Chapter 5

Authentication of People

Authorization, audit, and accountability all involve attributing actions to principals. For such attributions to be meaningful, the principals must first be authenticated. Methods to authenticate an important class of principals—people—are the subject of this chapter. The challenge is to devise schemes that exploit the strengths and accomodate the weaknesses of our human minds and bodies.

Computers are great for doing cryptographic calculations and have virtually infallible memories into which arbitrary bit strings can be stored for later retrieval. Humans are good at neither. So cryptographic protocols, which work well for authenticating computers, are impractical for authenticating people. Yet people do authenticate each other. We recognize the faces and voices of acquaintances; secret phrases or handshakes allow members of organizations to ascertain their shared affiliation; and you wear a school tie or sweatshirt to be recognized by others for where your studied.

Methods to authenticate people are grouped by security cognoscente into three broad categories.

> **Something you know.** You type a password to login to a computer or you enter a PIN to access your bank account from an ATM. []
>
> **Something you have.** You carry a wallet full of credentials (a driver's license, credit cards, a university id card) to certify your identities (as a driver, as a credit-worthy consumer, or as a student). []
>
> **Something you are.** You exhibit unique physical attributes (facial features, fingerprints, and DNA) and behavior (written signature, gait, voice) that others use to recognize you. []

And we can increase our confidence beyond what any single authentication method provides by requiring that multiple independent[1] methods be used to

[1] Recall from the Defense in Depth discussion (see page **??**) that two mechanisms are considered independent to the extent that an attack to compromise one is unlikely to compromise the other. For an authentication method, a *compromise* occurs if the attacker succeeds in impersonating somebody.

authenticate individuals. This is known as *multifactor authentication*, and the combination of two independent methods is known as *two-factor authentication*. Users of bank ATM machines are likely familiar with two-factor authentication involving "something you know" and "something you have". Here, ignorance of the "something you know" (a PIN) makes it difficult for an attacker to benefit from stealing the "something you have" (a bank card).

Privacy Pitfalls of Authentication. The goal of authenticating a person is to associate an identity with that principal, where we define an *identity* as any set of attributes and define an *identifier* as a name for an identity. The definition of identity, which might at first seem overly broad, is actually consistent with how this term is used informally. For example, you might have one identity as a driver (with attributes that include your name, an identfication number, and a record of driving violation convictions), another identity as a member of some political party (with attributes that include your name, your mailing address, and a list of financial donations you made to various candidates), and a third identity as a credit-worthy consumer (with attributes that include your name, an account number, a repayment history, and a credit limit).

Some of the attributes comprising an identity might be considered personal information. In learning that a person has been authenticated under that identity, we would then learn personal information. Authentication thus can lead to privacy violations. However, a user can exert control over which attributes are communicated to others simply by exercising care in choosing a suitable identity when being authenticated for each given activity—omit from that identity any personal information that should not be disclosed.

Good system design can go a long way towards reducing the potential for authentication to compromise privacy. Recommended practices here include:

> **Seek Consent.** Authenticate people only with their consent, and inform them if information about an identity will be saved. People using the system thus become aware they are relinquishing some control over the confidentiality of the personal information in that identity. []
>
> **Minimal Identity.** Authenticate only against identities that embody the smallest set of attributes needed for the task at hand. Personal information is thus not disclosed unnecessarily. []
>
> **Limit Storage.** Do not save information about authenticated identities unless there is a clear need. This reduces the chances that saved identity information can subsequently be retargeted for uses not implied by the user's consent that allowed its collection. []
>
> **Avoid Linking.** Eschew including the same unique attribute in different identities. A single, shared attribute allows linking the identities that contain this attribute, and that could violate privacy by revealing attributes comprising one identity to those who learn the other identity. []

To illustrate these system design practices, consider a magnetic-card scheme for limiting off-hours building-access to members of a community. With the system implemented at Cornell, each student and staff member is issued an id card; a unique id number (associated with the card owner) is encoded on each card's magnetic strip. Magnetic-card readers situated outside building doors are connected to computers, which are connected to actuators that control the door locks. Some of these computing systems log all building-access attempts. We offer the following observations.

> *Seek Consent.* Since one must take an action—place a card in a reader—to enter a building, authentication is being done with consent; no notice is being given about whether access attempts are being logged by the computers, however.
>
> *Minimal Identity.* We might debate whether the appropriate identity is being used for the authentication. Membership in the Cornell community is the sole attribute needed for building access, so an id card stating only that attribute should suffice. Since the id card in use stores a unique id number for the card holder, more information is being disclosed than necessary.
>
> *Limit Storage.* Should the system log the unique id number and time of day whenever a card holder enters a building? Such information might be useful if a crime is committed in that building, because this log identifies people that law enforcement officers might want to interview about unusual activity. However, this benefit must be weighed against the erosion of privacy and the potential for abuse that become possible once a building-entry log exists, since it now becomes possible to track an individual's comings and goings.
>
> *Avoid Linking.* Cornell's id cards are used in connection with a wide range of activities, such as borrowing books from the library, parking a car in the university garage, and charging food in certain university-run cafeterias. The unique id number on each id card enables linking an individual's actions, unnecessarily eroding privacy.

