**Trocas de Chaves**

Agora estamos entrando no reino da criptografia assimétrica (também chamada de criptografia de chave pública) com nosso primeiro primitivo criptográfico assimétrico: a troca de chaves. Uma troca de chaves é, como o nome sugere, uma troca de chaves. Por exemplo, Alice envia uma chave para Bob, e Bob envia uma chave para Alice. Isso permite que os dois pares concordem em um segredo compartilhado, que pode então ser usado para criptografar comunicações com um algoritmo de criptografia autenticada.

**AVISO** Como sugeri na introdução deste livro, há muito mais matemática envolvida na criptografia assimétrica; portanto, os próximos capítulos serão um pouco mais difíceis para alguns leitores. Não desanime! O que você aprenderá neste capítulo será útil para compreender muitos outros primitivos baseados nos mesmos fundamentos.

Este capítulo cobre:

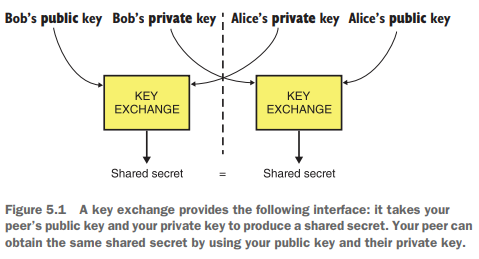
* O que são trocas de chaves e como elas podem ser úteis
* As trocas de chaves Diffie-Hellman e Diffie-Hellman de Curvas Elípticas
* Considerações de segurança ao usar trocas de chaves

**5.1 O que são trocas de chaves?**

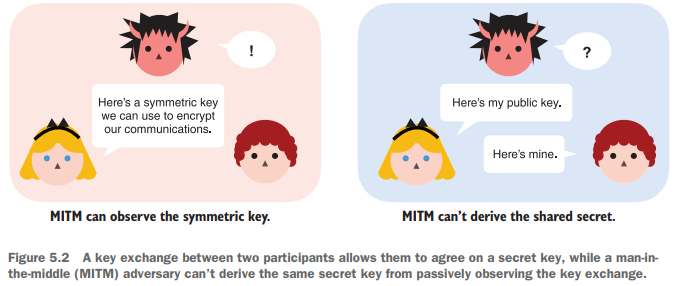
Vamos começar analisando um cenário onde Alice e Bob desejam se comunicar de forma privada, mas nunca conversaram antes. Isso motivará o que as trocas de chaves podem desbloquear na situação mais simples.

Para criptografar comunicações, Alice pode usar o primitivo de criptografia autenticada que você aprendeu no capítulo 4. Para isso, Bob precisa conhecer a mesma chave simétrica, então Alice pode gerar uma e enviá-la para Bob. Depois disso, eles podem simplesmente usar a chave para criptografar suas comunicações. Mas e se um adversário estiver espionando passivamente a conversa deles? Agora o adversário tem a chave simétrica e pode descriptografar todo o conteúdo criptografado que Alice e Bob estão enviando um ao outro! É aqui que usar uma troca de chaves pode ser interessante para Alice e Bob (e para nós mesmos no futuro). Ao usar uma troca de chaves, eles podem obter uma chave simétrica que um observador passivo não poderá reproduzir.

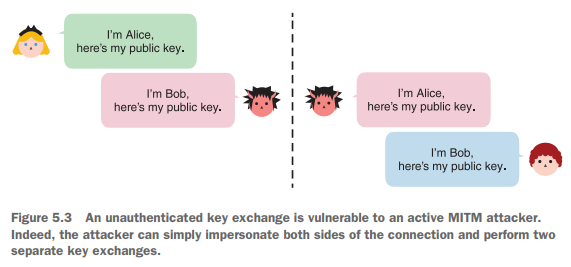
Uma troca de chaves começa com Alice e Bob gerando algumas chaves. Para isso, ambos usam um algoritmo de geração de chaves, que gera um par de chaves: uma chave privada (ou chave secreta) e uma chave pública. Alice e Bob então enviam suas respectivas chaves públicas um para o outro. Público aqui significa que adversários podem observar essas chaves sem consequências. Alice então usa a chave pública de Bob com sua própria chave privada para calcular o segredo compartilhado. Bob pode, de forma semelhante, usar sua chave privada com a chave pública de Alice para obter o mesmo segredo compartilhado. Eu ilustro isso na figura 5.1.



Sabendo como uma troca de chaves funciona de forma geral, podemos agora voltar ao nosso cenário inicial e ver como isso ajuda. Ao iniciar sua comunicação com uma troca de chaves, Alice e Bob produzem um segredo compartilhado para usar como chave para um primitivo de criptografia autenticada. Como qualquer adversário man-in-the-middle (MITM) que observar a troca não pode derivar o mesmo segredo compartilhado, eles não poderão descriptografar as comunicações. Eu ilustro isso na figura 5.2.



Observe que o MITM aqui é passivo; um MITM ativo não teria problemas em interceptar a troca de chaves e se passar por ambos os lados. Nesse ataque, Alice e Bob efetivamente realizariam uma troca de chaves com o MITM, ambos pensando que concordaram em uma chave um com o outro. O motivo disso ser possível é que nenhum de nossos personagens tem uma forma de verificar de quem realmente pertence a chave pública que recebem. A troca de chaves é não autenticada! Eu ilustro o ataque na figura 5.3.



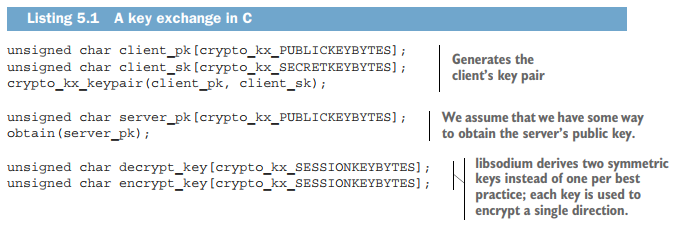
Agora vamos analisar um cenário diferente para motivar as trocas de chaves autenticadas. Imagine que você deseja executar um serviço que fornece a hora atual. No entanto, você não quer que essa informação seja modificada por um adversário MITM. Sua melhor opção é autenticar suas respostas usando os códigos de autenticação de mensagens (MACs) que você aprendeu no capítulo 3. Como os MACs exigem uma chave, você poderia simplesmente gerar uma e compartilhá-la manualmente com todos os seus usuários. Mas então, qualquer usuário agora possui a chave MAC que você está usando com os outros e pode algum dia usá-la para realizar o ataque MITM discutido anteriormente contra outra pessoa. Você poderia configurar uma chave diferente por usuário, mas isso também não é ideal. Para cada novo usuário que deseja se conectar ao seu serviço, você precisaria provisionar manualmente tanto seu serviço quanto o usuário com uma nova chave MAC. Não seria muito melhor se você não tivesse que fazer nada no lado do servidor?

