Distributed Systems: Paul Krzyzanowski

Distributed Systems Security

Cryptographic systems

Paul Krzyzanowski
April 20, 2021

> ***Goal:** Use symmetric cryptography, public key cryptography, random numbers, and hash functions to enable exchange encryption keys, provide secure communication, and ensure message.*

Security encompasses many things, including authenticating users and data, hiding data contents while they are at rest (e.g., stored in files) or in motion (moving over a network), destruction of data, masquerading as another server or user, providing false data (e.g., the wrong IP address for a DNS query), and physical premises security. We will focus only on a few topics here.

Security in distributed systems introduces two specific concerns that centralized systems do not have. The first is the use of a network where contents may be seen by other, possibly malicious, parties. The second is the use of servers. Because clients interact with services (applications) running on a server, the application rather than the operating system is responsible for authenticating the client and controlling access to services. Moreover, physical access to the system and the security controls configured for the operating system may be unknown to the client.

Computer security is about keeping systems, programs, and data secure. It addresses three broad areas: **confidentiality**, **integrity**, and **availability**. Together, these are referred to as the **CIA Triad**.

Confidentiality

Confidentiality deals with keeping resources and data hidden from, or inaccessible to, unauthorized individuales. It is addressed by access control mechanisms in operating systems or application software. If the data may be accesseed through the file system or visible over a network, confidentiality is addressed by encrypting the data. An application's decisions on whether data should be made accessible to a user depends on identification and authentication of the user or service.

Integrity

Integrity deals with the trustworthiness of the data or the resources. Integrity mechanisms are responsible for preventing unauthorized changes to data or detecting that changes have been made. Integrity mechanisms are used to validate the identity of users, systems, and services through authentication algorithms.

Availability

Availability is about having access to the data or computing services. It's the property that a system is accessible and properly functioning. Accessibility includes fault tolerance, recovery, and restoration.

Security is a systems issue and pervades the design of an entire system. It's not a module or add-on component. Security spans the hardware, firmware, and perating system up through the applicationsoftware. It includes all the networking and even the users. Security also includes the processes, procedures, and policies that are defined and implemented to ensure proper access, availability, and recovery.

The operating system

Traditionally, the operating system took on much of the responsibility for providing security to the system. The operating system is the gatekeeper for any process to access any resource; it validates and authorizes all access. The operating system handles user authentication via the login process and manages access control via file permissions and mechanisms that restrict access to system calls. The operating system's memory management system and process scheduler control acces to memory and the processor, respectively.

However, an operating system can only control the resoources that it owns. It has no control of what other systems are doing. With network-based services, it's very likely that the remote service you are accessing runs on a system on which you don't even have an account. When a process communicates with remote services, the data will flow on a non-secured network where bad actors may attempt to capture data or modify it.

Risks introduced by the Internet

> *"The internet was designed to be open, transparent, and interoperable. Security and identity management were secondary objectives in system design. This lower emphasis on security in the internet's initial design not only gives attackers a built-in advantage. It can also make intrusions difficult to attribute, especially in real time. This structural property of the current architecture of cyberspace means that we cannot rely on the threat of retaliation alone to deter potential attackers. Some adversaries might gamble that they could attack us and escape detection." — William J. Lynn III, Deputy Defense Secretary, 2010*

The Internet provides us with a powerful mechanism for communicating with practically any other computer in the world. It also provides a way for attackers to target systems from practically any other computer in the world. Even though the Internet was a direct descendent of the ARPANET, a project funded by the U.S. Defense Advanced Research Projects Agency (DARPA), its design priority was to interconnect multiple disparate networks and create a decentralized architecture. Security was not a design consideration of the Internet.

The network itself is "dumb"; it is responsible for routing packets from one system to another. As we'll see later, even this is problematic since no one entity owns the Internet or is responsible for defining the routing rules. The intelligence of the Internet is at the endpoints. Protocols such as TCP run on endpoint computers to provide the in-order reliable delivery of messages. Any system can join the Internet and start sending and receiving messages. Systems can even offer routing services, allowing new networks to extend the

Internet. This provides opportunities for redirecting messages and impersonating other systems.

