Authentication for Operating Systems

Introduction

Given that we need to deal with a wide range of security goals and security policies that are meant to achieve those goals, what do we need from our operating system? Operating systems provide services for processes, and some of those services have security implications. Clearly, the operating system needs to be careful in such cases to do the right thing, security-wise. But the reason operating system services are allowed at all is that sometimes they need to be done, so any service that the operating system might be able to perform probably should be performed — under the right circumstances.

Context will be everything in operating system decisions on whether to perform some service or to refuse to do so because it will compromise security goals. Perhaps the most important element of that context is who's doing the asking. In the real world, if your significant other asks you to pick up a gallon of milk at the store on the way home, you'll probably do so, while if a stranger on the street asks the same thing, you probably won't. In an operating system context, if the system administrator asks the operating system to install a new program, it probably should, while if a script downloaded from a random web page asks to install a new program, the operating system should take more care before performing the installation. In computer security discussions, we often refer to the party asking for something as the *principal*. Principals are security-meaningful entities that can request access to resources, such as human users, groups of users, or complex software systems.

So knowing who is requesting an operating system service is crucial in meeting your security goals. How does the operating system know that? Let's work a bit backwards here to figure it out.

Operating system services are most commonly requested by system calls made by particular processes, which trap from user code into the operating system. The operating system then takes control and performs some service in response to the system call. Associated with the calling process is the OS-controlled data structure that describes the process, so the operating system can check that data structure to determine the identity of the process. Based on that identity, the operating system now has the opportunity to make a policy-based decision on whether to perform the requested operation. In computer security discussions, the process or other active computing entity performing the request on behalf of a principal is often called its *agent*.

The request is for access to some particular resource, which we frequently refer to as the *object* of the access request[1]. Either the operating system has already determined this agent process can access the object or it hasn't. If it has determined that the process is permitted access, the OS can remember that decision and it's merely a matter of keeping

---

[1] Another computer science overloading of the word "object." Here, it does not refer to "object oriented," but to the more general concept of a specific resource with boundaries and behaviors, such as a file or an IPC channel.

track, presumably in some per-process data structure like the PCB, of that fact. For example, as we discovered when investigating virtualization of memory, per-process data structures like page tables show which pages and page frames can be accessed by a process at any given time. Any form of data created and managed by the operating system that keeps track of such access decisions for future reference is often called a *credential*.

If the operating system has not already produced a credential showing that an agent process can access a particular object, however, it needs information about the identity of the process' principal to determine if its request should be granted. Different operating systems have used different types of identity for principals. For instance, most operating systems have a notion of a user identity, where the user is, typically, some human being. (The concept of a user has been expanded over the years to increase its power, as we'll see later.) So perhaps all processes run by a particular person will have the same identity associated with them. Another common type of identity is a group of users. In a manufacturing company, you might want to give all your salespersons access to your inventory information, so they can determine how many widgets and whizz-bangs you have in the warehouse, while it wouldn't be necessary for your human resources personnel to have access to that information[2]. Yet another form of identity is the program that the process is running. Recall that a process is a running version of a program. In some systems (such as the Android Operating System), you can grant certain privileges to particular programs. Whenever they run, they can use these privileges, but other programs cannot.

Regardless of the kind of identity we use to make our security decisions, we must have some way of attaching that identity to a particular process. Clearly, this attachment is a crucial security issue. If you misidentify a janitor employee process as an accounting department employee process, you could end up with an empty bank account. (And, since the janitor is fleeing to Rio with the loot, your trash cans won't get emptied.) Or if you fail to identify your company president correctly when he's trying to give an important presentation to investors, you may find yourself out of a job once he determines that you're the one who preventing him from getting the next round of startup capital because the system didn't allow him to show his presentation.

On the other hand, since everything except the operating system's own activities are performed by some process, if we can get this right for processes, we can be pretty sure we will have the opportunity to check our policy on every important action. But we need to bear in mind one other important characteristic of operating systems' usual approach to authentication: once a principal has been authenticated, systems will almost always rely on that authentication decision for at least the lifetime of the process. This characteristic puts a high premium on getting it right. Mistakes won't be readily corrected.

---

[2] Remember the principle of least privilege from the previous chapter? Here's an example of using it. A rogue human services employee won't be able to order your warehouse emptied of pop-doodles if you haven't given such employees the right to do so. As you read through the security chapters of this book, keep your eyes out for other applications of the security principles we discussed earlier.

Which leads us to the crux of the problem.

THE CRUX OF THE PROBLEM
HOW CAN WE SECURELY IDENTIFY PROCESSES?

For systems that support processes belonging to multiple principals, how can we be sure that each process has the correct identity attached? As new processes are created, how can we be sure the new process has the correct identity? How can we be sure that malicious entities cannot improperly change the identity of a process?

Attaching Identities to Processes

Where do processes come from? Usually they are created by other processes. One simple way to attach an identity to a new process, then, is to copy the identity of the process that created it. The child inherits the parent's identity. Mechanically, when the operating system services a call from old process A to create new process B (`fork`, for example), it consults A's process control block to determine A's identity, creates a new process control block for B, and copies in A's identity. Simple, no?

