3 Message Authentication Codes (MACs)

In diesem Kapitel:

- Message Authentication Codes (MACs, Nachrichtenauthentifizierungscodes)
- Sicherheitseigenschaften und Fallstricke von MACs
- Die etablierten Standards für MACs

Mischt man eine Hashfunktion mit einem geheimen Schlüssel, erhält man einen sogenannten *Nachrichtenauthentifizierungscode* (MAC, Message Authentication Code), ein kryptografisches Primitiv, mit dem sich die Integrität von Daten schützen lässt. Das Hinzufügen eines geheimen Schlüssels ist Grundlage für jede Art von Sicherheit: Ohne Schlüssel gibt es weder Vertraulichkeit noch Authentifizierung. Hashfunktionen können zwar Authentifizierung oder Integrität für beliebige Daten gewährleisten, doch geschieht dies in der Regel über einen zusätzlichen vertrauenswürdigen Kanal, der nicht manipuliert werden kann. In diesem Kapitel erfahren Sie, wie sich mit einem MAC ein derartiger vertrauenswürdiger Kanal einrichten lässt und was sonst noch damit möglich ist.

HINWEIS

Für dieses Kapitel sollten Sie Kapitel 2 zum Thema Hashfunktionen gelesen haben.

3.1 Zustandslose Cookies, ein motivierendes Beispiel für MACs

Stellen Sie sich das folgende Szenario vor: Sie sind eine Webseite. Sie sind hell, farbenfroh und vor allem stolz darauf, eine Gemeinschaft treuer Nutzer zu bedienen. Um mit Ihnen zu interagieren, müssen sich Besucher zunächst anmelden, indem sie Ihnen ihre Anmeldeinformationen schicken, die Sie dann überprüfen müssen. Handelt es sich bei den Anmeldeinformationen um diejenigen, die der Benutzer bei

seiner ersten Anmeldung bzw. Registrierung hinterlegt hat, haben Sie den Benutzer erfolgreich authentifiziert.

Natürlich besteht eine Webbrowsing-Sitzung nicht nur aus einer, sondern aus vielen Anfragen. Um zu vermeiden, dass sich der Benutzer bei jeder Anfrage erneut authentifizieren muss, können Sie seinen Browser veranlassen, die Anmeldedaten zu speichern und sie bei jeder Anfrage automatisch erneut zu senden. Browser besitzen hierfür ein Feature, das genau das ermöglicht – Cookies! Allerdings sind Cookies nicht nur für Anmeldeinformationen gedacht. Sie können alles speichern, was der Benutzer mit jeder seiner Anfragen an Sie senden soll.

Obwohl dieser naive Ansatz gut funktioniert, werden Sie im Browser keine vertraulichen Informationen wie Benutzerkennwörter im Klartext speichern wollen. Stattdessen enthält ein Sitzungscookie meist eine zufällige Zeichenfolge, die unmittelbar nach der Anmeldung des Benutzers erzeugt wird. Der Webserver speichert die zufällige Zeichenfolge in einer temporären Datenbank unter dem Kontonamen des Benutzers. Wenn der Browser das Sitzungscookie auf irgendeine Weise veröffentlicht, werden keine Informationen über das Kennwort des Benutzers bekannt (obwohl es verwendet werden kann, um sich als der Benutzer auszugeben). Der Webserver hat auch die Möglichkeit, die Sitzung zu beenden, indem er das Cookie auf der Benutzerseite löscht, was sehr praktisch ist.

Gegen diesen Ansatz ist zwar nichts einzuwenden, doch in manchen Fällen lässt er sich möglicherweise nicht gut skalieren. Wenn Sie viele Server betreiben, könnte es lästig sein, die Zuordnung zwischen Ihren Benutzern und den Zufallszeichenfolgen auf allen Servern einzurichten. Stattdessen könnten Sie mehr Informationen auf der Browserseite speichern. Sehen Sie sich an, wie man das realisieren kann.

Naiverweise kann das Cookie einen Benutzernamen statt einer Zufallszeichenfolge enthalten. Allerdings ist dies offensichtlich ein Problem, da ich mich jetzt als ein beliebiger Benutzer ausgeben kann, indem ich den im Cookie enthaltenen Benutzernamen manuell ändere. Vielleicht können hier die Hashfunktionen weiterhelfen, die Kapitel 2 beschrieben hat. Nehmen Sie sich ein paar Minuten Zeit und überlegen Sie, wie Hashfunktionen einen Benutzer davon abhalten können, seine eigenen Cookies zu manipulieren.

Ein zweiter naiver Ansatz wäre es, nicht nur einen Benutzernamen im Cookie zu speichern, sondern auch noch einen Digest dieses Benutzernamens. Mit einer

Hashfunktion wie SHA-3 können Sie den Benutzernamen hashen. Abbildung 3–1 veranschaulicht dies. Glauben Sie, dass dies funktionieren kann?

Bei diesem Ansatz gibt es ein großes Problem. Wie Sie wissen, ist die Hashfunktion ein öffentlicher Algorithmus und kann von einem böswilligen Benutzer mit neuen Daten neu berechnet werden. Wenn man dem Ursprung des Hashwerts nicht vertrauen kann, bietet er keine Datenintegrität! Abbildung 3–2 zeigt, dass ein böswilliger Benutzer, der den Benutzernamen in seinem Cookie ändert, auch den Digest-Teil des Cookies einfach neu berechnen kann.

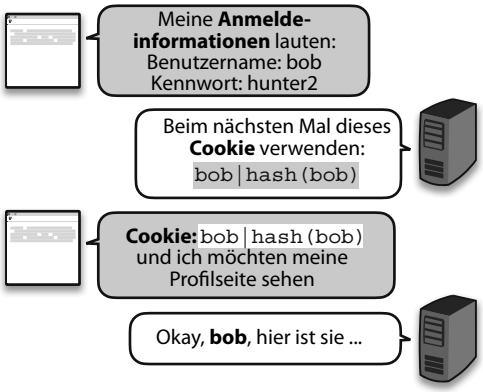


Abb. 3–1 *Um die Anfragen für einen Browser zu authentifizieren, verlangt ein Webserver vom Browser, einen Benutzernamen und einen Hashwert dieses Benutzernamens zu speichern und diese Informationen bei jeder darauf folgenden Anfrage zu senden.*

