Google IT Support Specialization

Course 5: IT security

02 pelcgbybtl cryptology

Table des matières

[1 Understanding IT security threat 2](#_Toc14375854)

[2 Pelcgbybtl cryptology 2](#_Toc14375855)

[2.1 Symmetric-encryption 2](#_Toc14375856)

[2.1.1 Cryptography 2](#_Toc14375857)

[2.1.2 Symmetric-cryptography 3](#_Toc14375858)

[2.1.3 Symmetric-encryption-algorithms 4](#_Toc14375859)

[2.1.4 Rob-what-a-security-engineer-does 6](#_Toc14375860)

[2.2 Public-key-or-asymmetric-encryption 6](#_Toc14375861)

[2.2.1 Asymmetric-cryptography 6](#_Toc14375862)

[2.2.2 Asymmetric-encryption-algorithms 8](#_Toc14375863)

[2.3 Hashing 9](#_Toc14375864)

[2.3.1 Hashing 9](#_Toc14375865)

[2.3.2 Hashing-algorithms 9](#_Toc14375866)

[2.3.3 Hashing-algorithms-continued 10](#_Toc14375867)

[2.4 Cryptography-applications 11](#_Toc14375868)

[2.4.1 Public-key-infrastructure 11](#_Toc14375869)

[2.4.2 Cryptography-in-action 13](#_Toc14375870)

[2.4.3 Securing-network-traffic 14](#_Toc14375871)

[2.4.4 Cryptographic-hardware 16](#_Toc14375872)

# Understanding IT security threat

# Pelcgbybtl cryptology

## Symmetric-encryption

### Cryptography

When you were little, did you and your siblings ever communicate in a secret language around your parents? It didn't really matter what you were talking about, as long as your parents didn't know what it was. That was the fun part, right? It may have seemed like a fun game when you were younger. But for as long as humans have been around, we've created ways to keep messages secret from others. In this lesson, we'll cover how this plays out through symmetric encryption, asymmetric encryption, and hashing. We'll also go over how to describe the most common algorithms in cryptography. And learn how to choose the most appropriate cryptographic method in any given scenario. But before we dive into the nitty-gritty details of cryptography, the various types that exist in our applications, let's go over some terminology and general principles that will help you understand the details later. The topic of cryptography, or hiding messages from potential enemies, has been around for thousands of years. It's evolved tremendously with the advent of modern technology, computers and telecommunications. Encryption is the act of taking a message, called plaintext, and applying an operation to it, called a cipher. So that you receive a garbled, unreadable message as the output, called ciphertext. The reverse process, taking the garbled output and transforming it back into the readable plain text is called decryption. For example, let's look at a simple cipher, where we substitute e for o and o for y. We'll take the plaintext Hello World and feed it into our basic cipher. What do you think the resulting ciphertext will be? Hopefully, you've got Holly Wyrld. It's pretty easy to decipher the ciphertext since this is a very basic example. There are much more complex and secure ciphers or algorithms that we'll cover later in the section. A cipher is actually made up of two components, the encryption algorithm and the key. The encryption algorithm is the underlying logic or process that's used to convert the plaintext into ciphertext. These algorithms are usually very complex mathematical operations. But there are also some very basic algorithms that we can take a closer look at that don't necessarily require a PhD in math to understand. The other crucial component of a cipher is the key, which introduces something unique into your cipher. Without the key, anyone using the same algorithm would be able to decode your message, and you wouldn't actually have any secrecy. So to recap, first you pick an encryption algorithm you'd like to use to encode your message, then choose a key. Now you have a cipher, which you can run your plaintext message through and get an encrypted ciphertext out ready to be sent out into the world, safe and secure from prying eyes. Doesn't this make you feel like an international person of mystery? Just wait, given that the underlying purpose of cryptography is to protect your secrets from being read by unauthorized parties, it would make sense that at least some of the components of a cipher would need to be kept secret too, right? You can keep the argument that by keeping the algorithm secret, your messages are secured from a snooping third party, and technically you wouldn't be wrong. This general concept is referred to as, security through obscurity, which basically means, if no one knows what algorithm were using or general security practice, then we're safe from attackers. Think of hiding your house key under you doormat, as long as the burglar doesn't know that you hide the spare key under the mat, you're safe. But once that information is discovered, all security goes out the window along with your valuables. So clearly, security through obscurity isn't something that you should rely on for securing communication or systems, or for your house for that matter. This overall concept of cryptography is referred to as Kerckhoff's principle. This principle states that a cryptosystem, or a collection of algorithms for key generation and encryption and decryption operations that comprise a cryptographic service should remain secure, even if everything about the system is known except for the key. What this means is that even if your enemy knows the exact encryption algorithm you use to secure your data, they're still unable to recover the plaintext from an intercepted ciphertext. You may also hear this principle referred to as Shannon's maxim or the enemy knows the system. The implications are the same. The system should remain secure, even if your adversary knows exactly what kind of encryption systems you're employing, as long as your keys remain secure. We already defined encryption, but the overarching discipline that covers the practice of coding and hiding messages from third parties is called cryptography. The study of this practice is referred to as cryptology. The opposite of this looking for hidden messages or trying to decipher coded message is referred to as cryptanalysis. These two fields have co-evolved throughout history with new ciphers and cryptosystems being developed as previous ones were broken or found to vulnerable. One of the earliest recorded descriptions of cryptanalysis is from a ninth century Arabian mathematician. Who described a method for frequency analysis to break coded messages. Frequency analysis is the practice of studying the frequency with which letters appear in ciphertext. The premise behind this type of analysis is that in written languages, certain letters appear more frequently than others, and some letters are more commonly grouped together than others. For example, the most commonly used letters in the English language are e, t, a, and o. The most commonly seen pairs of these letters are th, er, on, and an. Some ciphers, especially classical transposition and substitution ciphers preserve the relative frequency of letters in the plaintext. And so our potentially vulnerable to this type of analysis. During World War I and World War II, cryptography and cryptanalysis played an increasingly important role. There was a shift away from linguistics and frequency analysis and a move towards more mathematical based analysis. This was due to more complex and sophisticated ciphers being developed. A major turning point in the field of cryptanalysis was during World War II when the US allies began to incorporate sophisticated mathematics to aid in breaking access encryption schemes. This also saw the first use of automation technology applied to cryptanalysis in England at Bletchley Park. The first programmable digital computer, named Colossus, was developed to aid in this effort. While early computers were applied to breaking cryptography, this opened the door for a huge leap forward and a development of even more sophisticated and complex cryptosystems. Steganography is a related practice but distinctly different from cryptography. It's the practice of hiding information from observers, but not encoding it. Think of writing a message using invisible ink. The message is in plaintext and no decoding is necessary to read the text but the text is hidden from sight. The ink is invisible and must be made visible using a mechanism known to the recipient. Modern steganographic techniques include embedding messages and even files into other files like images or videos. To a casual observer, they would just see a picture of a cute puppy. But if you feed that image into steganography software, it would extract a message hidden within the image file. What's not so secret is how fun it is to learn about all of this spy stuff, don't you think? Stick around, because next, we'll talk about specific cryptographic methods and systems.

