Chapter 7. Passwords and Authentication

> "I haven't told him about you, but I have told him to trust absolutely whoever has the key word. You remember?"
>
> "Yes, of course. *Meshuggah*. What does it mean?"
>
> "Never mind." Abrams grinned.
>
> *Ensign Flandry*
> —POUL ANDERSON

7.1 Authentication Principles

Authentication is generally considered to be one of the most basic security principles. Absent bugs—admittedly a very large assumption—authentication effectively controls what system objects someone can use. In other words, it's important to get authentication right.

Most discussions of authentication start by describing the three basic forms: something you know (e.g., a password); something you have, such as a token or a particular mobile phone; and something you are, that is, some form of biometric. While this categorization is indeed useful, it understates the *systems* nature of authentication. The total environment—who will use it, how you deal with lost credentials, what the consequences are of lack of access or access by the wrong person, and more—is at least as important. The most important question of all is how people will actually use the authentication technology in the real world.

Another important thing to remember: we authenticate in many more situations today than we did in the not very distant past. Once upon a time, we would log in to a work machine or two. Now, we log in to many different web sites, mail systems, devices, doors, and even cars. The challenges, and hence the solutions, can differ.

7.2 Passwords

This book is about demythologizing security. Few areas are in sorer need of that than passwords. The trouble started with a classic—and still correct—paper by Morris and Thompson [1979], which among other things showed why guessable passwords were bad. However, that result is often misapplied today; in particular, insufficient attention is paid to the threat model (what assets you are protecting, and against whom) and to the tension between security and usability. ([Singer, W. Anderson, and Farrow 2013] gives a good presentation of the history of how we got to where we are and of the many mistakes and unjustified assumptions made along the way.)

The problem of password strength is easy to explain. Simple experiments using the classic Unix password-hashing algorithm show that given a hashed password, an early 2009 laptop—by no means a state-of-the-art computer—can try more than 150,000

password guesses per second. If the enemy has 1,000 such computers—trivial for any self-respecting botnet owner—all possible passwords of up to eight lowercase letters can be tried in less than half an hour. Even if digits are included in the mix, the guessing time is still only about 5¼ hours. The attacker's problem is often even simpler than that; people don't pick truly random strings like "gisegpoc" or "A*9kV#2jeCKQ"; they prefer words or names, or simple variants of these. A recent study based on the RockYou dataset, a list of passwords posted by some hackers, showed that 19 of the top 20 passwords found fit this model [Weir et al. 2010]; the list of most frequent choices included "`abc123`", "`princess`", and the ever-popular "`password`" (#4 on the list, preceded only by "`123456`", "`12345`", and "`123456789`").

It seems simple enough to solve—just choose strong passwords!—but it's very problematic in practice. Users today don't have just one or two passwords to remember, they have many dozen passwords—at the moment, I personally have well over 100—with importance ranging from online financial accounts down to stores, for-pay news sites, social networking sites, and assorted random places that simply want you to register. I cannot possibly remember that many different passwords, let alone that many strong ones. Besides, each site has different rules for what a password should be like. Some sites insist on punctuation; others ban it. Some want long passwords; others have length limits. Some insist on mixed case; others don't check that, but are case sensitive; still others are case insensitive. The most restrictive set of rules I've seen comes from a US Customs and Border Protection site used by the public.

This is the actual text, copied and pasted from their web site:

- Minimum Length : 8
- Maximum Length : 12
- Maximum Repeated Characters : 2
- Minimum Alphabetic Characters Required : 1
- Minimum Numeric Characters Required : 1
- Starts with a Numeric Character
- No User Name
- No past passwords
- At least one character must be `~!@#$%^&*()-_+!+={}[]\|;:/?.,<>"'`!

Good luck remembering your password for it, especially since you'll use that account about once a year.

Suppose you do forget your password. What then? *Every* real-world system has to have some provision for password recovery or reset. This is very much a trade-off between cost

and security; except for high-value sites (banks, employers, etc.), there are rarely secure solutions. This issue is discussed in more detail in Section 7.5.

Another important issue is the change in threat model. When Morris and Thompson wrote their paper, the primary danger was theft of the password file, followed by an offline guessing attack. Remember that in those days, /etc/passwd was world readable; anyone with unprivileged access to the machine could grab a copy. They certainly realized that the host or its login command could be subverted, in which case it was game over, but that wasn't the threat model their solution was intended to deal with. Unfortunately, today that is one of the most serious problems. The attackers aren't stupid; phishing attacks, compromising servers, and compromising client hosts are the easiest ways to grab passwords. But if the attacker has accomplished any of these, a strong password is no defense at all; a keystroke logger doesn't care about the number of special characters you've chosen. Is password strength obsolete [D. Florêncio, Herley, and Coskun 2007]?

Well, not necessarily. As is frequently the case, the correct answer is "it depends." In this case, it depends on the kinds of attacks you're trying to defend against and on your total system design:

1. What types of guessing attacks are you trying to guard against, online (where the attacker actually tries to log in) or offline, based on a stolen hashed password file?
2. Are the passwords in question employee passwords or user passwords? If the latter, are some users especially likely to be targeted? For example, do you have (or expect to have) celebrity users? There are unscrupulous tabloid papers and web sites that will pay handsomely for dirt about the famous; that in turn can attract more focused attacks.
3. More generally, are you concerned with opportunistic or targeted attacks?
4. What do you assume the enemy can do? Subvert client machines? Subvert your servers? Steal your password file? Launch phishing attacks? Bribe employees? Eavesdrop on communications?

I'll discuss the questions in order, though (of course) the answers interact.

Offline attacks against hashed passwords is the threat model against which strong passwords were suggested as a defense. If that's the risk, password strength is still relevant.

Online guessing is another matter. While password strength may be an issue, the attacker's problem is the effective guessing rate: How many tries per second does the system allow? Is there an upper bound, after which the account is locked for some period of time?

It's relatively straightforward to design a system to respond relatively slowly; indeed, Morris and Thompson designed their scheme to be inherently slow, to frustrate offline guessers as well. An obvious and frequently implemented variant is to slow things down more for each incorrect guess. That gets tricky, though. Is the slowdown per session? The attacker can counter by creating many sessions, perhaps even simultaneously. The

defender's obvious counter is to tie the guess rate to a login name, but that creates interesting synchronization and locking issues, especially on large-scale distributed systems. If there's too much contention for a per-user lock, simply trying incorrect guesses can constitute a denial-of-service attack against the legitimate user.

Taken to the limit, the notion of slowing down the response to an incorrect login attempt is to lock the account for some substantial period of time, on the order of hours. Again, an attacker can abuse this to lock out legitimate users. Some financial sites require human intervention to unlock accounts, but you can afford that only if you make a considerable profit for each user. By contrast, companies that make very little per user, such as social networking sites, can't afford people for that sort of event; they have to rely on automated processes. The question you have to ask yourself here is pretty obvious: What is the expected rate of loss from password-guessing attacks, compared with the cost of reset and the loss of annoyed legitimate users? That is, you're balancing the cost of an account breach against the cost of a forgotten password or locked-out account. Picking strong passwords cuts the cost of the former while increasing the cost of the latter. (If you're planning to turn this question into a spreadsheet, remember to include generous error bars; there are considerable uncertainties in many of the input parameters. "Never let your precision exceed your accuracy.")

Handling online password-guessing attacks is hard enough when the passwords belong to your customers. The question may be completely different when dealing with employee passwords (item 2). The risk of lockout can be quite serious if the locked-out accounts belong to your system administrators or security response team, especially if the denial of service is just one part of a larger, more serious attack [Grampp and Morris 1984]. In other words, detecting guessing attacks is easy; deciding how to respond is hard.

What Makes a Password Strong?

Suppose you do want to pick a strong password. Will compliance with, say, the rules mentioned on page 109 do it? Perhaps surprisingly, the answer is "not necessarily."

The essence of the problem is how to increase the attacker's work, ideally to the point that it is statistically extremely improbable for any guess to be right. Let's look at that list.

The first character must be numeric, so there are only ten choices. Only one digit is required; since switching between letters and numbers is inconvenient, most people will use a consecutive string of digits followed by a consecutive string of letters. A punctuation character is needed, but the easy, natural choice is to put a period at the end. A common choice, then, will be one or more digits, one or more letters, and a period, where the total number of digits and letters will be seven. There are 10 digits and 26 letters, and six choices for how many digits versus letters. The total number of combinations, then, is

$$\sum_{i=1}^{6} 10^i \cdot 26^{7-i}$$

which comes to 5,003,631,360. (It's actually a bit less than that because of the rule about repeated characters.) Using the assumptions from page 108—1,000 computers each doing 150,000 guesses per second—it will take about 30 seconds to try them all. Not everyone following those rules will have a guessable password, but the rules themselves don't guarantee it.

[D. Florêncio and Herley 2010] argue persuasively that password strength policy is determined more by whether or not users have a choice about using the site, rather than by security needs. Thus, monopoly providers—employers, government agencies, and so on—impose strong restrictions because they can. Shopping and advertising sites, which fear losing users—that is, opportunities for profit—impose weaker requirements: "We conclude that the sites with the most restrictive password policies do not have greater security concerns, they are simply better insulated from the consequences of poor usability."

The essence of a good password is given in Figure 7.1: unpredictability. The specific scheme it suggests—four random, common words—is decent; the trouble, though, is that most people won't pick random words nor will they put them in random order. (Besides, four words is probably too short, given today's attackers.) I suspect that it works well only if the system assigns passwords.

Are there other rules? Sure; the trick is simple: create a large-enough search space. But it can't be any large search space; it's got to be one from which people will actually choose (more or less) uniformly. Even picking eight random lowercase letters gives about 209 billion choices, more than 40 times as many as that convoluted set of rules might yield, but as we've seen that's insufficient; 209 billion isn't that large a number these days. The problem is that people's choices from the space prescribed by those rules are decidedly nonuniform. Password choice is a *people* problem.

