

# Базовая безопасность сервисов: защита Python- приложений



# Почему безопасность важна?

1

## Защита данных

Веб-сервисы ежедневно обрабатывают огромные объемы конфиденциальных данных, включая личную информацию пользователей, финансовые данные и пароли.

Несанкционированный доступ к этим данным может привести к серьезным последствиям.

2

## Предотвращение утечек

Ошибки в безопасности могут стать причиной утечек данных, что не только наносит ущерб репутации компании, но и ведет к значительным финансовым потерям и юридическим проблемам.

3

## Доверие пользователей

Потеря доверия пользователей из-за взломов и утечек данных может быть катастрофической. Восстановление репутации – долгий и дорогостоящий процесс, поэтому превентивные меры крайне важны.

4

## Цель занятия

Сегодня мы познакомимся с ключевыми уязвимостями, такими как SQL-инъекции, а также с фундаментальными методами защиты, включая использование JWT-токенов, надежное хеширование паролей и применение "соли".

# SQL-инъекции: что это и почему опасно?

SQL-инъекция – это тип атаки, при которой злоумышленник вставляет вредоносный SQL-код в поля ввода приложения (например, формы логина, поля поиска). Этот код затем выполняется базой данных, позволяя злоумышленнику получать несанкционированный доступ, изменять или удалять данные.

Последствия могут быть разрушительными: от раскрытия конфиденциальной информации до полного компрометирования сервера базы данных.

Пример уязвимого кода на Python:

```
import sqlite3

def get_user_data(username):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    # Уязвимый запрос!
    query = f"SELECT * FROM users WHERE username = '{username}'"
    cursor.execute(query)
    data = cursor.fetchone()
    conn.close()
    return data
```

# SQL-инъекции: уязвимый код

## Как это работает?

Представим, что злоумышленник вводит в поле username следующее значение: ' OR '1'='1

Тогда наш уязвимый SQL-запрос превратится в:

```
SELECT * FROM users WHERE username = '' OR '1'='1'
```

Поскольку условие '1'='1' всегда истинно, запрос вернет все строки из таблицы users, что является серьезной утечкой данных.

## Более продвинутые атаки

Злоумышленники могут также использовать SQL-инъекции для:

- Добавления, изменения или удаления данных.
- Выполнения команд операционной системы (если СУБД это позволяет).
- Получения информации о структуре базы данных.

# SQL-инъекции: безопасный код

## Параметризованные запросы

Это основной и наиболее эффективный метод защиты от SQL-инъекций. Вместо того чтобы вставлять пользовательские данные напрямую в строку запроса, вы используете плейсхолдеры (заполнители), а затем передаете данные отдельно.

```
# Безопасный параметризованный запрос
query = "SELECT * FROM users WHERE username = ?"
cursor.execute(query, (username,))
data = cursor.fetchone()
conn.close()
```

В этом случае база данных отличает код запроса от вводимых данных, предотвращая выполнение вредоносных команд.

## ORM (Object-Relational Mapping)

ORM-фреймворки, такие как SQLAlchemy для Python, абстрагируют взаимодействие с базой данных, генерируя безопасные запросы автоматически.

```
engine = create_engine('sqlite:///users.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)

def get_user_data_orm(username):
    session = Session()
    user = session.query(User).filter_by(username=username).first()
    session.close()
    return user
```