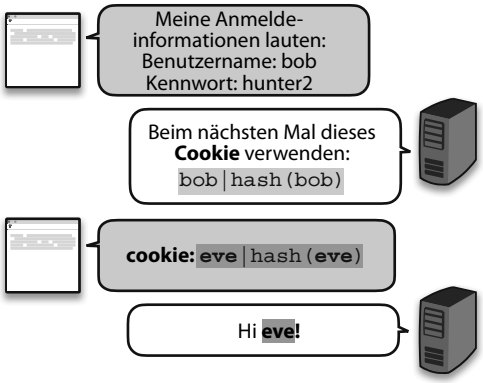


Abb. 3–2 *Ein böswilliger Benutzer kann die in seinen Cookies enthaltenen Informationen modifizieren. Wenn ein Cookie einen Benutzernamen und einen Hash enthält, lassen sich beide modifizieren, um sich als anderer Benutzer auszugeben.*

Dennoch ist es keineswegs töricht, einen Hashwert zu verwenden. Was könnten wir sonst tun? Es zeigt sich, dass es ein ähnliches Primitiv zur Hashfunktion gibt, einen MAC, der genau das tut, was wir brauchen.

Ein *MAC* (Message Authentication Code, Nachrichtenauthentifizierungscode) ist ein geheimer Schlüsselalgorithmus. Er übernimmt wie eine Hashfunktion eine Eingabe, aber auch einen geheimen Schlüssel (wer hätte das gedacht?). Dann erzeugt er eine eindeutige Ausgabe, die als *Authentifizierungstag* bezeichnet wird. Dieser Prozess ist deterministisch. Für den gleichen geheimen Schlüssel und dieselbe Nachricht erzeugt ein MAC das gleiche Authentifizierungstag. Abbildung 3–3 veranschaulicht dies.

Abb. 3–3 *Die Schnittstelle eines Nachrichtenauthentifizierungscodes (MAC). Der Algorithmus übernimmt einen geheimen Schlüssel und eine Nachricht und erzeugt deterministisch ein eindeutiges Authentifizierungstag. Ohne den Schlüssel sollte es nicht möglich sein, dieses Authentifizierungstag zu reproduzieren.*

Um sicherzustellen, dass ein Benutzer sein Cookie nicht manipulieren kann, greifen wir nun auf dieses neue Primitiv zurück. Meldet sich der Benutzer zum ersten Mal an, erzeugen Sie aus Ihrem geheimen Schlüssel und seinem Benutzernamen ein Authentifizierungstag und lassen ihn seinen Benutzernamen und das Authentifizierungstag in einem *Cookie* speichern. Da der Benutzer den geheimen Schlüssel nicht kennt, ist er nicht in der Lage, ein gültiges Authentifizierungstag für einen anderen Benutzernamen zu fälschen.

Um das Cookie zu validieren, gehen Sie genauso vor: Sie erzeugen ein Authentifizierungstag aus Ihrem geheimen Schlüssel und dem Benutzernamen, der im Cookie enthalten ist, und überprüfen, ob dies mit dem im Cookie enthaltenen Authentifizierungstag übereinstimmt. Wenn dies der Fall ist, muss es von Ihnen stammen, da Sie der Einzige waren, der ein gültiges Authentifizierungstag (mit Ihrem geheimen Schlüssel) hätte erzeugen können. Abbildung 3–4 veranschaulicht dies.

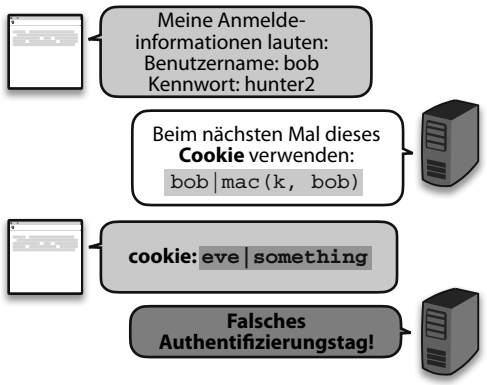


Abb. 3–4 *Ein böswilliger Benutzer manipuliert sein Cookie, kann aber kein gültiges Authentifizierungstag für das neue Cookie fälschen. Infolgedessen kann die Webseite die Authentizität und Integrität des Cookies nicht verifizieren und verwirft daher die Anfrage.*

Ein MAC ist wie eine private Hashfunktion, die nur Sie berechnen können, weil Sie den Schlüssel kennen. In gewisser Weise personalisieren Sie eine Hashfunktion mit einem Schlüssel. Die Beziehung zu Hashfunktionen hört damit aber nicht auf. Später in diesem Kapitel werden Sie sehen, dass MACs oftmals aus Hashfunktionen aufgebaut sind. Als Nächstes sehen wir uns ein anderes Beispiel mit praktischem Code an.

3.2 Ein Beispiel in Code

Bis jetzt waren Sie der Einzige, der einen MAC verwendet hat. Wir wollen nun etwas Code schreiben, um zu sehen, wie man MACs in der Praxis verwendet. Stellen Sie sich vor, Sie wollen mit jemandem kommunizieren und es ist Ihnen egal, ob andere Ihre Nachrichten lesen. Allerdings legen Sie Wert auf die Integrität der Nachrichten – sie dürfen nicht verändert werden! Eine Lösung besteht darin, dass sowohl Sie als auch Ihr Kommunikationspartner die Integrität Ihrer Kommunikation mit demselben Schlüssel und einem MAC schützen.

Für dieses Beispiel verwenden wir eine der beliebtesten MAC-Funktionen – den *hashbasierten Nachrichtenauthentifizierungscode* (HMAC) – mit der Programmiersprache Rust. HMAC ist ein Nachrichtenauthentifizierungscode, der im Kern eine Hashfunktion verwendet. Er ist mit verschiedenen Hashfunktionen kompatibel, wird aber meist mit SHA-2 verwendet. Wie Listing 3–1 zeigt, nimmt der sendende Teil einfach einen Schlüssel sowie eine Nachricht entgegen und gibt ein Authentifizierungstag zurück.

