I

Side drawer

Loading...

Course navigation bar

Page

1. Cryptography Foundations

Introduction

This is a media component which plays a video. Select the play / pause button to watch or listen.

Video Player

View Transcript

Welcome to Cryptography for Developers and Architects! This course explores the world of cryptography as it applies to building and architecting software systems.

Cryptography can be a daunting topic: implement it poorly and you can leave your data extremely vulnerable.

This course will arm you with the know-how to make sound decisions around cryptography in order to protect your data's confidentiality and integrity. Real-world examples — and mistakes! — will help you learn to design and implement cryptographic controls.

Incomplete

What is Cryptography?

Cryptography is a collection of mathematical disciplines and techniques used to protect the secrecy and integrity of data. The broad science of cryptography aims to mitigate common problems encountered during the transmission and storage of data.

**Mitigations provided by the use of cryptographic algorithms are:**

* Preventing plaintext data from being seen by unauthorized entities.
* Proving that data was generated by a given user/system.
* Proving that a user or system is who it claims to be.
* Ensuring that a user or system cannot deny having generated certain data.
* Preventing attackers from modifying data without detection.

Although cryptography does not eliminate security issues, it does make them more manageable by reducing the task of protecting a large amount of data to a matter of protecting a relatively small key.

Terminology

Before deep-diving into cryptography, it is important to understand a few core cryptographic concepts.

**Plaintext** – The data whose confidentiality needs to be protected is commonly referred to as plaintext (or sometimes cleartext). It is the state of the data before being passed to a cryptographic algorithm for encryption, or the state of data after being passed to a cryptographic algorithm for decryption.

**Ciphertext** – Ciphertext is what is produced after plaintext is encrypted using a cryptographic algorithm

**Cryptographic Primitive** – Cryptographic primitives are low-level cryptographic algorithms, or sets of predefined instructions, used to transform data. These primitives are considered to be the "building blocks" of cryptography. Examples include one-way hash functions, public-key cryptography, and digital signatures.

**Cryptographic Protocol** – A cryptographic protocol is responsible for performing the cryptographic function, often as sequences of cryptographic primitives. The protocol describes how the algorithms should be used. Examples include AES and 3DES.

**Encrypt** – To encrypt is the act of converting plaintext data into ciphertext.

**Decrypt** – To decrypt data is to transform from ciphertext format back to its original plaintext state.

Complete

What is cryptography used for?

Cryptographic systems can provide a number of services to protect data.

This component is an accordion comprised of collapsible content panels containing display text. Select the item titles to toggle the visibility of these content panels.

Confidentiality

Keeping data private is one of the most common uses of cryptography. Whether it's passwords sent during a log-on process or storing confidential medical records in a database, cryptography can assure that only users who have access to the appropriate key will get access to the data. Encryption is typically the cryptographic primitive used when confidentiality is required.

Data Integrity

You can use cryptography to ensure that unauthorized modifications of data during storage or transmission can be detected. Certain cryptographic primitives such as message authentication codes and digital signatures can safeguard data by providing a secure checksum.

Data Origin Authentication

You can use cryptography to verify that a message was generated by an authorized user or machine. Software update mechanisms (such as the Microsoft Windows Updater) use cryptographic primitives such as message authentication codes and digital signatures to verify that the data origin is trusted before executing or installing any downloaded files.

Entity Authentication

Using cryptography, you can verify the identity of a remote user (or system). A typical example is the TLS certificate of a web server providing proof to a user that they are connected to the correct server.

For example, when you enter credentials on a banking website, how can you be confident that the website belongs to the bank? Your browser uses cryptography to verify that sometime during the past couple of years, a widely trusted certificate authority verified that the key corresponding to the TLS certificate presented by the server actually belongs to it. As long as you know that the hostname in your browser's address bar is actually a hostname used by your bank, and as long as the bank's TLS key has not been stolen, you can be reasonably confident that your credentials will only be readable by the bank's servers.

Message authentication codes and digital signatures are the cryptographic primitives typically used for entity authentication. However, note that message authentication codes are not used for verifying the authenticity of websites; only digital signatures are used for that.

Non-Repudiation

Cryptographic functions can be utilized to help prove that a unique user has made a transaction request. It must ***not be possible*** for the user to refute their actions.

For example, a customer may request a transfer of money from their bank account to be paid to another account. Later, they claim never to have made the request and demand the money be refunded. Using cryptographic digital signatures, you can prove that the user in fact did authorize the transaction.

The concept of repudiation stems from the legal world, and it is heavily debated whether cryptography provides guaranteed non-repudiation in legal situations. For the purposes of this course, understand that cryptographic controls can aid in the non-repudiation process, but they are not considered a silver bullet. We discuss some of the reasons for the debate around digital signatures and non-repudiation in another lesson.

Real-World Cryptography

This is a media component which plays a video. Select the play / pause button to watch or listen.

Video Player

View Transcript

The concept of cryptography dates back thousands of years. The art of hiding and scrambling secret messages has been around since humans carved hieroglyphics on the walls. Today, cryptography is a widely studied science (perhaps even an art) that is an integral piece of any computer to computer communication.

Cryptography is used in many facets of computing, especially when it comes to the Internet. Without the application of cryptographic algorithms to network traffic, it would impossible to browse the web without someone or something listening in on every word.

Using cryptographic primitives (such as public-key encryption, symmetric-key encryption, message authentication codes, and digital signatures) allows web browsers to communicate securely with servers that they need to transact with.

Most modern desktops and laptops come with the ability to use what is called full-disk encryption. This data protection mechanism generally uses symmetric-key encryption to encrypt all the bits that go onto a hard drive, making the data unreadable by anybody who does not have access to the decryption key.

