Урок 6

Безопасность

Базовые принципы ИБ. Криптография. Хэширование. Авторизация и аутентификация. Уязвимости ПО. OWASP Top 10. Автоматизация проверки кода.

Введение

Ряда уязвимостей ПО можно избежать, если при разработке уделить чуть больше внимания деталям безопасности. В этом случае ответственность полностью на разработчике и архитекторе системы. Именно поэтому программисту любого направления важно иметь представление об угрозах и типичных ошибках в программном обеспечении.

На этом уроке рассмотрим базовые аспекты того, как обеспечить безопасность разрабатываемой программной системы — как общие, касающиеся всех языков программирования, так и специфические для Python.

Первый и основной совет: критически относитесь к разрабатываемой системе с точки зрения безопасности.

Помните: ни одна система не является безопасной! Безопасность — это не результат, а процесс.

Базовые принципы ИБ

Информационная безопасность — это процесс обеспечения конфиденциальности, целостности и доступности информации.

Конфиденциальность (от англ. confidence — доверие) — предотвращение утечки (разглашения) информации.

Целостность — в информатике (криптографии, теории телекоммуникаций, теории информационной безопасности) означает, что данные не были изменены при выполнении операции над ними — будь то передача, хранение или отображение.

Доступность — обеспечение доступа к информации и связанным с ней активам авторизованных пользователей по мере необходимости.

Введение в криптографию

Рассмотрим базовые аспекты криптографии, которые позволят продолжить углубленное изучение темы при необходимости.

Криптография — наука о методах обеспечения конфиденциальности (невозможности прочтения информации посторонним), целостности данных (невозможности незаметного изменения информации), аутентификации (проверки подлинности авторства или иных свойств объекта), а также невозможности отказа от авторства.

Симметричное шифрование заключается в том, что обе стороны, участвующие в обмене данными, имеют абсолютно одинаковые ключи для шифрования и расшифровки данных. Выполняется преобразование, позволяющее предотвратить просмотр информации третьей стороной. Современные представители: AES, 3DES, Шифр «Кузнечик», ГОСТ 28147-89, Salsa20.

Асимметричное шифрование. Предполагает использование в паре двух разных ключей — открытого и секретного. Если данные шифруются открытым ключом, расшифровать их можно только соответствующим секретным, и наоборот — если шифруются секретным ключом, расшифровка — только соответствующим открытым. Использовать открытый ключ из одной пары и секретный из другой — невозможно. Каждая пара асимметричных ключей связана математическими

зависимостями. Данный способ также нацелен на преобразование информации от просмотра третьей стороной. Современные представители: RSA, DSA, ГОСТ Р 34.10-2012.

Хэширование — преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хэш-функциями или функциями свертки, а их результаты — хэш-кодом, контрольной суммой или дайджестом сообщения (message digest). Результаты хэширования статистически уникальны. Последовательность, отличающаяся хотя бы одним байтом, не будет преобразована в то же значение. Современные представители: MD5 (признан ненадежным), SHA-1 (признан ненадежным), SHA-2, SHA-3, ГОСТ Р 34.11-2012.

HMAC (hash-based message authentication code, код аутентификации (проверки подлинности) сообщений, использующий хэш-функции) — в криптографии один из механизмов проверки целостности информации, позволяющий гарантировать, что данные, передаваемые или хранящиеся в ненадежной среде, не были изменены посторонними лицами. Механизм HMAC использует MAC, он описан в RFC 2104. MAC — стандарт, излагающий способ обмена данными и проверки целостности передаваемых данных с использованием секретного ключа. Два клиента, использующие HMAC, как правило, разделяют общий секретный ключ. HMAC — надстройка над MAC — механизм обмена данными с использованием секретного ключа (как в MAC) и хэш-функций. В зависимости от используемой хэш-функции выделяют HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD128, HMAC-RIPEMD160 и т.п.

Цифровая подпись используется, чтобы устанавливать подлинность документа, его происхождения и авторства. Исключает искажения информации в электронном документе. Современные стандарты цифровой подписи: DSS, PKCS.

Криптографический протокол — абстрактный или конкретный протокол, включающий набор криптографических алгоритмов. В его основе — набор правил, регламентирующих использование криптографических преобразований и алгоритмов в информационных процессах. Примеры криптографических протоколов: доказательство с нулевым разглашением, «Забывчивая передача», протокол конфиденциального вычисления.

Python и криптография

Рассмотрим решение наиболее распространенных задач, связанных с криптографией: шифрование данных, хэширование, создание цифровой подписи. Для простых задач в Python есть встроенные модули: hashlib (вычисление хэш-функций), hmac (создание HMAC), secrets (генерация безопасных псевдослучайных чисел, Python 3), getpass (безопасный ввод логина и пароля).

Для более расширенных, связанных с криптографией, потребуется установка дополнительных модулей. Стоит отметить такие сторонние модули, как PyCryptodome, Cryptography.

Хэширование

Модуль **hashlib** поддерживает следующие алгоритмы хэширования: SHA1, SHA224, SHA256, SHA384, SHA512, MD5. В Python 3.6 в модуль **hashlib** добавлены алгоритмы SHA3 (Keccak), SHAKE и BLAKE2.

Общая схема вычисления хэш-функции:

1. Создать объект хэш-функции.
2. Добавить в объект данные для вычисления хэш-функции (метод **update()**).
3. Получить значение хэш-функции (методы **digest()** и **hexdigest()**).

В качестве данных, для которых будет вычисляться хэш-функция, должна быть **строка байтов**.