Credibility of an Identity. For an identity to be meaningful, its constitutent attributes must actually hold for any individual who is authenticated under that identity. For example, an individual authenticated under an identity containing name-attribute "Sherlock Holmes" and address-attribute "221b Baker Street, London" ought to be named Sherlock Holmes and reside at 221b Baker Street in London. Yet this correspondence between reality and a computer's representation of that reality cannot be established by software alone, since software has no way to ascertain the veracity of assertions about the physical world.

The solution is to employ an *enrollment protocol* to check that all attributes comprising an identity do indeed hold for an individual as a prerequisite to allowing authentication of that individual under the given identity. This enrollment

protocol is typically performed by people and often entails judgement, so mistakes are possible. Typically, the subject provides physical evidence to support a claimed identity; a clerk then evaluates that evidence and decides whether it is compelling (according to institutionally-defined criteria). A driver's license and a passport might, for instance, suffice to establish the subject's name, whereas DNA samples might be required to prove parentage.

Many enrollment protocols used in practice are easily subverted. Documents and other forms of physical evidence can be forged or altered. Moreover, documents often describe attributes of their subjects but do not offer proof that can be independently checked. A birth certificate, for example, is ulitmately an official form of hearsay—somebody claims to witness a birth and attests to the date, time, location, and identity of the child's mother.[2] Even the content of unaltered and valid official documents is frequently misconstrued. For example, increased confidence derived by requiring both a passport and a driver's license is unfounded when these documents are not independent, as is frequently the case.[3]

Vendors of authentication services typically offer a choice of enrollment protocols. The choices include various means for obtaining low-confidence judgements, often based on credentials submitted by mail, as well as some means that yield high-confidence judgements, perhaps based on in-person interviews and even full-fledged background investigations. However, it is not clear how a system might use information about enrollment confidence levels. Systems are instead more likely to implement a suite of authentication methods and associate with each application an authentication method yielding a level of confidence deemed suitable for whatever task is implemented by that application. For example, banks and other financial services institutions often require high-confidence enrollment protocols and high-confidence authentication for customers making large withdrawls, but these same systems are satisfied by low-confidence authentication for customers making deposits.

5.1 Something You Know

Knowledge-Based Authentication. With *knowledge-based authentication*, each individual, as part of the enrollment protocol, provides the system with answers to a set of queries; thereafter, to authenticate that individual, the system poses some subset of those queries and compares the responses it receives with the anwers that were provided at enrollment. Queries are typically derived from an individual's habits and history, such as family (e.g., "What is your mother's

[2] In contrast, DNA testing provides an independently verificable proof of a highly probable biological connection between two people whose blood samples are available. One need only draw new blood and rerun the test.

[3] A birth certificate typically suffices for establishing your name and age when obtaining a driver's license, and a birth certificate and driver's license together suffice for obtaining a passport. With the passpoort contingent on the drivers license, they are not independent and there is no basis for increased confidence from having both.

maiden name?"), life experiences (e.g., "Have you ever lived in Ithaca?"), or current circumstance (e.g., "What is your place of employment?").

The most useful queries are those with "correct" answers that are not broadly known and cannot easily become known to an attacker attempting to impersonate someone.[4] Thus, finding good queries becomes difficult when there is a wealth of publicly available information about individuals. And the trends here are not favorable, with the ever-expanding world wide web and with new cultural norms that encourage details of individuals' lives to be revealed through participatation in on-line communities. Web search engines place at an attacker's fingertips the answers to an awful lot of the queries that knowledge-based authentication systems have tended to use.

Any knowledge-based authentication implementation must have access to "correct" answers for the queries it asks. This can be objectionable to people who are concerned about having personel information aggregated and stored, even though that information would presumably be stored in a way that prevents it from being read (except when a subject claiming to be that individual is being authenticated) and prevents it being written (except when an update is required, because the "correct" answer to a query has changed).

Knowledge-based authentication is certainly convenient, since it imposes little burden on individuals to remember things, carry things, or be subjected to biometric measurement. Moreover, by asking more questions of a subject, the confidence level for an authentication can be increased. But the approach ultimately depends on attackers not having access to information, which could be a flawed assumption. Moreover, once the "correct" answer to a knowledge-based authentication query for a given individual becomes known to an attacker, the only recourse is to invent new questions and use them for authenticating that individual. Such questions tend to become increasingly esoteric and ultimately impose a memory burden on the individual who must remember the answers.

Secret-Based Authentication. People can be authenticated on the basis of secrets they know, provided the secrets are not known to attackers and are difficult to guess or steal. We simply check that a person being authenticated as some identity knows the unique secrets associated with that identity. Embodiments of such secrets include *personal identification numbers* (PINs), which are typically 4 digits long; *passwords*, which are likely at least twice as long and contain any of the characters found on the keyboard; and *pass phrases*, which are significantly longer still and again can involve any characters found on the keyboard. We use the term password in this section, but what is said applies to PINs, pass phrases, and the other forms of secrets used for authenticating humans.

The implementation of a password scheme can be surprisingly subtle. People's difficulties with memorizing seemingly random strings coupled and the

[4] There is, for example, little benefit in asking violinist Nadja Solerno-Sonnenberg about her mother's maiden name, which would likely be "Solerno" given the hypnenated construction of her last name.

ever increasing power of computers mean that implementing a good password authentication scheme requires more than just storing passwords that users select and comparing what is stored with passwords that users enter. Those subtleties are discussed in the remainder of this section.