These are just a few simple examples of how cryptography helps us secure our data and experience the Internet in a more sane and secure way.

Cryptographic Primitives

Although you don't need to understand the complex mathematics behind cryptographic algorithms, it is important to know when and how to use those standard algorithms known as primitives to architect and develop a secure application. Let's take a closer look at some of the more common cryptographic primitives.

|  |  |
| --- | --- |
| **Encryption** | Encryption algorithms transform data (referred to as plaintext) to make it unreadable to anyone except those possessing special knowledge, usually referred to as a key. |
| **Hash Functions** | Hash functions are well-defined procedures or mathematical functions that convert a large, possibly variable-sized amount of data into a small fixed-length datum. |
| **Message Authentication Codes (MACs)** | Message authentication codes (MACs) convert a variable-sized amount of data to a fixed-length datum, similarly to hash functions. However, MACs also use a secret key that is shared between the sender and receiver. |
| **Digital Signatures** | Digital signatures are algorithms used for demonstrating the authenticity of a digital message or document. |

Knowledge Check

This component is a multiple choice question. Once you have selected an option select the submit button below

True or False: A file should be encrypted before sending it to a recipient in order to ensure it has not been tampered with upon opening.

2. Encryption

Introduction

This is a media component which plays a video. Select the play / pause button to watch or listen.

View Transcript

At its core, encryption is a simple concept. It is the application of mathematical functions to data in order to transform plaintext into ciphertext. Encryption aims to make the data unreadable to anyone except those who possess special knowledge (usually a key).

There are various types of encryption systems, and each has its own strengths and weaknesses. Some are fast once the sender and recipient agree upon a key, but contain no provisions for them to agree upon a key. Some make key management easier, but are significantly slower.

This lesson will help you understand these tradeoffs so that you can choose the best solution for you and design secure, high-performance systems.

Common Encryption Ciphers

In the world of encryption, a cipher is an algorithm used to perform encryption or decryption. Ciphers depend on a key to perform operations. Without knowledge of this key, it should be impossible to decrypt the data to recover plaintext.

Modern cryptographic ciphers can be categorized in two ways:

* **Block ciphers** take fixed-length blocks of plaintext data and perform encryption or decryption operations on them. Usually, the message is longer in length than the block size, which results in a series of sequential blocks of data that need to be encrypted or decrypted. The cipher operates on these blocks one at a time.
* **Stream ciphers** use a key to generate a keystream (a string of bits that is the same length as the data) and XOR the data with the keystream to encrypt or decrypt it.

This lesson primarily focuses on block ciphers. Block ciphers should be used in most applications requiring symmetric key cryptography unless there is a specific reason for using a stream cipher.

Currently, no stream ciphers exist that are well studied and widely accepted as being secure. The RC4 stream cipher was widely used until weaknesses were discovered in it. Stream ciphers are useful for cryptography implementations in hardware and in telecommunication applications, as their implementations tend to be simpler and faster.

Common Encryption Primitives

Each of these encryption primitives has strengths and weaknesses in particular use cases.

Securing Block Ciphers

There are several ways that block ciphers can be used in a cryptosystem. When architecting and developing your encryption plan, it is important to follow industry best practices and "future-proof" your system as much as possible. This means ensuring that your software can handle swapping out cryptographic algorithms and protocols if necessary.

For example, if you are using the AES encryption algorithm and a serious weakness is discovered in it in five years, you will need to swap it out for whatever is considered secure at that time, and will potentially need to re-encrypt all your existing data using the new algorithm under a new key.

Unless you are a cryptography expert, however, it is best to keep your implementation as straightforward and "boring" as possible. We will discuss encryption modes, padding modes, secret keys, and initialization vectors. These are the only variables that you should be choosing with encryption algorithms. Do not implement proprietary encryption algorithms, use any proprietary padding schemes, etc.Block Cipher Encryption Modes

Since a block cipher operates on relatively short, fixed-length blocks of data, encrypting or decrypting a piece of data typically requires encrypting or decrypting multiple blocks. A mode of operation specifies exactly how the cipher is applied to each block of data.

As with most things in cryptography, there are a number of different encryption modes to choose from. We will not go into great detail about each encryption mode, but instead will provide a few basic guidelines.

* **Electronic Codebook Mode (ECB)** should not be used to encrypt data except in some specific scenarios (such as encrypting a symmetric key using an asymmetric key block cipher like RSA).
* **Cipher block chaining (CBC)**is the most commonly used mode of operation, and when implemented correctly can offer a secure way to encrypt data. CBC mode can be vulnerable in certain situations, especially if a predictable initialization vector is in use. Also, it does not help protect data integrity, and subtle implementation mistakes can lead to attackers being able to decrypt encrypted data using padding oracle attacks.
* **Galois/counter mode (GCM)**is becoming widely supported, and can be used to securely protect both the confidentiality and integrity of data. This mode should be used whenever possible. When GCM mode is used, you must ensure that the same initialization vector is never reused to encrypt two separate pieces of data while using the same secret key.

Consult with a cryptography expert if you are considering implementing a custom solution or using a more obscure block cipher mode.

Initialization Vectors

Most block cipher encryption modes require an initialization vector (IV) for each encryption operation. An IV is a value used to generate completely different ciphertexts each time, even when you are encrypting the same plaintext multiple times.

**Some important facts about IVs:**

* Initialization vector requirements are generally dependent on the mode of operation.
* In general, the IV should be random and unique every time you encrypt some data.
* Deriving the IV from a predictable source or using a constant IV (for example, by hard-coding it) can make your system insecure: don't do it!
* Initialization vectors should not be reused, but they don't need to be kept secret. Only the secret key needs to be kept secret.

