**Parte 2**

Protocolos: As receitas  
da criptografia

Agora você está entrando na segunda parte deste livro, que irá fazer uso de boa parte do que você aprendeu na primeira parte. Pense desta forma: se os primitivos criptográficos que você aprendeu fossem os ingredientes básicos da criptografia, agora você está prestes a aprender algumas receitas. E há muito o que cozinhar!

Enquanto César talvez estivesse interessado apenas em criptografar suas comunicações, hoje a criptografia está em todos os lugares, e é bastante difícil acompanhá-la.

Nos capítulos 9, 10 e 11, eu mostro onde você provavelmente encontrará criptografia e como a criptografia é usada para resolver problemas do mundo real; ou seja, como a criptografia criptografa comunicações e como autentica participantes em protocolos. Em sua maior parte, é disso que a criptografia se trata. Os participantes podem ser numerosos ou poucos, e compostos de bits ou de carne. Como você perceberá rapidamente, a criptografia do mundo real trata-se de compensações e, dependendo do contexto, as soluções irão variar.

Os capítulos 12 e 13 o levarão a dois campos de rápida evolução da criptografia: criptomoedas e criptografia em hardware. O primeiro tema tem sido ignorado por muitos livros de criptografia. (Acredito que este livro, *Real-World Cryptography*, é o primeiro livro de criptografia a incluir um capítulo sobre criptomoedas.) O segundo tema, criptografia em hardware, também é frequentemente negligenciado; os criptógrafos frequentemente assumem que seus primitivos e protocolos rodam em um ambiente confiável, o que está cada vez menos sendo o caso. A criptografia em hardware trata de expandir os limites de onde a criptografia pode operar e de fornecer garantias de segurança quando os atacantes estão se aproximando de você.

Nos capítulos 14 e 15, toco no que há de mais atual: o que ainda não está aqui mas estará, e o que meio que já chegou. Você aprenderá sobre criptografia pós-quântica, que é um campo da criptografia que pode ser útil, dependendo se nós, como espécie humana, inventarmos computadores quânticos escaláveis. Esses computadores quânticos, baseados em novos paradigmas vindos do reino da física quântica, poderiam revolucionar a pesquisa e, talvez, até mesmo quebrar nossa criptografia... Você também aprenderá sobre o que chamo de "criptografia de próxima geração", primitivos criptográficos que raramente viram a luz do dia mas que você provavelmente verá com mais frequência à medida que forem estudados, se tornarem mais eficientes e forem adotados por desenvolvedores de aplicações. Finalmente, concluo o livro no capítulo 16 com algumas observações finais sobre a criptografia do mundo real e algumas palavras sobre ética.

**Transporte seguro**

O uso mais intenso de criptografia hoje em dia é provavelmente para criptografar comunicações. Afinal, a criptografia foi inventada para este propósito. Para fazer isso, aplicações geralmente não fazem uso direto de primitivos criptográficos como criptografia autenticada, mas, em vez disso, utilizam protocolos muito mais elaborados que abstraem o uso dos primitivos criptográficos. Eu chamo esses protocolos de *protocolos de transporte seguro*, por falta de um termo melhor.

Neste capítulo, você aprenderá sobre o protocolo de transporte seguro mais amplamente utilizado: o protocolo TLS (*Transport Layer Security*). Também abordarei de forma leve outros protocolos de transporte seguro e como eles diferem do TLS.

**9.1 Os protocolos de transporte seguro SSL e TLS**

Para entender por que os protocolos de transporte (protocolos usados para criptografar comunicações entre máquinas) existem, vamos percorrer um cenário motivador.

Quando você digita, por exemplo, http://example.com em seu navegador, o seu navegador utiliza vários protocolos para se conectar a um servidor web e recuperar a página solicitada. Um deles é o *Hypertext Transfer Protocol* (HTTP), que o navegador usa para informar ao servidor web do outro lado qual página ele deseja. O HTTP usa um formato legível por humanos. Isso significa que você pode observar as mensagens HTTP que estão sendo enviadas e recebidas através do fio e lê-las sem a ajuda de nenhuma outra ferramenta. Mas isso não é suficiente para que seu navegador comunique-se com o servidor web.

As mensagens HTTP são encapsuladas em outros tipos de mensagens, chamadas quadros TCP, que são definidos no *Transmission Control Protocol* (TCP). O TCP é um protocolo binário e, portanto, não é legível por humanos: você precisa de uma ferramenta para entender os campos de um quadro TCP. As mensagens TCP são ainda encapsuladas utilizando o *Internet Protocol* (IP), e as mensagens IP são encapsuladas em algo mais. Isso é conhecido como a *Internet protocol suite* e, por ser tema de muitos livros, não irei me aprofundar mais nisso.

Voltando ao nosso cenário, há uma questão de confidencialidade sobre a qual precisamos falar. Qualquer pessoa sentada no fio entre o seu navegador e o servidor web de example.com está em uma posição interessante: pode observar passivamente e ler seus pedidos, bem como as respostas do servidor. Pior, atacantes *MITM* podem também adulterar ativamente e reordenar mensagens. Isso não é bom.

Imagine as informações do seu cartão de crédito vazando toda vez que você compra algo na internet, suas senhas sendo roubadas quando você faz login em um site, suas fotos e mensagens privadas sendo furtadas enquanto você as envia aos seus amigos, e assim por diante. Isso assustou pessoas o suficiente para que, nos anos 1990, o predecessor do TLS — o protocolo SSL (*Secure Sockets Layer*) — surgisse. Embora o SSL pudesse ser usado em diferentes tipos de situações, ele foi inicialmente criado por e para navegadores web. Assim, começou a ser utilizado em combinação com o HTTP, estendendo-o para o *Hypertext Transfer Protocol Secure* (HTTPS). O HTTPS agora permitia que os navegadores protegessem suas comunicações com os diferentes sites que visitavam.

**9.1.1 Do SSL para o TLS**

Embora o SSL não tenha sido o único protocolo a tentar proteger parte da web, ele atraiu a maior parte da atenção e, com o tempo, tornou-se o padrão de fato. Mas essa não é toda a história. Entre a primeira versão do SSL e o que usamos atualmente, muita coisa aconteceu. Todas as versões do SSL (sendo a última o SSL v3.0) foram quebradas devido a uma combinação de mau design e maus algoritmos criptográficos. (Muitos dos ataques foram resumidos no RFC 7457.)

Após o SSL 3.0, o protocolo foi oficialmente transferido para o *Internet Engineering Task Force* (IETF), a organização responsável por publicar os padrões *Request For Comments* (RFCs). O nome SSL foi abandonado em favor de TLS, e o TLS 1.0 foi lançado em 1999 como o RFC 2246. A versão mais recente do TLS é o TLS 1.3, especificado no RFC 8446 e publicado em 2018. O TLS 1.3, ao contrário de seus predecessores, resulta de uma colaboração sólida entre a indústria e a academia. No entanto, hoje, a internet ainda está dividida entre várias versões diferentes de SSL e TLS, pois os servidores têm sido lentos em atualizar.

NOTA: Há muita confusão em torno dos dois nomes SSL e TLS. O protocolo agora se chama TLS, mas muitos artigos e até bibliotecas ainda escolhem usar o termo SSL.

O TLS tornou-se mais do que apenas o protocolo que protege a web; ele agora é usado em muitos cenários diferentes e entre muitos tipos diferentes de aplicações e dispositivos como um protocolo para proteger comunicações. Assim, o que você aprenderá sobre TLS neste capítulo não é apenas útil para a web, mas também para qualquer cenário em que comunicações entre duas aplicações precisem ser seguras.