### Symmetric-cryptography

So far, we've been talking pretty generally about cryptographic systems and focusing primarily on encryption concepts but not decryption. It makes sense that if you're sending a protected message to someone, you'd want your recipient to be able to decode the message and read it, and maybe even reply with a coded message of their own. So let's check out the first broad category of encryption algorithms and dive into more details about how it works along with some pros and cons. When we covered Kerchhoff's principle earlier, do you remember which component of the cipher is crucial to keep secret? That's right. The key must be kept private to ensure that an eavesdropper wouldn't be able to decode encrypted messages. In this scenario, we're making the assumption that the algorithm in use is what's referred to as symmetric-key algorithm. These types of encryption algorithms are called symmetric because they use the same key to encrypt and decrypt messages. Let's take a simple example of a symmetric key encryption algorithm to walk through the overall process of encrypting and decrypting a message. A substitution cipher is an encryption mechanism that replaces parts of your plaintext with ciphertext. Remember our hello world example from earlier. That's an example of substitution cipher since we're substituting some characters with different ones. In this case, the key would be the mapping of characters between plaintext and ciphertext without knowing what letters get replaced with. You wouldn't be able to easily decode the ciphertext and recover the plaintext. If you have the key or the substitution table, then you can easily reverse the process and decrypt the coded message by just performing the reverse operation. A well-known example of a substitution cipher is the Caesar cipher, which is a substitution alphabet. In this case, you're replacing characters in the alphabet with others usually by shifting or rotating the alphabet, a set of numbers or characters. The number of the offset is the key. Another popular example of this is referred to as R O T 13 or ROT-13, where the alphabet is rotated 13 places, but really ROT-13 is a Caesar cipher that uses a key of 13. Let's go back to our hello world example and walk through encoding it using our ROT-13 cipher. Our ciphertexts winds up being URYYB JBEYQ. To reverse this process and go back to the plaintext, we just performed the reverse operation by looking up the characters in the output side of the mapping table. You might notice something about the ROT-13 mapping table or the fact that we're offsetting the alphabet by 13 characters. Thirteen is exactly half of the alphabet. This results in the ROT-13 cipher being an inverse of itself. What this means is that you can recover the plaintext from ciphertext by performing the ROT-13 operation on the ciphertext. If we were to choose a different key, let's say eight, can we do the same thing? Let's check. Here's the mapping table for an offset of eight, which gives us the ciphertext of OLSSV DVYSK. If we run this through the cipher once more, we get the following output VSZZC KCFZR. That doesn't work to reverse the encryption process, does it? There are two more categories that symmetric key ciphers can be placed into. They're either block ciphers or they're stream ciphers. This relates to how the ciphers operate on the plaintext to be encrypted. A stream cipher as the name implies, takes a stream of input and encrypts the stream one character or one digit at a time, outputting one encrypted character or digit at a time. So, there's a one- to-one relationship between data in and encrypted data out. The other category of symmetric ciphers is block ciphers. The cipher takes data in, places that into a bucket or block of data that's a fixed size, then encodes that entire block as one unit. If the data to be encrypted isn't big enough to fill the block, the extra space will be padded to ensure the plaintext fits into the blocks evenly. Now generally speaking, stream ciphers are faster and less complex to implement, but they can be less secure than block ciphers. If the key generation and handling isn't done properly, if the same key is used to encrypt data two or more times, it's possible to break the cipher and to recover the plaintext. To avoid key reuse, initialization vector or IV is used. That's a bit of random data that's integrated into the encryption key and the resulting combined key is then used to encrypt the data. The idea behind this is if you have one shared master key, then generate a one-time encryption key. That encryption key is used only once by generating a new key using the master one and the IV. In order for the encrypted message to be decoded, the IV must be sent in plaintext along with the encrypted message. A good example of this can be seen when inspecting the 802.11 frame of a WEP encrypted wireless packet. The IV is included in plaintext right before the encrypted data payload. In the next video, we'll explore symmetric encryption in more detail, illustrating some of the more popular algorithms and dive into the pros and cons of using symmetric encryption.

### Symmetric-encryption-algorithms

In the last section, we covered the basics of what exactly symmetric encryption algorithms are and gave a basic example of the Caesar cipher, a type of substitution cipher. We couldn't possibly protect anything of value using the cipher though, right? There must be more complex and secure symmetric algorithms, right? Of course, there are. One of the earliest encryption standards is DES, which stands for Data Encryption Standard. DES was designed in the 1970s by IBM, with some input from the US National Security Agency. DES was adopted as an official FIPS, Federal Information Processing Standard for the US. This means that DES was adopted as a federal standard for encrypting and securing government data. DES is a symmetric block cipher that uses 64-bit key sizes and operates on blocks 64-bits in size. Though the key size is technically 64-bits in length, 8-bits are used only for parity checking, a simple form of error checking. This means that real world key length for DES is only 56-bits. A quick note about encryption key sizes since we haven't covered that yet. In symmetric encryption algorithms, the same key is used to encrypt as to decrypt, everything else being the same. The key is the unique piece that protects your data and the symmetric key must be kept secret to ensure the confidentiality of the data being protected. The key size, defined in bits, is the total number of bits or data that comprises the encryption key. So you can think of the key size as the upper limit for the total possible keys for a given encryption algorithm. Key length is super important in cryptography since it essentially defines the maximum potential strength of the system. Imagine an ideal symmetric encryption algorithm where there are no flaws or weaknesses in the algorithm itself. In this scenario, the only possible way for an adversary to break your encryption would be to attack the key instead of the algorithm. One attack method is to just guess the key and see if the message decodes correctly. This is referred to as a brute-force attack. Longer key lengths protect against this type of attack. Let's take the DES key as an example. 64-bits long minus the 8 parity bits gives us a key length of 56-bits. This means that there are a maximum of 2 to the 56th power, or 72 quadrillion possible keys. That seems like a ton of keys, and back in the 1970s, it was. But as technology advanced and computers got faster and more efficient, 64-bit keys quickly proved to be too small. What were once only theoretical attacks on a key size became reality in 1998 when the EFF, Electronic Frontier Foundation, decrypted a DES-encrypted message in only 56 hours. Because of the inherent weakness of the small key size of DES, replacement algorithms were designed and proposed. A number of new ones appeared in the 1980s and 1990s. Many kept the 64-bit block size, but used a larger key size, allowing for easier replacement of DES. In 1997, the NIST, National Institute of Standards and Technology, wanted to replace DES with a new algorithm, and in 2001, adopted AES, Advanced Encryption Standard, after an international competition. AES is also the first and only public cipher that's approved for use with top secret information by the United States National Security Agency. AES is also a symmetric block cipher similar to DES in which it replaced. But AES uses 128-bit blocks, twice the size of DES blocks, and supports key lengths of 128-bit, 192-bit, or 256-bit. Because of the large key size, brute-force attacks on AES are only theoretical right now, because the computing power required (or time required using modern technology) exceeds anything feasible today. I want to call out that these algorithms are the overall designs of the ciphers themselves. These designs then must be implemented in either software or hardware before the encryption functions can be applied and put to use. An important thing to keep in mind when considering various encryption algorithms is speed and ease of implementation. Ideally, an algorithm shouldn't be overly difficult to implement because complicated implementation can lead to errors and potential loss of security due to bugs introduced in implementation. Speed is important because sometimes data will be encrypted by running the data through the cipher multiple times. These types of cryptographic operations wind up being performed very often by devices, so the faster they can be accomplished with the minimal impact to the system, the better. This is why some platforms implement these cryptographic algorithms in hardware to accelerate the processes and remove some of the burden from the CPU. For example, modern CPUs from Intel or AMD have AES instructions built into the CPUs themselves. This allows for far greater computational speed and efficiency when working on cryptographic workloads. Let's talk briefly about what was once a wildly used and popular algorithm but has since been proven to be weak and is discouraged from use. RC4, or Rivest Cipher 4, is a symmetric stream cipher that gained widespread adoption because of its simplicity and speed. RC4 supports key sizes from 40-bits to 2,048-bits. So the weakness of RC4 aren't due to brute-force attacks, but the cipher itself has inherent weaknesses and vulnerabilities that aren't only theoretically possible, there are lots of examples showing RC4 being broken. A recent example of RC4 being broken is the RC4 NOMORE attack. This attack was able to recover an authentication cookie from a TLS-encrypted connection in just 52 hours. As this is an attack on the RC4 cipher itself, any protocol that uses this cipher is potentially vulnerable to the attack. Even so, RC4 was used in a bunch of popular encryption protocols, like WEP for wireless encryption, and WPA, the successor to WEP. It was also supported in SSL and TLS until 2015 when RC4 was dropped in all versions of TLS because of inherent weaknesses. For this reason, most major web browsers have dropped support for RC4 entirely, along with all versions of SSL, and use TLS instead. The preferred secure configuration is TLS 1.2 with AES GCM, a specific mode of operation for the AES block cipher that essentially turns it into a stream cipher. GCM, or Galois/Counter Mode, works by taking randomized seed value, incrementing this and encrypting the value, creating sequentially numbered blocks of ciphertexts. The ciphertexts are then incorporated into the plain text to be encrypted. GCM is super popular due to its security being based on AES encryption, along with its performance, and the fact that it can be run in parallel with great efficiency. You can read more about the RC4 NOMORE attack in the next reading. So now that we have covered symmetric encryption and some examples of symmetric encryption algorithms, what are the benefits or disadvantages of using symmetric encryption? Because of the symmetric nature of the encryption and decryption process, it's relatively easy to implement and maintain. That's one shared secret that you have to maintain and keep secure. Think of your Wi-Fi password at home. There's one shared secret, your Wi-Fi password, that allows all devices to connect to it. Can you imagine having a specific Wi-Fi password for each device of yours? That would be a nightmare and super hard to keep track of. Symmetric algorithms are also very fast and efficient at encrypting and decrypting large batches of data. So what are the downsides of using symmetric encryption? While having one shared secret that both encrypts and decrypts seems convenient up front, this can actually introduce some complications. What happens if your secret is compromised? Imagine that your Wi-Fi password was stolen and now you have to change it. Now you have to update your Wi-Fi password on all your devices and any devices your friends or family might bring over. What do you have to do when a friend or family member comes to visit and they want to get on your Wi-Fi? You need to provide them with your Wi-Fi password, or the shared secret that protects your Wi-Fi network. This usually isn't an issue since you hopefully know the person and you trust them, and it's usually only one or two people at a time. But what if you had a party at your place with 50 strangers? Side note, why are you having a party at your home with 50 strangers? Anyhow, how could you provide the Wi-Fi password only to the people you trust without strangers overhearing? Things could get really awkward really fast. In the next lesson, we'll explore other ways besides symmetric key algorithms to protect data and information.

