**Aleatoriedade e Segredos**

Este é o último capítulo da primeira parte deste livro, e eu tenho uma última coisa para lhe dizer antes de seguirmos para a segunda parte e aprendermos sobre protocolos reais utilizados no mundo real. É algo que eu negligenciei grosseiramente até agora — a aleatoriedade.

Você deve ter notado que, em todo algoritmo criptográfico que aprendeu (com exceção das funções hash), você precisou utilizar aleatoriedade em algum momento: chaves secretas, nonces, IVs, números primos, desafios, e assim por diante. Ao longo desses diferentes conceitos, a aleatoriedade sempre vinha de uma caixa preta mágica. Isso não é atípico. Em artigos acadêmicos de criptografia, a aleatoriedade é frequentemente representada por uma seta com um cifrão em cima. Mas em algum momento, precisamos nos perguntar: “De onde vem realmente essa aleatoriedade?”

Este capítulo aborda:

* O que é aleatoriedade e por que é importante
* Como obter uma aleatoriedade forte e produzir segredos
* As armadilhas da aleatoriedade

**8.1 O que é aleatoriedade?**

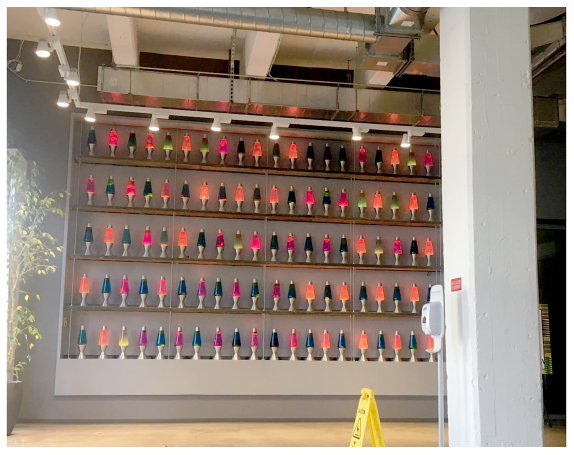
Todos entendem o conceito de aleatoriedade em certo grau. Seja jogando dados ou comprando bilhetes de loteria, todos já fomos expostos a ela. Meu primeiro encontro com a aleatoriedade foi ainda muito jovem, quando percebi que o botão RAND na minha calculadora produzia um número diferente toda vez que eu o pressionava. Isso me perturbou profundamente. Eu tinha pouco conhecimento sobre eletrônica, mas achava que conseguia entender algumas de suas limitações. Quando eu somava 4 e 5, com certeza alguns circuitos faziam o cálculo e me davam o resultado. Mas um botão aleatório? De onde vinham esses números aleatórios? Eu não conseguia entender.

Levou um tempo até que eu fizesse as perguntas certas e entendesse que as calculadoras na verdade trapaceavam! Elas tinham listas grandes de números aleatórios pré-programados e percorriam essas listas um a um. Essas listas exibiam uma boa aleatoriedade, significando que, ao observar os números aleatórios gerados, haveria tantos 1s quanto 9s, tantos 1s quanto 2s, e assim por diante. Essas listas simulavam uma distribuição uniforme: os números eram distribuídos em proporções iguais (uniformemente).

Quando números aleatórios são necessários para segurança e criptografia, então a aleatoriedade deve ser imprevisível. Claro, na época, ninguém usaria a "aleatoriedade" dessas calculadoras para nada relacionado à segurança. Em vez disso, aplicações criptográficas extraem aleatoriedade observando fenômenos físicos difíceis de prever.

Por exemplo, é difícil prever o resultado de uma jogada de dados, mesmo que lançar um dado seja um processo determinístico; se você soubesse todas as condições iniciais (como você está lançando o dado, o próprio dado, o atrito do ar, a aderência da mesa, e assim por diante), seria capaz de prever o resultado. No entanto, todos esses fatores impactam tanto o resultado final que uma pequena imprecisão no conhecimento das condições iniciais arruinaria nossas previsões. A extrema sensibilidade de um resultado às suas condições iniciais é conhecida como teoria do caos, e é a razão pela qual coisas como o clima são difíceis de prever com precisão após alguns dias.

A imagem a seguir é uma foto que tirei durante uma de minhas visitas à sede da Cloudflare em São Francisco. LavaRand é uma parede de luminárias de lava, que produzem formas de cera difíceis de prever. Uma câmera é posicionada em frente à parede para extrair e converter as imagens em bytes aleatórios.



**8.2 Aleatoriedade lenta? Use um gerador de números pseudoaleatórios (PRNG)**

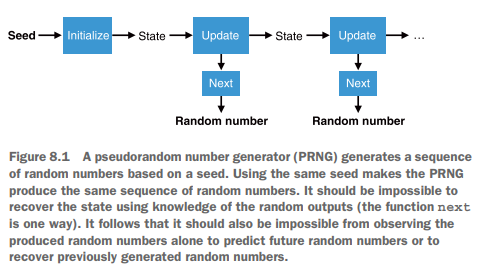
A aleatoriedade é usada em todos os lugares. Neste ponto, você já deve estar convencido de que isso é verdade pelo menos para a criptografia, mas surpreendentemente, a criptografia não é o único lugar que faz uso pesado de números aleatórios.

A aleatoriedade é usada em todos os lugares. Neste ponto, você já deve estar convencido de que isso é verdade pelo menos para a criptografia, mas surpreendentemente, a criptografia não é o único lugar que faz uso pesado de números aleatórios. Por exemplo, programas Unix simples como o ls também requerem aleatoriedade! Como um bug em um programa pode ser devastador se explorado, binários tentam se defender contra ataques de baixo nível usando uma série de truques; um deles é o **ASLR** (*address space layout randomization* — randomização do layout do espaço de endereçamento), que randomiza o layout de memória de um processo toda vez que ele é executado e, assim, exige números aleatórios. Outro exemplo é o protocolo de rede TCP, que utiliza números aleatórios toda vez que cria uma conexão, a fim de produzir uma sequência imprevisível de números e dificultar ataques que tentam sequestrar conexões. Embora tudo isso esteja além do escopo deste livro, é bom ter uma ideia de quanto a aleatoriedade acaba sendo usada por motivos de segurança no mundo real.

Eu já sugeri na seção anterior que, infelizmente, obter aleatoriedade imprevisível é um processo um tanto lento. Isso às vezes ocorre porque uma fonte de entropia demora para produzir ruído. Como resultado, os sistemas operacionais frequentemente otimizam sua produção de números aleatórios usando geradores de números pseudoaleatórios (**PRNGs**).