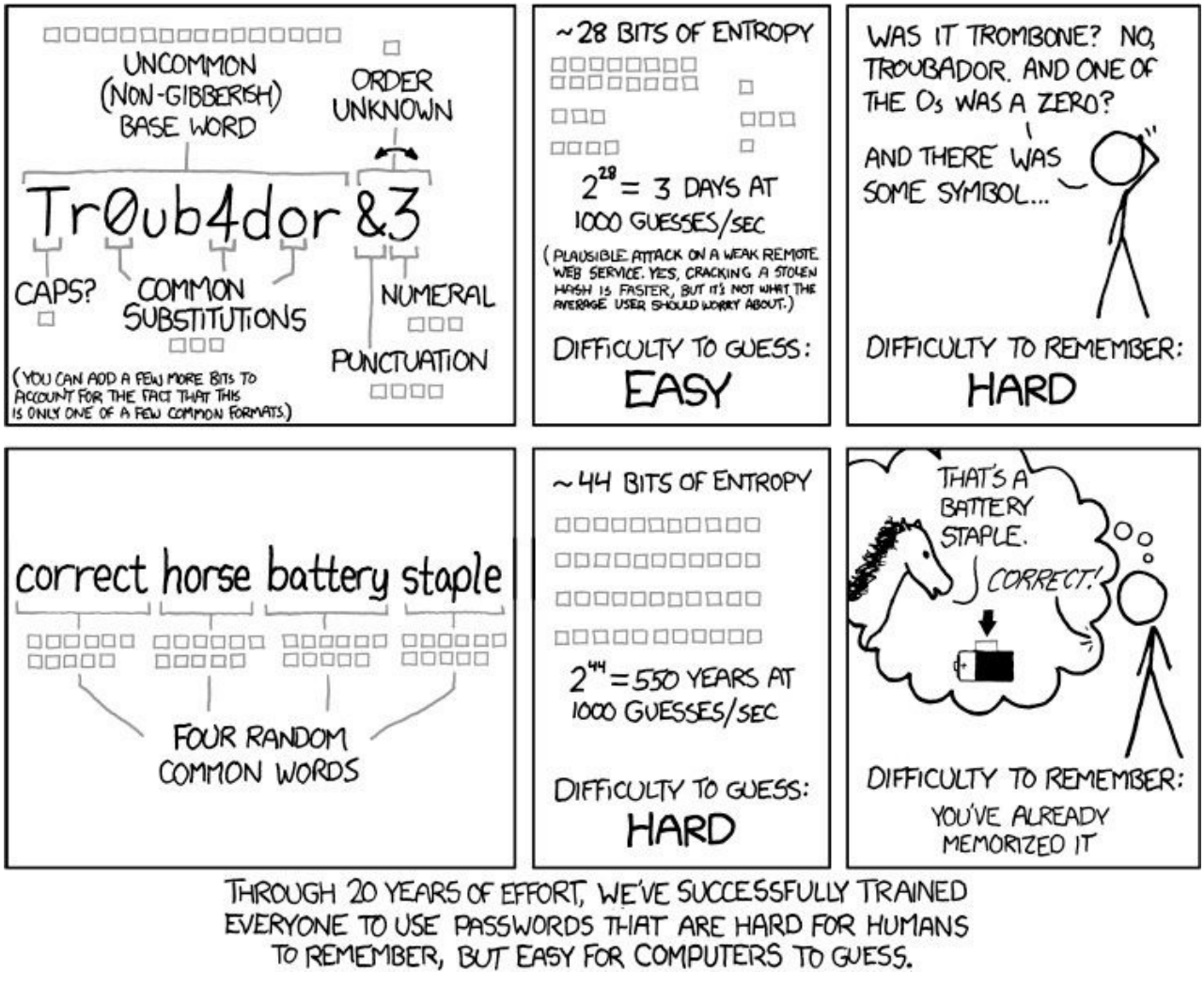

Figure 7.1: Picking a good password.

Employee passwords can be a simpler problem. The easiest solution is to avoid using passwords entirely, especially for external access. You can afford other forms of authentication, and you can invest in training. You can apply different policies if the login attempts are coming from the employee's usual haunts (perhaps a given IP address or LAN) rather than external logins. You can try to ensure that employees use different passwords for different services; in particular, if your employees are also users of your publicly available services, you want them to use different passwords. This is hard, though; people are aggressively uncooperative when dealing with policies that seem arbitrary. If external passwords like "plugh" and "xyzzy" are acceptable but your system insists on special characters in internal passwords, you'll undoubtedly find that the same people who picked "plugh" and "xyzzy" will use "plugh." and "xyzzy!" internally.

The issue of targeted versus opportunistic attacks (item 3) interacts strongly with item 2. If your organization is being targeted, cracking a single employee account can lead to profound damage. This may even be true for external service passwords, if they have privileged access to the service. Note carefully that often, a disgruntled employee or ex-employee is the most likely person to launch such an attack—and such a person might know things that an outsider doesn't know, such as the names of spouses or pets.

For user accounts, there is *always* some risk of targeted attacks. Some users will have to contend with a disgruntled spouse, a suspicious partner, a mischievous friend, or the like.

The last issue (item 4) is the easiest to understand: if the attacker can eavesdrop on communications or can subvert machines at either end, password strength is irrelevant. If the attacker can steal the password file, the issue is whether you store passwords in the clear or hashed. (Stealing a password file may not involve subverting a machine. How good is the physical security of your off-site backup media? You do store copies off-site, right? No? You have another serious problem.) If you store them in the clear, the attacker has won. (Do you store them in the clear? See Section 7.3 and Section 7.5.) A skilled attacker will almost certainly be able to compromise some client machines, at the very least, and will quite possibly be able to compromise the server. An Andromedan will be able to compromise the server and/or steal the password file.

Several conclusions can be drawn from all this. First, employee passwords, if used, should be very strong. However, you're better off using better authentication for employees. Second, while rate-limiting guesses is a good idea, you can't carry that too far except for high-value systems. Third, given the rate of compromise of random client systems, the stress on strong passwords is misguided for many, many systems; the cost of password loss (including the risks of secondary authentication; see below) is higher.

A closely related issue is enforced password changes, especially at frequent intervals. Conventional wisdom says that this is a good idea; often, though, it is counterproductive. Again, a thorough analysis of the threat model is necessary.

The original rationale for frequent password changes came from the US Department of Defense in 1985 [DoD 1985b]. They gave an equation to calculate the proper frequency. Unfortunately—and even apart from the gross uncertainties in some of the input values—the analysis ignores modern threats like keystroke loggers. (It was rather hard to plant a keystroke logger on a 1985-vintage terminal. Many were electromechanical, hardcopy devices that didn't even have CPUs; the few that did were mostly programmed in ROM and were largely immune to attack. Only a very few were "smart.") More seriously, it ignores user behavior.

Gene Spafford did a thorough analysis of the threats some years ago [Spafford 2006].

> … [P]eriodic password changing really only reduces the threats posed by guessing, and by weak cracking attempts. If any of the other attack methods succeed, the password needs to be changed immediately to be protected—a periodic change is likely to be too late to effectively protect the target system. Furthermore, the other attacks are not really blunted by periodic password changes. Guessing can be countered by enforcing good password selection, but this then increases the likelihood of loss by users forgetting the passwords. The only remaining threat is that periodic changes can negate cracking attempts, on average. However, that assumes that the passwords choices [*sic*] are appropriately random, the algorithms used to obfuscate them (e.g., encryption) are appropriately strong, and that the attackers do not have adequate computing/algorithmic resources to break the passwords during the period of use. This is not a sound

> assumption given the availability of large-scale bot nets [*sic*], vector computers, grid computing, and so on—at least over any reasonable period of time.

User response to password change requests makes matters even worse. Grampp and Morris observed many years ago [Grampp and Morris 1984] that people tended to use patterns when rotating passwords: a suffix of "03" for March, "04" for April, and so forth. A large-scale study by Zhang, Monrose, and Reiter [Zhang, Monrose, and Reiter 2010] confirmed this: they developed an algorithm that could guess about 41% of new passwords from seeing the old ones. Their scheme often succeeded even with online guesses; 17% of passwords could be found in five or fewer tries using just one of their algorithms. Again, their conclusion is unambiguous: "We believe our study calls into question the continued use of expiration and, in the longer term, provides one more piece of evidence to facilitate a move away from passwords altogether."

7.3 Storing Passwords: Users

Passwords present a conundrum. They should be (at the least) reasonably strong. People have far too many passwords; it simply isn't possible to remember that many random-looking strings—but the conventional wisdom is that writing down passwords is dangerous. The alternative, used by just about everyone, is to reuse the same password or one of a small set of passwords, for multiple purposes; this means that a single compromise can expose all of your resources. What's to be done, if you have to use passwords but you can't memorize them and you can't write them down and you can't reuse them?

As always, let's go back to our threat model. It turns out that writing down passwords (or some approximation thereof) isn't always a bad idea. If you're worried about joy hackers, in fact, it's probably a good idea; they're not likely to abandon their pizza box lair to steal your wallet. Sure, MI-31 might do something like that—indeed, the FBI has done similar things when pursuing foreign spies [Williams 2010]—but unless you're being targeted for attack by the Andromedans, there are easier ways for a skilled attacker to collect passwords, such as keystroke loggers. The only caveat is that passwords should not be stored "near" the resource being protected since that might expose them to the same attack.

There's a simple analogy here. People are regularly warned not to write down the PINs for their ATM cards. This is correct, as far as it goes, since you don't want someone who has stolen your wallet or purse to be able to loot your bank account. However, an *obfuscated* PIN—say, the last four digits of what appears to be a phone number in your address book—is probably OK, as long as it's not obvious what you've done. The Andromedans might trace every contact number you're carrying; the average pickpocket won't, and the cleverer thieves will resort to things like mag stripe skimmers and hidden cameras [Kormanik 2011].

In a computer context, “near” is a trickier concept. A strong sense of “not near” might be “on a different device”; thus, you could store your computer’s passwords on your phone. That may be an ideal, but good luck seeing the tiny special characters (your passwords are strong ones, right?) and copying them to your browser. Besides, if you sync your phone to your computer, the distance is less than you thought, and if you have a smart phone you’ll probably want the passwords usable on it, too, so you don’t really have as much separation as you thought you did.

A more usable notion of distance is “not in the same application.” That is, store your web passwords in some program other than your browser; if your browser is compromised (and for most users, browsers are their most vulnerable applications), the rest of your password stash is safe. You still have the ability to copy and paste passwords from the dedicated application. That won’t protect you if your computer’s OS is compromised, but neither will anything else. Of course, there are trade-offs here, too—if you copy and paste a password, it’s going to be on the clipboard, where other malware (or simple user carelessness) can paste it to the wrong place. A browser-based password manager has another advantage: it can protect you against phishing attacks. If you never type your web site passwords, but instead rely on the manager, you’re safe: it won’t send a password to your bank unless *it* believes you’re talking to the bank, and not to some site that merely looks like it. (A really good design would look at the certificate name, rather than just the URL, but that helps only if the login prompt is itself on an HTTPS-protected page. Too many are not.)

There are two more things to take into account: the confidentiality and availability of your password stash. Availability seems obvious; it’s not, because different people have different usage patterns.

The Security of Password Managers

There are a number of password managers available. How safe are they? I’m going to use a few real-world examples, not as reviews or recommendations of products—those would be silly to put in a book, in a field as fast moving as this—but to illustrate security design trade-offs. I’m ignoring usability, except for how it affects security; I’m also paying no attention to price or platforms supported. All of these are, of course, very important considerations when actually buying something.