Пример простого вычисления хэш-функции:

```
import hashlib

# Создание объекта хэш-суммы
h = hashlib.md5()
# Добавление данных для расчета суммы - можно добавлять только строку байтов
h.update(b'Python')
# Вывод хэш-суммы
print(h.hexdigest())
```

Модуль **hashlib** также содержит функции для хэширования (расширения) паролей. Хорошая функция хэширования паролей должна быть настраиваемой, вычисляться медленно и включать криптографическую «соль».

- **hashlib.pbkdf2_hmac(hash_name, password, salt, iterations, dklen=None)**. Реализует стандарт PKCS#5, использует HMAC в качестве псевдослучайного генератора.
    - **hash_name** — строка с именем алгоритма хэширования ('sha1', 'sha256' и т.д.);
    - **password, salt** — строки байтов, **salt** длиной 16 и более байт;
    - **iterations** — количество итераций (из расчета примерно 100000 для SHA-256);
    - **dklen** — длина расширенного ключа. Если не указана, берется длина выхода алгоритма хэширования.
- **hashlib.scrypt(password, *, salt, n, r, p, maxmem=0, dklen=64)** (необходимы Python 3.6 и OpenSSL 1.1+). Реализует функцию расширения ключей scrypt (RFC 7914).
    - **password, salt** — строки байтов, **salt** длиной 16 и более байт;
    - **n** — фактор значимости CPU/памяти;
    - **r** — размер блока;
    - **p** — фактор параллелизма;
    - **maxmem** — ограничение по памяти (по умолчанию в **OpenSSL** 1.1.0 — 32 MB);
    - **dklen** — длина расширенного ключа.

Пример вычисления надежного хэша для пароля:

```
import hashlib, binascii
dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
# Вычисленное значение можно хранить в БД
print(binascii.hexlify(dk))
```

HMAC

Рассмотрим задачу, когда необходимо организовать простую аутентификацию сообщений в распределенной системе (клиент–сервер), но нет необходимости реализовывать более сложную, вроде SSL/TLS.

Простым и эффективным решением будет использование стандартного модуля **hmac** (файлы **листинг 1** и **листинг 2**):

```
def server_authenticate(connection, secret_key):
    ''' Запрос аутентификаии клиента.
        connection - сетевое соединение (сокет);
        secret_key - ключ шифрования, известный клиенту и серверу
    '''
    # 1. Создается случайное послание и отправляется клиенту
    message = os.urandom(32)
    connection.send(message)

    # 2. Вычисляется HMAC-функция от послания с использованием секретного ключа
    hash = hmac.new(secret_key, message)
    digest = hash.digest()

    # 3. Пришедший ответ от клиента сравнивается с локальным результатом HMAC
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest, response)

def client_authenticate(connection, secret_key):
    ''' Аутентификация клиента на удаленном сервисе.
        Параметр connection - сетевое соединение (сокет);
        secret_key - ключ шифрования, известный клиенту и серверу
    '''
    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)
```

Идея в том, что по сетевому соединению сервер передает клиенту сообщение, состоящее из случайных байт (в данном случае — на основе функции **os.urandom()**). Клиент и сервер вместе вычисляют криптографическую хэш-функцию для этих случайных данных, используя HMAC-алгоритм и секретный ключ, известный только клиенту и серверу.

Клиент отправляет серверу вычисленное HMAC-значение, которое сервер сверяет со своим. На основании проверки делается вывод — использовать текущее сетевое соединение или сбросить его.

Сравнение результатов нужно проводить функцией **hmac.compare_digest()**, а не оператором **==**. Функция создана специально, чтобы предотвратить атаки по времени на HMAC-алгоритм.

Данные функции аутентификации можно встроить в процесс сетевого обмена следующим образом (см. **листинг 1** и **листинг 2**):

```
# ---------------------- Эхо-сервер ----------------------------------
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'our_secret_key'

def echo_handler(client_sock):
    ''' Эхо-обработка.
        Проводит аутентификацию клиента и отсылает его же запрос обратно (эхо).
    '''
    if not server_authenticate(client_sock, secret_key):
        client_sock.close()
        return

    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)

def echo_server(address):
    ''' Эхо-сервер.
        "Слушает" указанный адрес и общается с клиентом через echo_handler.
    '''
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)
    while True:
        conn, addr = sock.accept()
        echo_handler(conn)


echo_server(('', 9999))


# ------------- Клиент -------------------------------------------
''' Клиент подключается к серверу, проходит аутентификацию, далее -
    общается по обычному протоколу
'''
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'our_secret_key'

sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 9999))

client_authenticate(sock, secret_key)

sock.send(b'Hello, my secure server!')
resp = sock.recv(1024)
print('Сервер ответил: ', resp.decode())
```

Обычно HMAC-аутентификация используется во внутренних системах обмена сообщениями и в межпроцессном взаимодействии. Например, если требуется написать систему, которая включает множество процессов, взаимодействующих на кластере из нескольких машин. Тогда можно использовать данный подход, чтобы гарантировать, что только разрешенные процессы могут подключаться к другим. Модуль **muliprocessing** внутри использует аутентификацию, основанную на **HMAC**, чтобы установить соединения с подпроцессами.

Аутентификация соединения не подразумевает последующего шифрования — все сообщения будут передаваться в открытом виде по сети. Любой, кто имеет доступ к сети, может перехватить сетевой трафик.