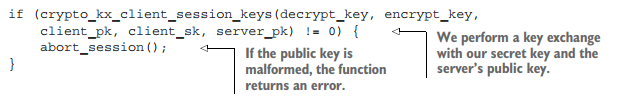
As trocas de chaves podem ajudar aqui! O que você poderia fazer é fazer seu serviço gerar um par de chaves de troca de chaves e provisionar qualquer novo usuário com a chave pública do serviço. Isso é conhecido como uma troca de chaves autenticada; seus usuários conhecem a chave pública do servidor e, portanto, um adversário MITM ativo não pode se passar por esse lado da troca de chaves. O que uma pessoa mal-intencionada pode fazer, entretanto, é realizar sua própria troca de chaves (já que o lado do cliente da conexão não é autenticado). Aliás, quando ambos os lados são autenticados, chamamos isso de troca de chaves mutuamente autenticada.

Esse cenário é extremamente comum, e o primitivo de troca de chaves permite que ele escale bem com o aumento de usuários. Mas esse cenário não escala bem se o número de serviços também aumenta! A internet é um bom exemplo disso. Temos muitos navegadores tentando se comunicar com segurança com muitos sites. Imagine se você tivesse que codificar a chave pública de todos os sites que um dia poderá visitar em seu navegador e o que acontece quando mais sites entram em operação?

Embora as trocas de chaves sejam úteis, elas não escalam bem em todos os cenários sem seu primitivo irmão — a assinatura digital. Mas isso é só uma prévia. No capítulo 7, você aprenderá sobre esse novo primitivo criptográfico e como ele ajuda a escalar a confiança em um sistema. As trocas de chaves raramente são usadas diretamente na prática. Elas são frequentemente apenas blocos de construção de um protocolo mais complexo. Dito isso, ainda podem ser úteis por si só em certas situações (por exemplo, como vimos anteriormente contra adversários passivos).

Vejamos agora como você usaria uma primitiva criptográfica de troca de chaves na prática. libsodium é uma das bibliotecas C/C++ mais conhecidas e amplamente utilizadas. A listagem a seguir mostra como você usaria libsodium na prática para realizar uma troca de chaves.





O libsodium oculta muitos detalhes do desenvolvedor, ao mesmo tempo em que expõe interfaces seguras. Neste caso, o libsodium utiliza o algoritmo de troca de chaves X25519, sobre o qual você aprenderá mais adiante neste capítulo. No restante deste capítulo, você aprenderá sobre os diferentes padrões usados ​​para trocas de chaves, bem como seu funcionamento interno.

**5.2 A troca de chaves Diffie-Hellman (DH)**

Em 1976, Whitfield Diffie e Martin E. Hellman escreveram seu artigo seminal sobre o algoritmo de troca de chaves Diffie-Hellman (DH) intitulado “New Directions in Cryptography”. Que título! O DH foi o primeiro algoritmo de troca de chaves inventado e uma das primeiras formalizações de um algoritmo criptográfico de chave pública. Nesta seção, apresento as bases matemáticas deste algoritmo, explico como ele funciona e, finalmente, falo sobre os padrões que especificam como usá-lo em uma aplicação criptográfica.

**5.2.1 Teoria dos grupos**

A troca de chaves DH é construída sobre um campo da matemática chamado teoria dos grupos, que é a base da maior parte da criptografia de chave pública atualmente. Por essa razão, passarei algum tempo neste capítulo fornecendo a você o básico sobre teoria dos grupos. Farei o meu melhor para fornecer boas intuições sobre como esses algoritmos funcionam, mas não há como evitar: haverá um pouco de matemática.

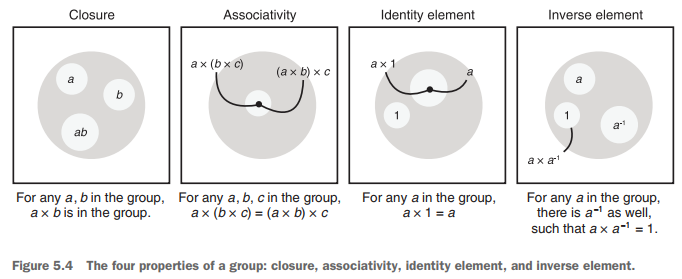
Vamos começar com a pergunta óbvia: o que é um grupo? São duas coisas:

* Um conjunto de elementos
* Uma operação binária especial (como + ou ×) definida sobre esses elementos

Se o conjunto e a operação conseguirem satisfazer algumas propriedades, então temos um grupo. E, se temos um grupo, então podemos fazer coisas mágicas... (mais sobre isso depois). Observe que o DH funciona em um grupo multiplicativo: um grupo onde a multiplicação é usada como operação binária definida. Por isso, o restante das explicações usa um grupo multiplicativo como exemplo. Frequentemente, também omitirei o símbolo × (por exemplo, escreverei a × b como ab).

Preciso ser um pouco mais específico aqui. Para que o conjunto e sua operação sejam um grupo, eles precisam das seguintes propriedades (como de costume, ilustro essas propriedades de forma mais visual na figura 5.4 para fornecer mais material para você assimilar este novo conceito):

* **Fechamento** — Operar sobre dois elementos resulta em outro elemento do mesmo conjunto. Por exemplo, para dois elementos do grupo a e b, a × b resulta em outro elemento do grupo.



* **Associatividade** — Operar sobre vários elementos ao mesmo tempo pode ser feito em qualquer ordem. Por exemplo, para três elementos do grupo a, b e c, então a(bc) e (ab)c resultam no mesmo elemento do grupo.
* **Elemento identidade** — Operar com este elemento não altera o resultado do outro operando. Por exemplo, podemos definir o elemento identidade como 1 em nosso grupo multiplicativo. Para qualquer elemento do grupo a, temos a × 1 = a.
* **Elemento inverso** — Existindo como um inverso para todos os elementos do grupo. Por exemplo, para qualquer elemento do grupo a, existe um elemento inverso a⁻¹ (também escrito como 1/a) tal que a × a⁻¹ = 1.

Posso imaginar que minha explicação de um grupo pode ser um pouco abstrata, então vejamos o que o DH usa como grupo na prática. Primeiro, o DH usa um grupo composto pelo conjunto dos inteiros estritamente positivos: 1, 2, 3, 4, ..., p – 1, onde p é um número primo e 1 é o elemento identidade. Padrões diferentes especificam números diferentes para p, mas intuitivamente, ele precisa ser um número primo grande para que o grupo seja seguro.

Em segundo lugar, o DH usa a multiplicação modular como operação especial. Antes de poder explicar o que é multiplicação modular, preciso explicar o que é aritmética modular.