If you reuse the IV to encrypt the same plaintext multiple times, the resulting ciphertext will be the same. This leaks information to a possible attacker. With several algorithms and encryption modes, there are also many other attacks possible if IVs are reused.

Block Cipher Padding Modes

Depending on the mode of operation, encrypting data with a block cipher may require padding since the mode of operation may only be designed to handle fixed-length blocks of data. Padding is applied when the plaintext's length is not the exact size of the algorithm's block size. The padding string is usually applied to the end of the message to meet the block size.

For example, AES encrypts blocks of 128 bits, and CBC mode can only handle fixed-length blocks of data. If you need to encrypt data whose length is not an exact multiple of 128 bits using AES in CBC mode, you will need to pad the last block of data so that it is 128 bits in length.

Developers don't need to worry about explicitly adding the padding before encryption and stripping it off after decryption. They only need to specify the padding mode to the cryptographic library.

The most widely supported padding type is PKCS5 or PKCS7. For most purposes, these two padding types are identical. PKCS5 needs to be specified with some cryptographic libraries, and PKCS7 needs to be specified with other cryptographic libraries. Unless another padding type is needed for a specific reason, we recommend PKCS5 or PKCS7 because they're widely supported.

However, recall that we recommend using GCM mode whenever possible; GCM mode can handle variable-length blocks of data and does not require any padding.

Using Asymmetric Key Encryption Securely

Given a public key, how do you know whether it belongs to the intended recipient or to an attacker impersonating the recipient? If an attacker convinces you to encrypt data using their public key, you will end up encrypting it so only the attacker can read It.

With public-key cryptography, you need a way to authenticate encryption keys.

There are currently three commonly used solutions to this problem. The most common is a public-key infrastructure. In a public-key infrastructure, there are trusted third parties called certificate authorities that verify their customers' identities and issue them certificates that cryptographically bind their identities to their public keys. This is the approach currently used by browsers to authenticate websites on the Internet.

Another solution to the public key authentication problem is identity-based encryption. This approach solves the problem by making the identity the public key. For example, a user's email address could be their public key. Although theoretically elegant, this solution has some practical problems and is not widely used because it requires a trusted third party to know secret keys.

Finally, the third solution is a web of trust. A web of trust can be thought of as a decentralized certificate authority. Its security depends on its users verifying other users' identities and vouching for them. This solution is used in Pretty Good Privacy (PGP).

Asymmetric Key Algorithms

Asymmetric key cryptography is typically only used to transport keys for symmetric key algorithms because asymmetric key cryptography is significantly slower than symmetric key cryptography. This allows the sender and the recipient to use the slower asymmetric key cryptography to agree upon a symmetric key, which in turn is used to perform actual data encryption and decryption. This is how the common SSL/TLS protocol works.

Because symmetric keys are relatively short (usually between 128 and 256 bits) they can be encrypted using just one operation of an algorithm, such as RSA. However, they are long enough and random enough that there is only a negligible chance of ever producing the same symmetric key twice. Therefore, the Electronic Codebook encryption mode is acceptable for use here. Remember that Electronic Codebook mode should generally not be used with symmetric key block ciphers except in some specific unusual scenarios.

There are two types of padding typically used with the RSA encryption algorithm: PKCS1 v1.5 and Optimal Asymmetric Encryption Padding (OAEP). Both are widely supported, but OAEP should be used whenever possible. PKCS1 v1.5 is vulnerable to a chosen-ciphertext attack known as Bleichenbacher's attack that in some scenarios can be used to decrypt protected data.

Knowledge Check

This component is a multiple choice question. Once you have selected an option select the submit button below

Which of the following is a mode of operation for a block cipher?



PKCS5



Hash Function



CBC



Initialization

SubmitShow feedback

**3. Cryptographic Hash Functions**

Introduction

Hash functions are one-way algorithms that can take an arbitrary-length piece of data and efficiently transform it into a fixed-length digest. Depending on the context, this digest is called a "fingerprint," a "checksum," or a "hash."

The goal is to get an easily computable, short, fixed-length representative for a piece of data. For example, the SHA-1 algorithm is able to hash arbitrary-length strings down to 160 bits. There is actually a length limit for all hash functions' inputs, but for practical purposes you don't need to worry about it. SHA-1, for example, allows hashing approximately 17 billion gigabytes of data.

Not all hash functions are cryptographic hash functions. A cryptographically secure hash function has three additional properties, each subtly different from the other:

* **Preimage resistance.** For a given digest, it is computationally infeasible to find an input that generates that digest.
* **Second-preimage resistance.** Given any piece of data, it is computationally infeasible to find another piece of data with the same digest.
* **Collision resistance.** It is computationally infeasible to find two distinct, random pieces of data that have the same digest.

These properties allow a digest to be securely used as a unique fixed-length representative for a piece of data.

Hash functions are not encryption functions; they do not require a key, and decryption is impossible.

Complete

Common Hash Function Algorithms

Some commonly used hash functions include MD5 and SHA-1. There are also the SHA-2 and SHA-3 families of hash functions, which include SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

However, commonly used does not necessarily equal safe. MD5 is considered *insecure* and should not be used in any applications where data integrity is important. SHA-1 is also reaching its end of life. You should use the SHA-2 and SHA-3 families of hash functions for any new applications that don't have backward compatibility concerns.

Consider the following when choosing a hashing algorithm for your system:

* Hash sizes of 256 bits are sufficient for most applications.
* As with encryption algorithms, design your application to cope with new hashes and algorithms.