**NOTA**  
Para contrastar com geradores de números aleatórios que não são projetados para serem seguros (e que são úteis em diferentes tipos de aplicações, como videogames), os PRNGs às vezes são chamados de **CSPRNGs** (*cryptographically secure PRNGs* — PRNGs criptograficamente seguros). O NIST, querendo fazer diferente (como de costume), frequentemente chama seus PRNGs de **deterministic random bit generators** (*DRBGs* — geradores determinísticos de bits aleatórios).

Um PRNG precisa de um segredo inicial, geralmente chamado de **seed** (semente), que podemos obter misturando diferentes fontes de entropia e então produzir muitos números aleatórios rapidamente. Eu ilustro um PRNG na figura 8.1.

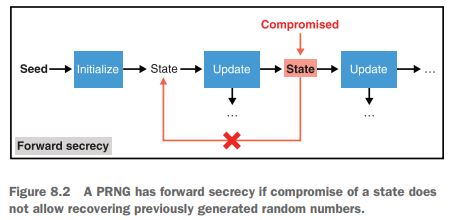
****

Os PRNGs criptograficamente seguros geralmente tendem a exibir as seguintes propriedades:

* **Determinístico** — Usar a mesma semente duas vezes produz a mesma sequência de números aleatórios. Isso é diferente da extração de aleatoriedade imprevisível de que falamos anteriormente: se você conhece uma semente usada por um PRNG, o PRNG deve ser completamente previsível. É por isso que a construção é chamada de *pseudoaleatória*, e isso é o que permite que um PRNG seja extremamente rápido.
* **Indistinguível de aleatório** — Na prática, você não deveria ser capaz de distinguir entre um PRNG produzindo um número aleatório de um conjunto de números possíveis e uma fadinha escolhendo imparcialmente um número aleatório do mesmo conjunto (assumindo que a fadinha conhece uma maneira mágica de escolher um número tal que qualquer número possível possa ser escolhido com igual probabilidade). Consequentemente, observar apenas os números aleatórios gerados não deveria permitir que ninguém recupere o estado interno do PRNG.

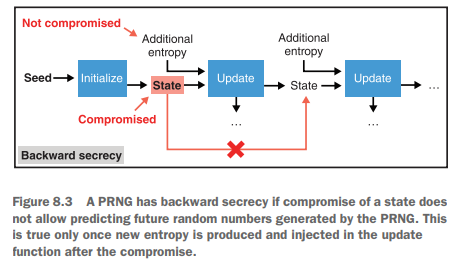
O último ponto é importante! Um PRNG simula a escolha de um número uniformemente aleatório, o que significa que cada número do conjunto tem chance igual de ser escolhido. Por exemplo, se seu PRNG produz números aleatórios de 8 bytes, o conjunto é composto de todas as possíveis cadeias de 8 bytes, e cada valor de 8 bytes deveria ter igual probabilidade de ser o próximo valor obtido do PRNG. Isso inclui valores que já podem ter sido produzidos anteriormente pelo PRNG.

Além disso, muitos PRNGs exibem propriedades de segurança adicionais. Um PRNG tem **sigilo direto** (*forward secrecy*) se um atacante aprendendo o estado (por exemplo, invadindo seu computador em algum momento) não consegue recuperar os números aleatórios previamente gerados. Eu ilustro isso na figura 8.2.

****

Obter o estado de um PRNG significa que você pode determinar todos os números pseudoaleatórios que ele gerará no futuro. Para evitar isso, alguns PRNGs têm mecanismos para se "curarem" periodicamente (caso haja um comprometimento). Essa cura pode ser alcançada reinjetando (ou re-semeando) nova entropia depois que o PRNG já foi semeado. Esta propriedade é chamada de **sigilo reverso** (*backward secrecy*). Eu ilustro isso na figura 8.3.

**NOTA**  
Os termos sigilo direto (*forward secrecy*) e sigilo reverso (*backward secrecy*) costumam causar confusão. Se você leu esta seção pensando: “o sigilo direto não deveria ser chamado de sigilo reverso e vice-versa?”, então você não está louco. Por essa razão, o sigilo reverso às vezes é chamado de **sigilo futuro** (*future secrecy*) ou até mesmo de **segurança pós-comprometimento** (*post-compromise security* — PCS).

****

Os PRNGs podem ser extremamente rápidos e são considerados métodos seguros para gerar grandes quantidades de valores aleatórios para fins criptográficos, se devidamente semeados. Usar um número previsível ou um número pequeno demais obviamente não é uma forma segura de semear um PRNG. Isso significa, na prática, que temos maneiras criptográficas seguras de expandir rapidamente um segredo de tamanho apropriado para bilhões de outras chaves secretas. Bem legal, certo? É por isso que a maioria (senão todas) das aplicações criptográficas não utilizam números aleatórios diretamente extraídos do ruído, mas sim os usam para semear um PRNG em um passo inicial e depois passam a gerar números aleatórios a partir do PRNG quando necessário.

Para ser seguro, um PRNG deve ser semeado com um segredo imprevisível. Mais precisamente, dizemos que o PRNG toma uma chave de *n* bytes amostrada uniformemente ao acaso. Isso significa que devemos escolher a chave aleatoriamente no conjunto de todas as cadeias de *n* bytes possíveis, onde cada cadeia de bytes tem a mesma chance de ser escolhida.

**8.3 Obtendo aleatoriedade na prática**

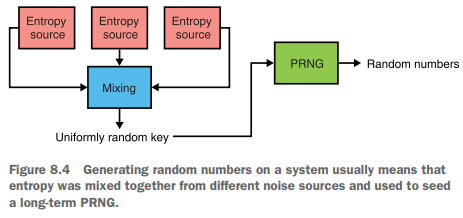
Você aprendeu sobre os três ingredientes que um sistema operacional precisa fornecer para disponibilizar números aleatórios criptograficamente seguros aos seus programas:

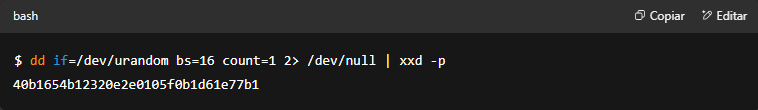
* **Fontes de ruído (noise sources)** — São maneiras pelas quais o sistema operacional obtém aleatoriedade bruta a partir de fenômenos físicos imprevisíveis, como a temperatura do dispositivo ou os movimentos do seu mouse.

**O backdoor do Dual-EC**  
Hoje, os PRNGs são construções baseadas principalmente em heurísticas. Isso porque construções baseadas em problemas matemáticos difíceis (como o logaritmo discreto) são lentas demais para serem práticas. Um exemplo notório é o Dual EC, inventado pela NSA, que se baseia em curvas elípticas. O PRNG Dual EC foi incluído em vários padrões, incluindo algumas publicações do NIST por volta de 2006, e não muito tempo depois, vários pesquisadores descobriram independentemente um possível backdoor no algoritmo. Isso foi posteriormente confirmado pelas revelações de Snowden em 2013, e um ano depois o algoritmo foi retirado de múltiplos padrões.