That's all well and good if all processes always have the same identity. We can create a primal process when our operating system boots, perhaps assigning it some special system identity not assigned to any human user. All other processes are its descendants and all of them inherit that single identity. But if there really is only one identity, we're not going to be able to implement any policy that differentiates the privileges of one process versus another.

We must arrange that some processes have different identities and use those differences to manage our security policies. Consider a multi-user system. We can assign identities to processes based on which human user they belong to. If our security policies are primarily about some people being allowed to do some things and others not being allowed to, we now have an idea of how we can go about making our decisions.

If processes have a security-relevant identity, like a user ID, we're going to have to set the proper user ID for a new process. In most systems, a user has a process that he works with ordinarily: the shell process in command line systems, the window manager process in window-oriented system – you had figured out that both of these had to be processes themselves, right? So when you type a command into a shell or double click on an icon to start a process in a windowing system, you are asking the operating system to start a new process under your identity.

Great! But we do have another issue to deal with. How did that shell or window manager get your identity attached to itself? Here's where a little operating system privilege comes in handy. When a user first starts interacting with a system, the operating system can start a process up for him. Since the operating system can fiddle with its own data structures, like the process control block, it can set the new process' ownership to the user who just joined the system.

Again, well and good, but how did the operating system determine the user's identity so it could set process ownership properly? You probably can guess the answer – the user logged in, implying that the user provided identity information to the OS proving who he was. We've now identified a new requirement for the operating system: it must be able to query identity from human users and verify that they are who they claim to be, so we can attach reliable identities to processes, so we can use those identities to implement our security policies. One thing tends to lead to another in operating systems.

So how does the OS do that? As should be clear, we're sort of building a towering security structure with unforeseeable implications based on the OS making the right decision here, so it's very important. Let's look at our options.

How to Authenticate Users?

So this human being walks up to a computer . . .

Assuming we leave aside the possibilities for jokes, what can be done to allow the computer's system to determine who this person is, with reasonable accuracy? First, if the person is not an authorized user of the system at all, we should totally reject his attempt to sneak in. Second, if he is an authorized user, we need to determine, which one?

Classically, authenticating the identity of human beings has worked in one of three ways:

1. Authentication based on what you know
2. Authentication based on what you have
3. Authentication based on what you are

When we say "classically" here, we mean "classically" in the, well, classical sense. Classically as in going back to the ancient Greeks and Romans. For example, Polybius, writing in the second century B.C., describes how the Roman army used "watchwords" to distinguish friends from foes [P46], an example of authentication based on what you know. A Roman architect named Celer wrote a letter of recommendation (which still survives) for one of his slaves to be given to an imperial procurator at some time in the 2nd century AD [C00] —authentication based on what the slave had. Even further back, in (literally) Biblical times, the Gileadites required refugees after a battle to say the word "shibboleth," since the enemies they sought (the Ephraimites) could not properly pronounce that word [J]. This was a form of authentication by what you are: a native speaker of the Gileadites' dialect or a speaker of the Ephraimite dialect.

Having established the antiquity of these methods of authentication, let's leap past several centuries of history to the Computer Era to discuss how we use them in the context of computer authentication.

Authentication By What You Know

Authentication by what you know is most commonly performed by using passwords. Passwords have a long (and largely inglorious) history in computer security, going back at least to the CTSS system at MIT in the early 1960s [MT79]. A password is a secret

known only to the party to be authenticated. By divulging the secret to the computer's operating system when attempting to log in, the party proves his identity. (You should be wondering about whether that implies that the system must also know the password, and what further implications that might have. We'll get to that.) The effectiveness of this form of authentication depends, obviously, on several factors. We're assuming other people don't know the party's password. If they do, the system gets fooled. We're assuming that no one else can guess it, either. And, of course, that the party in question must know (and remember) it.

Let's deal with the problem of other people knowing a password first. Leaving aside guessing, how could they know it? Someone who already knows it must let it slip, so the fewer parties who have to know it, the fewer parties we have to worry about. The person we're trying to authenticate has to know it, of course, since we're authenticating him based on him knowing it. We really don't want anyone else to be able to authenticate as that person to our system, so we'd prefer no third parties know the password. Thinking broadly about what a "third party" means here, that also implies the user shouldn't write the password down on a slip of paper, since anyone who steals the paper now knows the password. But there's one more party who would seem to need to know the password: our system itself. That suggests another possible vulnerability, since the system's copy of our password might leak out[3].

Actually, though, our system does not need to know the password. Think carefully about what the system is doing when it checks the password the user provides. It's checking to see if the user knows it, not to see what that password actually is. So if the user provides us the password, but we don't know the password, how on earth could our system do that?

You already know the answer, or at least you'll slap your forehead and say "I should have thought of that" once you hear it. Store a hash of the password, not the password itself. When the user provides you with what he claims to be the password, hash his claim and compare it to the stored hashed value. If it matches, you believe he knows the password. If it doesn't, you don't. Simple, no? And now your system doesn't need to store the actual password. That means if you're not too careful with how you store the authentication information, you haven't actually lost the passwords, just their hashes. By their nature, you can't reverse hashing algorithms, so the adversary can't use the stolen hash to obtain the password.