5.1.1 Choosing a Password

An obvious way to generate a password is to choose a string randomly from some large space of possible choices. Given a set comprising N characters, there are a total of N^L length L sequences of characters. For example, with the 52 lower- and upper case alphabetic characters plus the 10 digits found on a computer keyboard, the set of strings compising 10 characters contains $(52+10)^{10}$ or equivalently 839,299,365,868,340,224 (approximately 8.3×10^{17}) elements. Include punctuation symbols and blank spaces, which increasingly are being permitted in passwords, and the number of strings grows still larger.

A password can be generated by a computer, invented by the person it will be used to identify, or furnished by the system operator. None of these approaches is a panacea.

- People are notoriously bad at memorizing random strings, and passwords generated by a computer will seem random.[5] So computer-generated passwords are hard to memorize, hence likely to be written down. A password that is written down can be seen by others and stolen.

- Passwords invented by people are usually devised to be easy to remember—a word in the dictionary, a loved-one's name, a telephone number, a keyboard pattern (e.g., "asdf"), or some combination thereof. Unfortunately, a password drawn from that significantly smaller space will be considerably easier to guess.[6]

- A password furnished by a system operator must be either computer-generated or selected by some person; one of previous two cases then applies. Moreover, an operator-selected password is best changed immediately by the recepient, to prevent the operator from subsequently abusing knowledge of that password. One of the above cases would apply to that new password, too.

Usability typically trumps security, so most systems permit passwords invented by people. Additional means are then deployed to prevent attackers from

[5] Some have suggested employing rules for producing random passwords that are easy to remember because they are pronouceable. The rules might, for example, require every consonent be followed by a vowel and prohibit any two adjacent letters from both being vowels or consonents. However, imposing such restrictions reduces the amount of work required for an attacker to enumerate the space of possible password choices.

[6] The 20 volume *Oxford English Dictionary* (second edition) contains in excess of 5×10^5 words. Compare that with the 146,813,779,479,510 (approximately 1.46×10^{14}) strings having 10 or fewer alphabetic characters, and it becomes clear that there are considerably more random alphabetic strings than dictionary words.

successfully guessing these *weak passwords*, so named because of the relatively small space of likely password choices.

In *on-line guessing* attacks, the system itself is used by the attacker to check password guesses. One defense here is to limit the rate or number of password guesses allowed:

- Make authentication a time-consuming process by requiring that passwords be entered manually and/or that a long computation be performed by the system in checking a password.
- Impose a limit on unsuccessful guesses and, when that limit is reached, disconnect from the source and refuse later attempts to authenticate that identity (until a system operator intercedes).

By permitting fewer guesses, these measures reduce the chances that an attacker will succeed in making a correct guess. The second measure also increases the chances an attack will be detected while it is in progress or soon thereafter. Note, however, that limiting the number of unsuccessful password guesses for a given identity creates an opportunity for denial of service attacks; an attacker whose incorrect password guesses exceed the limit then blocks all authentications for that identity, which prevents some bona-fide user from accessing the system.

By restricting what information is returned when an authentication attempt fails, we further impede on-line guessing. Some systems prompt for the identifier being authenticated and immediately reply with an error if that identifier is not known to the system. Such replies confirm identifiers that could serve as targets of attack. A better design is for the system to request the identifier and corresponding password together; only after receiving both, does the system reply. And if either the identifier is unknown or the password is erroneous then a single, minimally-informative response is returned (e.g. "Authentication attempt failed.").

The system's response to a successful authentication attempt should indicate success, but we benefit if it also gives (i) the time when the last successful authentication with this identifier occured, and (ii) the times for any unsuccessful authentication attempts made since that last successful authentication. In reporting the last successful authentication, a user (who presumably will recall when he last authenticated that identifier) can detect a prior successful on-line guessing attack; in reporting past unsuccessful authentications, the user becomes aware that on-line guessing attacks had been attempted.

The strongest defense against attackers guessing weak passwords is to enforce a prohibition against users choosing them. To implement this, rules, such as those in Figure 5.1, are employed to reject easily guessed passwords. Some of the rules are obvious (e.g., avoid passwords that appear in a dictionary); other rules correspond to memory tricks that people use and attackers know people use (e.g., replacing a letter by a number that looks similar, such as replacing "e" in by "3" or "l" by "1"). No set of syntactic rules, however, will completely characterize easy to guess passwords, and laziness about memorization provides a strong incentive for discovering classes of passwords that are easy to remember

1. Strings easily derived from words or sequences of words appearing in an English or foreign language dictionary, because they appear verbatim or reversed, perhaps with the following transformations applied:
   (a) deleting vowels or spaces,
   (b) capitalizing some letters,
   (c) adding a suffix and/or prefix,
   (d) replacing one or more letters by similar looking non-alphabetic characters (e.g., "0" for "o") and/or by homophonic letters or clauses (e.g., "4" for "for").
2. Strings derived (in the sense of 1) from the user's identity—name, initials, login id, or other attributes.
3. Strings derived (in the sense of 1) from acronyms, names of people, or names of places
4. Strings shorter than a certain length (e.g., 8 characters).
5. Strings that do not contain a mix of upper- and lowercase characters.
6. Strings produced by typing simple patterns on the keyboard.
7. Strings comprising all numbers.
8. Strings that mix numbers and characters but resemble license plates, office numbers, etc.

