detection is largely unproven in practice. Also, as discussed in the IDS section of Chapter 8, a patient attacker may be able to make an anomaly appear to be normal. In addition, anomaly detection is not robust enough to be used as a standalone detection system, so it is usually combined with a signature detection system.

In any case, many people have very high hopes for the ultimate success of anomaly detection. However, today anomaly detection is primarily a challenging research problem rather than a practical security solution.

Next, we'll discuss some aspects of the future of malware. This discussion should make it clear that better malware detection tools will be needed, and sooner rather than later.

11.3.7 The Future of Malware

What does the future hold for malware? Below, we'll briefly consider a few possible attacks. Given the resourcefulness of malware developers, we can expect to see attacks based on these or similar ideas in the future [24, 289].

But before we discuss the future, let's briefly consider the past. Virus writers and virus detectors have been locked in mortal combat since the first virus detection software appeared. For each advance in detection, virus writers have responded with strategies that make their handiwork harder to detect.

One of the first responses of virus writers to the success of signature detection systems was *encrypted* malware. If an encrypted worm uses a different key each time it propagates, there will be no common signature. Often the encryption is extremely weak, such as a repeated XOR with a fixed bit pattern. The purpose of the encryption is not confidentiality, but to simply mask any possible signature.

The Achilles heel of encrypted malware is that it must include decryption code, and this code is subject to signature detection. The decryption routine typically includes very little code, making it more difficult to obtain

a signature, and yielding more cases requiring secondary testing. The net result is that signature scanning can be applied, but it will be slower than for unencrypted malware.

The next step in the evolution of malware was the use of *polymorphic* code. In a polymorphic virus the body is encrypted and the decryption code is morphed. Consequently, the signature of the virus itself (i.e., the body) is hidden by encryption, while the decryption code has no common signature due to the morphing.

Polymorphic malware can be detected using emulation. That is, suspicious code can be executed in an emulator. If the code is malware, it must eventually decrypt itself, at which point standard signature detection can be applied to the body. This type of detection will be much slower than a simple signature scan due to the emulation.

Metamorphic malware takes polymorphism to the limit. A metamorphic worm mutates before infecting a new system.[11] If the mutation is sufficient, such a worm can likely avoid any signature-based detection system. Note that the mutated worm must do the same thing as the original worm, but yet its internal structure must be different enough to avoid detection. Detection of metamorphic software is currently a challenging research problem [297].

Let's consider how a metamorphic worm might replicate [79]. First, the worm could disassemble itself and then strip the resulting code to a base form. Randomly selected blocks of code could be inserted into the assembly. These variations could include, for example, rearranging jumps and inserting dead code. The resulting code would then be assembled to obtain a worm with the same functionality as the original, but it would be unlikely to have a common signature.

While the metamorphic generator described in the previous paragraph sounds plausible, in reality it is surprisingly difficult to produce highly metamorphic code. As of the time of this writing, the hacker community has produce a grand total of one reasonably metamorphic generator. These and related topics are discussed in the series of papers [193, 279, 312, 330].

Another distinct approach that virus writers have pursued is speed. That is, viruses such as Code Red and Slammer have tried to infect as many machines as possible in as short of a time as possible. This can also be viewed as an attack aimed at defeating signature detection, since a rapid attack would not allow time for signatures to be extracted and distributed.

According to the late pop artist Andy Warhol, "In the future everybody will be world-famous for 15 minutes" [301]. A *Warhol worm* is designed to infect the entire Internet in 15 minutes or less. Recall that Slammer infected a large number of systems in 10 minutes. Slammer burned out the available

[11] Metamorphic malware is sometimes called "body polymorphic," since polymorphism is applied to the entire virus body.

bandwidth due to the way that it searched for susceptible hosts, and as a result, Slammer was too bandwidth-intensive to have infected the entire Internet in 15 minutes. A true Warhol worm must do "better" than Slammer. How is this possible?

One plausible approach is the following. The malware developer would do preliminary work to develop an initial "hit list" of sites that are susceptible to the particular exploit used by the worm. Then the worm would be seeded with this hit list of vulnerable IP addresses. Many sophisticated tools exist for identifying systems and these could help to pinpoint systems that are susceptible to a given attack.

When this Warhol worm is launched, each of the sites on its initial hit list will be infected since they are all known to be vulnerable. Then each of these infected sites can scan a predetermined part of IP address space looking for additional victims. This approach would avoid duplication and the resulting wasted bandwidth that caused Slammer to bog down.