There is a little more to it than that. The benefit we're getting by storing a hash of the password is that if the stored copy is leaked to an attacker, he doesn't know the passwords themselves. But it's not quite enough just to store something different from the password. We also want to ensure that whatever we store offers an attacker no help in guessing what the password is. If an attacker steals the hashed password, he should

---

[3] "Might" is too weak a word. The first known incident of such stored passwords leaking is from 1962 [MT79], and such leaks happen to this day with depressing regularity, and general much larger scope. [KA16] discusses a 2016 leak of over 100 million passwords stored in a readily usable form.

not be able to analyze the hash to get any clues about the password itself. There is a special class of hashing algorithms called *cryptographic hashes* that make it infeasible to use the hash to figure out what the password is, other than by actually passing a guess at the password through the hashing algorithm. One unfortunate characteristic of cryptographic hashes is that they're hard to design, so smart people don't even try. They use ones created by experts. That's what modern systems should do with password hashing: use a cryptographic hash that has been thoroughly studied and has no known flaws. At any given time, which cryptographic hashing algorithms meet those requirements may vary. At the time of this writing, SHA-3 is the US standard for cryptographic hash algorithms, and is a good choice.

TIP: AVOID STORING SECRETS

Storing secrets like plaintext passwords or cryptographic keys is a hazardous business, since the secrets usually leak out. Protect your system by not storing them if you don't need to. If you do need to, store them in a hashed form using a strong cryptographic hash. If you can't do that, encrypt them with a secure cipher. (Perhaps you're complaining to yourself that we haven't told you about those yet. Be patient.) Store them in as few places, with as few copies, as possible. Don't forget temporary editor files, backups, and the like, since the secrets may be there, too. Remember that anything you embed into an executable you give to others will not remain secret, so it's particularly dangerous to store secrets in executables.

Let's move on to the other problem: guessing. Can an attacker who wants to pose as a user simply guess the password? Consider the simplest possible password: a single bit, valued 0 or 1, like all other bits. If your password is a single bit long, then an attacker can try guessing "0" and have a 50/50 chances of being right. Even if he's wrong, if he can guess a second time, he now knows that the password is 1 and will correctly guess that.

Obviously, a one bit password is too easy to guess. How about an 8 bit password? Now there are 256 possible passwords you could choose. If the attacker guesses 256 times, he'll sooner or later guess right, taking 128 guesses, on average. Better than only having to guess twice, but still not good enough. It should be clear to you, at this point, that the length of the password is critical in being resistant to guessing. The longer the password, the harder to guess.

But there's another important factor, since we normally expect human beings to type in their passwords from keyboards or something similar. And given that we've already ruled out writing the password down somewhere as insecure, the person has to remember it. Early uses of passwords addressed this issue by restricting passwords to letters of the alphabet. While this made them typeable and easier to remember, it also cut down heavily on the number of bit patterns an attacker needed to guess to find someone's password, since all of the bit patterns that did not represent alphabetic characters would not appear in passwords. Over time, password systems have tended to expand the possible characters in a password, including upper and lower case letters, numbers, and special characters. The more possibilities, the harder to guess.

So we want long passwords composed of many different types of characters. But attackers know that people don't choose random strings of these types of characters as their passwords. They often choose names or familiar words, because those are easy to remember. Attackers trying to guess passwords will thus try lists of names and words before trying random strings of characters. This form of password guessing is called a *dictionary attack*, and it can be highly effective. The dictionary here isn't Websters (or even the OED), but rather is a specialized list of words, names, meaningful strings of numbers (like "123456"), and other character patterns people tend to use for passwords, ordered by the probability that they will be chosen as the password. A good dictionary attack can figure out 90% of the passwords for a typical site [G13].

ASIDE: PASSWORD VAULTS

One way you can avoid the problem of choosing passwords is to use what's called a password vault or key chain. This is an encrypted file kept on your computer that stores passwords. It's encrypted with a password of its own. To get passwords out of the vault, you must provide the password for the vault, reducing the problem of remembering a different password for every site to remembering one password. Also, it ensures that attackers can only use your passwords if they not only have the special password that opens the vault, but they have access to the vault itself. Of course, the benefits of securely storing passwords this way are limited to the strength of the passwords stored in the vault, since guessing and dictionary attacks will still work. Some password vaults will generate strong passwords for you—not very memorable ones, but that doesn't matter, since it's the vault that needs to remember it, not you.

If you're smart in setting up your system, an attacker really should not be able to run a dictionary attack on a login process remotely. With any care at all, the attacker will not guess a user's password in the first five or six guesses (alas, sometime literally no care is taken and the attacker will), and there's no good reason your system should allow a remote user to make 15,000 guesses at an account's password without getting it right. So by either shutting off access to an account when too many wrong guesses are made at its password, or (better) by drastically slowing down the process of password checking after a few wrong guesses (which makes a long dictionary attack take an infeasible amount of time), you can protect the account against such attacks.

