Лабораторная работа №2

Теоретические и учебно-методические материалы

На основание ниже приведенного материала используя для хеширования информации использовать свою фамилию.

Сделать скрины выполнения кода.

2.1 Криптографическое хеширование в Python

Модуль hashlib содержит в себе именованные конструкторы для каждой хеш-функции из hashlib.algorithms_guaranteed. Также все хеш-функции доступны через универсальный конструктор new. В качестве аргумента он принимает любую строку из algorithms_guaranteed. Именованные конструкторы быстрее, чем универсальный, поэтому стоит использовать их. Код ниже показывает, как создать экземпляр SHA-256 с помощью обоих конструкторов:

```
import hashlib

named = hashlib.sha256()
generic = hashlib.new('sha256')
```

Именованный конструктор

Универсальный конструктор

Экземпляру хеш-функции может быть задано сообщение с самого начала. Код ниже изначально помещает сообщение в экземпляр функции SHA-256. В отличие от встроенной функции hash, функции модуля hashlib требуют, чтобы сообщение было байтовой строкой:

```
>>> from hashlib import sha256
>>>
>>> message = b'message'
>>> hash_function = sha256(message)
```

Независимо от способа создания, любой экземпляр функции имеет идентичный внешний интерфейс (API). Открытые методы экземпляра SHA-256 не отличаются от открытых методов экземпляра MD5. Методы digest и hexdigest возвращают хеш как байтовую строку в первом случае и как шестнадцатеричный текст во втором:

```
>>> hash_function.digest()
b'\xabS\n\x13\xe4Y\x14\x98+y\xf9\xb7\xe3\xfb\xa9\x94\xcf\xd1\xf3\xfb"\xf7\x
1c\xea\x1a\xfb\xf0+F\x0cm\x1d'
```

>>>Хеш в виде байтовой строки

```
>>> hash_function.hexdigest()
'ab530a13e45914982b79f9b7e3fba994cfd1f3fb22f71cea1afbf02b460c6d1d'
```

Хеш в виде обыкновенной строки

Следующий пример демонстрирует коллизию MD5-хешей. Для этого используется метод digest. У обоих сообщений различаются лишь несколько символов (выделены полужирным):

```
>>> from hashlib import md5
>>> >>> x = bytearray.fromhex( ...
'd131dd02c5e6eec4693d9a0698aff95c2fcab5812467eab4004583eb8fb7f8955ad340609
f4b30283e4888325171415a085125e8f7cdc99fd91dbdf280373c5bd8823e3156348f5bae6da
cd436c919c6dd53e2b487da03fd02396306d248cda0e99f33420f577ee8ce54b67080a80d1e 44
```



```
c69821bcb6a8839396f9652b6ff72a70') >>>
>>> y = bytearray.fromhex( ...
'd131dd02c5e6eec4693d9a0698aff95c2fcab5012467eab4004583eb8fb7f8955ad340609
f4b30283e4888325f1415a085125e8f7cdc99fd91dbd7280373c5bd8823e3156348f5bae6da
cd436c919c6dd53e23487da03fd02396306d248cda0e99f33420f577ee8ce54b6708028 0d1e
c69821bcb6a8839396f965ab6ff72a70') >>>
>>> x == y
False
>>>
>>> md5(x).digest() == md5(y).digest()
True
```

Сообщения различаются

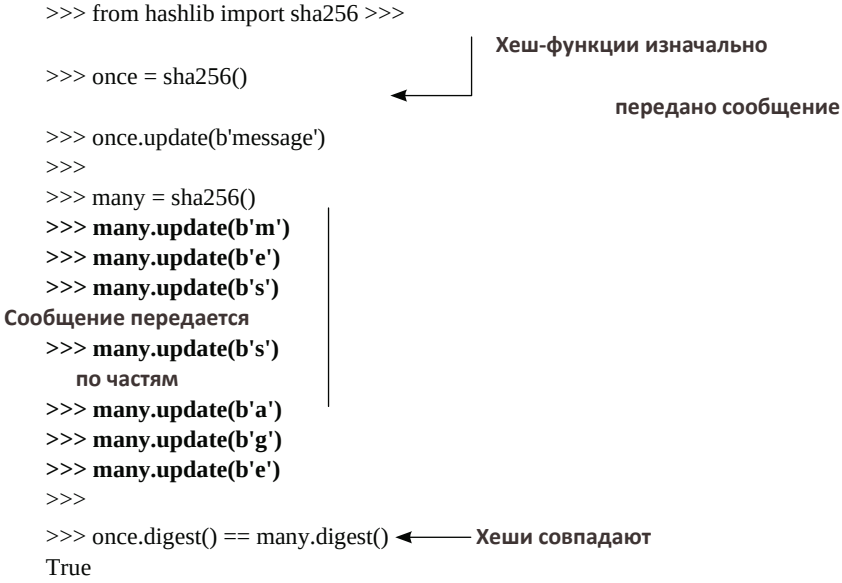
Хеши совпадают, коллизия

Хеш сообщения также может быть подсчитан с помощью метода update. В примере ниже отмечен его вызов. Это может пригодиться, когда хеш-функцию нужно создать в одном месте, а использовать в другом. От способа передачи сообщения хеш не изменяется:

```
>>> message = b'message'
>>>Экземпляр создан без сообщения
>>> hash_function = hashlib.sha256()
  >>> hash_function.update(message)
>>>
>>> hash_function.digest() == hashlib.sha256(message).digest()Хеши
True совпадают
```

Сообщение передано через метод update

Сообщение может быть разбито на части. При этом методу update можно передавать последующие части одна за одной, и хеш всего сообщения будет пересчитан. При этом переданные данные не копируются, и на них не сохраняется ссылка. Этот прием может пригодиться, когда объемное сообщение не может быть загружено в память разом. Хеши между одинаковыми сообщениями, переданными целиком и по частям, не различаются.



