Chapter 13
Authentication

ANTIPHOLUS OF SYRACUSE: To me she speaks; she moves me for her theme!
What, was I married to her in my dream?
Or sleep I now and think I hear all this?
What error drives our eyes and ears amiss?
Until I know this sure uncertainty,
I'll entertain the offer'd fallacy
— *The Comedy of Errors*, II, ii, 185–190.

Authentication is the binding of an identity to a principal. Network-based authentication mechanisms require a principal to authenticate to a single system, either local or remote. The authentication is then propagated. This chapter explores the question of authentication to a single system.

13.1 Authentication Basics

Subjects act on behalf of some other, external entity. The identity of that entity controls the actions that its associated subjects may take. Hence, the subjects must bind to the identity of that external entity.

Definition 13–1. *Authentication* is the binding of an identity to a subject.

The external entity must provide information to enable the system to confirm its identity. This information comes from one (or more) of the following:

1. What the entity knows (such as passwords or secret information)
2. What the entity has (such as a badge or card)
3. What the entity is (such as fingerprints or retinal characteristics)
4. Where the entity is (such as in front of a particular terminal)

The authentication process consists of obtaining the authentication information from an entity, analyzing the data, and determining if it is associated with that entity. This means that the computer must store some information about the entity. It also suggests that mechanisms for managing the data are required. We represent these requirements in an *authentication system* [221] consisting of five components:

1. The set $\mathcal{A}$ of authentication information is the set of specific information with which entities prove their identities.
2. The set $\mathcal{C}$ of complementary information is the set of information that the system stores and uses to validate the authentication information.
3. The set $\mathcal{F}$ of complementation functions that generate the complementary information from the authentication information. That is, for $f \in \mathcal{F}$, $f : \mathcal{A} \rightarrow \mathcal{C}$.
4. The set $\mathcal{L}$ of authentication functions that verify identity. That is, for $l \in \mathcal{L}$, $l : \mathcal{A} \times \mathcal{C} \rightarrow \{\textbf{true}, \textbf{false}\}$.
5. The set $\mathcal{S}$ of selection functions that enable an entity to create or alter the authentication and complementary information.

EXAMPLE: A user authenticates himself by entering a password, which the system compares with the cleartext passwords stored online. Here, $\mathcal{A}$ is the set of strings making up acceptable passwords, $\mathcal{C} = \mathcal{A}$, $\mathcal{F} = \{I\}$, and $\mathcal{L} = \{\textbf{eq}\}$, where I is the identity function and **eq** is **true** if its arguments are the same and **false** if they are not.

13.2 Passwords

Passwords are an example of an authentication mechanism based on what people know: the user supplies a password, and the computer validates it. If the password is the one associated with the user, that user's identity is authenticated. If not, the password is rejected and the authentication fails.

Definition 13–2. A *password* is information associated with an entity that confirms the entity's identity.

The simplest password is some sequence of characters. In this case, the *password space* is the set of all sequences of characters that can be passwords.

EXAMPLE: Suppose an installation requires each user to choose a sequence of 10 digits as a password. Then $\mathcal{A}$ has 10^{10} elements (from "0000000000" to "9999999999").

The set of complementary information may contain more, or fewer, elements than $\mathcal{A}$, depending on the nature of the complementation function. Originally, most systems stored passwords in protected files. However, the contents of such files might be accidentally exposed. Morris and Thompson [1380] recount an amusing example in which a Multics system editor swapped pointers to the temporary files being used to edit the password file and the message of the day file (printed whenever a user logged in); the result was that whenever a user logged in, the cleartext password file was printed. The solution is to use a one-way hash function to hash the password into a complement [2006].

EXAMPLE: The original UNIX password mechanism does not store the passwords online in the clear. Instead, one of 4,096 functions hashes the password into an 11-character string, and two characters identifying the function used are prepended [1380]. The 13-character string is then stored in a file.

A UNIX password is composed of up to eight ASCII characters; for implementation reasons, the ASCII NUL (0) character is disallowed. Hence, $\mathcal{A}$ is the set of strings of up to eight characters, each chosen from a set of 127 possible characters.[1] $\mathcal{A}$ contains approximately 6.9×10^{16} passwords. However, the set $\mathcal{C}$ contains strings of exactly 13 characters chosen from an alphabet of 64 characters. $\mathcal{C}$ contains approximately 3.0×10^{23} strings.

The UNIX hashing functions $f \in \mathcal{F}$ are based upon a permutation of the Data Encryption Standard. $\mathcal{F}$ consists of 4,096 such functions f_i, $0 \leq i < 4,096$.

The UNIX authentication functions in ($\mathcal{L}$) are *login*, *su*, and other programs that confirm a user's password during execution. This system supplies the proper element of $\mathcal{C}$; that information may not be available to the user. Some of these functions may be accessible over a network—for example, through the telnet or FTP protocols.

The selection functions in ($\mathcal{S}$) are programs such as *passwd* and *nispasswd*, which change the password associated with an entity.

Versions of UNIX and Linux now allow much longer passwords. For example, FreeBSD 10 (a variant of the UNIX operating system) allows passwords of up to 128 characters [709]. Also, the hash functions in $\mathcal{F}$ are MD5, Blowfish, SHA-256, SHA-512, an extended form of the original UNIX hashing functions (see Exercise 10), and a hash algorithm compatible with the Microsoft Windows NT scheme. The binary hashed output of these functions is translated into printable strings, the precise encoding used varying with the hash algorithm selected.

The goal of an authentication system is to ensure that entities are correctly identified. If one entity can guess another's password, then the guesser can impersonate the other. The authentication model provides a systematic way to analyze this problem. The goal is to find an $a \in \mathcal{A}$ such that, for $f \in \mathcal{F}$,

[1] In practice, some characters (such as the delete character) have special meanings and are rarely used.