Complete

Using Hash Functions: Password Storage

Due to their irreversible nature, hash functions are commonly used to store passwords. However, hashes on their own are not sufficient to protect passwords.

Traditionally, cryptographic hashes were used to store passwords because user passwords often do not need to be decrypted once they are stored. Password verification can simply be done by computing a hash of the password entered by the user and comparing it against the stored hash. Due to properties of cryptographic hash functions, it is very improbable that any other password will generate the same hash, making it an effective password verification mechanism. Also, since hash functions are not reversible, when given a password's hash from long-term storage, it will be difficult to derive the password from it.

However, hashing alone is no longer considered a strong password protection mechanism for two primary reasons:

* It is difficult for most users to remember long and complex passwords; therefore, users commonly choose weak and easily guessable passwords.
* Hash functions by their very definition are efficient; if your password database is stolen, an attacker can efficiently conduct brute force attacks, dictionary attacks, etc.

There are some ways to more effectively protect against offline password brute-forcing attacks (in case your users' password hashes are stolen):

* Add a secret key component to password hashes that is infeasible to brute-force and difficult to steal.
* Use an adaptive hash function that essentially runs the cryptographic hash function multiple times to generate the stored value; brute-forcing passwords will then require the attacker to compute a much larger number of hashes.

Also, add a unique, randomly generated salt to each hash (stored in the clear along with the hash) to protect against attacks that use pre-computed rainbow tables. A unique, random salt per user not only protects against pre-computed rainbow table attacks, it also helps ensure that if two users have the same password, their password hashes will be different.

Two approaches for password storage we recommend are:

* Salt + HMAC(Data=Salt+Password, Key=SecretKey)
* Salt + AdaptiveHash(Data=Salt+Password)

For more guidance on password storage, consult the [OWASP Password Storage Cheat Sheet](http://codisco.pe/CRYPTO3l1).

In summary, though the traditional password storage approaches of using simple cryptographic hashes or salted cryptographic hashes are common, they are no longer considered secure.

Complete

Using Hash Functions: Data Integrity

Hash functions are commonly used to help protect data integrity. There are three ways in which this is commonly done.

* ***Checksums.*** Often, there is a low-bandwidth secure channel and a high-bandwidth insecure channel available. For example, a software vendor may make its software available on many "mirror' sites that may not necessarily be trusted. However, the vendor may provide a checksum value to the recipient (for example, 155644321567) on Its own website over a TLS channel so that end users can download the software from any of the mirrors and can verify the authenticity using the hashes obtained over a secure channel.  
    
  Many websites do this incorrectly by supplying links to the mirror sites as well as a cryptographic hash on a page delivered over an insecure channel. With a site like that, an attacker can perform a man-in-the-middle attack to serve a modified version of the page containing a cryptographic hash of malicious software for a victim to install.
* ***Message Authentication Codes (MACs).***Message authentication codes are the analogue of symmetric key encryption algorithms.
* ***Digital Signatures.*** Digital signatures are the analogue of asymmetric key encryption algorithms.

Lifeboat's Sinking Ship

This is a media component which plays a video. Select the play / pause button to watch or listen.

Video Player

View Transcript

In early 2016, an online service called Lifeboat provided custom multiplayer environments to gamers that used the Minecraft mobile application. Lifeboat was hacked. This breach was surprisingly massive: over 7 million accounts were compromised.

Except perhaps it shouldn't have been that surprising. Researchers discovered that Lifeboat's user passwords were stored as unsalted MD5 hashes. To add icing to the cake, Lifeboat ***encouraged*** users to create "short, but difficult-to-guess passwords" because "This is not online banking." This made it easy work for attackers to crack every single hash.

This is an extreme example of what NOT to do with password storage. It also serves as a reminder that you should always treat your password database as though it is going to be compromised, and implement the guidelines contained in this lesson and the [OWASP Password Storage Cheat Sheet](http://codisco.pe/CRYPTO3l2).

Incomplete

Knowledge Check

This component is a multiple choice question. Once you have selected an option select the submit button below

When using hash functions to protect passwords, what is the random string of data used to prevent identical hashes from being stored?



Initialization vector



Signature



Salt



Checksum

**4. Message Authentication Codes (MAC)**

Introduction

Message authentication codes (MACs) are short pieces of information used to authenticate a message. A MAC is a fixed-length piece of data produced by an algorithm. This algorithm takes arbitrary-length data and a secret key as inputs and produces the fixed-length MAC value as the output. The MAC can later be used to verify the integrity of the message by anybody who has the secret key that was used to generate it. MACs are used to protect data authenticity and integrity.

MACs vs. Hashes

It's important to understand the difference between a MAC and a hash algorithm, as they each have very distinct use cases and implementation considerations. While MACs are often based on hash functions, they can also be based on other cryptographic algorithms, such as symmetric key encryption algorithms.

An unkeyed hash function is not sufficient for protecting data authenticity or integrity on its own, as the algorithm is publicly known and there is no secret key. Anybody can generate a hash of an arbitrary pieces of data. A message and its MAC, on the other hand, can be transmitted over an insecure channel, and a recipient that knows the key used to generate the MAC will be able to verify that the message was not tampered with in transit and was generated by an authorized entity that knew the shared secret key.

MAC algorithms are analogous to symmetric key encryption algorithms, and some of the same considerations apply for their usage (for example, the shared secret must be transmitted over a secure channel).

How It Works

It should be impossible for an attacker who does not have the shared secret between the sender and receiver to generate a valid MAC for a piece of data that will be accepted by the recipient.