Encryption Most password managers encrypt the passwords, though the one built in to Firefox makes the existence of a master password—the encryption key—optional. Some managers encrypt the URLs, too; others do not, which poses privacy risks for people who visit sketchy web sites.

Synchronization Synchronization between gadgets is crucial for people who use more than one device, especially if the stored passwords are strong, that is, random. LastPass uses its own cloud server; if it is penetrated, an attacker could

mount a guessing attack on the encrypted password store. 1Password can use Dropbox or iCloud for synchronization, including on their mobile apps (and the same caveats about the potential vulnerability of a cloud storage service apply). It can also sync over a LAN; is the protocol secure?

Given that password repositories are, after all, just files or collections of files, most password managers will work with more or less any synchronization system, up to and including flash drives and sneakernet.

Some products (e.g., LastPass) provide web access to your password collection, specifically to permit access from public machines. While undoubtedly convenient, encouraging use of sensitive passwords—including your master encryption password—from potentially insecure machines is dangerous.

Web authentication Many password managers (1Password, LastPass, Robo-Form, and more) will automatically fill in login forms in a browser window. Although a tremendous convenience, from a security perspective it's both good and bad: bad, because it fails the "nearness" test (page 115); good, because it helps prevent phishing attacks. Pass Safe instead makes it very easy to copy a password to the clipboard for easy pasting into a web form; significantly, it clears the clipboard when you close the application, thus preventing you from accidentally pasting the password elsewhere.

External hardware IronKey and CHIPDRIVE use external USB devices. That's great, since your passwords can't be stolen when the device isn't inserted, but human nature suggests that most people will leave their devices plugged in most of the time.

Finally, password managers are not easy to get right [Z. Li et al. 2014]. See Section 11.7 for some discussion on how to evaluate software packages.

If you use only one computer—more precisely, if you use a disjoint set of computers for different sets of accounts—this isn't a big concern; you simply store a (suitably protected) password stash on that computer. If you use two or more computers, though, you need something available to each of them. There are two basic answers: a portable device, such as a USB flash drive, or some form of cloud storage. Portable devices seem more secure, but if you encrypt the files that's not really an issue. They're also more subject to loss or simply being left in the wrong place. (Have you ever wanted to `grep` your house for your keys? You know—the keys on the key ring that have your flash drive attached? The ones your teenager borrowed, along with your car?) For that matter, it's too easy to leave a flash drive plugged in for too long.

There's another obvious form of portable storage: a piece of paper. Apart from failing the availability test (What's your backup for a piece of paper? How well does that piece of paper stand up to repeated folding?) it fails the confidentiality test, in that you probably have to write down the site name next to the username and password. This means that

anyone who steals that paper will have (apart from the rest of your wallet's contents) your credentials for, say, the Bank of Zork.[1] You may be able to obscure that one by listing the site as "J. Pierpont Flathead" or by writing "Hippolyta" for Amazon.com, but that's not a solution that is generically useful.

1. "Bank of Zork," http://www.thezorklibrary.com/history/bank_of_zork.html.

Confidentiality is usually achieved by encrypting the stored passwords; this is generally a good idea, *if* it is done correctly. Some cloud storage providers do offer encryption; their solutions are acceptable if and only if the decryption is done on your machine and they never see a decryption key. (Read the service description *very* carefully; some providers encrypt files only for transmission, or encrypt the stored files with a key that they themselves possess.) If your service provider doesn't do the right thing (or if you don't trust them or can't decrypt their service description), use an encrypted file or disk image. All modern operating systems provide such a facility, either in the base system or as an easy add-on.

A dedicated password storage application may be better, since it will automatically encrypt the storage and will generally do the right thing about caching recently used passwords. There are many from which to choose; make sure that the one you select will let you store the encrypted passwords in the proper spot for use with multiple devices (a cloud drive, a flash disk, etc.). Also pay attention to cross-platform compatibility, including with devices like phones and tablets. Think hard about the frequency and circumstances under which you reenter your master password; there's a trade-off here between ease of use and protection of the data.

The password with which you encrypt the stored passwords is extremely sensitive. If someone gains access to the encrypted store (by stealing your flash drive, hacking the cloud provider, seizing your computer via a search warrant, or what have you), this password is all that stands between the attacker and all of your accounts. It is worth considering whether you want to have different encryption passwords for sites of different sensitivity, but that's easier said than done. What's important is the probability of malware on your machine being able to get at the plaintext of the passwords. You may think that an online banking password deserves very strong protection (and it does), but if the computer in question is part of the accounts payable office of a medium-sized business, you'll need to get at that password more or less continuously anyway, so the encryption matters rather less than one might think. For that scenario, you're better off using a dedicated computer, one used for *no* other functions. (Brian Krebs has long advocated using a "Live" CD for banking, especially for small businesses [Krebs 2009]. It's not that Linux is inherently more secure—I don't think that it is—but it's less targeted, and using a Live CD means that infections won't be persistent; rebooting restores you to a clean state. And if you don't have a CD drive on your computer? Get a USB flash disk with a physical read-only switch. They're a bit hard to find, but they do exist.)

There is one more variant that should be considered. There are password managers that generate per-site passwords from the domain name and a user-specified master password (see, for example, [Halderman, Waters, and Felten 2005; Ross et al. 2005], but there are many others). The good thing is the inherent protection that design provides against both phishing and password reuse. The bad thing is that any site that legitimately receives its own password can launch a guessing attack on the master password; if found, it lets them generate *all* per-site passwords for that user. There's another problem: you can't change just one password. If a site makes you change your password (perhaps because they've been compromised), you can only do this by changing your master password, which in turn means that you have to change *all* of your passwords.

7.4 Password Compromise

Someday, your password or passwords or password file may be compromised. Now what?

If you're an individual, you need to do three things. First, change your password on the affected site. Second, change your password everywhere else you used that same one. That should be easy, since of course you don't reuse your own passwords, but you probably know people who do. Pass this advice on to them… . Finally, and most seriously, assess

what information or resources are associated with that password. A financial account? Check your statements *very* carefully, preferably on paper; there's malware in the wild that can tamper with online statements [Zetter 2009a]. Physical address information? Data that can be used for identity theft? Access to your own computers or files? In case of mass compromise, by skilled attackers who aren't targeting you, you're probably safe—they likely have information on far more people than they can profitably steal from—but if it's part of a targeted attack, you're in trouble.

If you run a large, public-facing site, you have a very different problem. The hardest part is figuring out exactly what information was compromised; that in turn is heavily dependent on the details of your system architecture. What machine was the target of the initial compromise? (Note that this may or may not be the same as the machine whose compromise you detected.) From there, what can an attacker do that cannot easily be done from the outside? Is there data that can be retrieved? If you have an ordinary database system that is exposed to random queries from the trusted—but not *trustworthy*—web server, everything in that database has to be considered exposed. You then have an ethical—and in many cases, legal—obligation to notify all affected users. Depending on the precise circumstances and applicable laws, you may also need to notify assorted credit card companies, government regulators, and so on.

How serious a problem this is depends heavily on your system architecture and database design. If you've stored *personally identifiable information (PII)*, you have considerably more liability under the laws of the European Union, Canada, and other civilized jurisdictions, but perhaps not under US law. Planning ahead can save you a great deal of grief. It is worth reading the report of the Office of the Privacy Commissioner of Canada [Privacy Commissioner 2007] on the TJX hack; briefly, if they had stored the hash of driver's license numbers rather than the numbers themselves, they'd have been able to achieve their business goals while neither exposing the information to compromise nor violating Canadian privacy law. (System design is discussed in more detail in Chapter 11; intrusion response is in Chapter 16.)

Notifying users is a separate problem. Many public sites have very little in the way of contact information for their users; often, it's at most an email address, and those tend to suffer from bitrot. You can and should use those email addresses for breach notification, but you shouldn't be surprised if many of your messages bounce or get caught by spam filters. Depending on the severity of the problem and what information you have, you may need to resort to paper mail; again, you will likely see a nontrivial number of undeliverable letters. There may also be legal constraints: some jurisdictions, such as New York state, require paper mail notification for PII breaches unless the user has consented in advance to email notification.

In many situations, the best approach is to modify what happens after login for those users: they're diverted to a special sequence that tells them what happened and goes through secondary authentication procedures to authorize a password change sequence. With luck—or good system design—the secondary authentication data is on a separate machine, one that wasn't compromised at the same time as the computer used to hold

passwords. How difficult it is to present a different set of screens to the user after login depends heavily on your system design. You may find you need this capability anyway to handle multiple languages, changes in terms of service, and so forth. Again, planning ahead will be a big help.

If the compromise is severe enough, it's possible you'll have to shut down your online presence. This has happened to some very big companies, such as Sony [Schreier 2011]; needless to say, pulling the plug can result in a serious hit to your bottom line.

You have more difficult decisions to make if passwords are compromised on an enterprise system. On one hand, those passwords are considerably more valuable than are those to, say, a newspaper's web site. On the other hand, if you lock people out of their work accounts you lose their productivity. Sometimes, that's the right trade-off. I've seen situations where employees were handed their new passwords when they walked into the building the next morning. That, however, works poorly if many people telecommute or are traveling. The good thing here is that you have much more in the way of secondary authentication and contact information. As before, of course, advance planning —including consultations with your corporate counsel—is a good idea.

7.5 Forgotten Passwords

It's a fact of life: users (including employees) *will* forget their passwords. What do you do about it? Traditionally, there are two approaches: send the password to them or create a new password and send that to them. Sometimes, a supplementary authentication procedure is employed for extra protection.

Users tend to like receiving their original password. After all, it's probably one they use elsewhere, possibly trivially modified to accommodate your strength rules; seeing it will remind them what they did: "Oh, yeah, I replaced the 'o' in 'password' with a '0.'" The fact that it's popular doesn't make it a good idea. The problem isn't so much that you've emailed it to them (that may or may not be safe, as I discuss below) as it is that you can only do this if you store cleartext passwords. If your login program can read that list, so can many other programs, including attackers' malware and your own corrupt employees. That in itself can be bad enough; what makes it really bad is that people reuse passwords. The same magic string that lets people into their favorite porn sites will quite likely be used to let them into their bank's or their employer's systems.

Generating a new password is a much better idea. True, people will probably change it to something they find more memorable, but they can do that anyway. Many sites that send out passwords insist on this, for no good reason. If strong random numbers [Eastlake, Schiller, and Crocker 2005] are used to generate the password, why not let it persist? What is the incremental risk? There seems to be some notion that "the system knows your password, and that's bad." The system will know your password as soon as you type it in; what's bad is if the system stores it. The other issue is where the user stores the notification of the new password; if that is likely to be insecure relative to the value of the resource being protected, a mandatory change may be in order.