Depending on the size of the initial hit list, the approach described above could conceivably infect the entire Internet in 15 minutes or less. No worm this sophisticated has yet been seen in the wild. Even Slammer relied on randomly generated IP addresses to spread its infection.

Is it possible to do "better" than a Warhol worm? That is, can the entire Internet be infected in significantly less than 15 minutes? A *flash worm* is designed to infect the entire Internet almost instantly.

Searching for vulnerable IP addresses is the slow part of any worm attack. The Warhol worm described above uses a smarter search strategy, where it relies on an initial list of susceptible systems. A flash worm could take this approach to the limit by embedding all susceptible IP addresses into the worm.

A great deal of work would be required to predetermine all vulnerable IP addresses, but there are hacker tools available that would significantly reduce the burden. Once all vulnerable IP addresses are known, the list could be partitioned between several initial worm variants. This would still result in large worms [79], but each time the worm replicates, it would split the list of addresses embedded within it, as illustrated in Figure 11.20. Within a few generations the worm would be reduced to a reasonable size. The strength of this approach is that it results in virtually no wasted time or bandwidth.

It has been estimated that a well-designed flash worm could infect the entire Internet in as little as 15 seconds! Since this is much faster than humans could possibly respond, any defense against such an attack must be automated. A conjectured defense against flash worms [79] would be to deploy many personal intrusion detection systems and to have a master IDS monitor these personal IDSs. When the master IDS detects unusual activity, it can let it proceed on a few nodes, while temporarily blocking it elsewhere. If the sacrificial nodes are adversely affected, then an attack is in progress,

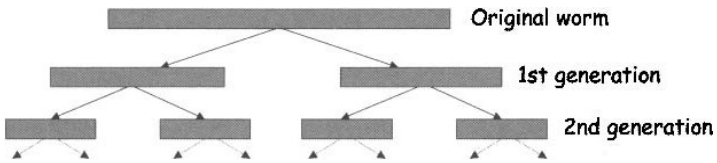


Figure 11.20: A Flash Worm

and it can be blocked elsewhere. On the other hand, if it's a false alarm, the other nodes are only delayed slightly. This defensive strategy shares many of the challenges associated with anomaly-based intrusion detection systems, as discussed in Chapter 8.

11.3.8 Cyber Diseases Versus Biological Diseases

It's currently fashionable to make biological analogies with computing. There are many such analogies that are applied to the field of security. In the field of malware and in particular, computer viruses, the analogy is fairly obvious.

There clearly are similarities between biological and computer "diseases." For example, in nature, if there are too few susceptible individuals, a disease will die out. A somewhat similar situation exists on the Internet, where too few susceptible systems may not allow a worm to become self-sustaining, particularly if the worm is randomly searching for vulnerable IP addresses.

There are, however, some significant differences between cyber diseases and biological diseases. For example, there is virtually no sense of distance on the Internet, so many of the models developed for biological diseases don't apply to cyber diseases.[12] Also, in nature, diseases attack more or less at random, while in computer systems hackers often specifically target the most desirable or vulnerable systems. As a result, computer attacks are potentially more focused and damaging than biological diseases. The important point here is that, although the biological analogy is useful, it cannot be taken too literally.

Finally, we note in passing that cell phones have not been plagued with malware to nearly the same degree as computer systems. Various explanations for this phenomenon have been given, with two of the more plausible being the relative diversity of mobile systems and inherently stronger security architectures. For a discussion of the Android security architecture and some of the difficulties of mounting a successful attack, see [211].

[12]However, with some cell phone attacks, proximity is required (e.g., attacks that rely on Bluetooth) while network-based attacks are also possible. So, cell phone attacks could include aspects of both biological viruses and computer viruses.

11.4 Botnets

A *botnet* is a collection of a large number of compromised machines under the control of a *botmaster*. The name derives from the fact that individual compromised machines are known as *bots* (shorthand for robots). In the past, such machines were often known as zombies.

Until recently, botmasters typically employed the Internet Relay Chat (IRC) protocol to manage their bots. However, newer botnets often use Peer-to-Peer (P2P) architectures since these are more difficult for authorities to track and shut down.

Botnets have proven ideal tools for sending spam and for launching distributed denial of service (DDoS) attacks. For example, a botnet was used in a highly-publicized denial of service attack on Twitter that was apparently aimed at silencing one well-known blogger from the Republic of Georgia [207].[13]

Botnets are a hot security topic, but at this point in time their activities in the wild are not completely understood. For example, there are wildly differing estimates for the sizes of various botnets [224].