To use a MAC in message transmission, the sender uses a strong cryptographic algorithm to compute the MAC value and sends that value along with the data. The receiver then passes the data and the shared secret key through the same algorithm and compares the output with the MAC that was received along with the message. If the values match, it is safe to assume that the data was generated by somebody who knew the secret key. If they do not match, the data has most likely been tampered with in transit and should be disregarded.

It's computationally difficult to forge a MAC that corresponds to a given piece of data without knowing the shared secret key between the sender and the receiver. As a result, you can reasonably assume that any received data with a valid MAC must have been generated by someone with the secret key.

Message authentication codes are the analogue of symmetric key encryption algorithms. They use the same key to sign and verify data. This means that the secret key must be shared in advance over a secure channel between the communicating parties. As with symmetric key encryption, the solution to the key exchange problem here is to use asymmetric key cryptography to transport the symmetric key. We'll examine this solution further when we look at TLS.

A Simple Example

Let's assume that Raymond and Hillary agree upon a random secret key with the value of "9fd529025bd2d0d33783e6a2e6615a8b". The key is long enough that it should be infeasible to brute-force.

Now, when Raymond wants to send Hillary the message "my secret message," he computes a MAC using the two inputs key="9fd529025bd2d0d33783e6a2e6615a8b", message="my secret message", and sends Hillary the message and the MAC.

When Hillary receives these values, she computes a MAC using the previously agreed upon secret and the received message. If the generated MAC matches the received MAC, then she can be reasonably certain that the message was generated by Raymond (or by herself since she has the same key).

An attacker can modify the message, but they will not be able to generate the correct MAC since they don't know Raymond and Hillary's shared key.

MAC Best Practices

The most commonly used message authentication code is HMAC, which is based on hash functions. Commonly used message authentication codes based on symmetric encryption algorithms are CMAC and OMAC.

Since MACs are used to protect data integrity and not confidentiality, it is a common practice to also apply an encryption layer to protect the confidentiality of the transmitted message. The best practice for order of operations is to encrypt the message first and then MAC the ciphertext. The MAC value should then be appended to the ciphertext. This way, the MAC can't leak information about the plaintext, and it protects against tampering of the ciphertext.

Knowledge Check

This component is a multiple choice question. Once you have selected an option select the submit button below

Message authentication codes are similar to \_\_\_\_\_\_\_ except that they also require a secret key as input.



encryption algorithms



cryptographic hash functions



digital signature algorithms



pseudo-random number generators

SubmitShow feedback

**5. Digital Signature**

Introduction

A digital signature is a cryptographic algorithm that binds a message to a person or entity. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, and that it was not altered in transit.

Digital signatures are similar to asymmetric key encryption algorithms, as they use different keys to sign and verify data. Digital signatures are similar to message authentication codes (MACs) in that they protect data integrity and provide data-origin authentication.

How Digital Signatures Work

Like handwritten signatures, digital signatures are unique to each signer. One party generates a key pair for a public-key algorithm such as RSA, and keeps the private half of the key pair secret. This secret half of the key pair is used as the signing key. The other half of the key pair can be distributed to anyone needing to verify the authenticity of messages from that sender. As with asymmetric key encryption algorithms, the private key (signing key in this case) should not be distributed publicly.

Digital signatures provide a cryptographic service that MACs don't provide: non-repudiation. In cryptographic theory, as long as the signer's private key is not compromised, and the digital signature algorithm is strong, the signer cannot deny having signed a message. In practice, however, software security enters the picture and additional requirements for non-repudiation include the following:

The software that performed the signing operation must be functioning correctly.

The machine on which the signing operation was performed must not have been compromised by malware.

A digital signature can be used with any kind of message, whether it is encrypted or not. However, non-repudiation may be difficult or impossible to achieve if a message is encrypted before it is signed.

For example, the recipient may have to reveal their decryption key to prove that the sender actually sent a ciphertext that decrypted to the given message. There would need to be additional proofs regarding the encryption mechanism used by the sender, any symmetric encryption keys used in encrypting the message, etc. It is best to avoid this complication and generate digital signatures of plaintext if non-repudiation is required.

Whether digital signatures can provide non-repudiation is not simply a technical matter; it is a legal matter as well. Consult a lawyer before designing a system that relies on non-repudiation provided by digital signatures.

Common Digital Signature Algorithms

Commonly used digital signature algorithms include RSA, the Digital Signature Algorithm (DSA), the Elliptic Curve Digital Signature Algorithm (ECDSA), and ElGamal.

Typically, digital signature algorithms are used in conjunction with cryptographic hash algorithms. Instead of signing a message directly, implementations generally involve hashing a message and then signing the hash. This is not only faster (because only a single asymmetric key operation is needed), but it is also more secure.

Digital Signature Implementation

As with asymmetric key encryption algorithms, we run into the problem of authenticating digital signature verification keys. However, digital signatures themselves are generally used to help solve this problem.

There are three common ways to authenticate public keys.

If identity-based signatures are used, the public key is the sender's identity. For example, a user's email address can be used as the key.

The other two approaches involve digital signatures themselves: public-key infrastructure and web of trust.

Public-Key Infrastructures

With public-key infrastructures, the public keys are signed by other trusted parties and the trust model is centralized. There are trusted third-parties known as certificate authorities that verify user or company identities and sign "certificates" using their private signing keys. The certificates contain identifying information for the user/company (typically known as the subject) as well as the subject's public key. The signature effectively binds information describing and identifying the subject to the subject's public verification key.