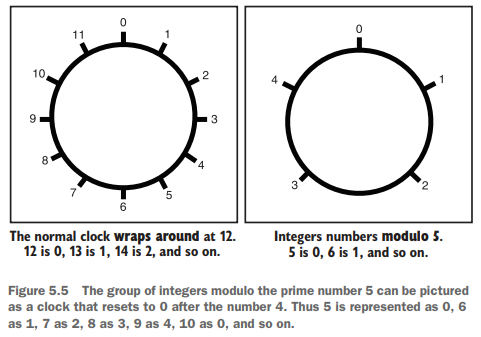
**Números primos**

Um número primo é um número que só pode ser dividido por 1 ou por ele mesmo. Os primeiros números primos são 2, 3, 5, 7, 11, e assim por diante. Números primos estão por toda parte na criptografia assimétrica! E, felizmente, temos algoritmos eficientes para encontrar grandes primos. Para acelerar as coisas, a maioria das bibliotecas criptográficas procura pseudo-primos (números que têm alta probabilidade de serem primos). Interessantemente, tais otimizações já foram quebradas várias vezes no passado; a ocorrência mais infame foi em 2017, quando a vulnerabilidade ROCA revelou mais de um milhão de dispositivos gerando primos incorretos para suas aplicações criptográficas.

Em segundo lugar, DH usa a multiplicação modular como uma operação especial. Antes de explicar o que é multiplicação modular, preciso explicar o que é aritmética modular. A aritmética, intuitivamente, trata de números que "envolvem" um certo número chamado módulo. Por exemplo, se definirmos o módulo como 5, dizemos que os números após 5 retornam a 1; por exemplo, 6 se torna 1, 7 se torna 2 e assim por diante. (Também notamos 5 como 0, mas como não está em nosso grupo multiplicativo, não nos importamos muito com isso.) A maneira matemática de expressar a aritmética modular é pegar o resto de um número e sua divisão euclidiana com o módulo. Tomemos, por exemplo, o número 7 e escrevamos sua divisão euclidiana com 5 como 7 = 5 × 1 + 2. Observe que o resto é 2. Então dizemos que 7 = 2 mod 5 (às vezes escrito como 7 ≡ 2 (mod 5)). Esta equação pode ser lida como 7 é congruente a 2 módulo 5. Similarmente

* 8 = 1 mod 7
* 54 = 2 mod 13
* 170 = 0 mod 17
* and so on

A maneira clássica de visualizar tal conceito é com um relógio. A Figura 5.5 ilustra esse conceito.



Uma multiplicação modular é bastante natural para definir esse conjunto de números. Tomemos a seguinte multiplicação como exemplo:

3 × 2 = 6

Com o que você aprendeu anteriormente, você sabe que 6 é congruente a 1 módulo 5 e, portanto, a equação pode ser reescrita como:

3 × 2 = 1 mod 5

Bem direto, não é? Observe que a equação anterior nos diz que 3 é o inverso de 2 e vice-versa. Também poderíamos escrever o seguinte:

3–1 = 2 mod 5

Quando o contexto é claro, a parte do módulo (mod 5 aqui) é frequentemente omitida nas equações. Portanto, não se surpreenda se eu às vezes omiti-la neste livro.

NOTA: Acontece que, quando usamos os números positivos módulo um número primo

apenas o elemento zero não possui inverso. (De fato, você consegue encontrar um elemento b tal que 0 × b = 1 mod 5?) Esta é a razão pela qual não incluímos

zero como um dos nossos elementos no grupo.

OK, agora temos um grupo, que inclui o conjunto de inteiros estritamente positivos 1, 2, ···, p – 1

para p um número primo, juntamente com a multiplicação modular. O grupo que formamos também

acontece ser duas coisas:

* Comutativo — A ordem das operações não importa. Por exemplo, dados dois

elementos do grupo a e b, então ab = ba. Um grupo que possui essa propriedade é frequentemente

chamado de grupo de Galois.

* Um corpo finito — Um grupo de Galois que possui mais propriedades, bem como uma operação

adicional (em nosso exemplo, também podemos somar números).

Devido ao último ponto, DH definido sobre esse tipo de grupo às vezes é chamado de Diffie-Hellman de Corpo Finito (FFDH). Se você entende o que é um grupo (e certifique-se de entender

antes de continuar lendo), então um subgrupo é apenas um grupo contido dentro do seu grupo original. Ou seja, é um subconjunto dos elementos do grupo. Operar sobre elementos do

subgrupo resulta em outro elemento do subgrupo, e cada elemento do subgrupo tem um

inverso no subgrupo, etc.

Um subgrupo cíclico é um subgrupo que pode ser gerado a partir de um único gerador (ou

base). Um gerador gera um subgrupo cíclico multiplicando-se repetidamente. Por

exemplo, o gerador 4 define um subgrupo composto pelos números 1 e 4:

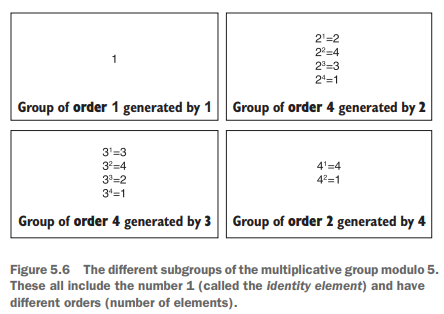
* 4 mod 5 = 4
* 4 × 4 mod 5 = 1
* 4 × 4 × 4 mod 5 = 4 (recomeçamos do início)
* 4 × 4 × 4 × 4 mod 5 = 1
* e assim por diante

NOTA: Também podemos escrever 4 × 4 × 4 como 43.

Acontece que, quando nosso módulo é primo, cada elemento do nosso grupo é um gerador

de um subgrupo. Esses diferentes subgrupos podem ter tamanhos diferentes, que chamamos de ordens.

Ilustro isso na figura 5.6



Tudo bem, agora você entendeu.

* Um grupo é um conjunto de números com uma operação binária que respeita algumas propriedades (fechamento, associatividade, elemento identidade, elemento inverso).
* DH funciona no grupo de Galois (um grupo com comutatividade), formado pelo conjunto de números estritamente positivos até um número primo (não incluído) e a multiplicação modular.
* Em um grupo DH, cada elemento é um gerador de um subgrupo.

Os grupos são o centro de uma enorme quantidade de primitivas criptográficas diferentes. É importante ter boas intuições sobre teoria de grupos se você quiser entender como outras primitivas criptográficas funcionam.

**5.2.2 O problema do logaritmo discreto: A base do Diffie-Hellman**

A segurança da troca de chaves DH baseia-se no problema do logaritmo discreto em um grupo, um problema que se acredita ser difícil de resolver. Nesta seção, apresento brevemente esse problema.

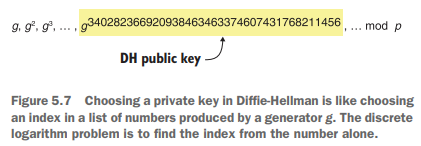
Imagine que eu pegue um gerador, digamos 3, e lhe forneça um elemento aleatório dentre aqueles que ele pode gerar, digamos 2 = 3ˣ mod 5 para algum x desconhecido por você. Perguntar a você "qual é o x?" é o mesmo que pedir que você encontre o logaritmo discreto de 2 na base 3. Assim, o problema do logaritmo discreto em nosso grupo consiste em descobrir quantas vezes multiplicamos o gerador por ele mesmo para produzir um determinado elemento do grupo. Este é um conceito importante! Reserve alguns minutos para pensar sobre ele antes de continuar.