Finally, it is often claimed that in the past most attacks were conducted primarily for fame within the hacker community, or for ideological reasons, or by script kiddies with little knowledge of what they were actually doing. That is, attacks were essentially just malicious pranks. In contrast (or so the claim goes), today attacks are primarily for profit. Some even believe that organized crime is behind most current attacks. The profit motive is plausible since earlier widespread attacks (Code Red, Slammer, etc.) were first and foremost designed to make headlines, whereas botnets strive to remain undetected. In addition, botnets are ideal for use in various subtle attack-for-hire scenarios. Of course, you should always be skeptical of those who hype any supposed threat, especially when they have a vested interest in the hype becoming conventional wisdom.[14]

11.5 Miscellaneous Software-Based Attacks

In this section we'll consider a few software-based attacks that don't fit neatly into any of our previous discussion. While there are numerous such attacks, we'll restrict our attention to a few representative examples. The topics we'll discuss are *salami attacks*, *linearization attacks*, *time bombs*, and the general issue of trusting software.

[13]Of course, this raised suspicion that Russian government intelligence agencies were behind the attack. However, the attack accomplished little, other than greatly increasing the fame of the attackee, so it's difficult to believe that any intelligence agency would be so stupid. On the other hand, "government intelligence" is an oxymoron.

[14]Or, more succinctly, "Beware the prophet seeking profit" [205].

11.5.1 Salami Attacks

In a salami attack, a programmer slices off a small amount of money from individual transactions, analogous to the way that you might slice off thin pieces from a salami.[15] These slices must be difficult for the victim to detect. For example, it's a matter of computing folklore that a programmer at a bank can use a salami attack to slice off fractional cents leftover from interest calculations. These fractional cents—which are not noticed by the customers or the bank—are deposited in the programmer's account. Over time, such an attack could prove highly lucrative for the dishonest programmer.

There are many confirmed cases of salami attacks. The following examples all appear in [158]. In one documented case, a programmer added a few cents to every employee payroll tax withholding calculation, but credited the extra money to his own tax. As a result, this programmer got a hefty tax refund. In another example, a rent-a-car franchise in Florida inflated gas tank capacity so it could overcharge customers for gas. An employee at a Taco Bell location reprogrammed the cash register for the late-night drive-through line so that $2.99 specials registered as $0.01. The employee then pocketed the $2.98 difference—a rather large slice of salami!

In a particularly clever salami attack, four men who owned a gas station in Los Angeles hacked a computer chip so that it overstated the amount of gas pumped. Not surprisingly, customers complained when they had to pay for more gas than their tanks could hold. But this scam was hard to detect, since the gas station owners were clever. They had programmed the chip to give the correct amount of gas whenever exactly 5 or 10 gallons was purchased, because they knew from experience that inspectors usually ask for 5 or 10 gallons. It took multiple inspections before they were caught.

11.5.2 Linearization Attacks

Linearization is an approach that is applicable in a wide range of attacks, from traditional lock picking to state-of-the-art cryptanalysis. Here, we consider an example related to breaking software, but it is important to realize that this concept has wide application.

Consider the program in Table 11.5, which checks an entered number to determine whether it matches the correct serial number. In this case, the correct serial number happens to be S123N456. For efficiency, the programmer decided to check one character at a time and to quit checking as soon as one incorrect character is found. From a programmer's perspective, this is a perfectly reasonable way to check the serial number, but it might open the door to an attack.

[15]Or the name might derive from the fact that a salami consists of bunch of small undesirable pieces that are combined to yield something of value.

Table 11.5: Serial Number Program

```
int main(int argc, const char *argv[])
{
   int i;
   char serial[9]="S123N456\n";
   if(strlen(argv[1]) < 8)
   {
      printf("\nError---try again.\n\n");
      exit(0);
   }
   for(i = 0; i < 8; ++i)
   {
      if(argv[1][i] != serial[i]) break;
   }
   if(i == 8)
   {
      printf("\nSerial number is correct!\n\n");
   }
}
```

How can Trudy take advantage the code in Table 11.5? Note that the correct serial number will take longer to process than any incorrect serial number. More precisely, the more leading characters that are correct, the longer the program will take to check the number. So, a putative serial number that has the first character correct will take longer than any that has an incorrect first character. Therefore, Trudy can select an eight-character string and vary the first character over all possibilities. If she can time the program precisely enough, she will find that the string beginning with S takes the most time. Trudy can then fix the first character as S and vary the second character, in which case she will find that a second character of 1 takes the longest. Continuing, Trudy can recover the serial number one character at a time. That is, Trudy can attack the serial number in linear time, instead of searching an exponential number of cases.