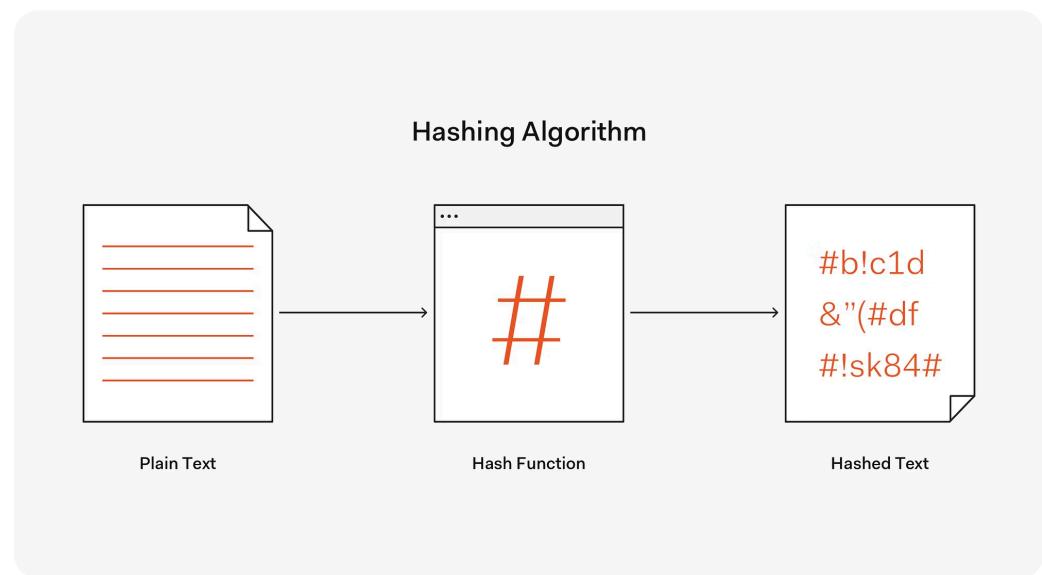
Использование ORM существенно снижает риск SQL-инъекций, так как он берет на себя ответственность за корректное экранирование данных.

# Хэширование паролей: что это и зачем?

Хэширование паролей – это процесс преобразования пароля в фиксированную строку символов (хеш), которая является односторонней. Это означает, что из хеша невозможно восстановить исходный пароль.

## Зачем хешировать?

- **Защита от утечек:** Если база данных будет скомпрометирована, злоумышленники получат только хеши, а не сами пароли.
- **Соответствие стандартам:** Многие регуляторные нормы требуют безопасного хранения паролей.



# Соль для паролей



"Соль" (salt) – это случайная строка данных, которая добавляется к паролю перед его хешированием. Каждому паролю должна соответствовать уникальная соль.

## Зачем нужна соль?

- **Защита от радужных таблиц:** это предварительно вычисленные хеши для распространенных паролей. Соль делает каждую хеш-сумму уникальной, даже если два пользователя имеют одинаковый пароль, что делает радужные таблицы бесполезными.
- **Защита от брутфорс-атак на несколько паролей одновременно:** Злоумышленник не может хешировать один и тот же пароль один раз и проверять его против всех хешей в базе данных. Ему придется хешировать каждый пароль с каждой уникальной солью, что значительно замедляет атаки.

# Реализация на Python

1

## Хеширование пароля с солью

```
from passlib.hash import bcrypt

def hash_password(password):
    # bcrypt автоматически генерирует соль и хеширует
    # пароль
    hashed_password = bcrypt.hash(password)
    return hashed_password

my_password = "securepassword123"
hashed = hash_password(my_password)
print(f"Хешированный пароль: {hashed}")
```

2

## Проверка пароля

```
def verify_password(password, hashed_password):
    return bcrypt.verify(password, hashed_password)

# Проверяем правильный пароль
is_correct = verify_password(my_password, hashed)
print(f"Пароль верен: {is_correct}")

# Проверяем неверный пароль
is_incorrect = verify_password("wrongpassword", hashed)
print(f"Пароль неверен: {is_incorrect}")
```

Использование `bcrypt` гарантирует, что соль будет уникальной для каждого пароля и интегрированной в сам хеш, что значительно повышает безопасность.

# JWT-токен: что это и зачем?

JSON Web Token (JWT) – это компактный, URL-безопасный способ представления информации между двумя сторонами в виде JSON-объекта. Эта информация может быть проверена и является доверенной, так как она подписана.

JWT-токены широко используются для авторизации и аутентификации в веб-приложениях, особенно в архитектурах без сохранения состояния (stateless), таких как RESTful API.