$f(a) = c \in \mathcal{C}$ and c is associated with a particular entity (or any entity). Because one can determine whether a is associated with an entity only by computing $f(a)$ or by authenticating via $l(a)$, we have two approaches for protecting the passwords, often used simultaneously:

1. Hide enough information so that one of a, c, or f cannot be found.
2. Prevent access to the authentication functions $\mathcal{L}$.

EXAMPLE: Many UNIX systems make the files containing complementation information readable only by *root*. These schemes, which use shadow password files, make the set of complements c in actual use unknown. Hence, there is insufficient information to determine whether or not $f(a)$ is associated with a user. Similarly, other systems make the set of complementation functions $\mathcal{F}$ unknown; again, the computation of the value $f(a)$ is not possible. This is an example of the first approach.

As an example of the second approach, consider a site that does not allow the *root* user to log in from a network. The *login* functions exist but always fail. Hence, one cannot test authentication of *root* with access to these functions over a network.

13.3 Password Selection

The goal of password selection schemes is to make the difficulty of guessing the password as great as possible without compromising the ability of the user to use that password. Several schemes designed to meet this criterion have been proposed, and we examine the most common ones here.

13.3.1 Random Selection of Passwords

In this scheme, passwords are generated randomly from some set of characters and some set of lengths. This minimizes the chances of an attacker guessing a password.

Theorem 13.1. Let the expected time required to guess a password be T. Then T is a maximum when the selection of any of a set of possible passwords is equiprobable.

Proof See Exercise 1.

Theorem 13.1 guides random selection of passwords in the abstract. In practice, several other factors mediate the result. For example, passwords selected

at random include very short passwords. Attackers try short passwords as initial guesses (because there are few enough of them so that all can be tried). This suggests that certain classes of passwords should be eliminated from the space of legal passwords P. The danger, of course, is that by eliminating those classes, the size of P becomes small enough for an exhaustive search.

Complicating these considerations is the quality of the random (or pseudo-random) number generator. If the period of the password generator is too small, the size of P allows every potential password to be tested. This situation can be obvious, although more often it is not.

EXAMPLE: Morris and Thompson [1380] tell about a PDP-11 system that randomly generated passwords composed of eight capital letters and digits, so to all appearances, $|P| = (26 + 10)^8 = 36^8$. Testing a password took 0.00156 seconds, so trying all possible passwords would require 140 years. The attacker noticed that the pseudorandom number generator used on the PDP-11 had a period of $2^{16} - 1$ (because the PDP-11 is a 16-bit machine). This meant that there were only $2^{16} - 1 = 65{,}535$ possible passwords. Trying all of them would take about 102 seconds. It actually took less than 41 seconds to find all the passwords.

Human factors also play a role in this problem. Psychological studies have shown that humans can repeat with perfect accuracy about eight meaningful items, such as digits, letters, or words [109, 455, 1348]. If random passwords are eight characters long, humans can remember one such password. So a person who is assigned two random passwords must write them down—and indeed, studies have shown this to be the case [14]. Although most authorities consider this to be poor practice, the vulnerabilities of written passwords depend on where a written password is kept. If it is kept in a visible or easily accessed place (such as taped to a terminal or a keyboard or pinned to a bulletin board), writing down the password indeed compromises system security. However, if wallets and purses are rarely stolen by thieves with access to the computer systems, writing a password down and keeping it in a wallet or purse is often acceptable.

Michele Crabb describes a clever method of obscuring the written password [473]. Let X be the set of all strings over some alphabet. A site chooses some simple transformation algorithm $t : X \rightarrow A$. Elements of X are distributed on pieces of paper. Before being used as passwords, they must be transformed by applying t. Typically, t is very simple; it must be memorized, and it must be changed periodically.

EXAMPLE: The transformation algorithm is: "Capitalize the third letter in the word, and append the digit 2." The word on the paper is "Swqgle3". The password will be "SwQgle32".

This scheme is most often used when system administrators need to remember many different passwords to access many different systems. Then, even if the paper is lost, the systems will not be compromised.

With computers, this method can use any transformation, including encryption [756].

13.3.2 Pronounceable and Other Computer-Generated Passwords

A compromise between using random, unmemorizable passwords and writing passwords down is to use pronounceable passwords. Gasser [752] did a detailed study of such passwords for the Multics system and concluded that they were viable on that system.

Pronounceable passwords are based on the unit of sound called a phoneme. In English, phonemes for constructing passwords are represented by the character sequences *cv*, *vc*, *cvc*, or *vcv*, where *v* is a vowel and *c* a consonant.

EXAMPLE: The passwords "helgoret" and "juttelon" are pronounceable passwords; "przbqxdf" and "zxrptglfn" are not.

The advantage of pronounceable passwords is that fewer phonemes need to be used to reach some limit, so that the user must memorize "chunks" of characters rather than the individual characters themselves. In effect, each phoneme is mapped into a distinct character, and the number of such characters is the number of legal phonemes. In general, this means that the number of pronounceable passwords of length n is considerably lower than the number of random passwords of length n. Hence, an offline dictionary attack is expected to take less time for pronounceable passwords than for random passwords.

Assume that passwords are to be at most 8 characters long. Were these passwords generated at random from a set of 96 printable characters, there would be 7.23×10^{15} possible passwords. But if there are 440 possible phonemes, generating passwords with up to 6 phonemes produces approximately the same number of possible passwords. One can easily generalize this from phonemes to words, with similar results. One way to alleviate this problem is through *key crunching* [811].

> **Definition 13–3.** Let n and k be two integers, with $n \geq k$. *Key crunching* is the hashing of a string of length n or less to another string of length k or less.

Cryptographic hash functions such as the ones in Section 10.4 are used for key crunching.

