**Capítulo 6**

**Criptografia assimétrica e criptografia híbrida**

No capítulo 4, você aprendeu sobre criptografia autenticada, um primitivo criptográfico usado para criptografar dados, mas limitado por sua simetria (ambos os lados de uma conexão precisavam compartilhar a mesma chave). Neste capítulo, vou remover essa restrição introduzindo a criptografia assimétrica, um primitivo para criptografar usando a chave de outra pessoa sem conhecer a chave. Sem surpresa, a criptografia assimétrica faz uso de pares de chaves e a criptografia usará chaves públicas em vez de chaves simétricas.

No meio deste capítulo, você verá que a criptografia assimétrica é limitada pelo tamanho dos dados que pode criptografar e pela taxa na qual pode criptografá-los. Para remover este obstáculo, mostrarei como misturar criptografia assimétrica com criptografia autenticada para formar o que chamamos de criptografia híbrida. Vamos começar!

**NOTA**: Para este capítulo, você precisará ter lido o capítulo 4 sobre criptografia autenticada e o capítulo 5 sobre trocas de chaves.

**Este capítulo cobre:**

* Criptografia assimétrica para criptografar segredos para uma chave pública
* Criptografia híbrida para criptografar dados para uma chave pública
* Os padrões para criptografia assimétrica e híbrida

**6.1 O que é criptografia assimétrica?**

O primeiro passo para saber como criptografar uma mensagem é entender a criptografia assimétrica (também chamada de criptografia de chave pública). Nesta seção, você aprenderá sobre este primitivo criptográfico e suas propriedades. Vamos dar uma olhada no seguinte cenário do mundo real: e-mails criptografados.

Você provavelmente sabe que todos os e-mails que você envia são enviados "em texto claro", podendo ser lidos por qualquer pessoa entre você e o provedor de e-mail do destinatário. Isso não é bom. Como consertar isso? Você poderia usar um primitivo criptográfico como o AES-GCM, que você aprendeu no capítulo 4. Para fazer isso, seria necessário configurar um segredo simétrico diferente para cada pessoa que queira enviar mensagens a você.

Mas não dá para esperar que você saiba com antecedência quem vai querer enviar mensagens, e gerar e trocar novas chaves simétricas se tornaria tedioso à medida que mais pessoas quisessem criptografar mensagens para você. É aí que a criptografia assimétrica ajuda, permitindo que qualquer pessoa em posse da sua chave pública criptografe mensagens para você. Além disso, apenas você pode descriptografar essas mensagens usando a chave privada associada, que apenas você possui. Veja a **<IMAGEM>** (Figura 6.1) para uma ilustração da criptografia assimétrica.

Para configurar a criptografia assimétrica, primeiro você precisa gerar um par de chaves via algum algoritmo. Como em qualquer função de configuração para algoritmos criptográficos, o algoritmo de geração de chaves aceita um parâmetro de segurança. Este parâmetro geralmente se traduz em “qual o tamanho desejado para suas chaves?”, onde tamanhos maiores significam mais segurança. A **<IMAGEM>** (Figura 6.2) ilustra este passo.

O algoritmo de geração de chaves gera um par de chaves composto por duas partes diferentes:

* A chave pública (como o nome indica) pode ser publicada e compartilhada sem muitas preocupações.
* A chave privada deve permanecer secreta.

Assim como nos algoritmos de geração de chaves de outros primitivos criptográficos, um parâmetro de segurança é exigido para decidir a segurança em bits do algoritmo. Qualquer um pode então usar a parte pública para criptografar mensagens, e você pode usar a parte privada para descriptografá-las como ilustrado na **<IMAGEM>** (Figura 6.3). Similar à descriptografia autenticada, a descriptografia pode falhar se apresentada com um texto cifrado incoerente.

Observe que até agora não falamos sobre autenticação. Considere ambos os lados da conexão:

* Você está criptografando para uma chave pública que acredita ser de Alice.
* Alice não sabe com certeza quem enviou esta mensagem.

Por enquanto, vamos imaginar que obtivemos a chave pública de Alice de uma maneira realmente segura. No capítulo 7, que aborda assinaturas digitais, você aprenderá como os protocolos do mundo real resolvem essa questão de bootstrapping na prática. Você também aprenderá no capítulo 7 como pode comunicar a Alice quem você realmente é, de forma criptográfica (dica: você assinará suas mensagens).