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn send_message(key: &[u8], message: &[u8]) -> Vec<u8> {
    let mut mac = Hmac::<Sha256>::new(key.into());

    mac.update(message);

    mac.finalize().into_bytes().to_vec()
}
```

Instanziiert HMAC mit einem geheimen Schlüssel und der Hashfunktion SHA-256.

Puffert weitere Eingaben für HMAC

Gibt das Authentifizierungstag zurück

Listing 3–1 *Eine authentifizierte Nachricht in Rust senden*

Auf der anderen Seite sieht der Ablauf ähnlich aus. Nachdem Ihr Freund sowohl die Nachricht als auch das Authentifizierungstag empfangen hat, kann er sein eigenes Tag mit demselben geheimen Schlüssel generieren und dann diese vergleichen. Ähnlich wie bei der Verschlüsselung müssen beide Seiten denselben geheimen Schlüssel verwenden, damit dies funktioniert. Listing 3–2 zeigt die Arbeitsweise.

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn receive_message(key: &[u8], message: &[u8],
  authentication_tag: &[u8]) -> bool {
    let mut mac = Hmac::<Sha256>::new(key);
    mac.update(message);

    mac.verify(&authentication_tag).is_ok()
}
```

Der Empfänger muss das Authentifizierungstag mit demselben Schlüssel und derselben Nachricht neu erstellen.

Prüft, ob das reproduzierte Authentifizierungstag mit dem empfangenen übereinstimmt.

Listing 3–2 *Eine authentifizierte Nachricht in Rust empfangen*

Beachten Sie, dass dieses Protokoll nicht perfekt ist: Es lässt Wiederholungen zu. Wenn eine Nachricht und ihr Authentifizierungstag zu einem späteren Zeitpunkt erneut abgespielt werden, sind sie immer noch authentisch, doch Sie haben keine Möglichkeit, zu erkennen, dass es sich um eine ältere Nachricht handelt, die Ihnen erneut gesendet wird. Später in diesem Kapitel stelle ich Ihnen eine Lösung vor. Nachdem Sie nun wissen, wofür ein MAC verwendet werden kann, gehe ich im nächsten Abschnitt auf einige »Fallstricke« von MACs ein.

3.3 Sicherheitseigenschaften eines MAC

Wie alle kryptografischen Primitive besitzen auch MACs ihre Eigenheiten und Fallstricke. Bevor ich näher darauf eingehe, möchte ich erläutern, welche Sicherheitseigenschaften MACs bieten und wie man sie richtig einsetzt. Dabei lernen Sie (in der angegebenen Reihenfolge), dass

- MACs gegen das Fälschen von Authentifizierungstags resistent sind,
- ein Authentifizierungstag eine Mindestlänge haben muss, um sicher zu sein,
- Nachrichten erneut wiedergegeben werden können, wenn sie naiv authentifiziert werden,
- das Verifizieren eines Authentifizierungstags fehleranfällig ist.

3.3.1 Fälschen eines Authentifizierungstags

Mit einem MAC will man im Allgemeinen verhindern, dass ein *Authentifizierungstag* für eine neue Nachricht *gefälscht* wird. Dieses Sicherheitsziel bedeutet, das jemand ohne Kenntnis des geheimen Schlüssels k das Authentifizierungstag $t = MAC(k, m)$ für beliebige Nachrichten m nicht berechnen kann. Das klingt fair, oder? Wir können eine Funktion nicht berechnen, wenn uns ein Argument fehlt.

MACs bieten jedoch viel mehr Sicherheit als das. In praktischen Anwendungen ist es Angreifern oftmals möglich, Authentifizierungstags für einige eingeschränkte Nachrichten zu erhalten. Dies war zum Beispiel in unserem einführenden Szenario der Fall, in dem ein Benutzer nahezu willkürliche Authentifizierungstags bekom-

men konnte, indem er sich mit einem verfügbaren Kontonamen anmeldete. Daher müssen MACs auch gegen diese stärkeren Angreifer sicher sein. Ein MAC enthält normalerweise einen Beweis dafür, dass ein Angreifer selbst nicht in der Lage ist, ein Authentifizierungstag für eine noch nie zuvor gesehene Nachricht zu fälschen, auch wenn der Angreifer Sie auffordern kann, die Authentifizierungstags für eine große Anzahl beliebiger Nachrichten zu erstellen.

HINWEIS

Man könnte sich fragen, wie sinnvoll der Nachweis einer solch extremen Eigenschaft ist. Wenn der Angreifer Authentifizierungstags für beliebige Nachrichten direkt anfordern kann, was bleibt dann noch zu schützen? Aber so funktionieren Sicherheitsbeweise in der Kryptografie: Sie nehmen den stärksten Angreifer und zeigen, dass selbst sein Angriff aussichtslos ist. Da der Angreifer in der Praxis normalerweise weniger mächtig ist, können wir darauf vertrauen, dass er erst recht nicht zum Zuge kommt, wenn schon ein stärkerer Angreifer nichts Böses ausrichten kann.

Gegen solche Fälschungen sollten Sie also geschützt sein, *solange der mit dem MAC verwendete geheime Schlüssel geheim bleibt*. Das impliziert, dass der geheime Schlüssel genügend zufällig (mehr dazu in Kapitel 8) und groß genug (üblicherweise 16 Byte) sein muss. Zudem ist ein MAC anfällig für die gleiche Art von mehrdeutigen Angriffen, wie sie Kapitel 2 gezeigt hat. Wenn Sie Strukturen authentifizieren wollen, müssen Sie sie serialisieren, bevor Sie sie mit einem MAC authentifizieren. Andernfalls könnte es trivial sein, die Struktur zu fälschen.

3.3.2 Längen des Authentifizierungstags

Ein weiterer möglicher Angriff gegen die Verwendung von MACs sind *Kollisionen*. Zur Erinnerung: Eine Kollision für eine Hashfunktion zu finden, bedeutet, zwei verschiedene Eingaben *X* und *Y* zu finden, sodass *HASH(X) = HASH(Y)* ist. Wir können diese Definition auf MACs erweitern, indem wir eine Kollision definieren, wenn *MAC(k, X) = MAC(k, Y)* für Eingaben *X* und *Y*.

Wie Sie in Kapitel 2 mit der Geburtstagsschranke gelernt haben, können Kollisionen mit hoher Wahrscheinlichkeit gefunden werden, wenn die Ausgabelänge unseres Algorithmus klein ist. Bei MACs kann zum Beispiel ein Angreifer, der Zugang zu einem Dienst hat, der 64-Bit-Authentifizierungstags produziert, mit hoher Wahrscheinlichkeit eine Kollision finden, indem er eine viel geringere Anzahl von Tags (2^{32}) anfordert. In der Praxis lässt sich eine derartige Kollision selten ausnutzen, doch es gibt Szenarios, in denen Kollisionsresistenz eine Rolle spielt. Aus diesem Grund wählen wir eine Größe für Authentifizierungstags, die solche Angriffe beschränkt. Im Allgemeinen werden 128-Bit-Authentifizierungstags verwendet, da sie genügend Resistenz bieten.