## Структура JWT:

- Header (Заголовок):** Содержит тип токена (JWT) и используемый алгоритм хеширования (например, HS256, RS256).
- Payload (Полезная нагрузка):** Содержит claims (заявления) – информацию о сущности (например, пользователе) и дополнительные данные. Стандартные claims включают `iss` (издатель), `exp` (срок действия), `sub` (субъект).
- Signature (Подпись):** Создается путем хеширования закодированных заголовка, полезной нагрузки и секретного ключа сервера. Это гарантирует целостность токена и его подлинность.

The screenshot shows the jwt.io website interface. At the top, there's a header with the jwt.io logo, navigation links like 'Debugger', 'Libraries', 'Introduction', 'Ask', and a note 'Crafted by Auth0 By Okta'. Below the header, a warning message says: 'Warning: JWTs are credentials, which can grant access to resources. Be careful where you paste them! We do not record tokens, all validation and debugging is done on the client side.' The main area is divided into two sections: 'Encoded' and 'Decoded'. The 'Encoded' section contains a long string of characters: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaaG4gRG9iIiwiWF0IjoxNTE2MjM5MDIyfQ.uXrIhrueuvbr4tD1ULho1QObboLVC-wIfJOEVE1Ezs. The 'Decoded' section shows the structure of the token:

```
{ "alg": "HS256", "typ": "JWT" }  
{"sub": "1234567890", "name": "John Doe", "iat": 1516239022}
```

Below the decoded payload, there's a 'VERIFY SIGNATURE' button with code: `HMACSHA256(  
base64UrlEncode(header) + "." +  
base64UrlEncode(payload),  
"3f0nf6nq3nf89q3nf09")`. A checked checkbox indicates 'secret base64 encoded'. At the bottom right, there's a blue 'SHARE JWT' button.

## Преимущества JWT:

- Компактность:** Легко передается в URL, POST-параметрах или HTTP-заголовках.
- Автономность:** Токен содержит всю необходимую информацию для проверки, не требуя обращения к базе данных на каждом запросе.
- Стандарт:** Широко распространен и поддерживается множеством библиотек.

## Когда использовать JWT?

- Авторизация:** После входа пользователя сервер выдает JWT, который затем используется клиентом для доступа к защищенным ресурсам.
- Обмен информацией:** Для безопасного обмена информацией между микросервисами или между клиентом и сервером.

# Реализация на Python

1

## Создание JWT-токена

```
import jwt
from datetime import *
SECRET_KEY = "your-very-secret-key"

def create_jwt_token(user_id):
    payload = {
        "user_id": user_id,
        "exp": datetime.utcnow() + timedelta(hours=1),
        "iat": datetime.utcnow()
    }
    token = jwt.encode(payload, SECRET_KEY,
algorithm="HS256")
    return token

user_token = create_jwt_token(123)
```

2

## Декодирование и проверка JWT-токена

```
def decode_jwt_token(token):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=
["HS256"])
        return payload
    except jwt.ExpiredSignatureError:
        print("Токен истёк.")
        return None
    except jwt.InvalidTokenError:
        print("Недействительный токен.")
        return None

decoded_payload = decode_jwt_token(user_token)
if decoded_payload:
    print(f"payload: {decoded_payload}")
    print(f"ID: {decoded_payload['user_id']}")
```

# ИТОГИ ЗАНЯТИЯ



## SQL-инъекции

1

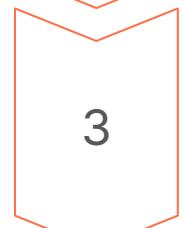
Научились предотвращать, используя параметризованные запросы и ORM.



## Хеширование паролей и соль

2

Освоили принципы безопасного хранения паролей с помощью `bcrypt`.



## JWT-токены

3

Поняли, как использовать для авторизации и безопасного обмена данными.



## Практическое применение

4

Теперь у вас есть базовые инструменты для создания более безопасных Python-приложений.

Помните, что безопасность – это непрерывный процесс. Всегда следите за новыми угрозами и лучшими практиками в области кибербезопасности.