Vamos seguir para a próxima seção, onde você aprenderá como a criptografia assimétrica é usada na prática (e também por que raramente é usada como está na prática).

**6.2 Criptografia assimétrica na prática e criptografia híbrida**

Você pode estar pensando que a criptografia assimétrica provavelmente já seria suficiente para começar a criptografar seus e-mails. Na realidade, a criptografia assimétrica é bastante limitada devido ao comprimento restrito das mensagens que pode criptografar. A velocidade de criptografia e descriptografia assimétricas também é lenta em comparação com a criptografia simétrica. Isso ocorre porque as construções assimétricas implementam operações matemáticas, enquanto os primitivos simétricos frequentemente apenas manipulam bits.

Nesta seção, você aprenderá sobre essas limitações, para que a criptografia assimétrica é realmente usada na prática e, por fim, como a criptografia supera esses impedimentos. Esta seção está dividida em duas partes para os dois principais casos de uso da criptografia assimétrica:

* **Trocas de chaves** — Você verá que é bastante natural realizar uma troca de chaves (ou acordo de chaves) com um primitivo de criptografia assimétrica.
* **Criptografia híbrida** — Você verá que os casos de uso da criptografia assimétrica são bastante limitados devido ao tamanho máximo do que pode ser criptografado. Para criptografar mensagens maiores, você aprenderá sobre um primitivo mais útil chamado criptografia híbrida.

**6.2.1 Trocas de chaves e encapsulamento de chaves**

Acontece que a criptografia assimétrica pode ser usada para realizar uma troca de chaves — o mesmo tipo das que vimos no capítulo 5! Para fazer isso, você pode começar gerando uma chave simétrica e criptografando-a com a chave pública de Alice — o que também chamamos de encapsular uma chave — como a **<IMAGEM>** (Figura 6.4) demonstra.

Você pode então enviar o texto cifrado para Alice, que será capaz de descriptografá-lo e obter a chave simétrica. Posteriormente, ambos terão um segredo compartilhado! A **<IMAGEM>** (Figura 6.5) ilustra o fluxo completo.

O uso da criptografia assimétrica para realizar uma troca de chaves é normalmente feito com um algoritmo chamado RSA (sigla dos inventores Rivest, Shamir e Adleman) e é usado em muitos protocolos da internet. Atualmente, o RSA frequentemente não é a forma preferida de realizar uma troca de chaves, sendo cada vez menos usado nos protocolos em favor do Elliptic Curve Diffie-Hellman (ECDH). Isso ocorre principalmente por razões históricas (muitas vulnerabilidades foram descobertas em implementações e padrões de RSA) e pela atratividade dos tamanhos menores de parâmetros oferecidos pelo ECDH.

**6.2.2 Criptografia híbrida**

Na prática, a criptografia assimétrica só consegue criptografar mensagens até um certo comprimento. Por exemplo, o tamanho das mensagens em texto claro que podem ser criptografadas por RSA é limitado pelos parâmetros de segurança usados durante a geração do par de chaves (e mais especificamente pelo tamanho do módulo). Hoje em dia, com os parâmetros de segurança utilizados (módulo de 4.096 bits), o limite é de aproximadamente 500 caracteres ASCII — bem pequeno. Portanto, a maioria das aplicações faz uso de criptografia híbrida, cuja limitação está ligada aos limites de criptografia do algoritmo de criptografia autenticada utilizado.

A criptografia híbrida possui, na prática, a mesma interface da criptografia assimétrica (veja a **<IMAGEM>** - Figura 6.6). As pessoas podem criptografar mensagens com uma chave pública, e quem possui a chave privada associada pode descriptografar as mensagens criptografadas. A verdadeira diferença está nos limites de tamanho das mensagens que você pode criptografar.

Por baixo dos panos, a criptografia híbrida é simplesmente a combinação de um primitivo criptográfico assimétrico com um primitivo criptográfico simétrico (daí o nome).