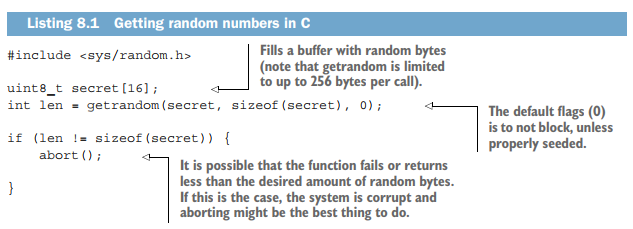
* **Limpeza e mistura (cleaning and mixing)** — Embora a aleatoriedade bruta possa ser de baixa qualidade (alguns bits podem ser enviesados), os sistemas operacionais limpam e misturam diversas fontes para produzir um bom número aleatório.
* **PRNGs** — Como os dois primeiros passos são lentos, um único valor uniformemente aleatório pode ser usado para semear um PRNG, que rapidamente produzirá números aleatórios.

Nesta seção, explicarei como os sistemas agrupam esses três conceitos para fornecer interfaces simplificadas aos desenvolvedores. Essas funções expostas pelo sistema operacional geralmente permitem que você gere um número aleatório com uma chamada de sistema. Por trás dessas chamadas de sistema há, de fato, um sistema agrupando fontes de ruído, um algoritmo de mistura e um PRNG (resumido na figura 8.4).

****Dependendo do sistema operacional e do hardware disponível, esses três conceitos podem ser implementados de maneiras diferentes. Em 2021, o Linux usa um PRNG baseado no cifra de fluxo ChaCha20, enquanto o macOS usa um PRNG baseado na função hash SHA-1. Além disso, a interface do gerador de números aleatórios exposta aos desenvolvedores varia conforme o sistema operacional. No Windows, a chamada de sistema BCryptGenRandom pode ser usada para produzir números aleatórios, enquanto em outras plataformas, um arquivo especial (geralmente chamado /dev/urandom) é exposto e pode ser lido para obter aleatoriedade. Por exemplo, no Linux ou macOS, pode-se ler 16 bytes no terminal usando o comando dd:



Um problema com o /dev/urandom é que ele pode não fornecer entropia suficiente (seus números não serão aleatórios o bastante) se usado muito cedo após a inicialização do dispositivo. Sistemas como o Linux e o FreeBSD oferecem uma solução chamada getrandom, que é uma chamada de sistema que fornece praticamente a mesma funcionalidade que ler do /dev/urandom. Em casos raros, onde não há entropia suficiente disponível para inicializar seu PRNG, o getrandom bloqueará a continuação do programa e aguardará até ser devidamente semeado. Por essa razão, recomendo que você use o getrandom se ele estiver disponível no seu sistema. O exemplo a seguir mostra como usar com segurança o getrandom em C:





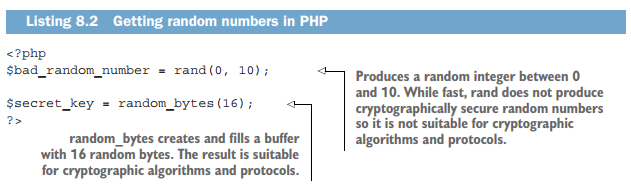
**Listagem 8.1 Obter números aleatórios em C**

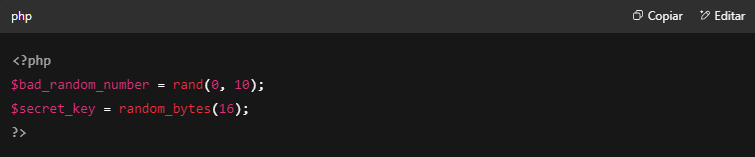
* Preenche um buffer com bytes aleatórios (note que o getrandom é limitado a até 256 bytes por chamada).
* A flag padrão (0) indica não bloquear, a menos que esteja devidamente semeado.
* É possível que a função falhe ou retorne menos bytes aleatórios do que o desejado. Se isso ocorrer, o sistema está corrompido e abortar pode ser a melhor opção.

Com esse exemplo em mente, também é bom destacar que muitas linguagens de programação possuem bibliotecas padrão e bibliotecas criptográficas que oferecem melhores abstrações. É fácil esquecer, por exemplo, que o getrandom retorna no máximo 256 bytes por chamada. Por essa razão, você deve sempre tentar gerar números aleatórios por meio da biblioteca padrão da linguagem de programação que está usando.

**AVISO**  
Note que muitas linguagens de programação expõem funções e bibliotecas que produzem números aleatórios previsíveis. Estes não são adequados para uso criptográfico! Certifique-se de usar bibliotecas de aleatoriedade que geram números aleatórios criptograficamente fortes. Normalmente o nome da biblioteca ajuda (por exemplo, você provavelmente consegue adivinhar qual usar entre os pacotes math/rand e crypto/rand em Golang), mas nada substitui ler a documentação!

A listagem 8.2 mostra como gerar alguns bytes aleatórios usando PHP 7. Qualquer algoritmo criptográfico pode usar esses bytes aleatórios, por exemplo, como uma chave secreta para criptografar com um algoritmo de criptografia autenticada. Cada linguagem de programação faz isso de forma diferente, portanto, consulte a documentação da sua linguagem para encontrar a forma padrão de obter números aleatórios para fins criptográficos.





**Listagem 8.2 Obter números aleatórios em PHP**

* rand(0, 10) produz um inteiro aleatório entre 0 e 10. Embora rápido, rand não produz números aleatórios criptograficamente seguros, portanto não é adequado para algoritmos e protocolos criptográficos.
* random\_bytes cria e preenche um buffer com 16 bytes aleatórios. O resultado é adequado para algoritmos e protocolos criptográficos.

Agora que você aprendeu como obter aleatoriedade criptograficamente segura em seus programas, vamos pensar nas considerações de segurança que você precisa ter em mente ao gerar aleatoriedade.

**8.4 Geração de aleatoriedade e considerações de segurança**

É bom lembrar, neste ponto, que qualquer protocolo útil baseado em criptografia requer boa aleatoriedade e que um PRNG quebrado pode tornar todo o protocolo ou algoritmo criptográfico inseguro. Deveria estar claro para você que um MAC é apenas tão seguro quanto a chave usada com ele, ou que o menor traço de previsibilidade normalmente destrói esquemas de assinatura como o ECDSA, e assim por diante.

Até agora, este capítulo pode dar a impressão de que gerar aleatoriedade deveria ser uma parte simples da criptografia aplicada, mas, na prática, não é. A aleatoriedade tem sido, na verdade, a fonte de muitos e muitos bugs na criptografia do mundo real devido a uma infinidade de problemas: usar um PRNG não criptográfico, semear mal um PRNG (por exemplo, usando o horário atual, que é previsível), e assim por diante.