But what if the attacker stole your password file? Since we assume you've been paying attention, it contains hashes of passwords, not passwords itself. But we also assume you paid attention when we told you to use a widely known cryptographic hash, and if you know it, so does the guy who stole your password file. If he's got your hashed passwords, the hashing algorithm, a dictionary, and some compute power, he can crank away at guessing your passwords at his leisure. Worse, if everyone used the same cryptographic hashing algorithm (which, in practice, they probably will), he only needs to run each possible password through the hash once and store the results. He's essentially translated his dictionary into hashed form. So when he steals your password file, he would just need to do string comparisons to your hashed passwords and his dictionary of hashed passwords. Much faster than running hashes.

There's a simple fix: before hashing a new password and storing it in your password file, generate a big random number (say 32 or 64 bits) and concatenate it to the password. Hash the result and store that. You also need to store that random number, since when the user tries to log in and provides his correct password, you'll need to take what he provided, concatenate the stored random number, and run that through the hashing algorithm. Otherwise, the password hashed by itself won't match what you stored. You typically store the random number (which is called a *salt*) in the password file right next to the hashed password. This concept was introduced in Robert Morris and Ken Thompson's early paper on password security [MT79].

Why does this help? The attacker can no longer create one translation of passwords in his dictionary to their hashes. He needs one translation for every possible salt, since the password files he steals are likely to have a different salt for every password. If the salt is 32 bits, that's 2^{32}different translations for each word in his dictionary, which makes the approach of pre-computing the translations infeasible. Instead, for each entry in the stolen password file, the dictionary attack must freshly hash each guess with the password's salt. The attack is still feasible if you have chosen passwords badly, but it's not nearly as cheap. Any good system that uses passwords and cares about security stores cryptographically hashed and salted passwords. If yours doesn't, you're putting your users at risk.

There are other troubling issues for the use of passwords, but many of those are not particular to operating systems, so we won't fling further mud at them here. Suffice it to say that there is a widely held belief in the computer security community that passwords are a technology of the past, and are no longer sufficiently secure for today's environments. At best, they can serve as one of several authentication mechanisms used in concert. This idea is called multi-factor authentication, with two-factor authentication being the version that gets the most publicity. You're perhaps already familiar with the concept: to get money out of an ATM, you need to know your personal identification number, or PIN. That's essentially a password. But you also need to provide further evidence of your identity . . .

Authentication by What You Have

Most of us have probably been in some situation where we had an identity card that we needed to show to get us into somewhere. At least, we've probably all attended some event where admission depended on having a ticket for the event. Those are both examples of authentication based on what you have, an ID card or a ticket, in these cases.

ASIDE: LINUX LOGIN PROCEDURES

Linux, in the tradition of earlier Unix systems, authenticates users based on passwords and then ties that identity to an initial process associated with the newly logged in user, much as described above. Here we will provide a more detailed step-by-step description of what actually goes on when a user steps up to a keyboard and tries to log in to a Unix system, as a solid example of how a real operating system handles this vital security issue.

1. A special login process displays a prompt asking for the user to type in his identity, in the form of a generally short user name. The user types his user name and hits carriage return. The name is echoed to the terminal.
2. The login process prompts for the user's password. The user types in the password, which is not echoed.
3. The login process looks up the name the user provided in the password file. If it is not found, the login process rejects the login attempt. If it is found, the login process determines the internal user identifier (a unique user ID number), the group (another unique ID number) that the user belongs to, the initial command shell that should be provided to this user once login is complete, and the home directory that shell should be started in. Also, the login process finds the salt and the salted, hashed version of the correct password for this user, which are permanently stored in a secure place in the system.
4. The login process combines the salt for the user's password and the password provided by the user and performs the hash on the combination. It compares the result to the stored version obtained in the previous step. If they do not match, the login process rejects the login attempt.
5. If they do match, fork a process. Set the user and group of the forked process to the values determined earlier. Change directory to the user's home directory and exec the shell process associated with this user (both the directory name and the type of shell were determined in step 3).

There are some other details associated with ensuring that we can log in another user on the same terminal after this one logs out that we don't go into here.

Note that in steps 3 and 4, login can fail either because the user name is not present in the system or because the password does not match the user name. Linux and most other systems do not indicate which condition failed, if one of them did. This choice prevents attackers from learning the names of legitimate users of the system just by typing in guesses, since they cannot know if they guessed a non-existent name or guessed the wrong password for a legitimate user name. Not providing useful information to non-authenticated users is generally a good security idea that has applicability in other types of systems.

Think a bit about why Linux's login procedure chooses to echo the typed user name when it doesn't echo the password. Is there no security disadvantage to echoing the user name, is it absolutely necessary to echo the user name, or is it a tradeoff of security for convenience? Why not echo the password?

When authenticating yourself to an operating system, things are a bit different. In special cases, like the ATM mentioned above, the device (which is, after all, a computer inside – you knew that, right?) has special hardware to read our ATM card. That hardware allows it to determine that, yes, we have that card, thus providing the further proof to go along with your PIN. Most desktop computers, laptops, tablets, smart phones, and the like do not have that special hardware. So how can they tell what we have?

If we have something that plugs into one of the ports on a computer, such as a hardware token that uses USB, then, with suitable software support, the operating system can tell whether the user trying to log in has the proper device or not. Some security tokens (sometimes called *dongles*) are designed to work that way.