To be sure, generating passwords isn't trivial. At a minimum, you need a good supply of random numbers [Eastlake, Schiller, and Crocker 2005]. There's a standard published algorithm for generating pronounceable passwords (such as [NIST 1993]), but that algorithm has problems [Ganesan and Davies 1994]. More seriously, the notion of generating pronounceable passwords is based on a false premise: not just that such a password is easier to remember, but that someone is going to want to remember it. As I've already mentioned, in the real world most people have far too many passwords and they can't even dream of remembering most of them. Typability—on whatever platforms are of interest, including smart phones—is far more important.

As noted, some sites use supplementary authentication mechanisms to validate lost password requests. In principle, this is a good idea; in practice, it's often insecure or unworkable. The challenge is fearsome and difficult: What sorts of questions can you ask a user that that person will *always* remember, but that cannot easily be discovered by an attacker? Your childhood pet? It might be on your Facebook page. Where you went to school? You may be listed on the school's page. That old standby, mother's maiden name? It's a venerable choice, having been used at least as early as 1882 [Bellovin 2011b; F. Miller 1882], but it doesn't work very well. Apart from the complexities of modern society compared with 1882's conventions [Newman 1989] or the fact that many women do not change their names when they marry, marriage records are public documents; an attacker can easily look up the data. You might think that this would happen only in a serious, targeted attack, perhaps by MI-31, but in fact the attack has been automated [Griffith and Jakobsson 2005] since many of these records have been put online.

Secondary authentication is more challenging for public figures, since there is frequently a lot of information available about such people. The Sarah Palin email account incident is a good case in point [Zetter 2008]; all the attacker needed to know was Palin's birthday, zip code, and the answer to a simple security question: where did she meet her spouse? It might be challenging to learn that about a random person, but for someone who is the subject of as much media attention as she was, it's pretty easy.

This illustrates another issue: those very close to a person know the answers to these questions. Divorce cases are particularly thorny, since soon-to-be-ex-spouses are often on their worst behavior. Snooping on email has happened in such circumstances [Springer 2010]; technical measures are likely to be inadequate. It is interesting to speculate what will happen in the future with online banking passwords.

The challenge is to ensure that the password, old or new, reaches the right individual and only the right individual. In an employment situation, the employee's supervisor might be the best person to handle it, since he or she presumably knows the employee. That doesn't work as well with today's distributed companies, where solitary employees can be in any part of the globe. Similarly, in university environments students are sometimes told to show up with their ID card to get their passwords reset; again, this is problematic with the rise of distance learning.

To address this, some have suggested "social authentication"—letting your friends

authenticate you. In one scheme, designed to deal with the loss of a token, another legitimate (but preconfigured) user can use his or her own credentials to obtain a temporary authentication code for a colleague [Brainard et al. 2006]. This is combined with the user's PIN for a single login session. The big risk here is that lots of people can now grant access; if they're careless about security or dishonest, you have a problem. A related notion relies on users' abilities to recognize their friends; unfortunately, that seems to be very susceptible to targeted attacks [H. Kim, Tang, and R. Anderson 2012].

The most common way to hand out reset passwords is by email. This can be perfectly appropriate; some resources are of little enough value that the risks of email being intercepted are acceptably low. For more valuable passwords, some form of out-of-band authentication is a better idea. Banks will generally send paper mail with new credentials; SMS messages are another good way, though the malware artists have started building phone apps that do nasty things when phones are used for authentication [Crossman 2013; Pauli 2014].

If you use email, the message will be in the user's inbox. How well protected is that, both immediately and over the long term for people who don't delete their email? For passwords of modest value, you're probably safe; for something like an employee password being sent to a free mail account, it's rather more dicey. (This is one of the rare instances where people have to use outside servers for work-related matters: more or less by definition, if they've forgotten their employee password they can't get at their internal email.) There are two good choices here: require an immediate password change, or send a URL for a password reset page. A lot depends on your overall system design—how hard is it to have special case handling of successful logins, versus how much access does your (externally facing) web server have to the password database. If you do send a URL, it should have two crucial properties: it should be usable only once, and it should only be usable for a limited amount of time.

Ultimately, the risk is that many secondary authentication systems are weaker than the primary one, especially against certain threats. The more you're the subject of a targeted attack, the more risk you're facing. A random 419 scammer halfway around the world won't steal an envelope from your mailbox; an Andromedan agent might.

7.6 Biometrics

> "We need something that will identify any representative of Civilization, positively and unmistakably, wherever he may be. It must be impossible of duplication, or even of imitation, to which end it must kill any unauthorized entity who attempts imposture."
>
> Dr. Nels Bergenholm in *First Lensman*
> —E. E. "Doc" Smith

There's a saying in the security business: if you think that biometrics are the answer, you're asking the wrong question. That's exaggerated, of course, but so are many of the

claims made for biometrics. More than any other form of authentication, biometrics must be looked at from a systems perspective.

Although many biometrics have been proposed over the years, including speech, typing rhythms, and hand geometry, three are of major importance: fingerprints (100+ years of criminology have given us reasonable assurance of their uniqueness; besides, scanners have become very cheap); iris scans, widely regarded as the most secure [Daugman 2006]; and facial recognition, due to the ubiquity of cameras and the potential in physical security situations for walk-through authentication. Acceptors generally don't store the actual image of any of these; rather, they store a *template*—more or less the equivalent of a hashed password—against which they match a submitted biometric. (As with many aspects of biometric, reality is often more complex. Some schemes, e.g., [Ballard, Kamara, and Reiter 2008] and [Pauli 2015], do store images; in others, the templates are effectively reversible.)

A biometric authentication system consists of a number of components: a human (or parts thereof), a sensor, a transmission mechanism, a biometric template database, and an algorithm are the minimum. An attack can target any of these, which means that they must all be protected.

Let's start with the human. The premise of biometric authentication is that it always Does the Right Thing. It can't be forged, it can't be forgotten, and it will always work. None of these are true. Researchers have successfully spoofed fingerprint readers with gelatin casts made from molds created from latent fingerprints [Matsumoto et al. 2002]. Facial recognition has been spoofed by photographs [Boehret 2011]. Thieves have chopped off people's fingers to fool cars' fingerprint readers [J. Kent 2005] and compromised biometric scanners [Whittaker 2015]. A non-trivial percentage of the population has fingerprints that cannot be read by common scanners [S. T. Kent and Millett 2003, p. 123]. And there's the obvious: someone who injures or loses a body part may not be able to use systems enrolled in before the incident.

Sensors are sometimes designed to compensate for some of the spoofing attacks. Many contain "liveness detectors": pulse detectors, thermal devices, and so on, that try to verify that the body part being monitored really is a live body part attached to a real, living person. Determined adversaries have been able to work around many of these defenses. For example, [Matsumoto et al. 2002] notes that capacitive detectors were intended to resist attacks that had been launched successfully against optical scanners, but their techniques worked against both kinds of sensors.

Suppose your laptop has an iris scanner. Can you use that to log in to some remote web site? Recall that the remote web site doesn't see a finger or a fingerprint; rather, it sees a stream of bits. An enemy who can eavesdrop on the transmission line can easily replay the bit stream, thus spoofing the "absolutely secure" biometric authenticator. At a minimum, the transmission link needs to be encrypted; depending on the operational environment, the sensor and encryptor may need to be inside a tamper-resistant enclosure.

The most misunderstood part of biometric authentication systems is the actual algorithm

used to do the match. Processing, say, a retinal image is not the same as checking a password. The latter will always yield a definite answer of "right" or "wrong." By contrast, the former is a probabilistic process; sometimes, the correct body part will be rejected, while at other times someone else's will be accepted. Worse yet, there is a tradeoff between the *true accept rate (TAR)* and the *false accept rate (FAR)*: the more you tune a system to reject impostors, the more likely it is that it will reject the real user. A 2004 report from NIST [Wilson et al. 2004] makes this clear. If the best available fingerprint scanner was tuned for a 1% FAR when trying to pick out an individual from a large collection, the TAR was 99.9%. But cutting the FAR to .01% cut the TAR to 99.4%. (Facial recognition was much worse; for the same FAR rates, the TAR was 90.3% and 71.5%.)

While the technology has improved since then,[2] the trade-off issue remains. There are several implications for systems design. The first is what the consequences are of a true accept failure (sometimes known as the *insult rate*). That is, if a legitimate user is rejected by the system, what happens? Can that person try again, perhaps repeatedly? As discussed earlier, locking someone out for too many password failures is often a good idea; do we do the same with biometrics? Do we resort to secondary authentication, with all of its costs and risks? There is no one answer to this question; a lot depends on your application and system design.

2. "NIST: Performance of Facial Recognition Software Continues to Improve," http://www.nist.gov/itl/iad/face-060314.cfm.

The FAR raises its own issues. One can be understood by simple mathematics: for any given FAR rate, with enough different biometric templates, the probability of a false match becomes quite acceptable. Suppose that our system is tuned for a FAR of .01%. If there are n entries in our database of acceptable biometrics (i.e., n faces or iris scans from $n/2$ users or full fingerprints from $n/10$ users), the odds of a successful attack are $.9999^n$. At n = 6,932, the odds tip in the attacker's favor. The countermeasure is obvious: require an assertion of identity before the scan, and match the input biometric against a single user's templates, rather than against your entire database. (The fingerprint scanners built in to many laptops generally don't do this, because n isn't high enough to matter. Undoubtedly, though, they're tuned for a relatively high FAR, in order to keep the TAR acceptable.)

The cost of a trial to the attacker also interacts with the FAR. If the cost is high enough—say, what will happen if Pat Terrorist tries to cross a border with Chris Clean-record's biometrically enabled passport—the attackers can't easily launch an impersonation attack. Conversely, if trials are cheap—a self-service visa kiosk?—the attack is feasible if there is a large enough supply of cooperative people with passports and clean records.

The last element of our abstract biometric authentication system is the template database. Given a template, can an attacker easily construct a fake biometric that matches it? If so, the legitimate users have a very serious problem if the database is ever compromised: most people have a very limited supply of fingerprints and irises to use, and even fewer faces. It's *much* easier to change your password than to change your eyeball.

Templates are supposed to be irreversible, much like the hash of a password, but some researchers have managed to attack them successfully [Galbally et al. 2013]. Beyond that, possession of the database allows for low-cost trials to exploit the FAR. The database compromise issue, which is a conceptual one rather than an artifact of today's technological failings, may be the ultimate limit on the growth of biometrics.

There are other difficult issues. What resources, precisely, are accessible via by biometric authentication? On a local system—say, a laptop that is unlocked by a fingerprint swipe—part of the answer is access to a local database of cryptographic keys, such as Apple's keychain. When passwords are used to authenticate access to the laptop, that password is converted to a key that is in turn used to decrypt the database. Converting a password to a key is straightforward; there are even standards saying how to do it properly, such as [Kaliski 2000]. Not so with biometrics; by their nature, they're inexact. Here's an experiment to try. Mount your camera on a tripod and take two indoor pictures of the exact same scene. Strip out the metadata (such as timestamps) and see whether the resulting files are identical. If they differ in even a single bit, they can't be used as keys.

