Distributed System Security

Introduction

An operating system can only control its own machine's resources. Thus, operating systems will have challenges in providing security in distributed systems, where more than one machine must cooperate. There are two large problems:

1. The other machines in the distributed system might not properly implement the security policies you want, or they might be adversaries impersonating trusted partners. We cannot control remote systems, but we still have to be able to trust validity of the credentials and capabilities they give us.
2. Machines in a distributed system communicate across a network that none of them fully control and that, generally, cannot be trusted. Adversaries often have equal access to that network and can forge, copy, replay, alter, destroy, and delay our messages, and generally interfere with our attempts to use the network.

As suggested earlier, cryptography will be the major tool we use here, but we also said cryptography was hard to get right. That makes it sound like the perfect place to use carefully designed standard tools, rather than to expect everyone to build their own. That's precisely correct. So,

THE CRUX OF THE PROBLEM
HOW TO PROTECT DISTRIBUTED SYSTEM OPERATIONS?

How can we secure a system spanning more than one machine? What tools are available to help us protect such systems? How do we use them properly? What are the areas in using the tools that require us to be careful and thoughtful?

The Role of Authentication

How can we handle our uncertainty about whether our partners in a distributed system are going to enforce our security policies? In most cases, we can't do much. At best, we can try to arrange to agree on policies and hope everyone follows through on those agreements. There are some special cases where we can get high-quality evidence that our partners have behaved properly, but that's not easy, in general. For example, how can we know that they are using full disk encryption, or that they have carefully wiped an encryption key we are finished using, or that they have set access controls on the local copies of their files properly? They can say they did, but how can we know?

Generally, we can't. But you're used to that. In the real world, your friends and relatives know some secrets about you, and they might have keys to get into your home, and if you loan them your car you're fairly sure you'll get it back. That's not so much because you have perfect mechanisms to prevent those trusted parties from behaving badly, but because you are pretty sure they won't. If you're wrong, perhaps you can detect that they haven't behaved well and take compensating actions (like changing your locks or calling the police to report your car stolen). We'll need to rely on similar approaches in distributed computer systems. We will simply have to trust that some parties will behave well. In some cases, we can detect when they don't and adjust our trust in the parties accordingly, and maybe take other compensating actions.

Of course, in the cyber world, our actions are at a distance over a network, and all we see are bits going out and coming in on the network. For a trust-based solution to work, we have to be quite sure that the bits we send out can be verified by our buddies as truly coming from us, and we have to be sure that the bits coming in really were created by them. That's a job for authentication. As suggested in the earlier authentication chapter, when working over a network, we need to authenticate based on a bundle of bits. Most commonly, we use a form of authentication based on what you know. Now, think back to the earlier chapters. What might someone running on a remote operating system know that no one else knows? How about a password? How about a private key?

Most of our distributed system authentication will rely on one of these two elements. Either you require the remote machine to provide you with a password, or you require it to provide evidence using a private key stored only on that machine[1]. In each case, you need to know something to check the authentication: either the password (or, better, a cryptographic hash of the password plus a salt) or the public key.

When is each appropriate? Passwords tend to be useful if there are a vast number of parties who need to authenticate themselves to one party. Public keys tend to be useful if there's one party who needs to authenticate himself to a vast number of parties. Why? With a password, the authentication provides evidence that somebody knows a password. If you want to know exactly who that is (which is usually important), only the party authenticating and the party checking can know it. With a public key, many parties can know the key, but only one party who knows the matching private key can authenticate himself. So we tend to use both mechanisms, but for different cases. When a web site authenticates itself to a user, it's done with PK cryptography. By distributing one single public key (to vast numbers of users), the web site can be authenticated by all its users. The web site need not bother keeping separate authentication information to authenticate itself to each user. When that user authenticates itself to the web site, it's done with a password. Each user must be separately authenticated to the web site, so we require a unique piece of identifying information for that user, preferably something that's easy for a person to use. Setting up and distributing public keys is hard, while setting up individual passwords is relatively easy.

How, practically, do we use each of these authentication mechanisms in a distributed system? If we want a remote partner to authenticate itself via passwords, we will require it to provide us with that password, which we will check. We'll need to encrypt the transport of the password across the network if we do that, since otherwise anyone

[1] We occasionally use other methods, such as smart cards or remote biometric readers. They are less common in today's systems, though. If you understand how we use passwords and public key cryptography for distributed system authentication, you can probably figure out how to make proper use of these other techniques, too. If you don't, you'll be better off figuring out the common techniques before moving to the less common ones.

eavesdropping on the network (which is easy for many wireless networks) will readily learn passwords sent unencrypted. Encrypting the password will require that we already have either a shared symmetric key or our partner's public key. So let's concentrate for the moment on how we handle getting that public key, either to use it directly or to set up the cryptography to protect the password in transit.