> *[2^{64} Authentifizierungstags anzufordern,] würde 250.000 Jahre bei einer kontinuierlichen 1-Gbps-Verbindung dauern, ohne dass der geheime Schlüssel K in der gesamten Zeit geändert würde.*
> – RFC 2104 (»HMAC: Keyed-Hashing for Message Authentication«, 1997)

Es mag kontraintuitiv erscheinen, ein 128-Bit-Authentifizierungstag zu verwenden, da wir 256-Bit-Ausgaben für Hashfunktionen benötigen. Allerdings sind Hashfunktionen öffentliche Algorithmen, die man *offline* berechnen kann, was einen Angreifer in die Lage versetzt, einen Angriff zu optimieren und in hohem Maße zu parallelisieren. Mit einer Verschlüsselungsfunktion wie einem MAC kann ein Angreifer den Angriff nicht effizient offline optimieren und ist gezwungen, Authentifizierungstags direkt von Ihnen anzufordern, wodurch der Angriff normalerweise wesentlich langsamer wird. Ein 128-Bit-Authentifizierungstag erfordert 2^{64} *Online*-Anfragen vom Angreifer, um Kollisionen mit einer 50-prozentigen Chance zu finden, was als ausreichend angesehen wird. Nichtsdestotrotz wird man ein Authentifizierungstag auf 256 Bit erhöhen, was ebenfalls möglich ist.

3.3.3 Replay-Angriffe

Was ich bisher noch nicht erwähnt habe, sind *Replay-Angriffe*. Schauen wir uns ein Szenario an, das für solche Angriffe anfällig ist. Stellen Sie sich vor, Alice und Bob kommunizieren offen über eine unsichere Verbindung. Um die Nachrichten vor Manipulationen zu schützen, hängen sie an jede ihrer Nachrichten ein Authentifizierungstag an. Genauer ausgedrückt, verwenden beide zwei verschiedene geheime Schlüssel, um verschiedene Seiten der Verbindung zu schützen (mit den bewährten Verfahrensweisen). Abbildung 3–5 veranschaulicht dies.

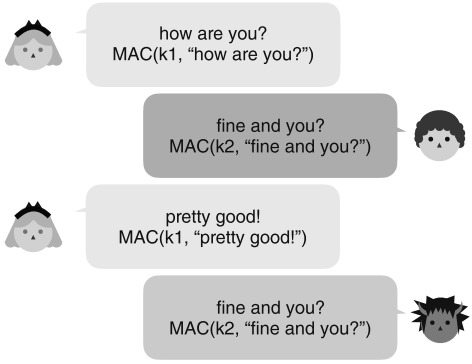


Abb. 3–5 *Zwei Benutzer, die die beiden Schlüssel k1 und k2 gemeinsam nutzen, tauschen Nachrichten mit Authentifizierungstags aus. Diese Tags werden von k1 oder k2 berechnet, abhängig von der Richtung der Nachrichten. Ein böswilliger Beobachter spielt eine der Nachrichten an den Benutzer zurück.*

In diesem Szenario hindert nichts einen böswilligen Beobachter daran, eine der Nachrichten an den Empfänger zurückzuspielen. Ein Protokoll, das sich auf einen

MAC stützt, muss dies aber berücksichtigen und Schutzmechanismen dagegen einrichten. Eine Möglichkeit ist es, einen inkrementierenden Zähler zum Eingang des MAC hinzuzufügen, wie Abbildung 3–6 zeigt.

In der Praxis sind die Zähler oftmals für eine feste Länge von 64 Bit ausgelegt. Damit lassen sich 2^{64} Nachrichten senden, bevor die Zählkapazität erreicht ist (und das Risiko besteht, dass der Zähler überläuft und von vorn beginnt).

Abb. 3–6 *Zwei Benutzer, die die Schlüssel k1 und k2 gemeinsam nutzen, tauschen Nachrichten zusammen mit Authentifizierungstags aus. Diese Tags werden von k1 oder k2 je nach Richtung der Nachrichten berechnet. Ein böswilliger Beobachter spielt eine der Nachrichten an den Benutzer zurück. Da das Opfer seinen Zähler inkrementiert hat, wird das Tag über* `2, fine and you?` *berechnet und stimmt nicht mit dem vom Angreifer gesendeten Tag überein. Somit kann das Opfer die wiederholte Nachricht erfolgreich zurückweisen.*

Wenn natürlich das gemeinsame Geheimnis häufig gewechselt wird (das heißt, die Teilnehmer einigen sich darauf, nach X Nachrichten ein neues gemeinsames Geheimnis zu verwenden), lässt sich die Größe des Zählers reduzieren und nach einer Schlüsselrotation auf 0 zurücksetzen. (Überzeugen Sie sich selbst davon, dass die Wiederverwendung desselben Zählers mit zwei verschiedenen Schlüsseln in Ordnung ist.) Auch hier gilt, dass Zähler aufgrund von zweideutigen Angriffen *niemals eine variable Länge* haben.