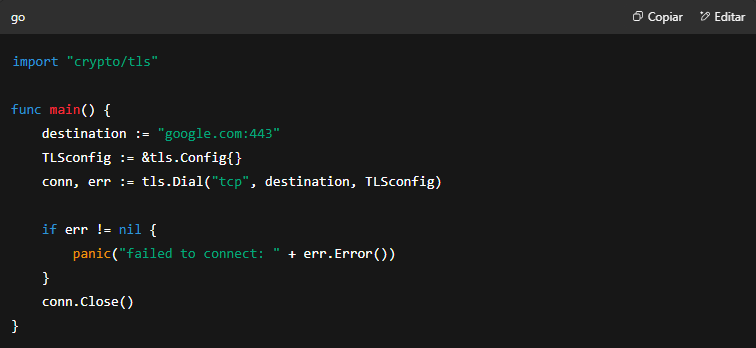
**9.1.2 Usando o TLS na prática**

Como as pessoas usam o TLS? Primeiro, vamos definir alguns termos. No TLS, os dois participantes que desejam proteger suas comunicações são chamados de *cliente* e *servidor*. Funciona da mesma forma que com outros protocolos de rede como TCP ou IP: o cliente é quem inicia a conexão, e o servidor é quem aguarda por uma conexão ser iniciada. Um cliente TLS é tipicamente construído a partir de:

* **Alguma configuração** — Um cliente é configurado com as versões de SSL e TLS que deseja suportar, os algoritmos criptográficos que está disposto a usar para proteger a conexão, formas de autenticar servidores, e assim por diante.
* **Alguma informação sobre o servidor com o qual deseja se conectar** — Inclui pelo menos um endereço IP e uma porta, mas, para a web, geralmente inclui um *fully qualified domain name* (nome de domínio totalmente qualificado), como example.com.

Dado esses dois argumentos, um cliente pode iniciar uma conexão com um servidor para produzir uma *sessão segura*, um canal que tanto o cliente quanto o servidor podem usar para compartilhar mensagens criptografadas entre si. Em alguns casos, uma sessão segura não pode ser criada com sucesso e falha no meio do caminho. Por exemplo, se um atacante tentar adulterar a conexão ou se a configuração do servidor não for compatível com a do cliente (mais sobre isso depois), o cliente falha ao estabelecer uma sessão segura.

Um servidor TLS é frequentemente muito mais simples, pois só precisa de uma configuração, semelhante à configuração do cliente. Um servidor então aguarda que clientes se conectem a ele para produzir uma sessão segura. Na prática, usar TLS no lado do cliente pode ser tão fácil quanto o exemplo a seguir (isto é, se você usar uma linguagem de programação como Golang):

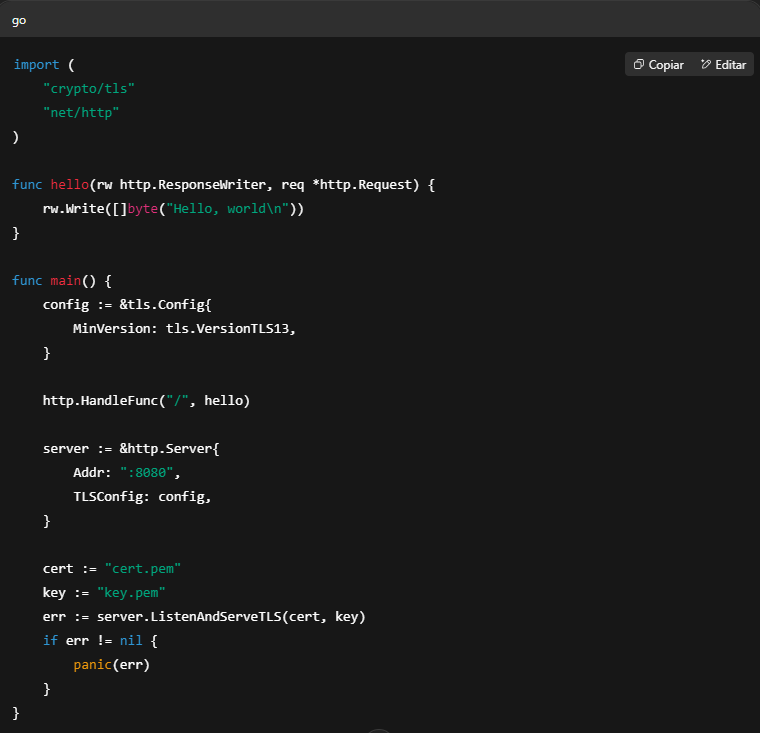


*Listing 9.1: Um cliente TLS em Golang*

O nome de domínio totalmente qualificado e a porta do servidor (443 é a porta padrão para HTTPS).

Uma configuração vazia serve como configuração padrão.

Como o cliente sabe que a conexão que estabeleceu é realmente com google.com e não com algum impostor? Por padrão, a implementação TLS do Golang usa a configuração do seu sistema operacional para descobrir como autenticar servidores TLS. (Mais adiante neste capítulo, você aprenderá exatamente como a autenticação no TLS funciona.) Usar o TLS no lado do servidor também é bem fácil. O próximo exemplo mostra quão simples isso é:



*Listing 9.2: Um servidor TLS em Golang*

Uma configuração mínima sólida para um servidor TLS 1.3.

Servindo uma página simples exibindo "Hello, world".

Um servidor HTTPS inicia na porta 8080.

Alguns arquivos .pem contendo um certificado e uma chave secreta (mais sobre isso depois).

O Golang e sua biblioteca padrão fazem muito por nós aqui. Infelizmente, nem todas as bibliotecas padrão das linguagens fornecem implementações de TLS fáceis de usar — se é que fornecem uma implementação de TLS —, e nem todas as bibliotecas de TLS oferecem implementações seguras por padrão! Por esse motivo, configurar um servidor TLS nem sempre é trivial, dependendo da biblioteca. Na próxima seção, você aprenderá sobre o funcionamento interno do TLS e suas diferentes sutilezas.

NOTA: O TLS é um protocolo que funciona sobre o TCP. Para proteger conexões UDP, podemos usar o DTLS (o D vem de *datagram*, o termo para mensagens UDP), que é bastante semelhante ao TLS. Por esse motivo, ignoro o DTLS neste capítulo.

**9.2 Como o protocolo TLS funciona?**

Como eu disse anteriormente, hoje o TLS é o padrão de fato para proteger comunicações entre aplicações. Nesta seção, você aprenderá mais sobre como o TLS funciona por baixo dos panos e como é usado na prática. Você achará esta seção útil para aprender como usar o TLS corretamente e também para entender como a maioria (senão todos) dos protocolos de transporte seguro funcionam. Você também entenderá por que é difícil (e fortemente desaconselhado) redesenhar ou reimplementar tais protocolos.

Em um nível alto, o TLS é dividido em duas fases, como destacado na lista a seguir. A Figura 9.1 ilustra essa ideia.

* **Uma fase de handshake (aperto de mão)** — Uma comunicação segura é negociada e criada entre dois participantes.
* **Uma fase pós-handshake** — As comunicações são criptografadas entre os dois participantes.

Neste ponto, como você aprendeu sobre criptografia híbrida no capítulo 6, você deve ter a seguinte (correta) intuição sobre como esses dois passos funcionam:

* O handshake é, em sua essência, simplesmente uma troca de chaves. O handshake termina com os dois participantes concordando em um conjunto de chaves simétricas.
* A fase pós-handshake trata puramente de criptografar mensagens entre os participantes. Essa fase usa um algoritmo de criptografia autenticada e o conjunto de chaves produzido ao final do handshake.

A maioria dos protocolos de segurança de transporte funciona dessa forma, e as partes interessantes desses protocolos quase sempre residem na fase de handshake. Em seguida, vamos dar uma olhada na fase de handshake.