We'll spend the rest of the chapter on securing the network connection, but please don't forget that even if you secure the network perfectly, you still face the major security challenge of the uncontrolled site you're interacting with on the other side of the network. If your compromised partner attacks you, it will offer little consolation that his attack was authenticated and encrypted.

Public Key Authentication for Distributed Systems

The public key doesn't need to be secret, but we need to be sure it really belongs to our partner. If we have a face-to-face meeting with him, he can give us his public key in some form or another, in which case we can be pretty sure it's his. That's limiting, though, since we often interact with partners who we never see face to face. For that matter, whose "face" belongs to Amazon or Google?

Fortunately, we can use the fact that secrecy isn't required to simply create a bunch of bits containing the public key. Anyone who gets a copy of the bits has the key. But how do they know for sure whose key it is? What if some other trusted party known to everyone who needs to authenticate our partner used their own public key to cryptographically sign that bunch of bits, verifying that they do indeed belong to our partner? If we could check that signature, we could then be sure that bunch of bits really does represent our partner's public key, at least to the extent that we trust that third party who did the signature.

This technique is how we actually authenticate web sites and many other entities on the Internet. Every time you browse the web or perform any other web-based activity, you use it. The signed bundle of bits is called a *certificate*. Essentially, it contains information about the party that owns the public key, the public key itself, and other information, such as an expiration date. The entire set of information, including the public key, is run through a cryptographic hash, and the result is encrypted with the trusted third party's private key, digitally signing the certificate. If you obtain a copy of the certificate, and can check the signature, you can learn someone else's public key, even if you have never met or had any direct interaction with them. In certain ways, it's a beautiful technology that empowers the whole Internet.

Let's briefly go through an example, to solidify the concepts. Let's say Frobazz Inc. wants to obtain a certificate for its public key, which is K_F. Frobazz Inc. pays big bucks to Acmesign Co., a widely trusted company whose business it is to sell certificates, to obtain a certificate signed by AcmeSign. Such companies are commonly called Certificate Authorities, or CAs, since they create authoritative certificates trusted by many parties. Acmesign checks up on Frobazz Inc. to ensure that the people asking for the certificate actually are legitimate representatives of Frobazz. Acmesign then makes very, very sure that the public key it's about to embed in a certificate actually is the one that Frobazz wants to use. Assuming it is, Acmesign runs a cryptographic hashing algorithm (perhaps SHA-3 which, unlike SHA-1, has not been cracked, as of 2018) on Frobazz's name, public key K_F, and other information, producing hash H_F. Acmesign then encrypts H_F with its own private key, P_A, producing digital signature S_F. Finally, Acmesign combines all of the information used to produce H_F, plus Acmesign's own identity and the signature S_F, into the certificate C_F, which it hands over to Frobazz, presumably in exchange for a bunch of money. Remember, C_F is just a bunch of bits.

Now Frobazz Inc. wants to authenticate itself over the Internet to one of its customers. If the customer already has Frobazz's public key, we can use public key authentication mechanisms directly. If the customer does not have the public key, Frobazz sends C_F to the customer. The customer examines the certificate, sees that it was generated by Acmesign using, say, SHA-3, and runs the same information that Acmesign hashed (all of which is in the certificate itself) through SHA-3, producing H_F'. Then the customer uses Acmesign's public key to decrypt S_F (also in the certificate), obtaining H_F. If all is well, H_F equals H_F', and now the customer knows that the public key in the certificate is indeed Frobazz's. Public key authentication can proceed[2]. If the two hashes aren't exactly the same, the customer knows that something fishy is going on and will not accept the certificate.

There are some wonderful properties about this approach to learning public keys. First, note that the signing authority (Acmesign, in our example) did not need to participate in the process of the customer checking the certificate. In fact, Frobazz didn't really, either. The customer can get the certificate from literally anywhere and obtain the same degree of assurance of its validity. Second, it only needs to be done once per customer. After obtaining the certificate and checking it, the customer has the public key he needs. From that point onward, he can simply store it and use it. If, for whatever reason, he loses it, he can either extract it again from the certificate (if that has been saved), or go through the process of obtaining the certificate all over again. Third, the customer had no need to trust the party claiming to be Frobazz until that identity had been proven by checking the certificate. The customer can keep that party at arm's length and proceed with caution until the certificate checks out.

Assuming you've been paying attention for the last few chapters, you should be saying to yourself, "now, wait a minute, isn't there a chicken-and-egg problem here?" We'll learn Frobazz's public key by getting a certificate for it. The certificate will be signed by Acmesign. We'll check the signature by knowing Acmesign's public key. But where did we get Acmesign's key? We really hope you did have that head-scratching moment and asked yourself that question, because if you did, you understand the true nature of the Internet authentication problem. Ultimately, we've got to bootstrap it. You've got to somehow or other obtain a public key for somebody that you trust. Once you do, if it's the right public key for the right kind of party, you can then obtain a lot of other public keys. But without something to start from, you can't do much of anything.