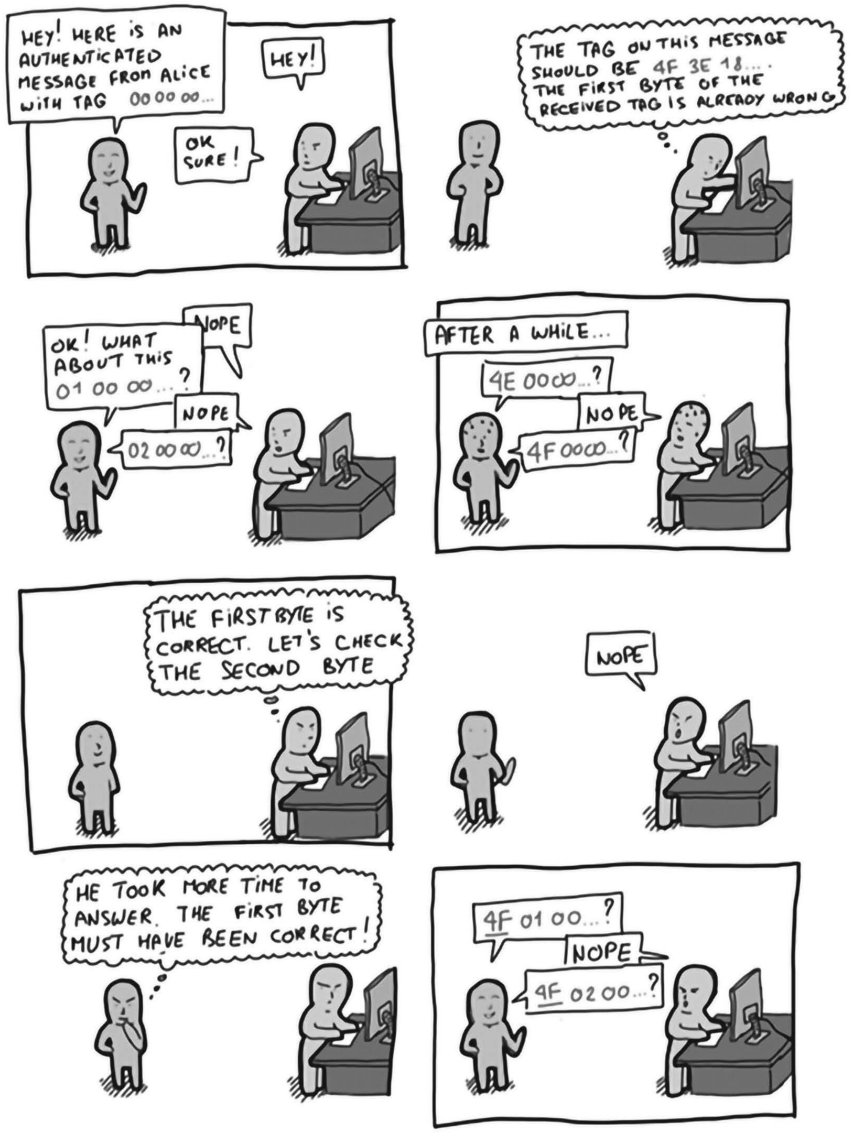
ÜBUNG

Können Sie herausfinden, wie ein Zähler mit variabler Länge es einem Angreifer ermöglichen könnte, ein Authentifizierungstag zu fälschen?

3.3.4 Authentifizierungstags in konstanter Zeit verifizieren

Dieses letzte Problem liegt mir sehr am Herzen, da ich diese Schwachstelle in vielen von mir kontrollierten Anwendungen gefunden habe. Beim Verifizieren eines Authentifizierungstags muss der Vergleich zwischen dem empfangenen Authentifizierungstag und dem von Ihnen berechneten Tag in konstanter Zeit erfolgen. Das

bedeutet, dass der Vergleich immer die gleiche Zeit in Anspruch nehmen sollte, sofern das empfangene Tag die richtige Größe hat. Werden die beiden Authentifizierungstags nicht in konstanter Zeit verglichen, liegt das wahrscheinlich daran, dass der Vergleich schon in dem Moment zurückkehrt, in dem sich die beiden Tags unterscheiden. Dies liefert in der Regel genügend Informationen, um Angriffe zu ermöglichen, die ein gültiges Authentifizierungstag Byte für Byte nachbilden können, indem sie messen, wie lange die eigentliche Verifizierung dauert. Ich erkläre dies mit dem folgenden Comicstrip. Diese Art von Angriffen bezeichnet man als *Timing-Angriffe* oder *Rechenzeitangriffe*.



Erfreulicherweise bieten die kryptografischen Bibliotheken, die MACs implementieren, auch Komfortfunktionen, um ein Authentifizierungstag in konstanter Zeit zu verifizieren. Wenn Sie sich fragen, wie sich das realisieren lässt, zeigt Listing 3–3, wie Golang einen Vergleich von Authentifizierungstags in konstanter Zeit implementiert.

```
for i := 0; i < len(x); i++ {
    v |= x[i] ^ y[i]
}
```

Listing 3–3 *Vergleich in konstanter Zeit in Golang*

Der Trick dabei ist, dass niemals verzweigt wird. Wie dies genau funktioniert, bleibt dem Leser als Übung überlassen.

3.4 MAC im wahren Leben

Nachdem ich das Wesen von MACs und deren Sicherheitseigenschaften erläutert habe, werfen wir einen Blick darauf, wie man sie in der Praxis einsetzt. Die folgenden Abschnitte befassen sich mit diesem Thema.

3.4.1 Authentifizierung von Nachrichten

MACs sind an vielen Stellen gebräuchlich, um zu gewährleisten, dass die Kommunikation zwischen zwei Computern oder zwei Benutzern nicht manipuliert wird. Dies ist sowohl erforderlich, wenn die Kommunikation im Klartext erfolgt, als auch in den Fällen, in denen die Kommunikation verschlüsselt abläuft. Ich habe bereits erklärt, wie dies geschieht, wenn die Nachrichten im Klartext übertragen werden, und in Kapitel 4 erläutere ich, wie dies bei verschlüsselter Kommunikation realisiert wird.

3.4.2 Schlüssel ableiten

Das Besondere bei MACs ist, dass sie oftmals vom Konzept her Bytes erzeugen sollen, die zufällig aussehen (wie Hashfunktionen). Basierend auf dieser Eigenschaft können Sie einen einzelnen Schlüssel implementieren, um Zufallszahlen zu erzeugen oder weitere Schlüssel zu generieren. In Kapitel 8 zu Geheimnissen und Zufälligkeit stelle ich die HMAC-basierte Schlüsselableitungsfunktion (HKDF) vor, die genau das tut, und zwar mit HMAC, einem der MAC-Algorithmen, um die es in diesem Kapitel geht.

Die pseudozufällige Funktion (PRF)

Stellen Sie sich die Menge aller Funktionen vor, die eine Eingabe variabler Länge übernehmen und eine zufällige Ausgabe mit fester Größe erzeugen. Wenn wir eine Funktion aus dieser Menge zufällig auswählen und sie als MAC (ohne Schlüssel) verwenden könnten, wäre das prima. Wir müssten uns nur darauf einigen, welche Funktion das sein soll (ähnlich wie bei der Einigung auf einen Schlüssel). Leider können wir nicht auf eine derartige Menge zurückgreifen, da sie viel zu groß ist. Allerdings können wir das Auswählen einer solchen zufälligen Funktion emulieren, indem wir etwas entwerfen, das dem nahekommt: Derartige Konstruktionen nennen wir *pseudozufällige Funktionen* (Pseudorandom Functions, *PRFs*). HMAC und die meisten praktischen MACs sind solche Konstruktionen. Allerdings werden sie durch ein Schlüsselargument randomisiert. Einen anderen Schlüssel auszuwählen, ist wie das Auswählen einer zufälligen Funktion.

