Computer Science and Engineering, UCSD — Winter 04
CSE 107: Introduction to Modern Cryptography — **Instructor:** Mihir Bellare
Class Notes: Passwords — January 5, 2004

Password-Based Authentication

As an introduction to security and cryptography, let us look at something we do extensively, namely to authenticate ourselves using passwords.

We consider a computer on which many different people have accounts. Each of these people has an associated user name. For example, we might have a user named Elaine Benes, whose user name is `ebenes`, and a user named Cosmo Kramer, whose user name is `cosmo`. A user name is a quantity that uniquely identifies a person in this context (if there were two people named Cosmo Kramer, they would get different user names) and is not considered a secret.

The computer manages our resources, providing access to files, data and programs. We want to control the access to these resources, for several reasons. Some people have more privileges than others. System managers are allowed to access certain files, programs or parts of the operating system that ordinary users are not. When many people use the same computer, we also want to protect them from each other. A particular person, meaning owner of a particular user name, should be allowed to access their own files or area in the computer, but not be allowed to access someone else's files. For example we don't want to allow Elaine access to Cosmo's mailbox file.

In order to perform this or any other kind of *access control*, we need *authentication*. This means that when someone logs in to the computer, they must first present their user name and prove that this is indeed their user name, and not someone else's. In other words, the computer should perform some check to make sure it is really Cosmo, not someone else, who is trying to access Cosmo's files.

In security and cryptography jargon, the malicious entity who may be trying to access resources to which it is not entitled, or perform some other illegal action, is often called the *adversary* or *attacker*. So the goal of authentication of a user name is to ensure that the person logging in is the real owner of this user name and not an adversary.

The standard authentication mechanism in use today is based on passwords. To each user name is associated a password. Usually, the password is chosen by the owner of the user name, although sometimes it might be issued by the system manager. The presumption (for security) is that the password associated to a user name is not known to any potentially adversarial entity.

We are now going to discuss one specific system, namely that of the UNIX (or Linux) operating systems.

1 The UNIX password system

A computer running the UNIX operating system maintains a *password file* that contains an entry for each password-possessing user. Entries for the two users mentioned above might look like this:

`ebenes` : go/dEBEg49seM : 1660 : 300 : Elaine BENES : /home/ebenes : /bin/bash
`cosmo` : JgF85z2z4/vfs : 1352 : 300 : Cosmo KRAMER : /home/cosmo : /bin/tcsh

Each entry consists of a number of fields, and has the form:

$$\texttt{username} : \underbrace{\mathsf{salt}}_{2}\underbrace{\mathsf{hpwd}}_{11} : \text{data} : \text{uid} : \text{gid} : \text{User's name} : \text{homedirectory} : \text{shell}$$

The number below a field, as written above, is the length of the field in bytes. Thus, the *hashed password* hpwd is a sequence (also called string) of characters that is 11 bytes long, meaning consists of 11 characters encoded in ASCII. The *salt* salt is a 12-bit string, but is encoded and stored as 2 bytes. (For example, for Cosmo Kramer, the salt is 'Jg' and the hashed password is 'F85z2z4/vfs'). The other fields are not relevant to our discussion.

How is this entry created? When a user obtains their account, they supply a password pwd. The system then picks at random a 12-bit value and encodes it appropriately to get a 2-byte salt, and computes the value $\mathsf{hpwd} = h(\mathsf{salt}\|\mathsf{pwd})$, where h is a fixed function that returns 11 bytes, and $\mathsf{salt}\|\mathsf{pwd}$ denotes the concatenation of the strings $\mathsf{salt}, \mathsf{pwd}$. The system then creates the above entry in its password file, storing the user's user name, salt and hpwd. (Note that the system does not store pwd. It discards it after having computed hpwd.)

We are not, for the moment, going to say how h works. Just assume there is some such function whose description, or code, is public.

How is authentication performed? When someone wants to log into the computer, they are asked to supply a user name and password:

ENTER USERNAME:
ENTER PASSWORD:

They might enter, say `cosmo` for the user name, and a password we will denote pwd'. The system then looks in the password file and retrieves the line for `cosmo`. It finds salt = 'Jg' and hpwd = 'F85z2z4/vfs'. It then computes $\mathsf{hpwd}' = h(\mathsf{salt}\|\mathsf{pwd}')$. If $\mathsf{hpwd}' = \mathsf{hpwd}$ then it accepts, meaning allows the user access, and if not it rejects, meaning does not allow the user access. Obviously if $\mathsf{pwd}' = \mathsf{pwd}$, meaning the password is correct, then access is granted, but if not, then we would expect access to be denied.