[2] And, indeed, must, since all this business with checking the certificate merely told the customer what Frobazz's public key was. It did nothing to assure the customer that whoever sent him the certificate actually was Frobazz or knew Frobazz's private key.

Where do you get that primal public key? Most commonly, it comes in a piece of software you obtain and install. The one you use most often is probably your browser, which typically comes with the public keys for several hundred trusted authorities. (You do know of several hundred companies out there that you trust with everything you do on the web, don't you? Well, know of them or not, you effectively trust them to that extent.) Whenever you go to a new web site that cares about security, it provides you with a certificate containing that site's public key, and signed by one of those trusted authorities pre-configured into your browser. You use the pre-configured public key of that authority to verify that the certificate is indeed proper, after which you know the public key of that web site. From that point onward, you can use the web site's public key to authenticate it. There are some serious caveats here (and some interesting approaches to addressing those caveats), but let's put those aside for the moment.

Anyone can create a certificate, not just those trusted CAs, either by getting one from someone whose business it is to issue certificates or simply by creating one from scratch, following a certificate standard (X.509 is the most commonly used certificate standard [I12]). The necessary requirement is that the party being authenticated and the parties performing the authentication must all trust whoever created the certificate. If they don't trust that party, why would they believe the certificate is correct?

If you are building your own distributed system, you can create your own certificates from a machine you (and other participants in the system) trust and can handle the bootstrapping issue by carefully hand-installing the certificate signing machine's public key wherever it needs to be. There are a number of existing software packages for creating certificates, and, as usual with critical cryptographic software, you're better off using an existing, trusted implementation rather than coding up one of your own. One example you might want to look at is PGP (available in both supported commercial versions and compatible but less supported free versions) [P16], but there are others. If you are working with a fixed number of machines and you can distribute the public key by hand in some reasonable way, you can dispense entirely with certificates. Remember, the only point of a PK certificate is to distribute the public key, so if your public keys are already where they need to be, you don't need certificates.

OK, one way or another you've obtained the public key you need to authenticate some remote machine. Now what? Well, anything they send you encrypted with their private key will only decrypt with their public key, so anything that decrypts properly with the public key must have come from them, right? Yes, it must have come from them at some point, but it's possible for an adversary to have made a copy of a legitimate message the site sent at some point in the past and then send it again it at some future date. Depending on exactly what's going on, that could cause trouble, since you may take actions based on that message that the legitimate site did not ask for. So usually we take measures to ensure that we're not being subjected to a *replay* attack. Such measures generally involve ensuring that each encrypted message contains unique information not in any other message. This feature is built in properly to standard cryptographic protocols, so if you follow our advice and use one of those, you will get protection from such replay attacks. If you insist on building your own cryptography, you'll need to learn a good deal more about this issue and will have to apply that knowledge very carefully.

Also, public key cryptography is expensive. We want to stop using it as soon as possible, but we also want to continue to get authentication guarantees. We'll see how to do that when we discuss SSL and TLS.

Password Authentication for Distributed Systems

The other common option used to authenticate in distributed systems is to use a password. As noted above, that will work best in situations where only two parties need to deal with any particular password: the party being authenticated and the authenticating party. They make sense when an individual user is authenticating himself to a site that hosts many users, such as when you log in to Amazon. They don't make sense when that site is trying to authenticate itself to an individual user, such as when a web site claiming to be Amazon wants to do business with you. Public key authentication works better there.

How do we properly handle password authentication over the network, when it is a reasonable choice? The password is usually associated with a particular user ID, so the user provides that ID and password to the site requiring authentication. That typically happens over a network, and typically we cannot guarantee that networks provide confidentiality. If our password is divulged to someone else, they'll be able to pose as us, so we must add confidentiality to this cross-network authentication, generally by encrypting at least the password itself (though encrypting everything involved is better). So a typical interchange with Alice trying to authenticate herself to Frobazz Inc.'s web site would involve the site requesting a user ID and password and Alice providing both, but encrypting them before sending them over the network.

The obvious question you should ask is, encrypting them with what key? Well, if Frobazz authenticated itself to Alice using PK, as discussed above, Alice can encrypt her user ID and password with Frobazz's public key. Frobazz Inc., having the matching private key, will be able to check them, but nobody else can read them. In actuality, there are various reasons why this alone would not suffice, including replay attacks, as mentioned above. But we can and do use Frobazz's private key to set up cryptography that will protect Alice's password in transit. We'll discuss the details in the section on SSL/TLS.