### Rob-what-a-security-engineer-does

My name Rob Anthony and I work as a Security Engineer on the Access Control Infrastructure Team. Some servers you only want certain people to access, say that they are sensitive. The financial systems is only the financial team should access them. So, that's what we do. We control the policies that give people access to certain things within Google. A security engineer is responsible for making sure that Google is protected. One of the big things that people want to do is monitor their servers. They have a server sets up, they want to be able to make sure that it's actually up. Need to have someone monitor there, or they might want to be able to access it remotely to be able to manage it. This machine might be across the country at some data center. Imagine if I'm talking to you and telling you something and there's a malicious person sitting next to us. If we're talking just plainly, then that malicious person can hear us. If we're talking through code encrypted channel, that person can't hear us. So, if someone has a design and they say, "Yeah, we just want to keep it simple. We just want to use Telnet." We'll be like, "Oh, no, no, no. I know setting up something much more secure is a little bit more about here. So, the reason why this is why you should do that."

## Public-key-or-asymmetric-encryption

### Asymmetric-cryptography

In the previous lesson, we covered one of two major categories that encryption ciphers fall into, symmetric key ciphers. In this next lesson, we'll cover the second class of ciphers called asymmetric or public key ciphers. Remember why symmetric ciphers are referred to as symmetric? It's because the same key is used to encrypt as to decrypt. This is in contrast to asymmetric encryption systems because as the name implies, different keys are used to encrypt and decrypt. So how exactly does that work? Well, let's imagine here that there are two people who would like to communicate securely, we'll call them Suzanne and Daryll. Since they're using asymmetric encryption in this example, the first thing they each must do is generate a private key, then using this private key, a public key is derived. The strength of the asymmetric encryption system comes from the computational difficulty of figuring out the corresponding private key given a public key. Once Suzanne and Daryll have generated private and public key pairs, they exchange public keys. You might have guessed from the names that the public key is public and can be shared with anyone, while the private key must be kept secret. When Suzanne and Daryll have exchanged public keys, they're ready to begin exchanging secure messages. When Suzanne wants to send Daryll an encrypted message, she uses Daryll's public key to encrypt the message and then send the ciphertext. Daryll can then use his private key to decrypt the message and read it, because of the relationship between private and public keys, only Daryll's private key can decrypt messages encrypted using Daryll's public key. The same is true of Susanne's key pairs. So when Daryll is ready to reply to Suzanne's message, he'll use Suzanne's public key to encode his message and Suzanne will use her private key to decrypt the message. Can you see why it's called asymmetric or public key cryptography? We've just described encryption and decryption operations using an asymmetric cryptosystem, but there's one other very useful function the system can perform, public key signatures. Let's go back to our friends Suzanne and Daryll. Let's say, Suzanne wants to send a message to Darryll and she wants to make sure that Daryll knows the message came from her and no one else, and that the message was not modified or tampered with. She could do this by composing the message and combining it with her private key to generate a digital signature. She then sends this message along with the associated digital signature to Daryll. We're assuming Suzanne and Daryll have already exchanged public keys previously in this scenario. Daryll can now verify the message's origin and authenticity by combining the message, the digital signature, and Suzanne's public key. If the message was actually signed using Susanne's private key and not someone else's and the message wasn't modified at all, then the digital signature should validate. If the message was modified, even by one whitespace character, the validation will fail and Daryll shouldn't trust the message. This is an important component of the asymmetric cryptosystem. Without message verification, anyone could use Daryll's public key and send him an encrypted message claiming to be from Suzanne. The three concepts that an asymmetric cryptosystem grants us are confidentiality, authenticity, and non-repudiation. Confidentiality is granted through the encryption-decryption mechanism. Since our encrypted data is kept confidential and secret from unauthorized third parties. Authenticity is granted by the digital signature mechanism, as the message can be authenticated or verified that it wasn't tampered with. Non-repudiation means that the author of the message isn't able to dispute the origin of the message. In other words, this allows us to ensure that the message came from the person claiming to be the author. Can you see the benefit of using an asymmetric encryption algorithm versus a symmetric one? Asymmetric encryption allows secure communication over an untrusted channel, but with symmetric encryption, we need some way to securely communicate the shared secret or key with the other party. If that's the case, it seems like asymmetric encryption is better, right? Well, sort of. While asymmetric encryption works really well in untrusted environments, it's also computationally more expensive and complex. On the other hand, symmetric encryption algorithms are faster, and more efficient, and encrypting large amounts of data. In fact, what many secure communications schemes do is take advantage of the relative benefits of both encryption types by using both, for different purposes. An asymmetric encryption algorithm is chosen as a key exchange mechanism or cipher. What this means, is that the symmetric encryption key or shared secret is transmitted securely to the other party using asymmetric encryption to keep the shared secret secure in transit. Once the shared secret is received, data can be sent quickly, and efficiently, and securely using a symmetric encryption cipher. Clever? One last topic to mention is somewhat related to asymmetric encryption and that's MACs or Message Authentication Codes, not to be confused with media access control or MAC addresses. A MAC is a bit of information that allows authentication of a received message, ensuring that the message came from the alleged sender and not a third party masquerading as them. It also ensures that the message wasn't modified in some way in order to provide data integrity. This sounds super similar to digital signatures using public key cryptography, doesn't it? While very similar, it differs slightly since the secret key that's used to generate the MAC is the same one that's used to verify it. In this sense, it's similar to symmetric encryption system and the secret key must be agreed upon by all communicating parties beforehand or shared in some secure way. This describes one popular and secure type of MAC called HMAC or a Keyed-Hash Message Authentication Code. HMAC uses a cryptographic hash function along with a secret key to generate a MAC. Any cryptographic hash functions can be used like Shahwan or MD5 and the strength or security of the MAC is dependent upon the underlying security of the cryptographic hash function used. The MAC is sent alongside the message that's being checked. The Mac is verified by the receiver by performing the same operation on the received message, then comparing the computed MAC with the one received with the message. If the MACs are the same, then the message is authenticated. There are also MACs based on symmetric encryption ciphers, either block or stream like DES or AES, which are called CMACs or Cipher-Based Message Authentication Codes. The process is similar to HMAC, but instead of using a hashing function to produce a digest, a symmetric cipher with a shared keys used to encrypt the message and the resulting output is used as the MAC. A specific and popular example of a CMAC though slightly different is CBC-MAC or Cipher Block Chaining Message Authentication Codes. CBC-MAC is a mechanism for building MACs using block ciphers. This works by taking a message and encrypting it using a block cipher operating in CBC mode. CBC mode is an operating mode for block ciphers that incorporates a previously encrypted block cipher text into the next block's plain text. So, it builds a chain of encrypted blocks that require the full, unmodified chain to decrypt. This chain of interdependently encrypted blocks means that any modification to the plain text will result in a different final output at the end of the chain, ensuring message integrity. In the next section, we'll check out some common examples of asymmetric encryption algorithms and systems. I'll see you there.