Figure 5.1: Rules Characterizing Weak Passwords

but the rules do not reject. Sooner or later attackers too will discover these password classes and include such passwords in on-line guessing attacks.

5.1.2 Storing Passwords

The obvious scheme for storing passwords is to use a file that contains the set of pairs $\langle uid_i, pass_i \rangle$, where uid_i is an identifier and $pass_i$ the associated password. The file system's authorization mechanism is then used to restrict which principals can access this *password file*.

- An ordinary user must not be able to read the password file or its backups, in order to prevent that user from obtaining the password for any identifier.
- An ordinary user must not be able to write the password file or its backups, in order to prevent that user from changing the password for an identifier (to then know that password).
- The program that authenticates users must be able to read the password file.

1. $H \longrightarrow System$: uid, $pass$
2. $System \longrightarrow H$: $\langle uid, \mathcal{H}(pass)\rangle \in HashPwd$

Figure 5.2: Password Query-Response Protocol

- The program used to add/remove user identifiers and reset passwords must be able to write the password file.

Security here requires trusting that the file system authorization mechanism has no vulnerabilities, that the system operator correctly sets the access control list for the password file whenever it is edited and stored, and that traces of file contents are not left to be read from unallocated memory or unallocated disk pages that previously stored the password file. Some find the need for this trust troubling, but if the trust is not misplaced then storing passwords in this way leaves only one avenue to attackers: on-line guessing.

An alternative for protecting the confidentiality of passwords is to compute a cryptographic hash function $\mathcal{H}(pass_i)$ for each each password $pass_i$ and store the set of pairs $\langle uid_i, \mathcal{H}(pass_i)\rangle$ in the password file. Because by definition $\mathcal{H}(\cdot)$ is a one-way function, knowledge of $\mathcal{H}(pass_i)$ reveals nothing about $pass_i$. Reading such a password file is no longer helpful to an attacker in search of passwords.

Letting *HashPwd* denote the set of $\langle uid_i, \mathcal{H}(pass_i)\rangle$ pairs stored in the password file, the protocol for a human H to be authenticated by the system will return *true* if and only if H enters some identifier uid and password $pass$, and the system determines that $\langle uid, \mathcal{H}(pass)\rangle \in HashPwd$ holds. Thus, the authentication protocol simply returns the value of the predicate $\langle uid, \mathcal{H}(pass)\rangle \in HashPwd$, as depicted in Figure 5.2.

It is not unusual to use MD5 or SHA-1 for $\mathcal{H}(\cdot)$. Another, albeit slower, alternative is to use a shared key encryption algorithm configured to encrypt some constant C, using $pass$ as the key; early versions of UNIX implemented $\mathcal{H}(\cdot)$ by iterating DES several times with 0 for C. Recall, faster is not necessarily better when it comes to the authentication—on-line guessing attacks are hindered by slower processing of authentication requests, because then fewer guesses can be attempted in a given interval.

Salt. Even when passwords are not stored in the clear, unrestricted read access to the password file enables *off-line guessing attacks*, also known as *dictionary attacks*. The attacker first obtains a list of candidate passwords or constructs such a list[7] with rules like those in Figure 5.1; this list is then used by the attacker in computing a *dictionary*, which is a set *Dict* of pairs $\langle w, \mathcal{H}(w)\rangle$ for each word w in the list of candidate passwords. Finally, the attacker exploits the unrestricted read access to *HashPwd* on the system to be attacked and generates

[7] In 2007, lists found on attacker web sites contained as many as 40 million entries.

the set of pairs

$$\{\langle uid, pass\rangle \mid \langle uid, \mathcal{H}(pass)\rangle \in \mathit{HashPwd} \ \wedge \ \langle pass, \mathcal{H}(pass)\rangle \in \mathit{Dict}\} \qquad (5.1)$$

by identifying elements having the same $\mathcal{H}(pass)$ value in both *HashPwd* and *Dict*. Each $\langle uid, pass\rangle$ pair in (5.1) enables the attacker to impersonate identifier *uid*.

The computation of *Dict* and (5.1) constitute a non-trivial up-front cost to the attacker. However, such an investment can be justified:

- The work to compute *Dict* is amortized, because the single *Dict* suffices for attacking every system whose password file is computed using cryptographic hash function $\mathcal{H}(\cdot)$. All instances of a given operating system typically use the same hash function, and therefore a single *Dict* can be used in attacking any target running that operating system.

- Beyond obtaining a copy of *HashPwd*, the computation involved in actually attacking a given system—the construction of (5.1)—need not be performed on that target system. The chance an attack will be discovered, hence the risk to the attacker, is thus reduced. Moreover, the attacker is now not limited by a target system's limited computational power or deliberately slow implementation of $\mathcal{H}(\cdot)$.

- The attack is likely to yield multiple user identifiers with passwords for the target system, so it provides the attacker with multiple avenues of compromise. This is ideal for the attacker not needing to impersonate a specific user, and many attacks only require initial access that impersonating an arbitrary user provides. So the offline guessing attack succeeds if any user identifier has a weak password.