2 Security of the system

We are now going to attempt to assess security of this system by considering various possible threats and their effectiveness. This is an exercise in learning to think about possible threat models and evaluating systems under them.

We begin by asking ourselves what the attacker may want to achieve. It may want to break in to some specific account on the computer, say that of `cosmo`. It may want to break into some account, no matter which, so as to get some access to the computer. It may want to break in to a super-user account to get root privileges. The easiest of these, breaking into some account no matter which, is still damaging, for in many cases attackers can thence exploit system vulnerabilities to obtain super-user status.

Next we ask ourselves what information it has to aid it in its task. We will assume that the password file is available to the attacker. (Later we will discuss why this is a good assumption to make and also discuss a varaint of the system where the password file is kept protected.) Also the attacker knows, and can compute, the function h, because the description of the system is public.

2.1 Inverting h

Suppose the attacker wants to gain access under a certain user name, say `cosmo`. Let us denote `cosmo`'s password by pwd. If the attacker can, somehow, find pwd, it can certainly gain access, for it would simply supply this password when asked to authenticate itself.

Actually, the attacker need not find pwd to gain access. It need only find a value pwd' such that $h(\mathsf{salt}\|\mathsf{pwd}') = \mathsf{hpwd}$. We call such a value a pre-image of hpwd under the function $h(\mathsf{salt}\|\cdot)$. While pwd is one such pre-image, there may be others, and these, even if not the actual password, will, if entered to the system by the attacker, result in the attacker being granted access under user name `cosmo`. Make sure you understand why this is true by returning to the description of the authentication process above.

In order to make finding pre-images as hard as possible, we design h to be what is called "one-way." Roughly, this means that f is easy to compute, but hard to invert: given f it is hard to find a string a such that $h(a) = f$. "Easy" and "hard" here refer to the computational effort (running time of a program) to required to accomplish the task in question. Thus, we are saying there is a program that given any a will quickly return $h(a)$, but there is no program that given f will quickly return a such that $h(a) = f$. (There might be a program that returns a, but not one that runs in a practical amount of time.) If h is one-way, the task of finding pre-images, as discussed above, is hopefully hard.

Whether or not h is one-way depends of course on the design of h. As an example, suppose we define h as follows. The system picks at random and then stores a 11-byte character string K. Given input a string a of bytes, h outputs the 11-byte character string f computed as follows:

```
Function $h(a)$
    For $i = 1, \ldots, 11$ do
        $f[i] \leftarrow K[i] + a[i] \bmod 256$
    Return $f$
```

Above, $s[j]$ denotes the j-th byte of a string s. A character is identified with its 256-bit (1 byte) ASCII code, and when we write an operation of the form $a \leftarrow b + c \bmod 256$ above, we mean convert b, c to their ASCII codes, add these numbers, take the remainder after division by 256 to get another ASCII code, and output the corresponding character. (Above, if a is less than 11 bytes long, it is padded with zeros first to make it exactly 11 bytes long.)

Is this h one-way? No. Remember that the system makes h public, meaning the value K is known to the attacker. The following simple and efficient algorithm, given f, returns a such that $h(a) = f$, showing that h is not one-way:

```
Function Invert-$h(f)$
    For $i = 1, \ldots, 11$ do
        $a[i] \leftarrow f[i] - K[i] \bmod 256$
    Return $a$
```

Now, given hpwd and salt, the attacker can run Invert-$h(f)$ on input $f = \mathsf{hpwd}$ to get a, and set pwd' to the last 9 bytes of a. Then $h(\mathsf{salt}\|\mathsf{pwd}') = \mathsf{hpwd}$, so the attacker has the means to access the system.

Naturally, UNIX does not design h as above. Instead, the design makes use of the DES block cipher that we will see later. The UNIX password hash function h is believed by experts to be one-way [4, 7].

Does this mean the UNIX system is secure? Not necessarily. In fact in practice it is highly vulnerable, for reasons that have little to do with the design of h.