The solution is a technology known as *fuzzy extractors* [Dodis, Reyzin, and A. Smith 2007]. Without going into the details, a fuzzy extractor generates a uniformly random string from noisy input; this string is suitable for use as a cryptographic key. Unfortunately, in practice biometrics tend to be too noisy to work well in such situations. It might be possible, but there are very few, if any, such products available today.

There's one more important reason to avoid biometrics: privacy. A biometric identifier is more or less the ultimate in PII; using one unnecessarily not only brings you into the ambit of various privacy laws, it exposes you to serious public relations problems should your database be stolen. Furthermore, because biometrics can't be changed, in some cases the consequences are serious and long-lasting [Volz 2015]:

> Part of the worry, cybersecurity experts say, is that fingerprints are part of an exploding field of biometric data, which the government is increasingly getting in the business of collecting and storing. Fingerprints today are used to run background checks, verify identities at borders, and unlock smart phones, but the technology is expected to boom in the coming decades in both the public and private sectors.
>
> "There's a big concern [with the OPM hack] not because of how much we're using fingerprints currently, but how we're going to expand using the technology in the next 5-10 years," said Robert Lee, cofounder of Dragos Security, which develops cybersecurity software.
>
> …
>
> One nightmare scenario envisioned by Ramesh Kesanupalli, an expert in biometrics, is that agents traveling across borders under aliases could be spotted for their true identities when their prints are scanned. Kesanupalli also warned that the fingerprints could end up somewhere on the black market, making biometrics

a novel good to be trafficked on the Internet that could be useful to a buyer for decades.

Where does this leave us? The risk of compromise of large template files seems high, given the rate of compromise of conventional password files. That, combined with the fact that a remote server sees only a bit stream, suggests that biometrics cannot and should not be used for general Internet authentication. On the other hand, use of a biometric when the submitter is under observation—a border checkpoint or a bank teller's station are good examples—is rather safer, since the ability of an attacker to spoof the input is considerably reduced. Even here, the insult rate issue has to be considered in the total system design: a biometric match failure does not always indicate enemy action, and hauling people off to the hoosegow because they've had cataract surgery (which will sometimes but not always affect the scan [Roizenblatt et al. 2004]) or have aged since the template was captured [Fenker and Bowyer 2011] does not seem like a good idea. (The privacy concerns apply to governments, too, even in the United States.)

Biometrics are also a reasonable authentication mechanism for local resources, such as an encrypted flash drive, an authentication token, or a phone. In such cases, the use of tamper-resistant enclosures is strongly suggested to forestall attempts at bypassing the authentication. There is still some risk from targeted attacks—MI-31 probably has your fingerprints and iris scans from the last time you crossed the border into Andromeda—so good sensors, liveness detectors, etc., are still a good idea. The insult rate problem can be dealt with by replacing the device (you always need to plan for lost authentication tokens; see below) and/or by storing a backup access key in a physically secure location. (Newer Apple iOS devices can be unlocked by a fingerprint. However, a PIN is required for the first unlock after each reboot, since the key used to encrypt sensitive portions of the devices' memory is derived in part from the PIN [Apple 2015]. The fingerprint template itself is stored in a secure part of the CPU.)

7.7 One-Time Passwords

The phrase *one-time password (OTP)* is often misused. It does not refer to a single technology; rather, it refers to any scheme that appears to accept a simple password that is never reused. That is, instead of being some static concept—a conventional password, or even a fingerprint or other biometric—what is sent is a dynamic value, one that depends implicitly or explicitly on time or past history.

Often, the notion of an OTP is conflated with a particular technology, such as RSA's popular SecurID token (Figure 7.2). Not so; there are many other types of OTP.

Figure 7.2: An RSA SecurID authentication token.

An OTP scheme has two crucial properties. First, its output must be effectively non-repeating. By "effectively" I mean that the odds of a repetition should be no greater than would occur by chance. Thus, if a single-use password P_i is drawn from the closed interval $[0,n-1]$, the probability that $P_i \in \{P_0,P_1,\ldots,P_{n-1}\}$ should be approximately $1/n$. Second, seeing some set of values should not allow an adversary to predict future values. That is, no matter how many P_x the adversary has seen, the odds of a successful guess at P_i should remain no better than $1/n$. Note that this property rules out the use of a secret permutation of $[0,n-1]$ as the sequence of OTPs, though we could do it if we relax our condition to say that the probability of a successful guess must merely be less than some suitably small ε and $i \ll n$.

In practice, OTP schemes generally depend on a strong cryptographic function and a secret. Thus, the SecurID tokens display $F(K,T)$ where T is the time. (The scheme is actually rather more complicated than that; for my purposes, this simplification will suffice.) An adversary who could invert F—today, AES is used—could recover K and thus generate responses. There are two obstacles. First, if F is strong (and AES is believed to be) the attacker won't be able to invert it, especially with the limited number of samples available compared with encrypted traffic. Second, for purely pragmatic reasons the output of F is often truncated. Thus, the display of a SecurID is generally capped at six digits, or about 20 bits, leaving 108 bits of the output of the AES encryption unknown. Even if you could find K for one particular value of those bits, you don't know that that value is actually the one that the actual token's calculation produced; there are, after all, $2^{108}-1$ other possibilities. Did you find the right K for the next T?

The same analysis holds true for typical challenge/response OTPs. In those, the server sends the client some random value N; the client (who is assumed to have a secure computing device) responds with $F(K,N)$. As before, F is hard to invert and only a truncated form of its output is transmitted.

There are OTP schemes that don't require the client to have anything more sophisticated than a piece of paper. Lamport's scheme [Lamport 1981] uses a noninvertible function F and a secret seed value x; he defines password i to be $F^{k-i}(x)$ where k is the maximum number of passwords that can be derived from x. Thus, if $k = 1{,}000$, the first user

password is $P_0 = F^{1,000}(x)$, the next are $P_2 = F^{999}(x)$, $P_3 = F^{998}(x)$, … . A user going on a trip could simply print out some number of passwords on a sheet of paper, and cross out each one as it is used. As before, the security of the scheme depends on the secrecy of some value (in this case, x rather than a key K) and the non-invertibility of F. There is one salient difference in the usual implementations [Haller 1995]: the output of F cannot be truncated, since a server expecting password i will have stored password $i - 1$—the last password successfully sent—and will verify it by calculating $F(P_i)$ to see whether it matches what it just received. (Clearly, one could define F to be the truncation of some F' that has a longer output; while this saves typing, it is the output of F—the truncated version—that is iterated. Consequently, there is no ambiguity resulting from the attacker's lack of knowledge of many bits of the output.)

One can carry this further: why bother with a public algorithm to generate P_i to print on a piece of paper; why not just give the user a printout of the next several of random passwords generated and stored by the server? Lamport rejected that idea because of the storage costs, but storage is much cheaper today. More seriously, if the server is storing many passwords, an attacker can steal that list, which isn't possible if just the last one used is stored. A second reason—that if the user has local computing capacity, he or she could simply type a password for x—is arguably a disadvantage today, since there may be a keystroke logger collecting x. In addition, if x can be a password, it is possible to run a password guesser on the F^i.

In fact, some banks do send such papers to their users, often with the password sequence protected by a scratch-off overlay. There's a variant that is used: a two-dimensional grid, where the bank will ask for, say, $\langle x_1,y_1 \rangle$ and $\langle x_2,y_2 \rangle$. Fundamentally, this is just a challenge/response scheme, where K is the user's grid and F is "look up two values in the table."

OTP schemes solve a lot of problems, but they all have certain limitations. For one thing, most rely on the user having something: the token, the key, perhaps the seed for Lamport's algorithm if a password isn't used, or some such. From a systems perspective, they're at least as challenging to manage as passwords. If a SecurID or challenge/response token is lost, it has to be replaced; this may involve an overnight express shipment for telecommuters or road warriors. The alternative—some form of manual single-use password set by a help desk—is effectively a reliance on secondary authentication techniques; as we have seen, those are often far weaker than ordinary passwords, let alone one-time passwords. This is the worst of all possible worlds: higher cost *and* less security.

Because physical objects are subject to theft as well as accidental loss, most sites using OTPs supplement them with a PIN. Of course, PINs can be forgotten just as easily as can passwords, thus taking us back to the world and costs of secondary authentication.

Apart from question of secondary authentication, how secure are OTPs? The answer is rather more mixed than appears at first glance.

There are two major benefits from the use of OTP schemes. First and foremost, the problem of password guessing is eliminated. (I'm assuming, of course, that the seed for Lamport's scheme is not a typed password.) Against many forms of targeted attack, this is a very significant advantage indeed. However, as we have seen, this is often a small part of the password problem. A second benefit is that a token can't easily be shared. Or rather, it can be lent, but when it is the authorized user no longer has possession of the token and hence can no longer log in. (Using "soft tokens" on phones is one way to limit this risk. While the phone isn't as tamper resistant as a dedicated hardware token, few people want to be without their toys.)

I'm sure many of you are now thinking, "But what about the one-time use property??!!" This is certainly a strength, but it's not nearly as significant as it once was. For one thing, the risk of over-the-air eavesdropping today is much less than it was even ten years ago. Many forms of cryptography are in widespread use, including VPNs. Anyone who logs in to a remote system without using encryption is vulnerable to many other forms of attack. Even with encryption, though, an attacker who has compromised one end of the connection or the other can steal a credential. True, a password stolen that way can be reused. If the attacker has compromised the server, it doesn't matter much; he or she is already in a position to do anything to any account. And if the client is compromised? Against modestly clever malware, it doesn't matter much.

Suppose you're sitting at your computer, typing one character at a time from your OTP device or paper. The malware is watching and waiting for you to type the next-to-last digit. It then sets up ten new connections to the server, replays the digits you've already typed, and then tries a separate guess at the last digit on each of the connections. One will succeed. For that matter, if your machine is compromised you don't even know that you've set up a single connection; you may be talking to the malware all along. Of course, when the malware collects the entire password, it simply says "connection dropped" or "password incorrect" or some such. Obviously you mistyped the password, right?

There are similar attacks at the server end. Suppose you fall for a phishing attack and enter a single-use password into a fake web site. The attacker can collect it and log in in your stead.

Clearly, the same stunts could be pulled with conventional passwords. OTPs do have an advantage—the stolen session or credentials can be used only once (and for time-based tokens, only for a limited time). But reduced harm is not the same as no harm.

There have also been phishing attacks on banks' paper-based OTP systems. In fact, I've heard of one scheme that told victims that they needed to revalidate their online access by entering the next three numbers in the sequence.