One defense against off-line guessing attacks is to change the password file in a way that makes the computation of (5.1) infeasible. To accomplish this, we might store with each uid_i a nonce n_i, called the *salt*, and append that nonce to $pass_i$ when computing cryptographic hash function $\mathcal{H}(\cdot)$. The password file now stores a set *HashSaltPwd* of triples, with a triple $\langle uid_i, n_i, \mathcal{H}(pass_i \cdot n_i)\rangle$ for each user identifier uid_i. Early versions of Unix used 12-bit numbers for salt; the nonce for a given *uid* was obtained by reading the real-time system clock when creating the account for *uid*.

We extend the protocol of Figure 5.2 as shown in Figure 5.3 to accomodate the per *uid* salt, where *HashSaltPwd*.`get`(*uid*) is a method that returns the unique triple in *HashSaltPwd* having the form $\langle uid, \ldots\rangle$.

With b bit nonces for salt, off-line guessing requires the attacker to compute in place of *Dict* the considerably larger set *SaltDict* of triples $\langle v, \mathcal{H}(w \cdot n)\rangle$ for each candidate password w and each value n satisfying $0 \le n \le 2^b - 1$. So *SaltDict* is 2^b times as large as *Dict*. Analogous to (5.1), the attacker constructs the set of pairs

$$\begin{aligned}\{\langle uid, pass\rangle \mid \quad & \langle uid, n, \mathcal{H}(pass \cdot n)\rangle \in \mathit{HashSaltPwd} \\ \wedge \ & \langle pass, n, \mathcal{H}(pass \cdot n)\rangle \in \mathit{SaltDict}\}\end{aligned} \qquad (5.2)$$

1. $H \longrightarrow System$: uid, $pass$
2. $System$: $\langle uid, n, p\rangle := HashSaltPwd.\texttt{get}(uid)$
3. $System \longrightarrow H$: $\mathcal{H}(pass \cdot n) = p$

Figure 5.3: Password Query-Response Protocol with Salt

1. $H \longrightarrow System$: uid, $pass$
2. $System \longrightarrow H$: $(\exists\, n :\ \langle uid, \mathcal{H}(pass \cdot n)\rangle \in HashPepperPwd)$

Figure 5.4: Password Query-Response Protocol with Pepper

and element $\langle uid, pass\rangle$ in that set enables the attacker to impersonate corresponding user uid.

Pepper. The cleartext salt in *HashSaltPwd* does not defend against what we will call a *limited* off-line guessing attack, whereby the attacker computes only that subset of *SaltDict* containing triples $\langle v, \mathcal{H}(w \cdot n)\rangle$ for which b bit salt n appears in *HashSaltPwd* on the target system. If *HashSaltPwd* contains N elements then this subset of *SaltDict* is only N times the size of *Dict*. Since $N \ll 2^b$ likely holds, the attacker replaces an infeasible computation with one that is feasible. Specifically, the attacker incurrs a cost proportional to $N|Dict|$ for computing a *SaltDict* that works against one target instead of a cost proportional to $2^b|Dict|$ for the *SaltDict* that works against all targets.

To defend against limited off-line guessing attacks, we might store a set *HashPepperPwd* of pairs $\langle uid_i, \mathcal{H}(pass_i \cdot n_i)\rangle$, where nonce n_i, called the *pepper*, is not stored elsewhere in the tuple for uid_i (as salt would be). A protocol for authenticating uid given a password $pass$ is given in Figure 5.4; step 2 returns *true* if and only if there is some pepper n for which $\langle uid, \mathcal{H}(pass \cdot n)\rangle$ is an element of *HashPepperPwd*.[8]

When pepper is used, the amount of work required to build a dictionary *PepperDict* for an off-line guessing attack is proportional to the number of pairs $\langle w, \mathcal{H}(w \cdot n)\rangle$, for w ranging over the candidate passwords and n ranging over possible pepper values. For b bits of pepper, this means that *PepperDict* is a factor of 2^b larger than *Dict*; the attacker's work is increased proprotionally. Moreover, unlike salt, pepper defends against limited off-line guessing attacks, because the absence of cleartext pepper in *HashPepperPwd* means an attacker has no way to generate the target-specific smaller dictionary that enables a limited off-line guessing attack. So b bits of pepper consitutes a better defense

[8]To determine whether such an n exists, the system must enumerate and check possible values. This search is best started each time at a (different) randomly selected place in the enumeration of possble pepper values; otherwise, the delay to authenticate uid would convey information about whether n is early or late in the enumeration of possible pepper values, thereby conveying information about the value of n and reducing the size of the search space, hence the work required of an attacker.

1. $H \longrightarrow System$: uid, $pass$
2. $System$: $\langle uid, s\rangle := HashSpicedPwd.\texttt{get}(uid)$
3. $System \longrightarrow H$: $(\exists\, ppr :\ \langle uid, s, \mathcal{H}(pass \cdot s \cdot ppr)\rangle \in HashPepperPwd)$

Figure 5.5: Password Query-Response Protocol with Salt and Pepper

than b bits of salt.

Pepper is not a panacea, though. The number of possible pepper values affects the time required to perform step 2 of the authentication protocol in Figure 5.4. If this delay is too large, then users lose patience. So the number of bits of pepper must be kept small enough so that the time it takes to enumerate possible pepper values in step 2 remains acceptable. The potency of pepper as a defense has a ceiling. Compare this to salt. The number of possible salt values is virtually unbounded, being constrained only by the storage required for *HashSaltPwd*. So with salt, a stronger defense can always be deployed just by increasing the number of bits for the salt.