<IMAGEM>

*Figura 9.1: Em um nível alto, os protocolos de transporte seguro primeiro criam uma conexão segura durante uma fase de handshake. Depois disso, aplicações em ambos os lados da conexão segura podem se comunicar com segurança.*

**9.2.1 O handshake do TLS**

Como você viu, o TLS (e a maioria dos protocolos de segurança de transporte) é dividido em duas partes: um handshake e uma fase pós-handshake. Nesta seção, você aprenderá primeiro sobre o handshake. O próprio handshake tem quatro aspectos sobre os quais quero falar:

* **Negociação** — O TLS é altamente configurável. Tanto o cliente quanto o servidor podem ser configurados para negociar uma gama de versões de SSL e TLS, bem como um menu de algoritmos criptográficos aceitáveis. A fase de negociação do handshake visa encontrar um terreno comum entre as configurações do cliente e do servidor a fim de conectar com segurança os dois pares.
* **Troca de chaves** — Todo o objetivo do handshake é realizar uma troca de chaves entre dois participantes. Qual algoritmo de troca de chaves usar? Isso é uma das coisas decididas como parte do processo de negociação cliente/servidor.
* **Autenticação** — Como você aprendeu no capítulo 5 sobre trocas de chaves, é trivial para atacantes MITM se passarem por qualquer lado de uma troca de chaves. Por isso, trocas de chaves devem ser autenticadas. Seu navegador precisa ter uma forma de garantir que está falando com google.com, por exemplo, e não com seu provedor de internet (ISP).
* **Reutilização de sessão** — Como os navegadores frequentemente se conectam aos mesmos sites repetidamente, as trocas de chaves podem ser custosas e podem atrasar a experiência do usuário. Por essa razão, mecanismos para acelerar sessões seguras sem refazer uma troca de chaves estão integrados no TLS.

Esta é uma lista abrangente! Rapidamente, vamos começar pelo primeiro item.

**NEGOCIAÇÃO NO TLS: QUAL VERSÃO E QUAIS ALGORITMOS?**

A maior parte da complexidade no TLS vem da negociação das diferentes partes móveis do protocolo. Notoriamente, essa negociação também foi a fonte de muitos problemas na história do TLS. Ataques como FREAK, LOGJAM, DROWN, entre outros, exploraram fraquezas presentes em versões antigas para quebrar versões mais recentes do protocolo (às vezes mesmo quando o servidor não suportava as versões mais antigas!). Embora nem todos os protocolos tenham versionamento ou permitam negociação de diferentes algoritmos, o SSL/TLS foi projetado para a web. Como tal, o SSL/TLS precisava de um meio de manter compatibilidade retroativa com clientes e servidores antigos que podem ser lentos para atualizar.

É isso que acontece na web hoje: seu navegador pode ser recente e atualizado e ter suporte ao TLS versão 1.3, mas ao visitar alguma página antiga, há chances de que o servidor por trás dela suporte apenas versões do TLS até 1.2 ou 1.1 (ou pior). Por outro lado, muitos websites precisam suportar navegadores antigos, o que se traduz em suportar versões antigas do TLS (já que alguns usuários ainda estão presos no passado).

**As versões antigas de SSL e TLS são seguras?**

A maioria das versões de SSL e TLS possui problemas de segurança, exceto as versões TLS 1.2 e TLS 1.3.  
Por que não simplesmente suportar a versão mais recente (1.3) e encerrar o assunto?  
A razão é que algumas empresas precisam suportar clientes antigos que não podem ser atualizados facilmente. Devido a esses requisitos, não é incomum encontrar bibliotecas que implementam mitigadores para ataques conhecidos a fim de dar suporte seguro a versões antigas. Infelizmente, essas mitigações são frequentemente complexas demais para serem implementadas corretamente.

Por exemplo, ataques bem conhecidos como Lucky13 e Bleichenbacher98 têm sido redescobertos repetidamente por pesquisadores de segurança em várias implementações de TLS que anteriormente tentaram corrigir as falhas. Embora seja possível mitigar um número de ataques em versões antigas do TLS, eu recomendaria não fazê-lo — e não sou o único dizendo isso. Em março de 2021, o IETF publicou o RFC 8996: “Descontinuando o TLS 1.0 e TLS 1.1”, efetivamente oficializando a descontinuação.

A negociação começa com o cliente enviando um primeiro pedido (chamado de *ClientHello*) ao servidor. O *ClientHello* contém:

* uma gama de versões de SSL e TLS suportadas,
* um conjunto de algoritmos criptográficos que o cliente está disposto a usar,
* e algumas informações adicionais que podem ser relevantes para o restante do handshake ou para a aplicação.

O conjunto de algoritmos criptográficos inclui:

* **Um ou mais algoritmos de troca de chaves** — O TLS 1.3 define os seguintes algoritmos permitidos para negociação:
  + ECDH com P-256, P-384, P-521,
  + X25519, X448,
  + e FFDH com os grupos definidos no RFC 7919.  
    (Falei sobre todos esses no capítulo 5.)  
    Versões anteriores do TLS também ofereciam trocas de chave RSA (abordadas no capítulo 6), mas elas foram removidas na versão mais recente.
* **Dois (para diferentes partes do handshake) ou mais algoritmos de assinatura digital** — O TLS 1.3 especifica:
  + RSA PKCS#1 versão 1.5 e o mais recente RSA-PSS,
  + bem como algoritmos mais recentes de curvas elípticas como ECDSA e EdDSA.  
    (Falei sobre eles no capítulo 7.)  
    Note que as assinaturas digitais são especificadas junto com uma função hash, o que permite negociar, por exemplo, RSA-PSS com SHA-256 ou SHA-512.
* **Uma ou mais funções hash a serem usadas com HMAC e HKDF** — O TLS 1.3 especifica SHA-256 e SHA-384, duas instâncias da função hash SHA-2. (Você aprendeu sobre SHA-2 no capítulo 2.)  
  Essa escolha de função hash não está relacionada àquela usada pelo algoritmo de assinatura digital.  
  Como lembrete:
  + HMAC é o código de autenticação de mensagens que você aprendeu no capítulo 3,
  + e HKDF é a função de derivação de chaves que cobrimos no capítulo 8.
* **Um ou mais algoritmos de criptografia autenticada** — Estes podem incluir:
  + AES-GCM com chaves de 128 ou 256 bits,
  + ChaCha20-Poly1305,
  + e AES-CCM.  
    (Falei sobre todos esses no capítulo 4.)

O servidor então responde com uma mensagem *ServerHello*, que contém um de cada tipo de algoritmo criptográfico, selecionados a partir da lista oferecida pelo cliente. A ilustração a seguir representa essa resposta:

<IMAGEM>

Se o servidor não conseguir encontrar um algoritmo que suporte, ele aborta a conexão. Embora, em alguns casos, o servidor não precise abortar imediatamente e possa solicitar ao cliente informações adicionais. Para isso, o servidor responde com uma mensagem chamada *HelloRetryRequest*, solicitando o dado faltante. O cliente então pode reenviar seu *ClientHello*, desta vez com a informação solicitada adicionada.

**TLS E TROCAS DE CHAVE COM SIGILO PARA FRENTE**

A troca de chaves é a parte mais importante do *handshake* do TLS! Sem ela, obviamente não há chave simétrica sendo negociada. Mas, para que uma troca de chaves aconteça, o cliente e o servidor devem primeiro trocar suas respectivas chaves públicas.

No TLS 1.2 e versões anteriores, o cliente e o servidor iniciam uma troca de chaves apenas depois que ambos os participantes concordam sobre qual algoritmo de troca de chaves usar. Isso acontece durante uma fase de negociação. O TLS 1.3 otimiza esse fluxo tentando fazer tanto a negociação quanto a troca de chaves ao mesmo tempo: o cliente escolhe especulativamente um algoritmo de troca de chaves e envia uma chave pública na primeira mensagem (*ClientHello*). Se o cliente falhar em prever a escolha de algoritmo de troca de chaves do servidor, então o cliente recorre ao resultado da negociação e envia um novo *ClientHello* contendo a chave pública correta. Os seguintes passos descrevem como isso pode ocorrer. Eu ilustro a diferença na Figura 9.2.