There's more to consider, starting with the server-side infrastructure. What happens if your servers are compromised? With several of the schemes I've just described—time-based authentication, challenge/response, and probably the two paper-based schemes—the

server knows the clients' secrets. Someone who hacked that database could then impersonate those clients indefinitely. Given the number of high-profile password databases that have been hacked, there is no reason to think that the OTP equivalents are immune.

In fact, the entire back-end infrastructure has to be seen as part of the security perimeter. An attacker who can modify account data, perhaps by using the administrative interface to record that a new token with a new (and known) *K* has been assigned to a given user, can take over that user's account. Don't underestimate this problem—SecurID succeeded so well in the marketplace precisely because they didn't sell just tokens or just cryptographic routines; they sold an entire *system*, of authentication software, servers, administrative code to add and delete users, databases, and so on. That was quite good, both as marketing strategy and because such code is definitely needed, but all of it and the machines it runs on are security sensitive. Can you protect them well enough? Note that any authentication system needs at least some of those components; engineering a secure system requires understanding which components you have and figuring out how to protect them.

Finally, there is the question of where the authentication secrets come from. Who picks *K* or *x*? Is that a secure process? If they're supplied by a vendor, does the vendor protect them properly? This is not an idle question; Lockheed was penetrated using data on the SecurID system stolen from RSA [Drew 2011]. Exactly what was stolen has never been disclosed, but one guess is information on the *K*s used by Lockheed. The attacker had to have had more data than that—users generally connect via a login name, but the tokens' keys are indexed by serial number—so either a penetration attempt needed the mapping between logins and serial numbers, or the attackers had to try an entire set of serial numbers (and hence keys) for some selected set of users.

7.8 Cryptographic Authentication

Cryptographic authentication is generally considered the strongest type. It can be, though sometimes implementation flaws vitiate the protection. As always, it is vital to look at the entire system, rather than just the 0s and 1s of the algorithm.

While there is no precise definition of *cryptographic authentication*, intuitively I mean a protocol where both parties are using cryptography and cryptographic secrets (i.e., keys) to do the authentication. In particular, it is a scheme in which the user's authentication isn't forwardable to another site, thus preventing *monkey-in-the-middle (MitM)* attacks. Furthermore, the process includes negotiating keying material for session encryption. Often, though not always, such mechanisms provide *bilateral authentication*. If users are to employ keys and cryptography, though, it implies that (a) they have sufficient, secure local computing capacity to do the necessary calculations, and (b) they have secure, long-term storage for keys. Sufficient local computing capacity is easy; *secure* capacity and *secure* key storage are another matter entirely.

Why, though, is cryptographic authentication so strong? Is that actually true? Or is it all perception? Some of the strength comes from the properties outlined above, but some of it may indeed be perception.

The biggest theoretical advantage of cryptographic authentication is that it is, in principle, based on random keys rather than on a weak password. Saying that, though, begs the question of key storage and protection. As I noted earlier, insecurity can't be destroyed, only moved around. If the cryptographic key is derived from a password, or a password is used to protect it, the inherent strength of the scheme isn't necessarily stronger than passwords. Consider, for example, a system that uses cryptographic authentication but uses PKCS #5 [Kaliski 2000] on the user's side to derive a cryptographic key from a password. The server doesn't need the password itself; it does need its copy of the key. But this value is in effect a hashed password, against which a guessing attack can still be launched.

Am I saying, then, that such a system is no stronger than ordinary passwords? Not quite; there's still a difference, but it's a bit more subtle: the user's password is not sent to the server, and hence can't be captured that way. This in turn gives rise to the anti-MitM property: the server does not receive anything that it can forward to another site. Phishing attacks are rendered harmless; no stealable, let alone reusable, credentials are transmitted.

The optional bilateral authentication property is even more useful, since a clever attacker may still try to trick a user into revealing sensitive information even if there's no authentic server on the other end of the connection. If the two parties can use cryptographic mechanisms to negotiate a session key, this key can be used for challenge/response authentication in both directions. We can turn that around. If we have a shared key, the bilateral authentication property can guarantee that there is no MitM; an attacker won't have the necessary long-term key to perform its part of the authentication dance. Do not, however, rely on users to notice failure of bidirectional authentication; make sure your systems will not operate in such cases.

The keying material can, of course, be used to encrypt the entire session. This provides the usual protections; that is particularly important in this context because the encryption protects against someone hijacking an authenticated session after it's fully set up. Even without that, though, cryptographic authentication is indeed very strong—*if* you can store and protect the private key.

There are three principal mechanisms that can be used for private key storage: password derivation, external devices, and locally encrypted storage. None is perfect; all have their disadvantages.

I've already described the issues with deriving a key from a password; per the analysis, though, it's still a better choice than sending the password directly. The most common way this is used, though, is with single-sign-on systems (Section 7.10) such as Kerberos. A word of warning: a keystroke logger is just as effective against this use of a password as against the more common use.

External device storage of keys is the most secure option. However, as discussed in Section 7.9, there are still significant concerns. They're mostly cost and convenience issues, but there is one security concern: if there's malware on your machine, it can use the

key, or sit in the middle of an authentication session you intend to initiate and use your key to initiate its own session instead.

The third option is probably the most common. Passwords can be used to encrypt .ssh private keys or private keys associated with certificates, especially with TLS. The problems with the other two schemes are present (I expect that keystroke loggers will be upgraded to steal the encrypted key file as well as the password used to protect it); the bigger issue, in most situations, is availability of the file containing the key. If your users frequently use more than one machine (and for many people, that's the norm, not the exception), the key file—often the same key file—has to be present on all of the machines. This can be addressed by using cloud storage or USB flash drives to hold the key; many see this as ideal, since it makes it possible to use such authentication on public kiosk machines or in Internet cafés. That's actually a disadvantage; such machines are notorious lairs of all sorts of malware. Using cryptographic authentication from an infected machine is just as insecure as any other way of using such a machine.

One final pessimistic note: a cryptographic authentication scheme is, ultimately, a cryptographic protocol; all of the warnings, caveats, and cautions that apply to cryptographic protocols in general apply here, too. *Don't* invent your own.

7.9 Tokens and Mobile Phones

Tokens—something you have—are a popular authentication mechanism for security-sensitive organizations. Often, this is a good idea: Using tokens avoids all of the weaknesses of passwords: their secrets can't be guessed, the authentication sequence is (almost always) not repeated, one can't share a token without losing use of it oneself, and so on. All that said, the risks and limits of tokens must be considered as well. As always, we must look at the problem from a systems perspective.

One obvious issue is the cost: tokens cost money and at first blush are more expensive than passwords. While there is certainly an initial cost, the total expense for authentication is rather more complex to assess. Passwords carry hidden costs, both in the form of complex secondary authentication mechanisms and in the much greater costs of recovering from password-related compromises. There is also the question of whether all relevant applications can adapt to tokens. The biggest incompatibility is the semantic mismatch between applications that instantiate many sessions over time and the single-use property of most token-based systems. The obvious example is anything web based. HTTP is stateless; in general, any new request or even any element on a single page can entail a separate TCP connection and hence separate authentication. That is clearly unacceptable. The usual answer, of course, is to use token-based authentication to create some other, longer-lived authentication scheme, such as a web cookie. This, though, creates a different secret, one that is not stored in the token and hence one that may be more vulnerable to abuse. Cross-site scripting attacks to steal cookies are the classic example.[3]

3. "Cross-site Scripting (XSS)," https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

It is important to realize that the flaw is not in token-based authentication per se; cross-site scripting attacks are based on the properties of web browsers, regardless of how the

initial authentication was done. From a systems perspective, though, the site is not getting the security it sought by deploying tokens.

A similar weakness can occur if some malware waits for the token-supplied authentication string to appear. This string is captured by the malware and used for its own login, rather than the user's; the desired site gets some random garbage, which of course produces a failed authentication. No one will particularly notice this, though; it will likely be interpreted as a random failure or a typographical error. (Thought experiment: how often have you had a web interaction fail, only to succeed when you resubmitted the same information? Did your security antennae twitch? Should they have?) Again, the problem is not in the concept of tokens; however, the benefits of tokens are not realized.

Apart from security risks, the question of application compatibility is sometimes a deal breaker. For common tokens and popular applications on mainstream platforms, there may not be a problem; the vendor may have provided a suitable interface. Alternately, the platform may use generic authentication interfaces that any application can use. As noted in Section 7.7, the software support for a token should comprise far more than just the token itself or a simple "Is this authentication valid?" routine.

Earlier, I spoke of the problem of forgotten passwords. Tokens, of course, can be forgotten, lost, or stolen. If authentication must be done by a token and the token is left at home, the user has no way to authenticate, and hence no way to work. What do you do? Lose a day's productivity? Fall back to secondary authentication? Employ some temporary authentication scheme? The answer, of course, depends on how you balance security and cost: a high-security site will pay the price and not use an insecure secondary authentication scheme. At the least, it will require in-person vouching by someone who knows the careless employee. Other sites, of course, will make other choices. An oft-suggested solution to some issues surrounding tokens is the so-called *soft token*: software that emulates a dedicated token running on some computer. A popular choice is a smart phone, since many people are more likely to forget their clothing than their phones (Figure 7.3). There is a serious danger lurking here, though: ordinary computing devices are not at all resistant to reverse engineering; it's relatively easy to build malware that extracts secrets from other applications.

Figure 7.3: Is she reading text messages before getting dressed? Taking a selfie? Not really; the sculpture is Erastus Dow Palmer's *Indian Girl, or the Dawn of Christianity*, and dates to 1856.

We don't have to posit the creation of phone-based malware that targets authentication systems; examples already exist. One popular one [M. J. Schwartz 2011] targets a different way in which phones are used as tokens: the server sends a random string to the user via an SMS ("text") message. Theoretically, only the user has that phone, and hence is the only one who will see the challenge; consequently, this seems to be a simple way to implement challenge/response authentication via a token. It's simple—but it may not be secure.

It may also pose a privacy issue. Most people have only one mobile phone number, which they may keep for life. This in turn makes it a persistent, unique identifier, bound to one individual. This is exactly the sort of thing that marketers love for matching profiles. If you sign up for a bunch of cloud services that use text messages as part of the login process—at this point, Google, Paypal, Dropbox, Apple's iCloud, and more all support this—you may be opening the privacy door very wide. It may be a good trade-off, since the privacy harm from having these accounts hacked is also very great, but it is a trade-off.

Phones, especially smart phones, are popular targets for thieves. Fortunately, the average street thief is unlikely to extract secrets or otherwise exploit the authentication

properties of a phone; the same, of course, cannot be said for MI-31. If a phone used for authentication to sensitive data is stolen, is it clear what the motive is [Allen 2012]?