Em nosso grupo de exemplo, você pode rapidamente descobrir que 3 é a resposta (de fato, 3³ = 2 mod 5). Mas se escolhermos um número primo muito maior que 5, as coisas ficam muito mais complicadas: torna-se difícil de resolver. Este é o ingrediente secreto por trás do Diffie-Hellman.

Agora você já sabe o suficiente para entender como gerar um par de chaves no DH:

1. Todos os participantes devem concordar com um número primo grande p e um gerador g.
2. Cada participante gera um número aleatório x, que se torna sua chave privada.
3. Cada participante deriva sua chave pública como gˣ mod p.

O fato de o problema do logaritmo discreto ser difícil significa que ninguém deve ser capaz de recuperar a chave privada a partir da chave pública. Eu ilustro isso na figura 5.7.



Embora tenhamos algoritmos para calcular logaritmos discretos, eles não são eficientes na prática. Por outro lado, se eu lhe der a solução x para o problema, você possui algoritmos extremamente eficientes à sua disposição para verificar que, de fato, eu lhe forneci a solução correta: gˣ mod p. Se você estiver interessado, a técnica de ponta para calcular a exponenciação modular chama-se *square and multiply*. Esse algoritmo computa o resultado de forma eficiente, percorrendo o x bit a bit.

**NOTA** Como em tudo na criptografia, não é impossível encontrar uma solução simplesmente tentando adivinhar. No entanto, escolhendo parâmetros suficientemente grandes (aqui, um número primo grande), é possível reduzir a eficácia de tal busca por uma solução a chances negligenciáveis. Isso significa que mesmo após centenas de anos de tentativas aleatórias, suas chances de encontrar uma solução ainda seriam estatisticamente próximas de zero.

Ótimo. Como usamos toda essa matemática no nosso algoritmo de troca de chaves DH? Imagine que:

* Alice possui uma chave privada a e uma chave pública A = gᵃ mod p.
* Bob possui uma chave privada b e uma chave pública B = gᵇ mod p.

Com o conhecimento da chave pública de Bob, Alice pode calcular o segredo compartilhado como Bᵃ mod p. Bob pode fazer um cálculo semelhante com a chave pública de Alice e sua própria chave privada: Aᵇ mod p. Naturalmente, podemos ver que esses dois cálculos acabam computando o mesmo número:

Bᵃ = (gᵇ)ᵃ = gᵃᵇ = (gᵃ)ᵇ = Aᵇ mod p

E essa é a mágica do DH. Do ponto de vista de um observador externo, apenas observar as chaves públicas A e B não ajuda em nada a calcular o resultado da troca de chaves gᵃᵇ mod p. A seguir, você aprenderá como aplicações do mundo real utilizam este algoritmo e os diferentes padrões existentes.

**Diffie-Hellman Computacional e Decisório**

A propósito, em criptografia teórica, a ideia de que observar ga mod p e gb mod p não ajuda a calcular gab mod p é chamada de Suposição Computacional de Diffie-Hellman (CDH). Ela é frequentemente confundida com a Suposição Decisória de Diffie-Hellman (DDH), que afirma intuitivamente que, dados ga mod p, gb mod p e z mod p, ninguém deveria ser capaz de adivinhar com segurança se o último elemento é o resultado de uma troca de chaves entre as duas chaves públicas (gab mod p) ou apenas um elemento aleatório do grupo. Ambas são suposições teóricas úteis que têm sido usadas para construir muitos algoritmos diferentes em criptografia.

**5.2.3 Os padrões do Diffie-Hellman**

Agora que você viu como o DH funciona, pode entender que os participantes precisam concordar com um conjunto de parâmetros, especificamente com um número primo p e um gerador g. Nesta seção, você aprenderá como as aplicações do mundo real escolhem esses parâmetros e os diferentes padrões existentes.

Primeiro de tudo está o número primo p. Como eu afirmei anteriormente, quanto maior, melhor. Como o DH baseia-se no problema do logaritmo discreto, sua segurança está diretamente correlacionada com os melhores ataques conhecidos para o problema. Qualquer avanço nessa área pode enfraquecer o algoritmo. Com o tempo, conseguimos obter uma boa ideia de quão rápidos (ou lentos) esses avanços são e de quanto é segurança suficiente. As melhores práticas atualmente conhecidas são utilizar um número primo de 2048 bits.

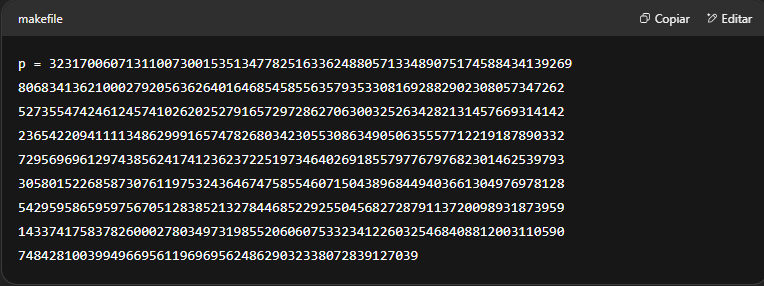
**NOTA** Em geral, o site <https://keylength.com> resume recomendações sobre tamanhos de parâmetros para algoritmos criptográficos comuns. Os resultados são retirados de documentos oficiais produzidos por grupos de pesquisa ou órgãos governamentais como a ANSSI (França), o NIST (EUA) e o BSI (Alemanha). Embora nem sempre concordem, frequentemente convergem para ordens de grandeza semelhantes.

No passado, muitas bibliotecas e softwares frequentemente geravam e fixavam seus próprios parâmetros. Infelizmente, às vezes descobriu-se que eles eram fracos ou, pior ainda, completamente comprometidos. Em 2016, alguém descobriu que o Socat, uma ferramenta popular de linha de comando, havia modificado seu grupo DH padrão com um grupo comprometido um ano antes, levantando a questão se isso havia sido um erro ou uma porta dos fundos intencional. Utilizar grupos DH padronizados pode parecer uma ideia melhor, mas o DH é um dos infelizes contraexemplos. Apenas alguns meses após o problema do Socat, Antonio Sanso, ao ler o RFC 5114, descobriu que o padrão também especificava grupos DH comprometidos.

