**3 Códigos de autenticação de mensagem**

Misture uma função de hash com uma chave secreta e você obterá algo chamado código de autenticação de mensagem (MAC), um primitivo criptográfico para proteger a integridade dos dados.  
A adição de uma chave secreta é a base de qualquer tipo de segurança: sem chaves não pode haver confidencialidade, nem autenticação. Enquanto funções de hash podem fornecer autenticação ou integridade para dados arbitrários, elas o fazem graças a um canal adicional confiável que não pode ser adulterado. Neste capítulo, você verá como um MAC pode ser usado para criar tal canal confiável e o que mais ele pode fazer também.

NOTA: Para este capítulo, você precisará ter lido o capítulo 2 sobre funções de hash.

**Este capítulo cobre:**

* Códigos de autenticação de mensagem (MACs)
* As propriedades de segurança e as armadilhas dos MACs
* Os padrões amplamente adotados para MACs

**3.1 Cookies sem estado, um exemplo motivador para MACs**

Imagine o seguinte cenário: você é uma página web. Você é brilhante, cheia de cores e, acima de tudo, orgulhosa de servir uma comunidade de usuários leais. Para interagir com você, os visitantes devem primeiro fazer login enviando suas credenciais, que você deve validar. Se as credenciais corresponderem às que foram usadas quando o usuário se registrou, então você autenticou o usuário com sucesso.

Obviamente, uma experiência de navegação na web não é composta de apenas uma, mas de muitas requisições. Para evitar que o usuário precise se autenticar a cada requisição, você pode fazer o navegador deles armazenar as credenciais e reenviá-las automaticamente em cada requisição. Os navegadores têm um recurso justamente para isso — cookies! Cookies não são apenas para credenciais. Eles podem armazenar qualquer coisa que você queira que o usuário envie a você em cada uma de suas requisições.

Embora essa abordagem ingênua funcione bem, geralmente você não quer armazenar informações sensíveis como senhas de usuários em texto puro no navegador. Em vez disso, um cookie de sessão normalmente carrega uma string aleatória, gerada logo após o login do usuário. O servidor web armazena essa string aleatória em um banco de dados temporário sob o apelido do usuário. Se o navegador publicar o cookie de sessão de alguma forma, nenhuma informação sobre a senha do usuário será vazada (embora ele possa ser usado para personificar o usuário). O servidor web também pode encerrar a sessão excluindo o cookie do seu lado, o que é conveniente.

Não há nada de errado com essa abordagem, mas em alguns casos, pode não ser escalável. Se você tiver muitos servidores, pode ser inconveniente ter todos compartilhando a associação entre seus usuários e as strings aleatórias. Em vez disso, você poderia armazenar mais informações do lado do navegador. Vejamos como podemos fazer isso.

Ingenuamente, você pode fazer o cookie conter um nome de usuário em vez de uma string aleatória, mas isso obviamente é um problema, pois agora eu posso me passar por qualquer usuário modificando manualmente o nome de usuário contido no cookie. Talvez as funções de hash que você aprendeu no capítulo 2 possam nos ajudar. Pense por alguns minutos em uma forma de as funções de hash impedirem um usuário de adulterar seus próprios cookies.

Uma segunda abordagem ingênua poderia ser armazenar não apenas um nome de usuário, mas também um resumo (digest) desse nome de usuário em um cookie. Você pode usar uma função de hash como SHA-3 para hashear o nome de usuário. Eu ilustro isso na figura 3.1.

<IMAGEM>

Há um grande problema com essa abordagem. Lembre-se, a função de hash é um algoritmo público e pode ser recomputada em novos dados por um usuário malicioso. Se você não confiar na origem de um hash, ele não fornece integridade de dados! De fato, a figura 3.2 mostra que se um usuário malicioso modificar o nome de usuário no cookie, ele também pode simplesmente recomputar a parte de resumo do cookie.

<IMAGEM>

Ainda assim, usar um hash não é uma ideia tola. O que mais podemos fazer? Acontece que há um primitivo semelhante à função de hash, um MAC, que fará exatamente o que precisamos.