ÜBUNG

Vorsicht: Nicht alle MACs sind PRFs. Können Sie erkennen, warum?

3.4.3 Integrität von Cookies

Um die Browser-Sitzungen Ihrer Benutzer zu verfolgen, können Sie ihnen einen zufälligen String (der ihren Metadaten zugeordnet ist) senden. Oder Sie senden die Metadaten direkt, wobei Sie sie mit einem Authentifizierungstag versehen, damit sie sich nicht verändern lassen. Dies habe ich im Einführungsbeispiel erläutert.

3.4.4 Hashtabellen

Programmiersprachen stellen in der Regel Datenstrukturen bereit, die sogenannten *Hashtabellen* (auch als Hashmaps, Dictionaries, assoziative Arrays usw. bezeichnet), die nicht-kryptografische Hashfunktionen verwenden. Wenn ein Dienst diese Datenstruktur in einer Weise zugänglich macht, dass sich die Eingabe der nicht-kryptografischen Hashfunktion von Angreifern steuern lässt, kann dies zu *DoS-(Denial of Service-)Angriffen* führen, das heißt, ein Angreifer kann den Dienst unbrauchbar machen. Um dies zu vermeiden, wird die nicht-kryptografische Hashfunktion normalerweise beim Start des Programms randomisiert.

Viele wichtige Anwendungen verwenden einen MAC mit einem zufälligen Schlüssel anstelle der nicht-kryptografischen Hashfunktion. Dies ist bei zahlreichen Programmiersprachen (wie Rust, Python und Ruby) oder bedeutenden Anwendungen (wie dem Linux-Kernel) der Fall. Alle verwenden sie *SipHash*, einen unpassend benannten MAC, der für kurze Authentifizierungstags optimiert ist, mit einem zufälligen Schlüssel, der beim Programmstart generiert wird.

3.5 MACs in der Praxis

Wie Sie gelernt haben, sind Nachrichtenauthentifizierungscodes (MACs) kryptografische Algorithmen, die zwischen einer oder mehreren Parteien verwendet werden können, um die Integrität und Authentizität von Informationen zu schützen. Da gängige MACs auch eine gute Zufälligkeit bieten, setzt man sie oftmals auch in verschiedenen Arten von Algorithmen ein, um deterministische Zufallszahlen zu erzeugen (zum Beispiel im TOTP-Algorithmus zur Erzeugung von zeitlich limitierten Einmalkennwörtern, den Sie in Kapitel 11 kennenlernen werden). In diesem Abschnitt sehen wir uns zwei standardisierte MAC-Algorithmen an, die man heutzutage verwenden kann – HMAC und KMAC.

3.5.1 HMAC, ein Hash-basierter MAC

Der am weitesten verbreitete MAC ist HMAC (für Hash-basierter MAC), den M. Bellare, R. Canetti und H. Krawczyk 1996 erfunden haben und der in RFC 2104, FIPS Publication 198 und ANSI X9.71 spezifiziert ist. Wie sein Name verrät, bietet HMAC eine Möglichkeit, Hashfunktionen mit einem Schlüssel zu verwenden. MACs mithilfe von Hashfunktionen zu erstellen, ist ein beliebtes Konzept, da Hashfunktionen in zahlreichen Implementierungen existieren, in Software schnell sind und auf den meisten Systemen auch von der Hardwareunterstützung profitieren. Wie ich in Kapitel 2 erläutert hatte, sollte SHA-2 aufgrund von *Length-Extension-Angriffen* nicht direkt verwendet werden, um Geheimnisse zu hashen (mehr dazu am Ende dieses Kapitels). Wie kann man herausfinden, wie man eine Hashfunktion in eine Verschlüsselungsfunktion umwandelt? HMAC löst diese Frage für uns. Hinter den Kulissen folgt HMAC den Schritten, die in Abbildung 3–7 visuell dargestellt sind:

1. Zuerst werden aus dem Hauptschlüssel zwei Schlüssel erzeugt: $k1 = k \oplus ipad$ und $k2 = k \oplus opad$, wobei *ipad* (innere Auffüllung) und *opad* (äußere Auffüllung) Konstanten sind und $\oplus$ die XOR-Operation symbolisiert.

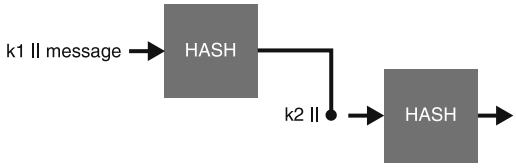


Abb. 3–7 *HMAC bildet einen Hashwert aus der Verkettung (||) eines Schlüssels* k1 *mit der Eingabenachricht und hasht dann die Verkettung eines Schlüssels* k2 *mit der Ausgabe der ersten Operation. Die Schlüssel* k1 *und* k2 *werden beide deterministisch aus einem geheimen Schlüssel* k *abgeleitet.*

2. Dann verkettet die Funktion einen Schlüssel `k1` mit der Nachricht und erzeugt dafür einen Hashwert.
3. Das Ergebnis wird mit dem Schlüssel `k2` verkettet und wieder gehasht.
4. Daraufhin entsteht das endgültige Authentifizierungstag.

Da HMAC anpassbar ist, ergibt sich die Größe des Authentifizierungstags aus der verwendeten Hashfunktion. Zum Beispiel verwendet HMAC-SHA256 die Hashfunktion SHA-256 und erzeugt ein Authentifizierungstag von 256 Bit, während HMAC-SHA512 ein Authentifizierungstag von 512 Bit erzeugt usw.

ACHTUNG

Man kann zwar die Ausgabe von HMAC beschneiden, um die Größe zu reduzieren, doch sollte ein Authentifizierungstag mindestens 128 Bit lang sein, wie ich weiter vorn erläutert habe. Dies wird allerdings nicht immer eingehalten und manche Anwendungen gehen bis zu 64 Bit herunter, weil sie ausdrücklich nur eine begrenzte Anzahl von Anfragen verarbeiten. Dieser Ansatz lebt mit Kompromissen und es ist wichtig, das Kleingedruckte zu lesen, bevor Sie etwas tun, das nicht dem Standard entspricht.

HMAC wurde im Hinblick darauf konstruiert, Beweise zu erleichtern. In mehreren Papern wird HMAC als sicher gegen Fälschungen bewiesen, solange die darunter liegende Hashfunktion einige gute Eigenschaften aufweist, wie es bei allen kryptografisch sicheren Hashfunktionen der Fall sein sollte. Infolgedessen können wir HMAC in Kombination mit einer großen Anzahl von Hashfunktionen nutzen. Heutzutage wird HMAC meistens mit SHA-2 eingesetzt.