Putting it Together. By combining salt and pepper, we overcome limitations of each. In the combined scheme, salt $salt_i$ and pepper ppr_i are both inputs to the cryptographic hash computed for each password $pass_i$; passwords are stored in a set *HashSpicedPwd* of tuples $\langle uid_i, salt_i, \mathcal{H}(pass_i \cdot salt_i \cdot ppr_i)\rangle$; and the protocol of Figure 5.5 is used. If there are bs bits of salt, bp bits of pepper, and N tuples in *HashSpicedPwd*, then a dictionary *SpicedDict* for an off-line guessing attack would be 2^{bs+bp} times larger than $|Dict|$, and the subset required for a limited off-line guessing attack would be $2^{bp}N$ times larger than $|Dict|$. Off-line attacks can be made infeasible by picking a large enough value of bs. But the defense against limited off-line guessing attacks depends on bp which, unfortunately, is limited by the processing speed of the target machine. Therefore, the prudent designer of a password storage scheme will employ defense in depth and use file system authorization to restrict read access to a password file that stores salted, hashed passwords.[9]

5.1.3 Authenticating Requests for Passwords

A program that requests your password might be running on behalf of an operating system trying to authenticate you. Or, it might be running on behalf of an attacker attempting to steal your password. You should provide your password to one and not the other. So having some means by which a human user can authenticate the source of any request for a password is crucial.

[9]The shadow password file `/etc/shadow` in LINUX and `/etc/master.passwd` in BSD Unix systems are read protected, and they store hashed, salted passwords. Both systems continue to support `/etc/passwd`, which is generally readible but does not contain any form of password (hashed or salted).

1. *User*: Activate the *secure attention key* (`SAK`) at the keyboard.
2. *OS keyboard driver*: Intercept each `SAK` activation and start the OS authentication routine. It displays a password request prompt (perhaps in a separate window) and then monitors the keyboard for the user's response.
3. *User*: Enter a userid and corresponding password.
4. *OS keyboard driver*: Forward that userid and password to the OS authentication routine.
5. *OS authentication routine*: Validate the userid and password.

Figure 5.6: Trusted Path Components and Interactions

Trusted Path. A *trusted path* is a trustworthy communications channel from an input device, like a keyboard, to a known destination program, like an operating system's authentication routine. One way that users can avoid being spoofed by password requests from an attacker's program is to open such a trusted path and respond along that. However, this defense requires a trustworthy mechanism users can invoke to open that trusted path.[10] The components and their interactions are given in Figure 5.6.

The security of this scheme depends on running a trustworthy keyboard driver and authentication routine. An attacker who alters either can steal passwords by creating a system that, to a human user, appears to behave just like what is outlined in Figure 5.6. You might think that rebooting the computer before activating the `SAK` would defend against such a compromised keyboard driver or authentication routine. However, it doesn't if the attacker has corrupted the version of the operating system stored on disk. Nor does rebooting from some other (known to be clean) boot media suffice if the attacker has modified the boot loader to read the corrupted image on disk no matter what boot image is specified. Defend the boot loader and then by exploiting BIOS vulnerabilities, an attacker can force the wrong boot loader to execute; defend the BIOS, and an attacker's ability to modify the hardware remains problematic.

Physically limiting access to the hardware does prevent an attacker from modifying that. And trustworthy hardware exists (and is increasingly common) for building a chain of trustworthy system layers culminating in a trusted path that itself is trustworthy.[11] Furthermore, even when all layers are not themselves trustworthy, repositioning a vulnerability from a higher layer (e.g., the operating system) down to a lower layer (e.g., the BIOS) could well increase the security of the system relative to a given threat. This is because exploiting the vulnerabilities at lower system layers requires considerable sophistication, which might exceed the capabilities of some threats. Fix Fwd ref

[10] In Microsoft's Windows NT operating systems, for example, typing `Ctrl-Alt-Delete` serves this purpose.

[11] Such hardware is the subject of section xxxx.

Visual Secrets. Appearances are another means for users to authenticate the source of a password request is order to decide whether that request is legitimate. The manner in which the request is displayed serves here as a *visual secret* that is shared between the requestor and responder; that secret might be a formatting convention, some specific text to display, and/or an image to present. A (computer) requestor demonstrates knowledge of the visual secret by the content of the request and how it is rendered on the display. A (human) responder validates this knowledge by inspecting the request; a password request that departs from what is expected is then treated with suspicion and ignored.

When visual secrets are being used, an attacker's program for making a request must spoof knowledge of the visual secret. Such attacks are made more difficult by two practices.

- *Share a different visual secret with each user.* The protocol for a password request might then involve two separate steps: (i) request a userid, and (ii) render the password request according to the shared visual secret (e.g.,, the image of a scene that user had selected) for that userid.[12] Now to succeed, the attacker's program must know the visual secret shared with each user to be spoofed, which presumably would require compromising a file system authorization mechanism.

- *Prevent arbitrary programs from rendering certain text or images in certain regions of a display.* By including one of these controlled outputs in a password request, we then prevent an attacker's program from rendering a legitimate request—whether or not the attacker's program knows the associated visual secret.