In other cases, since we're trying to authenticate a human user anyway, we make use of the person's capabilities to transfer information from whatever it is we have to the system we need to authenticate ourselves to. For example, some smart tokens display a number or character string on a tiny built-in screen. The human user types the information read off that screen into the computer's keyboard. The operating system does not get direct proof that the user has the device, but if only someone with access to the device could know what information he was supposed to type in, the evidence is nearly as good.

These kinds of devices rely on frequent changes of whatever information the device passes (directly or indirectly) to the operating system, perhaps every few seconds, perhaps every time the user tries to authenticate himself. Why? Well, if it doesn't, anyone who can learn the static information from the device no longer needs the device to pose as the user. The authentication mechanism has been converted from "something you have" to "something you know," and its security now depends on how hard it is for an attacker to learn that secret.

One weak point for all forms of authentication based on what you have is, what if you don't have it? What if you left it on your dresser bureau this morning? What if it slipped out of your pocket on your commute to work? What if a subtle pickpocket brushed up against you at the coffee shop and made off with it? You now have a two-fold problem. First, you don't have the magic item you need to authenticate yourself to the operating system. You can whine at your computer all you want, but it won't care. It will continue to insist that you produce the magic item you lost. Second, someone else has your magic item, and possibly they can pretend to be you, fooling the operating system that was relying on authentication by what you have. Note that the multi-factor authentication we mentioned earlier can save your bacon here, too. If the thief stole your security token, but doesn't know your password, he'll still have to guess that before he can pose as you.

If you study system security in practice for very long, you'll find that there's a significant gap between what academics (like me) tell you is safe and what happens in the real world. Part of this gap is because the real world needs to deal with real issues, like user convenience. Part of it is because security academics have a tendency to denigrate anything where they can think of a way to subvert it, even if that way is not itself particularly practical. One example in the realm of authentication mechanisms based on

what you have is authenticating a user to a system by sending a text message to the user's cell phone. The user then types his message into the computer. Thinking about this in theory, this sounds very weak. In addition to the danger of losing the phone, security experts like to think about exotic attacks where the text message is misdirected to the attacker's phone, allowing him to provide the secret information from the text message to the computer.

In practice, people usually have their phone with them and take reasonable care not to lose it. If they do lose it, they notice that quickly and take equally quick action to fix their problem. So there is likely to be a relatively small window of time between when your phone is lost and when systems learn that they can't authenticate you using that phone. Also in practice, redirecting text messages sent to cell phones is possible, but far from trivial. The effort involved is likely to outweigh any benefit the attacker would get from fooling the authentication system, at least in the vast majority of cases. So a mechanism that causes security purists to avert their gazes in horror in actual use provides quite reasonable security[4]. Keep this lesson in mind. Even if it isn't on the test, it may come in handy some time in your later career.

Authentication by What You Are

If you don't like methods like passwords and you don't like having to hand out smart cards or security tokens to your users, there is another option. Human beings (who are what we're talking about authenticating here) are unique creatures with physical characteristics that differ from all others, sometimes in subtle ways, sometimes in obvious ones. In addition to properties of the human body (from DNA at the base up to the appearance of our face at the top), there are characteristics of human behavior that are unique, or at least not shared by very many others. This observation suggests that if our operating system can only accurately measure these properties or characteristics, it can distinguish one person from another, solving our authentication problem.

This approach is very attractive to many people, most especially to those who have never tried to make it work. Going from the basic observation to a working, reliable authentication system is far from easy. But it can be made to work, to much the same extent as the other authentication mechanisms. We can use it, but it won't be perfect, and has its own set of problems and challenges.

Remember that we're talking about a computer program (either the OS itself or some separate program it invokes for the purpose) measuring a human characteristic and determining if it belongs to a particular person. Think about what that entails. What if we plan to use facial recognition with the camera on a smart phone to authenticate the owner of the phone? If we decide it's the right person, we allow whoever we took the

---

[4] However, in 2016 the United States National Institute of Standards and Technology issued draft guidance deprecating the use of this technique for two-factor authentication, at least in some circumstances. Here's another security lesson: what works today might not work tomorrow.

picture of to use the phone. If not, we give them the raspberry (in the cyber sense) and keep them out.

You should have identified a few challenges here. First, the camera is going to take a picture of someone who is, presumably, holding the phone. Maybe it's the owner, maybe it isn't. That's the point of taking the picture. If it isn't, we should assume whoever it is would like to fool us into thinking he's the actual owner. What if it's someone who looks a lot like the right user, but isn't? What if he's wearing a mask? What if he holds up a photo of the right user, instead of showing his own face? What if the lighting is dim, or he's not fully facing the camera? Alternately, what if it is the right user and he's not facing the camera, or the lighting is dim, or he just shaved off his beard?

Computer programs don't recognize faces the way people do. They do what programs always do with data: they convert it to zeros and ones and process it. So that "photo" you took is actually a collection of numbers, indicating shadow and light, shades of color, contrasts, and the like. OK, now what? Time to decide if it's the right person's photo or not! How?