**AVISO**: Você também poderia usar um primitivo simples de criptografia simétrica em vez de um primitivo de criptografia autenticada, mas a criptografia simétrica não protege contra alguém adulterar suas mensagens criptografadas. É por isso que nunca usamos apenas criptografia simétrica na prática (como visto no capítulo 4).

Vamos aprender como a criptografia híbrida funciona! Se você quiser criptografar uma mensagem para Alice, primeiro gera uma chave simétrica e criptografa sua mensagem com ela, e então usa um algoritmo de criptografia autenticada como ilustrado na **<IMAGEM>** (Figura 6.7).

Depois de criptografar sua mensagem, Alice ainda não poderá descriptografá-la sem o conhecimento da chave simétrica. Como fornecer essa chave simétrica à Alice? Criptografando assimetricamente a chave simétrica com a chave pública de Alice, como mostrado na **<IMAGEM>** (Figura 6.8).

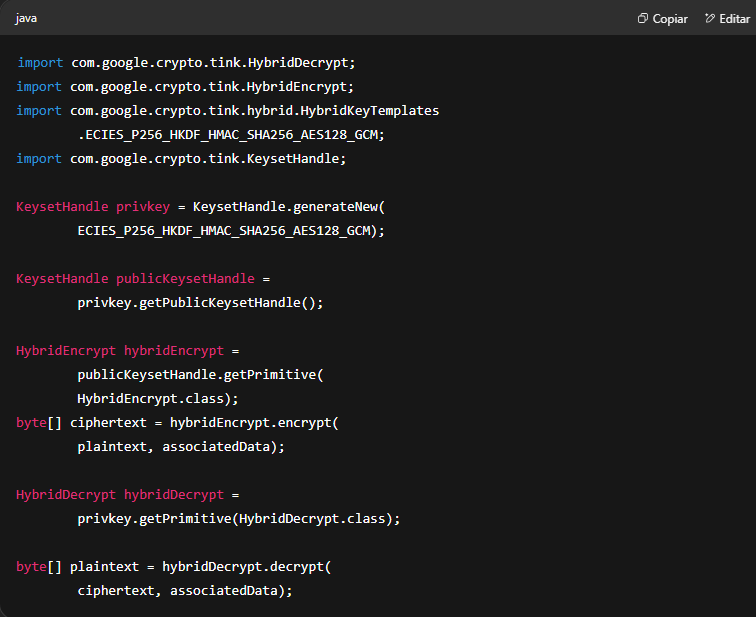
Finalmente, você pode enviar ambos os resultados para Alice. Estes incluem:

* A chave simétrica criptografada assimetricamente
* A mensagem criptografada simetricamente

Isso é informação suficiente para que Alice possa descriptografar a mensagem. Eu ilustro o fluxo completo na **<IMAGEM>** (Figura 6.9).

E é assim que podemos usar o melhor dos dois mundos: misturando criptografia assimétrica e criptografia simétrica para criptografar grandes quantidades de dados para uma chave pública. Muitas vezes chamamos a primeira parte assimétrica do algoritmo de mecanismo de encapsulamento de chaves (KEM) e a segunda parte simétrica de mecanismo de encapsulamento de dados (DEM).

Antes de seguirmos para a próxima seção e aprendermos sobre os diferentes algoritmos e padrões existentes para criptografia assimétrica e criptografia híbrida, vamos ver (na prática) como você pode usar uma biblioteca criptográfica para realizar criptografia híbrida. Para isso, escolhi a biblioteca de criptografia Tink. A Tink foi desenvolvida por uma equipe de criptógrafos do Google para apoiar grandes equipes dentro e fora da empresa. Devido à escala do projeto, decisões conscientes de design foram tomadas e funções seguras foram expostas para evitar que desenvolvedores utilizassem primitivos criptográficos de maneira incorreta. Além disso, o Tink está disponível em várias linguagens de programação (Java, C++, Obj-C e Golang).



**Observação para ajudá-lo a entender a string ECIES\_P256\_HKDF\_HMAC\_SHA256\_AES128\_GCM:**

* **ECIES** (Elliptic Curve Integrated Encryption Scheme) é o padrão de criptografia híbrida a ser utilizado.
* **P256** é a curva elíptica padronizada pelo NIST, que você aprendeu no capítulo 5.
* **HKDF** é uma função de derivação de chaves que você aprenderá no capítulo 8.
* **HMAC** é o código de autenticação de mensagens que você aprendeu no capítulo 3.
* **SHA-256** é a função hash que você aprendeu no capítulo 2.
* **AES-128-GCM** é o algoritmo de criptografia autenticada AES-GCM utilizando uma chave de 128 bits, que você aprendeu no capítulo 4.

