3.1.1 Генерация ключа

Любой секретный ключ должно быть сложно угадать. В этом разделе сравниваются две его разновидности: случайное число и кодовая фраза. В частности, описано, как их генерировать и когда какой тип ключа использовать.

Случайные числа

Для генерации случайных чисел не требуются сторонние библиотеки, в самом Python достаточно способов их создавать. Но не все они подходят для целей безопасности. Разработчики, как правило, используют функцию `os.urandom` в качестве криптобезопасного источника случайных чисел. В качестве аргумента она принимает целое число `size` и возвращает соответствующее количество случайных байтов. Источник этих байтов – сама операционная система. На UNIX-подобных системах это `/dev/urandom`, в Windows это `CryptGenRandom`.

```
>>> import os
>>>
>>> os.urandom(16)
b'\x07;`\xa3\xd1=wI\x95\xf2\x08\xde\x19\xd9\x94^'
```

В Python 3.6 специально для генерации криптобезопасных случайных чисел был добавлен модуль `secrets`. `os.urandom` замечательно справляется со своей задачей, однако в этой книге для генерации случайных чисел используется `secrets`. Он содержит в себе три удобные для этого функции. Все они принимают целое число как аргумент и возвращают число случайное. Эти числа могут быть в виде массива байтов, шестнадцатеричного текста либо текста для вставки в URL. У всех этих функций префикс `token_`:

```
>>> from secrets import token_bytes, token_hex, token_urlsafe
>>>
>>> token_bytes(16)                                          | Генерация
b'\x1d\x7f\x12\xadsu\x8a\x95[\xe6\x1b|\xc0\xaeM\x91'         | 16 случайных байт
>>>
>>> token_hex(16)                        | Генерация 16 случайных байт
'87983b1f3dcc18080f21dc0fd97a65b3'       | в виде шестнадцатеричного текста
>>>
>>> token_urlsafe(16)      | Генерация 16 случайных байт
'Z_HIRhlJBMPh0GYRcbICIg'   | для вставки в URL
```

Введите эту команду, чтобы сгенерировать 16 случайных байт на своем компьютере. Готов поспорить, наши числа будут отличаться:

```
$ python -c 'import secrets; print(secrets.token_hex(16))'
3d2486d1073fa1dcfde4b3df7989da55
```

Модуль `random` – это третий способ получения случайных чисел. Большинство функций из этого модуля возвращают небезопасные значения. В документации этого модуля сказано однозначно: «не должен использоваться для целей обеспечения безопасности» (https://docs.python.org/3/library/random.html). В документации же модуля `secrets` заявлено, что его «следует предпочитать генератору псевдослучайных чисел из модуля `random`» (https://docs.python.org/3/library/secrets.html).

> **ВНИМАНИЕ!** Ни при каких обстоятельствах не используйте модуль `random` для оборонных и криптографических задач. Он отлично подходит для прикладных целей, но не для построения защиты.

Кодовые фразы

Кодовой либо *парольной фразой* является набор случайных слов вместо чисел. В листинге 3.1 показано, как составить кодовую фразу из четырех слов с помощью модуля `secrets`. Слова берутся из файла, он служит словарем.

Вначале словарь загружается в оперативную память. Он обычно идет «из коробки» в UNIX-подобных системах. Если у вас другая ОС, подобный файл можно найти в интернете (https://raw.githubusercontent.com/eneko/data-repository/master/data/words.txt). С по-

мощью функции `secrets.choice` из перечня выбирается случайное слово. Она в целом применяется для того, чтобы вернуть случайный элемент из переданной последовательности.

Листинг 3.1 Генерация кодовой фразы из четырех слов

```
from pathlib import Path
import secrets

words = Path('/usr/share/dict/words').read_text().splitlines()

passphrase = ' '.join(secrets.choice(words) for i in range(4))

print(passphrase)
```

Загружаем словарь в память

Выбираем четыре случайных слова

Подобные словари используются и злоумышленниками для атак перебором. Поэтому брать из них слова, чтобы создать пароль, – неочевидная идея. Соль кроется в длине кодовой фразы. Например, фраза `whereat isostatic custom insupportableness` весит 42 байта. По данным www.useapassphrase.com, на взлом этой фразы уйдет 163 274 072 817 384 столетия. Успешная атака «грубой силой» на такой длинный ключ недостижима. Длина решает.

Случайное число и кодовая фраза по сути своей отвечают основным требованиям к ключу. Трудно подобрать что одно, что другое. Предложения из случайных слов в качестве ключа своим существованием обязаны несовершенству человеческой памяти.

> **СОВЕТ** Случайное число сложно запомнить, парольную фразу просто запомнить. Исходя из этого, и стоит выбирать, когда использовать число, а когда прибегнуть к кодовой фразе.

Случайные числа пригодятся тогда, когда человеку не нужно помнить ключ больше пары минут. Например, если таковы требования безопасности либо если это просто одноразовый код. Это может быть токен многофакторной аутентификации, а может быть временный код для сброса пароля. Как мы уже видели, методы `secrets.token_bytes`, `secrets.token_hex` и `secrets.token_urlsafe` все начинаются с `token_`. Приставка красноречиво сообщает, для чего стоит их применять.

