**Apêndice — Respostas aos exercícios**

**Capítulo 2**

Você pode dizer se uma função hash fornece ocultação e vinculação se usada como um esquema de comprometimento?

Uma função hash fornece ocultação graças à propriedade de resistência à pré-imagem; isto é, se sua entrada for suficientemente aleatória para que ninguém consiga adivinhá-la. Para resolver isso, você pode gerar um número aleatório e hasheá-lo junto com sua entrada, e posteriormente, pode revelar tanto sua entrada quanto o número aleatório para abrir seu comprometimento. Uma função hash fornece vinculação graças à propriedade de resistência à segunda pré-imagem.

Aliás, não há como essa sequência representar 256 bits (32 bytes), certo? Como isso é seguro então?

Não nos importamos com resistência a colisões. Apenas nos importamos com resistência à segunda pré-imagem. Assim, podemos truncar o resumo para reduzir seu tamanho.

Você consegue adivinhar como o Dread Pirate Roberts (o pseudônimo do webmaster do Silk Road) conseguiu obter um hash que contém o nome do site?

Dread Pirate Roberts criou muitas chaves até que uma delas acabasse hasheando para aquela representação base32 legal. O Facebook fez o mesmo e é acessível a partir de facebookcorewwwi.onion (<https://facebook.com/notes/protect-the-graph/making-connections-to-facebook-more-secure/1526085754298237>). Esses são chamados de **endereços vaidosos** (*vanity addresses*).

**Capítulo 3**

Você consegue descobrir como um contador de comprimento variável poderia possivelmente permitir que um atacante forjasse uma etiqueta de autenticação?

Observando a seguinte mensagem, onde || representa a concatenação de strings, MAC(k, "1" || "1 is my favorite number"), um atacante pode forjar uma etiqueta de autenticação válida para a décima primeira mensagem, MAC(k, "11" | " is my favorite number").

Atenção: nem todos os MACs são PRFs. Consegue ver por quê?

Imagine que a seguinte função é um MAC e PRF válido: MAC(chave, entrada), então a seguinte função seria um MAC válido? NEW\_MAC = MAC(chave, entrada) || 0x01? Seria um PRF válido? É um MAC válido pois previne falsificação, mas não é um PRF válido, pois é fácil distinguir sua saída de uma sequência totalmente aleatória (porque o último byte sempre será 1).

**Capítulo 6**

Usar o mesmo segredo compartilhado com todos seria muito ruim; consegue ver por quê?

Se eu posso criptografar mensagens para você com este segredo compartilhado, eu também posso descriptografar mensagens de outras pessoas.

Você entende por que não se pode usar imediatamente a saída da troca de chaves?

Lembre-se do que você aprendeu no capítulo 5 sobre trocas de chaves. Em (FF)DH, os cálculos ocorrem módulo um número primo grande p. Vamos pegar como exemplo um pequeno número primo, 65.537. Em hexadecimal, nosso p é escrito como 0x010001, e em binário, como 0000 0001 0000 0000 0000 0001. Em binário, repare nos zeros precedendo o primeiro um porque representamos nosso número em bytes (múltiplos de 8 bits).

Se você entende aritmética modular, sabe que números módulo este primo p nunca serão maiores, significando que os primeiros 7 bits sempre serão 0. Além disso, o oitavo bit, na maioria das vezes, será 0 ao invés de 1. Isso não é uniformemente aleatório. Idealmente, cada bit deveria ter a mesma probabilidade de ser 1 ou 0.

**Capítulo 7**

Como você viu no capítulo 3, etiquetas de autenticação produzidas por MACs devem ser verificadas em tempo constante para evitar ataques de temporização. Você acha que precisamos fazer o mesmo para verificação de assinaturas?

Não. Isso porque a verificação de uma etiqueta de autenticação envolve uma chave secreta. A verificação de uma assinatura envolve apenas uma chave pública e, portanto, não precisa ser feita em tempo constante.