Viu como tudo começa a se encaixar? Na próxima seção, você aprenderá sobre RSA e ECIES, os dois padrões amplamente adotados para criptografia assimétrica e criptografia híbrida.

**6.3 Criptografia assimétrica com RSA: o ruim e o menos ruim**

É hora de analisarmos os padrões que definem a criptografia assimétrica e a criptografia híbrida na prática. Historicamente, ambos esses primitivos não foram poupados pelos criptoanalistas, e muitas vulnerabilidades e fraquezas foram encontradas tanto nos padrões quanto nas implementações. É por isso que começarei esta seção com uma introdução ao algoritmo de criptografia de chave pública RSA e como **não** usá-lo. O restante do capítulo abordará os padrões atuais que você pode seguir para usar criptografia assimétrica e híbrida:

* **RSA-OAEP** — O principal padrão para realizar criptografia assimétrica com RSA
* **ECIES** — O principal padrão para realizar criptografia híbrida com Elliptic Curve Diffie-Hellman (ECDH)

**6.3.1 RSA "puro" (Textbook RSA)**

Nesta seção, você aprenderá sobre o algoritmo de criptografia de chave pública RSA e como ele foi padronizado ao longo dos anos. Isso é útil para entender outros esquemas seguros baseados em RSA.

Infelizmente, o RSA ficou com uma certa má fama desde que foi publicado pela primeira vez em 1977. Uma das teorias populares é que o RSA é fácil demais de entender e implementar, e, assim, muitas pessoas tentam fazê-lo por conta própria, o que leva a muitas implementações vulneráveis. É uma ideia interessante, mas não conta toda a história. Embora o conceito do RSA (frequentemente chamado de *textbook RSA*) seja inseguro se implementado de forma ingênua, até mesmo padrões foram encontrados quebrados! Mas não tão rápido — para entender essas questões, primeiro você precisará aprender como o RSA funciona.

Lembra do grupo multiplicativo de números módulo um primo p? (Falamos sobre isso no capítulo 5.) É o conjunto de inteiros estritamente positivos:

1, 2, 3, 4, ..., p – 1

Vamos imaginar que um desses números é nossa mensagem. Para p suficientemente grande, digamos com 4.096 bits, nossa mensagem pode conter aproximadamente no máximo 500 caracteres.

**NOTA**: Para computadores, uma mensagem é apenas uma série de bytes, que também pode ser interpretada como um número.

Já vimos que ao exponenciar um número (digamos, nossa mensagem), podemos gerar outros números que formam um subgrupo. Eu ilustro isso na **<IMAGEM>** (Figura 6.10).

Isso é útil para nós ao definirmos como criptografar com RSA. Para isso, publicamos um expoente público e (para *encryption*) e um número primo p. (Na realidade, p não pode ser primo, mas vamos ignorar isso por um momento.) Para criptografar uma mensagem m, calcula-se:

**ciphertext = mᵉ mod p**

Por exemplo, para criptografar a mensagem m = 2 com e = 2 e p = 5, calculamos:

**ciphertext = 2² mod 5 = 4**

E esta é a ideia por trás da criptografia com RSA!

**NOTA**: Normalmente, escolhe-se um número pequeno como expoente público e para que a criptografia seja rápida. Historicamente, os padrões e implementações parecem ter se fixado no número primo 65.537 como expoente público.

Muito bem! Agora você tem um método para as pessoas criptografarem mensagens para você. Mas como descriptografá-las? Lembre-se: se você continuar exponenciando um gerador, eventualmente voltará ao número original (veja a **<IMAGEM>** - Figura 6.11).