Um exemplo inclui programas que usam PRNGs de *userland* em vez de PRNGs do kernel, que estão por trás das chamadas de sistema. Os PRNGs de *userland* geralmente adicionam atrito desnecessário e, se mal utilizados, podem, no pior dos casos, quebrar todo o sistema. Esse foi notavelmente o caso com o PRNG oferecido pela biblioteca OpenSSL que foi incorporado em alguns sistemas operacionais em 2006, afetando inadvertidamente todas as chaves SSL e SSH geradas usando o PRNG vulnerável.

*Remover este código tem o efeito colateral de prejudicar o processo de semeadura para o PRNG do OpenSSL. Em vez de misturar dados aleatórios na semente inicial, o único valor aleatório que era usado era o ID atual do processo. Na plataforma Linux, o ID máximo de processo por padrão é 32.768, resultando em um número muito pequeno de valores de semente sendo usados para todas as operações do PRNG.*  
— H. D. Moore ("Debian OpenSSL Predictable PRNG Toys," 2008)

Por esta e outras razões que mencionarei mais tarde neste capítulo, é sensato evitar PRNGs de *userland* e ficar com a aleatoriedade fornecida pelo sistema operacional quando disponível. Na maioria das situações, utilizar a biblioteca padrão da linguagem de programação ou uma boa biblioteca de criptografia já deve ser suficiente.

*Não podemos continuar adicionando ‘boas práticas’ após ‘boas práticas’ ao que os desenvolvedores precisam manter na cabeça ao escrever código cotidiano.*  
— Martin Boßlet ("OpenSSL PRNG Is Not (Really) Fork-safe," 2013)

Infelizmente, nenhum volume de conselhos realmente pode prepará-lo para as muitas armadilhas de obter boa aleatoriedade. Como a aleatoriedade está no centro de todo algoritmo criptográfico, cometer pequenos erros pode levar a consequências devastadoras. É bom ter em mente os seguintes casos extremos caso você se depare com eles:

* **Processos que fazem *fork*** — Ao usar um PRNG de *userland* (algumas aplicações com requisitos de desempenho extremamente altos podem não ter outra escolha), é importante ter em mente que um programa que faz *fork* criará um novo processo filho que terá o mesmo estado do PRNG que seu pai. Consequentemente, ambos os PRNGs passarão a produzir a mesma sequência de números aleatórios dali em diante. Por essa razão, se você realmente quiser usar um PRNG de *userland*, é preciso ter cuidado para que os *forks* usem sementes diferentes para seus PRNGs.
* **Máquinas virtuais (VMs)** — A clonagem do estado do PRNG também pode se tornar um problema ao usar o PRNG do sistema operacional. Pense em VMs. Se todo o estado de uma VM for salvo e depois inicializado várias vezes a partir desse ponto, cada instância pode produzir exatamente a mesma sequência de números aleatórios. Isso às vezes é corrigido por hipervisores e sistemas operacionais, mas é bom verificar o que o hipervisor que você está usando faz antes de executar aplicações que requisitam números aleatórios em VMs.
* **Entropia no início da inicialização (early boot entropy)** — Enquanto os sistemas operacionais geralmente não têm problemas em reunir entropia em dispositivos operados por usuários devido ao ruído gerado pelas interações do usuário com o dispositivo, dispositivos embarcados e sistemas *headless* têm mais dificuldades para produzir boa entropia durante a inicialização. A história já mostrou que alguns dispositivos tendem a inicializar de maneira semelhante e acabam acumulando o mesmo ruído inicial do sistema, levando à utilização da mesma semente em seus PRNGs internos e à geração da mesma série de números aleatórios.

*Existe uma janela de vulnerabilidade — um buraco de entropia na inicialização — durante o qual o urandom do Linux pode ser totalmente previsível, pelo menos em sistemas de núcleo único. [...] Quando desativamos fontes de entropia que poderiam não estar disponíveis em dispositivos embarcados ou headless, o RNG do Linux produziu o mesmo fluxo previsível em cada inicialização.*  
— Heninger et al. ("Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," 2012)

Nesses casos raros, onde você realmente, realmente precisa obter números aleatórios logo no início da inicialização, é possível ajudar o sistema fornecendo alguma entropia inicial gerada a partir do getrandom ou do /dev/urandom de outra máquina já bem semeada. Diferentes sistemas operacionais podem oferecer esse recurso, e você deve consultar seus manuais (como sempre) se se encontrar nesta situação.

Se disponível, um TRNG fornece uma solução fácil para o problema. Por exemplo, CPUs modernas da Intel incorporam um chip especial de hardware que extrai aleatoriedade de ruído térmico. Essa aleatoriedade está disponível através de uma instrução chamada **RDRAND**.

**A controvérsia do RDRAND**  
Interessantemente, o RDRAND da Intel tem sido bastante controverso devido ao medo de *backdoors*. A maioria dos sistemas operacionais que integraram o RDRAND como fonte de entropia o mistura com outras fontes de entropia de forma contributiva. **Contributivo** aqui significa que uma fonte de entropia não pode forçar o resultado da geração de aleatoriedade.

**Exercício**  
Imagine, por um minuto, que a mistura de diferentes fontes de entropia fosse feita simplesmente aplicando um XOR entre elas. Você consegue ver como isso poderia falhar em ser contributivo?

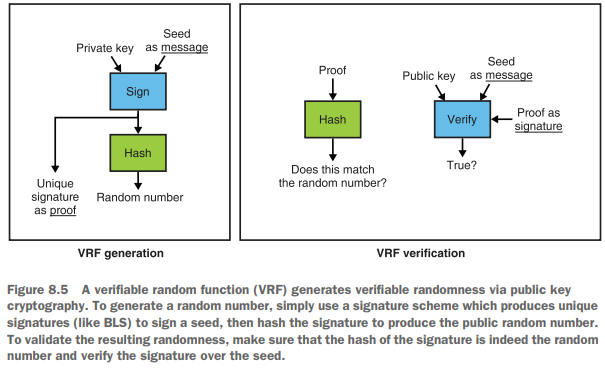
**8.5 Aleatoriedade pública**

Até agora, falei principalmente sobre aleatoriedade privada, o tipo que você pode precisar para suas chaves privadas. Às vezes, privacidade não é necessária e é preciso aleatoriedade pública. Nesta seção, apresento brevemente algumas maneiras de obter essa aleatoriedade pública. Distingo dois cenários:

* **Um para muitos (one-to-many)** — Você quer produzir aleatoriedade para outros.
* **Muitos para muitos (many-to-many)** — Um conjunto de participantes quer produzir aleatoriedade juntos.