Um MAC é um algoritmo de chave secreta que recebe uma entrada, como uma função de hash, mas também recebe uma chave secreta (quem poderia imaginar?). Ele então produz uma saída única chamada de tag de autenticação. Este processo é determinístico; dada a mesma chave secreta e a mesma mensagem, um MAC produz a mesma tag de autenticação. Eu ilustro isso na figura 3.3.

<IMAGEM>

**Um exemplo em código**

Para garantir que um usuário não possa adulterar seu cookie, vamos agora utilizar esse novo primitivo. Quando o usuário faz login pela primeira vez, você produz uma tag de autenticação a partir da sua chave secreta e do nome de usuário dele e faz com que ele armazene seu nome de usuário e a tag de autenticação em um cookie. Como ele não conhece a chave secreta, não poderá forjar uma tag de autenticação válida para um nome de usuário diferente.

Para validar o cookie, você faz o mesmo: produz uma tag de autenticação a partir da sua chave secreta e do nome de usuário contido no cookie e verifica se corresponde à tag de autenticação contida no cookie. Se corresponder, deve ter vindo de você, já que você foi o único que poderia ter produzido uma tag de autenticação válida (sob sua chave secreta). Eu ilustro isso na figura 3.4.

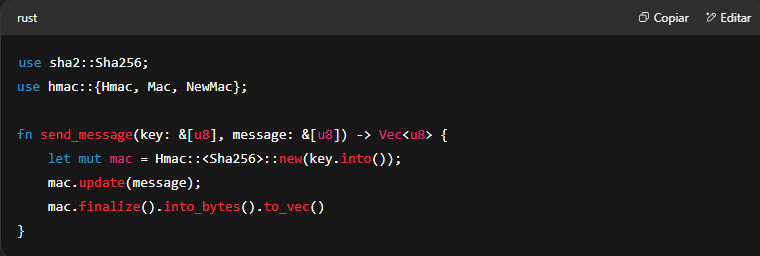
<IMAGEM>

Um MAC é como uma função de hash privada que apenas você pode calcular porque conhece a chave. Em certo sentido, você pode personalizar uma função de hash com uma chave. A relação com funções de hash não termina aí. Você verá mais adiante neste capítulo que MACs são frequentemente construídos a partir de funções de hash. A seguir, vejamos um exemplo diferente usando código real.

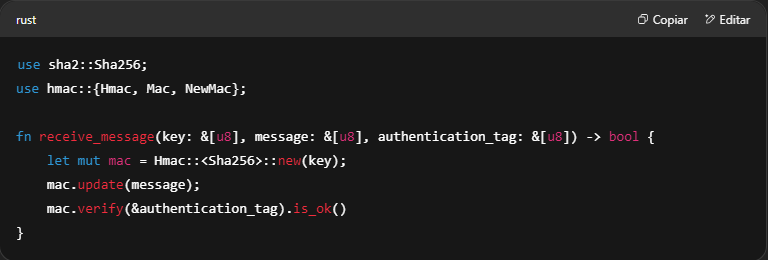
**3.2 Um exemplo em código**

Até agora, você era o único usando um MAC. Vamos aumentar o número de participantes e usar isso como motivação para escrever algum código e ver como os MACs são usados na prática. Imagine que você quer se comunicar com outra pessoa, e você não se importa com outras pessoas lendo suas mensagens. O que realmente importa, porém, é a integridade das mensagens: elas não devem ser modificadas! Uma solução é fazer com que você e seu correspondente usem a mesma chave secreta com um MAC para proteger a integridade de suas comunicações.

Para este exemplo, usaremos uma das funções de MAC mais populares — código de autenticação de mensagem baseado em hash (HMAC) — com a linguagem de programação Rust. HMAC é um código de autenticação de mensagem que usa uma função de hash como núcleo. Ele é compatível com diferentes funções de hash, mas é usado principalmente em conjunto com SHA-2. Como mostra a listagem a seguir, a parte de envio simplesmente pega uma chave e uma mensagem e retorna uma tag de autenticação.



Do outro lado, o processo é semelhante. Após receber tanto a mensagem quanto a tag de autenticação, seu amigo pode gerar sua própria tag com a mesma chave secreta e então comparar as duas. Assim como na criptografia, ambos os lados precisam compartilhar a mesma chave secreta para que isso funcione. A listagem a seguir mostra como isso funciona:



NOTA: Este protocolo não é perfeito: ele permite repetições (replays). Se uma mensagem e sua tag de autenticação forem reproduzidas posteriormente, ainda serão autênticas, mas você não terá como detectar que se trata de uma mensagem antiga sendo reenviada. Mais adiante neste capítulo, falarei sobre uma solução.

