3.2 HMAC-функции

HMAC-функции (hash-based message authentication code; код проверки подлинности сообщений, использующий хеш-функции) – распространенный способ применить любую хеш-функцию для проверки подлинности данных, как если бы ей можно было передать ключ. HMAC-функция принимает три параметра: сообщение, ключ и рядовую криптографическую хеш-функцию (рис. 3.2). Да, все верно: третьим аргументом функция ожидает другую функцию, чтобы перепоручить ей всю тяжелую работенку. HMAC-функция возвращает код проверки подлинности – MAC, – который и упоминается в ее названии. На самом деле MAC – всего лишь особый случай хеша. В этой книге ради простоты употребляется слово «хеш» вместо MAC.

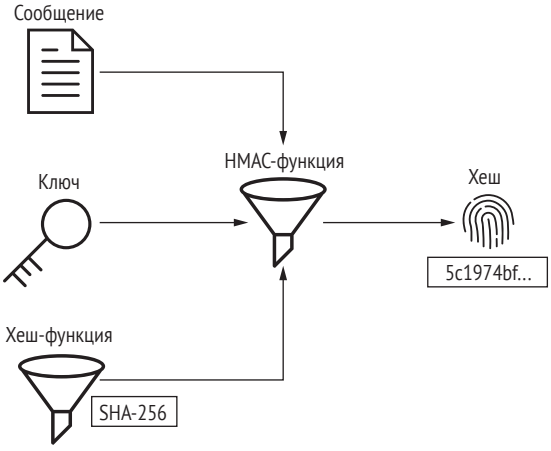

Рис. 3.2 HMAC-функции принимают три аргумента: сообщение, ключ и хеш-функцию

> **СОВЕТ** Вам пригодится знать HMAC-функции на зубок. Они понадобятся при решении многих задач, которые вам далее повстречаются в книге. В частности, вы их еще увидите, когда мы коснемся шифрования, управления сеансами, регистрации пользователей и сброса их паролей.

В Python для HMAC существует одноименный модуль `hmac`. В примере ниже функции этого модуля передается сообщение, ключ и ссылка на конструктор хеш-функции SHA-256. Ссылку на хеш-функцию ожидает именованный аргумент `digestmod`. В него допустимо передать любой конструктор хеш-функции из модуля `hashlib`.

```
>>> import hashlib
>>> import hmac
>>>
>>> hmac_sha256 = hmac.new(
...     b'key', msg=b'message', digestmod=hashlib.sha256)
```

ВНИМАНИЕ! Начиная с Python 3.8 именованный аргумент `digestmod` стал обязательным. Обязательно указывайте его, чтобы ваш код работал под разными версиями Python.

Экземпляр HMAC-функции повторяет поведение хеш-функции, переданной вовнутрь. Методы `digest` и `hexdigest` и свойство `digest_size` вам уже знакомы:

```
>>> hmac_sha256.digest() b"n\x9e\xf2\x9bu\xff\xfc[z\xba\xe5'\xd5\x8f\xda\xdb/
\xe4.r\x19\x01\x19v\x91 sC\x06_X\xedJ"
```
← Хеш в виде байтовой строки

```
>>> hmac_sha256.hexdigest()
'6e9ef29b75fffc5b7abae527d58fdadb2fe42e7219011976917343065f58ed4a'
```
← Хеш в виде шестнадцатеричного текста

```
>>> hmac_sha256.digest_size
32
```
← Длина хеша

Хеш-функция, находящаяся под капотом, определяет имя HMAC-функции. Например, если передать туда SHA-256, то результат будет называться HMAC-SHA256:

```
>>> hmac_sha256.name
'hmac-sha256'
```

HMAC-функции предназначены и в целом используются для проверки подлинности сообщений. Бывает, как в случае с системой документооборота Алисы, что сообщение записывается и читается одним и тем же приложением. Но также бывает, что чтение и запись сообщения разнесены между собой. Следующий раздел описывает подобный сценарий.

