Реферат

З програмування

«Криптографічні модулі в Python 3.10.0»

Автор: Венгер Андрій Олександрович

Студент 1 курсу Механіко-Математичного факультету

Київського національного університету імені Тараса Шевченка

2021

Зміст

[Модуль hashlib 2](#_Toc87898480)

[Що ж таке хеш-функція? 2](#_Toc87898481)

[Можливості модуля 2](#_Toc87898482)

[Застосування імплементації SHA-256 3](#_Toc87898483)

[Застосування імплементації BLAKE2 4](#_Toc87898484)

[Модуль hmac 5](#_Toc87898485)

[Застосування імплементації HMAC 5](#_Toc87898486)

[Модуль secrets 6](#_Toc87898487)

[Можливості модуля 6](#_Toc87898488)

[Генерування паролю 7](#_Toc87898489)

[Генерування токену 8](#_Toc87898490)

[Висновки 10](#_Toc87898491)

[Джерела інформації 10](#_Toc87898492)

# Модуль hashlib

Модуль hashlib надає змогу користуватися уніфікованим інтерфейсом, для якого існують численні імплементації відомих хеш-функцій (хеш-алгоритми). Перед тим, як ознайомитись з модулем, необхідно зрозуміти базову концепцію роботи хеш-функцій і усвідомити чому взагалі існує така багата кількість різноманітних алгоритмів для вирішення питання здобуття хешу з набору даних.

## Що ж таке хеш-функція?

Взагалі, хеш-функція — це функція, яка ставить у відповідність до набору даних довільної довжини якесь значення фіксованого розміру. Це значення, яке є результатом роботи функції, називають хешем або дайджестом.

Diagram

Description automatically generated

На діаграмі червоним кольором позначена ситуація, яка називається «колізія», тобто коли для двох різних наборів вхідних даних хеш-функція на виході дає однаковий дайджест. Причина цього прихована в означенні хеш-функції. Очевидно, що множина можливих аргументів є незліченою, коли множина можливих значень функції має фіксований розмір. Процент колізій є одним із методів оцінки хеш-функцій (менше значить краще).

Хеш-функція дає однакові результати для однакових аргументів, але навіть незначна зміна аргументу тягне за собою значну зміну в дайджесті. Також хеш-функція повинна бути нереверсивною, тобто щоб дізнатись аргумент за відомим хешем потрібно витратити неймовірно багато ресурсів (головне часу).

Хешування знаходь безліч застосувань у наш час. Структури даних (хеш-таблиці, дерева Меркла та ін.), криптографія, алгоритми пошуку даних та багато інших важливих частин інформатики та математики потребують застосування хешування.

## Можливості модуля

Модуль hashlib впроваджує імплементації алгоритмів хешування SHA1, SHA224, SHA256, SHA384 та SHA512 відповідно до FIPS 180-2 (Федеральні стандарти обробки інформації), а також RSA MD5 відповідно до RFC 1321. Насправді наявність зазначених алгоритмів забезпечується бібліотекою OpenSSL, тому це треба брати до уваги при розробці програмного забезпечення на тій чи іншій платформі.

Існує можливість перевірити доступні алгоритми хешування для поточної машини:

Text

Description automatically generated

Для кожного із зазначених алгоритмів в модулі є відповідний конструктор, який створює об’єкт з уніфікованим інтерфейсом. Концепція подальшої роботи є доволі простою. В об’єкт можна «додавати» порції даних за допомогою відповідного метода оновлення вмісту. Коли потрібно здобути хеш з даних, треба скористатися одним з деяких методів отримання дайджесту. Концептуально це є необхідний мінімум для користування модулем hashlib.

## Застосування імплементації SHA-256

Text

Description automatically generated

Перш за все, необхідно імпортувати відповідний модуль. На третьому рядку я створив новий об’єкт для роботи з алгоритмом хешування SHA-256, викликавши відповідний конструктор. Четвертий рядок ілюструє метод для оновлення вмісту об’єкту для хешування. Треба зазначити, що метод *update()* приймає в якості аргументу набір байтів. Але найчастіше треба хешувати текстові послідовності. Тому я застосував префікс *b* до рядка-аргументу метода оновлення вмісту. Це еквівалентно такому запису:



На п’ятому рядку друкую в консолі хеш, при чому користуюсь методом, який повертає дайджест у шістнадцятковому форматі. Альтернативою слугує здобуття набору байтів, як зазначено тут:



Метод *update()* не видаляє попереднє значення аргументу, а конкатенує аргументи, що продемонстровано далі:

Text

Description automatically generated

Дайджест є таким самим, як і у прикладі с одноразовим оновленням вмісту об’єкту.