Шифрование данных

Рассмотрим случай, когда необходимо выполнить шифрование данных — чтобы третья сторона не смогла прочитать их. Для этого можно воспользоваться библиотекой PyCryptodome. **PyCryptodome** (**PyCryptoDomeEx**) — это **fork** библиотеки **PyCrypto**, реализующей криптографические примитивы и функции на Python. Однако исходная библиотека **PyCrypto** не обновляется с 2014 года, поэтому имеет смысл использовать **PyCryptodome**. Она полностью совместима по API с **PyCrypto**. **PyCryptoDomeEx** дополняет/изменяет исходный API.

Установка стандартная:

```
pip install pycryptodome
```

Для шифрования данных в **PyCryptodome** есть поддержка нескольких алгоритмов:

- блочные шифры: AES, DES, 3DES, Blowfish;
- поточные шифры: Salsa20, ChaCha20, RC4.

Рассмотрим код, реализующий функции шифрования/расшифрования с использованием симметричного шифра **AES** (Advanced Encryption Standard) (см. **листинг 3**):

```
import os
from binascii import hexlify
from Cryptodome.Cipher import AES

plaintext = b'The rain in Spain'

def padding_text(text):
    ''' Выравнивание сообщения до длины, кратной 16 байтам.
        В данном случае исходное сообщение дополняется пробелами.
    '''
    pad_len = (16 - len(text) % 16) % 16
    return text + b' ' * pad_len

def _encrypt(plaintext, key):
    ''' Шифрование сообщения plaintext ключом key.
        Атрибут iv - вектор инициализации для алгоритма шифрования.
        Если не задается явно при создании объекта-шифра, генерируется случайно.
        Его следует добавить в качестве префикса к финальному шифру,
        чтобы была возможность правильно расшифровать сообщение.
    '''
    cipher = AES.new(key, AES.MODE_CBC)
    ciphertext = cipher.iv + cipher.encrypt(plaintext)
    return ciphertext

def _decrypt(ciphertext, key):
    ''' Расшифровка шифра ciphertext ключом key.
        Вектор инициализации берется из исходного шифра.
        Его длина для большинства режимов шифрования всегда 16 байт.
        Расшифровываться будет оставшаяся часть шифра.
    '''
    cipher = AES.new(key, AES.MODE_CBC, iv=ciphertext[:16])
    msg = cipher.decrypt(ciphertext[16:])
    return msg

# Осуществим шифрование сообщения алгоритмом AES
# key (строка байтов) - секретный ключ для симметричного шифрования.
# Ключ должен быть длиной 16 (AES-128), 24 (AES-192) или 32 (AES-256) байта.
key = b'Super Secret Key'
# Длина сообщения должна быть кратна 16, поэтому выполним выравнивание.
plaintext = padding_text(plaintext)

# Выполним шифрование
cipher = _encrypt(plaintext, key)
print(hexlify(cipher))

# Выполним расшифрование
msg = _decrypt(cipher, key)
print(msg)
```

Некоторые особенности использования шифра **AES** из модуля **PyCryptodome**:

- допустимые **длины ключа**: 16 (AES-128), 24 (AES-192), 32 байта (AES-256);
- **длина шифруемого сообщения** должна быть кратна 16 байтам, поэтому нужно выполнять выравнивание текста;
- **вектор инициализации (iv)** нужно задавать в конструкторе шифра **new**. Если вектор не указывается явно, он инициализируется библиотекой **PyCryptodome** автоматически случайным значением;
- шифр — объект, хранящий свое **состояние**, поэтому нельзя использовать один объект-шифр для разных сообщений.

Режимы работы блочных шифров

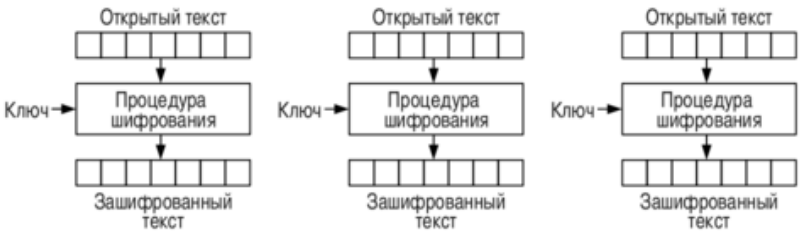
При работе с блочными шифрами (AES, 3DES, ГОСТ 28147-89 и прочими) нужно иметь представление о том, в каком режиме они будут использоваться.

Режим шифрования — метод применения блочного шифра (алгоритма) для преобразования последовательности блоков открытых данных в последовательность блоков зашифрованных данных. При этом для шифрования одного блока могут использоваться данные другого.

Обычно режимы шифрования используют для изменения процесса шифрования так, чтобы результат для каждого блока был уникальным вне зависимости от шифруемых данных и не позволял сделать выводы об их структуре. Блочные шифры шифруют данные блоками фиксированного размера. Поэтому потенциально возможна утечка информации о повторяющихся частях данных, шифруемых на одном и том же ключе. Этого позволяют избежать режимы шифрования.

Electronic code book (ECB)

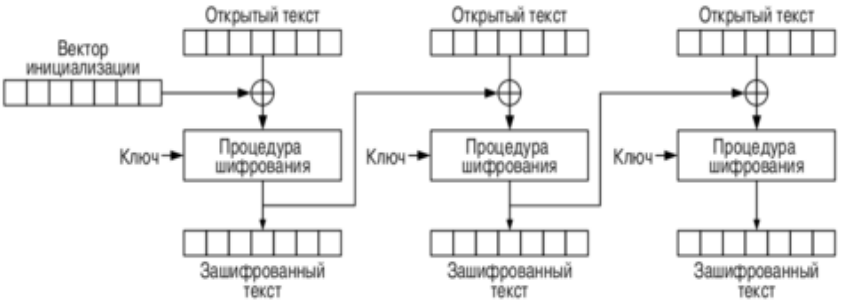
Режим электронной кодовой книги, или режим простой замены (ECB). Сообщение делится на блоки одинакового размера. Каждый блок **Pi** шифруется алгоритмом шифрования **Ek** с использованием ключа **k**.