We discussed issues of password choice and management in the chapter on authentication, and those all apply in the networking context. Otherwise, there's not that much more to say about how we'll use passwords, other than to note that after the remote site has verified the password, what does it actually know? That the site or user who sent the password knows it, and, to the strength of the password, that site or user is who it claims to be. But what about future messages that come in, supposedly from that site? Remember, anyone can create any message they want, so if all we do is verify that the remote site sent us the right password, all we know is that particular message is authentic. We don't want to have to include the password on every message we send, just as we don't want to use PK to encrypt every message we send. We will use both authentication techniques to establish initial authenticity, then use something else to tie that initial authenticity to subsequent interactions. Let's move right along to SSL/TLS to talk about how we do that, so we don't need to keep promising you that we'll get to it.

SSL/TLS

We saw in an earlier chapter that a standard method of communicating between processes in modern systems is the socket. That's equally true when the processes are on different machines. So a natural way to add cryptographic protection to communications crossing unprotected networks is to add cryptographic features to sockets. That's precisely what SSL (the Secure Socket Layer) was designed to do, many years ago. Unfortunately, SSL did not get it quite right. That's because it's pretty damned hard to get it right, not because the people who designed and built it were careless. They learned from their mistakes and created a new version of encrypted sockets called Transport Layer Security (TLS)[3]. You will frequently hear people talk about using SSL. They are usually treating it as a shorthand for SSL/TLS. SSL, formally, is insecure and should never be used for anything. Use TLS. The only exception is that some very old devices might run software that doesn't support TLS. In that case, it's better to use SSL than nothing. We'll adopt the same shorthand as others from here on, since it's ubiquitous.

The concept behind SSL is simple: move encrypted data through an ordinary socket. You set up a socket, set up a special structure to perform whatever cryptography you want, and hook the output of that structure to the input of the socket. You reverse the process on the other end. What's simple in concept is rather laborious in execution, with a number of steps required to achieve the desired result. There are further complications due to the general nature of SSL. The technology is designed to support a variety of cryptographic operations and many different ciphers, as well as multiple methods to perform key exchange and authentication between the sender and receiver.

The process of adding SSL to your program is intricate, requiring the use of particular libraries and a sequence of calls into those libraries to set up a correct SSL connection. We will not go through those operations step by step here, but you will need to learn about them to make proper use of SSL. Their purpose is, for the most part, to allow a wide range of generality both in the cryptographic options SSL supports and the ways you use those options in your program. For example, these setup calls would allow you to create one set of SSL connections using AES and another using Triple DES, if that's what you needed to do.

One common requirement for setting up an SSL connection that we will go through in a bit more detail is how to securely distribute whatever cryptographic key you will use for the connection you are setting up. Best cryptographic practice calls for you to use a brand new key to encrypt the bulk of your data for each connection you set up. You will use public/private keys for authentication many times, but as we discussed earlier, you need to use symmetric cryptography to encrypt the data once you have authenticated your

[3] Actually, even the first couple of versions of TLS didn't get it quite right. As of 2018, the current version of TLS is 1.2, and that's probably what you should use. TLS 1.3 is just around the corner, closing some vulnerabilities that TLS 1.2 is subject to, so at some point you'll need to go to TLS 1.3. This history should further reinforce the lesson of how hard it is to use cryptography properly, which in turn should motivate you to foreswear ever trying to roll your own crypto.

partner, and you want a fresh key for that. Even if you are running multiple simultaneous SSL connections with the same partner, you want a different symmetric key for each connection.

So what do you need to do to set up a new SSL connection? We won't go through all of the gory details, but, in essence, SSL needs to bootstrap a secure connection based (usually) on asymmetric cryptography when no usable symmetric key exists. (You'll hear "usually" and "normally" and "by default" a lot in SSL discussions, because of SSL's ability to support a very wide range of options, most of which are ordinarily not what you want to do.) The very first step is to start a negotiation between the client and the server. Each party might only be able to handle particular ciphers, secure hashes, key distribution strategies, or authentication schemes, based on what version of SSL they have installed, how it's configured, and how the programs that set up the SSL connection on each side were written. In the most common cases, the negotiation will end in both sides finding some acceptable set of ciphers and techniques that hit a balance between security and performance. For example, they might use RSA with 2048 bit keys for asymmetric cryptography, some form of a Diffie-Hellman key exchange mechanism (see the Aside on this mechanism) to establish a new symmetric key, SHA-3 to generate secure hashes for integrity, and AES with 256 bit keys for bulk encryption. A modern installation of SSL might support 50 or more different combinations of these options.

In some cases, it may be important for you to specify which of these many combinations are acceptable for your system, but often most of them will do, in which case you can let SSL figure out which to use for each connection without worrying about it yourself. The negotiation will happen invisibly and SSL will get on with its main business: authenticating at least the server (optionally the client), creating and distributing a new symmetric key, and running the communication through the chosen cipher using that key.