1. O cliente envia uma mensagem *ClientHello* TLS 1.3 anunciando que pode fazer uma troca de chaves X25519 ou X448. Também envia uma chave pública X25519.
2. O servidor não suporta X25519, mas suporta X448. Ele envia um *HelloRetryRequest* ao cliente anunciando que só suporta X448.
3. O cliente envia o mesmo *ClientHello*, mas agora com uma chave pública X448.
4. O *handshake* prossegue.

O TLS 1.3 está repleto dessas otimizações, que são importantes para a web. De fato, muitas pessoas ao redor do mundo têm conexões instáveis ou lentas, e é importante reduzir ao mínimo as comunicações não relacionadas à aplicação. Além disso, no TLS 1.3 (diferentemente das versões anteriores do TLS), todas as trocas de chaves são efêmeras. Isso significa que, para cada nova sessão, tanto o cliente quanto o servidor geram novos pares de chaves e os descartam assim que a troca de chaves termina. Isso proporciona *sigilo de encaminhamento* (forward secrecy) à troca de chaves: a violação das chaves de longo prazo do cliente ou do servidor não permitirá a um atacante descriptografar essa sessão, desde que as chaves privadas efêmeras tenham sido eliminadas com segurança.

Imagine o que aconteceria se, em vez disso, um servidor TLS usasse uma única chave privada para cada troca de chaves que realiza com seus clientes. Ao realizar trocas de chaves efêmeras e descartar as chaves privadas assim que o *handshake* termina, o servidor se protege contra tais atacantes. Eu ilustro isso na Figura 9.3.

<IMAGEM>

*Figura 9.2: No TLS 1.2, o cliente aguarda o servidor escolher qual algoritmo de troca de chaves usar antes de enviar uma chave pública. No TLS 1.3, o cliente especula quais algoritmos de troca de chaves o servidor escolherá e envia previamente uma chave pública (ou várias) na primeira mensagem, potencialmente evitando uma ida e volta adicional.*

Uma vez que as chaves públicas efêmeras são trocadas, uma troca de chaves é realizada, e chaves podem ser derivadas. O TLS 1.3 deriva diferentes chaves em diferentes momentos para criptografar diferentes fases com chaves independentes.

As duas primeiras mensagens, o *ClientHello* e o *ServerHello*, não podem ser criptografadas porque nenhuma chave pública foi trocada nesse ponto. Mas, após isso, assim que a troca de chaves acontece, o TLS 1.3 criptografa o restante do *handshake*. (Isso é diferente das versões anteriores do TLS, que não criptografavam nenhuma das mensagens do *handshake*.)

Para derivar as diferentes chaves, o TLS 1.3 usa o HKDF com a função hash negociada. O *HKDF-Extract* é usado na saída da troca de chaves para remover qualquer viés, enquanto o *HKDF-Expand* é usado com diferentes parâmetros de informação para derivar as chaves de criptografia. Por exemplo:

* tls13 c hs traffic (para “tráfego de handshake do cliente”) é usado para derivar chaves simétricas para o cliente criptografar para o servidor durante o *handshake*,
* e tls13 s ap traffic (para “tráfego de aplicação do servidor”) é usado para derivar chaves simétricas para o servidor criptografar para o cliente após o *handshake*.

Lembre-se, porém: trocas de chaves não autenticadas são inseguras! Em seguida, veremos como o TLS lida com isso.

**Exercício:**  
Uma violação da chave privada do servidor em algum momento seria devastadora, pois atacantes *MITM* poderiam então descriptografar todas as conversas previamente gravadas. Você entende como isso pode acontecer?

<IMAGEM>

*Figura 9.3: No TLS 1.3, cada sessão começa com uma troca de chaves efêmera. Se um servidor for comprometido em algum momento, nenhuma sessão anterior será impactada.*

**AUTENTICAÇÃO DO TLS E A INFRAESTRUTURA DE CHAVES PÚBLICAS DA WEB (WEB PKI)**

Após algumas negociações e depois da troca de chaves ter ocorrido, o *handshake* deve prosseguir. O que acontece a seguir é a outra parte mais importante do TLS — a **autenticação**.

Você viu no capítulo 5, sobre trocas de chaves, que é trivial interceptar uma troca de chaves e se passar por um ou ambos os lados da troca. Nesta seção, explicarei como seu navegador valida criptograficamente que está se comunicando com o site correto e não com um impostor. Mas, primeiro, vamos dar um passo atrás. Há algo que ainda não contei. Um *handshake* TLS 1.3 é, na verdade, dividido em três etapas distintas (como ilustra a Figura 9.4):

1. **Troca de chaves** — Esta fase contém as mensagens *ClientHello* e *ServerHello* que realizam algumas negociações e a troca de chaves. Todas as mensagens, incluindo mensagens de *handshake* após esta fase, são criptografadas.
2. **Parâmetros do servidor** — Mensagens nesta fase contêm dados adicionais de negociação por parte do servidor. São dados de negociação que não precisam estar contidos na primeira mensagem do servidor e que podem se beneficiar por estarem criptografados.
3. **Autenticação** — Esta fase inclui informações de autenticação tanto do servidor quanto do cliente.

Na web, a autenticação no TLS costuma ser unilateral. Apenas o navegador verifica que google.com, por exemplo, é de fato google.com, mas google.com não verifica quem você é (ou ao menos não como parte do TLS).

**TLS mutuamente autenticado**

A autenticação do cliente frequentemente é delegada à camada de aplicação na web, geralmente via um formulário solicitando suas credenciais. Dito isso, a autenticação do cliente também pode ocorrer no TLS se solicitada pelo servidor durante a fase de parâmetros do servidor. Quando ambos os lados da conexão são autenticados, falamos de TLS mutuamente autenticado (às vezes abreviado como *mTLS*).

A autenticação do cliente ocorre da mesma forma que a autenticação do servidor. Isso pode acontecer em qualquer ponto após a autenticação do servidor (por exemplo, durante o *handshake* ou na fase pós-*handshake*).

<IMAGEM>

*Figura 9.4: Um handshake TLS 1.3 é dividido em três fases: a fase de troca de chaves, a fase de parâmetros do servidor e (finalmente) a fase de autenticação.*

Agora, vamos responder à pergunta:  
**“Ao se conectar ao google.com, como seu navegador verifica que está realmente realizando o *handshake* com o google.com?”**  
A resposta é: usando a infraestrutura de chaves públicas da web (*web PKI*).

Você aprendeu sobre o conceito de infraestrutura de chaves públicas no capítulo 7, sobre assinaturas digitais, mas deixe-me reapresentar brevemente este conceito, pois ele é bastante importante para entender como a web funciona. Há dois lados na *web PKI*:

* Primeiro, os navegadores devem confiar em um conjunto de chaves públicas raiz que chamamos de autoridades certificadoras (*Certificate Authorities*, ou CAs).  
  Geralmente, os navegadores usarão um conjunto embutido de chaves públicas confiáveis ou confiarão no sistema operacional para fornecê-las.
* Segundo, websites que desejam usar HTTPS precisam de um meio para obter uma certificação (uma assinatura de sua chave pública de assinatura) dessas CAs.  
  Para isso, o proprietário do site (ou webmaster, como costumávamos dizer) deve provar a uma CA que possui um determinado domínio.

**NOTA**:  
Obter um certificado para o seu próprio website costumava envolver uma taxa.  
Isso não é mais o caso, pois CAs como o *Let’s Encrypt* fornecem certificados gratuitamente.

Para provar que você possui example.com, por exemplo, uma CA pode pedir para você hospedar um arquivo em example.com/some\_path/file.txt contendo alguns números aleatórios gerados para sua solicitação. A seguinte história em quadrinhos mostra essa troca:

*Aqui havia uma ilustração, portanto inserimos:*  
<IMAGEM>

Após isso, uma CA pode fornecer uma assinatura sobre a chave pública do website. Como a assinatura da CA geralmente é válida por alguns anos, dizemos que ela está sobre uma chave pública de longo prazo (em oposição a uma chave efêmera). Mais especificamente, as CAs não assinam diretamente as chaves públicas, mas sim **certificados** (mais sobre isso em breve).  
Um certificado contém a chave pública de longo prazo, juntamente com alguns metadados importantes adicionais como o nome de domínio da página web.