Основной **недостаток** режима **ECB** — сохранение статистических особенностей открытого текста (так как одинаковым блокам шифротекста соответствуют одинаковые блоки открытого текста).

Cipher block chaining (CBC)

Режим сцепления блоков шифротекста (**CBC**) показан на рисунке:



Недостатки CBC:

- возможность определения начала изменения данных при изменении шифротекста. **Если в пределах одного сектора до какого-то определенного места данные не меняются, то и шифроблоки останутся неизменными.**
- возможность изменения открытого текста при изменении порядка следования зашифрованных секторов. Это создает риск дешифрования открытого текста каждого перемещенного блока.
- возможность изменения блока шифротекста **Ci−1** путем изменения блока сообщения **Pi**;
- нельзя распараллеливать шифрование — поскольку блоки связаны и для шифрования каждого i-го блока требуется тот, что зашифрован на предыдущем шаге.

Достоинства CBC:

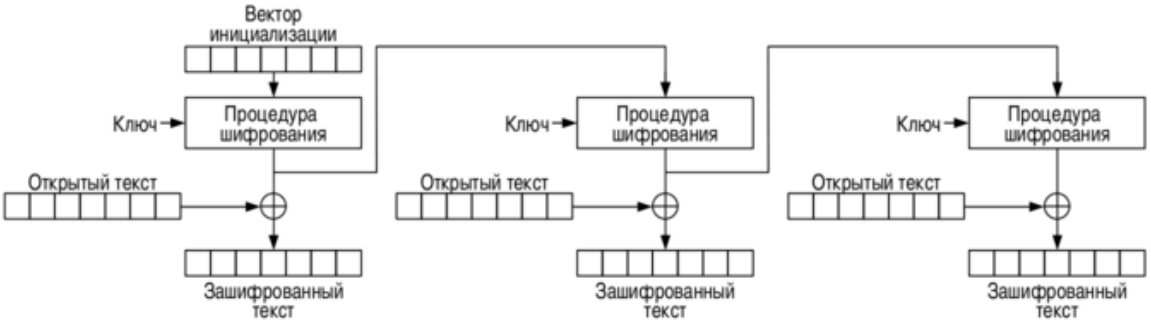
- постоянная скорость обработки блоков (скорость определяется эффективностью реализации шифра; время выполнения операции **xor** мало и им можно пренебречь);
- отсутствие статистических особенностей, характерных для режима **ECB** (поскольку каждый блок открытого текста «смешивается» с блоком шифротекста, полученным на предыдущем шаге);
- возможность распараллеливания расшифровки.

Cipher feed back (CFB)

Режим обратной связи по шифротексту, режим гаммирования с обратной связью (**CFB**). Во время шифрования каждый блок открытого текста складывается по модулю 2 с блоком, зашифрованным на предыдущем шаге.

Output feed back (OFB)

Режим обратной связи по выходу (**OFB**) превращает блочный шифр в синхронный шифр потока: он генерирует ключевые блоки, которые являются результатом сложения с блоками открытого текста, чтобы получить зашифрованный текст. Каждая операция блочного шифра обратной связи выхода зависит от всех предыдущих и поэтому не может быть выполнена параллельно.



Выбор режима шифрования

Выбор режима шифрования зависит от цели. Для обычного открытого текста можно использовать **CBC**, **CFB** или **OFB**. Для шифрования файлов лучше пользоваться **CBC**: значительно увеличивается безопасность, при ошибках в хранимых данных почти не бывает сбоев синхронизации. Конкретный режим зависит от требований. Выбор метода шифрования — это поиск компромисса между эффективностью и производительностью.

Уязвимости ПО

Количество киберугроз увеличивается пропорционально росту бизнеса, но как показала многолетняя практика, 99 % атак происходит:

- через десяток стандартных ошибок валидации входных данных;
- через уязвимые компоненты ПО от сторонних производителей;
- по халатности системных администраторов, использующих настройки и пароли, установленные по умолчанию.

Классификацией векторов атак и уязвимостей занимается сообщество ***OWASP*** (The Open Web Application Security Project). Это международная некоммерческая организация, сосредоточенная на анализе и улучшении безопасности программного обеспечения.

OWASP создали список из 10 самых опасных векторов атак на web-приложения — OWASP TOP-10. В нем сосредоточены самые опасные уязвимости, которые могут стоить организациям больших денег или деловой репутации, а то и самого бизнеса.

OWASP Top 10 от 2017 года

- A1 — инъекции (Injections);
- A2 — недочеты системы аутентификации и хранения сессий (Broken Authentication and Session Management);

- A3 — незащищенность критичных данных (Sensitive Data Exposure);
- A4 — уязвимость сайта или программы к XML-инъекциям (XML External Entities — XEE);
- A5 — уязвимость, связанная с методами авторизации, что позволяет злоумышленнику получить сверхпривилегии (Broken Access Control);
- A6 — небезопасная конфигурация (Security Misconfiguration);
- A7 — межсайтовый скриптинг (Cross Site Scripting — XSS);
- A8 — небезопасная десериализация (Insecure Deserialization);
- A9 — использование компонентов с известными уязвимостями (Using Components with Known Vulnerabilities);
- A10 — недостаточное журналирование и мониторинг (Insufficient Logging&Monitoring).

