Practical Cryptography with Go

Kyle Isom

Practical Cryptography With Go

Kyle Isom

This book is for sale at http://leanpub.com/gocrypto

This version was published on 2015-04-28

Tweet This Book!

Please help Kyle Isom by spreading the word about this book on Twitter!

The suggested hashtag for this book is #gocrypto.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#gocrypto

Contents

Chapter 1: Introduction

This is a book about cryptography: how to communicate securely. There are several objectives that cryptography aims to solve: **confidentiality**, **integrity**, and **authenticity**. It also helps solve some other problems that come up in secure communications, but it's important to remember that it isn't a complete solution to security problems. In this book, we'll look at how to build secure systems; some of the problems that cryptography does not solve will also be pointed out. This book will attempt to guide you in your attempt to understand how to use cryptography to secure your services, and illustrate it using the Go programming language.

As mentioned, the foundation of cryptographic security are the three goals of confidentiality, integrity, and authenticity. Confidentiality is the requirement that only the intended party can read a given message; integrity is the requirement that a message's contents cannot be tampered with; and authenticity is the requirement that the **provenance** (or origin) of a message can be trusted. Trust will play a large role in our secure systems, but there is no single solution to the problem. It will present many challenges in building secure systems. A cryptographic algorithm applies some transformations to data in order to achieve these goals, and various algorithms are applied to achieve different goals.

In order to discuss cryptography, a baseline vocabulary is needed. The following terms have specific meanings:

- The **plaintext** is the original message.
- The **ciphertext** is traditionally a message that has been transformed to provide confidentiality.
- A **cipher** is a cryptographic transformation that is used to encrypt or decrypt a message.
- A **message authentication code** (or **MAC**) is a piece of data that provides authenticity and integrity. A MAC algorithm is used both to generate and validate this code.
- To **encrypt** a message is to apply a confidentiality transformation, but is often used to describe a transformation that satisfies all three goals.
- To **decrypt** a message to reverse the confidentiality transformation, and often indicates that the other two properties have been verified.
- A **hash** or **digest algorithm** transforms some arbitrary message into a fixed-size output, also called a digest or hash. A cryptographic hash is such an algorithm that satisfies some specific security goals.
- A **peer** or **party** describes an entity involved in the communication process. It might be a person or another machine.

A secure communication system will protect against both **passive** and **active** attacks. A passive attack is one in which a party for whom a message is not intended is listening on the communications. An active attack is one in which some adversarial party is tampering with messages, and can inject, alter, or replay messages.

Cryptography should be used to solve specific problems, such as

- Eavesdropping: as is the case with in-person conversations, an attacker could listen in on traffic going in and out, potentially stealing secrets passed back and forth. The security goal of confidentiality will

mitigate this attack to an extent; while cryptography will obscure the contents of the message, by itself it doesn't hide the fact that two parties are communicating. An attacker might also be able to determine information based on the size of the messages.

- Tampering: traffic going in and out of the application could be modified en-route; the system needs to make sure that messages it receives have not been tampered with. The integrity goal is used to ensure messages haven't been tampered with.
- Spoofing: an attacker can pretend to be a legitimate user by faking certain details of a message. An attacker can use spoofing to steal sensitive information, forge requests from a user, or take over a legitimate session. Authentication helps to defend against this attack, by validating the identity of users and messages.

In this book, we'll look at the context of cryptography and some of the engineering concerns when building secure systems, symmetric and asymmetric (or public-key) cryptography, how to exchange keys, storing secrets, trust, and common use cases.

This book has an associated example code repository on Github at https://github.com/kisom/gocrypto/. This code has the code from the book, as well as some supplementary material that will be mentioned. As a general rule, the code in the book won't have comments; the code will be explained in the text. The example code is, however, commented.

Chapter 2: Engineering concerns and platform security

If cryptography is the proverbial vault door, it makes sense to evaluate the rest of the building and the foundation it's built on before deciding on the specifics of the vault door. It's often repeated that a secure system is only secure as its weakest component, which means the other parts of the system must be up to par before cryptography is useful. In this chapter, we'll look at the engineering concerns around secure systems, particularly for Unix-based systems.

Basic security

Security should provide **authentication**, **authorisation**, and **auditing**. Authentication means that the system verifies the identity of parties interacting with the system; authorisation verifies that they should be allowed to carry out this interaction; and auditing creates a log of security events that can be verified and checked to ensure the system is providing security. The end goal is some **assurance** that the system is secure.

Authentication

Authentication asks the question, "who am I talking to?"; it attempts to verify the **identity** of some party. Passwords are one means of authentication; they aren't a strong authentication mechanism because anyone who knows the password (whether because they chose it, were given it, or guessed it) will be authenticated. Multifactor authentication attempts to provide a stronger basis for assurance of an identity, and is based on three factors:

1. Something you know (such as a password)
2. Something you have (such as an authentication token of some kind)
3. Something you are (such as biometrics)

The most common multifactor authentication configuration found in common use is two-factor employing the first two factors. A user might be required to enter both their password and a time-based one-time password (such as with TOTP) from an app on their phone. The assumption here is that the key used to generate the TOTP is only present on the authenticator (i.e. the mail provider) and the user's phone, and that the user's phone hasn't been compromise or the key taken from it. Both sides share the key for this one-time password (OTP), and they do not have to communicate any additional information after setup to verify the password.

This is in contrast with two-step verification; an example is an SMS code sent to the phone. The user and server have to communicate, albeit over a different channel than the browser, to share this code. This still provides a channel for intercepting the code.

Authorisation

Authorisation asks the question, "should you be doing this?" Authorisation relies on an access control mechanism of some kind. This might be as simple as an access control list, where the system has a list of parties that should have access to a resource or should be allowed to perform some operation. The Unix security model uses a set of access control lists for reading, writing, and executing by the owner, the group the resource belongs to, and the world. It employs "discretionary access control": a user can explicitly change the values of those access control lists, giving other users and groups permission at their discretion. A mandatory access control model (such as provided by SELinux or AppArmor) operates on security levels or labels; each label is given a set of capabilities. Users or processes are given a label or security level, and they can only operate within the confines of the permitted capabilities.

As an example, a user might create a text file and opt to make it world-readable in the DAC model: any process or user can now access it. In the MAC model, access to that file would be restricted by label. If a process or user doesn't have permissions based on their label, they cannot access it, and the original user simply cannot share the text file in this way. The labels are assigned by an administrator or security officer, and the user cannot change this. Access control is no longer at the user's discretion, but mandatory.

An alternative to access control lists is the role-based access control security model is role-based security. On Unix systems, root has full control over the system; in a role-based system, this control is split among several roles, each of which has the minimum set of permissions to carry out that role. This model is also more fine grained that an access control list in that it can specify grant or permissions for specific operations.

Auditing

Security efforts are for nought if no one is auditing the system to ensure it is operating correctly. An audit log should be available, and its access restricted to auditors, that records security events. The events recorded and the details present will vary based on the requirements of the system. Auditors should also be confident that the audit log will not have been tampered with.

An attacker that successfully authenticates may not leave any indication that the system is compromised. The only way to identify the compromise is through positive auditing: that is, auditing the record of events that succeeded. Whether the risk of such a compromise outweighs the need to maintain usage privacy needs to be considered.

Policy

There should be a set of policies that clearly specify the authentication, authorisation, and auditing in a system. For a large organisation, this may be fairly complex. For an application given to users, it might be as simple as password-based and printing security failures to the system log. This isn't an engineering concern, per se, but it really must factor into a secure application.

Specifications

Specifications are a critical part of building a secure system to understand what its behaviour should be. From a security perspective, it's also important understand what its behaviour must not be. The security model is part of the system's specification, and care should be taken to build it properly.

Testing is an important part of the specification, as well. It is the assurance that the system behaves according to the specification. Unit tests verify the code paths within each unit of the system, functional tests verify that components and the system behaves as it should, and regression tests make sure that bugs aren't re-introduced. Integration tests may also be useful to verify compatibility.

Building secure systems depends on writing correct code. An incorrect system that is shipped is shipping with security flaws that will probably subvert any cryptographic security built in. The more code that is present in a system, the greater the attack surface: only the minimum code to implement a system that fits the specifications should be used. This includes any libraries used by the system: where possible, remove any unused functionality. This lowers the cost of the system as well: less test code has to be written, which reduces both the time and financial costs.

Security models

One of the key steps in designing a secure system is building a **security model**. A security model describes the assumptions that are made about the system, the conditions under which security can be provided, and identifies the threats to the system and their capabilities. A secure system cannot be built unless its characteristics and the problems it is intended to solve are understood. A security model should include an analysis of the system's attack surface (what components can come under attack), realistic threat vectors (where attacks come from), the parties that can attack the system and their capabilities, and what countermeasures will be required to provide security. The security model should not just cover the cryptographic components: the environment and platforms that the system will run in and on must be considered as well. It's also important to consider which problems are technical in nature, and which problems are social in nature. Trust is also a key consideration in this model; that is, understanding the roles, intended capabilities, and interactions of legitimate parties as well as the impact of a compromised trusted party.

From experience, it is extremely difficult to bolt security onto a system after it has been designed or built. It is important to begin discussing the security requirements during the initial stages of the system's design for the same reasons it is important to consider the other technical requirements. Failure to consider the load level of the system, for example, may result in poor architectural decisions being made that add a great deal of technical debt that impede a stable, reliable system. In the same manner, failure to consider the security requirements may result in similarly poor architectural decisions. A secure system must be reliable and it must be correct; most security vulnerabilities arise from exploiting parts of a system that do not behave correctly. Proper engineering is key to secure systems; clear specifications and both positive and negative testing (testing both that the system behaves correctly and that it fails gracefully) will greatly improve the system's ability to fulfill its security objectives. It's useful to consider security as a performance metric. The performance of a secure system relates to its ability to operate without being compromised, and its ability to recover from a compromise. The non-security performance of a secure system must also be considered: if the system is too slow or too difficult to use, it won't be used. An unused secure system is an insecure system as it is failing to provide message security.