Primeiro, imagine que você deseja gerar um fluxo de aleatoriedade de forma que muitos participantes possam verificá-lo. Em outras palavras, o fluxo deve ser imprevisível, mas impossível de ser alterado do seu ponto de vista. Agora, imagine que você tenha um esquema de assinatura que fornece assinaturas únicas com base em um par de chaves e uma mensagem. Com tal esquema de assinatura, existe uma construção chamada **função aleatória verificável (VRF — verifiable random function)** para obter números aleatórios de forma verificável (a figura 8.5 ilustra este conceito). O funcionamento é o seguinte:

1. Você gera um par de chaves e publica a chave de verificação. Também publica uma semente pública.
2. Para gerar números aleatórios, você assina a semente pública e aplica um hash na assinatura. O digest gerado é seu número aleatório, e a assinatura também é publicada como prova.
3. Para verificar o número aleatório, qualquer pessoa pode aplicar o hash na assinatura e verificar se corresponde ao número aleatório e verificar se a assinatura está correta com a semente pública e a chave de verificação.

****

Esta construção pode ser estendida para produzir muitos números aleatórios usando a semente pública como um contador. Como a assinatura é única e a semente pública é fixa, não há como o assinante gerar um número aleatório diferente.

Para resolver isso, o **Internet Draft** (um documento que pretende se tornar um RFC)  
<https://tools.ietf.org/html/draft-irtf-cfrg-vrf-08>  
especifica como implementar uma VRF usando ECDSA.

Em alguns cenários (por exemplo, um jogo de loteria), vários participantes podem querer decidir aleatoriamente quem será o vencedor. Chamamos isso de **faróis de aleatoriedade descentralizados (decentralized randomness beacons)**, já que seu papel é produzir a mesma aleatoriedade verificável mesmo se alguns participantes decidirem não participar do protocolo.

Uma solução comum é usar as VRFs discutidas anteriormente, mas não com uma única chave, e sim com uma **chave distribuída com limiar (threshold distributed key)**, uma chave dividida entre vários participantes e que produz uma assinatura válida e única para uma dada mensagem somente depois que um número mínimo (limiar) de participantes assina a mensagem.

Isso pode soar um pouco confuso, já que esta é a primeira vez que falo sobre chaves distribuídas. Saiba que você aprenderá mais sobre isso ainda neste capítulo.

Um farol de aleatoriedade descentralizado popular é chamado **drand** e é operado em conjunto por várias organizações e universidades. Ele está disponível em:  
<https://leagueofentropy.com>

*O principal desafio na geração de boa aleatoriedade é que nenhuma das partes envolvidas no processo de geração de aleatoriedade deve ser capaz de prever ou influenciar o resultado final. Uma rede drand não é controlada por nenhum de seus membros. Não há ponto único de falha, e nenhum dos operadores de servidores drand pode influenciar a aleatoriedade gerada pela rede.*  
— <https://drand.love> ("How drand works", 2021)

Agora que falei extensivamente sobre aleatoriedade e como os programas a obtêm hoje em dia, vamos mover a discussão para o papel dos segredos na criptografia e como podemos gerenciá-los.

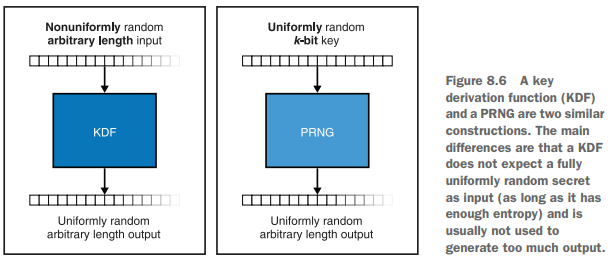
**8.6 Derivação de chaves com HKDF**

Os PRNGs não são as únicas construções que podem ser usadas para derivar mais segredos a partir de um segredo (em outras palavras, para esticar uma chave). Derivar vários segredos a partir de um segredo é, na verdade, um padrão tão frequente na criptografia que esse conceito tem um nome próprio: **derivação de chaves (key derivation)**. Vamos ver do que se trata.

**Exercício**  
Esquemas de assinatura como o BLS (mencionado na figura 8.5 e no capítulo 7) produzem assinaturas únicas, mas isso não é verdade para ECDSA e EdDSA. Você consegue ver o porquê?

Uma **função de derivação de chaves (KDF — key derivation function)** é semelhante a um PRNG em muitos aspectos, exceto por algumas sutilezas, conforme indicado na lista a seguir. As diferenças estão resumidas na figura 8.6.

* Uma KDF não necessariamente espera um segredo uniformemente aleatório (contanto que ele tenha entropia suficiente). Isso torna a KDF útil para derivar segredos a partir da saída de uma troca de chaves, que produz alta entropia, mas resultados enviesados (veja o capítulo 5). Os segredos resultantes são, por sua vez, uniformemente aleatórios, podendo assim ser usados em construções que exigem chaves uniformemente aleatórias.
* Uma KDF é geralmente usada em protocolos que exigem que os participantes rederivem as mesmas chaves várias vezes. Nesse sentido, uma KDF é esperada ser determinística, enquanto PRNGs às vezes oferecem sigilo reverso ao se re-semearem frequentemente com mais entropia.
* Uma KDF geralmente não é projetada para produzir muitos números aleatórios. Em vez disso, normalmente é usada para derivar um número limitado de chaves.



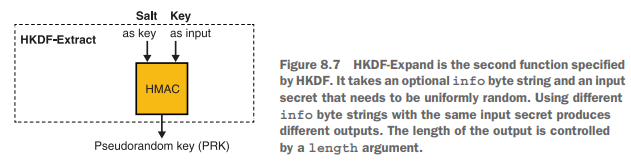
A KDF mais popular é a **HKDF (HMAC-based key derivation function)**. Você aprendeu sobre o HMAC (um MAC baseado em funções hash) no capítulo 3. O HKDF é uma KDF leve construída sobre o HMAC e definida na RFC 5869. Por essa razão, pode-se usar o HKDF com diferentes funções hash, embora ele seja mais comumente usado com SHA-2. O HKDF é especificado como duas funções distintas:

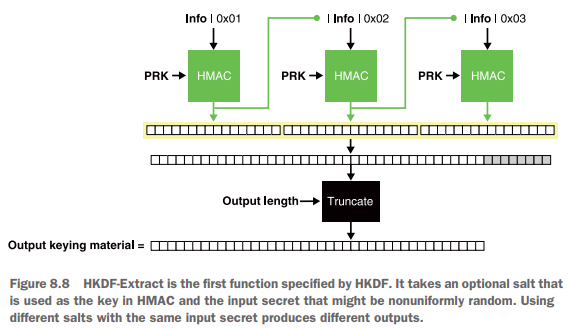
* **HKDF-Extract** — Remove vieses de uma entrada secreta, produzindo um segredo uniformemente aleatório.
* **HKDF-Expand** — Produz uma saída uniformemente aleatória de comprimento arbitrário. Como os PRNGs, espera um segredo uniformemente aleatório como entrada e, assim, geralmente é executado depois do HKDF-Extract.