Devido a todos esses problemas, protocolos e bibliotecas mais recentes têm convergido para ou descontinuar o DH em favor do Diffie-Hellman de Curvas Elípticas (ECDH) ou utilizar os grupos definidos no melhor padrão, o RFC 7919 (<https://www.rfc-editor.org/info/rfc7919>). Por esta razão, a melhor prática atualmente é usar o RFC 7919, que define vários grupos de tamanhos e níveis de segurança diferentes. Por exemplo, o *ffdhe2048* é o grupo definido pelo módulo primo de 2048 bits:

**Computacional e decisional Diffie-Hellman**  
A propósito, na criptografia teórica, a ideia de que observar gᵃ mod p e gᵇ mod p não ajuda você a calcular gᵃᵇ mod p é chamada de suposição computacional Diffie-Hellman (*computational Diffie-Hellman assumption*, CDH). Muitas vezes é confundida com a suposição decisional Diffie-Hellman (*decisional Diffie-Hellman assumption*, DDH), que, intuitivamente, afirma que, dado gᵃ mod p, gᵇ mod p e z mod p, ninguém deve ser capaz de adivinhar com confiança se o último elemento é o resultado de uma troca de chaves entre as duas chaves públicas (gᵃᵇ mod p) ou apenas um elemento aleatório do grupo. Ambas são suposições teóricas úteis que têm sido usadas para construir muitos algoritmos diferentes na criptografia.

O módulo primo p para *ffdhe2048* é o seguinte (linha quebrada para fins de legibilidade):



E com o gerador g = 2.

**NOTA** É comum escolher o número 2 como gerador, pois computadores são bastante eficientes em multiplicar por 2 usando uma simples instrução de deslocamento à esquerda (<<).

O tamanho do grupo (ou ordem) também é especificado como q = (p – 1)/2. Isso implica que tanto as chaves privadas quanto as chaves públicas terão em torno de 2048 bits em termos de tamanho. Na prática, esses tamanhos são bastante grandes para chaves (compare isso, por exemplo, com chaves simétricas, que normalmente têm 128 bits). Você verá na próxima seção que definir um grupo sobre curvas elípticas permite obter chaves muito menores para o mesmo nível de segurança.

**5.3 A troca de chaves Diffie-Hellman de Curvas Elípticas (ECDH)**

Acontece que o algoritmo DH, que acabamos de discutir, pode ser implementado em diferentes tipos de grupos, não apenas nos grupos multiplicativos módulo um número primo. Também acontece que um grupo pode ser formado a partir de curvas elípticas, um tipo de curvas estudadas na matemática. A ideia foi proposta em 1985 por Neal Koblitz e Victor S. Miller, independentemente, e muito mais tarde, em 2000, foi adotada quando algoritmos criptográficos baseados em curvas elípticas começaram a ser padronizados.

O mundo da criptografia aplicada rapidamente adotou a criptografia de curvas elípticas, pois ela proporcionava chaves muito menores do que a geração anterior de criptografia de chave pública. Comparado com os parâmetros de 2048 bits recomendados no DH, parâmetros de 256 bits eram possíveis com a variante de curvas elípticas do algoritmo.

**5.3.1 O que é uma curva elíptica?**

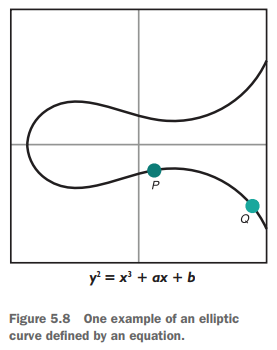
Agora vamos explicar como funcionam as curvas elípticas. Antes de mais nada, é bom entender que curvas elípticas são apenas curvas! Significando que são definidas por todas as coordenadas x e y que satisfazem uma equação. Especificamente, esta equação:

**y² + a₁xy + a₃y = x³ + a₂x² + a₄x + a₆**

para alguns valores de a₁, a₂, a₃, a₄ e a₆. Observe que, para a maioria das curvas práticas hoje, esta equação pode ser simplificada para a forma curta de Weierstrass:

**y² = x³ + ax + b (onde 4a³ + 27b² ≠ 0)**

Enquanto a simplificação não é possível para dois tipos de curvas (chamadas de curvas binárias e curvas de característica 3), estas são pouco usadas, de modo que utilizaremos a forma de Weierstrass no restante deste capítulo.



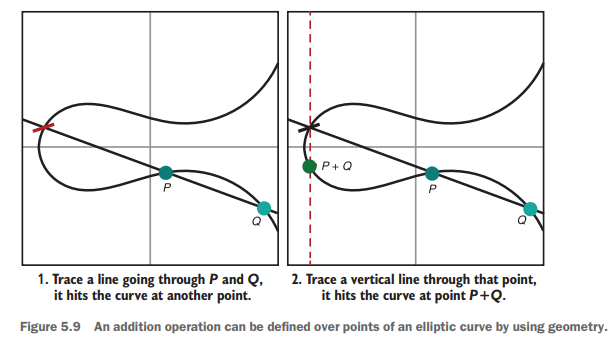
Em algum momento na história das curvas elípticas, descobriu-se que era possível construir um grupo sobre elas. A partir daí, implementar o DH sobre esses grupos foi algo direto. Utilizarei esta seção para explicar a intuição por trás da criptografia de curvas elípticas.

Grupos sobre curvas elípticas são frequentemente definidos como grupos aditivos. Diferentemente dos grupos multiplicativos definidos na seção anterior, o sinal de adição é usado aqui.

**NOTA** Usar adição ou multiplicação não importa muito na prática, é apenas uma questão de preferência. Enquanto a maior parte da criptografia usa notação multiplicativa, a literatura em torno das curvas elípticas gravitou em direção à notação aditiva, e, portanto, é essa que utilizarei ao me referir a grupos de curvas elípticas neste livro.

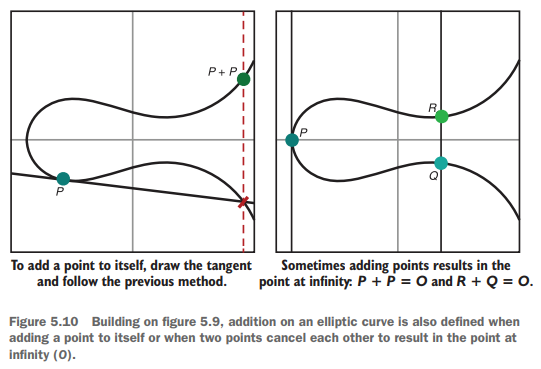
Desta vez, definirei a operação antes de definir os elementos do grupo. Nossa operação de adição é definida da seguinte forma. A figura 5.9 ilustra este processo.

1. Trace uma linha passando pelos dois pontos que deseja somar. A linha intersecta a curva em outro ponto.
2. Trace uma linha vertical a partir desse novo ponto. A linha vertical intersecta a curva em mais um ponto.
3. Este ponto é o resultado da soma dos dois pontos originais.



Existem dois casos especiais em que essa regra não funciona. Vamos defini-los também:

* **Como somar um ponto a ele mesmo?**  
  A resposta é traçar a tangente nesse ponto (em vez de traçar uma linha entre dois pontos).
* **O que acontece se a linha que traçamos no passo 1 (ou passo 2) não intersecta a curva em nenhum outro ponto?**  
  Bem, isso é embaraçoso, e precisamos que este caso especial funcione e produza um resultado. A solução é definir o resultado como um ponto inventado (algo que criamos). Este ponto recém-inventado é chamado de ponto no infinito (que geralmente escrevemos com a letra maiúscula O). A figura 5.10 ilustra esses casos especiais.