The security components must service the other objectives of the system; they must do something useful inside the specifications of the system.

As part of the specifications of a secure system, the choice of cryptographic algorithms must also be made. In this book, we will prefer the NaCl algorithms for greenfield designs: they were designed by a well-respected cryptographer who is known for writing well-engineered code, they have a simple interface that is easy to use, and they were not designed by NIST. They offer high performance and strong security properties. In other

cases, compatibility with existing systems or standards (such as FIPS) is required; in that case, the compatible algorithms should be chosen.

On errors

The more information an attacker has about why a cryptographic operation failed, the better the chances that they will be able to break the system. There are attacks, for example, that operate by distinguishing between decryption failures and padding failures. In this book, we either signal an error using a bool or with a generic error value such as "encryption failed".

We'll also check assumptions as early as possible, and bail as soon as we see something wrong.

Input sanitisation

A secure system also has to scrutinise its inputs and outputs carefully to ensure that they do not degrade security or provide a foothold for an attacker. It's well understood in software engineering that input from the outside world must be sanitised; sanity checks should be conducted on the data and the system should refuse to process invalid data.

There are two ways to do this: blacklisting (a default allow) and whitelisting (a default deny). Blacklisting is a reactive measure that involves responding to known bad inputs; a blacklisting system will always be reacting to new bad input it detects, perhaps via an attack. Whitelisting decides on a set of correct inputs, and only permits those. It's more work up front to determine what correct inputs look like, but it affords a higher assurance in the system. It can also be useful in testing assumptions about inputs: if valid input is routinely being hit by the whitelist, perhaps the assumptions about the incoming data looks like are wrong.

Memory

At some point, given current technologies, sensitive data will have to be loaded into memory. Go is a managed-memory language, which means the user has little control over memory, presenting additional challenges for ensuring the security of a system. Recent vulnerabilities such as Heartbleed show that anything that is in memory can be leaked to an attacker with access to that memory. In the case of Heartbleed, it was an attacker with network access to a process that had secrets in memory. **Process isolation** is one countermeasure: preventing an attacker from accessing a process's memory space will help mitigate successful attacks against the system. However, an attacker who can access the machine, whether via a physical console or via a remote SSH session, now potentially has access to the memory space of any process running on that machine. This is where other security mechanisms are crucial for a secure system: they prevent an attacker from reaching that memory space.

It's not just the memory space of any process that's running on the machine that's vulnerable, though. Any memory swapped to disk is now accessible via the file system. A secret swapped to disk now has two places where it's present. If the process is running on a laptop that is put to sleep, that memory is often written to disk. If a peripheral has direct memory access (DMA), and many of them do, that peripheral has access to all the memory in the machine, including the memory space of every process. If a program crashes and dumps

core, that memory is often written to a core file. The CPU caches can also store secrets, which might be an additional attack surface, particularly on shared environments (such as a VPS).

There are a few methods to mitigate this: using the stack to prevent secrets from entering the heap, and attempting to zero sensitive data in memory when it's no longer needed (though this is not always effective, e.g. [5]). In this book, we'll do this where it makes sense, but the caveats on this should be considered.

There is also no guarantee that secrets stored on disk can be completely and securely erased (short of applying a healthy dose of thermite). If a sector on disk has failed, the disk controller might mark the block as bad and attempt to copy the data to another sector, leaving that data still on the hardware. The disk controller might be subverted, as disk drives contain drive controllers with poorly (if at all) audited firmware.

In short, given our current technologies, memory is a difficult attack surface to secure. It's helpful to ask these following questions for each secret:

1. Does it live on disk for long-term storage? If so, who has access to it? What authorisation mechanisms ensure that only authenticated parties have access?
2. When it's loaded in memory, who owns it? How long does it live in memory? What happens when it's no longer used?
3. If the secrets lived on a virtual machine, how much trust can be placed in parties that have access to the host machine? Can other tenants (i.e. users of other virtual machines) find a way to access secrets? What happens when the machine is decomissioned?

Randomness

Cryptographic systems rely on sources of sufficiently random data. We want the data from these sources to be indistinguishable from ideally random data (a uniform distribution over the range of possible values). There has been historically a lot of confusion between the options available on Unix platforms, but the right answer (e.g. [6]) is to use `/dev/urandom`. Fortunately, `crypto/rand.Reader` in the Go standard library uses this on Unix systems.

Ensuring the platform has sufficient randomness is another problem, which mainly comes down to ensuring that the kernel's PRNG is properly seeded before being used for cryptographic purposes. This is a problem particularly with virtual machines, which may be duplicated elsewhere or start from a known or common seed. In this case, it might be useful to include additional sources of entropy in the kernel's PRNG, such as a hardware RNG that writes to the kernel's PRNG. The host machine may also have access to the PRNG via disk or memory allowing its observation by the host, which must be considered as well.

Time

Some protocols rely on clocks being synced between peers. This has historically been a challenging problem. For example, audit logs often rely on the clock to identify when an event occurred. One of the major challenges in cryptographic systems is checking whether a key has expired; if the time is off, the system may incorrectly refuse to use a key that hasn't expired yet or use a key that has expired. Sometimes, the clock is used for a unique value, which shouldn't be relied on. Another use case is a monotonically-increasing counter; a

clock regression (e.g. via NTP) makes it not-so-monotonic. Authentication that relies on time-based one-time passwords also require an accurate clock.

Having a real-time clock is useful, but not every system has one. Real-time clocks can also drift based on the physical properties of the hardware. Network time synchronisations work most of the time, but they are subject to network failures. Virtual machines may be subject to the clock on the host.

Using the clock itself as a monotonic counter can also lead to issues; a clock that has drifted forward may be set back to the correct time (i.e. via NTP), which results in the counter stepping backwards. There is a CPU clock that contains ticks since startup that may be used; perhaps bootstrapped with the current timestamp, with the latest counter value stored persistently (what happens if the latest counter value is replaced with an earlier value or removed?).

It helps to treat clock values with suspicion. We'll make an effort to use counters instead of the clock where it makes sense.

Side channels

A side channel is an attack surface on a cryptographic system that is based entirely on the physical implementation; while the algorithm may be sound and correct, the implementation may leak information due to physical phenomena. An attacker can observe timings between operations or differences in power usage to deduce information about the private key or the original message.

Some of these types of side channels include:

- timing: an observation of the time it takes some piece of the system to carry out an operation. Attackers have used this to even successfully attack systems over the network ([2]).
- power consumption: this is often used against smart cards; the attacker observes how power usage changes for various operations.
- power glitching: the power to the system is glitched, or brought to near the shutdown value for the CPU. Sometimes this causes systems to fail in unexpected ways that reveals information about keys or messages.
- EM leaks: some circuits will leak electromagnetic emissions (such as RF waves), which can be observed.

These attacks can be surprisingly effective and devastating. Cryptographic implementations have to be designed with these channels in mind (such as using the constant time functions in `crypto/subtle`); the security model should consider the potential for these attacks and possible countermeasures.

Privacy and anonymity

When designing the system, it should be determined what measure of privacy and anonymity should be afforded. In a system where anonymity is needed, perhaps the audit log should not record when events succeed but only when they fail. Even these failures can leak information: if a user mistypes their password, will it compromise their identity? If information such as IP addresses are recorded, they can be used to de-anonymise users when combined with other data (such as activity logs from the user's computer). Think carefully about how the system should behave in this regard.

Trusted computing

One problem with the underlying platform is ensuring that it hasn't been subverted; malware and rootkits can render other security measures ineffective. It would be nice to have some assurance that the parties involved are running on platforms with a secure configuration. Efforts like the Trusted Computing Group's Trusted Computing initiative aim to prove some measure of platform integrity and authenticity of the participants in the system, but the solutions are complex and fraught with caveats.

Virtual environments

The cloud is all the rage these days, and for good reason: it provides a cost-effective way to deploy and manage servers. However, there's an old adage in computer security that an attacker with physical access to the machine can compromise any security on it, and cloud computing makes getting "physical" access to some of these machines much easier. The hardware is emulated in software, so an attacker who gains access to the host (even via a remote SSH session or similar) has equivalent access. This makes the task of securing sensitive data, like cryptographic keys, in the cloud a dubious prospect given current technologies. If the host isn't trusted, how can the virtual machine be trusted? This doesn't just mean trust in the parties that own or operate the host: is their management software secure? How difficult is it for an attacker to gain access to the host? Can another tenant (or user on another virtual machine on the host) gain access to the host or other virtual machines they shouldn't be able to? If the virtual machine is decomissioned, is the drive sufficently wiped so that it never ends up in another tenants hands? Security models for systems deployed in a virtual environment need to consider the security of the host provider and infrastructure in addition to the system being developed, including the integrity of the images that are being run.

Public key infrastructure

When deploying a system that uses public key cryptography, determining how to trust and distribute public keys becomes a challenge that adds extra engineering complexity and costs to the system. A public key by itself contains no information; some format that contains any other required identity and metadata information needs to be specified. There are some standards for this, such as the dread X.509 certificate format (which mates a public key with information about the holder of the private key and holder of the public key that vouches for this public key). Deciding on what identity information to include and how it is to be verified should be considered, as should the lifetime of keys and how to enforce key expirations, if needed. There are administrative and policy considerations that need to be made; PKI is largely not a cryptographic problem, but it does have cryptographic impact.

Key rotation is one of the challenges of PKI. It requires determining the **cryptoperiod** of a key (how long it should be valid for); a given key can generally only encrypt or sign so much data before it must be replaced (so that it doesn't repeat messages, for example). In the case of TLS, many organisations are using certificates with short lifetimes. This means that if a key is compromised and revocation isn't effective, the damage will be limited. Key rotation problems can also act as DoS attack: if the rotation is botched, it can leave the system unusable until fixed.