Para provar ao seu navegador que o servidor com o qual está se comunicando é de fato google.com, o servidor envia uma cadeia de certificados como parte do *handshake* TLS. A cadeia compreende:

* Seu próprio certificado *leaf* (folha), contendo (entre outros):
  + o nome de domínio (google.com, por exemplo),
  + a chave pública de assinatura de longo prazo do Google,
  + bem como a assinatura de uma CA.
* Uma cadeia de certificados de CA intermediárias, da que assinou o certificado do Google até a CA raiz que assinou a última intermediária.

Isso é um pouco verboso, então ilustrei isso na Figura 9.5:

<IMAGEM>

*Figura 9.5: Os navegadores web precisam confiar apenas em um conjunto relativamente pequeno de CAs raiz para confiar em toda a web. Essas CAs são armazenadas no que é chamado de trust store (repositório de confiança). Para que um website seja confiável por um navegador, o site deve ter seu certificado folha assinado por uma dessas CAs. Às vezes, CAs raiz apenas assinam CAs intermediárias, que, por sua vez, assinam outras CAs intermediárias ou certificados folha. Isso é conhecido como a infraestrutura de chaves públicas da web (web PKI).*

A cadeia de certificados é enviada em uma mensagem TLS chamada *certificate*, tanto pelo servidor quanto pelo cliente (caso o cliente tenha sido solicitado a autenticar-se). Em seguida, o servidor pode usar seu par de chaves de longo prazo certificado para assinar todas as mensagens de *handshake* que foram recebidas e previamente enviadas, no que é chamado de mensagem *CertificateVerify*. A Figura 9.6 revisa esse fluxo, onde apenas o servidor se autentica.

A assinatura na mensagem *CertificateVerify* prova ao cliente o que o servidor viu até aquele momento. Sem essa assinatura, um atacante *MITM* poderia interceptar as mensagens de *handshake* do servidor e substituir a chave pública efêmera do servidor contida na mensagem *ServerHello*, permitindo que o atacante se passasse com sucesso pelo servidor.  
Reserve alguns momentos para entender por que um atacante não pode substituir a chave pública efêmera do servidor na presença da assinatura *CertificateVerify*.

<IMAGEM>

*Figura 9.6: A parte de autenticação de um handshake começa com o servidor enviando uma cadeia de certificados ao cliente. A cadeia de certificados começa com o certificado folha (o certificado contendo a chave pública do website e metadados adicionais como o nome do domínio) e termina com um certificado raiz confiado pelo navegador. Cada certificado contém uma assinatura do certificado imediatamente acima na cadeia.*

Finalmente, para encerrar oficialmente o *handshake*, ambos os lados da conexão devem enviar uma mensagem *Finished* como parte da fase de autenticação.  
Uma mensagem *Finished* contém uma tag de autenticação produzida por HMAC, instanciada com a função hash negociada para a sessão.  
Isso permite que tanto o cliente quanto o servidor digam um ao outro:

"Estas são todas as mensagens que enviei e recebi em ordem durante este *handshake*."

Se o *handshake* for interceptado e adulterado por atacantes *MITM*, essa verificação de integridade permite que os participantes detectem e abortem a conexão.  
Isso é especialmente útil, pois alguns modos de *handshake* não são assinados (mais sobre isso depois).

Antes de passar para outro aspecto do *handshake*, vamos examinar os certificados X.509. Eles são um detalhe importante de muitos protocolos criptográficos.

**Hora da história**

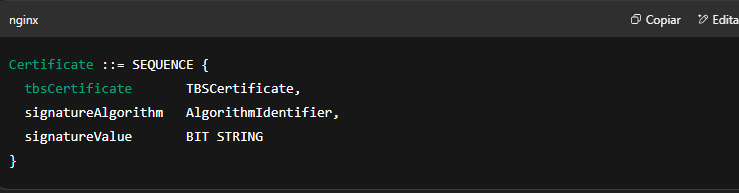
Alguns anos atrás, fui contratado para revisar um protocolo TLS personalizado feito por uma grande empresa. Descobriu-se que o protocolo tinha o servidor fornecendo uma assinatura que não cobria a chave efêmera. Quando informei sobre o problema, a sala inteira ficou em silêncio por um minuto inteiro. Era, claro, um erro substancial: um atacante que interceptasse o *handshake* personalizado e substituísse a chave efêmera por sua própria poderia se passar com sucesso pelo servidor.

A lição aqui é que não devemos reinventar a roda. Protocolos de transporte seguro são difíceis de acertar e, se a história mostrou algo, é que podem falhar de formas inesperadas.  
Em vez disso, deve-se confiar em protocolos maduros como o TLS e garantir o uso de uma implementação popular que tenha recebido uma quantidade substancial de atenção pública.

**AUTENTICAÇÃO VIA CERTIFICADOS X.509**

Embora os certificados sejam opcionais no TLS 1.3 (você sempre pode usar chaves puras), muitas aplicações e protocolos — não apenas a web — fazem uso intenso deles para certificar metadados adicionais. Especificamente, é utilizado o padrão de certificado **X.509 versão 3**.

O X.509 é um padrão bastante antigo, que foi projetado para ser flexível o suficiente para ser usado em uma infinidade de cenários: desde email até páginas web. O padrão X.509 usa uma linguagem de descrição chamada *Abstract Syntax Notation One* (ASN.1) para especificar as informações contidas em um certificado.  
Uma estrutura de dados descrita em ASN.1 se parece com isto:



Você pode literalmente ler isso como uma estrutura que contém três campos:

* **tbsCertificate** — O certificado a ser assinado (*to-be-signed*). Contém todas as informações que se deseja certificar. Para a web, pode conter:
  + um nome de domínio (por exemplo, google.com),
  + uma chave pública,
  + uma data de expiração, e assim por diante.
* **signatureAlgorithm** — O algoritmo usado para assinar o certificado.
* **signatureValue** — A assinatura de uma CA.

Você pode facilmente verificar o que há em um certificado X.509 conectando-se a qualquer website usando HTTPS e, então, utilizando funcionalidades do seu navegador para observar a cadeia de certificados enviada pelo servidor. Veja a Figura 9.7 como exemplo.

<IMAGEM>

\*Figura 9.7: Utilizando o *Certificate Viewer* do Chrome, podemos observar a cadeia de certificados enviada pelo servidor do Google.  
A CA raiz é a Global Sign, que é confiável pelo seu navegador.  
Abaixo na cadeia, uma CA intermediária chamada GTS CA 101 é confiável devido ao seu certificado conter uma assinatura da Global Sign.  
Por fim, o certificado folha do Google, válido para *.google.com (google.com, mail.google.com, e assim por diante), contém uma assinatura da GTS CA 101.*

Você pode encontrar certificados X.509 como arquivos .pem, que contêm conteúdo em base64 cercado por uma indicação legível de humanos sobre o que há dentro (neste caso, um certificado).  
O trecho a seguir representa o conteúdo de um certificado no formato .pem:



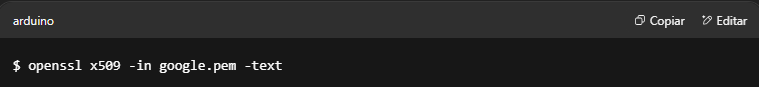
**Exercício:**  
Os valores signatureAlgorithm e signatureValue não estão contidos no certificado real tbsCertificate.  
Você sabe por quê?