2.2 Password-guessing attacks

Say passwords are usually at most 8 characters in length. Then the number of possible passwords is

$$256^1 + 256^2 + \cdots + 256^8 \ = \ 18,519,084,246,547,628,288 \approx 1.8 \cdot 10^{19} \ .$$

This is a very large number. With a space of potential passwords so large, one would assume that the possibility of an attacker guessing the password of a user is small.

Unfortunately, this would be true only if users choose their passwords at random. Many users don't. One sees users that choose passwords that are words, common names, or variations of their user names. Knowing this, an attacker can formulate a *dictionary* DICT, which is a relatively small set of candidate passwords. This would typically contain all words in the standard English dictionary, some foreign words, and common names. It is also common to put in the dictionary all strings of length 6 or less consisting of only lower-case letters: there are 321,272,407 such strings. A dictionary may have up to 500 million words, much less than the $1.8 \cdot 10^{18}$ potential candidate passwords.

Now, the attacker would like to try the candidate passwords in its dictionary. Perhaps the most obvious way to do this is via an *on-line* attack. The attacker sits at a computer terminal (or accesses the computer remotely via the network) and tries to log in, under the user name of the user it is attacking, each time trying a different password from its dictionary. Such an on-line attack, however, is unlikely to be successful, for reasons that we now discuss.

The system can be designed to "turn off" the account corresponding to a user name when more than some number (eg. three) of unsuccessful login attempts have been made under this user name. Meaning, once it sees three incorrect passwords tried under a particular user name, it refuses access under this user name, without even testing the password entered by the person attempting to log in. An attacker is thus limited to three guesses, and it is unlikely they will be able to guess a password in just three attempts.

The above does bring with it a cost, though, which is that it opens the system to *denial of service* attacks. An attacker can disable your account simply by making three attempts to log in as you, and then you have to go to the trouble to re-instate it.

However, even if the system does not have such a turn-off feature, it attempts to ensure that it takes a long time to try passwords. The system (purposely) takes its time saying whether a login attempt is successful or not. Say it takes 5 seconds to respond to an authentication request. This means that in an hour an attacker can try 720 passwords. That is actually not a lot; most dictionaries are significantly larger than that.

The effort that an attacker needs to invest to make an on-line attack successful thus seems large enough to deter such an attack.

Much more effective is an *off-line* attack, also called a *dictionary* attack. Knowing the entries $\mathsf{salt}, \mathsf{hpwd}$ from the password file, the attacker runs the following code on its own computer to find a corresponding password:

```
Dictionary-Attack(DICT, $\mathsf{salt}$, $\mathsf{hpwd}$)
    For all $\mathsf{pwd}' \in$ DICT do
        If $h(\mathsf{salt}\|\mathsf{pwd}') = \mathsf{hpwd}$ then return $\mathsf{pwd}'$ and halt
    Return FAIL
```

In order to slow down this attack, the Unix password hashing function h is designed so that it is slow to compute. However, one cannot make it too slow, since that would slow down the authentication process, inconveniencing the user. In the end, this has not helped much. It turns out that this attack is very effective in practice. Here are some statistics.

In 1979, Morris and Thompson were able to run the above attack on a number of systems and recover about 3,000 passwords in the course of a week. Their data on these passwords is frightening. Some passwords (0.5%) were only a single character. Another 2% were only two characters long, and 14% were three characters long. In all, 86% of the passwords on the systems they attacked could be recovered, being dictionary words of some sort.

That was a while back, and there has been much insistence since then that users improve their choices of passwords. System administrators ask that users pick passwords that do not fall into common dictionaries, asking people to avoid English words and common names, to make passwords long, to mix upper and lower case letters, and so on. However, even later studies have found that weak passwords remain. A study in 1992 recovered 15,000 passwords. It found that the average password length was 6.8 characters and and 29% consisted of only lower-case characters. In 2002, a British on-line bank found that the passwords of 50% of its users were the names of family members, 8% were names of pets, and 9% were celebrity names.

To improve security, one must choose passwords well. Note that even if you choose your own password well it does not guarantee security, because if other passwords on your system are poorly chosen an attacker can enter, and once it has entered it may be able to exploit some security weakness in the system to extend its reach from the single account it has cracked to all accounts. So it is important to educate people to choose passwords well.