Key revocation is part of the key rotation problem: how can a key be marked as compromised or lost? It turns out that marking the key this way is the easy part, but letting others know is not. There are a few approaches to

this in TLS: certificate revocation lists, which contain a list of revoked keys; OCSP (the online certificate status protocol), which provides a means of querying an authoritative source as to whether a key is valid; TACK and certificate transparency, which have yet to see large scale adoption. Both CRLs and OCSP are problematic: what if a key compromise is combined with a DDoS against the CRL or OCSP server? Users may not see that a key was revoked. Some choose to refuse to accept a certificate if the OCSP server can't be reached. What happens in the case of a normal network outage? CRLs are published generally on set schedules, and users have to request the CRL every so often to update it. How often should they check? Even if they check every hour, that leaves up to an hour window in which a compromised key might still be trusted.

Due to these concerns and the difficulty in providing a useful public key infrastructure, PKI tends to be a dirty word in the security and cryptographic communities.

What cryptography does not provide

Thought the encryption methods we'll discuss provide strong security guarantees, none provide any sort of message length obscurity; depending on this system, this may make the plaintext predictable even in the case of strong security guarantees. There's also nothing in encryption that hides when a message is sent if an attacker is monitoring the communications channel. Many cryptosystems also do not hide who is communicating; many times this is evident just from watching the communications channel (such as tracking IP addresses). By itself, cryptography will not provide strong anonymity, but it might serve as part of a building block in such a system. This sort of communications channel monitoring is known as traffic analysis, and defeating it is challenging.

Also, despite the unforgeability guarantees that we'll provide, cryptography won't do anything to prevent replay attacks. Replay attacks are similar to spoofing attacks, in which an attacker captures previously sent messages and replays them. An example would be recording a financial transaction, and replaying this transaction to steal money. Message numbers are how we will approach this problem; a system should never repeat messages, and repeated messages should be dropped by the system. That is something that will need to be handled by the system, and isn't solved by cryptography.

Data lifetimes

In this book, when we send encrypted messages, we prefer to do so using ephemeral message keys that are erased when communication is complete. This means that a message can't be decrypted later using the same key it was encrypted with; while this is good for security, it means the burden of figuring out how to store messages (if that is a requirement) falls on the system. Some systems, such as Pond (https://pond.imperialviolet.org/), enforce a week lifetime for messages. This forced erasure of messages is considered the social norm; such factors will have to play into decisions about how to store decrypted messages and how long to store them.

There's also the transition between data storage and message traffic: message traffic is encrypted with ephemeral keys that are never stored, while stored data needs to be encrypted with a long-term key. The system architecture should account for these different types of cryptography, and ensure that stored data is protected appropriately.

Options, knobs, and dials

The more options a system has for picking the cryptography it uses, the greater the opportunity for making a cryptographic mistake. For this reason, we'll avoid doing this here. Typically, we'll prefer NaCl cryptography, which has a simple interface without any options, and it is efficient and secure. When designing a system, it helps to make a well-informed opinionated choice of the cryptography used. The property of **cryptographic agility**, or being able to switch the cryptography out, may be useful in recovering from a suspected failure. However, it may be prudent to step back and consider why the failure happened and incorporate that into future revisions.

Compatibility

The quality of cryptographic implementations ranges wildly. The fact that there is a good implementation of something in Go doesn't mean that a good implementation will exist in the language used to build other parts of the system. This must factor into the system's design: how easy is it to integrate with other components, or to build a client library in another language? Fortunately, NaCl is widely available; this is another reason why we will prefer it.

Conclusion

Cryptography is often seen as the fun part of building secure systems; however, there are a lot of other work that needs to be done before cryptography enters the picture. It's not a one-size-fits-all security solution; we can't just sprinkle some magic crypto dust on an insecure system and make it suddenly secure. We also have to make sure to understand our problems, and ensure that the problems we are trying to solve are actually the right problems to solve with cryptography. The challenges of building secure systems are even more difficult in virtual environments. It is crucial that a security model be part of the specification, and that proper software engineering techniques are observed to ensure the correctness of the system. Remember, one flaw, and the whole system can come crashing down. While it may not be the end of the world as we know it, it can cause signficant embarassment and financial costs—not only to you, but to the users of the system.

Further reading

1. [Anders08] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, Second Edition*. Wiley, April 2008.
2. [Brum03] D. Brumley, D. Boneh. "Remote timing attacks are practical." In Proceedings of the 12th USENIX Security Symposium, 2003.
3. [Ferg10] N. Ferguson, B. Schneier, T. Kohno. *Cryptography Engineering*. Wiley, March 2010.
4. [Graff03] M. G. Graff, K. R. van Wyk. *Secure Coding: Principles and Practices*. OReily Media, June 2003.
5. [Perc14] C. Percival. "How to zero a buffer." http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html 2014-09-04.
6. [Ptac14] T. Ptacek. "How to safely generate a random number." http://sockpuppet.org/blog/2014/02/25/safely-generate-random-numbers/ 2014-02-25.
7. [Viega03] J. Viega and M. Messier. *Secure Programming Cookbook for C and C++*. OReilly Media, July 2003.

Chapter 3: Symmetric Security

Symmetric cryptography is the simplest form of cryptography: all parties share the same key. It also tends to be the fastest type of cryptography. Fundamentally, the key used by a symmetric algorithm is a sequence of bytes that are used as the input to a transformation algorithm that operates on bits. Key distribution with symmetric cryptography is more difficult than with asymmetric cryptography as the transmission of sensitive key material requires a secure channel. In the next chapter, we'll look at means of exchanging keys.

There are two components to symmetric encryption: the algorithm that provides confidentiality (which is a block or stream cipher), and the component that provides integrity and authenticity (the MAC algorithm). Most ciphers do not provide both in the same algorithm, but those that do are called Authenticated Encryption (AE), or Authenticated Encryption with Additional Data (AEAD), ciphers. In this chapter, we'll consider four **ciphersuites**: NaCl, AES-GCM, AES-CTR with an HMAC, and AES-CBC with an HMAC; a ciphersuite is a selection of algorithms that we'll use to provide security. The code from this chapter can be found in the `chapter3` package in the example code.

Indistinguishability

One of the properties of a secure encryption system is that it provides **indistinguishability**. There are two particular kinds of indistinguishability that are relevant here: IND-CPA, or indistinguishability under a chosen plaintext attack, and IND-CCA, or indistinguishability under a chosen ciphertext attack.

In IND-CPA, the attacker sends a pair of messages that are the same length to the server to be encrypted. The server chooses one of the messages, encrypts it, and sends back the ciphertext. The attacker should not be able to determine which message was encrypted. This property maintains confidentiality. It's useful to consider the requirement that messages be the same length: the length of an encrypted message is related to the length of the original message in most ciphers. That is, encrypting a message does not hide its length.

In IND-CCA, the attacker submits ciphertexts of its own choosing that the server decrypts. After some observations, the attacker submits two challenge ciphertexts and the server picks one at random to decrypt and send back to the attacker. The attacker should not be able to distinguish which ciphertext the plaintext corresponds to. This attack is often seen in symmetric security as a padding oracle attack, in which the encryption scheme used does not include a message authentication code (such as AES-CBC without an HMAC), and can allow the attacker to recover the key used for encryption. There are two variants of IND-CCA; the first (IND-CCA1) means that an attacker cannot submit new ciphertexts after the challenge is sent. The second (IND-CCA2, or adaptive CCA) allows the attacker to continue submitting ciphertexts after the challenge. This may seem like a trivial difference, but a system that is IND-CCA1 secure but not IND-CCA2 secure enables the padding oracle attack.

Both the confidentiality component (such as AES-CBC) and the integrity and authenticity component (such as HMAC-SHA-256) are required for security.

Another indistinguishability requirement that is desirable is that our key material be indistinguishable from random, specifically from the uniform distribution.

Authenticity and integrity

Why are the integrity and authenticity components required? One common thing that crops up when people try to build a secure system is that only the confidentiality aspect is used: the programmer will use just AES with some non-authenticated mode (like the CBC, OFB, or CTR modes). AES ciphertexts are malleable: they can be modified and the receiver is often none the wiser. In the context of an encrypted instant message or email or web page, it might seem this modification would become obvious. However, an attacker can exploit the different responses between invalid ciphertext (when decryption fails) and invalid message (the plaintext is wrong) to extract the key. Perhaps the most well-known such attack is the padding oracle attack. In other cases, the invalid plaintext might be used to exploit bugs in the message handler. This is especially problematic in systems that use encryption to secure messages intended for automated systems.

Appending an HMAC or using an authenticated mode (like GCM) requires that the attacker prove they have the key used to authenticate the message. Rejecting a message that fails the MAC reduces the possibility of an invalid message. It also means that an attacker that is just sending invalid data has their message dropped before even wasting processor time on decryption.

To effectively authenticate with an HMAC, the HMAC key should be a different key than the AES key. In this book, we use HMACs with AES, and we'll append the HMAC key to the AES key for the full encryption or decryption key. For AES-256 in CBC or CTR mode with an HMAC-SHA-256, that means 32 bytes of AES key and 32 bytes of HMAC key for a total key size of 64 bytes; the choice of HMAC-SHA-256 is clarified in a later section.

There are several choices for how to apply a MAC. The right answer is to encrypt-then-MAC:

1. Encrypt-and-MAC: in this case, we would apply a MAC to the plaintext, then send the encrypted plaintext and MAC. In order to verify the MAC, the receiver has to decrypt the message; this still permits an attacker to submit modified ciphertexts with the same problems described earlier. This presents a surface for IND-CCA attacks.
2. MAC-then-encrypt: a MAC is applied and appended to the plaintext, and both are encrypted. Note that the receiver still has to decrypt the message, and the MAC can be modified by modifying the resulting ciphertext, which is a surface for IND-CCA attacks as well.
3. Encrypt-then-MAC: encrypt the message and append a MAC of the ciphertext. The receiver verifies the MAC, and does not proceed to decrypt if the MAC is invalid. This removes the IND-CCA surface.

Moxie Marlinspike's Cryptographic Doom Principle [Moxie11] contains a more thorough discussion of this.