### Asymmetric-encryption-algorithms

So, one of the first practical asymmetric cryptography systems to be developed is RSA, name for the initials of the three co-inventors. Ron Rivest, Adi Shamir and Leonard Adleman. This crypto system was patented in 1983 and was released to the public domain by RSA Security in the year 2000. The RSA system specifies mechanisms for generation and distribution of keys along with encryption and decryption operation using these keys. We won't go into the details of the math involved, since it's pretty high-level stuff and beyond the scope of this class. But, it's important to know that the key generation process depends on choosing two unique, random, and usually very large prime numbers. DSA or Digital Signature Algorithm is another example of an asymmetric encryption system, though its used for signing and verifying data. It was patented in 1991 and is part of the US government's Federal Information Processing Standard. Similar to RSA, the specification covers the key generation process along with the signing and verifying data using the key pairs. It's important to call out that the security of this system is dependent on choosing a random seed value that's incorporated into the signing process. If this value was leaked or if it can be inferred if the prime number isn't truly random, then it's possible for an attacker to recover the private key. This actually happened in 2010 to Sony with their PlayStation 3 game console. It turns out they weren't ensuring this randomized value was changed for every signature. This resulted in a hacker group called failOverflow being able to recover the private key that Sony used to sign software for their platform. This allowed moders to write and sign custom software that was allowed to run on the otherwise very locked down console platform. This resulted in game piracy becoming a problem for Sony, as this facilitated the illicit copying and distribution of games which caused significant losses in sales. I've included links to more about this in the next reading, in case you want to dive deeper. Earlier, we talked about how asymmetric systems are commonly used as key exchange mechanisms to establish a shared secret that will be used with symmetric cipher. Another popular key exchange algorithm is DH or Diffie-Hellman named for the co-inventors. Let's walk through how the DH key exchange algorithm works. Let's assume we have two people who would like to communicate over an unsecured channel, and let's call them Suzanne and Daryll. I've grown pretty fond of these two. First, Suzanne and Daryl agree on the starting number that would be random and will be very large integer. This number should be different for every session and doesn't need to be secret. Next, each person chooses another randomized large number but this one is kept secret. Then, they combine their shared number with their respective secret number and send the resulting mix to each other. Next, each person combines their secret number with the combined value they received from the previous step. The result is a new value that's the same on both sides without disclosing enough information to any potential eavesdroppers to figure out the shared secret. This algorithm was designed solely for key exchange, though there have been efforts to adapt it for encryption purposes. It's even been used as part of a PKI system or Public Key Infrastructure system. We'll dive more into PKI systems later in this course. Elliptic curve cryptography or ECC is a public key encryption system that uses the algebraic structure of elliptic curves over finite fields to generate secure keys. What does that even mean? Well, traditional public key systems, make use of factoring large prime numbers whereas ECC makes use of elliptic curves. And elliptic curve is composed of a set of coordinates that fit in equation, similar to something like Y to the second equals X to the third, plus A X plus B. Elliptic curves have a couple of interesting and unique properties. One is horizontal symmetry, which means that at any point in the curve can be mirrored along the x axis and still make up the same curve. On top of this, any non-vertical line will intersect the curve in three places at most. Its this last property that allows elliptic curves to be used in encryption. The benefit of elliptic curve based encryption systems is that they are able to achieve security similar to traditional public key systems with smaller key sizes. So, for example, a 256 bit elliptic curve key, would be comparable to a 3,072 bit RSA key. This is really beneficial since it reduces the amount of data needed to be stored and transmitted when dealing with keys. Both Diffie-Hellman and DSA have elliptic curve variants, referred to as ECDH and ECDSA, respectively. The US NEST recommends the use of EC encryption, and the NSA allows its use to protect up the top secret data with 384 bit EC keys. But, the NSA has expressed concern about EC encryption being potentially vulnerable to quantum computing attacks, as quantum computing technology continues to evolve and mature. I'm going to buy Suzanne and Darryl drink today for all their hard work. In the meantime, we've cooked up an assignment for you that will test your encryption and decryption skills. Take your time to decode all the details, and I'll see you all in the next lesson.