Certificates

The actual identifying information contained in a certificate (as well as the verification steps that are performed by a certificate authority) depend on the type of certificate issued. For example, with a Domain Validation certificate, the certificate authority only ensures that a given hostname currently possesses the private half of a given public key. With an Extended Validation certificate, the certificate authority also verifies that the certificate is being issued to a legitimate entity, and the certificate contains identifying information about the entity to whom the certificate was issued

Verification Keys

Anyone with the certificate authority's verification key can confirm that the certificate authority performed the verification steps required for the given type of certificate. At a minimum, given a valid certificate, this means that the public key in the certificate belongs to the subject, and that the subject information in the certificate is accurate.

Certificate Authority Keys

The final problem is how users get the verification keys for certificate authorities. Most modern operating systems and browsers contain a pre-installed set of certificate authority keys that are trusted. The operating system and browser vendors obtain these keys over trusted channels and distribute them in trust stores as part of their software. End users can also add additional certificate authority keys to their trust stores at a later time. Webs of trust have a decentralized trust model where arbitrary users can vouch for a key belonging to a given subject.

Putting it Together with TLS

TLS (and its predecessor SSL) is a protocol that operates directly on top of TCP. It is commonly used to provide encrypted communications across the Internet.

Building a TLS connection requires nearly every cryptographic primitive we've discussed. Let's go through the handshake step-by-step when RSA is used for session key transport.

Step 1

After building a TCP connection between the sender and receiver, the TLS handshake is initiated by the client. The client will send a number of details to the server, including which version of TLS it is using and preferred cipher suites / compression methods.

Step 2

The server picks a cipher suite / compression method.

Step 3

The server sends a digitally signed certificate that contains details about the server, including its public encryption key. The certificate is generally signed by a certificate authority trusted by the client.

Step 4

The client verifies the integrity of the certificate by using the certificate authority's public key. This step essentially involves generating a hash of the certificate's contents and comparing it against the result of performing a public key operation, using the certificate authority's public key against the digital signature in the certificate.

Step 5

Having verified that the certificate is valid and the server is who it says it is, the client generates a session key and encrypts it using the server's public encryption key.

Step 6

This encrypted key is sent to the server, which is able to decrypt it using its private key.

Step 7

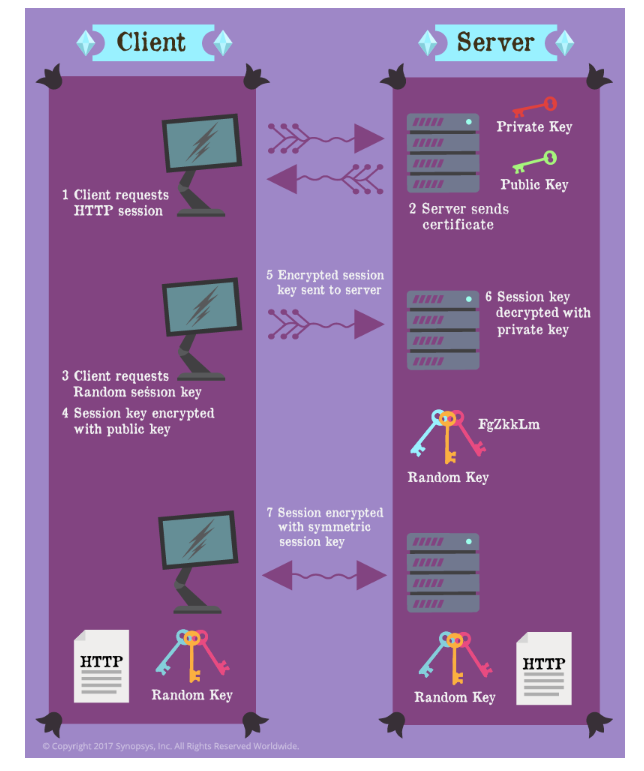
The established session key is used to derive encryption and MAC keys. Symmetric encryption and MACs are then used to secure further communication between the client and server. Only the client and server know these keys.

Knowledge Check

This component is a multiple choice question. Once you have selected an option select the submit button below

True or False: Hashes and digital signatures both can ensure that a given website belongs to the organization who claims to own it.

**Correct! Correct!**  
Digital signatures are used to ensure the identity of websites and organizations, but hashes are used as integrity checks to make sure the data has not been tampered with.



**6. Building a Secure Cryptosystem**

Introduction

The vast majority of attacks on cryptographic systems share one goal: obtain the plaintext of an encrypted message. This lesson will cover common attacks and misconfigurations related to cryptographic systems and outline what you as a developer can do to ensure that the confidentiality and integrity of your application's data is protected.

Over the years, cryptography researchers and academics have published many theoretical attack models and scenarios. It is impossible for us to cover the intricate details of all of these in a short course like this. However, we will cover practical guidelines that you — as an individual responsible for building software systems securely — can use to protect your data.

Things to Consider

Whether you are building an application from the ground up or just pushing out a small feature, it is crucial to take a step back and consider any new or changed functionality related to cryptography. Even minor mistakes in cryptography usage can introduce significant vulnerabilities into systems.

The topic of cryptography can seem daunting to a developer and may even feel like a burden. But with some background knowledge, it is easy to use cryptographic primitives and protocols securely in most situations. Let's look at some basic questions to consider any time you are considering using cryptography to protect the integrity and/or confidentiality of sensitive data.

**What cryptographic services are required? Do you need to protect the data's confidentiality, integrity, or both?** Answering this question will help you choose the right type of cryptographic primitives. One common mistake is encrypting data in an attempt to protect its integrity. Although the ciphertext looks random and seems opaque, this does not mean that an attacker cannot modify it intelligently. Of course, as we discussed earlier, some block cipher encryption modes like GCM can protect both data confidentiality and data integrity.