3.2.1 Проверка подлинности данных между системами

Допустим, Алисина система теперь должна принимать документы от Боба. Алиса хочет знать наверняка, что по пути ни одно сообщение не было изменено каверзной Мэллори. Алиса и Боб договорились о взаимодействии:

- Алиса и Боб согласовали общий секретный ключ;
- Боб высчитывает хеш документа через HMAC-функцию;
- Боб отправляет документ и его хеш Алисе;
- Алиса высчитывает хеш документа через HMAC-функцию;
- Алиса сравнивает свой хеш с хешем Боба.

Рисунок 3.3 иллюстрирует их действия. Если хеш, пришедший со стороны Боба, совпадает с хешем, который получается у Алисы, то можно утверждать два факта:

- сообщение отправил некто, обладающий тем же ключом; вероятно, Боб;
- Мэллори не могла изменить это сообщение во время его передачи.

Рис. 3.3 Алиса удостоверяется, что данные присланы Бобом, с помощью общего ключа и HMAC-функции

Вот листинг, который показывает реализацию этого взаимодействия на стороне Боба. Для хеширования сообщения используется HMAC-SHA256.

Листинг 3.3 Боб применяет HMAC-функцию перед отправкой сообщения

```
import hashlib
import hmac
import json

hmac_sha256 = hmac.new(b'shared_key', digestmod=hashlib.sha256)
message = b'from Bob to Alice'
hmac_sha256.update(message)
hash_value = hmac_sha256.hexdigest()

authenticated_msg = {
   'message': list(message),
   'hash_value': hash_value, }
outbound_msg_to_alice = json.dumps(authenticated_msg)
```

Боб хеширует документ

Хеш отправляется вместе с документом

А вот реализация со стороны Алисы. Она тоже использует HMAC-SHA256 для подсчета хеша полученного сообщения. Если оба MAC совпадают, считается, что сообщение подлинное.

Листинг 3.4 Алиса применяет HMAC-функцию после получения весточки от Боба

```
import hashlib
import hmac
import json

authenticated_msg = json.loads(inbound_msg_from_bob)
message = bytes(authenticated_msg['message'])

hmac_sha256 = hmac.new(b'shared_key', digestmod=hashlib.sha256)
hmac_sha256.update(message)
hash_value = hmac_sha256.hexdigest()

if hash_value == authenticated_msg['hash_value']:
    print('доверенное сообщение')
    ...
```

Алиса высчитывает хеш на своей стороне

И сравнивает оба значения

Даже если это сообщение будет идти через Мэллори, то у нее никак не получится заставить Алису поверить измененному сообщению. Так как у злоумышленницы нет ключа, который есть у Алисы и Боба, она не может высчитать подходящий сообщению хеш. Если взломщица изменит сообщение либо хеш на пути к получателю, то пришедший хеш не будет совпадать с хешем, который посчитает Алиса.

Приглядитесь к последним строкам листинга 3.4. Обратите внимание: Алиса использует оператор ==, чтобы сравнить хеши между собой. Хотите верьте, хотите нет, но эта деталь открывает для Мэллори уязвимость, которой она может воспользоваться. Как злоумышленники пользуются атаками по времени, расскажет следующий раздел.

3.3 Атака по времени

В основе проверки как целостности данных, так и подлинности сообщения лежит сравнение хешей. Казалось бы, что может быть проще, чем сравнить две строки, но эта простота обманчива. Оператор == выдает False, как только повстречает первое несоответствие между его операндами. Как минимум ему придется сравнить первый символ, а как максимум – в случае полного либо почти полного совпадения – придется сравнить все символы. Самое важное здесь заключается в том, что оператор == будет сравнивать строки дольше, если они начинаются одинаково. Думаю, вы уже заметили уязвимость.