## Hashing

### Hashing

So far, we've talked about two forms of encryption, symmetric and asymmetric. In this next lesson, we're going to cover a special type of function that's widely used in computing and especially within security, hashing. No, not the breakfast kind, although those are delicious. Hashing or a hash function is a type of function or operation that takes in an arbitrary data input and maps it to an output of a fixed size, called a hash or a digest. The output size is usually specified in bits of data and is often included in the hashing function name. What this means exactly is that you feed in any amount of data into a hash function and the resulting output will always be the same size. But the output should be unique to the input, such that two different inputs should never yield the same output. Hash functions have a large number of applications in computing in general, typically used to uniquely identify data. You may have heard the term hash table before in context of software engineering. This is a type of data structure that uses hashes to accelerate data lookups. Hashing can also be used to identify duplicate data sets in databases or archives to speed up searching of tables or to remove duplicate data to save space. Depending on the application, there are various properties that may be desired, and a variety of hashing functions exist for various applications. We're primarily concerned with cryptographic hash functions which are used for various applications like authentication, message integrity, fingerprinting, data corruption detection and digital signatures. Cryptographic hashing is distinctly different from encryption because cryptographic hash functions should be one directional. They're similar in that you can input plain text into the hash function and get output that's unintelligible but you can't take the hash output and recover the plain text. The ideal cryptographic hash function should be deterministic, meaning that the same input value should always return the same hash value. The function should be quick to compute and be efficient. It should be infeasible to reverse the function and recover the plain text from the hash digest. A small change in the input should result in a change in the output so that there is no correlation between the change in the input and the resulting change in the output. Finally, the function should not allow for hash collisions, meaning two different inputs mapping to the same output. Cryptographic hash functions are very similar to symmetric key block ciphers and that they operate on blocks of data. In fact, many popular hash functions are actually based on modified block ciphers. Lets take a basic example to quickly demonstrate how a hash function works. We'll use an imaginary hash function for demonstration purposes. Lets say we have an input string of "Hello World" and we feed this into a hash function which generates the resulting hash of E49A00FF. Every time we feed this string into our function, we get the same hash digest output. Now let's modify the input very slightly so it becomes "hello world", all lower case now. While this change seems small to us, the resulting hash output is wildly different, FF1832AE. Here is the same example but using a real hash function, in this case md5sum. Hopefully, the concept of hash functions make sense to you now. In the next section, we will explore some examples of hashing algorithms and dive into weaknesses or attacks on hash functions.

### Hashing-algorithms

In this section, we'll cover some of the more popular hashing functions, both currently and historically. MD5 is a popular and widely used hash function designed in the early 1990s as a cryptographic hashing function. It operates on a 512 bit blocks and generates 128 bit hash digests. While MD5 was published in 1992, a design flaw was discovered in 1996, and cryptographers recommended using the SHA-1 hash, a more secure alternative. But, this flaw was not deemed critical, so the hash function continued to see widespread use and adoption. In 2004, it was discovered that MD5 is susceptible to hash collisions, allowing for a bad actor to craft a malicious file that can generate the same MD5 digest as another different legitimate file. Bad actors are the worst, aren't they? Shortly after this flaw was discovered, security researchers were able to generate two different files that have matching MD5 hash digests. In 2008, security researchers took this a step further and demonstrated the ability to create a fake SSL certificate, that validated due to an empty five hash collision. Due to these very serious vulnerabilities in the hash function, it was recommended to stop using MD5 for cryptographic applications by 2010. In 2012, this hash collision was used for nefarious purposes in the flame malware, which used the forge Microsoft digital certificate to sign their malware, which resulted in the malware appearing to be from legitimate software that came from Microsoft. You can learn more about the flame malware in the next reading. When design flaws were discovered in MD5, it was recommended to use SHA-1 as a replacement. SHA-1 is part of the secure hash algorithm suite of functions, designed by the NSA and published in 1995. It operates a 512 bit blocks and generates 160 bit hash digest. SHA-1 is another widely used cryptographic hashing functions, used in popular protocols like TLS/SSL, PGP SSH, and IPsec. SHA-1 is also used in version control systems like Git, which uses hashes to identify revisions and ensure data integrity by detecting corruption or tampering. SHA-1 and SHA-2 were required for use in some US government cases for protection of sensitive information. Although, the US National Institute of Standards and Technology, recommended stopping the use of SHA-1 and relying on SHA-2 in 2010. Many other organizations have also recommended replacing SHA-1 with SHA-2 or SHA-3. And major browser vendors have announced intentions to drop support for SSL certificates that use SHA-1 in 2017. SHA-1 also has its share of weaknesses and vulnerabilities, with security researchers trying to demonstrate realistic hash collisions. During the 2000s, a bunch of theoretical attacks were formulated and some partial collisions were demonstrated, but full collisions using these methods requires significant computing power. One such attack was estimated to require $2.77 million in cloud computing CPU resources, Wowza. In 2015, a different attack method was developed that didn't demonstrate a full collision but this was the first time that one of these attacks was demonstrated which had major implications for the future security of SHA-1. What was only theoretically possible before, was now becoming possible with more efficient attack methods and increases in computing performance, especially in the space of GPU accelerated computations in cloud resources. A full collision with this attack method was estimated to be feasible using CPU and GPU cloud computing for approximately $75 to $120,000 , much cheaper than previous attacks. You can read more about these attacks and collisions in the next reading. In early 2017, the first full collision of SHA-1 was published. Using significant CPU and GPU resources, two unique PDF files were created that result in the same SHA-1 hash. The estimated processing power required to do this was described as equivalent of 6,500 years of a single CPU, and 110 years of a single GPU computing non-stop. That's a lot of years. There's also the concept of a MIC, or message integrity check. This shouldn't be confused with a MAC or message authentication check, since how they work and what they protect against is different. A MIC is essentially a hash digest of the message in question. You can think of it as a check sum for the message, ensuring that the contents of the message weren't modified in transit. But this is distinctly different from a MAC that we talked about earlier. It doesn't use secret keys, which means the message isn't authenticated. There's nothing stopping an attacker from altering the message, recomputing the checksum, and modifying the MIC attached to the message. You can think of MICs as protecting against accidental corruption or loss, but not protecting against tampering or malicious actions.

### Hashing-algorithms-continued

We've already alluded to attacks on hashes. Now let's learn more details, including how to defend against these attacks. One crucial application for cryptographic hash functions is for authentication. Think about when you log into your e-mail account. You enter your e-mail address and password. What do you think happens in the background for the e-mail system to authenticate you? It has to verify that the password you entered is the correct one for your account. You could just take the user supplied password and look up the password on file for the given account and compare them. Right? If they're the same, then the user is authenticated. Seems like a simple solution but does that seem secure to you? In the authentication scenario, we'd have to store user passwords in plain text somewhere. That's a terrible idea. You should never ever store sensitive information like passwords in plain text. Instead, you should do what pretty much every authentication system does, store a hash of the password instead of the password itself. When you log into your e-mail account the password you entered is run through the hashing function and then the resulting hash digest is compared against the hash on file. If the hashes match, then we know the password is correct, and you're authenticated. Password shouldn't be stored in plain text because if your systems are compromised, passwords for other accounts are ultimate prize for the attacker. If an attacker manages to gain access to your system and can just copy the database of accounts and passwords, this would obviously be a bad situation. By only storing password hashes, the worst the attacker would be able to recover would be password hashes, which aren't really useful on their own. What if the attacker wanted to figure out what passwords correspond to the hashes they stole? They would perform a brute force attack against the password hash database. This is where the attacker just tries all possible input values until the resulting hash matches the one they're trying to recover the plain text for. Once there's a match, we know that the input that's generated that matches the hash is the corresponding password. As you can imagine, a brute force attack can be very computationally intensive depending on the hashing function used. An important characteristic to call out about brute force attacks is, technically, they're impossible to protect against completely. A successful brute force attack against even the most secure system imaginable is a function of attacker time and resources. If an attacker has unlimited time and or resources any system can be brute force. Yikes. The best we can do to protect against these attacks, is to raise the bar. Make it sufficiently time and resource intensive so that it's not practically feasible in a useful timeframe or with existing technology. Another common method to help raise the computational bar and protect against brute force attacks is to run the password through the hashing function multiple times, sometimes through thousands of interactions. This would require significantly more computations for each password guess attempt. That brings us to the topic of rainbow tables. Don't be fooled by the colorful name. These tables are used by bad actors to help speed up the process of recovering passwords from stolen password hashes. A rainbow table is just a pre-computed table of all possible password values and their corresponding hashes. The idea behind rainbow table attacks is to trade computational power for disk space by pre-computing the hashes and storing them in a table. An attacker can determine what the corresponding password is for a given hash by just looking up the hash in their rainbow table. This is unlike a brute force attack where the hash is computed for each guess attempt. It's possible to download rainbow tables from the internet for popular password lists and hashing functions. This further reduces the need for computational resources requiring large amounts of storage space to keep all the password and hash data. You may be wondering how you can protect against these pre-computed rainbow tables. That's where salts come into play. And no, I'm not talking about table salt. A password salt is additional randomized data that's added into the hashing function to generate the hash that's unique to the password and salt combination. Here's how it works. A randomly chosen large salt is concatenated or tacked onto the end of the password. The combination of salt and password is then run through the hashing function to generate hash which is then stored alongside the salt. What this means now for an attacker is that they'd have to compute a rainbow table for each possible salt value. If a large salt is used, the computational and storage requirements to generate useful rainbow tables becomes almost unfeasible. Early Unix systems used a 12 Bit salt, which amounts to a total of 4,096 possible salts. So, an attacker would have to generate hashes for every password in their database, 4,096 times over. Modern systems like Linux, BSD and Solaris use a 128 bit salt. That means there are two to the 128 power possible salt values, which is over 340 undecillion. That's 340 with 36 zeros following. Clearly, 128 bit salt raises the bar high enough that a rainbow table attack wouldn't be possible in any realistic time-frame. Just another scenario when adding salt to something makes it even better. That runs out our lesson on hashing functions. Up next we'll talk about real world applications of cryptography and explain how it's used in various applications and protocols. But first, a project that will help you get hands on with hashing. Hashtag, get it done.