* **Does the ciphertext ever need to be decrypted?** Often, you just need to know whether a given value matches an existing protected value stored in a database. If this is the only required operation, you do not actually need to use a cryptographic primitive that allows decryption. For example, if your application needs to authenticate users using a username and password, there is no need to store the password in a format that can be decrypted. Doing so would mean that anyone with access to your database might be able to decrypt all users' passwords. Storing a salted one-way adaptive hash or a salted, keyed hash of the password is a better solution.
* **Do you need to search through the data based on partial matches?** If so, it is going to be nearly impossible to encrypt the data securely in a way that allows for efficient searching. Searching will involve potentially decrypting all of the data. This is something you will have to keep in mind when designing your application. If, for example, matching based on partial social security numbers or credit card numbers is a requirement, then encrypting means that searching will be a slow operation.
* **How many senders will need to protect data to be sent to a recipient?** Each symmetric key is only appropriate for sharing data among two entities. If you have more entities that need to communicate, do not share the same key among them. There are some Instances where doing this is necessary; however, the risks need to be evaluated. In general, if you have more than two entities that will be communicating, you should consider using asymmetric encryption. Otherwise, setting up pairwise symmetric keys can quickly become an unmanageable problem.

Architecting your Cryptosystem

Cryptography is one area of software development where it doesn't pay to come up with your own unique, cutting-edge solution. When rolling out a cryptographic system, it is recommended to only use well-known and "battle-tested" primitives. In short, never create your own cryptography systems.

One more time for good measure.

**Do. Not. Roll. Your. Own. Crypto.**

So how do you choose between the vast array of cryptographic primitives, protocols, ciphers, and key lengths? You will need to choose appropriate algorithms that provide the cryptographic services that you need.

Common mistakes include using proprietary or outdated algorithms. Organizations sometimes think that proprietary algorithms will be more secure because attackers will not know the algorithms and so will not be able to break them. However, unless designed by cryptography experts, cryptographic algorithms are likely to be very insecure, and may even be vulnerable to ciphertext-only attacks. It is almost always easier to incorporate a standard algorithm that has already been implemented in many cryptographic libraries than it is to invent a new one that is most likely going to be very weak.

You will then need to choose the correct parameters for the cryptographic primitive. For an encryption algorithm, this often includes the padding and encryption mode (and sometimes the key length). Common mistakes when using encryption algorithms include hard-coding cryptographic keys, using electronic codebook mode for encrypting a large amount of data using a symmetric key block cipher, and using a constant initialization vector.

Do you have requirements to compress your encrypted data? Encrypted data cannot be compressed well, but compressed data can be encrypted. If you use compression, make sure to compress the data before encrypting it.

Key Management

Key management is a critical part of building a hardened cryptosystem. Key management includes how keys are generated, exchanged, stored, rotated, and so on. An entire course can be dedicated just to the topic of key management, so we will be brief here. Below are some guidelines to consider when it comes to key management.

This component is an accordion comprised of collapsible content panels containing display text. Select the item titles to toggle the visibility of these content panels.

Key Length

When generating any cryptographic key, length plays an important role. Make the key long enough to make brute-force type attacks (where every possible key is attempting using powerful computers) infeasible. For example, a 128-bit symmetric key means that there are 340,282,366,920,938,463,463,374,607,431,768,211,456 possible keys to try using a brute-force attack. Current best practice is to utilize keys with 128 bits of strength. For a symmetric cipher like AES, this means at least 128-bit keys. For the RSA algorithm, this means keys with at least a 3072-bit modulus.

Visit [https://www.keylength.com](http://codisco.pe/CRYPTO6l1) for more details.

Key Rotation

Rotate keys frequently enough. Sometimes, keys may be hard-coded in source code, making them difficult to change. In other cases, the protocol being used may not allow keys to be changed easily (for example, the outdated "Wired Equivalent Privacy" protocol).

SSL/TLS Connections

Consider configuring SSL/TLS connections to use cipher suites that provide**forward secrecy**. These cipher suites typically contain the string DHE (for Ephemeral Diffie-Hellman) or ECDHE (for Ephemeral Elliptic Curve Diffie Hellman). If these cipher suites are used and the discrete logarithm problem continues to be difficult to solve, then eavesdropping attacks will not be sufficient to recover the plaintext in a SSL/TLS connection, even if the server's private key is known. Attackers who have compromised the server's private key can, however, perform active man-in-the-middle attacks to compromise traffic during future connections.

Hardware Security Module (HSM)

Utilize an hardware security module (HSM) when possible for the storage of cryptographic keys. HSMs are dedicated crypto processors that are specifically designed for the protection of the cryptographic key lifecycle. HSMs come in many flavors to fit a variety of budgets, and several cloud providers offer virtualized HSMs at reasonable rates.

Storage

If possible, do not store cryptographic keys in plaintext in files on disk.

Isolation

Always isolate keys as much as possible. Do not use the same keys in different environments (such as development, staging, production, etc.). Developers should never have access to keys used in production environments.