Isso deve lhe dar uma ideia de como implementar a descriptografia: descobrir de quanto você precisa elevar um texto cifrado para recuperar o gerador original (que é a mensagem). Suponhamos que você conheça tal número, que chamaremos de expoente privado d (d de *decryption*). Se você receber:

**ciphertext = messageᵉ mod p**

você poderá elevá-lo à potência d para recuperar a mensagem:

**ciphertextᵈ = (messageᵉ)ᵈ = messageᵉ×ᵈ = message mod p**

A matemática real por trás de encontrar este expoente privado d é um pouco complicada. Simplificando, você calcula o inverso de e módulo a ordem (número de elementos) do grupo:

**d = e⁻¹ mod ordem**

Temos um algoritmo eficiente para calcular inversos modulares (como o algoritmo Euclidiano Estendido), e portanto isso não é um problema. No entanto, temos outro problema! Para um primo p, a ordem é simplesmente p – 1, e assim, é fácil para qualquer um calcular o expoente privado. Isso porque todos os elementos nesta equação, exceto d, são públicos.

**Teorema de Euler**

Como obtivemos a equação anterior para calcular o expoente privado d? O teorema de Euler afirma que, para m coprimo com p (ou seja, que não possuem fatores em comum):

**m^ordem ≡ 1 mod p**

Para ordem, temos o número de elementos no grupo multiplicativo criado pelos inteiros módulo p. Isso implica, por sua vez, que para qualquer múltiplo inteiro:

**m^(1 + múltiplo × ordem) ≡ m × (m^ordem)^múltiplo ≡ m mod p**

Isto nos diz que a equação que estamos tentando resolver:

**m^(e×d) ≡ m mod p**

pode ser reduzida a:

**e × d ≡ 1 + múltiplo × ordem**

que pode ser reescrita como:

**e × d ≡ 1 mod ordem**

Isso, por definição, significa que d é o inverso de e módulo ordem.

**<IMAGEM>** (Figura 6.11)

**A sacada central do RSA**

Uma maneira de evitar que outros calculem o expoente privado a partir do expoente público é esconder a ordem do nosso grupo. Essa é a ideia brilhante por trás do RSA: se o nosso módulo não for mais um primo, mas sim o produto de dois primos N = p × q (onde p e q são grandes primos conhecidos apenas por você), então a ordem do nosso grupo multiplicativo não é fácil de calcular, desde que p e q permaneçam secretos!

Estamos bem agora! Recapitulando, assim funciona o RSA:

**Para geração de chaves:**

1. Gere dois grandes números primos p e q.
2. Escolha um expoente público e aleatório ou um fixo como e = 65537.
3. Sua chave pública é o expoente público e e o módulo público N = p × q.
4. Derive seu expoente privado d = e⁻¹ mod (p – 1)(q – 1).
5. Sua chave privada é o expoente privado d.

**Para criptografia:**

Calcule **messageᵉ mod N**.

**Para descriptografia do texto cifrado:**

Calcule **ciphertextᵈ mod N**.

**<IMAGEM>** (Figura 6.12)

Dizemos que o RSA depende do problema de fatoração. Sem o conhecimento de p e q, ninguém consegue calcular a ordem; portanto, ninguém além de você pode calcular o expoente privado a partir do expoente público. Isso é semelhante a como o Diffie-Hellman dependia do problema do logaritmo discreto (veja a **<IMAGEM>** - Figura 6.13).

Assim, o RSA puro (textbook RSA) opera módulo um número composto N = p × q, onde p e q são dois grandes primos que precisam permanecer secretos. Agora que você entende como o RSA funciona, vejamos o quão inseguro ele é na prática e o que os padrões fazem para torná-lo seguro.

**6.3.2 Por que não usar RSA PKCS#1 v1.5**

Você aprendeu sobre o *textbook RSA*, que é inseguro por padrão por vários motivos. Antes de aprender sobre a versão segura do RSA, vejamos o que você precisa evitar.

Há muitos motivos pelos quais você não pode usar o RSA puro diretamente. Um exemplo é que, se você criptografar mensagens pequenas (por exemplo m = 2), então um atacante malicioso pode simplesmente criptografar todos os números pequenos entre 0 e 100, por exemplo, e rapidamente verificar se algum de seus números criptografados coincide com o seu texto cifrado. Se coincidir, ele saberá o que você criptografou.