## Cryptography-applications

### Public-key-infrastructure

In this lesson, we're going to cover PKI, or Public Key Infrastructure. Spoiler alert, this is a critical piece to securing communications on the Internet today. Earlier we talked about Public Key Cryptography and how it can be used to securely transmit data over an untrusted channel and verify the identity of a sender using digital signatures. PKI is a system that defines the creation, storage and distribution of digital certificates. A digital certificate is a file that proves that an entity owns a certain public key. A certificate contains information about the public key, the entity it belongs to and a digital signature from another party that has verified this information. If the signature is valid and we trust the entity that signed the certificate, then we can trust the public key to be used to securely communicate with the entity that owns it. The entity that's responsible for storing, issuing, and signing certificates is referred to as CA, or Certificate Authority. It's a crucial component of the PKI system. There's also an RA, or Registration Authority, that's responsible for verifying the identities of any entities requesting certificates to be signed and stored with the CA. This role is usually lumped together with the CA. A central repository is needed to securely store and index keys and a certificate management system of some sort makes managing access to storage certificates and issuance of certificates easier. There are a few different types of certificates that have different applications or uses. The one you're probably most familiar with is SSL or TLS server certificate. This is a certificate that a web server presents to a client as part of the initial secure setup of an SSL, TLS connection. Don't worry, we'll cover SSL, TLS in more detail in a future lesson. The client usually a web browser will then verify that the subject of the certificate matches the host name of the server the client is trying to connect to. The client will also verify that the certificate is signed by a certificate authority that the client trusts. It's possible for a certificate to be valid for multiple host names. In some cases, a wild card certificate can be issued where the host name is replaced with an asterisk, denoting validity for all host names within a domain. It's also possible for a server to use what's called a Self Sign Certificate. You may have guessed from the name. This certificate has been signed by the same entity that issued the certificate. This would basically be signing your own public key using your private key. Unless you already trusted this key, this certificate would fail to verify. Another certificate type is an SSL or TLS client certificate. This is an optional component of SSL, TLS connections and is less commonly seen than server certificates. As the name implies, these are certificates that are bound to clients and are used to authenticate the client to the server, allowing access control to a SSL, TLS service. These are different from server certificates in that the client certificates aren't issued by a public CA. Usually the service operator would have their own internal CA which issues and manages client certificates for their service. There are also code signing certificates which are used for signing executable programs. This allows users of these signed applications to verify the signatures and ensure that the application was not tampered with. It also lets them verify that the application came from the software author and is not a malicious twin. We've mentioned certificate authority trust, but not really explained it. So let's take some time to go over how it all works. PKI is very much dependent on trust relationships between entities, and building a network or chain of trust. This chain of trust has to start somewhere and that starts with the Root Certificate Authority. These root certificates are self signed because they are the start of the chain of trust. So there's no higher authority that can sign on their behalf. This Root Certificate Authority can now use the self-signed certificate and the associated private key to begin signing other public keys and issuing certificates. It builds a sort of tree structure with the root private key at the top of the structure. If the root CA signs a certificate and sets a field in the certificate called CA to true, this marks a certificate as an intermediary or subordinate CA. What this means is that the entity that this certificate was issued to can now sign other certificates. And this CA has the same trust as the root CA. An intermediary CA can also sign other intermediate CAs. You can see how this extension of trust from one root CA to intermediaries can begin to build a chain. A certificate that has no authority as a CA is referred to as an End Entity or Leaf Certificate. Similar to a leaf on a tree, it's the end of the tree structure and can be considered the opposite of the roots. You might be wondering how these root CAs wind up being trusted in the first place. Well, that's a very good question. In order to bootstrap this chain of trust, you have to trust a root CA certificate, otherwise the whole chain is untrusted. This is done by distributing root CA certificates via alternative channels. Each major OS vendor ships a large number of trusted root CA certificates with their OS. And they typically have their own programs to facilitate distribution of root CA certificates. Most browsers will then utilize the OS provided store of root certificates. Let's do a deep dive into certificates beyond just their function. The X.509 standard is what defines the format of digital certificates. It also defines a certificate revocation list or CRL which is a means to distribute a list of certificates that are no longer valid. The X.509 standard was first issued in 1988 and the current modern version of the standard is version 3. The fields defined in X.509 certificate are, the version, what version of the X.509 standard certificate adheres to. Serial number, a unique identifier for their certificate assigned by the CA which allows the CA to manage and identify individual certificates. Certificate Signature Algorithm, this field indicates what public key algorithm is used for the public key and what hashing algorithm is used to sign the certificate. Issuer Name, this field contains information about the authority that signed the certificate. Validity, this contains two subfields, Not Before and Not After, which define the dates when the certificate is valid for. Subject, this field contains identifying information about the entity the certificate was issued to. Subject Public Key Info, these two subfields define the algorithm of the public key along with the public key itself. Certificate signature algorithm, same as the Subject Public Key Info field, these two fields must match. Certificate Signature Value, the digital signature data itself. There are also certificate fingerprints which aren't actually fields in the certificate itself, but are computed by clients when validating or inspecting certificates. These are just hash digests of the whole certificate. You can read about the full X.509 standard in the next reading. And alternative to the centralized PKI model of establishing trust and binding identities is what's called the Web of Trust. A Web of Trust is where individuals instead of certificate authorities sign other individuals' public keys. Before an individual signs a key, they should first verify the person's identity through an agreed upon mechanism. Usually by checking some form of identification, driver's license, passport, etc. Once they determine the person is who they claim to be, signing their public key is basically vouching for this person. You're saying that you trust that this public key belongs to this individual. This process would be reciprocal, meaning both parties would sign each other's keys. Usually people who are interested in establishing web of trust will organize what are called Key Signing Parties where participants performed the same verification and signing. At the end of the party everyone's public key should have been signed by every other participant establishing a web of trust. In the future when one of these participants in the initial key signing party establishes trust with a new member, the web of trust extends to include this new member and other individuals they also trust. This allows separate webs of trust to be bridged by individuals and allows the network of trust to grow.