Agora que você sabe para que um MAC pode ser usado, falarei sobre algumas das “pegadinhas” dos MACs na próxima seção.

**3.3 Propriedades de segurança de um MAC**

MACs, como todos os primitivos criptográficos, têm suas peculiaridades e armadilhas. Antes de prosseguirmos, fornecerei algumas explicações sobre quais propriedades de segurança os MACs oferecem e como usá-los corretamente. Você aprenderá (nesta ordem) que:

* MACs são resistentes contra a falsificação de tags de autenticação.
* Uma tag de autenticação precisa ter um comprimento mínimo para ser segura.
* Mensagens podem ser reproduzidas se autenticadas de forma ingênua.
* Verificar uma tag de autenticação é propenso a erros.

**3.3.1 Falsificação de tag de autenticação**

O objetivo geral de segurança de um MAC é impedir a falsificação da tag de autenticação em uma nova mensagem. Isso significa que, sem o conhecimento da chave secreta k, não se pode calcular a tag de autenticação t = MAC(k, m) em mensagens m de sua escolha. Isso parece justo, certo? Não podemos calcular uma função se nos falta um argumento.

Os MACs fornecem muito mais garantia do que isso, entretanto. Aplicações do mundo real frequentemente permitem que atacantes obtenham tags de autenticação em algumas mensagens restritas. Por exemplo, esse foi o caso no nosso cenário introdutório, onde um usuário podia obter quase qualquer tag de autenticação ao se registrar com um apelido disponível. Portanto, os MACs precisam ser seguros mesmo contra esses atacantes mais poderosos. Um MAC geralmente vem com uma prova de que, mesmo que um atacante possa pedir que você produza tags de autenticação para um grande número de mensagens arbitrárias, o atacante ainda não deve ser capaz de forjar uma tag de autenticação para uma mensagem nunca antes vista por conta própria.

NOTA: Alguém poderia se perguntar como provar tal propriedade extrema é útil. Se o atacante pode solicitar diretamente tags de autenticação em mensagens arbitrárias, então o que resta para proteger? Mas é assim que as provas de segurança funcionam em criptografia: elas consideram o atacante mais poderoso e mostram que, mesmo assim, o atacante não tem chance. Na prática, o atacante geralmente é menos poderoso e, assim, temos confiança de que, se um atacante poderoso não pode fazer algo ruim, um menos poderoso tem ainda menos recursos.

Assim, você deve estar protegido contra tais falsificações desde que a chave secreta usada com o MAC permaneça secreta. Isso implica que a chave secreta deve ser suficientemente aleatória (mais sobre isso no capítulo 8) e suficientemente longa (normalmente 16 bytes). Além disso, um MAC é vulnerável ao mesmo tipo de ataque ambíguo que vimos no capítulo 2. Se você estiver tentando autenticar estruturas, certifique-se de serializá-las antes de autenticá-las com um MAC; caso contrário, a falsificação pode ser trivial.

**3.3.2 Comprimentos da tag de autenticação**

Outro possível ataque contra o uso de MACs são colisões. Lembre-se, encontrar uma colisão para uma função de hash significa encontrar duas entradas diferentes X e Y tais que HASH(X) = HASH(Y). Podemos estender essa definição para MACs definindo uma colisão quando MAC(k, X) = MAC(k, Y) para entradas X e Y.

Como aprendemos no capítulo 2 com o limite do aniversário (birthday bound), colisões podem ser encontradas com alta probabilidade se o comprimento da saída do nosso algoritmo for pequeno. Por exemplo, com MACs, um atacante que tenha acesso a um serviço produzindo tags de autenticação de 64 bits pode encontrar uma colisão com alta probabilidade solicitando um número bem menor (2³²) de tags. Tal colisão raramente é explorável na prática, mas existem alguns cenários onde a resistência a colisões importa. Por esta razão, queremos um tamanho de tag de autenticação que limite tais ataques. Em geral, são usadas tags de autenticação de 128 bits, pois fornecem resistência suficiente.