A human user's ability to distinguish among possible renderings of a request is critical in order for a visual secret to authenticate the source of that request. Our tendancy to overlook small details when reading text or viewing images thus can be a handicap, leading to a vulnerability. For instance, most people will not differentiate between displayed characters and a displayed image containing those same characters, which means that a program, by displaying images, can circumvent defenses that prevent the display of certain character sequences. People also don't look carefully at status fields to see whether they display expected values. Thus, even a tamper-proof means for displaying the name of a requestor as the shared visual secret fails to defend against bogus requestors whose names are spelled similarly to *bona fide* requestors.

Our insentivitiy to rendering, for example, helps enable *phishing* attacks, where attackers attempt to acquire private information by directing users to a web site designed to resemble the web site for some legitimate institution. Users are induced to visit the attacker's web site by being presented with a link that closely resembles a link for the site being spoofed (e.g., `www.paypa1.com` versus `www.paypal.com`), they are not careful about checking the various visual secrets

[12] To prevent attackers from determining whether a particular user id is valid, the system must not behave differently in step (ii) when a unknown user id is entered in step (i). One solution is to invent and use a new shared visual secret in that case.

web browsers display when visiting that (attacker's) site, and they enter private information (a password, perhaps) which then becomes known to the attacker.

Although users today are not particularly good at detecting small differences in renderings, they are quite good at recognizing and distinguishing pictures of different scenes or people. Thus, certain visual secrets—those involving larger, richer images—do work as a means for humans to authenticate the source of a request. Moreover, improvements in user interfaces in conjunction with an expanding population of more-experienced users ought to mean that, in the future, users should get better at detecting less pronounced but neverthless significant differences in how requests are rendered. In short, the future for visual secrets looks promising.

5.1.4 Password Pragmatics

For most of us, a single password is far easier to devise and remember than multiple passwords. So we are tempted to stick with one password, reusing it for all our various different identities (e.g., computer accounts on different machines, various web sites, etc.). Security suffers.

- If passwords have only a limited lifetime, then a compromised password sooner or later would become useless to an attacker—whether or not that password's compromise ever becomes known.

- If separate passwords are employed for different identities, then an attacker who compromises one password could impersonate only the one identity. Compare this with the attacker's pay-off when a number of identities all have the same password; compromising that password enables all of those identities to be impersonated.

So passwords should be changed regularly, and separate identities should have separate passwords. How might systems support these practices?

Password Expiration. Some systems periodically force users to select new passwords—either after a fixed interval has elapsed (typically, months) or after the password has been used a number of times. The system might even store a user's expired passwords and prevent new passwords from resembling their predecessors, although this functionality can often be circumvented by resourceful users—for example, if passwords expire every month, then `Mycroft01`, `Mycroft02`, ..., `Mycroft12`, are easy to derive once `Mycroft` has been memorized, and might each be accepted by the system as different enough. Unfortunately, passwords forming a sequence that is easy for a user to generate are probably also easy for an attacker to reconstruct if some password in the sequence has been compromised. An attacker who successfully compromises `Mycroft02` in February, for instance, would be wise to try `Mycroft05` in May.

Infrequent users pose a special problem for password expiration schemes. Such a user might well resent having to access the system periodically solely for

the purpose of replacing a password before it expires. Yet to permit authentication based on an expired password—even if the authenticated user must immediately selects a new password—is risky, because an attacker who has compromised an old password would now get to pick the new password, thereby circumventing the defense that password expiration was intended to provide in the first place.

Password Hashes. Human principals can, with a minimum of effort, have a distinct password $pass_I$ for each of their identities I by (i) memorizing a single secret s and (ii) combining s and I to derive the password: $pass_I = \mathcal{F}(s, I)$. You might, for example, select `Moriarity` as your secret, memorize it, and then (as needed) produce password `MWoWrWiEaBrYiCtOyM` for your identity at Ebay, `MWoWrWiCaNrNiCtOyM` for CNN, and so on.[13] However, as noted above, such schemes risk producing weak passwords if a single compromised password provides enough information about secret s and combining function $\mathcal{F}(\cdot)$ so that an attacker can easily guess that principal's passwords for other identities.

The weakness in this approach stems from the choice of $\mathcal{F}(\cdot)$, which is likely more dictated by our human nature to shun doing mental and/or manual calculations than anything else. Were $\mathcal{F}(\cdot)$ selected to be non-invertable, an attacker who compromised one or more passwords would learn nothing about the underlying secret, even knowing $\mathcal{F}(\cdot)$; the attacker is then no better off for synthezizing other passwords generated from that same secret. Cryptographic hash functions are non-invertable. Thus, given a cryptograhic hash function $\mathcal{H}(\cdot)$, a strong password for each of a principal's identities would be obtained if the principal memorizes a single secret s and uses *password hash* $\mathcal{H}(s \cdot I)$ as the password $pass_I$ for identity I.

Few humans will be able to calculate $\mathcal{H}(s \cdot I)$ in their heads, nor would very many be willing to undertake a paper and pencil computation every time they must be authenticated. This restricts the applicability of password hashes to settings where some form of computer is available to aid in generating passwords. The human principal provides the secret s (from memory), and inputs it along with the identiy I for which a password is sought; the computer produces the password by evaluating $\mathcal{H}(s \cdot I)$ and then erases copies of s.