Pronounceable password mechanisms often suffer from a "smallest bucket" problem [736]. The probabilities of the particular phonemes, and hence the passwords, are not uniform, either because users reject certain generated passwords as unpronounceable (or impossible for that particular user to remember) or

because the phonemes are not equiprobable in the chosen natural language. So the generated passwords tend to cluster into "buckets" of unequal distribution. If an attacker can find a "bucket" containing an unusually large number of passwords, the search space is reduced dramatically. Similarly, if an attacker can find a bucket with an unusually small number of passwords that users might be likely to select, the search space again is reduced dramatically. Indeed, Ganesan and Davies [736] examined two pronounceable password schemes and found the distribution of passwords into the buckets to be nonuniform.

13.3.3 User Selection of Passwords

Psychological studies have shown that people can remember items (such as passwords) better when they create them. This *generation effect* [557, 1762] means that user-created passwords will be more memorable than computer-generated ones. Countering this advantage is that users tend to select familiar passwords such as dictionary words, as discussed above. Thus, when users can select passwords, the selection mechanism should constrain what passwords users are allowed to select. This technique, called *proactive password selection* [222], enables users to propose passwords they can remember, but rejects any that are deemed too easy to guess. It has several variations; the most widely used are text-based passwords.

The set of passwords that are easy to guess is derived from experience coupled with specific site information and prior studies [905, 1380, 1798]. Klein [231, 1063] took 13,892 password hashes and used a set of dictionaries to guess passwords. He found that 8% of the guessed passwords were dictionary words, 4% were commonly used names, and 3% were user or account names. He also found that 17.5% of the passwords had a length of under 6, 34.7% were of length 6, 24.4% were of length 7, and 23.4% were of length 8 (the maximum length allowed).

Later studies [301, 308, 1313, 2041] have produced similar results. Unlike Klein's, these studies did not guess passwords, using instead user surveys or publicly disclosed passwords. Nevertheless, they are illuminating. For example, Bryant and Campbell [308] found 68.8% of the passwords were between six and nine characters long, and 18.6% were longer. Further, alphanumeric passwords dominate. Other studies confirm this.

Some categories of passwords that researchers have found easy to guess are any of the following:

1. Account names
2. User names
3. Computer names
4. Dictionary words
5. Patterns from the keyboard
6. Letters, digits, or letters and digits only
7. Passwords used in the past

8. Passwords with too many characters in common with the previous (current) password
9. Variants such as replacing letters with control characters, "a" with "2" or "4", "e" with "3", "h" with "4", "i" with "1", "l" with "1", "o" with "0", "s" with "5" or "$", and "z" with "5"

Additionally, passwords that are short are easily found (the length of "short" depends on the current technology).

EXAMPLE: The strings "hello" and "mycomputer" are poor passwords because they violate criteria 4. The strings "qwertyuiop[" and "311t3$p32k" are also poor as the first is the top row of letters from a U.S. keyboard (violating criterion 5), and the second is the word "elitespeak" modified as in criterion 9.

Good passwords can be constructed in several ways. Perhaps the best way is to pick a verse from an obscure text or poem (or an obscure verse from a well-known poem) and select the characters for the string from its letters. An experiment showed passwords generated in this way were more resistant to guessing than passwords generated in other ways [1114].

EXAMPLE: Few people can recite the third verse of "The Star-Spangled Banner" (the national anthem of the United States):

> And where is that band who so vauntingly swore
> That the havoc of war and the battle's confusion
> A home and a country should leave us no more?
> Their blood has wiped out their foul footsteps' pollution.
> No refuge could save the hireling and slave
> From the terror of flight, or the gloom of the grave:
> And the star-spangled banner in triumph doth wave
> O'er the land of the free and the home of the brave

Choose the second letter of each word of length 4 or greater of the first four lines, alternating case, and add punctuation from the end of the lines followed by a "+" and the initials of the author of the poem: "WtBvStHbChCsLm?TbWtF.+FSK". This is also a password that is hard to guess. But see Exercise 5.

Definition 13–4. A *proactive password checker* is software that enforces specific restrictions on the selection of new passwords. These restrictions are called the *password policy.*

A proactive password checker must meet several criteria [231]:

1. It must always be invoked. Otherwise, users could bypass the proactive mechanism.
2. It must be able to reject any password in a set of easily guessed passwords (such as in the list above).

3. It must discriminate on a per-user and a per-site basis. Passwords suitable at one organization may be very poor at another.
4. It should have a pattern-matching facility. Many common passwords, such as "aaaaa", are not in dictionaries but are easily guessed. A pattern-matching language makes detecting these patterns simple.
5. It should be able to execute subprograms and accept or reject passwords based on the results. This allows the program to handle spellings that are not in dictionaries. It also allows administrators to extend the password filtering in unanticipated ways.
6. The tests should be easy to set up, so administrators do not erroneously allow easily guessed passwords to be accepted.

EXAMPLE: The proactive password checker OPUS [1799] addresses the sizes of dictionaries. Its goal is to find a compact representation for very large dictionaries. Bloom filters provide the mechanism. Each word in the dictionary is run through a hash function that produces an integer h_i of size less than some parameter n. This is repeated for k different hash functions, producing k integers $h_1, \ldots, h_k$. The OPUS dictionary is represented as a bit vector of length n. To put the word into the OPUS dictionary, bits $h_1, \ldots, h_k$ are set.

When a user proposes a new password, that word is run through the same hash functions. Call the output $h'_1, \ldots, h'_k$. If any of the bits $h'_1, \ldots, h'_k$ are not set in the OPUS dictionary, the word is not in the OPUS dictionary and is accepted. If all are set, then to some degree of probability the word is in a dictionary fed to OPUS and should be rejected.

EXAMPLE: Ganesan and Davies [736] propose a similar approach. They generate a Markov model of the dictionary, extract information about trigrams, and normalize the results. Given a proposed password, they test to see if the word was generated by the Markov model extracted from the dictionary. If so, it is deemed too easy to guess and is rejected.