There are numerous methods to choose good passwords. One should make them long (8 characters or more), mix upper and lower case letters, numbers, and symbols, and avoid English words, people's names, celebrity names, phrases or words from movies, science-fiction novels or comics, and so on. Try to make it random. Of course, the downside is remembering your password, but this is a trade-off we need to make for security.

There are now several web-available password-cracking software packages that include a dictionary [5, 6]. These can be used to try to find passwords on any system. They are used as tools by system administrators to discover bad passwords and warn users to change them. Of course, they can also be exploited by attackers.

This is a good time to say something about the ethics of security. As a student in this course, you are asked and expected to make ethical use of any knowledge you obtain here. If you use or device a password finding tool, do it for good, meaning to help people improve their password choices, rather than doing it in order to break into systems.

2.3 Rationale for the salt

One question you may have is, why the salt? Because, if not, the dictionary attack becomes even more effective, as we now explain.

Let us consider the UNIX password system without the salt. Now, an entry in the password file has the form

`username` : hpwd ...

where $\mathsf{hpwd} = h(\mathsf{pwd})$. Suppose an attacker now creates a table T as follows:

```
For all $\mathsf{pwd}' \in \text{DICT}$ do
    $x \leftarrow h(\mathsf{pwd}')$ ; $T[x] \leftarrow \mathsf{pwd}'$
```

With this table in hand, a simple table lookup suffices to see whether the password of a particular user is in the dictionary. Given the above entry in the password file, the attacker simply sees whether entry $T[\mathsf{hpwd}]$ is defined. If so, the value of this entry is a password for this user, and the system is broken.

This is much more effective than the standard dictionary attack, because the table can be computed once and for all and stored. Then, a given password file on a given system can be quickly checked against the table to find passwords. In contrast, with a salt present, one has to compute such a table anew for each user name being attacked, using the salt associated to that user name, as in the Dictionary-Attack attack algorithm above. This means it takes much longer to attack a password file.

Of course, it is possible to build the table so that it covers all possible salt values:

```
For all $\mathsf{pwd}' \in \text{DICT}$ do
    For all $y \in \{0,1\}^{12}$ do
        Let $\mathsf{salt}$ be the 2-byte encoding of $y$
        $x \leftarrow h(\mathsf{salt}\|\mathsf{pwd}')$ ; $T[\mathsf{salt}\|x] \leftarrow \mathsf{pwd}'$
```

Remember the salt is a 12-bit value encoded into 2-bytes, so we are trying all $2^{12} = 4,096$ salt values above. Now, even with a salt present, a simple table lookup suffices to see whether the password of a particular user is in the dictionary. Given the $\mathsf{salt}, \mathsf{hpwd}$ from the password file, the attacker simply sees whether entry $T[\mathsf{salt}\|\mathsf{hpwd}]$ is defined. If so, the value of this entry is a password for this user, and the system is broken. However, building this table takes $2^{12} = 4,096$ times the amount of work and storage space as in the case where the salt is absent. Increasing the salt size makes this even more effective.

2.4 Protecting the password file

We have assumed the attacker has access to the password file. If the attacker did not have this file, it would not have the values of $\mathsf{salt}, \mathsf{hpwd}$, and then the attacker would not be able to mount a dictionary attack. Thus, we would like to deny it the password file. However, as we now discuss, it is more prudent to assume the attacker does have this file.

In the original UNIX system, the password file was not protected, meaning could be read by any user. This continues to be the case on many systems, where one can type `ypcat passwd` and obtain the entries of the password file. One might think that the attacker, however, does not have the file, for it is after all not a user; isn't the purpose of the attack to enter the computer? This argument is however not quite right. In fact, the attacker may be a user, trying to break in to another user's account. It may also be the case that password files entries are shared across different computers, and the attacker has an account on another computer having password file entries in common with the attacked one. It would thus be inadvisable to assume the attacker does not have access to the password file.

Nowadays, it is recommended that the system protect the password file. This means it is not readable by an ordinary user, only by a super-user. This certainly won't hurt, but it is not a cure. It is still prudent to assume that the attacker knows the password file. One reason is *insider attacks*. System administrators, being super-users, can read the password file. We might trust them as long as they are system administrators, but they might be fired and leave their jobs. They could take the password file with them, and start hacking it later. Or, they might be bribed by an attacker to reveal the password file.