The Internet introduces several security risks:

- **Action at a distance**. Attackers do not need to be physically present by the computer to attack it. They can also be in a different state or even a different country.
- **Asymmetric power**. Actors can project or harness significant force for attack. This differs dramatically from the physical world where, for example, a small nation would not risk attacking a large one that could easily overpower it. Offense can be much more effective than defense. The person or people who started Conflicker were able to harness the power of millions of computers worldwide that they could use to launch attacks or crack passwords. A **Distributed Denial of Service** (**DDoS**) attack is an example of the use of asymmetric force. A company has only so many servers on the network. If an attacker can harness a large number of computers to send requests to the company's servers, the servers will get overloaded and be unable to process legitimate requests. With a DDoS attack, the attacker will assemble a **botnet** by attacking a large set of computers and infectious them with malicious software (**malware**). These computers, owned by innocent people, will periodically contact a **command & control server** to receive directions on what attack to carry out.
- **Actors can be anonymous**. Actors can be anonymous. No one knows with any certainty who ran some of the biggest attacks that the world has ever experienced. Trust becomes a big challenge. When you're. Interacting with your bank or you really communicating with your bank? How do you know and how does the bank know that it's really you?
- **There are no borders**. There are no checkpoints. With very few exceptions, countries have many ISPs and many points of connection to other countries. You can neither shut down nor filter all the data.
- **Indistinctive data**. It is difficult to distinguish valid data from malicious attacks. A malicious user logging in may look the same as a legitimate user. Code is a bunch of bits and it is not always possible to tell whether it is harmful until it is executed. The network will just route packets to their desired destination.

Automation enables attacks on a large scale. It is easy to cast a wide net via scripting. You can try thousands or millions of potential targets and see if any of them have security weaknesses that can you exploit. Attacks that have even minuscule rates of return or small chances of success are now profitable. This includes email scams, transferring fractional cents from bank accounts, and attempts to exploit known weaknesses.

Cryptography

Cryptography on its own is not a panacea. Its use will not ensure that a system is secure. However, it is an important component in building secure distributed systems.

It is used to implement mechanisms for:

- **Confidentiality**: encrypting data so that others cannot read the contents of a message (or file).
- **Authentication**: proving the origin of a message (or the entity sending that message).
- **Integrity**: validating that the message has not been modified, either maliciously or accidentally.

- **Nonrepudiation**: ensuring that senders cannot falsely deny that they authored a message.

A good cryptography algorithm (**cryptosystem**) has three properties:

1. Ciphertext should look random. There should be no discernable statistical patterns in it and no hints on what content is in the original message.

2. There is no way to extract the original plaintext or key from the ciphertext. The only way to find the message and the key should be through a brute-force attack, which is an exhaustive search through all possible keys.

3. Keys should be long enough to make a brute-force attack not feasible. A "short" AES key is 128 bits. With the fastest supercomputer, it will take a billion billion years to crack that key. If you harness a billion computers, then you can crack it in a billion years. Each additional bit of a key doubles the number of possible keys (a key of length *n* has 2^n~ possible keys). The more common AES key size is 256 bits, which will take 2^{128}, or 3.4×10^{38} times as long as cracking a 128-bit key.

Confidentiality: encryption

Confidentiality is the classic application of cryptography: obscuring the contents of a message so that it looks like a random bunch of bits that can only be decoded by someone with the proper knowledge. To do this, we start with our **plaintext** message, *P* (the term *plaintext* refers to unencrypted content; it does not need to be text). By applying an **encryption** algorithm, or **cipher**, to it, we convert the plaintext to ciphertext, $C=E(P)$. We can **decrypt** this message to recover the plaintext: $P=D(C)$.

There are two classes of ciphers: symmetric and public key.

Symmetric cryptography

A **symmetric** cipher uses the same to decrypt a message as was used to encrypt it: $C=E_K(P)$ and $P=D_K(C)$. Examples of popular symmetric encryption algorithms include AES (Advanced Encryption Standard), DES (Data Encryption Standard), and Blowfish.

Public key cryptography

A **public key** cipher uses two mathematically-related keys, called a **key pair**. Knowing one key does not enable you to derive the other key of the pair. One of these keys is called the **private key** because it is never shared. The other key is called the **public key** because it is freely shared. A message encrypted with one of the keys can only be decrypted with the other key of the pair. For a pair of keys (a, A), where a is the private key and A is the public key, if $C=E_a(P)$ then $P=D_A(C)$.

A message encrypted with your private key can be decrypted only with your public key. Anyone can do the decryption but you are the only one who could perform the encryption. This becomes the basis for authentication.