If it were a password, we could have stored the right password (or, better, a hash of the right password) and done a comparison of what got typed in (or its hash) to what we stored. If it's a perfect match, authenticate. Otherwise, don't. Can we do the same with this collection of zeros and ones that represent the picture we just took? Can we have a picture of the right user stored permanently in some file and compare the data from the camera to that file?

Probably not in the same way we compared the passwords. Consider one of those factors we just mentioned above: lighting. If the picture we stored in the file was taken under bright lights and the picture coming out of the camera was taken under dim lights, the two sets of zeros and ones are most certainly not going to match. In fact, it's quite unlikely that two pictures of the same person, taken a second apart under identical conditions, would be represented by exactly the same set of bits. So clearly we can't do a comparison based on bit-for-bit equivalence.

Instead, we need to compare based on a higher-level analysis of the two photos, the stored one of the right user and the just-taken one of the person who claims to be that user. Generally this will involve extracting higher-level features from the photos and comparing those. We might, for example, try to calculate the length of the nose, or determine the color of the eyes, or make some kind of model of the shape of the mouth. Then we would compare the same feature set from the two photos.

Even here, though, an exact match is not too likely. The lighting, for example, might slightly alter the perceived eye color. So we'll need to allow some sloppiness in our comparison. If the feature match is "close enough," we authenticate. If not, we don't. We will look for close matches, not perfect matches, which brings the nose of the camel of tolerances into our authentication tent. If we are intolerant of all but the closest matches, on some days we will fail to match the real user's picture to the stored version. (That's called a *false negative*, since we incorrectly decided not to authenticate.) If we are too tolerant of differences in the measured versus stored data, we will authenticate a

user who is not who he claims to be. (That's called a *false positive*, since we incorrectly decided to authenticate.)

ASIDE: SECURITY SNAKE OIL

If you spend much time worrying about security in a production environment, you'll run into a lot of folks who would like you to pay them for their security product, which invariably is described as tremendously improving the security of your environment. Some of these products are excellent and will indeed help you secure your systems. Others, however, are of no use at all. Like Old West medicine show barkers, they're just selling snake oil that won't cure anything.

Biometrics are one area where this phenomena arises. If you listen to the manufacturers, no one ever marketed a biometric authentication system that was less than perfect. Unfortunately, that's not always quite the truth. A good cautionary tale comes from Japanese researchers who obtained copies of 11 commercially available fingerprint readers, went to a hobby store and purchased around $10 worth of supplies, and tried to deceive the readers. They used the hobby supplies to make gummy fingers, copying fingerprints onto them in various ways. They were able to fool the fingerprint readers at rates between 68% and 100% using their gummy fingers [M+02].

Security snake oil shows up in other places, like cryptography. The bigger the claims, the less likely they are to be true. The more "revolutionary" the technology, the more likely it is to be ineffective. To complicate life, sometimes the claims are true, but remember that an extraordinary claim requires extraordinary proof [TR78]

The nature of biometrics is that any implementation will have a characteristic false positive and false negative rate. Both are bad, so you'd like both to be low. For any given implementation of some biometric authentication technique, you can typically tune it to achieve some false positive rate, or tune it to achieve some false negative rate. But you can't usually minimize both. As the false positive rate goes down, the false negative rate goes up, and vice versa.

Figure 1 shows the typical relationship between these error rates. Note the circle at the point where the two curves cross. That point represents the *crossover error rate*, a common metric for describing the accuracy of a biometric. It represents an equal tradeoff between the two kinds of errors. It's not always the case that one tunes a biometric system to hit the crossover error rate, since you might care more about one kind of error than the other. For example, a smart phone that frequently locks its legitimate user out because it doesn't like today's fingerprint reading is not going to be popular, while the chances of a thief who stole the phone having a similar fingerprint are low. Perhaps low false negatives matter more here. On the other hand, if you're opening a bank vault with a retinal scan, once in a while requiring the bank manager to provide a scan a second time isn't too bad, while allowing a robber to open the vault with his bogus fake eye would be a disaster. Low false positives might be better here.

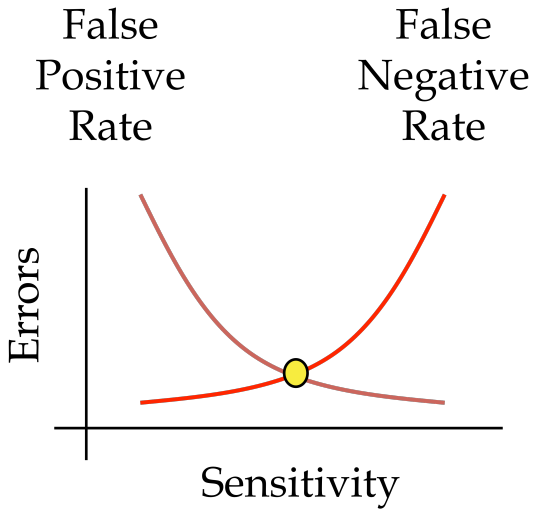

Figure 1. The Crossover Error Rate

Leaving aside the issues of reliability of authentication using biometrics, another big issue for using human characteristics to authenticate is that many of the techniques for measuring them require special hardware not likely to be present on most machines. Many computers (including smart phones, tablets, and laptops) are likely to have cameras, but embedded devices and server machines probably don't. Relatively few machines have fingerprint readers, and even fewer are able to measure more exotic biometrics. While a few biometric techniques (such as measuring typing patterns) require relatively common hardware that is likely to be present on many machines anyway, there aren't many such techniques. Even if a special hardware device is available, the convenience of using them for this purpose can be limiting.