Os padrões resolvem esse problema tornando suas mensagens grandes demais para serem atacadas por força bruta dessa forma. Especificamente, eles maximizam o tamanho de uma mensagem (antes da criptografia) com um preenchimento não determinístico. Por exemplo, o padrão RSA PKCS#1 v1.5 define um preenchimento que adiciona uma quantidade de bytes aleatórios à mensagem. Eu ilustro isso na **<IMAGEM>** (Figura 6.14).

O PKCS#1 é na verdade o primeiro padrão baseado em RSA, publicado como parte de uma série de documentos Public Key Cryptography Standard (PKCS) escritos pela empresa RSA no início dos anos 90. Embora o PKCS#1 tenha corrigido alguns problemas conhecidos, em 1998 Bleichenbacher descobriu um ataque prático contra o PKCS#1 v1.5 que permitia a um atacante descriptografar mensagens criptografadas com o preenchimento especificado pelo padrão. Como exigia um milhão de mensagens, ficou conhecido como o *ataque da milhão de mensagens*. Posteriormente, foram encontradas mitigaçãoes, mas, curiosamente, ao longo dos anos o ataque foi redescoberto repetidamente, à medida que os pesquisadores constataram que as mitigaçãoes eram difíceis de implementar com segurança (ou mesmo impossíveis).

**Ataques de cifra escolhida adaptativa (CCA2)**

O ataque da milhão de mensagens de Bleichenbacher é um tipo de ataque chamado de *chosen ciphertext attack adaptativo* (CCA2) na criptografia teórica. CCA2 significa que, para realizar este ataque, o atacante pode submeter mensagens arbitrárias criptografadas com RSA (cifra escolhida), observar como isso influencia a descriptografia e continuar o ataque com base nas observações anteriores (a parte adaptativa). CCA2 é frequentemente usado para modelar atacantes em provas de segurança criptográfica.

O motivo pelo qual o ataque foi possível está no fato de que os textos cifrados de RSA são maleáveis: você pode adulterar um texto cifrado RSA sem invalidar sua descriptografia. Se eu observar o texto cifrado **c = mᵉ mod N**, então posso submeter o seguinte texto cifrado:

**3ᵉ × mᵉ = (3m)ᵉ mod N**

o qual será descriptografado como:

**((3m)ᵉ)ᵈ = (3m)ᵉ×ᵈ = 3m mod N**

Usei o número 3 como exemplo aqui, mas eu poderia multiplicar a mensagem original por qualquer número que quisesse. Na prática, uma mensagem deve ser bem-formada (devido ao preenchimento), e portanto, adulterar um texto cifrado deveria invalidar a descriptografia. No entanto, acontece que, às vezes, mesmo após a modificação maliciosa, o preenchimento é aceito após a descriptografia.

Bleichenbacher utilizou essa propriedade em seu ataque da milhão de mensagens contra o RSA PKCS#1 v1.5. Seu ataque funciona interceptando uma mensagem criptografada, modificando-a e enviando-a à pessoa responsável pela descriptografia. Observando se essa pessoa consegue descriptografá-la (isto é, se o preenchimento permaneceu válido), obtemos alguma informação sobre a mensagem.

Como as duas primeiras bytes são **0x0002**, sabemos que a descriptografia está abaixo de certo valor. Repetindo este processo iterativamente, podemos restringir esse intervalo até obter a mensagem original.

Mesmo sendo o ataque de Bleichenbacher bem conhecido, ainda hoje há muitos sistemas em uso que implementam RSA PKCS#1 v1.5 para criptografia. Como parte do meu trabalho como consultor de segurança, encontrei muitos aplicativos vulneráveis a este ataque — portanto, tome cuidado!

**6.3.3 Criptografia assimétrica com RSA-OAEP**

Em 1998, a versão 2.0 do mesmo padrão PKCS#1 foi lançada com um novo esquema de preenchimento para RSA chamado **Optimal Asymmetric Encryption Padding (OAEP)**. Ao contrário de seu predecessor PKCS#1 v1.5, o OAEP não é vulnerável ao ataque de Bleichenbacher e, portanto, é um padrão robusto para uso em criptografia RSA atualmente. Vamos ver como o OAEP funciona e impede os ataques discutidos anteriormente.

Primeiro, vale mencionar que, como a maioria dos algoritmos criptográficos, o OAEP vem com um algoritmo de geração de chaves. Este recebe um parâmetro de segurança, como ilustrado na **<IMAGEM>** (Figura 6.15).