### Cryptography-in-action

In this section, we'll dive into some real world applications of the encryption concepts that we've covered so far. In the last section, we mentioned SSL/TLS when we were talking about digital certificates. Now that we understand how digital certificates function and the crucial roles CAs play, let's check out how that fits into securing web traffic via HTTPS. You've probably heard of HTTPS before, but do you know exactly what it is and how it's different from HTTP? Very simply, HTTPS is the secure version of HTTP, the Hypertext Transfer Protocol. So how exactly does HTTPS protect us on the Internet? HTTPS can also be called HTTP over SSL or TLS since it's essentially encapsulating the HTTP traffic over an encrypted, secured channel utilizing SSL or TLS. You might hear SSL and TLS used interchangeably, but SSL 3.0, the latest revision of SSL, was deprecated in 2015, and TLS 1.2 is the current recommended revision, with version 1.3 still in the works. Now, it's important to call out that TLS is actually independent of HTTPS, and is actually a generic protocol to permit secure communications and authentication over a network. TLS is also used to secure other communications aside from web browsing, like VoIP calls such as Skype or Hangouts, email, instant messaging, and even Wi-Fi network security. TLS grants us three things. One, a secure communication line, which means data being transmitted is protected from potential eavesdroppers. Two, the ability to authenticate both parties communicating, though typically, only the server is authenticated by the client. And three, the integrity of communications, meaning there are checks to ensure that messages aren't lost or altered in transit. TLS essentially provides a secure channel for an application to communicate with a service, but there must be a mechanism to establish this channel initially. This is what's referred to as a TLS handshake. I'm more of a high five person myself, but we can move on. The handshake process kicks off with a client establishing a connection with a TLS enabled service, referred to in the protocol as ClientHello. This includes information about the client, like the version of the TLS that the client supports, a list of cipher suites that it supports, and maybe some additional TLS options. The server then responds with a ServerHello message, in which it selects the highest protocol version in common with the client, and chooses a cipher suite from the list to use. It also transmits its digital certificate and a final ServerHelloDone message. The client will then validate the certificate that the server sent over to ensure that it's trusted and it's for the appropriate host name. Assuming the certificate checks out, the client then sends a ClientKeyExchange message. This is when the client chooses a key exchange mechanism to securely establish a shared secret with the server, which will be used with a symmetric encryption cipher to encrypt all further communications. The client also sends a ChangeCipherSpec message indicating that it's switching to secure communications now that it has all the information needed to begin communicating over the secure channel. This is followed by an encrypted Finished message which also serves to verify that the handshake completed successfully. The server replies with a ChangeCipherSpec and an encrypted Finished message once the shared secret is received. Once complete, application data can begin to flow over the now the secured channel. High five to that. The session key is the shared symmetric encryption key using TLS sessions to encrypt data being sent back and forth. Since this key is derived from the public-private key, if the private key is compromised, there's potential for an attacker to decode all previously transmitted messages that were encoded using keys derived from this private key. To defend against this, there's a concept of forward secrecy. This is a property of a cryptographic system so that even in the event that the private key is compromised, the session keys are still safe. The SSH, or secure shell, is a secure network protocol that uses encryption to allow access to a network service over unsecured networks. Most commonly, you'll see SSH use for remote login to command line base systems, but the protocol is super flexible and has provisions for allowing arbitrary networks and traffic over those ports to be tunneled over the encrypted channel. It was originally designed as a secure replacement for the Telnet protocol and other unsecured remote login shell protocols like rlogin or r-exec. It's very important that remote login and shell protocols use encryption. Otherwise, these services will be transmitting usernames and passwords, along with keystrokes and terminal output in plain text. This opens up the possibility for an eavesdropper to intercept credentials and keystrokes, not good. SSH uses public key cryptography to authenticate the remote machine that the client is connecting to, and has provisions to allow user authentication via client certificates, if desired. The SSH protocol is very flexible and modular, and supports a wide variety of different key exchange mechanisms like Diffie-Hellman, along with a variety of symmetric encryption ciphers. It also supports a variety of authentication methods, including custom ones that you can write. When using public key authentication, a key pair is generated by the user who wants to authenticate. They then must distribute those public keys to all systems that they want to authenticate to using the key pair. When authenticating, SSH will ensure that the public key being presented matches the private key, which should never leave the user's possession. PGP stands for Pretty Good Privacy. How's that for a creative name? Well, PGP is an encryption application that allows authentication of data along with privacy from third parties relying upon asymmetric encryption to achieve this. It's most commonly used for encrypted email communication, but it's also available as a full disk encryption solution or for encrypting arbitrary files, documents, or folders. PGP was developed by Phil Zimmerman in 1991 and it was freely available for anyone to use. The source code was even distributed along with the software. Zimmerman was an anti nuclear activist, and political activism drove his development of the PGP encryption software to facilitate secure communications for other activists. PGP took off once released and found its way around the world, which wound up getting Zimmerman into hot water with the US federal government. At the time, US federal export regulations classified encryption technology that used keys larger than 40 bits in length as munitions. This meant that PGP was subject to similar restrictions as rockets, bombs, firearms, even nuclear weapons. PGP was designed to use keys no smaller than 128-bit, so it ran up against these export restrictions, and Zimmerman faced a federal investigation for the widespread distribution of his cryptographic software. Zimmerman took a creative approach to challenging these restrictions by publishing the source code in a hardcover printed book which was made available widely. The idea was that the contents of the book should be protected by the first amendment of the US constitution. Pretty clever? The investigation was eventually closed in 1996 without any charges being filed, and Zimmerman didn't even need to go to court. You can read more about why he developed PGP in the next reading. PGP is widely regarded as very secure, with no known mechanisms to break the encryption via cryptographic or computational means. It's been compared to military grade encryption, and there are numerous cases of police and government unable to recover data protected by PGP encryption. In these cases, law enforcement tend to resort to legal measure to force the handover of passwords or keys. Originally, PGP used the RSA algorithm, but that was eventually replaced with DSA to avoid issues with licensing.