Також є можливість дізнатися довжину дайджесту в байтах:

Text

Description automatically generated

## Застосування імплементації BLAKE2

Модуль hashlib містить велику кількість імплементацій хеш-алгоритмів і всі вони відрізняються за призначенням, крипостійкістю і ще низкою характерних показників. Деякі з алгоритмів мають багато налаштувань, які дозволяють контролювати їх роботу. Наведу приклад хеш-функції BLAKE2.

Text

Description automatically generated

Не буду детально пояснювати кожен рядок, але загострю увагу на ключових моментах. По-перше, blake2b є оптимізованим алгоритмом для 64-бітних платформ і може мати значення довжиною від 1 до 64 байтів. У четвертому рядку я створив об’єкт з довжиною дайджесту 10 байтів, у восьмому – 15 байтів. Ключовим аспектом тут є те, що дайджести для одного і того ж аргументу ніяк не зв’язані (можна було думати, що 15-байтовий хеш буде починатися з десяти байтів, які і є 10-байтовим хешем, але це не так).

A screenshot of a computer

Description automatically generated with medium confidence

Цей приклад демонструє застосування хешування з так званою сіллю, яка є ще одним з методів захисту від здобуття аргументу хеш-функції, коли дайджест є відомим. Такий метод можна застосовувати для зберігання паролей в базі даних, щоб нівелювати розшифровку даних за допомогою райдужних таблиць. Очевидно, що сіль потрібно генерувати рандомно, а не хардкодити, як це зроблено в прикладі, але я хотів спросити ситуацію. Як бачимо, дайджести знов різні для однакових аргументів хеш-функції, але цього разу це пояснюється додаванням тієї самої солі в процесі хешування (можна міркувати про неї, як про додатковий набір байтів, що долучається до аргументу, а значить змінює результат хеш-функції).

# Модуль hmac

Доволі часто потрібно передавати інформацію через публічне середовище. Це тягне за собою низку проблем, з-поміж яких виділяється питання підробки даних (атака «людина посередні»). Здавалося б, можна передавати хеш даних разом із повідомленням, але здобути дайджест з даних може хто завгодно, тобто це не вирішує проблему аутентифікації. Саме для цього був створений механізм HMAC, який дозволяє перевіряти не тільки цілісність даних, але й гарантувати аутентифікованість відправника. HMAC є абстракцією, що застосовує звичайні хеш-алгоритмам, тобто можливо застосовувати існуючі функції, що має гарний вплив на зворотну сумісність.

Відповідний модуль hmac надає можливість користуватися імплементацією алгоритму, зазначеного у стандарті RFC 2104.

## Застосування імплементації HMAC

Text

Description automatically generated

Розглянемо ситуацію. Ми передаємо данні через відкритий канал, але застосовуємо HMAC на основі SHA-256. Якщо зловмисник перехопить наші данні й якщо йому відом алгоритм хешування даних, то він легко може змінити інформацію й здобути з неї дайджест. Тоді на іншому боці неможливо було б виявити втручання. Але так як ми користуємось приватним ключем, який зловмисник не знає, то коли втручання буде помічено, тобто ми зрозуміємо, що хеш був створений не аутентифікованою стороною (не той приватний ключ).

Звісно, необхідно імпортувати відповідні модулі до проекту. З модуля hashlib візьмемо імплементацію алгоритму SHA-256. На третьому рядку створена змінна з приватним ключем, який треба зашейріти з другою стороною взаємодії (наприклад, застосовуючи модифіковані версії протоколу Діффі-Хеллмана). Далі створено повідомлення, яке буде передано через публічний канал. Рядки 5-7 демонструють здобуття дайджесту повідомлення за допомогою HMAC на основі SHA-256. Рядки 9-11 емулюють спробу зловмисника підробити хеш повідомлення. Як бачимо, дайджест не співпадає з відповідним результатом роботи HMAС-функції, попри те, що зловмисник застосував той самий алгоритм SHA-256 з тим самим аргументом. Тобто друга сторона зрозуміє, що відбулось втручання.

# Модуль secrets

Модуль secrets надає змогу генерувати випадкові числа, при чому гарантує криптографічну стійкість отриманих результатів. Де можуть стати в нагоді рандомні числа? Ціла низка питань – паролі, токени аутентифікації, приватні ключі та ін. Хтось може буди знайомим з модулем random, але він генерує псевдо-випадкові числа, що не мають криптографічної стійкості й захищеності.

## Можливості модуля

Text

Description automatically generated

Доволі простий приклад з отриманням рандомного цілого числа с пів інтервалу [0;10).

Text

Description automatically generated

Можна отримати необхідну кількість рандомних бітів. Тут я генерую 10 випадкових бітів, але відповідна функція *randbits()* повертає ціле число, в даному випадку в діапазоні [0;1024]. Результат першого другу є те саме ціле число, результат другого – це ж саме число у бінарному форматі.