Password hashes are the basis of a web browser plug-in to enable users who memorize a single secret to authenticate themselves using different passwords at different web sites. Password fields in web page forms are delimited by a special HTML tag. This tag allows the plug-in to immediately replace any text *pass* a user enters into such a password field with $\mathcal{H}(pass \cdot url)$, where *url* is the url of the web page being visited. Th form sent back to the web site thus contains a site-specific passwords, even though the human user has not memorized or entered any site-specific passwords.

[13] Here, function $\mathcal{F}(s, I)$ interleaves its arguments s and I. For example, `MWoWrWiEaBrYiCtOyM` $= \mathcal{F}($`Moriarity`, `WWW.EBAY.COM`$)$.

Password Hygiene. To focus only on system mechanisms for preventing password compromise ignores the other potential target of attack: humans who might be induced to reveal their passwords. The defense against such attacks is educated users and adopting practices that discourage certain easily exploited user behaviors. Both are discussed in what follows.

Sharing Passwords. The human penchant for being helpful sometimes can be turned into a vulnerability. A common attacker's ploy is to telephone a user and pretend to be a system operator needing that user's password in order to fix an urgent problem. The user, who like most people wants to be helpful, reveals his password. In a variant, the attacker calls a system operator, pretending to be a *bona fide* user who has an urgent need to access the system but has forgotten his password. The operator, trying to be helpful, resets that user's password and informs the attacker of the new password. Neither of these attacks could succeed if system designs never required one user to impersonate another and, therefore, users could be educated never to reveal their passwords to others.

Some users who share their passwords with others are simply confused about the difference between authorization and authentication. We all have occassion to leave our house keys with people we trust, so they can take in the mail or feed Mena the cat. The lock on the front door is an authorization mechanism; it grants access to those who possess our house key. So a password, because it unlocks access to a computer, is equated (erroneously) with a house key; sharing the password is then presumed (erroneously) to be the way others are granted computer access. In fact, a computer's login authorization mechanism grants computer access to users having certain identities, and the password is simply an input to an authentication mechanism (not to the authorization mechanism) for establishing that identity. So sharing your password is tantamount to granting your identity to somebody else—a far cry from granting computer access to somebody else (as was presumed). Although the real problem is user education, notice that this misguided behavior is not available to users practicing the dictim about never revealing their passwords to others.

Recording Passwords. Few people are able to memorize even a modest number of strong passwords. That reality forces the rest of us to choose between two options:

1. employ only a few strong passwords (because memorizing those is a managable prospect) rather than having a separate password for each identity, or

2. record passwords in some written or electronic form, so there is no need to memorize them.

Each option brings vulnerabilities that enable a password-based authentication mechanism to be compromised.

Option 1 sacrifices a form of defense in depth. When the same password is used for multiple identities, then by compromising the password for one identity an attacker can be authenticated under those other identities as well. If different identities have access to different resources, then this undermines the Principle of Least Privilege. Yet there are contexts where having the same password for multiple identities is perfectly reasonable—notably, situations where having multiple, distinct identities is not the user's choice. For example, newspapers and other web-based information providers create an identity for every user they serve, and these sites typically employ password authentication for those users; a user who would have been satisfied with having a single identity to access all those news sites might well employ the same password for all of the identities these sites imposed.

With option 2, our concern should be defending against attackers accessing the password list. One classic example is the handwritten note so often found affixed to a system console in a computer room and giving the passwords for system operator accounts. Anyone who gains entry to that computer room (say, by convincing the janitor to unlock the door) can use this password to become a system operator. Another example is the printed list of passwords so carefully locked away in a cabinet until it is discarded (without first being shredded), so it becomes available to any attacker willing to rummage through the trash.[14] Encryption and physical means (i.e., physical lock and key) are both viable ways to protect password lists. When passwords are being stored in encrypted form, we should be concerned about vulnerabilities that cause the decryption key to be revealed; when physical security is used, our concern should be with attackers circumventing that structure.

Observing Passwords. Even a password that has not been written down is potentially vulnerable to theft by attackers who can observe or monitor emanations from a system. Some of these attacks are straightforward; others require considerable expertise in signal-processing:

- An attacker might peer over the user's shoulder and watch the password being typed or might use a telescope to watch from a distance. This is known as *shoulder surfing*. The obvious defense is to shield from view the keyboard being used to enter a password. Keyboards on ATM machines, for example, are often situated so they will be obscured by a user's body.

- Instead of watching the keyboard, an attacker might instead observe the system's display as a password is typed. In fact, direct visual observation is not required. Researchers have been able to recover the contents of a CRT display by monitoring the diffuse reflections from walls, and they have been able to recover the contents of an LCD display by monitoring RF radiation from the video cable connecting the LCD to the computer.

[14] The practice of combing through commercial or residential trash in search of confidential information is sometimes referred to as *dumpster diving*.

Here, the obvious defense is not to echo the typed characters on the display when a password is entered.

- Individual key clicks reveal information about what keys are being depressed, and monitoring systems have been devised to do the signal processing needed recover a user's keyboard inputs from recordings of such acoustic emanations. Background noise (or music) is easily filtered out, so that is not a very effective defense. Physical security to prevent the deployment by attackers of microphones or other listening devices seems to be the best defense against such attacks.