Always use either an authenticated mode (like GCM), or encrypt-then-MAC.

NaCl

NaCl[1] is the Networking and Cryptography library that has a symmetric library (secretbox) and an asymmetric library (box), and was designed by Daniel J. Bernstein. The additional Go cryptography packages contain an implementation of NaCl. It uses a 32-byte key and 24-byte nonces. A nonce is a number used once:

[1] https://nacl.cr.yp.to/

a nonce should never be reused in a set of messages encrypted with the same key, or else a compromise may occur. In some cases, a randomly generated nonce is suitable. In other cases, it will be part of a stateful system; perhaps it is a message counter or sequence number.

The secretbox system uses a stream cipher called "XSalsa20" to provide confidentiality, and a MAC called "Poly1305". The package uses the data types `*[32]byte` for the key and `*[24]byte` for the nonce. Working with these data types may be a bit unfamiliar; the code below demonstrates generating a random key and a random nonce and how to interoperate with functions that expect a `[]byte`.

```
const (
        KeySize   = 32
        NonceSize = 24
)

// GenerateKey creates a new random secret key.
func GenerateKey() (*[KeySize]byte, error) {
            key := new([KeySize]byte)
            _, err := io.ReadFull(rand.Reader, key[:])
            if err != nil {
                    return nil, err
            }

            return key, nil
}

// GenerateNonce creates a new random nonce.
func GenerateNonce() (*[NonceSize]byte, error) {
            nonce := new([NonceSize]byte)
            _, err := io.ReadFull(rand.Reader, nonce[:])
            if err != nil {
                    return nil, err
            }

            return nonce, nil
}
```

NaCl uses the term **seal** to mean securing a message (such that it now is confidential, and that its integrity and authenticity may be verified), and **open** to mean recovering a message (verifying its integrity and authenticity, and decrypting the message).

In this code, randomly generated nonces will be used; in the key exchange chapter, this choice will be clarified. Notably, the selected key exchange methods will permit randomly chosen nonces as secure means of obtaining a nonce. In other use cases, this may not be the case! The recipient will need some way of recovering the nonce, so it will be prepended to the message. If another means of getting a nonce is used, there might be a different way to ensure the recipient has the same nonce used to seal the message.

```
var (
        ErrEncrypt = errors.New("secret: encryption failed")
        ErrDecrypt = errors.New("secret: decryption failed")
)

// Encrypt generates a random nonce and encrypts the input using
// NaCl's secretbox package. The nonce is prepended to the ciphertext.
// A sealed message will the same size as the original message plus
// secretbox.Overhead bytes long.
func Encrypt(key *[KeySize]byte, message []byte) ([]byte, error) {
            nonce, err := GenerateNonce()
            if err != nil {
                    return nil, ErrEncrypt
            }

            out := make([]byte, len(nonce))
            copy(out, nonce[:])
            out = secretbox.Seal(out, message, nonce, key)
            return out, nil
}
```

Decryption expects that the message contains a prepended nonce, and we verify this assumption by checking the length of the message. A message that is too short to be a valid encryption message is dropped right away.

```
// Decrypt extracts the nonce from the ciphertext, and attempts to
// decrypt with NaCl's secretbox.
func Decrypt(key *[KeySize]byte, message []byte) ([]byte, error) {
            if len(message) < (NonceSize + secretbox.Overhead) {
                    return nil, ErrDecrypt
            }

            var nonce [NonceSize]byte
            copy(nonce[:], message[:NonceSize])
            out, ok := secretbox.Open(nil, message[NonceSize:], &nonce, key)
            if !ok {
                    return nil, ErrDecrypt
            }

            return out, nil
}
```

Keep in mind that random nonces are not always the right choice. We'll talk more about this in a chapter on key exchanges, where we'll talk about how we actually get and share the keys that we're using.

Behind the scenes, NaCl will encrypt a message, then apply a MAC algorithm to this ciphertext to get the

final message. This procedure of "encrypt-then-MAC" is how to properly combine an encryption cipher and a MAC.

AES-GCM

If AES is required or chosen, AES-GCM is often the best choice; it pairs the AES block cipher with the GCM block cipher mode. It is an AEAD cipher: authenticated encryption with additional data. It encrypts some data, which will be authenticated along with some optional additional data that is not encrypted. The key length is 16 bytes for AES-128, 24 bytes for AES-192, or 32 bytes for AES-256. It also takes a nonce as input, and the same caveats apply to the nonce selection here. Another caveat is that GCM is difficult to implement properly, so it is important to vet the quality of the packages that may be used in a system using AES-GCM.

Which key size should you choose? That depends on the application. Generally, if there's a specification, use the key size indicated. Cryptography Engineering ([Ferg10]) recommends using 256-bit keys; that's what we'll use here. Again, the security model for your system should dictate these parameters. In the AES examples in this chapter, changing the key size to 16 will suffice to switch to AES-128 (and 24 for AES-192). The nonce size does not change across the three versions.

Unlike most block cipher modes, GCM provides authentication. It also allows the for the authentication of some additional, unencrypted data along with the ciphertext. Given that it is an AEAD mode (which provides integrity and authenticity), an HMAC does not need to be appended for this mode.

The `AEAD` type in the `crypto/cipher` package uses the same "open" and "seal" terms as NaCl. The AES-GCM analogue of the NaCl encryption above would be:

```
1  // Encrypt secures a message using AES-GCM.
2  func Encrypt(key, message []byte) ([]byte, error) {
3          c, err := aes.NewCipher(key)
4          if err != nil {
5                  return nil, ErrEncrypt
6          }
7
8          gcm, err := cipher.NewGCM(c)
9          if err != nil {
10                 return nil, ErrEncrypt
11         }
12
13         nonce, err := GenerateNonce()
14         if err != nil {
15                 return nil, ErrEncrypt
16         }
17
18         // Seal will append the output to the first argument; the usage
19         // here appends the ciphertext to the nonce. The final parameter
20         // is any additional data to be authenticated.
21         out := gcm.Seal(nonce, nonce, message, nil)
```

```
22          return out, nil
23  }
```

This version does not provide any additional (unencrypted but authenticated) data in the ciphertext.

Perhaps there is a system in which the message is prefixed with a 32-bit sender ID, which allows the receiver to select the appropriate decryption key. The following example will authenticate this sender ID:

```
1   // EncryptWithID secures a message and prepends a 4-byte sender ID
2   // to the message.
3   func EncryptWithID(key, message []byte, sender uint32) ([]byte, error) {
4           buf := make([]byte, 4)
5           binary.BigEndian.PutUint32(buf, sender)
6
7           c, err := aes.NewCipher(key)
8           if err != nil {
9                   return nil, ErrEncrypt
10          }
11
12          gcm, err := cipher.NewGCM(c)
13          if err != nil {
14                  return nil, ErrEncrypt
15          }
16
17          nonce, err := GenerateNonce()
18          if err != nil {
19                  return nil, ErrEncrypt
20          }
21
22          buf = append(buf, nonce)
23          buf := gcm.Seal(buf, nonce, message, message[:4])
24          return buf, nil
25  }
```

In order to decrypt the message, the receiver will need to provide the appropriate sender ID as well. As before, we check some basic assumptions about the length of the message first, accounting for the prepended message ID and nonce.

```
func DecryptWithID(message []byte) ([]byte, error) {
        if len(message) <= NonceSize+4 {
                return nil, ErrDecrypt
        }

    // SelectKeyForID is a mock call to a database or key cache.
        id := binary.BigEndian.Uint32(message[:4])
        key, ok := SelectKeyForID(id)
        if !ok {
                return nil, ErrDecrypt
        }

        c, err := aes.NewCipher(key)
        if err != nil {
                return nil, ErrDecrypt
        }

        gcm, err := cipher.NewGCM(c)
        if err != nil {
                return nil, ErrDecrypt
        }

        nonce := make([]byte, NonceSize)
        copy(nonce, message[4:])

        // Decrypt the message, using the sender ID as the additional
        // data requiring authentication.
        out, err := gcm.Open(nil, nonce, message[4+NonceSize:], message[:4])
        if err != nil {
                return nil, ErrDecrypt
        }
        return out, nil
}
```

If the message header is altered at all, even if the new sender ID returns the same key, the message will fail to decrypt: any alteration to the additional data results in a decryption failure.

AES-CTR with HMAC

The last options you should consider, if you have a choice, are AES-CTR and AES-CBC with an HMAC. In these ciphersuites, data is first encrypted with AES in the appropriate mode, then an HMAC is appended. In this book, we assume the use of these ciphersuites only when required as part of a specification or for compatibility.

CTR also uses a nonce; again, the nonce must be only ever used once with the same key. Reusing a nonce can be catastrophic, and will leak information about the message; the system will now fail the indistinguishability

requirements and therefore becomes insecure. If there is any question as to whether a nonce is unique, a random nonce should be generated. If this is being used for compatibility with an existing system, you'll need to consider how that system handles nonces.

If you're using AES-CTR, you're probably following along with some sort of specification that should specify which HMAC construction to use. The general rule of thumb from the FIPS guidelines is HMAC-SHA-256 for AES-128 and HMAC-SHA-384 for AES-256; Cryptography Engineering ([Ferg10]) and [Perc09] recommend HMAC-SHA-256. We'll use HMAC-SHA-256 with AES-256.