We can use Diffie-Hellman key exchange to create the key (and SSL frequently does), but we need to be sure who we are sharing that key with. SSL offers a number of possibilities for doing so, which include skipping authentication and hoping for the best (not generally a good option, but still supported as of TLS version 1.2). The most common method is for the client to obtain a certificate containing the server's public key (typically by having the server send it to the client) and to use the public key in that certificate to verify the authenticity of the server's messages. It is possible for the client to obtain the certificate through some other means, though less common. Note that having the server send the certificate is every bit as secure (or insecure) as having the client obtain the certificate through other means. Certificate security is not based on the method used to transport it, but on the cryptography embedded in the certificate.

With the certificate in hand (however the client got it), the Diffie-Hellman key exchange can now proceed in an authenticated fashion. The server will sign its Diffie-Hellman messages with its private key, which will allow the client to determine that its partner in this key exchange is the correct server. Typically, the client does not provide (or even have) its own certificate, so it cannot sign its Diffie-Hellman messages. This implies that when SSL's Diffie-Hellman key exchange completes, typically the client is pretty sure who the server is, but the server has no clue about the client's identity. (Again, this need not be the case for all uses of SSL. SSL includes connection creation options where both parties know each other's public key and the key exchange is authenticated on both sides. Those options are simply not the most commonly used ones, and particularly are not the ones typically used to secure web browsing.)

ASIDE: DIFFIE-HELLMAN KEY EXCHANGE

What if you want to share a secret key between two parties, but they can only communicate over an insecure channel, where eavesdroppers can hear anything they say? You might think this is an impossible problem to solve, but you'd be wrong. Two extremely smart cryptographers named Whitfield Diffie and Martin Hellman solved this problem years ago, and their solution is in common use. It's called Diffie-Hellman key exchange.

Here's how it works. Let's say Alice and Bob want to share a secret key, but currently don't share anything, other than the ability to send each other messages. First, they agree on two numbers, n (a large prime number) and g (which is primitive $mod\ n$). They can use the insecure channel to do this, since n and g don't need to be secret. Alice chooses a large random integer, say x, calculates $X = g^x\ mod\ n$, and sends X to Bob. Bob independently chooses a large random integer, say y, calculates $Y= g^y\ mod\ n$, and sends Y to Alice. The eavesdroppers can hear X and Y, but since Alice and Bob didn't send x or y, the eavesdroppers don't know those values. It's important that Alice and Bob keep x and y secret.

Alice now computes $k=Y^x\ mod\ n$, and Bob computes $k=X^y\ mod\ n$. Alice and Bob get the same value k from these computations. Why? Well, $Y^x\ mod\ n = (g^y\ mod\ n)^x\ mod\ n$, which in turn equals $g^{yx}\ mod\ n$. $X^y\ mod\ n = (g^x\ mod\ n)^y\ mod\ n = g^{xy}\ mod\ n$, which is the same thing Alice got. Nothing magic there, that's just how exponentiation and modulus arithmetic work. Ah, the glory of mathematics! So k is the same in both calculations and is known to both Alice and Bob.

What about those eavesdroppers? They know g, n, X, and Y, but not x or y. They can compute $k'=X^Y\ mod\ n$, but that is not equal to the k Alice and Bob calculated. They do have approaches to derive x or y, which would give them enough information to obtain k, but those approaches requires them either to perform a calculation for every possible value of n (which is why you want n to be very large) or to compute a discrete logarithm. Computing a discrete logarithm is a solvable problem, but it's computationally infeasible for large numbers. So if the prime n is large (and meets other properties), the eavesdroppers are out of luck. How large? 600 digit primes should be good enough.

Neat, no? But there is a fly in the ointment, when one considers using Diffie-Hellman over a network. It ensures that you securely share a key with someone, but gives you no assurance of who you're sharing the key with. Maybe Alice is sharing the key with Bob, as she thinks and hopes, but maybe she's sharing it with Mallory, who posed as Bob and injected his own Y. Since we usually care who we're in secure communication with, we typically augment Diffie-Hellman with an authentication mechanism to provide the assurance of our partner's identity.

Recalling our discussion earlier in this chapter, it actually isn't a problem for the server to be unsure about the client's identity at this point, in many cases. As we stated earlier, the client will probably want to use a password to authenticate itself, not a public key extracted from a certificate. As long as the server doesn't permit the client to do anything requiring trust before the server obtains and checks the client's password, the server probably doesn't care who the client is, anyway. Many servers offer some services to anonymous clients (such as providing them with publicly available information), so as long as they can get a password from the client before proceeding to more sensitive subjects, there is no security problem. The server can ask the client for a user ID and password later, at any point after the SSL connection is established. Since creating the SSL connection sets up a symmetric key, the exchange of ID and password can be protected with that key.