Есть еще OWASP-рейтинг уязвимостей для мобильных приложений (OWASP Mobile Top 10), руководство по обзору кода OWASP и другие проекты (OWASP Project).

Стоит обратить внимание на рекомендации от OWASP — например, принципы создания безопасного приложения на уровне дизайна (OWASP Security by Design Principles):

1. Минимизировать площадь воздействия атаки (Minimize attack surface area).
2. Укреплять безопасные значения (Establish secure defaults).
3. Принцип наименьших привилегий (Principle of Least privilege).
4. Принцип глубокой защиты (Principle of Defence in depth).
5. Делать системные сбои безопасными (Fail securely).
6. Не доверять сервисам (Don’t trust services).
7. Разделение обязанностей (Separation of duties).
8. Избегать принципа «безопасности через сокрытие» (Avoid security by obscurity).
9. Безопасность должна быть простой (Keep security simple).
10. Корректно исправлять ошибки безопасности (Fix security issues correctly).

Что такое уязвимость

Термин **«уязвимость»** (англ. vulnerability) обозначает недостаток в системе, используя который, можно намеренно нарушить ее целостность и вызвать неправильную работу. Уязвимость может быть результатом ошибок программирования и проектирования системы, ненадежных паролей, вирусов и других вредоносных программ, скриптовых и SQL-инъекций. Некоторые уязвимости известны только теоретически, другие же активно используются и имеют известные эксплойты (например, SecurityLab: Уязвимости).

Распространенные типы уязвимостей:

- Нарушения безопасности доступа к памяти:

    - Переполнение буфера;
    - Висячие указатели.
- Ошибки проверки вводимых данных:
    - Ошибки форматирующей строки;
    - Неверная поддержка интерпретации метасимволов командной оболочки;
    - SQL-инъекция;
    - Инъекция кода;
    - Email-инъекция;
    - Обход каталогов;
    - Межсайтовый скриптинг в веб-приложениях;
    - Межсайтовый скриптинг при наличии SQL-инъекции.
- Состояния гонки:
    - Ошибки «времени-проверки-ко-времени-использования»;
    - Гонки символьных ссылок.
- Ошибки путаницы привилегий:
    - Подделка межсайтовых запросов в веб-приложениях.
- Эскалация привилегий:
    - Shatter attack.

По рейтингу уязвимостей на 2010, 2013, 2017 годы первое место занимают **уязвимости внедрения кода** (инъекции — Injection).

Относительно Python можно выделить несколько вариантов, когда инъекции возможны:

- SQL-инъекции;
- использование небезопасных функций **eval/exec**;
- использование модуля **pickle** для сериализации и хранения/передачи объектов.

SQL-инъекции

Для примера рассмотрим базу данных **sqlite3**, имеющую таблицу **USER** с данными для авторизации пользователей (логин, пароль) (полный код смотрите в файле **листинг 4**). Таблица заранее заполнена данными:

id	login	password
1	‘admin’	`'21232f297a57a5a743894a0e4a801fc3'`

2	‘user’	'ee11cbb19052e40b07aac0ca060c23ee'
3	‘guest’	'084e0343a0486ff05530df6c705c8bb4'

Требуется реализовать функцию, возвращающую данные пользователя по его ID в базе данных.

В самом простом случае запрос с SQL-инъекцией будет выглядеть так:

```
# Пример простой SQL-инъекции.
# Строка подобного рода уязвима к SQL-инъекциям:
week_select_1 = "SELECT * FROM USER WHERE id = "
week_select_1_2 = "SELECT * FROM USER WHERE id = {}"

curr.execute(week_select_1 + ('1 OR 1=1'))
res = curr.fetchall()
print('Результат первой уязвимой строки с запросом: ')
print(res)

curr.execute(week_select_1_2.format('1 OR 1=1'))
res = curr.fetchall()
print('Результат первой модифицированной строки с запросом: ')
print(res)
```

В данном случае будут неутешительные результаты:

```
[(1, 'admin', '21232f297a57a5a743894a0e4a801fc3'),
 (2, 'user', 'ee11cbb19052e40b07aac0ca060c23ee'),
 (3, 'guest', '084e0343a0486ff05530df6c705c8bb4')]
```

Были извлечены все записи из таблицы USER. Такое поведение SQL-запроса может вызвать недоумение, однако с точки зрения языка SQL **никаких ошибок в запросе нет**.

Дело в том, что итоговый запрос примет вид:

```
SELECT * FROM USER WHERE id = **1 OR 1=1**
```

Логический оператор **OR** выдает результат **True**, если хотя бы одно из условий истинно. В данном случае все логическое выражение будет истинно, и его можно было бы переписать так:

```
SELECT * FROM USER WHERE 1
```

Может прийти мысль защититься от инъекции, добавив в строку запроса кавычки. Однако этот способ тоже имеет недостатки:

```
# Попытка защититься, экранировав кавычками отдельные параметры запроса.
# Однако такая строка тоже может быть уязвима:
week_select_2 = 'SELECT * FROM USER WHERE id = "{}" AND login = "{}"'

curr.execute(week_select_2.format('" or ""="', '" or ""="'))
```

```
res = curr.fetchall()
print('Результат второй уязвимой строки с запросом: ')
print(res)
```