Se você decodificar o conteúdo base64 entre *BEGIN CERTIFICATE* e *END CERTIFICATE*, acabará com um certificado codificado em **DER** (*Distinguished Encoding Rules*).  
O DER é uma codificação binária determinística (apenas uma forma de codificar) usada para traduzir certificados X.509 em bytes.  
Todas essas codificações costumam confundir iniciantes!  
Recapitulo tudo isso na Figura 9.8:

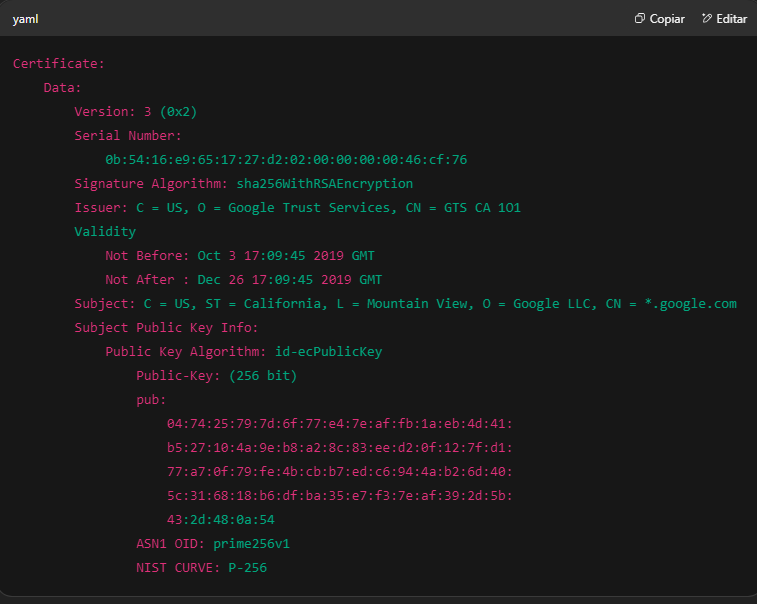
<IMAGEM>

*Figura 9.8: No canto superior esquerdo, um certificado X.509 é escrito utilizando a notação ASN.1.  
Depois é transformado em bytes via codificação DER para ser assinado.  
Como isso não é texto facilmente manipulável por humanos, é codificado em base64.  
Por fim, o dado base64 é envolvido com algumas informações contextuais no formato PEM.*

O DER apenas codifica informações como “aqui está um inteiro” ou “isto é um array de bytes”.  
Os nomes dos campos descritos no ASN.1 (como tbsCertificate) se perdem após a codificação.  
Decodificar DER sem o conhecimento da descrição original ASN.1 de cada campo torna-se, portanto, inútil.  
Ferramentas de linha de comando como o OpenSSL permitem decodificar e traduzir em termos humanos o conteúdo de um certificado codificado em DER.  
Por exemplo, se você baixar o certificado do google.com, pode usar o seguinte comando:



O que produzirá uma saída como:



**PRÉ-CHAVES COMPARTILHADAS E REUTILIZAÇÃO DE SESSÃO NO TLS, OU COMO EVITAR TROCAS DE CHAVES**

As trocas de chaves podem ser custosas e, às vezes, não são necessárias. Por exemplo, você pode ter duas máquinas que se conectam apenas uma à outra e talvez não queira lidar com uma infraestrutura de chave pública (PKI) para proteger suas comunicações.  
O TLS 1.3 oferece uma maneira de evitar essa sobrecarga com **pré-chaves compartilhadas** (*pre-shared keys* — PSKs).  
Uma PSK é simplesmente um segredo que tanto o cliente quanto o servidor conhecem, o qual pode ser usado para derivar chaves simétricas para a sessão.

No TLS 1.3, um *handshake* com PSK funciona da seguinte forma:

* O cliente anuncia em sua mensagem *ClientHello* que suporta uma lista de identificadores de PSK.
* Se o servidor reconhecer um dos IDs de PSK, poderá indicar isso em sua resposta (*ServerHello*), e ambos podem então evitar realizar uma troca de chaves (se assim desejarem).

Ao fazer isso, a fase de autenticação é pulada, tornando a mensagem *Finished* ao final do *handshake* ainda mais importante para prevenir ataques *MITM*.

Outro caso de uso para PSKs é a **reutilização de sessão** (*session resumption*).  
Reutilização de sessão consiste em reaproveitar segredos criados a partir de uma sessão ou conexão anterior.  
Se você já se conectou ao google.com e já verificou sua cadeia de certificados, realizou uma troca de chaves, concordou com um segredo compartilhado etc., por que repetir tudo isso novamente alguns minutos ou horas depois ao revisitá-lo?  
O TLS 1.3 oferece uma forma de gerar uma PSK após um *handshake* ter sido concluído com sucesso, que pode ser usada em conexões subsequentes para evitar refazer um *handshake* completo.

Se o servidor quiser oferecer esse recurso, ele pode enviar uma mensagem *New Session Ticket* a qualquer momento durante a fase pós-*handshake*.  
O servidor pode criar esses **tickets de sessão** de várias formas. Por exemplo, o servidor pode enviar um identificador, associado às informações relevantes em um banco de dados.  
Esta não é a única forma, mas como esse mecanismo é bastante complexo e, na maioria das vezes, desnecessário, não irei aprofundar mais este assunto neste capítulo.

**Cliente *random* e servidor *random***

Um leitor atento pode ter notado que as chaves públicas efêmeras trazem aleatoriedade para a sessão, e sem elas as chaves de sessão simétricas ao final do *handshake* poderiam acabar sendo sempre as mesmas.  
Usar diferentes chaves simétricas para diferentes sessões é extremamente importante, pois você não quer que essas sessões sejam correlacionadas.  
Pior ainda, como as mensagens criptografadas podem ser diferentes entre sessões, isso poderia levar a reutilizações de *nonce* e suas consequências catastróficas (veja o capítulo 4).

Para mitigar isso, tanto as mensagens *ClientHello* quanto *ServerHello* possuem um campo de aleatoriedade, que é gerado aleatoriamente para cada nova sessão (frequentemente chamado de **client random** e **server random**).  
Como esses valores aleatórios são usados na derivação das chaves simétricas no TLS, isso efetivamente randomiza as chaves de sessão simétricas para cada nova conexão.

**9.2.2 Como o TLS 1.3 criptografa dados de aplicação**

Uma vez que o *handshake* acontece e as chaves simétricas são derivadas, tanto o cliente quanto o servidor podem enviar dados de aplicação criptografados um ao outro. Mas não é só isso: o TLS também garante que tais mensagens não possam ser repetidas nem reordenadas!  
Para fazer isso, o *nonce* usado pelo algoritmo de criptografia autenticada começa em um valor fixo e é incrementado a cada nova mensagem.  
Se uma mensagem for repetida ou reordenada, o *nonce* será diferente do esperado e a descriptografia falhará. Quando isso ocorre, a conexão é encerrada.

A partir do TLS 1.3, caso o servidor assim permita, os clientes têm a possibilidade de enviar dados criptografados como parte de suas primeiras mensagens, logo após a mensagem *ClientHello*.  
Isso significa que os navegadores não precisam necessariamente esperar até o final do *handshake* para começar a enviar dados de aplicação ao servidor.  
Esse mecanismo é chamado de **early data** ou **0-RTT** (*zero round trip time*).  
Ele só pode ser usado em combinação com uma PSK, pois permite a derivação de chaves simétricas já durante a mensagem *ClientHello*.

**NOTA:**  
Este recurso foi bastante controverso durante o desenvolvimento do padrão TLS 1.3 porque um atacante passivo pode repetir um *ClientHello* observado seguido dos dados 0-RTT criptografados.  
Por isso, o 0-RTT deve ser usado apenas com dados de aplicação que possam ser repetidos com segurança.

Para a web, os navegadores tratam toda consulta GET como idempotente, o que significa que requisições GET não devem alterar estado no lado do servidor e destinam-se apenas a recuperar dados (diferente das requisições POST, por exemplo).  
Isso, claro, nem sempre é verdade, e aplicações já foram conhecidas por fazerem o que bem entendem.  
Por essa razão, caso você esteja diante da decisão de usar ou não 0-RTT, é mais simples simplesmente **não usá-lo**.