### Securing-network-traffic

Let's talk about securing network traffic. As we've seen, encryption is used for protecting data both from the privacy perspective and the data integrity perspective. A natural application of this technology is to protect data in transit, but what if our application doesn't utilize encryption? Or what if we want to provide remote access to internal resources too sensitive to expose directly to the Internet? We use a VPN, or Virtual Private Network solution. A VPN is a mechanism that allows you to remotely connect a host or network to an internal private network, passing the data over a public channel, like the Internet. You can think of this as a sort of encrypted tunnel where all of our remote system's network traffic would flow, transparently channeling our packets via the tunnel through the remote private network. A VPN can also be point-to-point, where two gateways are connected via a VPN. Essentially bridging two private networks through an encrypted tunnel. There are a bunch of VPN solutions using different approaches and protocols with differing benefits and tradeoffs. Let's go over some of the more popular ones. IPsec, or Internet Protocol Security, is a VPN protocol that was designed in conjunction with IPv6. It was originally required to be standards compliant with IPv6 implementations, but was eventually dropped as a requirement. It is optional for use with IPv6. IPsec works by encrypting an IP packet and encapsulating the encrypted packet inside an IPsec packet. This encrypted packet then gets routed to the VPN endpoint where the packet is de-encapsulated and decrypted then sent to the final destination. IPsec supports two modes of operations, transport mode and tunnel mode. When transport mode is used, only the payload of the IP packet is encrypted, leaving the IP headers untouched. Heads up that authentication headers are also used. Header values are hashed and verified, along with the transport and application layers. This would prevent the use of anything that would modify these values, like NAT or PAT. In tunnel mode, the entire IP packet, header, payload, and all, is encrypted and encapsulated inside a new IP packet with new headers. While not a VPN solution itself, L2TP, or Layer 2 Tunneling Protocol, is typically used to support VPNs. A common implementation of L2TP is in conjunction with IPsec when data confidentially is needed, since L2TP doesn't provide encryption itself. It's a simple tunneling protocol that allows encapsulation of different protocols or traffic over a network that may not support the type of traffic being sent. L2TP can also just segregate and manage the traffic. ISPs will use the L2TP to deliver network access to a customer's endpoint, for example. The combination of L2TP and IPsec is referred to as L2TP IPsec and was officially standardized in ietf RFC 3193. The establishment of an L2TP IPsec connection works by first negotiating an IPsec security association. Which negotiates the details of the secure connection, including key exchange, if used. It can also share secrets, public keys, and a number of other mechanisms. I've included a link to more info about it in the next reading. Next, secure communication is established using Encapsulating Security Payload. It's a part of the IPsec suite of protocols, which encapsulates IP packets, providing confidentiality, integrity, and authentication of the packets. Once secure encapsulation has been established, negotiation and establishment of the L2TP tunnel can proceed. L2TP packets are now encapsulated by IPsec, protecting information about the private internal network. An important distinction to make in this setup is the difference between the tunnel and the secure channel. The tunnel is provided by L2TP, which permits the passing of unmodified packets from one network to another. The secure channel, on the other hand, is provided by IPsec, which provides confidentiality, integrity, and authentication of data being passed. SSL TLS is also used in some VPN implementations to secure network traffic, as opposed to individual sessions or connections. An example of this is OpenVPN, which uses the OpenSSL library to handle key exchange and encryption of data, along with control channels. This also enables OpenVPN to make use of all the cyphers implemented by the OpenSSL library. Authentication methods supported are pre-shared secrets, certificate-based, and username password. Certificate-based authentication would be the most secure option, but it requires more support and management overhead since every client must have a certificate. Username and password authentication can be used in conjunction with certificate authentication, providing additional layers of security. It should be called out that OpenVPN doesn't implement username and password authentication directly. It uses modules to plug into authentication systems, which we'll cover in the next module. OpenVPN can operate over either TCP or UDP, typically over port 1194. It supports pushing network configuration options from the server to a client and it supports two interfaces for networking. It can either rely on a Layer 3 IP tunnel or a Layer 2 Ethernet tap. The Ethernet tap is more flexible, allowing it to carry a wider range of traffic. From the security perspective, OpenVPN supports up to 256-bit encryption through the OpenSSL library. It also runs in user space, limiting the seriousness of potential vulnerabilities that might be present. There are a lot of acronyms to take in, so take a minute to go over them and read more about them, and I'll see you in the next video.

### Cryptographic-hardware

Welcome back. Let's dive right in. Another interesting application of cryptography concepts, is the Trusted Platform Module or TPM. This is a hardware device that's typically integrated into the hardware of a computer, that's a dedicated crypto processor. TPM offers secure generation of keys, random number generation, remote attestation, and data binding and sealing. A TPM has unique secret RSA key burned into the hardware at the time of manufacture, which allows a TPM to perform things like hardware authentication. This can detect unauthorized hardware changes to a system. Remote attestation is the idea of a system authenticating its software and hardware configuration to a remote system. This enables the remote system to determine the integrity of the remote system. This can be done using a TPM by generating a secure hash of the system configuration, using the unique RSA key embedded in the TPM itself. Another use of this secret hardware backed encryption key is data binding and sealing. It involves using the secret key to derive a unique key that's then used for encryption of data. Basically, this binds encrypted data to the TPM and by extension, the system the TPM is installed in, sends only the keys stored in hardware in the TPM will be able to decrypt the data. Data sealing is similar to binding since data is encrypted using the hardware backed encryption key. But, in order for the data to be decrypted, the TPM must be in a specified state. TPM is a standard with several revisions that can be implemented as a discrete hardware chip, integrated into another chip in a system, implemented in firmware software or virtualize then a hypervisor. The most secure implementation is the discrete chip, since these chip packages also incorporate physical tamper resistance to prevent physical attacks on the chip. Mobile devices have something similar referred to as a secure element. Similar to a TPM, it's a tamper resistant chip often embedded in the microprocessor or integrated into the mainboard of a mobile device. It supplies secure storage of cryptographic keys and provides a secure environment for applications. An evolution of secure elements is the Trusted Execution Environment or TEE which takes the concept a bit further. It provides a full-blown isolated execution environment that runs alongside the main OS. This provides isolation of the applications from the main OS and other applications installed there. It also isolates secure processes from each other when running in the TEE. TPMs have received criticism around trusting the manufacturer. Since the secret key is burned into the hardware at the time of manufacture, the manufacturer would have access to this key at the time. It is possible for the manufacturer to store the keys that could then be used to duplicate a TPM, that could break the security the module is supposed to provide. There's been one report of a physical attack on a TPM which allowed a security researcher to view and access the entire contents of a TPM. But this attack required the use of an electron microscope and micron precision equipment for manipulating a TPM circuitry. While the process was incredibly time intensive and required highly specialized equipment, it proved that such an attack is possible despite the tamper protections in place. You can read more about it just after this video. TPMs are most commonly used to ensure platform integrity, preventing unauthorized changes to the system either in software or hardware, and full disk encryption utilizing the TPM to protect the entire contents of the disk. Full Disk Encryption or FDE, as you might have guessed from the name, is the practice of encrypting the entire drive in the system. Not just sensitive files in the system. This allows us to protect the entire contents of the disk from data theft or tampering. Now, there are a bunch of options for implementing FDE. Like the commercial product PGP, Bitlocker from Microsoft, which integrates very well with TPMs, Filevault 2 from Apple, and the open source software dm-crypt, which provides encryption for Linux systems. An FDE configuration will have one partition or logical partition that holds the data to be encrypted. Typically, the root volume, where the OS is installed. But, in order for the volume to be booted, it must first be unlocked at boot time. Because the volume is encrypted, the BIOS can't access data on this volume for boot purposes. This is why FDE configurations will have a small unencrypted boot partition that contains elements like the kernel, bootloader and a netRD. At boot time, these elements are loaded which then prompts the user to enter a passphrase to unlock the disk and continue the boot process. FDE can also incorporate the TPM, utilizing the TPM encryption keys to protect the disk. And, it has platform integrity to prevent unlocking of the disk if the system configuration is changed. This protects against attacks like hardware tampering, and disk theft or cloning. Before we wrap up this module on encryption, I wanted to touch base on the concept of random. Earlier, when we covered the various encryption systems, one commonality kept coming up that these systems rely on. Did you notice what it was? That's okay if you didn't. It's the selection of random numbers. This is a very important concept in encryption because if your number selection process isn't truly random, then there can be some kind of pattern that an adversary can discover through close observation and analysis of encrypted messages over time. Something that isn't truly random is referred to as pseudo-random. It's for this reason that operating systems maintain what's referred to as an entropy pool. This is essentially a source of random data to help seed random number generators. There's also dedicated random number generators and pseudo-random number generators, that can be incorporated into a security appliance or server to ensure that truly random numbers are chosen when generating cryptographic keys. I hope you found these topics in cryptography interesting and informative. I know I did when I first learned about them. In the next module, we'll cover the three As of security, authentication, authorization and accounting. These three As are awesome and I'll tell you why in the next module. But before we get there, one final quiz on the cryptographic concept we've covered so far.