Eu sei que este ponto no infinito é algo de um nível de estranheza superior, mas não se preocupe muito com isso. É realmente apenas algo que inventamos para que a operação de adição funcione. Ah, e a propósito, ele se comporta como um zero e é o nosso elemento identidade:

**O + O = O**

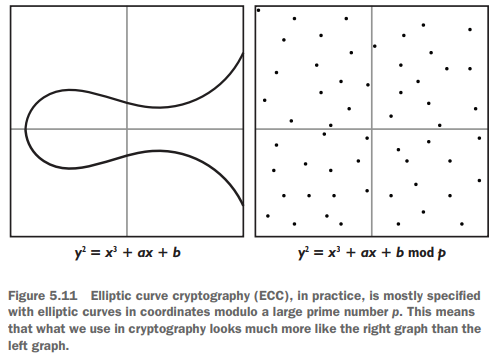
e para qualquer ponto P na curva:

**P + O = P**

Tudo certo. Até agora, vimos que para criar um grupo sobre uma curva elíptica, precisamos de:

* Uma equação de curva elíptica que define um conjunto de pontos válidos.
* Uma definição do que significa adição neste conjunto.
* Um ponto imaginário chamado ponto no infinito.

Sei que é muita informação para absorver, mas ainda falta uma última coisa. A criptografia de curvas elípticas faz uso do tipo de grupo anteriormente discutido, definido sobre um campo finito. Na prática, isso significa que nossas coordenadas são os números 1, 2, ..., p – 1 para algum número primo grande p. Isso deve soar familiar! Por esta razão, ao pensar em criptografia de curvas elípticas, você deve imaginar um gráfico muito mais parecido com o da direita na figura 5.11.



E é isso! Agora temos um grupo sobre o qual podemos fazer criptografia, da mesma forma que tínhamos um grupo formado pelos números (excluindo o zero) módulo um número primo e a operação de multiplicação para o Diffie-Hellman. Como podemos fazer Diffie-Hellman com este grupo definido sobre curvas elípticas? Vamos ver como o logaritmo discreto funciona agora neste grupo.

Vamos pegar um ponto G e somá-lo a ele mesmo x vezes para produzir outro ponto P via a operação de adição que definimos. Podemos escrever isso como P = G + ... + G (x vezes) ou usar uma notação matemática resumida para escrever como:

**P = [x]G**, que se lê "x vezes G".

O problema do logaritmo discreto em curvas elípticas (ECDLP) é encontrar o número x conhecendo apenas P e G.

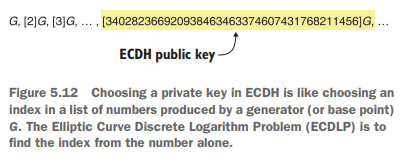
**NOTA** Chamamos **[x]G** de multiplicação escalar, pois x normalmente é chamado de escalar nesses grupos.

**5.3.2 Como funciona a troca de chaves Diffie-Hellman de Curvas Elípticas (ECDH)**

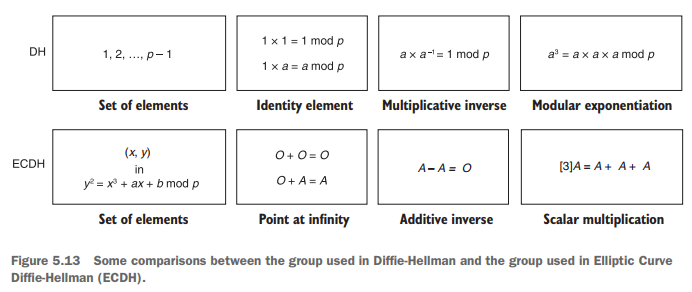
Agora que construímos um grupo sobre curvas elípticas, podemos instanciar o mesmo algoritmo de troca de chaves Diffie-Hellman sobre ele. Para gerar um par de chaves no ECDH:

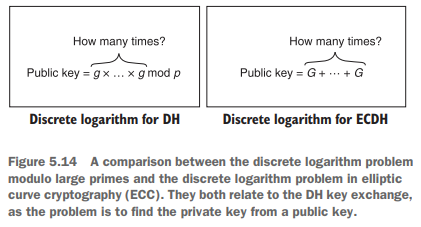
1. Todos os participantes concordam com uma equação de curva elíptica, um campo finito (muito provavelmente um número primo), e um gerador G (normalmente chamado de ponto base na criptografia de curvas elípticas).
2. Cada participante gera um número aleatório x, que se torna sua chave privada.
3. Cada participante deriva sua chave pública como **[x]G**.

Como o problema do logaritmo discreto em curvas elípticas é difícil, você adivinhou, ninguém deve ser capaz de recuperar sua chave privada apenas observando sua chave pública. Eu ilustro isso na figura 5.12.



Tudo isso pode ser um pouco confuso, já que a operação que definimos para nosso grupo DH era a multiplicação, e para uma curva elíptica agora usamos adição. De novo, essas distinções não importam nem um pouco porque são equivalentes. Você pode ver uma comparação na figura 5.13.

Agora você deve estar convencido de que a única coisa que importa para a criptografia é que tenhamos um grupo definido com sua operação, e que o logaritmo discreto para esse grupo seja difícil. Para completar, a figura 5.14 mostra a diferença entre o problema do logaritmo discreto nos dois tipos de grupos que vimos.



Uma última observação sobre a teoria: o grupo que formamos sobre curvas elípticas difere do grupo que formamos com inteiros estritamente positivos módulo um número primo. Devido a algumas dessas diferenças, os ataques mais potentes conhecidos contra DH (conhecidos como cálculo de índice ou ataques de peneira de campo numérico) não funcionam bem nos grupos de curvas elípticas. Essa é a principal razão pela qual os parâmetros para ECDH podem ser muito menores do que os parâmetros para DH no mesmo nível de segurança.

Certo, terminamos com a teoria. Vamos voltar à definição do ECDH. Imagine que:

* Alice possui uma chave privada **a** e uma chave pública **A = [a]G**.
* Bob possui uma chave privada **b** e uma chave pública **B = [b]G**.

Com o conhecimento da chave pública de Bob, Alice pode calcular o segredo compartilhado como **[a]B**. Bob pode fazer um cálculo semelhante com a chave pública de Alice e sua própria chave privada: **[b]A**. Naturalmente, podemos ver que esses dois cálculos acabam computando o mesmo ponto:

**[a]B = [a][b]G = [ab]G = [b][a]G = [b]A**

Nenhum adversário passivo deve ser capaz de derivar o ponto compartilhado apenas observando as chaves públicas. Soa familiar, não é? A seguir, vamos falar sobre os padrões.

**5.3.3 Os padrões para Diffie-Hellman de Curvas Elípticas**

A criptografia de curvas elípticas manteve sua força total desde que foi apresentada em 1985.