**Capítulo 8**

Imagine por um minuto que misturar diferentes fontes de entropia fosse feito simplesmente aplicando XOR entre elas. Consegue ver como isso poderia deixar de ser contributivo?

Uma fonte de entropia com backdoor poderia definir sua saída como o XOR de todas as outras fontes de entropia, efetivamente cancelando toda a entropia para 0.

Esquemas de assinatura como BLS (mencionado na figura 8.5 e no capítulo 7) produzem assinaturas únicas, mas isso não é verdade para ECDSA e EdDSA. Consegue ver por quê?

No ECDSA, o signatário pode escolher diferentes *nonces* para produzir diferentes assinaturas para o mesmo par de chaves e mensagem. Embora o EdDSA seja um algoritmo de assinatura que deriva o *nonce* de forma determinística com base na mensagem a ser assinada, isso não significa que o signatário não possa usar qualquer *nonce* se assim desejar.

**Capítulo 9**

Um comprometimento da chave privada do servidor em algum momento seria devastador, pois atacantes MITM poderiam então descriptografar todas as conversas previamente gravadas. Você entende como isso pode acontecer?

O atacante poderia então retroceder no tempo e se passar pelo servidor no momento em que o *handshake* foi realizado. De fato, o atacante agora possui a chave privada do servidor. Todas as outras informações necessárias para realizar a troca de chaves e derivar as chaves simétricas pós-*handshake* são públicas.

Os valores signatureAlgorithm e signatureValue não estão contidos no certificado propriamente dito, tbsCertificate. Sabe por quê?

A Autoridade Certificadora (CA) precisa assinar o certificado, o que leva a um paradoxo: a assinatura não pode fazer parte da própria assinatura. Assim, a CA deve anexar a assinatura ao certificado. Outros padrões e protocolos podem usar técnicas diferentes. Por exemplo, você poderia incluir a assinatura como parte do tbsCertificate e fingir que ela é composta apenas de zeros ao assinar ou verificar o certificado.

**Capítulo 10**

Você sabe por que o conteúdo do e-mail é comprimido antes de ser criptografado e não depois?

Um *ciphertext* é indistinguível de uma sequência aleatória segundo a definição de um cifrador. Por causa disso, algoritmos de compressão são incapazes de encontrar padrões para comprimir dados criptografados de forma eficiente. Por essa razão, a compressão sempre é aplicada antes da criptografia.

Você consegue pensar em uma maneira não ambígua de assinar uma mensagem?

Em uma linha: autentique o contexto. Uma forma de fazer isso é incluir os nomes do remetente e do destinatário e suas chaves públicas na assinatura e, em seguida, criptografá-la.

**Capítulo 11**

Às vezes, aplicativos tentam resolver o problema de o servidor aprender as senhas dos usuários no registro fazendo com que o cliente aplique hash (talvez com um hash de senha) na senha antes de enviá-la ao servidor. Você consegue determinar se isso realmente funciona?

Aplicar hash no lado do cliente, por si só, não funciona como demonstrado pelo infame ataque *pass-the-hash* (<https://en.wikipedia.org/wiki/Pass_the_hash>); se o servidor armazena diretamente a senha com hash de Alice, então qualquer um que a roube pode usá-la como senha para se autenticar como Alice. Alguns aplicativos realizam tanto o hash no lado do cliente quanto no servidor, o que, nesse caso, pode talvez impedir que um atacante ativo conheça a senha original (embora um atacante ativo possa ser capaz de desabilitar o hash no lado do cliente atualizando o código do aplicativo cliente).

Imagine um protocolo onde você deve inserir o PIN correto de 4 dígitos para se conectar com segurança a um dispositivo. Quais são as chances de adivinhar corretamente um PIN?

É uma chance em 10.000 de acertar. Você ficaria feliz se estivesse jogando na loteria com essas probabilidades.