“[Solicitar 2⁶⁴ tags de autenticação] levaria 250.000 anos em um link contínuo de 1Gbps, e sem alterar a chave secreta K durante todo esse tempo.”

— RFC 2104 (“HMAC: Keyed-Hashing for Message Authentication,” 1997)

Usar uma tag de autenticação de 128 bits pode parecer contraintuitivo porque queremos saídas de 256 bits para funções de hash. Mas funções de hash são algoritmos públicos que podem ser computados offline, o que permite que um atacante otimize e paralelize um ataque de forma intensa. Com uma função com chave como um MAC, um atacante não pode otimizar o ataque offline de forma eficiente e é forçado a solicitar diretamente as tags de autenticação de você, o que geralmente torna o ataque muito mais lento. Uma tag de autenticação de 128 bits requer 2⁶⁴ consultas online por parte do atacante para ter 50% de chance de encontrar colisões, o que é considerado suficientemente grande. Ainda assim, pode-se querer aumentar uma tag de autenticação para 256 bits, o que também é possível.

**3.3.3 Ataques de repetição (Replay)**

Uma coisa que ainda não mencionei são os ataques de repetição. Vejamos um cenário que é vulnerável a tais ataques. Imagine que Alice e Bob se comunicam em aberto usando uma conexão insegura. Para proteger as mensagens contra adulteração, eles anexam a cada mensagem uma tag de autenticação. Mais especificamente, ambos usam duas chaves secretas diferentes para proteger lados diferentes da conexão (como boa prática). Eu ilustro isso na figura 3.5.

<IMAGEM>

Neste cenário, nada impede que um observador malicioso reproduza uma das mensagens para seu destinatário. Um protocolo que depende de um MAC deve estar ciente disso e construir proteções contra isso. Uma maneira é adicionar um contador incremental à entrada do MAC, como mostrado na figura 3.6.

Na prática, contadores são frequentemente de comprimento fixo de 64 bits. Isso permite o envio de 2⁶⁴ mensagens antes de o contador se esgotar (e correr o risco de dar a volta e se repetir).

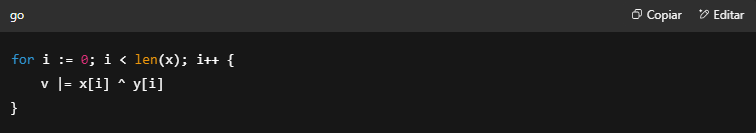
<IMAGEM>

Claro, se o segredo compartilhado for rotacionado frequentemente (significando que após X mensagens, os participantes concordam em usar um novo segredo compartilhado), então o tamanho do contador pode ser reduzido e redefinido para 0 após uma rotação de chave. (Você deve se convencer de que reutilizar o mesmo contador com duas chaves diferentes está OK.) Novamente, contadores nunca são de comprimento variável por causa de ataques ambíguos.

**3.3.4 Verificando tags de autenticação em tempo constante**

Essa última armadilha me é cara, pois encontrei essa vulnerabilidade muitas vezes em aplicações que auditei. Ao verificar uma tag de autenticação, a comparação entre a tag de autenticação recebida e a que você calcula deve ser feita em tempo constante. Isso significa que a comparação deve sempre levar o mesmo tempo, assumindo que a recebida tem o tamanho correto. Se o tempo para comparar as duas tags de autenticação não for em tempo constante, provavelmente é porque retorna no momento em que as duas tags diferem. Isso geralmente fornece informação suficiente para permitir ataques que podem recriar byte por byte uma tag de autenticação válida medindo quanto tempo leva para a verificação terminar. Eu explico isso na tirinha a seguir. Chamamos esses tipos de ataques de ataques de tempo (timing attacks).

Felizmente para nós, bibliotecas criptográficas que implementam MACs também fornecem funções convenientes para verificar uma tag de autenticação em tempo constante. Se você estiver se perguntando como isso é feito, a listagem 3.3 mostra como o Golang implementa uma comparação de tag de autenticação em tempo constante:



O truque é que nenhum desvio condicional (branch) é tomado. Como isso funciona exatamente fica como exercício para o leitor.

**Exercício:**  
Você consegue descobrir como um contador de comprimento variável poderia permitir que um atacante forjasse uma tag de autenticação?