The above exemplifies an important element of security reasoning, namely to consider a variety of threats. Insider attacks by disgruntled or bribed system administrators may not be the first threats that spring to the mind of someone inexperienced in security, but the truth is that insider attacks are an important contribution to the breaking of real world systems and must be addressed as well as possible.

2.5 Trojan horses

Here is another interesting class of attacks to consider. Say you are using a computer room like one of the undergraduate computer labs, where many people use the same terminal in the course of the day. Say the attacker is a user who has an account, but wants to obtain other people's passwords. The attacker writes a program which displays a login screen, with the usual prompts

ENTER USERNAME:
ENTER PASSWORD:

When you (the victim) come along, we think it is the usual login screen, and enter your user name and password. The attacker's program, however, simply copies your password to a file and saves it. It then returns some innocuous message, for example "system temporarily unavailable," and goes back to the fake login screen. You will just move to another terminal, thinking this one has a problem, not realizing that the attacker now has your password.

Can you think of ways of countering this attack? Consider the following. Is there a way to detect whether a terminal at which you are trying to log in is a trojan horse or the actual system? What about re-booting, meaning turning the system on and off?

2.6 Logging in over the network

Unix was designed before the Internet, and is a system for users to log on to physically accessed computers. With the advent of networking, people wanted to log on to systems remotely. This was done by using the existing authentication process, executed over the network. Namely, a remote user would send its user name and password to the computer across the communication link. The computer would grant or deny access according to the rules above. The telnet protocol does this.

Not a good idea. The problem is that the communication link over which the password is traveling could be vulnerable. An attacker can have access to this link, and put a "sniffer" on it, thereby being able to read all data sent across the link. It then picks up the password. For this reason, the use of telnet is not recommended, and it is even turned off on many systems. To access a computer remotely, you should use a tool that encrypts the password, for example ssh. We will learn more about these types of tools, that involve a lot of cryptography, later.

2.7 The password hashing function

Let us return to the function h for some remarks that move us further into cryptography. We are still postponing its description, which must wait until we have said what DES is, but we want to discuss its properties.

We said above it should be one-way, and we said this meant that given f it is hard to find a such that $h(a) = f$. Someone who thought about this statement may have realized that it is hardly precise. Surely, it cannot be hard to compute a from f for all value of f. For example, if $f = h(0)$, we know, given f, that $a = 0$ works. The precise definition of a one-way function, that we will see later, says that computing a from f should be hard when f is itself obtained by picking a *random* a' and setting $f = h(a')$. Further thought should then indicate that the relevance of this property to the security of the password system is unclear, for the hpwd values in the password file are not obtained as output of h on random values, but rather as outputs of h strings of the form salt‖pwd. But pwd is typically not random, and salt is known to the attacker. So the one-wayness of h does not necessarily imply that an attacker cannot invert it in the UNIX password system setting.

The actual h chosen by UNIX does appear to have the properties necessary for security of the password scheme, but what this discussion is trying to point out is that it is not clear what these properties are. We have not precisely *defined* them, making the task of analyzing the security of h hard.

A study of the UNIX password hashing algorithm that takes into account the above issues can be found in [7].

References

[1] W. Belgers. UNIX password security. `http://www.ja.net/CERT/Belgers/UNIX-password-security.html`.

[2] D. Feldmeier and P. Karn. UNIX password security, ten years later. *Advances in Cryptology – CRYPTO '89*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989. `http://www.ja.net/CERT/Feldmeier_and_Karn/crypto_89.ps`.

[3] D. Klein. Foiling the cracker: a survey of, and improvements to, password security. *Proceedings of the Usenix Workshop*, 1990.

[4] M. Luby and C. Rackoff. A study of password security. *Journal of Cryptology* Vol. 1, No. 3, 1989.

[5] A. Moffat. Crack version 4.1: A sensible password checker for UNIX. `http://www.crypticide.org/users/alecm/security/crack-v4.1-whitepaper.ps.gz`.

[6] John the Ripper password cracker. `http://www.openwall.com/john/`.

[7] D. Wagner and I. Goldberg. Proofs of security for the UNIX password hashing algorithm. *Advances in Cryptology – ASIACRYPT '00*, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed., Springer-Verlag, 2000. `http://www.cs.berkeley.edu/~daw/papers/crypt3-asia00.ps`.