For reasons like this, tokens should *always* be used together with some other form of authentication, such as a password, PIN, or biometric. Great care has to be taken in the design of any such two-factor authentication scheme, especially if the second factor is sent directly to the token rather than to the server: MI-31 can quite likely bypass authentication done by the token itself, unless the token was specifically (and competently) designed to resist just this threat. Soft tokens are especially vulnerable to this threat; absent strong evidence for the security of the underlying platform, their use should be eschewed in high-risk scenarios.

One more point must be made about tokens: you should always have some plan for invalidating lost tokens and switching authentication to new ones. Tokens *will* need replacement, whether because of loss, theft, fire, accident, or simple hardware failure. A corollary is that if any data is accessible solely through a secret resident on the device, you'd better have a backup copy of that secret. Encrypting your files with a key stored only on a user's smart card is a recipe for disaster; keep another copy safe somewhere else.

7.10 Single-Sign-On and Federated Authentication

Single-sign-on (SSO) schemes use a two-phase authentication scheme: the user somehow authenticates to a central server; the fact of this authentication is then communicated to any other systems the user wishes to communicate with. A federated authentication scheme is essentially the same thing, save that the central server is an outside party that can vouch for identities to many other outside sites. There are three primary issues for an internal SSO system: the initial authentication, how the additional authentications are done, and the user interface to the latter. There are additional issues when external parties are involved; those will be discussed later.

The first question is relatively easily disposed of: in theory, any standard authentication system can be used, and the usual strengths and weaknesses apply. There is one very important caveat, though: an SSO is a very tempting target for attackers, especially the more skilled ones, and hence requires more than the usual protection. By extension, since a login to such a server gives a lot of access, stronger forms of user authentication are probably a good idea. At a minimum, the authentication used for an SSO server should be at least as strong as you would want for any given system that will trust that authentication.

There is another issue: the form of SSO authentication used interacts with the later authentications. In particular, these later authentications sometimes rely on some sensitive material passed to the user's computer by the SSO server. This transmission needs to be protected, which implies some sort of cryptography; that in turn favors cryptographic schemes that permit easy setup of an encrypted channel. The alternative, used for web-based SSO schemes, relies on the use of a TLS-protected session to the SSO server.

A web-based authentication system will probably rely on *cookies* [Barth 2011] to maintain the logged-in state. However, cookies can be returned only to the site that set them, so some other mechanism must be used to pass the login information to other web

sites. Generally, this is done by out-of-band communication between such web sites and the SSO site.

It helps to consider the one possible (and oversimplified) sequence of operations. Assume that some user Chris first contacts www.ReallyAwesomeSSO.com and logs in. That site sends Chris's browser a cookie. (If Chris is really lucky, that cookie will have appropriate cryptographic protections, but that's another matter.) She then contacts the site she really wants to visit: FeralAmoebae.com. The page from FeralAmoebae.com contains a URL—an IFRAME, an image, some JavaScript, or what have you—pointing to www.ReallyAwesomeSSO.com and containing some per-session unique string. Chris's browser therefore contacts www.ReallyAwesomeSSO.com and sends back the identifying cookie. When www.ReallyAwesomeSSO.com sees it, it uses its out-of-band channel to tell FeralAmoebae.com which user has connected using that unique string. (More accurately, FeralAmoebae.com will ask www.ReallyAwesomeSSO.com what user corresponds to that string.)

The best-known web-based SSO service at the moment is *Facebook Connect*.[4] Its operation (based on the IETF's OAuth 2.0 design [Hardt 2012]) is rather more complex than my outline, partly because it's a real system and not a toy example and partly for a more substantive reason: it requests *authorization* from the user for what information it will send FeralAmoebae.com (or whomever). This is part of the third question: How easy is it for the user to control what is sent to whom? In this case, the big issue is user privacy; Facebook knows a lot about people, and not all of them want all of that to be sent to every random web site they visit that happens to have a contract with Facebook.

4. "Authentication—Facebook Developers," http://developers.facebook.com/docs/authentication/.

There is an additional privacy issue: in schemes like this, the SSO knows every affiliated site you visit. Is this acceptable? It may be fine for public-facing web sites, though not necessarily to their users. For employers, though, this is rarely a good choice.

There are more general SSO schemes than the web-based one I've just outlined. Imagine a world of multiple identity providers. A user could have accounts with several of these. After logging in to one, the SSO server returns some sort of cryptographically sealed object. To log in to some other systems, the user designates which of the cached identities—that is, which of the several cryptographically protected identities she has available—should be forwarded to the desired site. That site, in turn, is told the identity provider involved and Chris's identity as known to that provider.

Again, the user interface is crucial. If nothing else, Chris wants to make sure that her employer doesn't receive her credentials from HiTechEmploymentAgencySSO.com, since it would be rather embarrassing to Chris. From the employer's perspective, it has to decide which identity providers it trusts. Only its own? Facebook's? The local government's? The one it knows is run by a front organization for MI-31? In a federated system, what can matter the most is the tuple ⟨provider, identity⟩. This is, of course, the crucial distinction between authentication and authorization: which tuples will you authorize?

There are a number of systems based on this paradigm. Microsoft's is one of the best known,[5] but there are many others, including at least one open-source system. The concept of federated identities is at the heart of the *National Strategy for Trusted Identities in Cyberspace (NSTIC)*, the White House's scheme for identification on the net [White House 2011].

5. "A Guide to Claims-Based Identity and Access Control (2nd Edition)," https://msdn.microsoft.com/en-us/library/ff423674.aspx.

There are other products intended for SSO within an organization. Kerberos, described earlier, is one such. In such cases, issues of reliability and privacy don't arise as much. However, the user interface question is still crucial: if a user has nice, transparent access to everything within a company, any malware that user is running has the same access.

7.11 Storing Passwords: Servers

How should sites store password databases? What about other sorts of authentication data? Let's look at passwords first. Note well: I assume that there *will* be some sort of security failure at your server complex; if there isn't, there's no reason to do anything fancy. But if you assume perfect security and you're wrong, the results can be disastrous.

Back when the world was young, passwords were stored in cleartext in a read-protected file. They weren't encrypted because pre-DES, there were no suitable encryption algorithms. Besides, operating system file protections were thought to be good enough. The classic Morris and Thompson paper [1979] showed why that was a bad idea, so people switched to hashed passwords but in readable files. Password-guessing attacks remained an issue, so most vendors eventually switched to hashed passwords in read-protected files.

That's all well and good for single machines. It doesn't work nearly as well for networked complexes of machines or for large server complexes with vast numbers of users who have no traditional login access to most machines. They might be users of your ISP, or subscribers to your mail service, or customers of your web site, but they all have logins and passwords. And you're probably operating at a scale not seen on single hosts; you'll have somewhere between tens of thousands and possibly tens of millions of accounts. (Facebook claims more than 1.4 billion active users, more than 18% of the world's population.[6] And they all have passwords.)

6. "Company Info," https://newsroom.fb.com/company-info/.

Clearly, many enterprises need a password or authentication server of some type. How should it be protected? The analysis has to start with two questions: what are the operational needs for this server, and what are the consequences of a breach? Initially, let's look at the second question.

The type of authentication you require plays a large role in answering this question. If your users are employing public key authentication, the database is relatively benign; a leak exposes only public keys. Public keys are, by definition, allowed to be public; seeing

one doesn't permit an attacker to learn the corresponding private keys. In that case, there is little reason to take special precautions with the data. (Under certain circumstances, it is possible for a private key and hence the corresponding public key to be derived from a password; a colleague and I proposed such a scheme many years ago [Bellovin and Merritt 1993]. This opens the risk of password-guessing attacks; see below.)

One step down is the case in which stolen data allows illicit entry to your own services, but to no others. A Kerberos database or other collections of symmetric keys fall into this category. This is clearly a disaster for you, but not for your users (except, of course, the damage they suffer from abuse of their login on your system).

The worst situation is if plaintext passwords are compromised. As I noted earlier, people reuse logins and passwords; a compromise on your site is likely to lead to a compromise of many accounts on many other sites. Quite apart from any moral blame you might incur, it is quite conceivable that you're running a risk of legal liability. Since the only reason to store a plaintext password is for password recovery, the answer is simple: don't do that.

Some people will claim that there's another reason: using a password as a key for a symmetric cipher requires both sides to have the password. That's not quite correct. What both sides need is the same shared secret; rather than the password itself, it should be some value deterministically and irreversibly derived from the password. The simplest solution is to store, say, the MD5 hash of the password; that value, though, is useful when attacking other sites that use the same scheme. Instead, hash (or better yet, HMAC [Bellare, Canetti, and Krawczyk 1996; Krawczyk, Bellare, and Canetti 1997]) the password with your service name. This value—HMAC(PW, https://www.example.com)—is useful when talking to you and only when talking to you.

Guessing attacks are a risk against any storage of data derived from passwords. The problem and its solution go back to Morris and Thompson [Morris and Thompson 1979]: add a salt and iterate the hash. More details on modern versions are given in [Kaliski 2000]. Unfortunately, that isn't straightforward in a distributed environment, where both sides may need to calculate the shared secret before any communication takes place. (*Encrypted Key Exchange (EKE)* [Bellovin and Merritt 1992] is such a protocol.) Instead, calculate a different hash:

$$H'(\text{username, site, password})$$

and use the high-order 64 bits as the salt and the low-order 18–24 bits as the iteration count. Why 18–24 bits? The purpose of the iterated hashing is to slow down dictionary attacks; if every password is hashed 100,000 times, an attacker processing a series of guesses is slowed down to 1/100,000 the previous rate. Unfortunately, the good guys are slowed down, too, so we have to pick a suitable compromise. Informal experiments show that about 300,000 iterations of MD5 are about right for slower smart phones. Salting is vital, since it guards against precomputation attacks.

Server-side considerations for iteration count are a bit more complex. You need to know the peak period login rate u (users per second); c, the number of CPUs you can dedicate to

iterated hashes; and *t*, the CPU time per hash. The maximum iteration count is then $h \cdot u/c$. If it's too low—that is, if it makes guessing attackers easier—you need more CPUs.

The challenge/response, time-based, and paper-based OTP schemes generally do not suffer from password-guessing vulnerabilities, so other sites are safe. However, the data is useful for attacking your own site's accounts and hence must be carefully protected. Lamport's scheme is quite nice in this regard; the stored data cannot be used for new authentication, on your site or elsewhere, and it inherently uses iteration. If the maximum count is set high enough, there will always be a substantial base number of iterations that an attacker will have to do even if the seed secret is derived from a password. (However, if you're relying on the limited-number-of-logins property of his scheme, you should iterate many times before using that value to seed the algorithm.)

A summary of the risks to different types of authentication data storage is given in Table 7.1.