Here, we'll encrypt by selecting a random nonce, encrypting the data, and computing the MAC for the ciphertext. The nonce will be prepended to the message and the MAC appended. The message will be encrypted in-place. The key is expected to be the HMAC key appended to the AES key.

```
const (
        NonceSize = aes.BlockSize
        MACSize = 32 // Output size of HMAC-SHA-256
        CKeySize = 32 // Cipher key size - AES-256
        MKeySize = 32 // HMAC key size - HMAC-SHA-256
)

var KeySize = CKeySize + MKeySize

func Encrypt(key, message []byte) ([]byte, error) {
        if len(key) != KeySize {
                return nil, ErrEncrypt
        }

        nonce, err := util.RandBytes(NonceSize)
        if err != nil {
                return nil, ErrEncrypt
        }

        ct := make([]byte, len(message))

        // NewCipher only returns an error with an invalid key size,
        // but the key size was checked at the beginning of the function.
        c, _ := aes.NewCipher(key[:CKeySize])
        ctr := cipher.NewCTR(c, nonce)
        ctr.XORKeyStream(ct, message)

        h := hmac.New(sha256.New, key[CKeySize:])
        ct = append(nonce, ct...)
        h.Write(ct)
        ct = h.Sum(ct)
        return ct, nil
}
```

In order to decrypt, the message length is checked to make sure it has a nonce, MAC, and a non-zero message size. Then, the MAC is checked. If it's valid, the message is decrypted.

```
func Decrypt(key, message []byte) ([]byte, error) {
        if len(key) != KeySize {
                return nil, ErrDecrypt
        }

        if len(message) <= (NonceSize + MACSize) {
                return nil, ErrDecrypt
        }

        macStart := len(message) - MACSize
        tag := message[macStart:]
        out := make([]byte, macStart-NonceSize)
        message = message[:macStart]

        h := hmac.New(sha256.New, key[CKeySize:])
        h.Write(message)
        mac := h.Sum(nil)
        if !hmac.Equal(mac, tag) {
                return nil, ErrDecrypt
        }

        c, _ := aes.NewCipher(key[:CKeySize])
        ctr := cipher.NewCTR(c, message[:NonceSize])
        ctr.XORKeyStream(out, message[NonceSize:])
        return out, nil
}
```

AES-CBC

The previous modes mask the underlying nature of the block cipher: AES operates on blocks of data, and a full block is needed to encrypt or decrypt. The previous modes act as stream ciphers, where messages lengths do not need to be a multiple of the block size. CBC, however, does not act in this way, and requires messages be padded to the appropriate length. CBC also does not use nonces in the same way.

In CBC mode, each block of ciphertext is XOR'd with the previous block. This leads to the question of what the first block is XOR'd with. In CBC, we use a sort of dummy block called an initialisation vector. It may be randomly generated, which is often the right choice. We also noted that with the other encryption schemes, it was possible to use a message or sequence number as the IV: such numbers should not be directly used with CBC. They should be encrypted (using AES-ECB) with a **separate** IV encryption key. An IV should never be reused with the same message and key.

The standard padding scheme used is the PKCS #7 padding scheme. We pad the remaining bytes with a byte containing the number of bytes of padding: if we have to add three bytes of padding, we'll append `0x03 0x03`

`0x03` to the end of our plaintext.

```
func pad(in []byte) []byte {
        padding := 16 - (len(in) % 16)
        for i := 0; i < padding; i++ {
                in = append(in, byte(padding))
        }
        return in
}
```

When we unpad, we'll take the last byte, check to see if it makes sense (does it indicate padding longer than the message? Is the padding more than a block length?), and then make sure that all the padding bytes are present. Always check your assumptions about the message before accepting the message. Once that's done, we strip the padding characters and return the plain text.

```
func unpad(in []byte) []byte {
        if len(in) == 0 {
                return nil
        }

        padding := in[len(in)-1]
        if int(padding) > len(in) || padding > aes.BlockSize {
                return nil
        } else if padding == 0 {
                return nil
        }

        for i := len(in) - 1; i > len(in)-int(padding)-1; i-- {
                if in[i] != padding {
                        return nil
                }
        }
        return in[:len(in)-int(padding)]
}
```

The padding takes place outside of encryption: we pad before encrypting data and unpad after decrypting.

Encryption is done by padding the message and generating a random IV.

```
func Encrypt(key, message []byte) ([]byte, error) {
        if len(key) != KeySize {
                return nil, ErrEncrypt
        }

        iv, err := util.RandBytes(NonceSize)
        if err != nil {
                return nil, ErrEncrypt
        }

        pmessage := pad(message)
        ct := make([]byte, len(pmessage))

        // NewCipher only returns an error with an invalid key size,
        // but the key size was checked at the beginning of the function.
        c, _ := aes.NewCipher(key[:CKeySize])
        ctr := cipher.NewCBCEncrypter(c, iv)
        ctr.CryptBlocks(ct, pmessage)

        h := hmac.New(sha256.New, key[CKeySize:])
        ct = append(iv, ct...)
        h.Write(ct)
        ct = h.Sum(ct)
        return ct, nil
}
```

Encryption proceeds much as with CTR mode, with the addition of message padding.

In decryption, we validate two of our assumptions:

1. The message length should be a multiple of the AES block size (which is 16). HMAC-SHA-256 produces a 32-byte MAC, which is also a multiple of the block size; we can check the length of the entire message and not try to check only the ciphertext. A message that isn't a multiple of the block size wasn't padded prior to encryption, and therefore is an invalid message.
2. The message must be at least four blocks long: one block for the IV, one block for the message, and two blocks for the HMAC. If an HMAC function is used with a larger output size, this assumption will need to be revisited.

The decryption also checks the HMAC before actually decrypting the message, and verifies that the plaintext was properly padded.

```
func Decrypt(key, message []byte) ([]byte, error) {
        if len(key) != KeySize {
                return nil, ErrEncrypt
        }

        // HMAC-SHA-256 returns a MAC that is also a multiple of the
        // block size.
        if (len(message) % aes.BlockSize) != 0 {
                return nil, ErrDecrypt
        }

        // A message must have at least an IV block, a message block,
        // and two blocks of HMAC.
        if len(message) < (4 * aes.BlockSize) {
                return nil, ErrDecrypt
        }

        macStart := len(message) - MACSize
        tag := message[macStart:]
        out := make([]byte, macStart-NonceSize)
        message = message[:macStart]

        h := hmac.New(sha256.New, key[CKeySize:])
        h.Write(message)
        mac := h.Sum(nil)
        if !hmac.Equal(mac, tag) {
                return nil, ErrDecrypt
        }

        // NewCipher only returns an error with an invalid key size,
        // but the key size was checked at the beginning of the function.
        c, _ := aes.NewCipher(key[:CKeySize])
        ctr := cipher.NewCBCDecrypter(c, message[:NonceSize])
        ctr.CryptBlocks(out, message[NonceSize:])

        pt := unpad(out)
        if pt == nil {
                return nil, ErrDecrypt
        }

        return pt, nil
}
```

Messages v. streams

In this book, we operate on **messages**: discrete-sized chunks of data. Processing streams of data is more difficult due to the authenticity requirement. How do you supply authentication information? Let's think about encrypting a stream, trying to provide the same security properties we've employed in this chapter.

We can't encrypt-then-MAC: by it's nature, we usually don't know the size of a stream. We can't send the MAC after the stream is complete, as that usually is indicated by the stream being closed. We can't decrypt a stream on the fly, because we have to see the entire ciphertext in order to check the MAC. Attempting to secure a stream adds enormous complexity to the problem, with no good answers. The solution is to break the stream into discrete chunks, and treat them as messages. Unfortunately, this means we can't encrypt or decrypt `io.Readers` and `io.Writers` easily, and must operate on `[]byte` messages. Dropping the MAC is simply not an option.

Conclusions

In this chapter, we've elided discussion about how we actually get the keys (usually, generating a random key isn't useful). This is a large enough topic to warrant discussion in a chapter of its own.

Some key points:

1. Prefer NaCl where you can. Use AES-GCM if AES is required and you have a choice. Use AES-CBC and AES-CTR for compatibility.
2. Always encrypt-then-MAC. Don't ever just encrypt.
3. Always check assumptions about the message, including its authenticity, before decrypting the message.
4. Think about how you're getting nonces and IVs, and whether it's the appropriate method.

Further reading

1. [Ferg10] N. Ferguson, B. Schneier, T. Kohno. *Cryptography Engineering*. Wiley, March 2010.
2. [Moxie11] Moxie Marlinspike. "The Cryptographic Doom Principle." http://www.thoughtcrime.org/blog/the-cryptographic-doom-principle/
3. [Perc09] C. Percival. "Cryptographic Right Answers." http://www.daemonology.net/blog/2009-06-11-cryptographic-right-answers.html, 2009-06-11.
4. [Rizzo10] J. Rizzo, T. Duong. "Practical padding oracle attacks." In Proceedings of the 4th USENIX conference on Offensive technologies (WOOT'10). USENIX Association, Berkeley, CA, USA, 1-8. 2010.
5. [Vaud02] S. Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...." In Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT '02), Lars
6. Knudsen (Ed.). Springer-Verlag, London, UK, UK, 534-546. 2002.

Chapter 4: Secure Channels and Key Exchange

In the previous chapter, we looked at symmetric security. While fast, symmetric cryptography requires both sides to share the same key. However, it's assumed that if encryption is required, the communication medium is insecure. In this chapter, we'll look at some mechanisms for exchanging keys over an insecure channel.

Secure channel

The goal of this chapter is to set up a **secure channel** between two peers: one in which an attacker cannot eavesdrop or forge messages. Typically, we'll use symmetric security to provide a secure channel, and some key exchange mechanism to set up the initial keys. This channel will be established in two directions, which means that there should be a separate set of keys (the combined encryption and authentication keys) for communications in both directions.

What does the secure channel look like? It'll be message-oriented, for the same reasons described in the previous chapter. It will be built on top of an insecure channel, like the Internet, that can't be trusted. An attacker might be able to disrupt or manipulate this insecure channel, for example; we also can't hide how large messages are or who is talking. Ideally, messages will not be replayable or have their order changed (and that it will be apparent when either occurs).