*Os Estados Unidos, o Reino Unido, o Canadá e certas outras nações da OTAN adotaram alguma forma de criptografia de curvas elípticas para sistemas futuros a fim de proteger informações classificadas em seus governos e entre eles.*  
— NSA (“The Case for Elliptic Curve Cryptography”, 2005)

A padronização do ECDH tem sido bastante caótica. Muitos órgãos de padronização trabalharam para especificar um grande número de curvas diferentes, o que foi seguido por muitas discussões acaloradas sobre qual curva era mais segura ou mais eficiente. Uma grande quantidade de pesquisa, liderada principalmente por Daniel J. Bernstein, apontou o fato de que várias curvas padronizadas pelo NIST poderiam potencialmente fazer parte de uma classe de curvas mais fracas, conhecidas apenas pela NSA.

*Já não confio nas constantes. Acredito que a NSA as manipulou por meio de seus relacionamentos com a indústria.*  
— Bruce Schneier (“The NSA Is Breaking Most Encryption on the Internet”, 2013)

Atualmente, a maioria das curvas em uso vem de alguns poucos padrões, e a maioria das aplicações fixou-se em duas curvas: **P-256** e **Curve25519**. No restante desta seção, falarei sobre essas curvas.

O **NIST FIPS 186-4**, “Digital Signature Standard”, inicialmente publicado como um padrão para assinaturas em 2000, contém um apêndice especificando 15 curvas para uso no ECDH. Uma dessas curvas, **P-256**, é a mais amplamente utilizada na internet. A curva também foi especificada em **Standards for Efficient Cryptography (SEC) 2, v2**, “Recommended Elliptic Curve Domain Parameters”, publicado em 2010 sob outro nome: **secp256r1**.

A **P-256** é definida com a equação reduzida de Weierstrass:

**y² = x³ + ax + b mod p**

onde:

* **a = –3**

e

* **b = 41058363725152142129326129780047268409114441015993725554835256314039467401291**

e

* **p = 2²⁵⁶ – 2²²⁴ + 2¹⁹² + 2⁹⁶ – 1**

Isto define uma curva de ordem prima:

**n = 115792089210356248762697446949407573529996955224135760342422259061068512044369**

o que significa que há exatamente **n** pontos na curva (incluindo o ponto no infinito).

O ponto base é especificado como:

**G = (48439561293906451759052585252797914202762949526041747995844080717082404635286,  
36134250956749795798585127919587881956611106672985015071877198253568414405109)**

A curva fornece 128 bits de segurança. Para aplicações que utilizam outros algoritmos criptográficos oferecendo 256 bits de segurança ao invés de 128 bits de segurança (por exemplo, AES com chave de 256 bits), a curva **P-521** está disponível no mesmo padrão para corresponder ao nível de segurança.

O **RFC 7748**, “Elliptic Curves for Security”, publicado em 2016, especifica duas curvas: **Curve25519** e **Curve448**. A **Curve25519** oferece aproximadamente 128 bits de segurança, enquanto a **Curve448** oferece cerca de 224 bits de segurança para protocolos que desejam se proteger contra potenciais avanços nos ataques contra curvas elípticas. Falarei aqui apenas da **Curve25519**, que é uma curva de Montgomery definida pela equação:

**y² = x³ + 486662x² + x mod p**, onde **p = 2²⁵⁵ – 19**

A **Curve25519** tem ordem:

**n = 2²⁵² + 27742317777372353535851937790883648493**

E o ponto base usado é:

**G = (9, 14781619447589544791020593568409986887264606134616475288964881837755586237401)**

A combinação de ECDH com **Curve25519** é frequentemente chamada de **X25519**.

**Podemos confiar em P-256?**

Curiosamente, diz-se que P-256 e outras curvas definidas no FIPS 186-4 são geradas a partir de uma semente. Para P-256, a semente é conhecida como a sequência de bytes 0xc49d360886e704936a6678e1139d26b7819f7e90. Já falei sobre essa noção de números "sem nada na manga" — constantes que visam provar que não havia espaço para backdoors no design do algoritmo. Infelizmente, não há muita explicação por trás da semente P-256 além do fato de que ela é especificada ao longo do parâmetro da curva.

**5.4 Ataques de subgrupos pequenos e outras considerações de segurança**

Hoje, eu recomendaria que você usasse **ECDH** em vez de **DH** devido ao tamanho das chaves, à ausência de ataques fortes conhecidos, à qualidade das implementações disponíveis e ao fato de que as curvas elípticas são fixas e bem padronizadas (ao contrário dos grupos DH, que são bastante desorganizados). Este último ponto é bastante importante! Utilizar **DH** significa potencialmente usar padrões comprometidos (como o **RFC 5114** mencionado anteriormente), protocolos excessivamente permissivos (muitos protocolos, como versões antigas do **TLS**, não especificam quais grupos DH usar), softwares que usam grupos DH personalizados e comprometidos (como o problema do **socat** mencionado anteriormente) e assim por diante.

Se você precisar usar o **Diffie-Hellman**, certifique-se de aderir aos padrões. Os padrões que mencionei anteriormente fazem uso de primos seguros como módulo: primos da forma **p = 2q + 1** onde **q** é outro número primo. A ideia é que grupos dessa forma só tenham dois subgrupos: um pequeno de tamanho 2 (gerado por –1) e um grande de tamanho **q**. (Este é o melhor que se pode obter, a propósito; não existem grupos de ordem prima em **DH**.) A escassez de subgrupos pequenos previne um tipo de ataque conhecido como **ataque de subgrupo pequeno** (mais sobre isso em breve). Primos seguros criam grupos seguros por dois motivos:

* A ordem de um grupo multiplicativo módulo um primo **p** é calculada como **p – 1**.
* A ordem dos subgrupos de um grupo são os fatores da ordem do grupo (este é o **Teorema de Lagrange**).

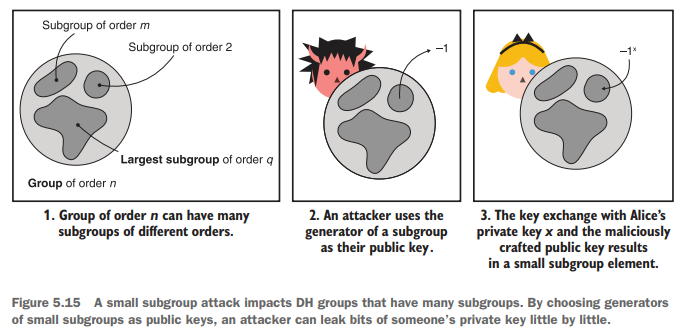
Assim, a ordem de nosso grupo multiplicativo módulo um primo seguro é **p – 1 = (2q + 1) – 1 = 2q**, que tem fatores 2 e **q**, o que significa que seus subgrupos só podem ter ordem 2 ou **q**. Em tais grupos, ataques de subgrupo pequeno não são possíveis porque não há subgrupos pequenos suficientes. Um **ataque de subgrupo pequeno** é um ataque a trocas de chaves no qual um atacante envia várias chaves públicas inválidas para vazar bits de sua chave privada gradualmente, sendo que essas chaves públicas inválidas são geradores de subgrupos pequenos.