A final word about SSL/TLS: it's a protocol, not a software package. There are multiple different software packages that implement this protocol. Ideally, if they all implement the protocol properly, they all interact correctly. However, they use different code to implement the protocol. As a result, software flaws in one implementation of SSL/TLS might not be present in other implementations. For example, the Heartbleed attack was based on implementation details of OpenSSL [H14], but was not present in other implementations, such as the version of SSL/TLS found in Microsoft's Windows operating system. It is also possible that the current protocol definition of SSL/TLS contains protocol flaws that would be present in any compliant implementation. If you hear of a security problem involving SSL, determine whether it is a protocol flaw or an implementation flaw before taking further action. If it's an implementation flaw, and you use a different implementation, you might not need to take any action in response.

Other Authentication Approaches

While passwords and public keys are the most common ways to authenticate a remote user or machines, there are other options.

One such option is used all the time. After you have authenticated yourself to a web site by providing a password, as we described above, the web site will continue to assume that the authentication is valid. It won't ask for your password every time you click a link or perform some other interaction with it. (And a good thing, too. Imagine how much of a pain it would be if you had to provide your password every time you wanted to do anything.) If your session is encrypted at this point, it could regard your proper use of the cryptography as a form of authentication; but you might even be able to quit your web browser, start it up again, navigate back to that web site, and still be treated as an authenticated user, without a new request for your password. At that point, you're no longer using the same cryptography you used before, since you would have established a new session and set up a new cryptographic key. How did your partner authenticate that you were the one receiving the new key?

In such cases, the site you are working with has chosen to make a security tradeoff. It verified your identity at some time in the past using your password and then relies on another method to authenticate you in the future. A common method is to use *web cookies*. Web cookies are pieces of data that a web site sends to a client with the intention that the client store that data and send it back again whenever the client next communicates with the server. Web cookies are built into most browsers and are handled invisibly, without any user intervention. With proper use of cryptography, a server that has verified the password of a client can create a web cookie that securely stores the client's identity. When the client communicates with the server again, the web browser automatically includes the cookie in the request, which allows the server to verify the client's identity without asking for his password again[4].

If you spend a few minutes thinking about this authentication approach, you might come up with some possible security problems associated with it. The people designing this technology have dealt with some of these problems, like preventing an eavesdropper from simply using a cookie he copied as it went across the network. However, there are other security problems (like someone other than the legitimate user using the computer that was running the web browser and storing the cookie) that can't be solved with these kinds of cookies, but could have been solved if you required the user to provide the password every time. When you build your own system, you will need to think about these sorts of security tradeoffs yourself. Is it better to make life simpler for your user by not asking for her password except when absolutely necessary, or is it better to provide your user with improved security by frequently requiring proof of her identity? The point isn't that there is one correct answer to this question, but that you need to think about such questions in the design of your system.

There are other authentication options. One example is a challenge/response protocol. The remote machine sends you a challenge, typically in the form of a number. To authenticate yourself, you must perform some operation on the challenge that produces a response. This should be an operation that only the authentic party can perform, so it probably relies on the use of a secret that party knows, but no one else does. The secret is applied to the challenge, producing the response, which is sent to the server. The server must be able to verify that the proper response has been provided. A different challenge is sent every time, requiring a different response, so attackers gain no advantage by listening to and copying down old challenges and responses. Thus, the challenges and responses need not be encrypted. Challenge/response systems usually perform some kind of cryptographic operation, perhaps a hashing operation, on the challenge plus the secret to produce the response. Such operations are better performed by machines than people, so either your computer calculates the response for you or you have a special hardware token that takes care of it. Either way, a challenge/response system requires pre-arrangement between the challenging machine and the machine trying to authenticate itself. The hardware token or the data secret must have been set up and distributed before the challenge is issued.

[4] You might remember from the chapter on access control that we promised to discuss protecting capabilities in a network context using cryptography. That, in essence, is what these web cookies are. After a user authenticates itself with another mechanism, the remote system creates a cryptographic capability for that user that no one else could create, generally using a key known only to that system. That capability/cookie can now be passed back to the other party and used for future authorization operations. The same basic approach is used in a lot of other distributed systems.

Another authentication option is to use an authentication server. In essence, you talk to a server that you trust and that trusts you. The party you wish to authenticate to must also trust the server. The authentication server vouches for your identity in some secure form, usually involving cryptography. The party who needs to authenticate you is able to check the secure information provided by the authentication server and thus determine that the server verified your identity. Since the party you wish to communicate with trusts the authentication server, it now trusts that you are who you claim to be. In a vague sense, certificates and CAs are an offline version of such authentication servers. There are more active online versions that involve network interactions of various sorts between the two machines wishing to communicate and one or more authentication servers. Online versions are more responsive to changes in security conditions than offline versions like CAs. An old certificate that should not be honored is hard to get rid of, but an online authentication server can invalidate authentication for a compromised party instantly and apply the changes immediately. The details of such systems can be quite complex, so we will not discuss them in depth. Kerberos is one example of such an online authentication server [NT95].