**3.4 MAC no mundo real**

Agora que eu apresentei o que são MACs e quais propriedades de segurança eles fornecem, vamos dar uma olhada em como as pessoas os utilizam em cenários reais. As seções a seguir tratam disso.

**3.4.1 Autenticação de mensagens**

MACs são usados em muitos lugares para garantir que as comunicações entre duas máquinas ou dois usuários não sejam adulteradas. Isso é necessário tanto nos casos em que as comunicações estão em texto claro quanto nos casos em que estão criptografadas. Eu já expliquei como isso acontece quando as comunicações são transmitidas em texto claro, e no capítulo 4, explicarei como isso é feito quando as comunicações são criptografadas.

**3.4.2 Derivação de chaves**

Uma particularidade dos MACs é que eles frequentemente são projetados para produzir bytes que parecem aleatórios (como funções de hash). Você pode usar essa propriedade para implementar uma única chave que gera números aleatórios ou produz mais chaves. No capítulo 8 sobre segredos e aleatoriedade, apresentarei a função de derivação de chave baseada em HMAC (HKDF), que faz exatamente isso usando o HMAC, um dos algoritmos de MAC de que falaremos neste capítulo.

**A função pseudorrandômica (PRF)**

Imagine o conjunto de todas as funções que recebem uma entrada de comprimento variável e produzem uma saída aleatória de tamanho fixo. Se pudéssemos escolher uma função aleatoriamente desse conjunto e usá-la como um MAC (sem uma chave), seria ótimo. Bastaria concordar sobre qual função usar (algo como concordar em uma chave). Infelizmente, não podemos ter tal conjunto, pois ele seria grande demais, mas podemos emular a escolha de tal função aleatória projetando algo suficientemente próximo: chamamos essas construções de funções pseudorrandômicas (PRFs). HMAC e a maioria dos MACs práticos são tais construções. Eles são randomizados por um argumento de chave. Escolher uma chave diferente é como escolher uma função aleatória.

**Exercício:**  
Atenção: nem todos os MACs são PRFs. Você consegue ver por quê?

**3.4.3 Integridade de cookies**

Para rastrear as sessões de navegador de seus usuários, você pode enviar a eles uma string aleatória (associada aos seus metadados) ou enviar diretamente os metadados, anexados com uma tag de autenticação para que não possam ser modificados. Isso é o que expliquei no exemplo introdutório.

**3.4.4 Tabelas de hash**

Linguagens de programação geralmente expõem estruturas de dados chamadas tabelas de hash (também chamadas de hashmaps, dicionários, arrays associativos e assim por diante) que fazem uso de funções de hash não criptográficas. Se um serviço expuser essa estrutura de dados de forma que a entrada da função de hash não criptográfica possa ser controlada por atacantes, isso pode levar a ataques de negação de serviço (DoS), ou seja, um atacante pode tornar o serviço inutilizável. Para evitar isso, a função de hash não criptográfica geralmente é randomizada no início do programa.

Muitas aplicações importantes usam um MAC com uma chave aleatória no lugar da função de hash não criptográfica. Esse é o caso de muitas linguagens de programação (como Rust, Python e Ruby), ou de aplicações importantes (como o kernel do Linux). Todas fazem uso do SipHash, um MAC com nome ruim, otimizado para tags de autenticação curtas, com uma chave aleatória gerada no início do programa.

**3.5 Códigos de autenticação de mensagem (MACs) na prática**

Você aprendeu que MACs são algoritmos criptográficos que podem ser usados entre uma ou mais partes a fim de proteger a integridade e a autenticidade das informações. Como os MACs amplamente utilizados também exibem boa aleatoriedade, os MACs também são frequentemente usados para produzir números aleatórios de forma determinística em diferentes tipos de algoritmos (por exemplo, o algoritmo TOTP — time-based one-time password — que você aprenderá no capítulo 11). Nesta seção, veremos dois algoritmos MAC padronizados que podem ser utilizados atualmente — HMAC e KMAC.