Результирующий запрос примет вид:

```
SELECT * FROM USER WHERE id = "" or ""="" AND login = "" or ""=""
```

Логическое выражение усложнилось, но результат его тоже будет истинным. Основное влияние окажет сравнение пустых строк (""="") — оно является истинным. В итоге получится выражение, которое так же извлечет все данные из таблицы **USER**:

```
SELECT * FROM USER WHERE 1 AND 1
```

Кроме выборки данных, SQL-инъекции могут приводить к изменению/удалению данных из БД. Такая инъекция может работать не на каждой СУБД:

```
# Простая SQL-инъекция с большими возможностями
week_select_3 = "SELECT * FROM USER WHERE id = {}"

curr.execute(week_select_3.format('1; DROP TABLE USER;'))
res = curr.fetchall()
print('Результат третьей уязвимой строки с запросом: ')
print(res)
```

В данном случае результирующий запрос примет вид:

```
SELECT * FROM USER WHERE id = 1; DROP TABLE USER;
```

Будет выполнено два запроса вместо одного.

- При работе с **SQlite3** для данного запроса будет получено исключение, так как запрос выполняется через метод **execute**, а не **executemany**:

```
sqlite3.Warning: You can only execute one statement at a time.
```

Чтобы избежать SQL-инъекций в Python-коде, необходимо использовать **параметризованные запросы** (что указано в документации по DB-api) или ORM (**sqlalchemy**, **PonyORM**, **peewee**):

```
You shouldn't assemble your query using Python's string operations because
doing so is insecure; it makes your program vulnerable to an SQL injection
attack.
Instead, use the DB-API's parameter substitution.
```

Можно переписать первый запрос с использованием параметра:

```
week_select_1 = "SELECT * FROM USER WHERE id = ?"
```

```
curr.execute(week_select_1, '1 OR 1=1')
```

Результатом такого запроса будет исключение:

```
sqlite3.ProgrammingError: Incorrect number of bindings supplied.
The current statement uses 1, and there are 8 supplied.
```

Функция eval

Среди встроенных функций Python есть **eval** — она разбирает и исполняет указанное выражение.

Прототип:

```
eval(expression, globals=None, locals=None)
```

Параметры

- **expression** — выражение, которое требуется выполнить, в виде строки. Либо объект кода;
- **globals=None** — ожидается **dict**. Словарь глобального пространства, относительно которого следует исполнить выражение. Если указан, но не содержит атрибута **__builtins__,** перед разбором выражения данные указанного пространства будут дополнены данными общего глобального пространства. Так выражение будет иметь доступ ко всем встроенным модулям.
- **locals=None** — ожидается объект-отображение (например, **dict**). Локальное пространство, в котором следует исполнить выражение. Если не указано, используется словарь глобального пространства.

  Данная функция может выполнить любую строку кода Python (иногда требуется немного изменить исходную строку кода). Так, можно выполнить любую системную команду через модуль **os** — например, команду **echo** (см. **листинг 5**):

```
# Будьте осторожны с функцией eval:
eval("__import__('os').system('echo evil_eval BU-ga-ga')")
```

Функция **eval** может быть «точкой входа» для инъекций кода. Если внутри кода программист обращается к функции **eval** с параметром-строкой, которая вводится пользователем/читается из файла настроек, — у злоумышленника появляется возможность повлиять на ту строку, которая будет выполнена функцией **eval**. Например, удалить все содержимое файловой системы (**rm -rf / (Unix) / format C: (Windows)**).

Иногда разработчик пытается «обезопасить» функцию **eval**, указав для нее пустой словарь **__builtins__**:

```
eval("__import__('os').system('echo evil_eval coming again')",
{'__builtins__':{}})
```

Выполнение такой строки будет невозможно.

Однако этот подход также не гарантирует абсолютную безопасность. Есть возможность сформировать строку кода для выполнения функцией **eval**, которая будет приводить к краху процесса интерпретатора (актуально для CPython-интерпретатора). Такой подход может использовать

злоумышленник, чтобы привести к отказу в обслуживании (Denial of Service, DoS) (см. «И снова про опасность eval()» (Хабр)):

```
# Будьте аккуратны, код ниже приведет к некорректному завершению работы
интерпретатора
s = """ (lambda fc=(
        lambda n: [
            c for c in
                ().__class__.__bases__[0].__subclasses__()
                if c.__name__ == n
            ][0]
        ):
        fc("function")(
            fc("code")(
                0,0,0,0,0,b"BO00OM",(),(),(),"","",0,b""
            ),{}
        )()
    )() """
eval(s, {'__builtins__':{}})
```

Пояснения к коду

1. Словарь **__builtins__** теперь пустой — нет возможности обратиться к функции **__import__**;
2. Однако есть возможность получить все классы-наследники для класса **object**:

   ```
   ().__class__.__bases__[0].__subclasses__()
   ```

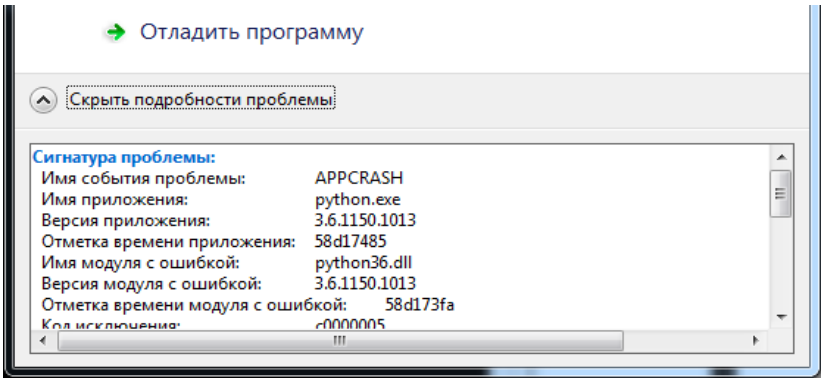
3. Чтобы найти класс по имени, можно организовать цикл, который потом использовать в генераторе списка:

   ```
   [c for c in ().__class__.__bases__[0].__subclasses__() if c.__name__ == n][0]
   ```

4. Создать обычную функцию в eval-строке нельзя, но можно lambda-функцию:

   ```
   lambda n: [c for c in ().__class__.__bases__[0].__subclasses__()
              if c.__name__ == n][0]
   ```

5. В Python есть внутренние классы **function** и **code**, которые служат для создания объекта-функции и объекта кода функции соответственно (см. Exploring Python Code Objects).
6. Финальный код будет создавать объект-функцию с указанным объектом кода.
7. Есть параметр **stacksize**, задающий размер стека для функции (там хранятся ее аргументы и переменные). В данном случае размер стека равен 0, что приведет к некорректному выполнению созданной функции и исключению **SEGFAULT**.



Модуль pickle

Модуль **pickle** служит для сохранения Python-объектов (сериализация/десериализация). Однако при десериализации не проверяется содержимое объекта. Это может использовать злоумышленник, чтобы внедрить вредоносный код в сохраненные pickle-данные (см. документацию на модуль pickle).

Рассмотрим уязвимости модуля **pickle** подробнее.

Строка ниже выполнит системную функцию **echo** (см. **листинг 6**):

```
import pickle
pickle.loads(b"cos\nsystem\n(S'echo I am Evil Pickle-module!'\ntR.")
```

При работе с распределенными системами (например, «клиент–сервер») может возникать необходимость синхронизации/передачи объектов между разными процессами/машинами. Иногда такую задачу решают, сохраняя состояние объекта и используя модуль **pickle**.

Рассмотрим клиент-серверную систему, где сервер создает небезопасный объект, сериализует его при помощи модуля **pickle** и передает по сети:

```
# -------- Сервер, передающий pickle-объект по сети ------------
import subprocess
import socket

class EvilPayload:
    """ Функция __reduce__ будет выполнена при распаковке объекта
    """
    def __reduce__(self):
        """ Запустим на машине клиента безобидный Notepad (или другой редактор)
        """
        import os
        os.system("echo You've been hacked by Evil Pickle!!! > evil_msg.txt")
        return (subprocess.Popen, (('notepad','evil_msg.txt'),))

# Реализуем простой сокет-сервер для демонстрации примера.
# Клиентское приложение находится в файле evil_pickle_client.py
def evil_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(("localhost", 9999))
    print('Зловещий сервер запущен...')
    sock.listen()
```

```
    conn, addr = sock.accept()

    print('К нам попался клиент', addr)
    print('Отправляем ему "троянца"...')
    # Отсылаем опасный объект "доверчивому" клиенту
    conn.send(pickle.dumps(EvilPayload()))

evil_server()
```

В классе **EvilPayload** реализован метод **__reduce__** — он будет вызван автоматически при десериализации pickle-объекта.

Для демонстрации распаковки такого объекта, получаемого по сети, реализуем клиентскую часть системы (см. **листинг 7**):

```
# --- Простой сокет-клиент для демонстрации работы с pickle-данными -------
import pickle
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("localhost", 9999))

# Получаем опасное сообщение
message = sock.recv(1024)
# Распаковываем, "радуемся" - нас взломали...
pickle.loads(message)
sock.close()
```

Рекомендации по более безопасному использованию модуля **pickle**:

1. По возможности — использовать шифрование сетевого трафика (SSL/TLS).
2. Если шифрование невозможно — применять электронную подпись для подтверждения данных.
3. Если pickle-данные сохраняются на диск — проверить, что только доверенные процессы могут менять их.
4. По возможности вместо **pickle** использовать, например, модуль **JSON**.

Общие выводы

1. Необходимо фильтровать/проверять ввод данных от пользователя (например, регулярными выражениями).
2. Внимательно читать документацию для используемых функций.

Автоматизация проверки кода

Не всегда мы можем контролировать использование небезопасных библиотек и функций в проекте. Для автоматизации проверок безопасности Python-кода существуют две библиотеки:

1. Safety.
2. Bandit.

Они представляют собой не только библиотеки, но и утилиты командной строки, выполняющие проверку Python-кода. Запуск утилит проверки может быть встроен в процесс непрерывной интеграции с утилитами **tox**, **buildbot**, **jenkins**.

Safety

Safety — утилита командной строки (и Python-модуль), которая позволяет проверять текущее виртуальное окружение и файл **requirements.txt** на наличие небезопасных пакетов как локально, так и на CI-сервере (сервер непрерывной интеграции).

Safety работает с открытой базой данных уязвимостей, расположенной на Github — Safety DB. Данные в репозитории обновляются порталом ***pyup.io*** раз в месяц.

Установка стандартная:

```
pip install safety
```

Проверка текущего виртуального окружения выполняется командой:

```
safety check
```

Проверка файла зависимостей (requirements) выполняется командой:

```
safety check -r requirements.txt
```

Проект Safety имеет ряд дополнительных утилит:

- **Safety CI** — интеграция с Github, позволяет проверять коммиты пользователя и Pull Request-ы;
- **Safety Django** — пакет для Django, который в админке выдает предупреждения, если установленная версия Django небезопасна;
- **Safety Bar** (alpha) — приложение для меню в macOS;
- **pre-commit hook** by Lucas Cimon — перехватчик (hook) для git, который выполняется перед коммитом и проверяет Python-зависимости проекта по **Safety DB**. Подробнее узнать о git-hooks можно в Pro Git book.