How great is the advantage for Trudy in this linearization attack? Suppose the serial number is eight characters long and each character has 128 possible values. Then there are $128^8 = 2^{56}$ possible serial numbers. If Trudy must randomly guess complete serial numbers, she would obtain the serial number in about 2^{55} tries, which is an enormous amount of work. On the other hand, if she can use a linearization attack, an average of only $128/2 = 64$ guesses

are required for each letter, for a total expected work of about $8 \cdot 64 = 2^9$. This makes an otherwise infeasible attack into a trivial attack.

A real-world example of a linearization attack occurred in TENEX [235], a timeshare system used in ancient times.[16] In TENEX, passwords were verified one character at a time, so the system was subject to a linearization attack similar to the one described above. However, careful timing was not even necessary. Instead, it was possible to arrange for a "page fault" to occur when the next unknown character was guessed correctly. Then a user-accessible page fault register would tell the attacker that a page fault had occurred and, therefore, that the next character had been guessed correctly. This attack could be used to crack any password in seconds.

11.5.3 Time Bombs

Time bombs are another interesting class of software-based attacks. We'll illustrate the concept with an infamous example. In 1986, Donald Gene Burleson told his employer to stop withholding taxes from his paycheck. Since this isn't legal, the company refused. Burleson, a tax protester, made it known that he planned to sue his company. Burleson used company time and resources to prepare his legal case against his company. When the company discovered what Burleson was doing, they fired him [240].

It later came to light that Burleson had been developing malicious software. After he was fired, Burleson triggered his "time bomb" software, which proceeded to delete thousands of records from the company's computer.

The Burleson story doesn't end here. Out of fear of embarrassment, the company was reluctant to pursue a legal case, despite their losses. Then in a bizarre twist, Burleson sued his former employer for back pay, at which point the company finally sued Burleson. The company eventually won, and in 1988 Burleson was fined $11,800. The case took two years to prosecute at a cost of tens of thousands of dollars and resulted in little more than a slap on the wrist. The light sentence was likely due to the fact that laws regarding computer crime were unclear at that early date. In any case, this was one of the first computer crime cases in the United States, and many cases since have followed a similar pattern. In particular, companies are often reluctant to pursue such cases for fear that it will damage their reputation.

11.5.4 Trusting Software

Finally, we consider a philosophical question with practical significance: Can you ever trust software? In the fascinating article [303], the following thought experiment is discussed. Suppose that a C compiler has a virus. When

[16] The 1960s and 1970s, that is. In computing, that's the age when dinosaurs roamed the earth.

compiling the `login` program, this virus creates a backdoor in the form of an account with a known password. Also, if the C compiler is recompiled, the virus incorporates itself into the newly compiled C compiler.

Now suppose that you suspect that your system is infected with a virus. You want to be absolutely certain that you fix the problem, so you decide to start over from scratch. You recompile the C compiler, then use it to recompile the operating system, which includes the `login` program. You haven't gotten rid of the problem, since the backdoor was once again compiled into the `login` program.

Analogous situations could arise in the real world. For example, imagine that an attacker is able to hide a virus in your virus scanning software. Or consider the damage that could be done by a successful attack on online virus signature updates—or other automated software updates.

Software-based attacks might not be obvious, even to an expert who examines the source code line by line. For example, in the Underhanded C Contest, the rules state in part that [70]

> ...in this contest you must write code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform at its apparent function. To be more specific, it should do something subtly evil.

Some of the programs submitted to this contest are extremely subtle and they demonstrate that it is possible to make evil code look innocent.

We'll return to the theme of trusting software when we discuss operating systems in Chapter 13. Specifically, we will outline an ambitious design for a trusted operating system.

11.6 Summary

In this chapter, we discussed some of the security threats that arise from software. The threats considered here come in two basic flavors. The plain vanilla flavor consists of unintentional software flaws that attackers can sometimes exploit. The classic example of such a flaw is the buffer overflow, which we discussed in some detail. Another common flaw with security implications is a race condition.

The more exotic flavor of software security threats arise from intentionally malicious software, or malware. Such malware includes the viruses and worms that plague users today, as well as trojans and backdoors. Malware writers have developed highly sophisticated techniques for avoiding detection, and they appear set to push the envelope much further in the near future. Whether detection tools are up to the challenge posed by the next generation of malware is an open question.