Esse algoritmo recebe um parâmetro de segurança, que representa um número de bits. Assim como no Diffie-Hellman, as operações ocorrem no conjunto de números módulo um número grande. Quando falamos sobre a segurança de uma instância do RSA, geralmente nos referimos ao tamanho desse módulo grande. Isso é semelhante ao Diffie-Hellman, se você se lembra.

Atualmente, a maioria das diretrizes (veja <https://keylength.com>) estima um módulo entre 2.048 e 4.096 bits para fornecer segurança de 128 bits. Como essas estimativas variam bastante, a maioria das aplicações parece, por precaução, adotar parâmetros de 4.096 bits.

**NOTA**: Vimos que o grande módulo do RSA não é primo, mas o produto **N = p × q** de dois grandes números primos p e q. Para um módulo de 4.096 bits, o algoritmo de geração de chaves normalmente divide isso ao meio e gera tanto p quanto q com cerca de 2.048 bits cada.

Para criptografar, o algoritmo primeiro preenche a mensagem e a mistura com um número aleatório gerado por criptografia. O resultado é então criptografado com RSA. Para descriptografar o texto cifrado, o processo é invertido, conforme mostra a **<IMAGEM>** (Figura 6.16).

O OAEP utiliza essa mistura para garantir que, mesmo que alguns bits do que foi criptografado com RSA vazem, nenhuma informação sobre o texto original possa ser obtida. De fato, para reverter o preenchimento OAEP, você precisa obter (praticamente) todos os bytes do texto cifrado preenchido pelo OAEP! Além disso, o ataque de Bleichenbacher não deve mais funcionar porque o esquema torna impossível obter um texto plano bem-formado através da modificação de um texto cifrado.

**NOTA**: *Awareness plaintext* (conhecimento do texto plano) é a propriedade que torna difícil demais para um atacante criar um texto cifrado que se descriptografe com sucesso (claro, sem a ajuda da criptografia). Devido ao *plaintext-awareness* proporcionado pelo OAEP, o ataque de Bleichenbacher não funciona nesse esquema.

Dentro do OAEP, MGF significa *mask generation function* (função geradora de máscara). Na prática, uma MGF é uma função de saída extensível (XOF); você aprendeu sobre XOFs no capítulo 2. Como as MGFs foram inventadas antes das XOFs, são construídas usando uma função hash que processa repetidamente a entrada com um contador (veja a **<IMAGEM>** - Figura 6.17). E é assim que o OAEP funciona!

Além disso, o design do OAEP não é perfeito; melhores construções foram propostas e padronizadas ao longo dos anos. Um exemplo é o **RSA-KEM**, que possui provas de segurança mais robustas e é muito mais simples de implementar com segurança. Você pode observar como o design é muito mais elegante na **<IMAGEM>** (Figura 6.18).

Observe a função de derivação de chaves (**KDF**) utilizada aqui. É outro primitivo criptográfico que pode ser substituído por uma MGF ou uma XOF. Falarei mais sobre o que são as KDFs no **capítulo 8** sobre aleatoriedade e segredos.

Atualmente, a maioria dos protocolos e aplicações que usam RSA ainda implementa o inseguro PKCS#1 v1.5 ou o OAEP. Por outro lado, cada vez mais protocolos estão abandonando a criptografia RSA em favor do **Elliptic Curve Diffie-Hellman (ECDH)** tanto para trocas de chaves quanto para criptografia híbrida. Isso é compreensível, pois o ECDH fornece chaves públicas menores e, em geral, conta com padrões muito melhores e implementações muito mais seguras.

**O ataque de *padding oracle* de Manger**

Apenas três anos após o lançamento do padrão OAEP, James Manger descobriu um ataque de temporização, semelhante ao ataque da milhão de mensagens de Bleichenbacher (mas muito mais prático), contra o OAEP caso não seja implementado corretamente. Felizmente, é muito mais simples implementar o OAEP com segurança em comparação ao PKCS#1 v1.5, e vulnerabilidades em implementações deste esquema são muito mais raras.