	Scheme	*Potential Damage*
1	Plaintext passwords	Immediate login to your site and others
2	Simple hashed shared secrets	Immediate login to at least your site; easy guessing attacks against your site and others
3	Paper schemes	Immediate login to your site only
4	Time-based and challenge/response	Immediate login to your site only
5	Password-based Lamport	Guessing attacks against your site and others
6	Non-password Lamport; public key	None

Table 7.1: Risks from Compromise of Stored Authentication Data, in Approximate Order of Decreasing Risk

It is tempting to suggest that usernames also be hashed before storage. After all, knowing that user smb has password `123456` is rather different from knowing that user 79e0f325804dafbdaef73b3b17c0fd8d has that password or even password e10adc3949ba59abbe56e057f20f883e. Unfortunately, it's probably fruitless; the attacker can quite likely figure out a large portion of the usernames from other data lying around the system, and it's rather easy to do the necessary hashes.

Let's turn our attention to the other question I posed near the start of this section: What are the operational needs? A site that stores passwords also has to store secondary authentication information. Most public-facing sites and many internal sites have such information; it's quite critical and much harder to protect by hashing because there are so few choices for so many of the common fields. Place of birth? There are fewer than 20,000 incorporated places in the United States.[7] Favorite color? Most people don't know that many color names.[8] Mother's maiden name? More than 90% of American surnames can be found in fewer than 100,000 guesses.[9] One can even find lists of common pet names online. I suspect that the numbers are comparable for other countries, though the

data may be harder to obtain for some. In other words, typical secondary authentication data is almost as risky to store as plaintext passwords.

7. “Population Estimates,” http://www.census.gov/popest/data/intercensal/cities/cities2010.html.

8. “Color Survey Results,” http://blog.xkcd.com/2010/05/03/color-survey-results/.

9. “Demographic Aspects of Surnames from Census 2000,” http://www2.census.gov/topics/genealogy/2000surnames/surnames.pdf.

Other important operational needs, beyond secondary authentication, include adding users, deleting users, changing or resetting the password, and (of course) verifying a login attempt. Don’t neglect the fact that an authentication server is, among other things, a computer, which means that it has all of the usual computer needs: software maintenance, disk backup and recovery, database synchronization with the other replicas of the authentication files, sysadmin login for routine troubleshooting, and more.

There is another, more subtle concern: database consistency. It’s never a good idea to store the same data in two different places; the two instances *will* get out of sync. Sites typically store other, non-sensitive profile information on their users, whether for direct operational needs (Which mail server holds this person’s email?), revenue related (Which type of targeted ads are believed to be most effective?), or simply user preferences such as preferred language. If you store that sort of information with the authentication data, you increase the attack surface; if you store it separately, you have more consistency problems. Worse yet, things like credit card numbers are sensitive in a different way and may merit their own secure storage.

Table 7.1 makes it clear that if you are using safe authentication technologies (i.e., the last row in the table), it doesn’t much matter where you store the data; the most convenient server will suffice. Conversely, the risks from at least the first two rows and probably the first four are sufficiently great that extra care is needed. Such authentication data should be stored on a separate server, with a lot of attention paid to the protocols and operational environment. (Design issues are discussed in Chapter 11.) Only the fifth row, password-based Lamport, presents a difficult choice; keeping the data in the general user profile database is a defensible choice, but if you need secure authentication storage anyway (e.g., for secondary authentication data) you may as well put the primary data there as well (but see Chapter 11 for other considerations).

7.12 Analysis

Figure 7.4 summarizes the properties of a number of different authentication mechanisms, when dealing with different issues: threats, forgetting or losing something, and so on. What's striking is that none of the analyzed mechanisms are good under all circumstances. Password authentication, that much-maligned mechanism, is better than most when it comes to granting temporary access or the need to trust external parties. Most alternatives concentrate on the most glaring issues with passwords, users forgetting their passwords, attackers guessing them, or capture of a password by phishing sites or keystroke loggers. Almost all are weaker under other circumstances. In fact, most pairs of mechanisms fall short, too, though the combination of a password sent to a site and some form of federated authentication not relying on passwords comes close. The real benefit from avoiding passwords is that you're not vulnerable if some other password-using site is compromised.

	Guessing	Forgetting	Device loss	Server file stolen	Temp access	External trust	Phishing/ logging
Passwords	✗	✗	✓	✗	✓	✓	✗✗
Lamport's	?	✗	✗	✗	?	✓	?
Chall/resp	✓	✓	✗	✗✗	✗	✓	✓
SMS	✓	✓	?	✓	✗	?	✓
Time-based	✓	✓	✗	✗✗	?	✗	✓
Crypto	✓	✓	?	✗, ✓	?	✓	✓
Biometric	✓	✓	?	✗	✗	✓	✗✗
Federated	?	?	✓	✓	?	✗	?

✓	No particular problem; strength of this mechanism
?	Some trouble or implementation-dependent
✗	Significant risk
✗✗	Very serious risk

Figure 7.4: Properties of different authentication mechanisms.

There are no perfect solutions here. Even read requests from /dev/brain run afoul of the bilateral authentication. issue. (While that could, presumably, be solved by writes to /dev/brain by the verifying computer, the mind boggles at what could happen if that process were hacked… .) That said, sites need to pick *some* authentication solution, despite all of the limitations of usability, human frailty, and so on. A few points stand out:

- Passwords are not suitable for high-security needs. This includes most logins for medium and large enterprises. Even smaller enterprises should move away from passwords if the threat model so indicates.
- That said, passwords will not go away, even in sensitive environments; converting all applications to use stronger authentication is at best time consuming. It will be a very long time before web sites convert to any other authentication mechanism. Accordingly, technical means, such as password managers, should be used to cope with the password reuse problem. This will also help with the password strength

problem, though as noted this isn't as big a threat as is commonly trumpeted.

- Implement bilateral authentication; it's strong protection against phishing. Some password managers do this automatically: they'll send a password only if they recognize the site, and they're not fooled by clever email messages.
- Master passwords—those used with password managers or SSO systems, those used to decrypt private keys, and so forth—are especially crucial and need the best protection. These should indeed be "strong."
- Like much of security, authentication is a systems issue. Special care must be taken with secondary authentication mechanisms and password reset schemes.
- Plan for exceptions. Know in advance how to handle lost or stolen passwords, compromised servers, and more.

Finally, there are many fads in authentication. As discussed above, *all* schemes have their limitations and weaknesses. Decide based on the threat model and your operational environment.

Picking a Strong Password?

There are two strategies for picking a good password, practical and theoretical.

The practical approach is simple: use anything that won't be found by the attackers' patterns. Thus, if attackers are generating passwords based on lowercase letters, you'd be safe if your simple password used only uppercase. The problem, of course, is that you don't know what the attackers do, and they could change it easily enough. In other words, a practical approach won't work very well. (However, if you want to do it, the best thing to do is to use a multiword phrase; so few people do that that most attackers won't bother trying.)

Looking at it theoretically, you want a password space so large that it can't be searched. If you have s symbols in your "alphabet" and n letters, then the size of the guess space g is, quite obviously, $g = s^n$. Thus, in the example shown in Figure 7.1, $s = 2^{11} = 2{,}048$ and $n = 4$, giving $g = 2^{44}$ or about $1.8 \cdot 10^{13}$. What is crucial is that your password be chosen uniformly from that space.

The next question is how large g should be. That depends on the attacker's computing resources and how long you want your password to resist attack. To futureproof ourselves, increase the numbers from p. 108 to grant the attacker 1,000,000 machines that can do 10,000,000 guesses per second: 10^{13}. At that rate, the space covered by that algorithm will be exhausted in a couple of seconds. Clearly, that's not good enough against an enemy with those resources.

We can compensate by either using five words or choosing our words from a list that's twice as long. The former gives us $3.6 \cdot 10^{17}$ possibilities; the latter $2.8 \cdot$

10^{14}, and may be harder to remember because the words are less common. Choosing six words takes us to $7.3 \cdot 10^{20}$; guessing time comes to about 116 days, which is probably good enough.

Any other algorithm can be evaluated this way. Suppose you're restricted to eight characters, digits, or mixed case letters only. (Yes, there are such sites, even today.) How well do you fare? If you choose randomly, you get $n = 8$, $s = 62$, and hence $g = 2.1 \cdot 10^{14}$. That's clearly not good enough, but under those silly rules it's the best you can do. Simply using 10 characters instead of eight takes us to $8.3 \cdot 10^{18}$, which is probably adequate. Beware of simple "fixes" like adding punctuation, since many people will just append a period or comma. Suppose you did that instead of the eighth alphameric character. That *cuts* the search space to $62^7 \cdot 3$, for seven characters, those seven followed by a period, or those seven followed by a comma, giving just $1.0 \cdot 10^{13}$ choices.

I could go through more arithmetic, but it's really just a simple exercise in combinatorics. Pick your own algorithm—but remember, if you don't choose randomly from the space, the calculations are very different. Early password guessers succeeded, despite much slower computers than we have today, because people tended to pick *words*, and English words have only about 2.3 bits/letter [C. E. Shannon 1948; C. E. Shannon 1951], giving an effective $g = 3.5 \cdot 10^6$.

The algorithm from Figure 7.1? If you let a generator pick a few random words from a list, you're fine. However, if you ask the generator for ten sequences and pick the "easiest to memorize," you've cut the search space dramatically—except for the very practical point I mentioned at the top of this box.

A more interesting question is what would cause me to change these recommendations. The strengths and weaknesses of passwords are likely to remain pretty stable for the foreseeable future. On the attacking side, the basic techniques have been known since 1979; while there have been performance improvements and storage capacity changes, these are not revolutionary. We are even less likely to see changes in how humans cope with passwords, since *homo sapiens 2.0* isn't even in beta yet.

Improvements in tokens, and in particular in their cost and usability, are more likely. Many different versions have shown up over the years; with the notable exception of the SecurID, none have really caught on. It is worth restating why SecurID succeeded: they sold a complete *system*, not just tokens and basic support software. A competitor would have to match that and more; it would have to support logins to more devices of interest (smart phones? cars?) and be some combination of cheaper, more secure, or more usable. The latter is probably the greatest technological barrier, but the other issues are non-trivial. The best chance for a change is if a major vendor (probably Microsoft or Apple) were switch to tokens as the preferred login scheme, with suitable support and (most likely) subsidized tokens.

I doubt that biometrics will displace passwords in the next 10–20 years. While we will

certainly see improvements in correctness and in sensor design, some of the other issues—dealing with database compromise, changing biometrics after compromise, remote authentication—are inherent in the problem statement and will not go away.

The variable most likely to change is how people authenticate. If some other style catches on, such as federated authentication, the role of passwords will indeed diminish. Given the many variables here—cost, privacy, trust, compatibility, security, and more—it is difficult to make any concrete predictions. All that said, it does seem to be the scenario to watch most closely.