Польза кодовых фраз проявляется, когда человеку нужно запомнить ключ надолго. Фраза будет хорошим выбором для пароля от сайта или от консоли SSH. Увы, большинство пользователей сети не применяют фразы в качестве паролей. Многие сайты и не предлагают посетителям использовать их.

Важно отдавать себе отчет, что эти числа и фразы могут не только решать проблемы – создавать их они тоже умеют, если между этими двумя вариантами был сделан неправильный выбор. Вот два примера, когда человек вынужден запоминать случайное число. В первом случае оно нужно для доступа к информации. Забыл число – доступ

потерял. Во втором случае сисадмин на память не надеется – у него заботливо приклеен листочек с числом прямо на монитор. Но останется ли оно таким образом в секрете?

Допустим, в качестве одноразового ключа использована кодовая фраза. Вот вам приходит письмо для сброса пароля или код многофакторной авторизации с подобной фразой. Если кто-то заглянет вам через плечо, фразу запомнить будет проще, чем случайное число достаточной длины.

> **ПРИМЕЧАНИЕ** Для упрощения ключи далее будут указаны прямо в примерах кода. На боевой системе все они должны храниться в службе управления ключами, а не среди исходного кода. Неплохим выбором могут быть Amazon's AWS Key Management Service (https://aws.amazon.com/kms/) и Google's Cloud Key Management Service (https://cloud.google.com/security-key-management).

Теперь вам известно, как сгенерировать безопасный ключ, а также когда использовать случайное число, а когда парольную фразу. Эти знания на протяжении книги пригодятся вам еще не раз, и впервые буквально в следующем разделе.

3.1.2 Хеширование с ключом

Некоторым хеш-функциям можно передать необязательный аргумент – ключ. Он, как видно на рис. 3.1, подается на вход функции вместе с сообщением. Как и обычно, на выходе нас ждет посчитанный хеш.

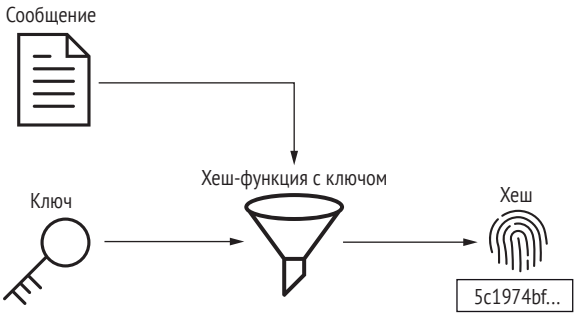

Рис. 3.1 Хеш-функция с ключом принимает два аргумента

Хеш зависит от переданного ключа. Хеш одного и того же сообщения будет разным, если передан разный секрет. Хеши будут идентичны, если пары ключ–сообщение будут совпадать. Вот пример хеширования с ключом функцией BLAKE2, которой при необходимости можно передать ключ:

```
>>> from hashlib import blake2b
>>>
>>> m = b'same message'
>>> x = b'key x'  <-- Первый ключ
>>> y = b'key y'  <-- Первый ключ
>>>
>>> blake2b(m, key=x).digest() == blake2b(m, key=x).digest()
True
>>> blake2b(m, key=x).digest() == blake2b(m, key=y).digest()
False
```

Разные ключи, разные хеши

Тот же ключ, тот же хеш

Хеширование с ключом позволит Алисе добавить уровень обороны от Мэллори в системе документооборота. Алиса сможет сохранять документы на пару с хешем, который технически может подсчитать только она. У Мэллори больше не получится изменить документ и просто пересчитать его хеш. Без ключа у нее не выйдет то же самое значение, которое вышло бы у законной владелицы. В итоге документ просто не пройдет проверку. Теперь система документооборота защищает от случайного повреждения данных и злонамеренных изменений.

Листинг 3.2 Улучшенная Алисина защита

```
import hashlib
from pathlib import Path

def store(path, data, key):
    data_path = Path(path)
    hash_path = data_path.with_suffix('.hash')

    hash_value = hashlib.blake2b(data, key=key).hexdigest()

    with data_path.open(mode='x'), hash_path.open(mode='x'):
        data_path.write_bytes(data)
        hash_path.write_text(hash_value)

def is_modified(path, key):
    data_path = Path(path)
    hash_path = data_path.with_suffix('.hash')

    data = data_path.read_bytes()
    original_hash_value = hash_path.read_text()

    hash_value = hashlib.blake2b(data, key=key).hexdigest()

    return original_hash_value != hash_value
```

Подсчет хеша документа с ключом

Запись документа и хеша по разным файлам

Чтение документа и хеша из хранилища

Пересчет хеша с ключом

Сравнение хеша текущего содержимого документа со значением с диска

Рядовые хеш-функции вроде SHA-256 не позволяют указать ключ – и таких функций большинство. Именно поэтому команда одаренных ребят разработала механизм проверки подлинности со-

общений, использующий хеш-функции без ключа. Разговор в следующем разделе пойдет о функциях, реализующих этот алгоритм.