The easiest way to prevent a replay is to keep track of message numbers; this number might be serialised as part of the message. For example, a message might be considered as the pairing of a message number and some message contents. We'll track numbers for both sent and received messages.

```
type Message struct {
        Number   uint32
        Contents []byte
}
```

Serialising the message appends the contents to the 4-byte message number. The `out` variable is initialised with only four bytes, but with a capacity that accounts for the message contents.

```
func MarshalMessage(m Message) []byte {
        out := make([]byte, 4, len(m.Contents) + 4)
        binary.BigEndian.PutUint32(out[:4], m.Number)
        return append(out, m.Contents...)
}
```

Unmarshaling a message first checks the assumption that the message contains a sequence number and at least one byte of contents. Then, the message number and contents are extracted.

```
func UnmarshalMessage(in []byte) (Message, bool) {
        m := Message{}
        if len(in) <= 4 {
                return m, false
        }

        m.Number = binary.BigEndian.Uint32(in[:4])
        m.Contents = in[4:]
        return m, true
}
```

Including message numbers is only useful if they're being checked. We'll keep track of message numbers for a given session: as we'll see in this chapter, we exchange unique keys for each session, so we only track message numbers for the scope of a session. A message replayed outside of a session won't be decryptable, so we won't worry about it. We'll also want to keep track of the message number of messages we've sent and messages we've received separately. We also keep separate keys for each direction.

```
type Channel io.ReadWriter

type Session struct {
        lastSent uint32
        sendKey *[32]byte

        lastRecv uint32
        recvKey *[32]byte

        Channel Channel
}

func (s *Session) LastSent() uint32 {
        return s.lastSent
}

func (s *Session) LastRecv() uint32 {
        return s.lastRecv
}
```

When we encrypt a message as part of a session, we'll set the message number to the last sent message number plus one. The serialised message number and contents are then encrypted, and handed off to be sent.

```
func (s *Session) Encrypt(message []byte) ([]byte, error) {
        if len(message) == 0 {
                return nil, secret.ErrEncrypt
        }

        s.lastSent++
        m := MarshalMessage(Message{s.lastSent, message})
        return secret.Encrypt(s.sendKey, m)
}
```

Now, ensuring that we have an incremented message number is a requirement for decryption. If the message number hasn't incremented, we assume it's a replayed message, and considered it a decryption failure.

```
func (s *Session) Decrypt(message []byte) ([]byte, error) {
        out, err := secret.Decrypt(s.recvKey, message)
        if err != nil {
                return nil, err
        }

        m, ok := UnmarshalMessage(out)
        if !ok {
                return nil, secret.ErrDecrypt
        }

        if m.Number <= s.lastRecv {
                return nil, secret.ErrDecrypt
        }

        s.lastRecv = m.Number

        return m.Contents, nil
}
```

This example could be trivially extended to include other message metadata; using an AEAD like GCM means that this metadata doesn't need to be encrypted. However, we'll prefer to encrypt as much as we can to limit the amount of information about messages to an eavesdropper.

A more complete example of sessions can be found in the example source code in the `chapter4/session/` package.

Password-based key derivation

The simplest mechanism for exchanging keys is to use a password to derive the key. There are a few choices for doing this, but the one we'll prefer is scrypt; it is provided in the `golang.org/x/crypto/scrypt` package.

Internally, scrypt uses another key derivation function (KDF), called PBKDF2, but it increases the resource requirements. This requires an implementation to choose between using a lot of memory or a lot of CPU in an attempt to reduce the effectiveness of hardware implementations, making them more expensive to produce.

The resource requirements of scrypt are controlled by its parameters:

- `N`: the number of iterations. In the original presentation[Perc09b] for the algorithm, 16384 (2^{14}) is recommended for interactive logins and 1048576 (2^{20}) for file encryption. In this book, we use 2^{20} as a default for securing cryptographic secrets; a one-time initial cost to derive the key is acceptable for the applicatons we'll use it for.
- `r`: relative memory cost parameter (controls the blocksize in the underlying hash). The original presentation recommends a value of 8.
- `p`: relative CPU cost parameter. The original presentation recommends a value of 1.

On the machine I'm writing this on, a 2010 ThinkPad with a 2.67 GHz quad-core Intel Core i5 M560 and 6G of memory, scrypt takes on average 6.3s to derive a key; there isn't an appreciable timing difference between generating 32-byte NaCl or AES-GCM keys and generating 80-byte AES-CBC or AES-CTR keys due to the way that key material is derived.

In order to generate a key, scrypt takes a password and a salt. The salt is not a secret, and we'll prepend it to data encrypted using an scrypt-derived key. The salt is passed to PBKDF2, and is used to prevent an attacker from just storing `(password,key)` pairs as a different salt yields a different output from scrypt. Each password has to be checked with the salt used to derive the key. We'll use randomly-generated 256-bit salts as recommended in [Ferg10].

In the previous chapter, it was mentioned that our key exchange method permitted us to use random nonces. When we use scrypt for key exchange, each encryption uses a new salt. This effectively means that each encryption will use a different key, even though we're using the same passphrase. This means it is astronomically unlikely that we'll reuse a nonce with the same key.

Using scrypt in Go is a one-liner (plus error checking), so let's try encrypting something with NaCl using a passphrase. The first step is to write the key derivation function, which mostly translates the byte slice returned by `scrypt.Key` to the `*[32]byte` used by the `secretbox` package.

```
1  // deriveKey generates a new NaCl key from a passphrase and salt.
2  func deriveKey(pass, salt []byte) (*[secret.KeySize]byte, error) {
3          var naclKey = new([secret.KeySize]byte)
4          key, err := scrypt.Key(pass, salt, 1048576, 8, 1, secret.KeySize)
5          if err != nil {
6                  return nil, err
7          }
8
9          copy(naclKey[:], key)
10         util.Zero(key)
11         return naclKey, nil
12 }
```

The encryption function will take a passphrase and a message, generate a random salt, derive a key, encrypt the message, and prepend the salt to the resulting ciphertext.

```
func Encrypt(pass, message []byte) ([]byte, error) {
        salt, err := util.RandBytes(SaltSize)
        if err != nil {
                return nil, ErrEncrypt
        }

        key, err := deriveKey(pass, salt)
        if err != nil {
                return nil, ErrEncrypt
        }

        out, err := secret.Encrypt(key, message)
        util.Zero(key[:]) // Zero key immediately after
        if err != nil {
                return nil, ErrEncrypt
        }

        out = append(salt, out...)
        return out, nil
}
```

The derived key is only needed in the call to `secret.Encrypt`: it is derived immediately before the call, and it's zeroised immediately after the call. This is an attempt to keep the secret in memory only as long as it's needed. Likewise, the passphrase should be immediately zeroised by the caller after a call to this function. Error handling can wait until after the key has been zeroised, which helps to prevent the possibility of the key material leaking out of this function.

To decrypt, we check our assumption about the length of the message (that is, a properly encrypted passphrase-secured message will have a salt prepended, plus the NaCl nonce and overhead). Then, we derive the key, decrypt, and return the plaintext.

```
const Overhead = SaltSize + secretbox.Overhead + secret.NonceSize

func Decrypt(pass, message []byte) ([]byte, error) {
        if len(message) < Overhead {
                return nil, ErrDecrypt
        }

        key, err := deriveKey(pass, message[:SaltSize])
        if err != nil {
                return nil, ErrDecrypt
        }
```

```
12
13          out, err := secret.Decrypt(key, message[SaltSize:])
14          util.Zero(key[:]) // Zero key immediately after
15          if err != nil {
16                  return nil, ErrDecrypt
17          }
18
19          return out, nil
20  }
```

Once again, an attempt is made to limit the scope of key material in memory. It's no guarantee, but it's the best we can do.

Using passwords is also called a "pre-shared secret"; pre-shared means that the password has to be exchanged over some other secure channel. For example, your wireless network probably uses encryption with a pre-shared secret (your wireless password); this password is used to secure network communications, and you probably tell your friends the password which is fairly secure under the assumption that an attacker targeting your network will not be physically present (unless they have installed a remote-access tool on your machine and is listening on the microphone). Generally, the attackers who can eavesdrop on this verbal key exchange aren't the ones that you are concerned about eavesdropping on your network traffic.

The `chapter4/passcrypt` package in the example code contains an example of password-based encryption.

Asymmetric key exchange: ECDH

Another alternative is to agree on symmetric keys by performing a key exchange using asymmetric keys. In particular, we'll use an asymmetric algorithm based on elliptic curves ([Sull13]) called the Elliptic Curve Diffie-Hellman key agreement protocol, which is a variant on the Diffie-Hellman key exchange for elliptic curves.

An asymmetric encryption algorithm, or public key encryption algorithm, is one in which the key used to encrypt and the key used to decrypt are different. The key that's used to encrypt is made public, and anyone who has this public key can encrypt a message to it. The decryption key is kept private by the holder. In symmetric security, all communicating parties have to share keys, which means `(N * (N - 1)) / 2` keys. With asymmetric keys, only N public keys have to be exchanged.

In an asymmetric key exchange, both sides have a key pair consisting of a private component and a public component. They send each other their public components, even over an insecure channel, and ECDH means that combining your private key and their public key will yield the same symmetric key as combining their private key and your public key. However, ECDH can only be performed between keys that use the same curve.

Generating elliptic curve key pairs is generally a fast operation, and in this book, we'll use **ephemeral** key pairs for sessions: key pairs that are generated at the start of each session, and discarded at least once the session is over, if not earlier. Using ephemeral keys limits the damage of key compromise: an attacker who manages to recover an ephemeral key can only decrypt messages for that session.

Introducing asymmetric cryptography into a system brings a lot of other concerns, as mentioned in the engineering concerns chapter. It comes down to trust, and at some point there needs to be a root of trust

that all participants in a particular key exchange agree to trust. This is not a trivial concern, and should be considered carefully. While it does afford the ability to perform key agreement over an insecure channel without a pre-shared secret, the benefits of this will need to be weighed against the other factors. For example, we'll look at digital signatures in an upcoming chapter, but for now it suffices to note that often a long-term identity key is used to sign session keys to prove their ownership. Now both sides have to determine how they'll get the other's identity key, how they'll know they can trust it, and how they can match the public key to the peer they're talking to. Some method for distributing and trusting public keys has to be determined.