**3.5.1 HMAC, um MAC baseado em hash**

O MAC mais amplamente utilizado é o HMAC (de hash-based MAC), inventado em 1996 por M. Bellare, R. Canetti e H. Krawczyk, e especificado no RFC 2104, na publicação FIPS 198 e no padrão ANSI X9.71. HMAC, como o nome indica, é uma maneira de usar funções de hash com uma chave. Usar uma função de hash para construir MACs é um conceito popular, pois funções de hash possuem implementações amplamente disponíveis, são rápidas em software e também se beneficiam de suporte em hardware na maioria dos sistemas. Lembre-se de que mencionei no capítulo 2 que SHA-2 não deve ser usado diretamente para hashear segredos devido a ataques de extensão de comprimento (length-extension attacks) (mais sobre isso no final deste capítulo). Como se descobre como transformar uma função de hash em uma função com chave? É isso que o HMAC resolve para nós. Sob o capô, o HMAC segue estes passos, que ilustro visualmente na figura 3.7:

1. Primeiro cria duas chaves a partir da chave principal: k1 = k ⊕ ipad e k2 = k ⊕ opad, onde ipad (preenchimento interno) e opad (preenchimento externo) são constantes, e ⊕ é o símbolo da operação XOR.
2. Em seguida, concatena uma chave, k1, com a mensagem e faz o hash disso.
3. O resultado é concatenado com a chave k2 e novamente hasheado.
4. Isso produz a tag de autenticação final.

<IMAGEM>

Como o HMAC é personalizável, o tamanho de sua tag de autenticação é ditado pela função de hash usada. Por exemplo, HMAC-SHA256 faz uso do SHA-256 e produz uma tag de autenticação de 256 bits, HMAC-SHA512 produz uma tag de 512 bits, e assim por diante.

**AVISO:**  
Embora seja possível truncar a saída do HMAC para reduzir seu tamanho, uma tag de autenticação deve ter no mínimo 128 bits, como discutimos anteriormente. Isso nem sempre é respeitado, e algumas aplicações vão até 64 bits devido ao tratamento explícito de uma quantidade limitada de consultas. Existem compensações com essa abordagem, e mais uma vez, é importante ler as especificações antes de fazer algo não padrão.

O HMAC foi construído dessa forma para facilitar as provas de segurança. Em vários artigos, o HMAC foi provado ser seguro contra falsificações desde que a função de hash subjacente mantenha algumas boas propriedades, as quais todas as funções de hash criptograficamente seguras devem possuir. Por causa disso, podemos usar o HMAC em combinação com um grande número de funções de hash. Hoje, o HMAC é usado principalmente com SHA-2.

**3.5.2 KMAC, um MAC baseado em cSHAKE**

Como o SHA-3 não é vulnerável a ataques de extensão de comprimento (isso foi, na verdade, um requisito para a competição do SHA-3), faz pouco sentido usar SHA-3 com HMAC em vez de algo como SHA-3-256(chave || mensagem), que funcionaria bem na prática. É exatamente isso que o KMAC faz.

O KMAC faz uso do cSHAKE, a versão personalizável da função de saída extensível (XOF) SHAKE, que você viu no capítulo 2. O KMAC codifica de forma inequívoca a chave MAC, a entrada e o comprimento de saída solicitado (KMAC é uma espécie de MAC de saída extensível) e fornece isso como entrada para o cSHAKE absorver (veja a figura 3.8). O KMAC também usa “KMAC” como nome de função (para personalizar o cSHAKE) e pode, adicionalmente, receber uma string de personalização definida pelo usuário.

Interessantemente, como o KMAC também absorve o comprimento de saída solicitado, várias chamadas com comprimentos de saída diferentes fornecem resultados totalmente diferentes, o que raramente é o caso para XOFs em geral. Isso torna o KMAC uma função bastante versátil na prática.

<IMAGEM>

**3.6 SHA-2 e ataques de extensão de comprimento**

Mencionamos várias vezes que não se deve hashear segredos com SHA-2, pois ele não é resistente a ataques de extensão de comprimento. Nesta seção, buscamos fornecer uma explicação simples desse ataque.

Voltemos ao nosso cenário introdutório, ao passo em que tentamos simplesmente usar SHA-2 para proteger a integridade do cookie. Lembre-se de que isso não era suficiente, já que o usuário pode adulterar o cookie (por exemplo, adicionando um campo admin=true) e recomputar o hash sobre o cookie. De fato, o SHA-2 é uma função pública e nada impede o usuário de fazer isso. A figura 3.9 ilustra isso.