One further issue you want to think about when considering using biometric authentication is whether there is any physical gap between where the biometric quantity is measured and where it is checked. In particular, checking biometric readings provided by an untrusted machine across the network is hazardous. What comes in across the network is simply a pattern of bits spread across one or more messages, whether it represents a piece of a web page, a phoneme in a VoIP conversation, or part of a scanned fingerprint. Bits is bits, and anyone can create any bit pattern they want. If an adversary knows what the bit pattern representing your fingerprint looks like, they may not need your finger, or even a fingerprint scanner, to create it and feed it to your machine. When the hardware performing the scanning is physically attached to your machine, there is less opportunity to slip in a spurious bit pattern that didn't come from the device. When the hardware is on the other side of the world on a machine you have no control over, there is a lot more opportunity. The point here is to be careful with biometric authentication information provided to you remotely.

ASIDE: OTHER AUTHENTICATION POSSIBILITIES

Usually, what you know, what you have, and what you are cover the useful authentication possibilities, but sometimes there are other options. Consider going into the Department of Motor Vehicles to apply for a driver's license. You probably go up to a counter and talk to someone behind that counter, perhaps giving him a bunch of personal information, maybe even giving him some money to cover a fee for the license. Why on earth did you believe that person was actually a DMV employee who was able to get you a legitimate driver's license? You probably didn't know him, he didn't show you an official ID card, he didn't recite the secret DMV mantra that proved he was an initiate of that agency. You believed it because he was standing behind a particular counter, which is the counter DMV employees stand behind. You authenticated him based on where he was.

Once in a while, that approach can be handy in computer systems, most frequently in mobile or pervasive computing. If you're tempted to use it, think carefully about how you're obtaining the evidence that the subject really is in a particular place. It's actually fairly tricky.

What else? Perhaps you can sometimes authenticate based on what someone does. If you're looking for personally characteristic behavior, like their typing pattern or delays between commands, that's a type of biometric. (Google plans to introduce multifactor authentication of this kind in its Android phones by 2017, for example.) But you might be less interested in authenticating exactly who they are versus authenticating that they belong to the set of Well Behaved Users. Many web sites, for example, care less about who their visitors are and more about whether they use the web site properly. In this case, you might authenticate their membership in the set by their ongoing interactions with your system.

In all, it sort of sounds like biometrics are pretty terrible for authentication, but that's the wrong lesson. For that matter, previous sections probably made it sound like all methods of authentication are terrible. Certainly none of them are perfect, but your task as a system designer is not to find the perfect authentication mechanism, but to use mechanisms that are well suited to your system and its environment. A good fingerprint reader built in to a smart phone might do its job quite well. A long, unguessable password can provide a decent amount of security. Well-designed smart cards can make it nearly impossible to authenticate yourself without having them in your hand. And where each type of mechanism fails, you can perhaps correct for that failure by using a second or third authentication mechanism that doesn't fail in the same cases.

Authenticating Non-Humans

No, we're not talking about aliens or extra-dimensional beings, or even your cat. If you think broadly about how computers are used today, you'll see that there are many circumstances in which no human user is associated with a process that's running. Consider a web server. There really isn't some human user logged in whose identity should be attached to the web server. Or think about embedded devices, such as a smart

light bulb. Nobody logs in to a light bulb, but there is certainly code running there, and quite likely it is process-oriented code.

Mechanically, the operating system need not have a problem with the identities of such processes. Simply set up a user called `webserver` or `lightbulb` on the system in question and attach the identity of that "user" to the processes that are associated with running the web server or turning the light bulb on and off. But that does lead to the question of how you make sure that only real web server processes are tagged with that identity. We wouldn't want some arbitrary user on the web server machine creating processes that appear to belong to the server, rather than to that user.

One approach is to use passwords for these non-human users, as well. Simply assign a password to the web server user. When does it get used? When it's needed, which is when you want to create a process belonging to the web server, but you don't already have one in existence. The system administrator could log in as the web server user, creating a command shell and using it to generate the actual processes the server needs to do its business. As usual, the processes created by this shell process would inherit their parent's identity, `webserver`, in this case. More commonly, we skip the middleman (here, the login) and provide some mechanism whereby the privileged user is permitted to create processes that belong not to him, but to some other user, such as `webserver`. Alternately, we can provide a mechanism that allows a process to change its ownership, so the web server processes would start off under some other user's identity (such as the system administrator's) and change their ownership to `webserver`. Yet another approach is to allow a temporary change of process identity, while still remembering the original identity. (We'll say more about this last approach in a future chapter.) Obviously, any of these approaches require strong controls, since they allow one user to create processes belonging to another user.