Both these methods are excellent techniques for reducing the space required to represent a dictionary. However, they do not meet all the requirements of a proactive password checker and should be seen as part of such a program rather than as sufficient on their own.

EXAMPLE: A "little language" designed for proactive password checking [223] is based on these requirements. The language includes functions for checking whether or not words are in a dictionary (a task that could easily use the techniques of OPUS or Ganesan and Davies). It also included pattern matching and the ability to run subprograms, as well as the ability to compare passwords against previously chosen passwords.

The language contains pattern matching capabilities. If the variable *gecos* contained the string

```
Matt Bishop, 2209 Watershed Science
```

then the expression

```
setpat "$gecos" "^\([\^,]*\), \(.*\)$" name office
```

matches the pattern with the value of *gecos* (obtained by prefixing a "$" to the variable name). The strings matched by the subpatterns in "\(" and "\)" are assigned to the variables *name* and *office* (so *name* is "Matt Bishop" and *office* is "2209 Watershed Science"). Equality and inequality operators work as string operators. All integers are translated to strings before any operations take place. In addition, functions check whether words are in a file or in the output of programs.

A logical extension of passwords is the passphrase.

Definition 13–5. A *passphrase* is a password composed of multiple words and, possibly, other characters.

Given advances in computing power, passwords that were once deemed secure are now easily discoverable. A passphrase increases the length of passwords while allowing the user to pick something that is easier to remember. Passphrases may come from known texts, or from the user's own imagery.

EXAMPLE: Continuing with "The Star Spangled Banner," one might generate a passphrase from the third and sixth line: "A home and a country should leave us no more? From the terror of flight, or the gloom of the grave." Another example comes from the comic *xkcd* [1401], where the passphrase "correct horse battery staple" has as a memory aid a horse looking at something and saying "That's a battery staple" and someone saying "Correct!"

The memorability of user-selected passwords and passphrases is a good example of how environment affects security. Given the ubiquity of web servers that require passwords, people will reuse passwords. One study [688] found that the average user has between six and seven passwords, and each is shared among about four sites; the researchers obtained this information from users who opted in to a component of a browser that recorded information including statistics about their web passwords (but *not* the password itself). The question is how people select these passwords.

Ana Maria de Alvaré [514] observed that users are unlikely to change a password until they find the password has been compromised; she also found they construct passwords that are as short as the password changing mechanism will allow.

Passphrases appear no harder to remember than passwords. Keith et al. [1018] conducted an experiment with three groups, one of which received no guidance about password selection, one of which had to select passwords that were at least 7 characters long, and one of which needed to pick a passphrase of at least 15 characters; the latter two groups' selections also had to have an uppercase letter, a lowercase letter, and a nonletter. They then monitored login attempts to a course

website over the term, and found no significant difference in the login failure rates after correcting for typographical errors. However, there were significantly more typographical errors among the users with passphrases. Later work [1019] showed that if passphrases consisted of text such as found in normal documents (called "word processing mode" or WPM), the rate of typographical errors dropped. Of course, one potential problem is that users may choose words from a small set when creating passphrases.

One widely used method for keeping track of passwords is to encipher them and store them using a password wallet.

> **Definition 13–6.** A *password wallet* or *password manager* is a mechanism that encrypts a set of a user's passwords.

The wallet allows users to store multiple passwords in a repository that is itself encrypted using a single cryptographic key, so the users need only remember that key (sometimes called a *master key*). The advantage is that the master key can be quite complex, as that is the only password the user need remember. But there are two disadvantages: accessibility and cascading disclosure.

The user must have access to the password wallet whenever she needs a password. The widespread use of portable computing on cellular telephones, tablets, and laptops has ameliorated this availability problem, but absent such devices the user needs access to the system or systems where the wallet is stored. Further, if the user's master password is discovered, for example because it is easy to guess or the system with the password wallet is compromised, then *all* passwords in the wallet are also disclosed.

13.3.4 Graphical Passwords

These schemes require a user to draw something to authenticate. The system encodes this drawing in some way and compares it to a stored encoding of an initial drawing made by that user. If the two match, the user is successfully authenticated. Here, $\mathcal{A}$ is the set of all possible graphical elements, $\mathcal{C}$ the stored representation of the graphical elements, and $\mathcal{F}$ the set of functions that compare the graphical elements to the complementary information.

Biddle, Chiasson, and van Oorschot [197] categorize these schemes based on how the user is expected to remember the password.

Recall-based systems require users to recall and draw a secret picture. The selection functions $s \in \mathcal{S}$ typically supply a grid for the users to draw their picture. The picture can be a continuous stroke or several disconnected strokes. The system encodes this as a sequence of coordinates. When a user logs in and supplies the password, the entered password is encoded and the result compared to the stored encoding. Some systems require an exact match; others require that the entered drawing be "close enough" to the stored one.

Recognition-based systems require users to recognize a set of images. Typically, a user is presented with a collection of images and must select one or more

images from that set. These systems use images of faces, art images, and pictures of all kinds. Then the user must select those images (or a subset of them) from sets of images presented when authentication is required. A variation is to require the images to be selected in the order users first chose them in.

Cued-recall systems require users to remember and identify specific locations within an image. The system assigns the user an image, and the user initializes his password by selecting some number of places or points in the image. To authenticate, the user simply selects the same set when presented with the image. The effectiveness of this scheme depends in part upon how the system determines when the points selected are close enough to the initial points.

The expectation is that the set of possible graphical elements $|\mathcal{A}|$ in each of the schemes is sufficiently large to inhibit guessing attacks. However, just as with text-based passwords, the graphical elements are often selected from a much smaller space. For example, users often draw or select patterns of some kind in recall-based systems, and select prominent features or points in cued-based systems, so searching based on common patterns is often fruitful, just as searching for common words and patterns is often fruitful in text-based passwords. Recognition-based systems suffer from a similar problem; for example, when the images are faces, user selection is influenced by race and gender [197].