Vamos primeiro analisar o HKDF-Extract, que eu ilustro na figura 8.7.  
Tecnicamente, uma função hash já é suficiente para uniformizar a aleatoriedade de uma cadeia de bytes de entrada (lembre-se de que a saída de uma função hash deve ser indistinguível de aleatória), mas o HKDF vai além e aceita uma entrada adicional: um **salt** (sal). Assim como no hash de senhas, um salt diferencia os diferentes usos do HKDF-Extract dentro do mesmo protocolo. Embora este salt seja opcional e configurado como uma cadeia de bytes zerada se não for usado, recomenda-se que você o utilize. Além disso, o HKDF não espera que o salt seja um segredo; ele pode ser conhecido por todos, incluindo adversários. Em vez de uma função hash, o HKDF-Extract usa um MAC (especificamente o HMAC), que coincidentemente possui uma interface que aceita dois argumentos.

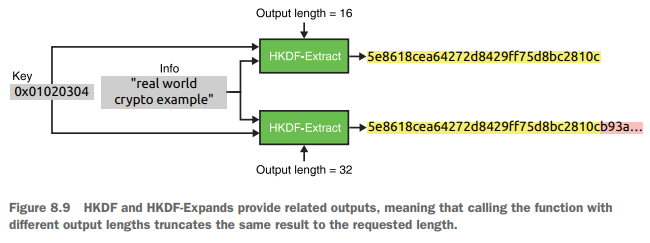
Agora vejamos o HKDF-Expand, ilustrado na figura 8.8. Se sua entrada secreta já for uniformemente aleatória, você pode pular o HKDF-Extract e usar diretamente o HKDF-Expand.

Assim como o HKDF-Extract, o HKDF-Expand também aceita um argumento adicional e opcional de personalização chamado **info**. Enquanto o salt serve para fornecer separação de domínio entre chamadas dentro do mesmo protocolo para o HKDF (ou HKDF-Extract), o **info** serve para diferenciar sua versão do HKDF (ou HKDF-Expand) de outros protocolos.  
Você também pode especificar quanto de saída deseja, mas tenha em mente que o HKDF não é um PRNG e não é projetado para derivar uma grande quantidade de segredos. O HKDF é limitado pelo tamanho da função hash que você utiliza; mais precisamente, se você usar o SHA-512 (que produz saídas de 512 bits) com o HKDF, estará limitado a 512 × 255 bits = 16.320 bytes de saída para uma dada chave e uma cadeia de bytes **info**.

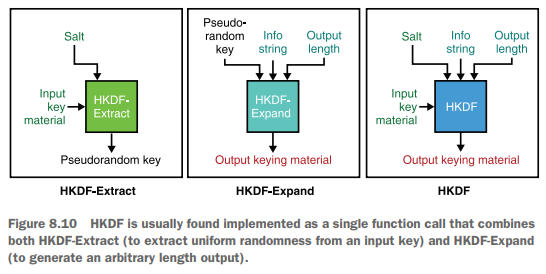
****

****

Chamar o HKDF ou o HKDF-Expand várias vezes com os mesmos argumentos, exceto pelo comprimento de saída, produz a mesma saída truncada para o comprimento solicitado (veja a figura 8.9). Essa propriedade é chamada de **saídas relacionadas (related outputs)** e pode, em cenários raros, surpreender os projetistas de protocolos. É bom ter isso em mente.

** (Figura 8.9: Exemplo de outputs relacionados com truncamento)**

A maioria das bibliotecas criptográficas combina o HKDF-Extract e o HKDF-Expand em uma única chamada, como ilustrado na figura 8.10. Como sempre, certifique-se de ler o manual (neste caso, a RFC 5869) antes de usar o HKDF.

****

**(Figura 8.10: HKDF como chamada única, combinando Extract e Expand)**

O HKDF não é a única forma de derivar múltiplos segredos a partir de um segredo. Uma abordagem mais ingênua é usar funções hash. Como as funções hash não esperam uma entrada uniformemente aleatória e produzem saídas uniformemente aleatórias, elas servem para essa tarefa. Entretanto, as funções hash não são perfeitas, pois sua interface não leva em conta separação de domínio (não possuem argumento de personalização) e seu comprimento de saída é fixo. A boa prática é evitar funções hash quando se pode usar uma KDF.

Ainda assim, alguns algoritmos bem aceitos utilizam funções hash para essa finalidade. Por exemplo, você aprendeu no capítulo 7 que o esquema de assinatura Ed25519 hasheia uma chave de 256 bits com SHA-512 para produzir duas chaves de 256 bits.

As **funções de saída estendida (XOFs — extended output functions)** que vimos no capítulo 2 (SHAKE e cSHAKE) também podem ser usadas como KDFs! Lembre-se de que um XOF:

* Não espera uma entrada uniformemente aleatória.
* Pode produzir uma saída praticamente infinita de dados uniformemente aleatórios.

Além disso, o **KMAC** (um MAC tratado no capítulo 3) não possui o problema de saídas relacionadas que mencionei anteriormente. De fato, o argumento de comprimento do KMAC randomiza a saída do algoritmo, atuando efetivamente como um argumento adicional de personalização.

Por fim, existe um caso especial para entradas de baixa entropia. Pense, por exemplo, em senhas, que podem ser relativamente fáceis de adivinhar comparadas a uma chave de 128 bits. As funções de derivação de chaves baseadas em senha usadas para hash de senhas (vistas no capítulo 2) também podem ser usadas para derivar chaves.

**8.7 Gerenciamento de chaves e segredos**

Muito bem, tudo certo, sabemos como gerar números aleatórios criptográficos e sabemos como derivar segredos em diferentes tipos de situações. Mas ainda não saímos do mato.

Agora que estamos usando todos esses algoritmos criptográficos, acabamos tendo que manter muitas chaves secretas. Como armazenamos essas chaves? E como evitamos que esses segredos extremamente sensíveis sejam comprometidos? E o que fazemos se um segredo for comprometido? Esse problema é comumente conhecido como **gerenciamento de chaves (key management)**.

*Criptografia é uma ferramenta para transformar toda uma gama de problemas em problemas de gerenciamento de chaves.*  
— Lea Kissner (2019, <http://mng.bz/eMrJ>)

Embora muitos sistemas optem por manter as chaves próximas à aplicação que faz uso delas, isso não significa necessariamente que as aplicações não tenham recursos quando coisas ruins acontecem.  
Para se preparar contra uma eventual violação ou um bug que vaze uma chave, a maioria das aplicações sérias emprega duas técnicas de defesa em profundidade:

* **Rotação de chaves (key rotation)** — Associando uma data de expiração a uma chave (geralmente uma chave pública) e substituindo sua chave por uma nova periodicamente, você pode “curar” uma eventual violação. Quanto menor a data de expiração e maior a frequência de rotação, mais rápido você poderá substituir uma chave que possa estar conhecida por um invasor.
* **Revogação de chaves (key revocation)** — A rotação de chaves nem sempre é suficiente, e você pode querer cancelar uma chave assim que souber que ela foi comprometida. Por essa razão, alguns sistemas permitem que você consulte se uma chave foi revogada antes de utilizá-la. (Você aprenderá mais sobre isso no próximo capítulo sobre transporte seguro.)