**6.4 Criptografia híbrida com ECIES**

Embora existam muitos esquemas de criptografia híbrida, o padrão mais amplamente adotado é o **Elliptic Curve Integrated Encryption Scheme (ECIES)**. O esquema foi especificado para ser usado com ECDH e está incluído em muitos padrões, como ANSI X9.63, ISO/IEC 18033-2, IEEE 1363a e SECG SEC 1. Infelizmente, cada padrão parece implementar uma variante diferente, e diferentes bibliotecas criptográficas implementam a criptografia híbrida de forma diferente, em parte devido a isso.

Por esta razão, raramente vejo duas implementações semelhantes de criptografia híbrida no mundo real. É importante entender que, embora isso seja irritante, se todos os participantes do protocolo utilizarem a mesma implementação ou documentarem os detalhes do esquema de criptografia híbrida que implementaram, não haverá problemas.

O ECIES funciona de forma semelhante ao esquema de criptografia híbrida explicado na **seção 6.2**. A diferença é que implementamos a parte KEM com uma troca de chaves ECDH em vez de com um primitivo de criptografia assimétrica. Vamos explicar isso passo a passo.

Primeiro, se você quiser criptografar uma mensagem para Alice, você realiza uma troca de chaves (EC)DH com a chave pública de Alice e um par de chaves que você gera para a ocasião (isso é chamado de par de chaves efêmero). Você pode então usar o segredo compartilhado obtido com um algoritmo de criptografia simétrica autenticada como o AES-GCM para criptografar uma mensagem maior para ela. A **<IMAGEM>** (Figura 6.19) ilustra isso.

Depois disso, você pode enviar a chave pública efêmera e o texto cifrado para Alice. Alice pode usar sua chave pública efêmera para realizar uma troca de chaves com seu próprio par de chaves. Ela pode então usar o resultado para descriptografar o texto cifrado e recuperar a mensagem original. O resultado será ou a mensagem original ou um erro caso a chave pública ou a mensagem criptografada tenham sido adulteradas durante o trânsito. A **<IMAGEM>** (Figura 6.20) ilustra o fluxo completo.

E é basicamente assim que o ECIES funciona. Também existe uma variante do ECIES usando o Diffie-Hellman, chamada **IES**, que funciona de forma bastante semelhante, mas poucas pessoas parecem usá-la.

**Removendo viés na saída de uma troca de chaves**

Observe que simplifiquei a **Figura 6.20**. A maioria dos primitivos de criptografia autenticada espera uma chave simétrica uniformemente aleatória. Como a saída de uma troca de chaves geralmente não é uniformemente aleatória, precisamos passar o segredo compartilhado por uma KDF ou uma XOF (como visto no capítulo 2) previamente. Você aprenderá mais sobre isso no **capítulo 8**.

Não ser uniformemente aleatório aqui significa que, estatisticamente, alguns bits do resultado da troca de chaves podem ser 0 com mais frequência que 1, ou o contrário. Os primeiros bits podem sempre ser 0, por exemplo.

E é isso para os diferentes padrões que você pode usar. No próximo capítulo, você aprenderá sobre assinaturas, que serão o último — e talvez o mais importante — algoritmo de criptografia de chave pública da parte 1.

**Resumo**

* Raramente usamos criptografia assimétrica para criptografar mensagens diretamente. Isso se deve às limitações de tamanho relativamente pequenas dos dados que a criptografia assimétrica pode criptografar.
* A criptografia híbrida pode criptografar mensagens muito maiores, combinando criptografia assimétrica (ou uma troca de chaves) com um algoritmo de criptografia simétrica autenticada.
* O padrão RSA PKCS#1 versão 1.5 para criptografia assimétrica está quebrado na maioria dos contextos. Prefira o algoritmo RSA-OAEP padronizado no RSA PKCS#1 versão 2.2.
* O ECIES é o esquema de criptografia híbrida mais amplamente utilizado. É preferido em relação aos esquemas baseados em RSA devido ao tamanho de seus parâmetros e à sua dependência em padrões sólidos.
* Diferentes bibliotecas criptográficas podem implementar a criptografia híbrida de forma diferente. Isso não é um problema na prática se os aplicativos interoperáveis utilizarem as mesmas implementações.

**Exercício**

Você consegue perceber por que não pode usar diretamente a saída da troca de chaves?