Choosing Protocols and Algorithms

**Here are some guidelines for choosing cryptographic algorithms and protocols for your application:**

* Avoid proprietary cryptographic algorithms and protocols. They are typically not vetted by the security and cryptography community and rely on "security through obscurity" instead of sound mathematics. Instead, use standard algorithms such as AES and RSA, and standard protocols such as IPSec, TLS, and SSH. Cryptographic algorithms are designed to be secure as long as the secret key is unknown to the attacker. There is no need to attempt to add any security through obscurity through proprietary algorithms.
* Never allow the security of any cryptographic design or implementation to depend on the secrecy of the algorithm or protocol itself. Rather, the algorithm or protocol should remain secure even if the specifications and implementations are made publicly available.
* Avoid the following outdated algorithms and protocols:
  + MD5, SHA-0, and SHA-1 for hashing algorithms
  + DES and RC4 for symmetric encryption
  + SSL 1.0, 2.0, 3.0, and TLS 1.0
* Monitor standard bodies such as NIST for future recommendations. Cryptography is a constantly changing field. As new discoveries in cryptanalysis are made, older algorithms will be found unsafe. In addition, as computing power increases, the feasibility of brute-force attacks will render other cryptosystems or the use of certain key lengths unsafe. Luckily, cryptography is not a rapidly changing field. For example, the RSA algorithm was first publicly described in 1977; however, it is still widely used today (although with longer keys). You should design your systems to be able to adapt to new cryptographic algorithms as necessary, but frequent changes are generally not needed.
* If you must comply with government or other compliance initiatives such as PCI DSS, make sure to check for specific cryptographic algorithm recommendations.
* The Cryptographic Lifecycle

Typically, cryptographic algorithms become weaker over time due to the invention of new cryptanalytic attacks and more computing power being available to attackers.

As an example, consider the MD5 hashing algorithm.

The algorithm was designed in 1991 and was considered theoretically secure for a couple of years. Some theoretical attacks against it were discovered in 1993 and 1995, but even in 1996 it was still considered secure from a practical point of view.

As additional attacks were discovered, the algorithm's weaknesses were revealed. In 2005, someone created two PostScript documents with the same MD5 hash as well as two X.509 certificates containing the same MD5 hash. This meant that MD5 could no longer be trusted for protecting data integrity for any practical purposes.

However, MD5's use continued for a while, and in 2008 an intermediate certificate authority certificate was created using an MD5 collision. This essentially meant that the person in possession of the certificate could impersonate any secure website. Attacks against one cryptographic primitive can have far-reaching effects on protocols that use the primitives.

However, note that an algorithm being broken in one way does not necessarily mean that it is useless for all purposes. For example, MD5 is still safe for storing hashed passwords, and it is commonly used for that purpose. Its use is not recommended for new systems, but legacy systems do not necessarily need to be updated if they store MD5 hashes of passwords.

The main lesson to be learned in this example is that cryptographic algorithms must be reviewed regularly and weak algorithms should be replaced. Since no useful algorithms are unconditionally secure, this will likely always be necessary.

Computational Security

Most modern cryptographic algorithms provide either computational/practical security or ad hoc/heuristic security. These are the least stringent models of security.

Computational or practical security means that the best-known ways to attack an algorithm require more computational power than an attacker will have. To be considered secure in this model, the algorithm must have been well-studied for a significant period of time. Algorithms that have this type of security are often related to difficult problems, but there exists no proof that the algorithm is as difficult to attack as solving the difficult problems.

The RSA algorithm is in this category. Its security is related to the problem of factoring large numbers, but there's no proof that the ability to factor large numbers is required to attack it.

The least stringent model of security is ad-hoc security. Almost all modern cryptographic algorithms fall in this category. This model requires convincing arguments that all currently known types of attacks would require an infeasible amount of computational resources to succeed against the given cryptographic algorithm. However, new types of attacks discovered in the future could completely compromise the security of the algorithm.

Case Study: SSL v2

SSL version 2 contained two significant weaknesses that caused SSL version 3 to be released about one year after its release. The main vulnerability was present during the handshake when the cipher suite was chosen.

SSL cipher negotiation typically begins with the client attempting to connect to a server and sending a list of cipher suites it supports. The server then chooses a cipher suite from the list supplied by the client or returns an error if it does not support any of them. The server may use any criteria of its choice to select a cipher suite; however, it is understood that if possible, the server will attempt to choose secure cipher suites over insecure ones.

However, SSL version 2 contained no integrity protection for the handshake process. An attacker could modify the list of cipher suites sent by the client so that the server only received the least secure cipher suite supported by the client. If the server also supported this cipher suite, it would be chosen because the server would think that it was the only suite supported by the client. Thus, an attacker could cause the SSL session to use the least secure cipher suite supported by both the client and the server. If the cipher suite was sufficiently weak, the attacker would be able to decrypt all traffic sent in the SSL session or even be able to inject traffic into it.

Takeaways

Let's wrap this lesson up with a few key takeaways.

The first takeaway is that you should never create your own cryptographic algorithms or protocols. Even experts miss subtle issues that are later discovered and fixed (for example, the issues in SSLv2). Attackers will find a way to exploit custom, one-off algorithms that are not designed by cryptography experts. You should rely on hardened protocols that have been in the public eye and have been heavily scrutinized over many years.

The second takeaway is to always perform due diligence when implementing any kind of cryptographic system. One small oversight or misconfiguration can result in total compromise.

It is also crucial to keep your cryptographic libraries up-to-date and ensure that you are always using the latest supported versions of cryptographic protocols. New versions are typically released to address security concerns brought by a new attack or advances in computing power. Staying up-to-date will help you avoid known vulnerabilities.

Complete

Knowlege Check

This component is a multiple choice question. Once you have selected an option select the submit button below

Which of these is the BEST way to choose a cryptographic protocol?



Use a custom-built protocol supplied by a security vendor



Implement a protocol that was recommended in an online web framework tutorial



Analyze what the protocol is being used for and reference a source such as NIST to help choose



Use the protocol that will introduce the least amount of latency

Introduction to Cryptography for Architects and Developers Assessment