**9.3 O estado da web criptografada hoje**

Hoje, os padrões estão pressionando pela descontinuação de todas as versões de SSL e TLS que não sejam TLS 1.2 e TLS 1.3.  
No entanto, devido a clientes e servidores legados, muitas bibliotecas e aplicações continuam suportando versões mais antigas do protocolo (até mesmo o SSL versão 3, às vezes!).  
Isso não é simples, e devido ao número de vulnerabilidades que você precisa defender, muitas mitigações difíceis de implementar devem ser mantidas.

**AVISO:**  
Usar o TLS 1.3 (e TLS 1.2) é considerado seguro e uma boa prática.  
Usar qualquer versão inferior significa que você precisará consultar especialistas e descobrir como evitar vulnerabilidades conhecidas.

Por padrão, os navegadores ainda se conectam a servidores web usando HTTP, e os websites ainda precisam solicitar manualmente a uma CA para obter um certificado.  
Isso significa que, com os protocolos atuais, a web nunca será totalmente criptografada, embora algumas estimativas indiquem que 90% do tráfego web global estava criptografado em 2019.

O fato de, por padrão, seu navegador sempre usar uma conexão insegura também é um problema.  
Os servidores web hoje em dia geralmente redirecionam usuários que acessam suas páginas via HTTP para o HTTPS.  
Os servidores web também podem (e frequentemente o fazem) instruir navegadores a usarem HTTPS em conexões subsequentes.  
Isso é feito via um cabeçalho de resposta HTTPS chamado **HTTP Strict Transport Security (HSTS)**.  
Ainda assim, a primeira conexão com um website permanece não protegida (a menos que o usuário digite manualmente *https* na barra de endereços) e pode ser interceptada para remover o redirecionamento para HTTPS.

Além disso, outros protocolos web como NTP (para obter o horário atual) e DNS (para obter o IP por trás de um nome de domínio) ainda estão, em sua maioria, não criptografados e vulneráveis a ataques *MITM*.  
Embora existam esforços de pesquisa para melhorar esse quadro, estes ainda são vetores de ataque dos quais devemos estar cientes.

Há outra ameaça aos usuários de TLS: **CAs mal-comportadas**.  
E se, hoje, uma CA decidir assinar um certificado para o seu domínio e uma chave pública que ela controla?  
Se ela conseguir obter uma posição *MITM*, poderia começar a se passar pelo seu website junto aos seus usuários.

A solução óbvia, se você controla o lado cliente da conexão, é não usar a *web PKI* (e confiar em sua própria PKI) ou realizar o **pinning** de um certificado ou chave pública.

**Pinagem de certificado ou chave pública**

Estas são técnicas onde o certificado do servidor (ou, mais comumente, um hash dele), ou a chave pública, são codificados diretamente no código do cliente.  
Se o servidor não apresentar o certificado esperado, ou o certificado não contiver a chave pública de longo prazo esperada, o cliente aborta a conexão durante a fase de autenticação do *handshake*.

Essa prática é frequentemente usada em aplicações móveis, já que estas sabem exatamente qual é a chave pública ou o certificado esperado do servidor (diferente dos navegadores que precisam se conectar a um número infinito de servidores).  
Codificar certificados e chaves públicas nem sempre é possível, contudo, e dois outros mecanismos coexistem para lidar com certificados inválidos:

* **Revogação de certificados** — Como o próprio nome indica, permite que uma CA revogue um certificado e avise os navegadores sobre isso.
* **Monitoramento de certificados** — Este é um sistema relativamente novo que força as CAs a registrarem publicamente cada certificado emitido.

A história da revogação de certificados tem sido, historicamente, conturbada.  
A primeira solução proposta foram as **Listas de Revogação de Certificados (CRLs — *Certificate Revocation Lists*)**, que permitiam às CAs manterem uma lista de certificados revogados, aqueles que não eram mais considerados válidos.  
O problema com as CRLs é que elas podem crescer bastante e precisam ser constantemente consultadas.

As CRLs foram descontinuadas em favor do **Protocolo de Status de Certificado Online (OCSP — *Online Certificate Status Protocol*)**, que são interfaces web simples que podem ser consultadas para verificar se um certificado foi revogado ou não.  
O OCSP tem seus próprios problemas:

* exige que as CAs mantenham um serviço altamente disponível que possa responder aos pedidos OCSP,
* vaza informações de tráfego web para as CAs,
* e os navegadores frequentemente optam por ignorar as solicitações OCSP que expiram (para não prejudicar a experiência do usuário).

A solução atual é complementar o OCSP com o **OCSP stapling**:

* o próprio website fica encarregado de consultar a CA por um status assinado do seu certificado e anexa (*staples*) essa resposta ao seu certificado durante o *handshake* TLS.  
  Eu reviso as três soluções na Figura 9.9:

<IMAGEM>

\*Figura 9.9: A revogação de certificados na web teve três soluções populares:

* Listas de Revogação de Certificados (CRLs),
* Protocolo de Status de Certificado Online (OCSP),
* e OCSP stapling.\*

A revogação de certificados pode não parecer uma funcionalidade de segurança primordial (especialmente para sistemas menores em comparação com a World Wide Web) — até que um certificado seja comprometido.  
Assim como um cinto de segurança de carro, a revogação de certificados é um recurso de segurança inútil na maior parte do tempo, mas pode salvar vidas em casos raros.  
Na segurança, chamamos isso de **defesa em profundidade** (*defense in depth*).

**NOTA:**  
Para a web, a revogação de certificados provou ser uma boa decisão.  
Em 2014, o bug Heartbleed acabou sendo um dos mais devastadores na história do SSL e TLS.  
A implementação SSL/TLS mais utilizada (OpenSSL) apresentava um bug de leitura além do buffer (ler além do limite de um array), permitindo que qualquer um enviasse uma mensagem especialmente construída a qualquer servidor OpenSSL e recebesse um dump de sua memória — frequentemente revelando suas chaves privadas de longo prazo.

Ainda assim, se uma CA realmente agir de forma maliciosa, ela pode optar por não revogar certificados maliciosos ou por não relatá-los.  
O problema é que estamos confiando cegamente em um número não desprezível de atores (as CAs) para fazerem a coisa certa.  
Para resolver esse problema em larga escala, o **Certificate Transparency** foi proposto em 2012 pelo Google.

A ideia por trás do Certificate Transparency é forçar as CAs a adicionarem cada certificado emitido a um enorme log de certificados, visível a todos.  
Para isso, navegadores como o Chrome agora rejeitam certificados se eles não incluírem provas de inclusão em um log público.  
Essa transparência permite que você verifique se um certificado foi emitido de forma incorreta para um domínio que você possui (não deveria haver outros certificados além dos que você mesmo solicitou no passado).

Observe que o Certificate Transparency depende de pessoas monitorando os logs de seus próprios domínios para detectar certificados inválidos **depois do fato**.  
As CAs também precisam reagir rapidamente e revogar certificados mal emitidos assim que forem detectados.  
Em casos extremos, os navegadores às vezes removem CAs mal comportadas de seus repositórios de confiança (*trust stores*).  
O Certificate Transparency, portanto, não é tão poderoso quanto o **pinning** de certificado ou de chave pública, que mitiga diretamente comportamentos indevidos das CAs.

**9.4 Outros protocolos de transporte seguro**

Você já aprendeu sobre o TLS, que é o protocolo mais popular para criptografar comunicações.  
Mas ainda não terminamos. O TLS não é o único na categoria de protocolos de transporte seguro.  
Muitos outros protocolos existem, e você provavelmente já está usando alguns deles.  
Ainda assim, a maioria desses protocolos são variantes do TLS, personalizadas para suportar casos de uso específicos.  
Esse é o caso, por exemplo, dos seguintes:

* **SSH (Secure Shell)** — O protocolo e aplicação mais amplamente utilizados para conectar-se com segurança a um terminal remoto em outra máquina.
* **WPA (Wi-Fi Protected Access)** — O protocolo mais popular para conectar dispositivos a pontos de acesso de rede privada ou à internet.
* **IPSec** — Um dos protocolos de rede virtual privada (*VPNs*) mais populares para conectar diferentes redes privadas entre si.  
  É amplamente usado por empresas para interligar redes de escritórios.  
  Como o próprio nome indica, atua na camada IP e é frequentemente encontrado em roteadores, firewalls e outros dispositivos de rede.  
  Outra VPN popular é o **OpenVPN**, que faz uso direto do TLS.