Text

Description automatically generated

Існує метод для вибору випадкового елементу з набору даних. Він має назву *choice()* і в якості аргументу приймає набір даних, з якого треба вибрати рандомний елемент. В даному прикладі роблю 5 ітерацій циклу, на кожній з яких друкую вибраний елемент. Залишаю позицію курсора в консолі на тому самому рядку, замінюючи кінець результату другу з ‘\n’ на ‘ ‘, для спрощення ілюстрації друку.

Хочу розібрати більш реальні кейси застосування модуля secrets.

## Генерування паролю

Дуже часто треба створювати стійкі паролі, подальші приклади ілюструють можливі вирішення цієї проблеми.

Text

Description automatically generated

Імпортую додатково модуль string для зручного здобуття набору символів, з якого будуть братися відповідні елементи для генерування паролю. Протягом десяти ітерацій циклу конкатеную змінну паролю з рандомним символом, що обирається за допомогою відомого метода *choice()* з набору даних, якому належать літери латинського алфавіту (тому що система кодів ASCII підтримує виключно їх), а також цифри. В результаті отримуємо десятизнаковий пароль.

Можна використати скорочений запис:

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

В цьому прикладі генерую пароль довжиною 10 символів, який містить щонайменше 2 цифри, одну велику літеру та один знак пунктуації. Застосовую нескінченний цикл *while*, на кожній ітерації генерую пароль з набору літер, цифр та знаків пунктуації. Перевіряю чи він відповідає зазначеним вимогам. Якщо ні, то генерую наступний, якщо так, то закінчую цикл та друкую результат в терміналі.

## Генерування токену

Досить часто трапляється необхідність генерувати рандомні токени, тобто рандомні рядки, які можуть застосовуватись у таких випадках, як: аутентифікація (JWT, cookies та ін.), зміна паролю, відновлення сесії та ін. Модуль secrets має декілька методів для вирішення зазначених питань.

Text

Description automatically generated

Рядок 3 демонструє генерування рандомного байтового токену довжиною 10 байт. Рядок 5 генерує токен тієї ж довжини, але у шістнадцятковому форматі. Наприкінці, рядок 7 створює токен розміром 10 байтів, але закодований у форматі Base64, який можна безпечно застосовувати в частинах URL, бо він буде містити тільки латинські літери (великі й маленькі), цифри та символи *+* і */*.

Взагалі існує безліч застосувань для токенів, наведу одне з найпопулярніших:

Text

Description automatically generated

Приклад ілюструє створення токену для відновлення паролю. Генерую URL-safe токен, який потрібно долучити до запису юзера в базі даних, щоб потім змапити запит на відновлення паролю з конкретним користувачем. Створюю лінк на відновлення паролю, який потрібно зашейрити з юзером. Звісно, треба обмежувати термін дії токена та видаляти його після активації, але це вже тема іншої бесіди про сервіси аутентифікації. До речі, я не вказував довжину токену, тому був застосований дефолтний параметр, що сьогодні становить 32 байта (зростає з часом, бо обчислювальні потужності не стоять на місці).

# Висновки

У цьому рефераті я спробував показати приклади застосування основних криптографічних модулів інтерпретатору Python. Модуль hashlib містить численні імплементації хеш-алгоритмів на підставі бібліотеки OpenSSL. Модуль hmac надає змогу перевіряти цілісність інформації, переданої через незахищений канал. Модуль secrets має численні методи для генерування випадкових значень, які є криптографічно стійкими. Взагалі, тема криптографії заслуговує цілих наукових трактатів, висвітленні модулі реалізують неможливо маленьку частину цього цікавого питання, а приклади й того менше. Потрібно усвідомити, що сьогодні системи стають все більш складними, а стандарти безпеки більш суворими. Без початкових знань криптографії та безпеки в мережі неможливо уявити будування більш-менш успішного проекту. Мова Python та її інтерпретатор є чудовими інструментами, які мають багатий криптографічний функціонал, який, інколи, представлений високими абстракціями. І все ж таки необхідно розуміти на чому ґрунтуються ті чи інші імплементації, чому деякі алгоритми є кращими за інші в специфічних умовах, усвідомлювати які загрози існують і знати як їх уникнути.

# Джерела інформації

1. [Модуль hashlib](https://docs.python.org/3/library/hashlib.html)
2. [Модуль hmac](https://docs.python.org/3/library/hmac.html)
3. [Модуль secrets](https://docs.python.org/3/library/secrets.html)
4. [Хеш-функція (англійська)](https://en.wikipedia.org/wiki/Hash_function)
5. [Хеш-функція (українська)](https://uk.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F)
6. [HMAC (англійська)](https://en.wikipedia.org/wiki/HMAC)
7. [HMAC (українська)](https://uk.wikipedia.org/wiki/HMAC)
8. [Сіль в криптографії](https://en.wikipedia.org/wiki/Salt_(cryptography))