Por exemplo, um atacante poderia escolher –1 (o gerador de um subgrupo de tamanho 2) como chave pública e enviá-la para você. Ao realizar sua parte da troca de chaves, o segredo compartilhado resultante será um elemento do subgrupo pequeno (–1 ou 1). Isso ocorre porque você acabou de elevar o gerador do subgrupo pequeno (a chave pública do atacante) à potência de sua chave privada. Dependendo do que você fizer com aquele segredo compartilhado, o atacante poderia adivinhar qual é, e vazar alguma informação sobre sua chave privada.

Com nosso exemplo de chave pública maliciosa, se sua chave privada fosse par, o segredo compartilhado seria 1; se sua chave privada fosse ímpar, o segredo seria –1. Como resultado, o atacante aprendeu um bit de informação: o bit menos significativo de sua chave privada. Muitos subgrupos de tamanhos diferentes podem fornecer mais oportunidades para o atacante aprender mais sobre sua chave privada até que a chave inteira seja recuperada. Eu ilustro este problema na figura 5.15.

Embora sempre seja uma boa prática verificar se a chave pública recebida está no subgrupo correto, nem todas as implementações fazem isso. Em 2016, um grupo de pesquisadores analisou 20 implementações diferentes de **DH** e descobriu que nenhuma estava validando chaves públicas (ver “**Measuring small subgroup attacks against Diffie-Hellman**” de Valenta et al.). Certifique-se de que as implementações de **DH** que você está utilizando o façam! Você pode fazer isso elevando a chave pública à ordem do subgrupo, o que deve devolver o elemento identidade se for um elemento desse subgrupo.

Por outro lado, curvas elípticas permitem grupos de ordem prima. Ou seja, não possuem subgrupos pequenos (além do subgrupo de tamanho 1 gerado pelo elemento identidade) e, portanto, são seguras contra ataques de subgrupo pequeno. Bem... nem tão rápido. Em 2000, Biehl, Meyer e Müller descobriram que ataques de subgrupo pequeno são possíveis mesmo em grupos de ordem prima de curvas elípticas devido a um ataque chamado **ataque de curva inválida**.



A ideia por trás dos ataques de curva inválida é a seguinte: primeiro, as fórmulas para implementar a multiplicação escalar para curvas elípticas que usam a forma reduzida de Weierstrass **y² = x³ + ax + b** (como a **P-256** do NIST) são independentes da variável **b**. Isso significa que um atacante pode encontrar diferentes curvas com a mesma equação, exceto pelo valor de **b**, e algumas dessas curvas terão muitos subgrupos pequenos. Você provavelmente já sabe onde isso vai dar: o atacante escolhe um ponto em outra curva que exibe subgrupos pequenos e o envia a um servidor alvo. O servidor prossegue com a troca de chaves realizando uma multiplicação escalar com o ponto fornecido, efetivamente realizando uma troca de chaves em uma curva diferente. Este truque acaba reabilitando o ataque de subgrupo pequeno, mesmo em curvas de ordem prima.

A forma óbvia de corrigir isso é, novamente, validar chaves públicas. Isso pode ser feito facilmente verificando se a chave pública não é o ponto no infinito e inserindo as coordenadas recebidas na equação da curva para ver se elas descrevem um ponto na curva definida. Infelizmente, em 2015, Jager, Schwenk e Somorovsky mostraram em **“Practical Invalid Curve Attacks on TLS-ECDH”** que várias implementações populares não realizavam essas verificações. Se estiver usando **ECDH**, eu recomendaria usar a troca de chaves **X25519** devido à qualidade do design (que leva em consideração ataques de curva inválida), à qualidade das implementações disponíveis e à resistência contra ataques de temporização por design.

A **Curve25519** tem uma ressalva: ela não é um grupo de ordem prima. A curva tem dois subgrupos: um subgrupo pequeno de tamanho 8 e um subgrupo grande usado para o **ECDH**. Além disso, o design original não prescrevia a validação de pontos recebidos, e as bibliotecas, por sua vez, não implementaram essas verificações. Isso levou a problemas encontrados em diferentes tipos de protocolos que estavam utilizando o primitivo de formas mais exóticas. (Um desses problemas eu encontrei no protocolo de mensagens **Matrix**, do qual falarei no capítulo 11.)

Não verificar chaves públicas pode gerar comportamentos inesperados com o **X25519**. O motivo é que o algoritmo de troca de chaves não tem comportamento contributivo: ele não permite que ambas as partes contribuam para o resultado final da troca de chaves. Especificamente, um dos participantes pode forçar o resultado da troca de chaves a ser todo zero enviando um ponto no subgrupo pequeno como chave pública. O **RFC 7748** menciona esse problema e propõe verificar se o segredo compartilhado resultante não é o valor todo zero, mas deixa a decisão para o implementador! Eu recomendaria garantir que sua implementação realize essa verificação, embora seja improvável que você encontre problemas a menos que use o **X25519** de maneira não padronizada.

Como muitos protocolos dependem da **Curve25519**, isso tem sido um problema para mais do que apenas trocas de chaves. O **Ristretto**, o rascunho de internet que em breve se tornará RFC, é uma construção que adiciona uma camada extra de codificação à **Curve25519**, simulando efetivamente uma curva de ordem prima (ver <https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-01>). A construção vem ganhando força, pois simplifica as suposições de segurança feitas por outros tipos de primitivos criptográficos que desejam se beneficiar da **Curve25519**, mas desejam um campo de ordem prima.

**Resumo**

* Trocas de chaves não autenticadas permitem que duas partes concordem em um segredo compartilhado, evitando que um atacante passivo man-in-the-middle (MITM) consiga derivá-lo.
* Uma troca de chaves autenticada impede que um MITM ativo se passe por um dos lados da conexão, enquanto uma troca de chaves mutuamente autenticada impede que um MITM ativo se passe por ambos os lados.
* Pode-se realizar uma troca de chaves autenticada conhecendo a chave pública da outra parte, mas isso nem sempre escala, e as assinaturas permitirão cenários mais complexos (veja o capítulo 7).
* **Diffie-Hellman (DH)** é o primeiro algoritmo de troca de chaves inventado e ainda é amplamente utilizado.
* O padrão recomendado para usar **DH** é o **RFC 7919**, que inclui vários parâmetros para escolher. A menor opção é o parâmetro primo de 2048 bits recomendado.
* **Diffie-Hellman de Curvas Elípticas (ECDH)** possui tamanhos de chave muito menores que o **DH**. Para 128 bits de segurança, **DH** precisa de parâmetros de 2048 bits, enquanto o **ECDH** precisa de parâmetros de 256 bits.
* As curvas mais amplamente utilizadas para **ECDH** são **P-256** e **Curve25519**. Ambas oferecem 128 bits de segurança. Para 256 bits de segurança, **P-521** e **Curve448** estão disponíveis nos mesmos padrões.
* Certifique-se de que as implementações verifiquem a validade das chaves públicas recebidas, pois chaves inválidas são fonte de muitos bugs.