Conversely, if $C=E_A(P)$ then $P=D_a(C)$. A message encrypted with your public key can be decrypted only with your private key. Anyone can now perform the encryption but you are the only one can decrypt the message. This becomes the basis for confidentiality and secure communication. Examples of popular public key encryption algorithms include RSA and ECC (Elliptic Curve Cryptography).

Communication with symmetric cryptography

Communicating with symmetric key encryption requires both parties to share the same secret key. Alice encrypts a message for Bob using that shared secret key and Bob decrypts it with the same key.

Communication with public key cryptography

Public key cryptography does not require the parties to share a secret key. If Alice and Bob have each other's public keys, which are not secret, they can securely send data to each other. Alice will encrypt a message with Bob's public key. Anybody can do this but only Bob will be able to decrypt the data using his private key. Similarly, Bob can encrypt a message with Alice's private key, which only Alice can decypt with her private key.

Key distribution

Symmetric encryption algorithms are the dominant means of encrypting files and large streams of data. They are much faster than public key algorithms and key generation is far easier: a key is just a random set of bits rather than two huge numbers with specific properties. The biggest problem with symmetric cryptography is **key distribution**: ensuring that both sides get the key in a way that no eavesdropper can observe it. For instance, you cannot just send it as a network message containing the key. There are several techniques to handle the problem of key distribution.

1. Manually, by using **pre-shared keys** (**PSK**). This requires setting up the keys ahead of time, such as registering a password or PIN among all the parties that will communicate.

2. Using a **trusted third party**. A trusted third party is a trusted server that knows everyone's keys. If two systems, let's call them Alice and Bob, want to communicate, the third party will help both of them get the key they need. To communicate, they will use a **session key**. This is the name for a throw-away key that will be used for one communication session. One way of sharing it is for Alice to create it (it's just a random number), encrypt it with her secret key, and send it to the trusted third party, which we will name Trent. Since Trent has all the keys, he can decrypt the message with Alice's secret key to extract the session key. He can then encrypt it for Bob using Bob's secret key and send it to Bob.

Alternatively, Alice can ask Trent to create a session key that both Alice and Bob will share. Trent will generate the key, encrypt it for Alice, and send it to Alice. He will take the same key, encrypt it for Bob, and send it to Bob. Now they both share a key.

1. Using the **Diffie-Hellman key exchange** algorithm. Diiffie-Hellman is a key distribution algorithm, not en encryption algorithm. It uses a **one-way function**, which is a mathematical function where there is no known way of computing the inverse function to do this. Unfortunately, the algorithm uses the terms *public key* and *private key* even though it is not an encryption algorithm; the keys are just numbers, one of which is kept private. Two parties can apply the one-way function to generate a **common key** that is derived from one public key and one private key. Alice generates a common key that is *f(Alice_private_key, Bob_public_key)* and Bob generates a common

key that is *f(Bob_private_key, Alice_public_key)*. The magic of the mathematics in Diffie-Hellman is that both common keys will be identical and can be used as a symmetric encryption key.

2. Avoid symmetric cryptography and just use **public key cryptography**. Life becomes easy. If Alice wants to send a message to Bob, she simply encrypts it with Bob's public key. Only Bob will be able to decrypt it using his private key. If Bob wants to send a message to Alice, he will encrypt it with her public key. Only she will be able to decrypt it using her private key. There are several concerns with using this. First, we need to ensure that everyone has trusted copies of public keys (Alice needs to be sure she has Bob's public key rather than that of an imposter saying he's Bob). Second, we need to be mindful of the fact that public key cryptography and key generation are both much slower than symmetric cryptography. Third, public key cryptography is considered to be less secure for encrypting large amounts of data. The algorithms have been shown to be vulnerable to chosen plaintext attacks – where an attacker can get you to put certain content in your message – and it gives them a certain edge in trying to recover the key.

Hybrid cryptosystem

To get the best of both worlds – the convenience of public key cryptography and the speed of symmetric cryptgraphy – we often rely on **hybrid cryptography**: we use public key cryptography only to encrypt a symmetric session key (a small amount of data). From then on, we use symmetric cryptography and encrypt data with that session key.