3.5.2 KMAC – ein MAC, der auf cSHAKE basiert

Da SHA-3 nicht anfällig für Length-Extension-Angriffe ist (was sogar eine Anforderung für den SHA-3-Wettbewerb war), macht es wenig Sinn, SHA-3 mit HMAC zu verwenden, anstatt etwas wie `SHA-3-256(key || message)` zu senden, was in der Praxis gut funktioniert. Genau das ist es, was *KMAC* tut.

KMAC stützt sich auf cSHAKE, die anpassbare Version der erweiterbaren Ausgabefunktion SHAKE, die Sie in Kapitel 2 kennengelernt haben. KMAC codiert eindeutig den MAC-Schlüssel, die Eingabe und die angeforderte Ausgabelänge (KMAC ist eine Art von MAC mit erweiterbarer Ausgabe) und übergibt dieses an cSHAKE als zu absorbierende Eingabe (siehe Abb. 3–8). Zudem verwendet KMAC die Zeichenfolge »KMAC« als Funktionsname (um cSHAKE anzupassen) und kann darüber hinaus einen benutzerdefinierten Anpassungsstring übernehmen.

Da KMAC auch die angeforderte Ausgabelänge absorbiert, liefern mehrere Aufrufe mit verschiedenen Ausgabelängen interessanterweise vollkommen unterschiedliche Ergebnisse, was bei XOFs im Allgemeinen selten der Fall ist. Dies macht KMAC in der Praxis zu einer recht vielseitigen Funktion.

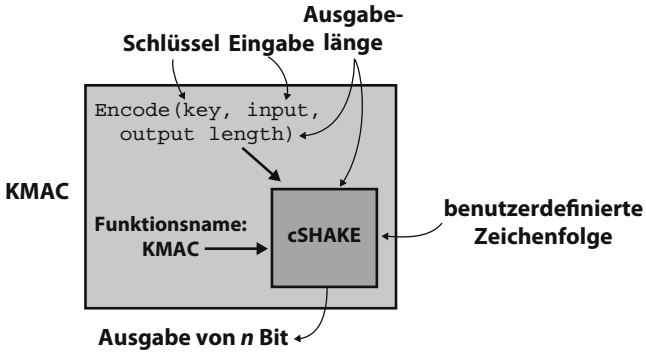


Abb. 3–8 *KMAC ist einfach ein Wrapper um cSHAKE. Um einen Schlüssel zu verwenden, codiert der Algorithmus (auf eindeutige Weise) den Schlüssel, die Eingabe und die Ausgabelänge als Eingabe für cSHAKE.*

3.6 SHA-2- und Length-Extension-Angriffe

Ich habe bereits mehrfach erwähnt, dass man Geheimnisse nicht mit SHA-2 hashen sollte, da diese Funktion gegenüber *Length-Extension-Angriffen* nicht resistent ist. In diesem Abschnitt möchte ich eine einfache Erklärung für derartige Angriffe geben.

Kehren wir zu unserem Einführungsszenario zurück, und zwar zu dem Schritt, bei dem wir versucht haben, die Integrität des Cookies einfach per SHA-2 zu schützen. Das hat sich aber als nicht gut genug herausgestellt, da der Benutzer die Möglichkeit hat, das Cookie zu manipulieren (indem er zum Beispiel ein Feld `admin=true` hinzufügt) und den Hash über dem Cookie neu zu berechnen. Denn SHA-2 ist eine öffentliche Funktion und niemand hindert den Benutzer daran, solche Manipulationen auszuführen. Abbildung 3–9 veranschaulicht dies.

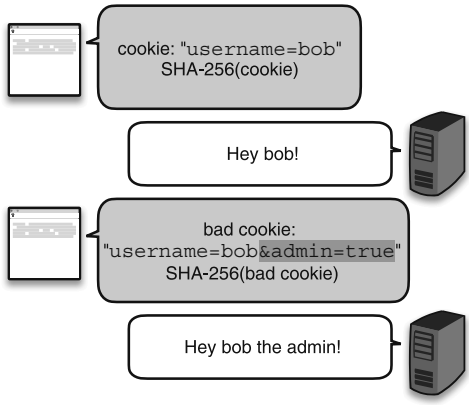


Abb. 3–9 *Eine Webseite sendet ein Cookie gefolgt von einem Hash dieses Cookies an einen Benutzer. Der Benutzer muss dann das Cookie in jeder darauf folgenden Anfrage senden, um sich selbst zu authentifizieren. Leider kann ein böswilliger Benutzer das Cookie manipulieren und den Hash neu berechnen, wodurch die Integritätsprüfung ausgehebelt wird. Die Webseite akzeptiert dann das Cookie als gültig.*

Die nächstbeste Idee war, einen geheimen Schlüssel hinzuzufügen und darüber den Hash zu bilden. Auf diese Weise kann der Benutzer den Digest nicht neu berechnen, da der geheime Schlüssel erforderlich ist, ähnlich wie bei einem MAC. Beim Empfang des manipulierten Cookies berechnet die Seite `SHA-256(key || tampered_cookie)`, wobei `||` die Verkettung der beiden Werte darstellt, sodass das Ergebnis wahrscheinlich nicht mit dem übereinstimmt, was der böswillige Benutzer gesendet hat. Abbildung 3–10 veranschaulicht diesen Ansatz.

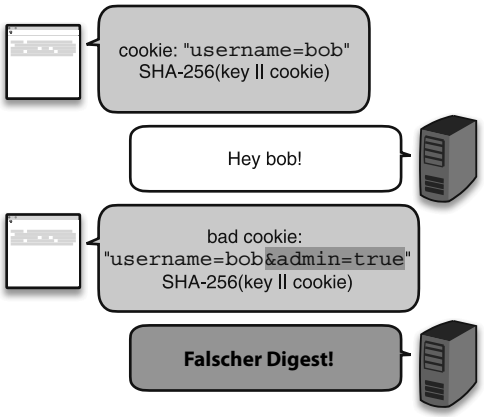


Abb. 3–10 *Da der Hash des Cookies mit dem Schlüssel gebildet wird, könnte man meinen, dass ein böswilliger Benutzer, der sein eigenes Cookie manipulieren will, nicht in der Lage wäre, den korrekten Digest über dem neuen Cookie zu berechnen. Später werden Sie sehen, dass dies bei SHA-256 nicht der Fall ist.*

Leider weist SHA-2 eine störende Eigenheit auf: Von einem Digest über einer Eingabe kann man den Digest einer Eingabe und weitere Informationen berechnen. Was bedeutet das? Sehen Sie sich Abbildung 3–11 an, wo jemand SHA-256 als `SHA-256(secret||input1)` verwendet.