A automação é muitas vezes indispensável para o uso bem-sucedido dessas técnicas, pois um sistema bem azeitado é muito mais apto a funcionar corretamente em tempos de crise. Além disso, você também pode associar uma função específica a uma chave para limitar as consequências de um comprometimento. Por exemplo, você poderia diferenciar duas chaves públicas em uma aplicação fabricada: a **chave pública 1**, usada apenas para assinar transações, e a **chave pública 2**, usada apenas para fazer trocas de chaves. Isso permite que o comprometimento da chave privada associada à chave pública 2 não afete a assinatura de transações.

Se alguém não quiser deixar chaves armazenadas em mídias de armazenamento do dispositivo, existem soluções de hardware que visam impedir a extração de chaves. Você aprenderá mais sobre isso no capítulo 13, sobre criptografia de hardware.

Por fim, existem diversas maneiras para as aplicações delegarem o gerenciamento de chaves. Isso é comum em sistemas operacionais móveis, que fornecem **key stores** ou **key chains**, os quais armazenam as chaves para você e até mesmo realizam operações criptográficas!

Aplicações que vivem na nuvem às vezes têm acesso a serviços de gerenciamento de chaves na nuvem (**cloud key management services**). Esses serviços permitem que uma aplicação delegue a criação de chaves secretas e operações criptográficas, evitando assim se preocupar com as muitas formas de atacar essas chaves.  
Ainda assim, como nas soluções de hardware, se uma aplicação for comprometida, ela ainda será capaz de fazer qualquer tipo de requisição ao serviço delegado.

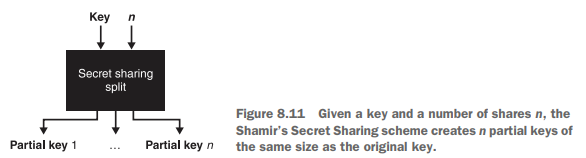
**NOTA**  
Não existem balas de prata, e você ainda deve considerar o que pode fazer para detectar e responder a um comprometimento.

O gerenciamento de chaves é um problema complexo e extenso que está além do escopo deste livro, então não me prolongarei muito neste tópico. Na próxima seção, abordarei técnicas criptográficas que tentam contornar o problema de gerenciamento de chaves.

**8.8 Descentralizar a confiança com criptografia de limiar**

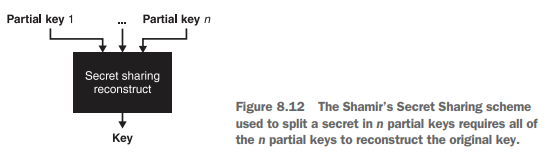
O gerenciamento de chaves é um vasto campo de estudo que pode ser bastante incômodo de investir, já que os usuários nem sempre têm os recursos para implementar as melhores práticas, nem as ferramentas disponíveis no mercado. Felizmente, a criptografia tem algo a oferecer para aqueles que querem aliviar o fardo do gerenciamento de chaves. A primeira técnica de que falarei é o **compartilhamento de segredo (secret sharing)** (ou **divisão de segredo — secret splitting**). O compartilhamento de segredo permite que você divida um segredo em várias partes que podem ser compartilhadas entre um conjunto de participantes. Aqui, um segredo pode ser qualquer coisa que você quiser: uma chave simétrica, uma chave privada de assinatura, e assim por diante.

Normalmente, uma pessoa chamada **dealer** (distribuidor) gera o segredo, depois o divide e compartilha as diferentes partes entre todos os participantes antes de apagar o segredo original. O esquema de compartilhamento de segredo mais famoso foi inventado por Adi Shamir (um dos co-inventores do RSA) e é chamado de **Shamir’s Secret Sharing (SSS)**. Eu ilustro esse processo na figura 8.11.

****

**(Figura 8.11: esquema de divisão em n partes)**

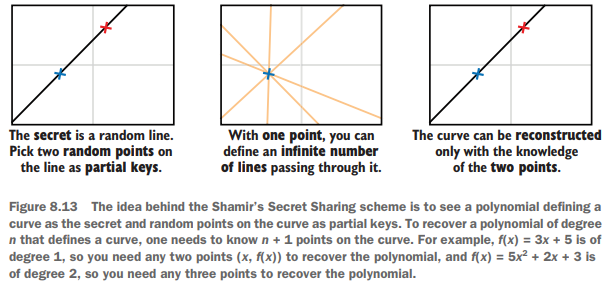
Quando chega o momento e o segredo é necessário para realizar alguma operação criptográfica (como criptografar, assinar, etc.), todos os detentores de partes precisam devolver suas partes privadas ao dealer, que é responsável por reconstruir o segredo original. Tal esquema impede que atacantes possam mirar um único usuário, já que cada parte isoladamente é inútil, e força os atacantes a comprometer todos os participantes antes de poder explorar uma chave! Eu ilustro isso na figura 8.12.

****

**(Figura 8.12: necessidade de reunir as n partes para reconstruir o segredo)**

Os fundamentos matemáticos por trás do algoritmo do esquema não são tão difíceis de entender! Então, permitam-me dedicar alguns parágrafos para dar uma ideia simplificada do esquema.

Imagine uma linha reta aleatória em um espaço bidimensional, e digamos que sua equação — **y = ax + b** — seja o segredo. Ao ter dois participantes segurando dois pontos aleatórios da linha, eles podem colaborar para recuperar a equação da linha. O esquema generaliza para polinômios de qualquer grau e, portanto, pode ser usado para dividir um segredo em um número arbitrário de partes. Isso é ilustrado na figura 8.13.

****

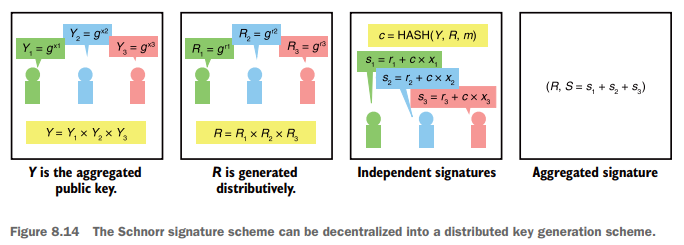
**(Figura 8.13: reconstrução com polinômios de grau n)**

O compartilhamento de segredo é uma técnica frequentemente adotada devido à sua simplicidade. No entanto, para ser útil, as partes da chave precisam ser reunidas em um único local para recriar a chave cada vez que ela for usada em uma operação criptográfica. Isso cria uma janela de oportunidade na qual o segredo fica vulnerável a roubos ou vazamentos acidentais, efetivamente nos levando de volta a um modelo de ponto único de falha.  
Para evitar esse problema de ponto único de falha, existem várias técnicas criptográficas que podem ser úteis em diferentes cenários.