Итак, Мэллори атакует. Сперва она создает документ, который она хочет подсунуть Алисе как файл от Боба. Взломщица не знает, какой хеш от этого документа получится у Алисы, ведь у нее нет ключа. Но из передаваемого сообщения она знает, что хеш длиной 64 символа и это шестнадцатеричный текст – у каждого символа 16 возможных значений.

Следующим шагом злоумышленница хочет знать, какой у правильного хеша первый символ. Возможных вариантов всего шестнадцать – и Мэллори создает шестнадцать хешей, которые начинаются с разных символов. Она берет первый хеш, прикрепляет его к файлу и отправляет Алисе. Затем повторяет это еще пятнадцать раз с оставшимися хешами. После каждой отправки она измеряет и записывает время, за которое система документооборота ответила ей. Взломщица повторяет одни и те же отправки много-много раз, чтобы накопить статистику времени ответа. В какой-то момент ей становится понятно, что система отвечает чуточку медленнее на один из шестнадцати вариантов. Таким образом становится известен первый из 64 символов верного хеша. Рисунок 3.4 показывает, как Мэллори узнает первый символ корректного хеша.

Рис. 3.4 Если хеш начинается с *b*, то система откликается чуточку позднее. Первый символ найден

Затем Мэллори повторяет процедуру для 63 оставшихся символов 64-значного хеша, и таким образом узнаёт хеш целиком. Это пример *атаки по времени*. Ее принцип заключается в том, что взломщик узнаёт информацию, о которой знать не должен, опосредованно – по времени отклика системы. Злоумышленник измеряет время, которое тратит сервис на операцию, и по замерам строит догадки о недоступном ему содержимом. Здесь этой операцией является сравнение строк.

Безопасные системы сравнивают хеши за постоянное время. Малая толика быстродействия намеренно жертвуется ради закрытия уязвимости. Модуль `hmac` содержит функцию `compare_digest`, которая сравнивает хеши за одинаковое время. Она работает как оператор `==`, но отличается от него временно́й сложностью алгоритма. Когда `compare_digest` обнаруживает разницу хешей, она не возвра-

щает результат преждевременно, а все равно сравнивает все символы. В результате и среднее, и максимальное, и минимальное время работы равны между собой. Взломщик по-прежнему может подать на вход системе произвольный хеш для сравнения, но узнать второй операнд с помощью атаки по времени уже не получится:

```
>>> from hmac import compare_digest
>>>
>>> compare_digest('alice', 'mallory')  | Разные аргументы,
False                                   | время выполнения неизменно
>>> compare_digest('alice', 'alice')    | Одинаковые аргументы,
True                                    | время выполнения неизменно
```

Всегда сравнивайте хеши через `compare_digest`. Для перестраховки сравнивайте их через `compare_digest`, даже если это просто проверка на целостность данных. Эта функция принимает на вход как обыкновенные строки, так и байтовые. Она еще не раз повстречается в книге: она будет появляться в примерах кода, как это было только что.

Атака по времени является подтипом атак по сторонним каналам. *Атаки по сторонним каналам* задействуют информацию о физических процессах, чтобы опосредованно извлечь недоступные данные. Время, издаваемые звуки, потребление электроэнергии, электромагнитное излучение и радиоволны в частности, выделяемое тепло – все это может быть использовано. К подобным атакам не стоит относиться беспечно – они осуществимы на практике. Атаки по сторонним каналам уже применялись для извлечения ключей шифрования, подделки цифровых подписей и несанкционированного доступа к информации.

Итоги

- Для проверки подлинности данных используется хеширование с ключом.
- Если человеку нужно держать ключ в памяти, используйте кодовую фразу.
- Если не нужно, то используйте случайное число.
- Положитесь на HMAC-функции для вычисления хеша с ключом при решении типичных задач.
- В Python встроена поддержка HMAC-функций через модуль `hmac`.
- Во избежание атак по времени сравнивайте хеши за одинаковое время.