<IMAGEM>

A ideia seguinte foi adicionar uma chave secreta ao que estamos hasheando. Dessa forma, o usuário não poderia recomputar o resumo, já que a chave secreta seria necessária, assim como um MAC. Ao receber o cookie adulterado, a página computa SHA-256(chave || cookie\_adulterado), onde || representa a concatenação dos dois valores e obtém algo que provavelmente não corresponderá ao que o usuário malicioso enviou. A figura 3.10 ilustra essa abordagem.

<IMAGEM>

Infelizmente, o SHA-2 tem uma peculiaridade incômoda: a partir de um resumo sobre uma entrada, pode-se calcular o resumo de uma entrada e algo mais. O que isso significa? Vejamos a figura 3.11, onde se usa SHA-256(chave || entrada1).

<IMAGEM>

A figura 3.11 é altamente simplificada, mas imagine que entrada1 é a string user=bob. Perceba que o resumo obtido é, efetivamente, o estado intermediário completo da função de hash nesse ponto. Nada impede alguém de fingir que a seção de preenchimento faz parte da entrada e continuar a dança Merkle–Damgård. Na figura 3.12, ilustramos esse ataque, onde se pega o resumo e computa o hash de entrada1 || padding || entrada2. Em nosso exemplo, entrada2 é &admin=true.

<IMAGEM>

Essa vulnerabilidade permite continuar o hashing a partir de um resumo dado, como se a operação não tivesse terminado. Isso quebra nosso protocolo anterior, como a figura 3.13 ilustra.

<IMAGEM>

O fato de que o primeiro padding agora precisa fazer parte da entrada pode impedir que alguns protocolos sejam exploráveis. Ainda assim, a menor alteração pode reintroduzir uma vulnerabilidade. Por essa razão, nunca se deve hashear segredos com SHA-2. É claro, existem outras maneiras corretas de fazer isso (por exemplo, SHA-256(k || mensagem || k) funciona), o que o HMAC fornece. Assim, use HMAC se quiser usar SHA-2 e use KMAC se preferir SHA-3.

**Resumo**

* Códigos de autenticação de mensagem (MACs) são algoritmos criptográficos simétricos que permitem que uma ou mais partes que compartilham a mesma chave verifiquem a integridade e autenticidade das mensagens.
  + Para verificar a autenticidade de uma mensagem e sua tag de autenticação associada, pode-se recomputar a tag de autenticação da mensagem e uma chave secreta, e então comparar as duas tags de autenticação. Se forem diferentes, a mensagem foi adulterada.
  + Sempre compare uma tag de autenticação recebida com uma calculada em tempo constante.
* Embora os MACs protejam a integridade das mensagens por padrão, eles não detectam quando mensagens são repetidas.
* MACs padronizados e amplamente aceitos são os padrões HMAC e KMAC.
* Pode-se usar o HMAC com diferentes funções de hash. Na prática, o HMAC é frequentemente usado com a função de hash SHA-2.
* As tags de autenticação devem ter comprimento mínimo de 128 bits para evitar colisões e falsificação de tags de autenticação.
* Nunca use SHA-256 diretamente para construir um MAC, pois isso pode ser feito de maneira incorreta. Sempre use uma função como o HMAC para isso.

**Conteúdo do Capítulo 3:**

* 3 Códigos de autenticação de mensagem
* 3.1 Cookies sem estado, um exemplo motivador para MACs
* 3.2 Um exemplo em código
* 3.3 Propriedades de segurança de um MAC
  + 3.3.1 Falsificação de tag de autenticação
  + 3.3.2 Comprimentos da tag de autenticação
  + 3.3.3 Ataques de repetição
  + 3.3.4 Verificando tags de autenticação em tempo constante
* 3.4 MAC no mundo real
  + 3.4.1 Autenticação de mensagens
  + 3.4.2 Derivação de chaves
  + 3.4.3 Integridade de cookies
  + 3.4.4 Tabelas de hash
* 3.5 Códigos de autenticação de mensagem (MACs) na prática
  + 3.5.1 HMAC, um MAC baseado em hash
  + 3.5.2 KMAC, um MAC baseado em cSHAKE
* 3.6 SHA-2 e ataques de extensão de comprimento