In use, we often distinguish between **long-term** (**permanent**) keys versus **ephemeral** keys. Long-term keys are stored and used over and over. A user's password is an example of a long-term key. Public keys are often long-term keys. They are not changed frequently because of the hassle of generating new key pairs and distributing the updates. Ephemeral keys are thosse that are spontaneously created when needed. The use of ephemeral keys minimizes the amount of data we encrypt with the same key. This provides less data for the cryptanalyst; if a key is ever discovered, it will not result in the ability to decode a lot of data. Ephemeral keys are also important in cases where we need to generate a key spontaneously for two parties to be able to communicate. For instance, Alice and Bob can use the Diffie-Hellman algorithm to come up with a common key and use it to encrypt a random key that will be used only for that communication session. This type of key is called a **session key**.

Message Integrity & Non-repudiation

Without cryptography, we often resort to various error detecting functions to detect whether there is any data corruption. For example, ethernet uses CRC (cyclic redundancy checksums) and IP headers use 16-bit checksums. A **cryptographic hash function** is similar in purpose: a function that takes an arbitrary amount of data and generates a fixed set of bits. The hash is far longer than typical checksums – typically 256 or 512 bits – and hence is highly unlikely to result in collisions, where multiple messages result in the same hash value. A cryptographic hash of a message, *H=hash(M)* should have the following properties:

- It produces a **fixed-length output**. Regardless of the input, the output of a hash function is a fixed size. Common hash sizes are 256 or 512 bits.
- It is deterministic. A hash of the smae message will always yield the same output.
- It is a **one-way function**. Given a hash value *H*, it should be difficult to figure out what the input is. (*N.B.: in cryptography, the term "difficult" means that there are no known shortcuts other that trying every possible input*).

- It is **collision resistant**. Given a hash value *H*, it should be difficult to find another message *M'*, such that *H=hash(M')*
- The **output does not give any information about the input**. This is a property called *diffusion*, which is also important for cryptographic algorithms. Changing even a single bit in a message should, on average, change half of the bits in the resulting hash. There will be no detectable correspondence between any input data and any part of the resulting hash.
- It is efficient. We want to be able to generate these easily to validate file contents and messages.

Hash functions serve as the basis of message integrity. By providing a hash of a message along with the message, you can detect that the message has not been modified. If even a bit of the message is modified, the provided hash will no longer match the hash of this modified message.

Message authentication codes (MAC)

Just augmenting a message with its hash is not sufficient to ensure message integrity. It will allow us to detect that a message was accidentally modified but an attacker who modifies a message can easily recompute a hash function. To prevent the hash from being modified, we add a secret ingredient to the message we are hashing.

A **message authentication code** (**MAC**) is a hash of a message that is combined with a symmetric (secret) key. In the simplest form, one can concatenate the key with the message and hash the result. Various MAC functions implement different forms of this. For example, the HMAC function hashes a key exclusive-ored with a bit pattern that is concatenated with the same key exclusive-ored with another bit pattern and then concatenation with the message.

If Alice and Bob share a key, *K*, Alice can send Bob a message along with a hash created from *M* and *K*. Bob can hash *M* and *K* and compare the result with Alice's MAC. If an intruder modifies the message, she will not be able to create a valid MAC since she would not know what the key is.

Digital signatures

A **digital signature** is similar to a MAC but the hash is encrypted with the sender's private key. Here, Alice will send Bob a message along with a hash that is encrypted with her private key. Bob can validate the message and signature by decrypting the signature using Alice's public key and comparing the result with a hash of the message. A digital signature provides **non-repudiation**: Alice cannot claim that she did not create the message since only Alice knows her private key. Nobody but Alice could have created the signature.

We can combine covert messaging together with message integrity to ensure that a recipient can detect if a message has been modified. One way of doing this is:

1. Alice creates a session key, S (a random number).
2. She encrypts the session key with Bob's public key, *B*: $E_B(S)$ and sends it to Bob.
3. Bob decrypts the session key using his private key, *b*: $S = D_b(E_B(S))$. He now knows the session key.
4. Alice encrypts the message with the session key: $E_S(M)$.

5. Alice creates a signature by encrypting the hash of the message with her private key, *a*: $E_a(H(M))$.
6. She sends both the encrypted message and signature to Bob.
7. Bob decrypts the message using the session key: $M = D_S(E_S(M))$.
8. Bob decrypts the signature using Alice's public key *A*: $D_A(E_a(H(M)))$_
9. If the decrypted hash matches the hash of the message, $H(M)$, Bob is convinced that the message has not been modified since Alice sent it.

Last modified April 21, 2021.