Human factors play a large role in the effectiveness of graphical passwords. Stobert and Biddle [1827] studied the memorability of graphical passwords and found that users could remember recognition-based graphical passwords the best but slow login times hindered their usability. They proposed a scheme that combined recall-based and recognition-based schemes and found it more effective than either individual scheme. A different experiment used a cued-recall system that presented image components in different ways when the password was created. The distribution of passwords changed depending on the order of component presentation [1879]. As the security depends on the distribution of passwords this means that presentation must be considered when users are selecting cued-recall-based passwords.

13.4 Attacking Passwords

Guessing passwords requires either the set of complementation functions and complementary information or access to the authentication functions. In both approaches, the goal of the defenders is to maximize the time needed to guess the password. A generalization of Anderson's Formula [51] provides the fundamental basis.

Let P be the probability that an attacker guesses a password in a specified period of time. Let G be the number of guesses that can be tested in one time unit. Let T be the number of time units during which guessing occurs. Let N be the number of possible passwords. Then $P \geq \frac{TG}{N}$.

EXAMPLE: Let R be the number of bytes per minute that can be sent over a communication line, let E be the number of characters exchanged when logging in, let S be the length of the password, and let A be the number of characters in the alphabet from which the characters of the password are drawn. The number of possible passwords is $N = A^S$, and the number of guesses per minute is $G = \frac{R}{E}$. If the period of guessing extends M months, this time in minutes is $T = 4.32 \times 10^4 M$. Then

$$P \geq \frac{4.32 \times 10^4 M}{A^S} \frac{R}{E}$$

or $A^S \geq \frac{4.32 \times 10^4 MR}{PE}$, the original statement of Anderson's Formula.

EXAMPLE: Let passwords be composed of characters drawn from an alphabet of 96 characters. Assume that 5×10^8 guesses can be tested each second. We wish the probability of a successful guess to be 0.001 over a 365-day period. What is the minimum password length that will give us this probability?

From the formulas above, we want $N \geq \frac{TG}{P} = \frac{(365 \times 24 \times 60 \times 60)(5 \times 10^8)}{0.001} \approx 1.58 \times 10^{19}$. Thus, we must choose an integer S such that

$$\sum_{i=0}^{S} 96^i \geq N = 1.58 \times 10^{19}$$

This holds when $S \geq 10$. So, to meet the desired conditions, passwords of at least length 10 must be required.

Several assumptions underlie these examples. First, the time required to test a password is constant. Second, all passwords are equally likely to be selected. The first assumption is reasonable, because the algorithms used to validate passwords are independent of the password's length, or the variation is negligible. The second assumption usually does not hold, leading attackers to focus on those passwords they expect users to select. This leads to dictionary attacks.

Definition 13–7. A *dictionary attack* is the guessing of a password by repeated trial and error.

The name of this attack comes from the list of words (a "dictionary") used for guesses. The dictionary may be a set of strings in random order or (more usually) a set of strings in decreasing order of probability of selection.

13.4.1 OffLine Dictionary Attacks

This version assumes the sets $\mathcal{F}$ of complementation functions and $\mathcal{C}$ of complementary information is known to the attacker.

> **Definition 13–8.** In an *offline dictionary attack*, the attacker takes each guess g and computes $f(g)$ for each $f \in \mathcal{F}$. If $f(g)$ corresponds to the complementary information for entity E, then g authenticates E under f.

EXAMPLE: Attackers who obtain a UNIX system's password file can use the (known) complementation functions to test guesses. (Many programs automate this process [2057].) This is an offline attack. But the attackers need access to the system to obtain the complementation data in the password file. To gain access, they may try to guess a password using the authentication function. They use a known account name (such as *root*) and guess possible passwords by trying to log in. This is an online attack.

The issue of efficiency controls how well an authentication system withstands dictionary attacks. Precomputation speeds up offline dictionary attacks. One method is due to Martin Hellman [893], and was originally designed to find DES keys.

Let $f \in \mathcal{F}$ be the complementation function. Choose m passwords $sp_i \in \mathcal{A}$. Let $r : \mathcal{C} \rightarrow \mathcal{A}$ be a function that transforms an element of $\mathcal{C}$ into an element of $\mathcal{A}$ (r is called the *reduction function*). For each password, let $xp_{i,0} = sp_i$, and compute $xp_{i,j} = f(r(xp_{i,j-1}))$ for $j = 1, \ldots, t$. Let $ep_i = xp_{i,t}$. The pairs (sp_i, ep_i) are then stored in a table T, and the intermediate values discarded. Figure 13–1a shows this process, and Figure 13–1b shows the stored table.

An attacker wants to determine the password p that has the complementary information $c = f(p)$. First, the attacker looks at the ep_i in the table. If there is an i such that $ep_i = c$, then p is the value in the next-to-last column in Figure 13–1a because that value was used to produce ep_i. To find it, the attacker reconstructs

$$
\begin{array}{lll}
sp_1 \rightarrow xp_{1,1} = f(sp_1) \rightarrow xp_{1,2} = f(r(xp_{1,1})) \rightarrow \cdots \rightarrow ep_1 = f(r(xp_{1,t-1})) & (sp_1, ep_1) \\
sp_2 \rightarrow xp_{2,1} = f(sp_2) \rightarrow xp_{2,2} = f(r(xp_{2,1})) \rightarrow \cdots \rightarrow ep_2 = f(r(xp_{2,t-1})) & (sp_2, ep_2) \\
\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots & \vdots \\
sp_m \rightarrow xp_{m,1} = f(sp_m) \rightarrow xp_{m,2} = f(r(xp_{m,1})) \rightarrow \cdots \rightarrow ep_m = f(r(xp_{m,t-1})) & (sp_m, ep_m)
\end{array}
$$

(a) (b)

Figure 13–1 Search tables: (a) the computation process; (b) what is stored.

the chain of hashes leading up to ep_i. If c does not match any of the ep_i, then the attacker computes $f(p)$ and compares it to the ep_i. On a match, p is in the second-to-last column; to find it, the attacker reconstructs that chain. The process iterates until the password is found or until the password is determined not to be in the table.