Функции контрольного суммирования

Свойство digest_size хранит длину хеша в байтах. Напомним, что SHA-256 – это 256-битная хеш-функция, как и гласит имя:

```
>>> hash_function = hashlib.sha256(b'message')
>>> hash_function.digest_size
32
>>> len(hash_function.digest()) * 8
256
```

Криптографические хеш-функции являются по определению детерминированными. Они само собой работают одинаково на любой платформе. Входные данные из примеров в этой главе дадут идентичный результат на любом компьютере на любом языке программирования через любой API. Следующие две команды демонстрируют это правило с помощью Python и Ruby. Если две реализации одной и той же криптографической функции выдают разный хеш, то это значит, что как минимум с одной из них что-то серьезно не так:

```
$ python -c 'import hashlib; print(hashlib.sha256(b"m").hexdigest())'
62c66a7a5dd70c3146618063c344e531e6d4b59e379808443ce962b3abd63c5a
$ ruby -e 'require "digest"; puts Digest::SHA256.hexdigest "m"'
62c66a7a5dd70c3146618063c344e531e6d4b59e379808443ce962b3abd63c5a
```

Встроенная же в Python функция hash, напротив, по умолчанию является детерминированной только в рамках отдельного процесса Python. Приведенные две команды показывают два различных вычисленных хеша в двух *различных* процессах языка Python:

```
$ python -c 'print(hash("message"))'
```

Одинаковое

```
8865927434942197212 сообщение
$ python -c 'print(hash("message"))'
3834503375419022338 Разные хеши
```

ВНИМАНИЕ! Встроенная функция hash ни при каких обстоя тельствах не должна быть использована для криптографических расчетов. Она отрабатывает очень быстро, но ее сопротивление поиску коллизий несопоставимо с SHA-256.

Возможно, вы уже задались вопросом: так разве хеши – это не обыкновенные контрольные суммы? Ответ – нет, и следующий раздел рассказывает почему.

2.2 Функции контрольного суммирования

Между хеш-функциями и функциями контрольного суммирования есть кое-что общее. *Хеш-функции* принимают данные и выдают хеш, функции контрольного суммирования принимают данные и выдают контрольную сумму. Что хеш, что контрольная сумма – числа. Эти числа нужны для выявления непредусмотренного изменения данных, обычно при их хранении и передаче.

Python имеет встроенную поддержку функций контрольного суммирования, как то функция подсчета циклического избыточного кода (cyclic redundancy check – CRC) и Adler-32. Они находятся в модуле zlib. Следующий пример показывает типичный случай применения CRC. Сначала блок повторяющихся данных были сжат, а затем распакован. Контрольная сумма блока данных была посчитана до преобразования и после него, как выделено в листинге. В конце происходит обнаружение ошибок, для чего контрольные суммы до и после сравниваются между собой:

```
>>> import zlib
>>>
>>> message = b'this is repetitious' * 42 Вычисление контрольной
   >>> checksum = zlib.crc32(message)          суммы сообщения
>>>
>>> compressed = zlib.compress(message) Сжатие и распаковка
>>> decompressed = zlib.decompress(compressed) сообщения
>>>
   >>> zlib.crc32(decompressed) == checksum   При сравнении контрольных сумм
True                                          ошибок не обнаружено
```

Криптографические хеш-функции и функции контрольного суммирования похожи между собой, но, несмотря на это, путать их не стоит. Дело в том, что контрольное суммирование происходит быст рее, но в ущерб криптографической стойкости. Иначе говоря, для криптографической хеш-функции сложно найти коллизию, но и подсчет занимает больше ресурсов. Контрольная же сумма высчитывается быстро, но найти коллизию среди этих сумм несложно. Например,

CRC и Adler-32 куда быстрее, чем SHA-256, но похвастать достаточным сопротивлением поиску коллизий они не могут, их несчетное множество. Буквально один пример:

```
>>> zlib.crc32(b'gnu')
1774765869
>>> zlib.crc32(b'codding')
1774765869
```

Попадись такая коллизия в работе функции SHA-256, ее огласка бы имела эффект разорвавшейся бомбы. Функции контрольного суммирования лишь с натяжкой можно использовать для проверки целостности данных. Скорее, такие функции применяются для *обнаружения ошибок*, но не для проверки данных на целостность.

ВНИМАНИЕ! Функции контрольного суммирования ни за что не должны быть использованы для целей обеспечения безопасности. В то же время криптографические хеш-функции могут быть использованы вместо функций контрольного суммирования, но ценой значительного роста требуемой вычислительной мощности.

Этот раздел рассказал о том, что для криптографического хеширования надо использовать модуль haslib, а не zlib. Следующая глава продолжит рассказ о подсчете хешей. В ней рассказывается, как использовать модуль hmac для хеширования с ключом. Такие хеши применяются для проверки подлинности данных.

Итоги

- Хеш-функции детерминированно превращают сообщения в хеши одинаковой длины.
- Для проверки целостности данных нужно использовать криптографические хеш-функции.
- По умолчанию для криптографии стоит использовать SHA-256.
- Использование MD5 либо SHA-1 для целей защиты небезопасно.
- Для криптохеширования в Python нужно применять модуль hashlib.
- Функции контрольного суммирования не подходят для криптографического хеширования.
- Алиса, Боб и Чарли – мирные жители.
- Ева и Мэллори – мафия.