As mentioned above, passwords are the most common authentication method used to determine if a process can be assigned to one of these non-human users. Sometimes no authentication of the non-human user is required at all, though. Instead, certain other users (like trusted system administrators) are given the right to assign new identities to the processes they create, without providing any further authentication information than their own. In Linux and other Unix systems, the `sudo` command offers this capability. For example,

```
sudo –u webserver apache2
```

would indicate that the `apache2` program should be started under the identity of `webserver`, rather than under the identity of whoever ran it. This command might require the user running it to provide his own authentication credentials (for extra certainty that it really is the privileged user asking for it, and not some random visitor accessing his computer during the privileged user's coffee break), but would not require authentication information associated with `webserver`. Any sub-processes created by `apache2` would, of course, inherit the identity of `webserver`. We'll say more about `sudo` in the chapter on access control.

One final identity issue we alluded to earlier is that sometimes we wish to identify not just individual users, but groups of users who share common characteristics, usually security-related characteristics. For example, we might have four or five system administrators, any one of whom is allowed to start up the web server. Instead of associating the privilege with each one individually, it's advantageous to create a system-meaningful group of users with that privilege. We would then indicate that the four or five administrators are members of that group. This kind of group is another example of a security-relevant principal, since we will make our decisions on the basis of group membership, rather than individual identity. When one of the system administrators wished to do something requiring group membership, we would check that she was a member. We can either associate a group membership with each process, or use the process' individual identity information as an index into a list of groups that people belong to. The latter is more flexible, since it allows us to put each user into an arbitrary number of groups.

Most modern operating systems, including Linux and Windows, support these kinds of groups, since they provide ease and flexibility in dealing with application of security policies. They handle group membership and group privileges in manners largely analogous to those for individuals. For example, a child process will usually have the same group-related privileges as its parent. When working with such systems, it's important to remember that group membership provides a second path by which a user can obtain access to a resource, which has its benefits and its dangers.

Summary

If we want to apply security policies to actions taken by processes in our system, we need to know the identity of the processes, so we can make proper decisions. We start the entire chain of processes by creating a process at boot time belonging to some system user whose purpose is to authenticate users. They log in, providing authentication information in one or more forms to prove their identity. The system verifies their identity using this information and assigns their identity to a new process that allows the user to go about his business, which typically involves running other processes. Those other processes will inherit the user's identity from their parent process. Special secure mechanisms can allow identities of processes to be changed or to be set to something other than the parent's identity. The system can then be sure that processes belong to the proper user and can make security decisions accordingly.

Historically and practically, the authentication information provided to the system is either something the authenticating user knows (like a password or PIN), something he has (like a smart card or proof of possession of his smart phone), or something he is (like his fingerprint or voice scan). Each of these approaches has its strengths and weaknesses. A higher degree of security can be obtained by using multi-factor authentication, which requires a user to provide evidence of more than one form, such as requiring both a password and a one-time code that was texted to his smart phone.

References

[C00] Letter of recommendation to Tiberius Claudius Hermeros
Celer the Architect
Circa 100 A.D.
This letter introduced a slave to the imperial procurator, thus providing said curator evidence that the slave was who he claimed to be. You can read it in translation at http://papyri.info/ddbdp/c.ep.lat;;81.

[G13] "Anatomy of a hack: even your 'complicated' password is easy to crack"
Dan Goodin
http://www.wired.co.uk/article/password-cracking, May 2013.
A description of how three experts used dictionary attacks to guess a large number of real passwords, with 90% success.

[J] Judges 12, verses 5-6
The Bible
An early example of the use of biometrics. Failing this authentication had severe consequences, as the Gileadites slew mispronouncers, some 42,000 of them according to Judges.

[KA16] "VK.com Hacked! 100 Million Clear Text Passwords Leaked Online"
Swati Khandelwal
http://thehackernews.com/2016/06/vk-com-data-breach.html
One of many recent reports of stolen passwords stored in plaintext form.

[MT79] "Password Security: A Case History"
Robert Morris and Ken Thompson
Communications of the ACM, Vol. 22, No. 11, 1979.
A description of the use of passwords in early Unix systems. It also talks about password shortcomings from more than a decade earlier, in the CTSS system. And it was the first paper to discuss the technique of password salting.

[M+02] "Impact of Artificial "Gummy" Fingers on Fingerprint Systems"
Tsutomu Matsumoto, Hiroyuki Matsumoto, Koji Yamada, and Satoshi Hoshino
SPIE Vol. #4677, January 2002.
A neat example of how simple ingenuity can reveal the security weaknesses of systems. In this case, the researchers showed how easy it was to fool commercial fingerprint reading machines.

[P46] "The Histories"
Polybius
Circa 146 B.C.
A history of the Roman Republic up to 146 B.C. Polybius provides a reasonable amount of detail not only about how the Roman Army used watchwords to authenticate themselves, but how they distributed them where they needed to be, which is still a critical element of using passwords.

[TR78] "On the Extraordinary: An Attempt at Clarification"
Marcello Truzzi
Zetetic Scholar, Vol. 1, No. 1, p. 11, 1978.
Truzzi was a scholar who investigated various pseudoscience and paranormal claims. He is unusual in this company in that he insisted that one must actually investigate such claims before dismissing them, not merely assume they are false because they conflict with scientific orthodoxy.