Rivest [533, p. 100] suggested a simple optimization. Rather than choose the endpoints based on a parameter t, choose them based on a property of the value of the endpoints, for example that the first n bits are 1, that is expected to hold after t iterations. Then, given a hash, iterate the complementation function (and reduction function) until an endpoint is generated. This produces variable-length chains.

One problem with these tables lies in *collisions*. When the computation of two chains produces the same value at any intermediate point, the chains merge. *Rainbow tables* [1469] allow collisions without merging. To do this, multiple reduction functions are used. So for each password, $xp_{i,j} = f(r_i(xp_{i,j-1}))$ for $j = 1, \ldots, t$. This allows collisions, but in order for two chains to merge, the collisions must occur for the same value j in both chains.

13.4.1.1 Salting

If an offline dictionary attack is aimed at finding *any* user's password (as opposed to *a particular* user's password), a technique known as *salting* increases the amount of work required [1380]. Salting makes the choice of complementation function a function of randomly selected data. Ideally, the random data is different for each user. Then, to determine if the string s is the password for any of a set of n users, the attacker must perform n complementations, each of which generates a different complement. Thus, salting increases the work by the order of the number of users.

EXAMPLE: Linux and UNIX-like systems use salts that are generated when the password is selected. As an example, FreeBSD 10 defines three schemes.

In the *traditional* scheme, the salt is a 12-bit integer chosen at random. The specific complementation function depends on the salt. The E table in the DES (see Figure F–3a) is perturbed in one of $2^{12} = 4,096$ possible ways—if bit i in the salt is set, table entries i and $i + 24$ are exchanged [1380]—and the message of all 0 bits is enciphered using the password as a key. The result of the encipherment is then enciphered with the password, iterating this procedure until 25 encipherments have occurred. The resulting 64 bits are mapped into 11 characters chosen from a set of 64 characters. The salt is split into two sets of 6 bits, and those sets are mapped to printable characters using the same alphabet. The 11-character representation of output is appended to the 2-character representation of the salt. The authentication function is chosen on the basis of the salt also; hence, the salt must be available to all programs that need to verify passwords.

In the *extended* scheme, the system stores a salt of 24 bits and a count of 24 bits, and the password can be any length. The password is transformed into a DES key by enciphering the first 8 bytes with itself using the DES; the result

is xor'ed with the next 8 bytes of the password, and the result is enciphered with itself. This continues until all characters of the password have been used. Next, the salt is used to perturb the E-table as in the traditional scheme. The result of the password transformation is used as the key to encipher the message of all 0 bits, as in the traditional scheme, but the encipherment iterates the number of times indicated by count, rather than 25. The result is then transformed into a printable string, using the same technique as in the traditional algorithm.

In the *modular* scheme, one of five algorithms is used: MD5, Blowfish, SHA256, SHA-512, and the scheme used in Windows NT. The salts in these cases are treated as character strings and combined with the password during the hashing. The advantage to these schemes over the traditional one is that the password can be any length, and the length of the salt depends on the algorithm used (for example, SHA-256 and SHA-512 allow a salt of at most 16 characters).

In all cases, the salt and password hash are stored as a string. To determine which scheme is used, the system looks at the first characters of the stored string. If the first character is "_", then the extended scheme is being used. If it is "$", the following characters up to the next "$" indicate which algorithm of the modular scheme is to be used. Otherwise, the traditional scheme is used.

13.4.2 OnLine Dictionary Attacks

If either the complementary information or the complementation functions are unavailable, the authentication functions $l \in \mathcal{L}$ may be used.

> **Definition 13–9.** In an *online dictionary attack*, the attacker supplies the guess g to an authentication function $l \in \mathcal{L}$. If l returns **true**, g is the correct password.

Although using the authentication functions that systems provide for authorized users to log in sounds difficult, the patience of some attackers is amazing. One group of attackers guessed passwords in this manner for more than two weeks before gaining access to one target system.

Unlike an offline dictionary attack, this attack cannot be prevented, because the authentication functions must be available to enable legitimate users to access the system. The computer has no way of distinguishing between authorized and unauthorized users except by knowledge of the password.

Defending against such attacks requires that the authentication functions be made difficult for attackers to use, or that the authentication functions be made to react in unusual ways. Several types of techniques are common.

Backoff techniques increase the time between interactions as the number of interactions increases. One such technique, exponential backoff, begins when a user attempts to authenticate and fails. Let x be a parameter selected by the system administrator. The system waits $x^0 = 1$ second before reprompting for the name and authentication data. If the user fails again, the system reprompts after $x^1 = x$

seconds. After n failures, the system waits x^{n-1} seconds. Other backoff techniques use arithmetic series rather than geometric series (reprompting immediately, then waiting x seconds, then waiting $2x$ seconds, and so forth).

EXAMPLE: If a user fails to supply a valid name and the corresponding password in three tries, FreeBSD 9.0 applies a linear backoff scheme. It adds a five-second delay in prompting for every attempt beyond the third try.

On the Web, this approach is infeasible as one can simply disconnect from the site and then reconnect. Instead, websites use CAPTCHAs,[2] which are visual or audio tests that are easy for humans to solve, but difficult for humans to solve [1946]. The test may require the user to type a sequence of characters presented on a grainy background with the characters positioned at various angles. It may have the user identify specific objects in a set of images with complex or cluttered backgrounds. Concerns about the ease of use of CAPTCHAs, especially audio CAPTCHAs, have been raised [326, 2042], and several services exist that will solve CAPTCHAs on request [1388].