NaCl: Curve25519

In NaCl, generating new key pairs and performing a key exchange is simple, and provided by functions in `golang.org/x/crypto/nacl/box`.

```
pub, priv, err := box.GenerateKey(rand.Reader)
```

We can use the `Session` defined earlier to set up to perform the key exchange. First, a new session is created over a `Channel` and the keys are allocated.

```
func NewSession(ch Channel) *Session {
        return &Session{
                recvKey: new([32]byte),
                sendKey: new([32]byte),
            Channe: ch,
        }
}
```

The `keyExchange` function takes byte slices and fills in the appropriate key material. It also attempts to zeroise the key material after it's no longer needed.

```
func keyExchange(shared *[32]byte, priv, pub []byte) {
        var kexPriv [32]byte
        copy(kexPriv[:], priv)
        util.Zero(priv)

        var kexPub [32]byte
        copy(kexPub[:], pub)

        box.Precompute(shared, &kexPub, &kexPriv)
        util.Zero(kexPriv[:])
}
```

Finally, the key exchange computes the send and receive keys. The `dialer` option should be `true` for the party that initiated the conversation (by dialing the other side).

```
func (s *Session) KeyExchange(priv, peer *[64]byte, dialer bool) {
        if dialer {
                keyExchange(s.sendKey, priv[:32], peer[:32])
                keyExchange(s.recvKey, priv[32:], peer[32:])
        } else {
                keyExchange(s.recvKey, priv[:32], peer[:32])
                keyExchange(s.sendKey, priv[32:], peer[32:])
        }
        s.lastSent = 0
        s.lastRecv = 0
}
```

The `session` example contains a `Dial` and `Listen` function that set up a key exchange over the channel.

The `Precompute` function actually carries out the key exchange; however, the NaCl package can also do the key exchange for each message. This may be less efficient, depending on the application.

```
out := box.Seal(nil, message, nonce, peerPublic, priv)
```

This might be useful for a one-off message. In this case, it's usually useful to generate a new ephemeral key pair for each message that is used for the key exchange, and pack the ephemeral public key at the beginning so the receiver can decrypt. The `chapter4/naclbox` package contains an example of securing messages this way.

In NaCl, all key pairs use the same elliptic curve, Curve25519 ([DJB05]). This makes the interface simple, and simplifies exchanging public keys: they will always be 32 bytes. As long as we're using Curve25519, we won't have to worry about which curve the sender might be using.

NIST curves

The Go standard library has support for the commonly-used NIST curves, of which there are several. In order to communicate using these NIST curves, both sides have to agree on which curve to use. This adds additional overhead to the previously mentioned considerations: now, the system designer has to consider not only where keys come from and how they're trusted, but which type of curve to use. There is serious concern[2] over the provenance of these curves and whether they have been designed with an NSA backdoor; they should be used only when required for compatibility or as part of a specification. They can be generated using

```
priv, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
```

The three curves of interest are P256, P384, and P521 (or secp256r1 / prime256v1, secp384r1, and secp521r1). P521 is believed to have an equivalent security level to AES-256 (notwithstanding questions about the integrity of the curve), so it is appropriate for use with the AES-256 plus HMAC ciphersuites.

[2] https://www.schneier.com/blog/archives/2013/09/the_nsa_is_brea.html#c1675929

Doing this with the Go `crypto/ecdsa` package is more involved. We first have to verify that the curves match, and then perform the scalar multiplication (or point multiplication). The number that is returned is the shared key, but it won't be uniformly random. We do want our shared key to have this indistinguishability property, so we'll compute the SHA-512 digest of this number which produces a 64-byte value: this is large enough to use with AES-256-CBC with HMAC-SHA-256 and AES-256-CTR with HMAC-SHA-256.

```
var ErrKeyExchange = errors.New("key exchange failed")

func ECDH(priv *ecdsa.PrivateKey, pub *ecdsa.PublicKey) ([]byte, error) {
        if prv.PublicKey.Curve != pub.Curve {
                return nil, ErrKeyExchange
        } else if !prv.PublicKey.Curve.IsOnCurve(pub.X, pub.Y) {
                return nil, ErrKeyExchange
        }

        x, _ := pub.Curve.ScalarMult(pub.X, pub.Y, prv.D.Bytes())
        if x == nil {
                return nil, ErrKeyExchange
        }

        shared := sha512.Sum512(x.Bytes())
        return shared[:secret.KeySize], nil
}
```

Notice the check to ensure that the public key is valid: failing to check this can result in exposing the private key ([Antip03]). We can't trust the parser to do this validation for us, so we'll check it here.

The `crypto/ecdsa` keys are used here instead of the ones defined in the `crypto/elliptic` package due to the support for serialising ECDSA keys in the `crypto/x509` package (i.e. using the `MarshalPKIXPublicKey` and `ParsePKIXPublicKey` functions).

```
func ParseECPublicKey(in []byte) (*ecdsa.PublicKey, error) {
        // UnmarshalPKIXPublicKey returns an interface{}.
        pub, err := x509.ParsePKIXPublicKey(in)
        if err != nil {
                return nil, err
        }

        ecpub, ok := pub.(*ecdsa.PublicKey)
        if !ok {
                return nil, errors.New("invalid EC public key")
        }

        return ecpub, nil
}
```

Due to the concerns about the NIST curves, we'll mostly avoid using them in this book (though they are defined for some other systems we'll look at), and will primarily stick to Curve25519.

The `chapter4/nistecdh` package in the example code contains the relevant example.

Other key exchange methods

Password-authenticated key exchanges are another mechanisms for agreeing on a symmetric key that also use a public key. I haven't seen any implementations in Go, so we won't be using them in this book.

Another mechanism is the original Diffie-Hellman algorithm; one implementation is at "github.com/dchest/dhgroup14". We won't use DH except in the context of ECDH in this book, though.

More complex mechanisms such as that found in Kerberos are also used.

We'll look at exchanging keys using an asymmetric algorithm called RSA later, though this won't be our preferred mechanism. Elliptic curve keys can be generated fast enough to make ephemeral keys practical; RSA key generation is much slower and RSA implementations are difficult to get right. The alternative is to use RSA to sign the ephemeral key; this is often used with Diffie-Hellman.

Practical: File encryptor

To put this into practice, try to build a file encryption program that uses passwords to generate the key. First, think about how the program should work: should it encrypt multiple files, or a single file? What does the security model look like? What kinds of platforms does it need to run on? Are there compatibility requirements?

My solution to this is in the filecrypt[3] repository. Your requirements and specification (and therefore security model) may be different, so you may end up with a different solution.

Further reading

1. [Antip03] A. Antipa, D. R. L. Brown, A. Menezes, R. Struik,
2. A. Vanstone. Validation of Elliptic Curve Public Keys, Lecture Notes in Computer Science, Volume 2567 pages 211-223, Springer. 2003.
3. [DJB05] D. J. Bernstein. "Curve25519: new Diffie-Hellman speed records." Proceedings of PKC 2006, to appear. 15 November, 2005.
4. [Ferg10] N. Ferguson, B. Schneier, T. Kohno. *Cryptography Engineering*. Wiley, March 2010.
5. [Hao2008] F. Hao, P. Ryan. "Password Authenticated Key Exchange by Juggling." Proceedings of the 16th International Workshop on Security Protocols, 2008.
6. [Perc09a] C. Percival. "Stronger Key Derivation Via Sequential Memory-Hard Functions." BSDCan'09, May 2009.
7. [Perc09b] C. Percival. "scrypt: A new key derivation function." BSDCan'09, May 2009.

[3] https://github.com/kisom/filecrypt/

8. [Sull13] N. Sullivan. “A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography.” https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/ 24 October 2013.
9. [Wu2000] T. Wu. “The SRP Authentication and Key Exchange System.” RFC 2945, IETF and the Intenret Society, September 2000.

Chapter 5: Digital signatures

Another use for asymmetric algorithms is in digitally signing messages; it is based on the assumptions that only the holder can produce the signature for a given message (i.e. signatures are unforgeable), and that a signature is valid for only a single message. In this chapter, we'll talk about digital signatures with Ed25519, which we'll use with Curve25519 encryption keys; ECDSA, which we'll use with the NIST curves; and RSA. In the algorithms we'll use here, the message isn't signed directly; rather, the hash of the message is signed. Sometimes we'll have to provide the hash of the message ourselves, other times it will be computer for us.

Cryptographic hashing algorithms

We've seen hash functions already: SHA-256 was used as part of the HMAC in the symmetric security chapter, and SHA-512 was used as part of the NIST ECDH. We haven't talked about what they do yet, or when they should be used.

A hash function maps some arbitrary input to a fixed size output. For example, no matter what `m` is, `SHA-256(m)` always produces a 32-byte output. In non-cryptographic uses, this is sometimes called a checksum. In cryptography, there are stricter requirements for hash functions, and we only use cryptographic hash functions that satisfy these requirements. In this book, the term "hash function" implies a cryptographic hash function.

The most basic property is **collision resistance**: for any two messages m^1 and m^2 where m^1 does not equal m^2, the hash of m^1 should not equal m^2. We don't want two inputs to collide and produce the same hash if they're not the same.

We also want our hash functions to have **preimage resistance**. This means that if we have some hash function output `x`, we shouldn't be able to find some `m` that hashes to `x`. A hash function should be one-way: easy to compute the hash from a message, but infeasible to compute the message from a hash.

Another property is that changing `m` should result in a change in `h(m)`. `h(m)` should be indistiguishable from random, which means that an attacker who sees `h(m)` and `h(m')` (where `m'` is any modification, even a single-bit, of `m`) shouldn't be able to determine that they came from related messages.

Considering that a hash function outputs a fixed-size output and accepts arbitrary input, there will be some collisions; we accept computationally infeasible as a requirement. A successful attack must occur in time to exploit that attack. On the other hand, a hash function must be computationally efficient and quickly produce a hash from its input.

The hash functions we use in this book are in the SHA-2 family of hash functions, which are named according to the size of the output they produce: SHA-256 produces 32-byte (256-bit) hashes, SHA-384 produces 48-byte (384-bit) hashes, and so forth. The SHA-1 hash is sometimes specified, but there have been attacks on it that make it unsuitable for security purposes. Note that the attacks on SHA-1 do not extend to HMAC-SHA-1; this construction is still thought to be secure[4]. We'll use the SHA-2 family for consistency, not due to security concerns.