Por exemplo, imagine um protocolo que aceita uma transação financeira somente se ela for assinada por Alice. Isso coloca uma grande responsabilidade sobre Alice, que pode ter medo de ser alvo de atacantes.  
Para reduzir o impacto de um ataque contra Alice, podemos, em vez disso, alterar o protocolo para aceitar (sobre a mesma transação) um número **n** de assinaturas de **n** chaves públicas diferentes, incluindo a de Alice. Um atacante teria que comprometer todas as **n** assinaturas para forjar uma transação válida!  
Tais sistemas são chamados de **multisignatures (multiassinaturas)** e são amplamente adotados no espaço das criptomoedas.

Esquemas ingênuos de multiassinaturas, no entanto, podem adicionar alguma sobrecarga inconveniente. De fato, o tamanho da transação em nosso exemplo cresce linearmente com o número de assinaturas exigidas. Para resolver isso, alguns esquemas de assinatura (como o esquema de assinatura BLS) podem comprimir várias assinaturas em uma única. Isso é chamado de **agregação de assinaturas (signature aggregation)**.  
Alguns esquemas de multiassinaturas vão ainda mais longe na compressão ao permitir que as **n** chaves públicas sejam agregadas em uma única chave pública.  
Essa técnica é chamada de **geração de chave distribuída (DKG — distributed key generation)** e faz parte de um campo da criptografia chamado **computação multipartidária segura (secure multi-party computation)**, que abordarei no capítulo 15.

O DKG permite que **n** participantes calculem colaborativamente uma chave pública sem jamais terem a chave privada associada exposta durante o processo (ao contrário do SSS, não há um dealer).  
Se os participantes quiserem assinar uma mensagem, eles podem então criar uma assinatura colaborativamente usando as partes privadas de cada participante, que podem ser verificadas com a chave pública previamente criada. Novamente, a chave privada nunca existe fisicamente, prevenindo o problema de ponto único de falha do SSS.  
Como você viu as assinaturas Schnorr no capítulo 7, a figura 8.14 mostra a intuição por trás de um esquema Schnorr DKG simplificado.

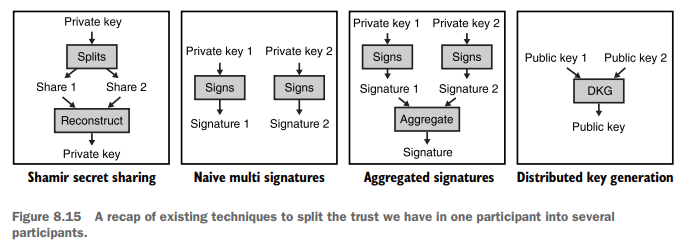
****

**(Figura 8.14: DKG com assinatura Schnorr distribuída)**

Por fim, observe que:

* Cada esquema que mencionei pode ser projetado para funcionar mesmo quando apenas um limiar **m** de **n** participantes participa do protocolo. Isso é muito importante, pois a maioria dos sistemas reais deve tolerar um número de participantes maliciosos ou inativos.
* Esses tipos de esquemas podem funcionar com outros algoritmos criptográficos assimétricos. Por exemplo, usando **criptografia de limiar para descriptografia (threshold encryption)**, um conjunto de participantes pode colaborar para descriptografar assimetricamente uma mensagem.

Eu resumo todos esses exemplos na figura 8.15.

** (Figura 8.15: resumo das técnicas para dividir a confiança entre participantes)**

Os esquemas de limiar são um novo paradigma importante no mundo do gerenciamento de chaves, e é uma boa ideia acompanhar seu desenvolvimento. O NIST atualmente possui um grupo de criptografia de limiar que organiza workshops e tem a intenção de padronizar primitivas e protocolos no longo prazo.

**Resumo**

* Um número é escolhido uniformemente e aleatoriamente de um conjunto se ele foi selecionado com probabilidade igual em relação a todos os outros números desse conjunto.
* **Entropia** é uma métrica para indicar quanta aleatoriedade uma cadeia de bytes possui. Alta entropia refere-se a cadeias de bytes que são uniformemente aleatórias, enquanto baixa entropia refere-se a cadeias de bytes que são fáceis de adivinhar ou prever.
* **Geradores de números pseudoaleatórios (PRNGs)** são algoritmos que tomam uma semente uniformemente aleatória e geram (na prática) uma quantidade quase infinita de aleatoriedade que pode ser usada para propósitos criptográficos (como chaves criptográficas, por exemplo), desde que a semente seja suficientemente grande.
* Para obter números aleatórios, deve-se confiar na biblioteca padrão da linguagem de programação ou em suas bibliotecas criptográficas bem conhecidas. Se estas não estiverem disponíveis, os sistemas operacionais geralmente fornecem interfaces para obter números aleatórios:
  + O Windows oferece a chamada de sistema **BCryptGenRandom**.
  + O Linux e o FreeBSD oferecem a chamada de sistema **getrandom**.
  + Outros sistemas Unix-like geralmente possuem um arquivo especial chamado **/dev/urandom** que exibe aleatoriedade.
* **Funções de derivação de chave (KDFs)** são úteis em cenários onde se deseja derivar segredos a partir de um segredo enviesado, mas com alta entropia.
* O **HKDF (HMAC-based key derivation function)** é a KDF mais amplamente usada e é baseada no HMAC.
* **Gerenciamento de chaves** é o campo de manter segredos… bem, secretos. Consiste principalmente em decidir onde armazenar segredos, expirar e rotacionar proativamente os segredos, decidir o que fazer quando os segredos são comprometidos, e assim por diante.
* Para aliviar o fardo do gerenciamento de chaves, pode-se dividir a confiança de um participante único em vários participantes.

**Parte 1 — Primitivas: Os ingredientes da criptografia**  
**8 Aleatoriedade e segredos**

* 8.1 O que é aleatoriedade?
* 8.2 Aleatoriedade lenta? Use um gerador de números pseudoaleatórios (PRNG)
* 8.3 Obtendo aleatoriedade na prática
* 8.4 Geração de aleatoriedade e considerações de segurança
* 8.5 Aleatoriedade pública
* 8.6 Derivação de chaves com HKDF
* 8.7 Gerenciamento de chaves e segredos
* 8.8 Descentralizar a confiança com criptografia de limiar