Some Higher Level Tools

In some cases, we can achieve desirable security effects by working at a higher level. HTTPS (the cryptographically protected version of the HTTP protocol) and SSH (a competitor to SSL most often used to set up secure sessions with remote computers) are two good examples.

HTTPS

HTTP, the protocol that supports the World Wide Web, does not have its own security features. Nowadays, though, much sensitive and valuable information is moved over the web, so sending it all unprotected over the network is clearly a bad idea. Rather than come up with a fresh implementation of security for HTTP, however, HTTPS takes the existing HTTP definition and connects it to SSL/TLS. SSL takes care of establishing a secure connection, including authenticating the web server using the certificate approach discussed earlier and establishing a new symmetric encryption key known only to the client and server. Once the SSL connection is established, all subsequent interactions between the client and server use the secured connection. To a large extent, HTTPS is simply HTTP passed through an SSL connection.

That does not devalue the importance of HTTPS, however. In fact, it is a useful object lesson. Rather than spend years in development and face the possibility of the same kinds of security flaws that other developers of security protocols inevitably find, HTTPS makes direct use of a high quality transport security tool, thus replacing an insecure transport with a highly secure transport at very little development cost.

HTTPS obviously depends heavily on authentication, since we want to be sure we aren’t communicating with malicious web sites. HTTPS uses certificates for that purpose. Since HTTPS is intended primarily for use in web browsers, the certificates in question are gathered and managed by the browser. Modern browsers come configured with the public keys of many certificate signing authorities (CAs, as we mentioned earlier).

Certificates for web sites are checked against these signing authorities to determine if the certificate is real or bogus. Remember, however, what a certificate actually tells you, assuming it checks out: that at some moment in time the signing authority thoughts it was a good idea to vouch that a particular public key belongs to a particular party. There is no implication that the party is good or evil, that the matching private key is still secret, or even that the certificate signing authority itself is secure and uncompromised, either when it created the certificate or at the moment you check it. There have been real world problems with web certificates for all these reasons. Remember also that HTTPS only vouches for authenticity. An authenticated web site using HTTPS can still launch an attack on your client. An authenticated attack, but that won't be much consolation if it succeeds.

Not all web browsers always supported HTTPS, typically because they didn't have SSL installed or configured. In those cases, a web site using HTTPS only would not be able to interact with the client, since the client couldn't set up its end of the SSL socket. The standard solution for web servers was to fall back on HTTP when a client claimed it was unable to use HTTPS. When the server did so, no security would be applied, just as if the server wasn't running HTTPS at all. As ability to support HTTPS in browsers and client machines has become more common, there has been a push towards servers insisting on HTTPS, and refusing to talk to clients who can't or won't speak HTTPS. This approach is called HSTS (HTTP Strict Transport Security). HSTS is an option for a web site. If the web site decides it will support HSTS, all interactions with it will be cryptographically secured for any client. Clients who can't or won't accept HTTPS will not be allowed to interact with such a web site. HSTS is used by a number of major web sites, including Google, but is far from ubiquitous as of 2018.

While HTTPS is primarily intended to help secure web browsing, it is sometimes used to secure other kinds of communications. Some developers have leveraged HTTP for purposes rather different than standard web browsing, and, for them, using HTTPS to secure their communications is both natural and cheap. However, you can only use HTTPS to secure your system if you commit to using HTTP as your application protocol, and HTTP was intended primarily to support a human-based activity. HTTP messages, for example, are typically encoded in ASCII and include substantial headers designed to support web browsing needs. You may be able to achieve far greater efficiency of your application by using SSL, rather than HTTPS. Or you can use SSH.

SSH

SSH stands for "Secure Shell," which accurately describes the original purpose of the program. SSH is available on Linux and other Unix systems, and to some extent on Windows systems. SSH was envisioned as a secure remote shell, but it has been developed into a more general tool for allowing secure interactions between computers. Most commonly this shell is used for command line interfaces, but SSH can support many other forms of secure remote interactions. For example, it can be used to protect remote X Windows sessions. Generally, TCP ports can be forwarded through SSH, providing a powerful method to protect interactions between remote systems.

SSH addresses many of the same problems seen by SSL, often in similar ways. Remote users must be authenticated, shared encryption keys must be established, integrity must be checked, and so on. SSH typically relies on public key cryptography and certificates to authenticate remote servers. Clients frequently do not have their own certificates and private keys, in which case providing a user ID and password is permitted. SSH supports other options for authentication not based on certificates or passwords, such as the use of authentication servers (such as Kerberos). Various ciphers (both for authentication and for symmetric encryption) are supported, and some form of negotiation is required between the client and the server to choose a suitable set.