[4] https://www.schneier.com/blog/archives/2005/02/sha1_broken.html

Hashes are often misused; they've been used in an attempt to secure passwords or for authentication. What we'll use them for is to remove structure from some data or to produce an ID from some arbitrary input. In the symmetric security chapter, we used SHA-256 with the HMAC construction to produce an authenticated message ID. In the key exchange chapter, we used SHA-512 to remove some of the structure from the numbers that were produced to make them better suited to use as keys.

The problem with using hashes to secure passwords comes back to the requirement that hashes be efficient. An attacker with a set of hashed passwords can quickly mount attacks aimed at recovering those passwords. In authentication, the way the SHA-2 hashes are constructed leaves them vulnerable to a length extension attack (for an example, see [Duong09]).

Another misuse of hashes is for integrity checks, such as the SHA-256 digest of a file on a download page. This doesn't provide any authenticity, and an attacker who can replace the download file is likely able to replace the digest: there's no guarantee that the digest was provided by a legitimate party. HMACs aren't the right choice either, as anyone with the HMAC key to verify it could produce their own. This is a case where digital signatures are quite useful, so long as the public key is well known. If the public key is just served on the download page and the attacker can swap files on the page, they can swap out the public key.

Forward secrecy

The ECDH key exchange presented us with a problem: how do we trust the public keys we're receiving? Assuming that we've chosen the asymmetric route and have a key distribution and trust mechanism, we have a set of tools for establishing this trust. Our trust mechanism won't be employed for session keys; these last only the lifetime of a session. Instead, we employ long-term identity keys that are then used to sign session keys. At the start of a session, the session keys are signed and the other side verifies the signature.

A key feature of this arrangement is that the session key and the identity key, are unrelated and will always be separate keys in the systems that we build. In the case of NaCl, they'll even be different curves. This follows the security principle of using separate keys for authentication and secrecy. If an attacker is able to compromise the identity key, they won't be able to decrypt any previous sessions using that key. They can use the identity key to sign any future sessions, however.

Identity keys are, by themselves, just numbers. They carry no innate information about the bearer. Using identity keys relies on a well-defined key distribtion plan that maps public keys to identities and provides a mechanism for participants to obtain the appropriate public keys (and, usually, any associated identity metadata). They also need some way to determine how much a key has been trusted. We'll talk about this problem more in a future chapter, and we'll see how some real world systems approach the problem. It is a decidedly unsexy problem, but it is critical to a successful secure system.

Let's consider the scenario where a session encryption key is compromised, and the scenario where an identity key is compromised. In the first case, keys are only in scope for a single session. A compromise here breaks the security of that session, and warrants examining the failure to determine countermeasures and appropriate action. However, the key damage is limited to affected sessions. If an identity key is compromised, the same attention has to be paid. However, there's usually additional overhead in communicating the compromise to other peers, and getting them to mark the compromised key as untrusted. Depending on the key distribution mechanism, replacing this key may be difficult and/or expensive.

Ed25519

Adam Langley has an implementation of Ed25519 on Github[5], which is the implementation we'll use in this book. Ed25519 private keys and signatures are 64 bytes in length, while public keys are 32 bytes. This implementation uses SHA-512 internally to hash the message, and so we pass messages directly to it.

Generating keys is done as with curve25519 keys:

```
pub, priv, err := ed25519.GenerateKey(rand.Reader)n
```

Signatures are made using the `Sign` function and verified with the `Verify` function:

```
sig, err := ed25519.Sign(priv, message)

if !ed25519.Verify(pub, message) {
        // Perform signature verification failure handling.
}
```

We'll prefer Ed25519 signatures in this book: the interface is simple, keys are small, and the algorithm is efficient. Signatures are also **deterministic** and don't rely on randomness for signatures; this means they do not compromise security in the case of a PRNG failure, which we'll talk about more in the section on ECDSA. This property was one of the key motivators for the development of this algorithm.

ECDSA

The elliptic curve digital signature algorithm, or ECDSA, is a signature algorithm that we'll use with the previously mentioned NIST curves. Just as ECDH is the elliptic curve variant of DH, ECDSA is the elliptic curve variant of the original DSA.

There is a serious weakness in DSA (which extends to ECDSA) that has been exploited in several real world systems (including Android Bitcoin wallets and the PS3); the signature algorithm relies on quality randomness (bits that are indistinguishable from random); once the PRNG enters a predictable state, signatures may leak private keys. Systems that use ECDSA must be aware of this issue, and pay particular attention to their PRNG.

ECDSA signatures usually provide a pair of numbers in the signature called `r` and `s`. The most common way to serialise these two numbers is using ASN.1 as defined in [SEC1], and we'll use this when signing with ECDSA.

[5] https://github.com/agl/ed25519

```
type ECDSASignature struct {
        R, S *big.Int
}

func SignMessage(priv *ecdsa.PrivateKey, message []byte) {
        hashed := sha256.Sum256(message)
        r, s, err := ecdsa.Sign(rand.Reader, priv, hashed[:])
        if err != nil {
                return nil, err
        }

        return asn1.Marshal(ECDSASignature{r, s})
}

func VerifyMessage(pub *ecdsa.PublicKey, message []byte, signature []byte) bool {
        var rs ECDSASignature

        if _, err := asn.Unmarshal(signature, &rs); err != nil {
                return false
        }

        hashed := sha256.Sum256(message)
        return ecdsa.Verify(pub, hashed[:], rs.R, rs.S)
}
```

RSA

RSA is a different type of asymmetric algorithm than the ones we've seen so far; it isn't based on elliptic curves and it is has historically been more weidely used than elliptic curve cryptography. Notably, TLS and PGP both make heavy use of RSA, although elliptic curve cryptography is increasingly used more.

There are two signature schemes for RSA defined in the PKCS #1 standard ([PKCS1]): PKCS #1 v1.5 and the Probabilistic Signature Scheme. These are both signature schemes with appendix, where the signature and message are separate; typically, the signature is appended to the message. In contrast, there are some signature schemes (though not defined for RSA) with message recovery, where the signature verification step produces the original message as a byproduct of the verification.

There have been many vulnerabilities found in various implementations of RSA signatures; the most recent of which is BERserk[6]. Even if the Go implementation is well-written, the libraries used in other components may not be. Go also makes some other proper choices, such as picking an acceptable public exponent. The RSA encryption situation is even bleaker. We'll prefer to avoid RSA where we can; there's a lot that can go wrong with it, and it's difficult to get right. It will be something we use when we need compatibility (like with TLS); in this case, we'll be following a specification. We'll also prefer PSS in applications that are written entirely in Go for its stronger security proof ([Lind06], [Jons01]).

[6] http://www.intelsecurity.com/resources/wp-berserk-analysis-part-1.pdf

RSA signatures with PKCS #1 v1.5 are done using the `SignPKCS1v15` function. Likewise, PSS signatures are performed using `SignPSS`. In both cases, the message must be hashed before sending it to the function. The `SignPSS` function also takes an `rsa/PSSOptions` value; this should be defined in your programs as

```
var opts = &rsa.PSSOptions{}
```

You shouldn't change any of the `opts` values unless specifically told to do so by a specification.

The following are example functions that sign using SHA-256 as the hash function.

```
func SignPKCS1v15(priv *rsa.PrivateKey, message []byte) ([]byte, error) {
        hashed := sha256.Sum256(message)
        return rsa.SignPKCS1v15(rand.Reader, priv, crypto.SHA256, hashed[:])
}

func SignPSS(priv *rsa.PrivateKey, message []byte) ([]byte, error) {
        hash := sha256.Sum256(message)
        return rsa.SignPSS(rand.Reader, priv, crypto.SHA256, hashed[:], opts)
}
```

Conclusions

Digital signatures provide a useful mechanism in certain scenarios for providing authenticity and integrity, but their use should be weighed against the resulting requirement for public key infrastructure. We'll prefer Ed25519 as part of the NaCl suite for its robustness against PRNG failures, its simplicity, security, and efficiency. The quality of any libraries used in other languages that are used to build components of the system should also be checked, as they are difficult to get right.

Practical: Sessions with identities

Extend the session example from the previous example to sign the session keys. Keep in mind the need to consider the mechanism for distributing and verifying signature keys. Before writing the code, you should write a security model describing the signature key distribution and verification. The `chapter5/sessions` subpackage contains a solution to this problem.

A real-world solution can also be found in the go-schannel[7] package on Github.

Further reading

1. [Duong09] T. Duong, J. Rizzo. "Flickr's API Signature Forgery Vulnerability." http://netifera.com/research/flickr_api_signature_forgery.pdf, 28 September 2009.
2. [Ferg10] N. Ferguson, B. Schneier, T. Kohno. *Cryptography Engineering*. Wiley, March 2010.

[7] https://github.com/kisom/go-schannel

3. [Jons01] J. Jonsson, "Security Proofs for the RSA-PSS Signature." RSA Laboratories Europe, Stockholm, Sweden, July 2001.
4. [Lind06] C. Lindenberg, K. Wirt, J. Buchmann. "Formal Proof for the Correctness of RSA-PSS.", Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, 2006.
5. [PKCS1] J. Jonsson, B. Kaliski. "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1." RFC 3447, IETF and the Internet Society, February 2003.
6. [SEC1] D. R. L. Brown. Standards for Efficient Cryptography 1: Elliptic Curve Cryptography. Certicom Corp, May 2009.

Appendix: Crypto Review of Chapters

This appendix lists the chapters that have had a cryptographic review from someone else. I'd love to get feedback on this, and I'm open to any reviews.

- Chapter 1: **not reviewed**
- Chapter 2: **not reviewed**
- Chapter 3: **not reviewed**
- Chapter 4: **not reviewed**
- Chapter 5: **not reviewed**

Please bear this in mind when reading the book.