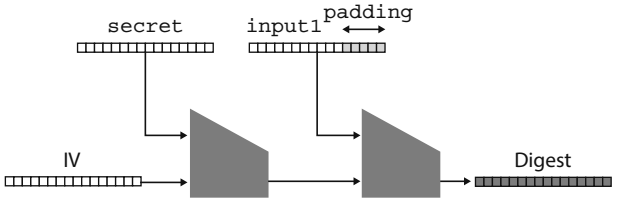


Abb. 3–11 *SHA-256 bildet den Hashwert eines Geheimnisses, das mit einem Cookie (hier* `input1` *genannt) verkettet ist. Wie Sie wissen, arbeitet SHA-256 mithilfe der Merkle-Damgård-Konstruktion, um iterativ eine Komprimierungsfunktion über Blöcke der Eingabe aufzurufen, ausgehend von einem Initialisierungsvektor (IV).*

Abbildung 3–11 ist zwar stark vereinfacht, aber stellen Sie sich vor, dass `input1` der String `user=bob` ist. Beachten Sie, dass der erhaltene Digest effektiv der vollständige Zwischenzustand der Hashfunktion an diesem Punkt ist. Nichts hindert

uns daran, so zu tun, als sei der Padding-Abschnitt Teil der Eingabe, um den Merkle-Damgård-Tanz fortzusetzen. Abbildung 3–12 veranschaulicht diesen Angriff, bei dem jemand den Digest nimmt und den Hash von `input1 || padding || input2` berechnet. In unserem Beispiel hat `input2` den Wert `&admin=true`.

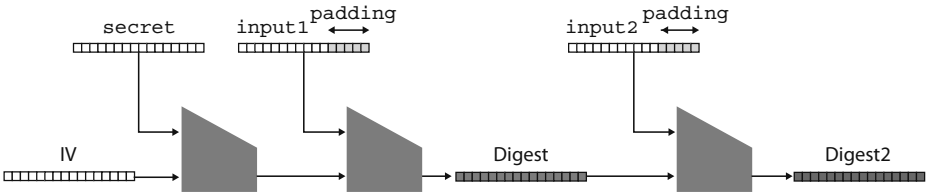


Abb. 3–12 *Die Ausgabe der Hashfunktion SHA-256 für ein Cookie (der mittlere Digest) wird verwendet, um den Hash auf weitere Daten zu erweitern und einen Hash (den rechten Digest) des Geheimnisses, das mit* `input1`*, den ersten Auffüllungsbytes (*`padding`*) und* `input2` *verkettet ist, zu erzeugen.*

Diese Schwachstelle ermöglicht es, das Hashing ausgehend von einem gegebenen Digest fortzusetzen, als ob die Operation nicht beendet worden wäre. Dies verletzt unser vorheriges Protokoll, wie Abbildung 3–13 verdeutlicht.

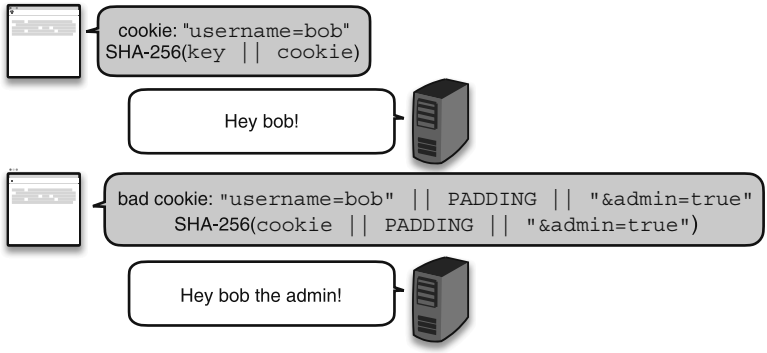


Abb. 3–13 *Ein Angreifer verwendet erfolgreich einen Length-Extension-Angriff, um sein Cookie zu manipulieren, und berechnet den richtigen Hashwert mithilfe des vorherigen Hashwerts.*

Die Tatsache, dass die erste Auffüllung nun Teil der Eingabe sein muss, könnte bedeuten, dass sich einige Protokolle nicht nutzen lassen. Dennoch kann die kleinste Änderung erneut eine Schwachstelle einführen. Aus diesem Grund sollte man niemals Geheimnisse mit SHA-2 hashen. Natürlich gibt es verschiedene andere Möglichkeiten, dies korrekt zu tun (zum Beispiel funktioniert `SHA-256(k||message||k)`). Und genau das bietet HMAC. Verwenden Sie also HMAC, wenn Sie mit SHA-2 arbeiten, und KMAC, wenn Sie SHA-3 bevorzugen.

Zusammenfassung

- Message Authentication Codes (MACs, Nachrichtenauthentifizierungscodes) sind symmetrische kryptografische Algorithmen, die es einer oder mehreren Parteien, die sich denselben Schlüssel teilen, ermöglichen, die Integrität und Authentizität von Nachrichten zu verifizieren.
  - Um die Authentizität einer Nachricht und ihres zugeordneten Authentifizierungstags zu verifizieren, kann man das Authentifizierungstag der Nachricht und einen geheimen Schlüssel neu berechnen und dann die beiden Authentifizierungstags vergleichen. Unterscheiden sie sich, wurde die Nachricht manipuliert.
  - Vergleichen Sie immer ein empfangenes Authentifizierungstag mit einem berechneten Tag in konstanter Zeit.
- MACs schützen zwar standardmäßig die Integrität von Nachrichten, doch sie erkennen nicht, wenn Nachrichten erneut abgespielt werden.
- Standardisierte und etablierte MACs sind die HMAC- und die KMAC-Standards.
- HMAC kann man mit verschiedenen Hashfunktionen verwenden. In der Praxis wird HMAC oftmals mit der Hashfunktion SHA-2 verwendet.
- Authentifizierungstags sollten eine minimale Länge von 128 Bit haben, um Kollisionen und Fälschungen von Authentifizierungstags zu verhindern.
- Verwenden Sie SHA-256 niemals direkt, um einen MAC zu erstellen, da dies fehlerhaft geschehen kann. Nehmen Sie dazu immer eine Funktion wie HMAC.