An alternate approach is *disconnection*. After some number of failed authentication attempts, the connection is broken and the user must reestablish it. This technique is most effective when connection setup requires a substantial amount of time, such as redialing a telephone number. It is less effective when connections are quick, such as over a network.

EXAMPLE: If a user fails to supply a valid name and the corresponding password in 10 tries, FreeBSD 9.0 breaks the connection.

Disabling also thwarts online dictionary attacks. If n consecutive attempts to log in to an account fail, the account is disabled until a security manager can reenable it. This prevents an attacker from trying too many passwords. It also alerts security personnel to an attempted attack. They can take appropriate action to counter the threat.

One should consider carefully whether to disable accounts and which accounts to disable. A (possibly apocryphal) story concerns one of the first UNIX vendors to implement account disabling. No accounts were exempt from the rule that three failed logins disabled the account. An attacker broke into a user account, and then attempted to log in as *root* three times. The system disabled that account. The system administrators had to reboot the system to regain *root* access.

EXAMPLE: Linux systems and Windows 7, 8, and 10 systems have the ability to disable accounts after failed logins. Typically, the Linux *root* account cannot be disabled. The Windows *administrator* account can be locked out (the equivalent of "disabled" in this context) from network logins, but not from local logins.

[2]The acronym stands for "Completely Automated Public Turing tests to tell Computers and Humans Apart."

Jailing gives the unauthenticated user access to a limited part of the system in order to gull that user into believing that he or she has full access. The jail then records the attacker's actions. This technique is used to determine what the attacker wants or simply to waste the attacker's time.

EXAMPLE: An attacker was breaking into the computers of AT&T Bell Laboratories. Bill Cheswick detected the attack and simulated a slow computer system. He fed the attacker bogus files and watched what the attacker did. He concluded that keeping the jail was not an effective way to discover the attacker's goals [402].

One form of the jailing technique is to plant bogus data on a running system, so that after breaking in the attacker will grab the data. (This technique, called *honeypots*, is often used in intrusion detection. See Section 27.3.2.1, "Containment Phase.") Clifford Stoll used this technique to help trap an attacker who penetrated computers at the Lawrence Berkeley Laboratory. The time required to download the bogus file was sufficient to allow an international team to trace the attacker through the international telephone system [1829, 1831].

13.4.3 Password Strength

How well the password selection schemes work to produce passwords that are difficult to guess requires an examination of selected passwords. Some data sets come from users or system administrators who cooperate with the researchers performing the study; others come from data sets gathered by attackers who compromise the passwords in some way and distribute them, or are themselves compromised.

A NIST report [324] uses the standard definition of entropy (see Appendix C) and defines two additional types of entropy.

Definition 13–10. *Guessing entropy* is the expected amount of work to guess the password of a selected user.

Definition 13–11. Given a set of passwords, *min-entropy* is the expected amount of work to guess any single password in the set.

Computing these requires that the distribution of passwords be known.

EXAMPLE: Suppose passwords are randomly assigned. Each password is composed of 8 characters drawn from a set of 94 characters each of which is equally likely to be chosen. Then the entropy of each password is $lg 94^8 \approx lg(6.1 \times 10^{15}) \approx 52.4$. As any password in the set of possible passwords is equally likely to be assigned, the guessing entropy and min-entropy are the same.

When users select passwords, the password policy controlling the selection affects the entropy of the passwords.

EXAMPLE: The NIST report considers three scenarios: one where users can select any password, one in which users can select any password not in a dictionary, and one in which users must include a mixture of case and nonalphabetic characters in their password. Following various studies [1699, 1728], they assume:

1. The entropy of the first character is 4 bits.
2. The entropy of the next seven characters (2–8) is 2 bits per character.
3. The entropy of the next twelve characters (9–20) is 1.5 bits per character.
4. The entropy of all subsequent characters is 1 bit per character.
5. If the password policy is that of the second scenario, up to 6 bits are added to the computed entropy.
6. If the password policy is that of the last scenario, 6 bits are added to the computed entropy.

Under these assumptions, the guessing entropies of an 8-character password under each of the three scenarios are 18, 24, and 30 bits respectively (contrast that to 52.4 bits for a random password), and for a 16-character password, the guessing entropies are 30, 32, and 38 bits respectively (and for a random password of that length, the guessing entropy is 105.4 bits).

Computing the min-entropy is much more difficult because, as noted, users often select easy to guess passwords. So the NIST report provides a password policy that will ensure at least 10 bits of min-entropy. That policy disallows:

1. Detectable permutations of user names; or
2. Passwords matching a list of at least 50,000 common passwords, ignoring case.

They note that requiring users to choose passwords of at least 15 characters will probably produce a min-entropy of at least 10.

Unfortunately, there is no way to convert Shannon entropy into guessing entropy [1929]; indeed, Weir et al. [1989] show experimentally that these metrics do not reflect the strength of passwords in practice. Bonneau [264] presents alternate metrics including one based on the attacker's desired success rate. He validated this metric using a study of anonymized statistical data gathered from 70,000,000 Yahoo! passwords.[3]

Florêncio and Herley [688] gathered data about passwords used to access websites. For privacy reasons, they did not record passwords. Instead, they

[3]The passwords themselves were *not* collected, and Yahoo!'s legal team approved the data gathering and analysis [264].

divided the set of characters into four alphabets: lowercase letters ("a" to "z," 26 characters), uppercase letters ("A" to "Z," 26 characters), digits ("0" to "9," 10 characters), and other (22 special characters). Let the number of characters in the *alphabets* used be α and the password length be L. They calculated the password strength as $\log_2 \alpha^L$. When they applied their metric to passwords used at various websties, they noted that the more important the service provided to the user, the stronger the password seemed to be; passwords for the New York Times websites, for example, averaged a strength of 37.86, whereas passwords for employees accessing a corporate website had an average strength of 51.36.