Todos esses protocolos geralmente reimplementam o paradigma de *handshake*/pós-*handshake* e adicionam alguns temperos próprios.  
Reinventar a roda não está livre de problemas — por exemplo, vários dos protocolos Wi-Fi já foram quebrados.

Para finalizar este capítulo, quero apresentar a você o **Noise protocol framework**.  
O Noise é uma alternativa muito mais moderna ao TLS.

**9.5 O framework de protocolo Noise: Uma alternativa moderna ao TLS**

O TLS está agora bastante maduro e é considerado uma solução sólida na maioria dos casos, devido à atenção que recebe.  
Ainda assim, o TLS adiciona muita complexidade às aplicações que o utilizam, por razões históricas, restrições de compatibilidade retroativa e complexidade geral.

De fato, em muitos cenários onde você controla todos os pontos finais (*endpoints*), talvez não precise de todos os recursos que o TLS oferece.  
A próxima melhor solução chama-se **Noise protocol framework**.

O framework de protocolo Noise remove a complexidade de execução do TLS ao evitar qualquer negociação em tempo de execução durante o *handshake*.  
Um cliente e um servidor que utilizam Noise seguem um protocolo linear que não ramifica.  
Em contraste com o TLS, que pode seguir diversos caminhos dependendo das informações contidas nas diferentes mensagens do *handshake*, o Noise desloca toda a complexidade para a fase de design.

Os desenvolvedores que desejam usar o framework Noise devem decidir qual instância ad hoc do framework desejam que sua aplicação utilize.  
(É por isso que é chamado de *framework* de protocolo e não de protocolo.)  
Assim, eles precisam decidir:

* quais algoritmos criptográficos serão utilizados,
* qual lado da conexão será autenticado,
* se uma pré-chave compartilhada será usada,
* e assim por diante.

Após isso, o protocolo é implementado e se transforma em uma série rígida de mensagens, o que pode ser um problema caso seja necessário atualizar o protocolo posteriormente mantendo a compatibilidade retroativa com dispositivos que não podem ser atualizados.

**9.5.1 Os muitos handshakes do Noise**

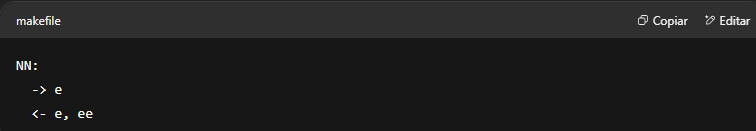
O framework de protocolo Noise oferece diferentes **padrões de handshake** que você pode escolher.  
Os padrões de handshake geralmente vêm com um nome que indica o que está acontecendo.

Por exemplo, o padrão de handshake **IK** indica que:

* a chave pública do cliente é enviada como parte do *handshake* (o primeiro **I** vem de *immediate*),
* e que a chave pública do servidor é conhecida previamente pelo cliente (o **K** vem de *known*).

Uma vez escolhido um padrão de *handshake*, aplicações que o utilizam jamais tentarão executar qualquer outro padrão de *handshake*.  
Em oposição ao TLS, isso torna o Noise um protocolo simples e linear na prática.

No restante desta seção, usarei um padrão de *handshake* chamado **NN** para explicar como o Noise funciona.  
Ele é simples o suficiente para ser explicado, mas inseguro porque os dois **N** indicam que nenhuma autenticação ocorre em ambos os lados.  
Na notação do Noise, o padrão é escrito assim:



Cada linha representa um padrão de mensagem, e a seta indica a direção da mensagem.  
Cada padrão de mensagem é uma sucessão de **tokens** (aqui, há apenas dois: e e ee) que dita o que ambos os lados da conexão precisam fazer:

* -> e — Significa que o cliente deve gerar um par de chaves efêmero e enviar a chave pública ao servidor.  
  O servidor interpreta esta mensagem de forma diferente: deve receber uma chave pública efêmera e armazená-la.
* <- e, ee — Significa que o servidor deve gerar um par de chaves efêmero e enviar sua chave pública ao cliente,  
  então deve executar uma troca de chaves Diffie-Hellman (DH) com:
  + a chave efêmera do cliente (o primeiro e),
  + e sua própria chave efêmera (o segundo e).  
    Por sua vez, o cliente deve receber a chave pública efêmera do servidor e também realizar o DH.

**NOTA:**  
O Noise usa uma combinação de tokens definidos para especificar diferentes tipos de *handshake*.  
Por exemplo:

* o token s indica uma chave estática (outro nome para chave de longo prazo),
* o token es indica que ambos os participantes devem realizar um DH usando a chave efêmera do cliente e a chave estática do servidor.

Ao final de cada padrão de mensagem (-> e e <- e, ee), o emissor também pode transmitir um *payload*.  
Se uma troca DH já tiver ocorrido anteriormente (o que não é o caso no primeiro padrão de mensagem -> e), o *payload* é criptografado e autenticado.  
Ao final do *handshake*, ambos os participantes derivam um conjunto de chaves simétricas e começam a criptografar as comunicações de forma semelhante ao TLS.

**9.5.2 Um handshake com Noise**

Uma particularidade do Noise é que ele autentica continuamente sua transcrição de *handshake*.  
Para conseguir isso, ambos os lados mantêm duas variáveis:

* um **hash** (h),
* e uma **chave de encadeamento** (ck — *chaining key*).

Cada mensagem enviada ou recebida é processada no hash, atualizado com o valor anterior de h.  
Eu ilustro isso na Figura 9.10:

<IMAGEM>

*Figura 9.10: No framework de protocolo Noise, cada lado da conexão mantém um digest h de todas as mensagens que foram enviadas e recebidas durante o handshake.  
Quando uma mensagem é enviada e criptografada com um algoritmo AEAD (Authenticated Encryption with Associated Data), o valor atual de h é usado como dados associados a fim de autenticar o handshake até aquele ponto.*

Ao final de cada padrão de mensagem (-> e e <- e, ee), um *payload* (potencialmente vazio) é criptografado com um algoritmo AEAD (criação autenticada com dados associados — discutido no capítulo 4).  
Quando isso ocorre, o valor h é autenticado pelo campo de dados associados do AEAD.  
Isso permite que o Noise verifique continuamente que ambos os lados da conexão estão vendo exatamente a mesma sequência de mensagens e na mesma ordem.

Além disso, sempre que uma troca DH ocorre (várias podem acontecer durante um *handshake*), sua saída é combinada com a chave de encadeamento anterior (ck) no HKDF,  
o que deriva uma nova chave de encadeamento e um novo conjunto de chaves simétricas para autenticar e criptografar as mensagens subsequentes.  
Eu ilustro isso na Figura 9.11:

<IMAGEM>

*Figura 9.11: No framework de protocolo Noise, cada lado da conexão mantém uma chave de encadeamento ck.  
Este valor é usado para derivar uma nova chave de encadeamento e novas chaves de criptografia a cada vez que uma troca DH é realizada.*

Isso faz do Noise um protocolo simples em tempo de execução; não há bifurcações e ambos os lados da conexão simplesmente seguem o que precisam fazer.  
Bibliotecas que implementam o Noise também são extremamente simples e acabam tendo algumas centenas de linhas, em comparação com centenas de milhares de linhas das bibliotecas TLS.  
Enquanto o Noise é mais complexo de ser utilizado e exige desenvolvedores que compreendam como o Noise funciona para integrá-lo em uma aplicação,  
ele é uma forte alternativa ao TLS.