A typical use of SSH provides a good example of a common general kind of network security vulnerability called a *man-in-the-middle attack*. This kind of attack occurs when two parties think they are communicating directly, but actually are communicating through a malicious third party without knowing it. That third party sees all of the messages passed between them, and can alter such messages or inject new messages without their knowledge[5].

Well-designed network security tools are immune to man-in-the-middle attacks of many types, but even a good tool like SSH can sometimes be subject to them. If you use SSH much, you might have encountered an example yourself. When you first use SSH to log into a remote machine you've never logged into before, you probably don't have the public key associated with that remote machine. How do you get it? Often, not through a certificate or any other secure means, but simply by asking the remote site to send it to you. Then you have its public key and away you go, securely authenticating that machine and setting up encrypted communications. But what if there's a man in the middle when you first attempt to log into the remote machine? In that case, when the remote machine sends you its public key, the man in the middle can discard the message containing the correct public key and substitute one containing his own public key. Now you think you have the public key for the remote server, but you actually have the public key of the man in the middle. That means the man in the middle can pose as the remote server and you'll never be the wiser. The folks who designed SSH were well aware of this problem, and if you ever do use SSH this way, up will pop a message warning you of the danger and asking if you want to go ahead despite the risk. Folk wisdom suggests that everyone always says "yes, go ahead" when they get this message, including network security professionals. For that matter, folk wisdom suggests that all messages warning a user of the possibility of insecure actions are always ignored, which should suggest to you just how much security benefit will arise from adding such confirmation messages to your system.

SSH is not built on SSL, but is a separate implementation. As a result, the two approaches each have their own bugs, features, and uses. A security flaw found in SSH will not necessarily have any impact on SSL, and vice versa.

[5] Think back to our aside on Diffie-Hellman key exchange and the fly in the ointment. That's a perfect case for a man-in-the-middle attack, since an attacker can perhaps exchange a key with one correct party, rather than the two correct parties exchanging a key, without being detected.

Summary

Distributed systems are critical to modern computing, but are difficult to secure. The cornerstone of providing distributed system security tends to be ensuring that the insecure network connecting system components does not introduce new security problems. Messages sent between the components are encrypted and authenticated, protecting their privacy and integrity, and offering exclusive access to the distributed service to the intended users. Standard tools like SSL/TLS and public keys distributed through X.509 certificates are used to provide these security services. Passwords are often used to authenticate remote human users.

Symmetric cryptography is used for transport of most data, since it is cheaper than asymmetric cryptography. Often, symmetric keys are not shared by system participants before the communication starts, so the first step in the protocol is typically exchanging a symmetric key. As discussed in previous chapters, key secrecy is critical in proper use of cryptography, so care is required in the key distribution process. Diffie-Hellman key exchange is commonly used, but it still requires authentication to ensure that only the intended participants know the key.

As mentioned in earlier chapters, building your own cryptographic solutions is challenging and often leads to security failures. A variety of tools, including SSL/TLS, SSH, and HTTPS, have already tackled many of the challenging problems and made good progress in overcoming them. These tools can be used to build other systems, avoiding many of the pitfalls of building cryptography from scratch. However, proper use of even the best security tools depends on an understanding of the tool's purpose and limitations, so developing deeper knowledge of the way such tools can be integrated into one's system is vital to using them to their best advantage.

Remember that these tools only make limited security guarantees. They do not provide the same assurance that an operating system gets when it performs actions locally on hardware under its direct control. Thus, even when using good authentication and encryption tools properly, a system designer is well advised to think carefully about the implications of performing actions requested by a remote site, or providing sensitive information to that site. What happens beyond the boundary of the machine the OS controls is always uncertain and thus risky.

References

[H14] The Heartbleed Bug
http://heartbleed.com/
A web page providing a wealth of detail on this particular vulnerability in the OpenSSL implementation of the SSL/TLS protocol.

[I12] "Information technology – Open Systems Interconnection – The Directory: Public-key and Attribute Certificate Frameworks"
ITU-T, 2012
The ITU-T document describing the format and use of an X.509 certificate. Not recommended for light bedtime reading, but here's where it's all defined.

[NT94] “Kerberos: An authentication service for computer networks”
B. Clifford Neuman and Theodore Ts'o
IEEE Communications Magazine, Volume 32, No. 9, 1994
An early paper on Kerberos by its main developers. There have been new versions of the system and many enhancements and bug fixes, but this paper is still a good discussion of the intricacies of the system.

[P16] The International PGP Home Page
http://www.pgpi.org/, 2016.
A page that links to lots of useful stuff related to PGP, including downloads of free versions of the software, documentation, and discussion of issues related to it.