Kelley et al. used an alternate approach [1020]. They used Mechanical Turk to create a study that required users to select a password that conformed to one of seven policies, and then use it again several days later to obtain a small payment. Users were given one of two scenarios, the first involving a password protecting low value information (an online survey), and the second a password protecting high value information (email). The passwords had to conform to a policy chosen from seven possible policies, such as "passwords must have at least 16 characters" or "passwords must have at least 8 characters, including mixed case letters, a symbol, and a digit; it may not contain a dictionary word." They collected 12,000 passwords. Next, they defined *guess numbers* as the number of guesses required to guess a specific password for a specific algorithm. Thus, each password had a guess number for the brute-force algorithm based on a Markov model [2093] and another guess number of a heuristic algorithm [1990]. They found that the best password policy changed as more guesses were made; ultimately, requiring passwords to be at least 16 characters long had the greatest guessing number.

Password meters provide an estimate of password strength [348,363]. When users change passwords, the presence of a password meter matters more than the meter's design, and their influence depends on the context in which the password will be used; if the user considers the account iportant, the password chosen will be stronger than when the meter is not present [617].

13.5 Password Aging

Guessing passwords requires that access to the sets of complements and complementation functions or the set of authentication functions be obtained. If none of these have changed by the time the password is guessed, then the attacker can use the password to access the system.

Consider the last sentence's conditional clause. The techniques discussed in Section 13.3 attempt to negate the part saying "the password is guessed" by making that task difficult. The other part of the conditional clause, "if none of these have changed," provides a different approach: ensure that, by the time a password is guessed, it is no longer valid.

Definition 13–12. *Password aging* is the requirement that a password be changed after some period of time has passed or after some event has occurred.

Assume that the expected time to guess a password is 90 days. Then changing the password more frequently than every 90 days will, in theory, reduce the probability that an attacker can guess a password that is still being used. In practice, aging by itself ensures little, because the estimated time to guess a password is an average; it balances those passwords that can be easily guessed against those that cannot. If users can choose passwords that are easy to guess, the estimation of the expected time must look for a minimum, not an average. Hence, password aging works best in conjunction with other mechanisms such as the ones discussed in this chapter.

There are problems involved in implementing password aging. The first is forcing users to change to a different password. The second is providing notice of the need to change and a user-friendly method of changing passwords.

Password aging is useless if a user can simply change the current password to the same thing. One technique to prevent this is to record the n previous passwords. When a user changes a password, the proposed password is compared with these n previous ones. If there is a match, the proposed password is rejected. The problem with this mechanism is that users can change passwords n times very quickly, and then change them back to the original passwords. This defeats the goal of password aging.

An alternate approach is based on time. In this implementation, the user must change the password to one other than the current password. The password cannot be changed for a minimum period of time. This prevents the rapid cycling of passwords. However, it also prevents the user from changing the password should it be compromised within that time period.

EXAMPLE: UNIX and UNIX-like systems use the time period method to age passwords (when password aging is turned on). They record the time of the last change, the minimum time before which the password can be changed again, and the time by which the password must be changed. Different systems use different formats. Linux systems record the information in terms of days since January 1, 1970; FreeBSD 10 systems record it in terms of seconds since midnight of that epoch.

If passwords are selected by users, the manner in which users are reminded to change their passwords is crucial. Users must be given time to think of good passwords or must have their password choices checked. Grampp and Morris [807] point out that, although there is no formal statistical evidence to support it, they have found that the easiest passwords to guess are on systems that do not give adequate notice of upcoming password expirations. A study by Tam et al. [1854] found that sending a warning message one day before the password expired was optimal.

EXAMPLE: Most System V-based UNIX systems give no warnings or reminders before passwords expire. Instead, when users try to log in, they are told that their passwords have expired. Before they can complete the logins, they must change their passwords as part of the login process. Linux systems, on the other hand, give warning messages every time a user logs in within some period of time before the password expires. The default period of time is two weeks, but can be changed by the system administrator.

In this vein, a further weakness of password aging is how users select the next password. A study by Zhang, Monrose, and Reiter [2093] found that people who were given previous passwords for an account with password aging were able to guess the current password 41% of the time. Further, passwords for 17% of the accounts were guessed in no more than five tries. Chiasson and van Oorschot [404] show that the optimal benefit from password aging is to reduce the attacker's expectation of success during the interval between password changes from 1 (certain success when passwords are not aged) to 0.632 for each period. They conclude that password aging's effectiveness is offset by the user interaction issues.

13.5.1 One-Time Passwords

The ultimate form of password aging occurs when a password is valid for exactly one use.

> **Definition 13–13.** A *one-time password* is a password that is invalidated as soon as it is used.

The problems in any one-time password scheme are the generation of random (or pseudorandom) passwords and the synchronization of the user and the system. The former problem is solved by using a cryptographic hash function or enciphering function such as HMAC-SHA-1, and the latter either by having the system inform the user which password it expects—for example, by numbering all the user's passwords and having the system provide the number of the one-time password it expects—or synchronizing based on time or a counter.

EXAMPLE: S/Key [852, 853] implements a one-time password scheme. It uses a technique first suggested by Lamport [1130] to generate the passwords. Let h be a one-way hash function (S/Key uses MD4 or MD5, depending on the version). Then the user chooses an initial seed k, and the key generator calculates

$$h(k) = k_1, h(k_1) = k_2, \ldots, h(k_{n-1}) = k_n$$

The passwords, in the order they are used, are

$$p_1 = k_n, p_2 = k_{n-1}, \ldots, p_{n-1} = k_2, p_n = k_1$$