Bandit

Bandit — это линтер (анализатор кода), направленный на безопасность, от сообщества OpenStack Security (https://github.com/openstack/bandit).

Bandit разработан для поиска проблем безопасности в Python-коде. Для этого он обрабатывает каждый файл проекта, строит AST-дерево (abstract syntax tree) и запускает соответствующие плагины для каждой AST-ветки.

После сканирования всех файлов Bandit создает отчет.

Установка:

```
pip install bandit
```

Соответствие версии интерпретатора

Bandit выполняет построение AST-дерева с использованием python-модуля **ast**. Модуль **ast** может разбирать код, соответствующий версии интерпретатора, из которой он импортируется. Соответственно, запускать Bandit нужно в версии интерпретатора Python, для которой написан код исследуемого проекта.

Запуск Bandit

Полная проверка всех файлов в указанной директории:

```
bandit -r путь/к/файлу
```

Возможен запуск с определенным профилем, где могут быть указаны плагины для запуска:

```
bandit examples/*.py -p ShellInjection
```

Настройка

Можно произвести собственную настройку через файл конфигурации, где указать списки тестов для запуска и пропуска, исключения директорий, отдельно настроить плагины.

Каждый исследуемый проект может иметь свой файл с расширением **.bandit**, где указываются опции командной строки для:

- исключения директорий;
- пропуска тестов;
- тестов для запуска.

Небезопасные версии ПО

Уязвимости могут появляться в интерпретаторе/компиляторе, а также сторонних библиотеках.

Одной из самых известных систем классификации уязвимостей является CVE (Common Vulnerabilities and Exposures), которая курируется компанией NCSD (National Cyber Security Division) при одном из министерств США.

По сути, CVE — это «словарь» известных уязвимостей, имеющий строгие описательные критерии, что отличает его от Bugtrack-ленты. Полностью CVE можно найти в Национальной Базе Уязвимостей США (NVD — nvd.nist.gov) или на официальном сайте (cve.mitre.org/data/downloads). База уязвимостей распространяется в нескольких форматах: xml, html, csf, xsd schema.

Приведем несколько уязвимостей в интерпретаторе Python, обнаруженных в последние два года (2015, 2016).

#	CVE ID	Тип уязвимости	Счет	Описание
1	CVE-2015-5652	+Priv	7.2	Уязвимость — локальному пользователю поднять свои привилегии на Windows-системе
2	CVE-2016-5636	Overflow	10.0	Целочисленное переполнение в функции **get_data** в модуле **zipimport.c** в CPython
3	CVE-2016-0772	Bypass	5.8	Ошибка в библиотеке **smtplib** в CPython позволяет провести атаку «человек посередине» и обойти TLS-защиту
4	CVE-2013-7440		4.3	Функция **ssl.match_hostname** в CPython некорректно обрабатывает маски в именах хостов, что дает возможность обмана серверов

За все время существования интерпретатора Python обнаружено не так много уязвимостей. Но стоит о них знать, а перед использованием компонентов в проекте — выяснить, есть ли у них известные уязвимости.

Вредные советы

1. Доверяй пользовательским данным. Всегда.
2. Храни пароли в открытом виде.
3. Проводи авторизацию по незашифрованному каналу.
4. Придумай свою криптографию.
5. Никогда не подписывай данные.
6. Следуй принципу «Security through obscurity».
7. Не заглядывай в список **CVE** — это страшно.
8. Всегда используй устаревшее ПО.
9. Устанавливай неизвестные библиотеки.
10. Пиши сложный код.
11. Все помещай в репозиторий (пароли, явки, ip-адреса).
12. Не делай бэкапы.

Практическое задание

1. Реализовать аутентификацию пользователей на сервере.
2. Реализовать декоратор **@login_required**, проверяющий авторизованность пользователя для выполнения той или иной функции.

3. Реализовать хранение паролей в БД сервера (пароли не хранятся в открытом виде — хранится хэш-образ от пароля с добавлением криптографической соли).

4. * Реализовать возможность сквозного шифрования сообщений (использовать асимметричный шифр, ключи которого хранятся только у клиентов).

Дополнительные материалы

1. Python, pickle и внедрение кода.
2. PYCON RUSSIA 2017. (Без) опасный Python. Иван Цыганов.
3. Python Pickle Security Problems and Solutions.
4. Использование модуля pickle на своих объектах (Хабр).
5. SQL injection для начинающих. Часть 1 (Хабр).
6. w3schools - SQL Injection.
7. Алгоритм RSA (Khan Academy) - хорошее видео.
8. Алгоритм DSA.
9. Python 3: An Intro to Encryption.
10. OWASP TOP-10: 2017.
11. OWASP Secure Coding Practices.
12. Awesome Cryptography (A curated list of cryptography resources and links).
13. Подписываем данные: HMAC на практике в API и Web-формах (Хабр).

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. OWASP Romania Conference 2014. Secure Coding with Python.
3. OWASP Top 10-2017. The Ten Most Critical Web Application Security Risks.
4. Меряем уязвимости – классификаторы и метрики компьютерных брешей.
5. Обзор угроз безопасности WEB-приложений.

Приложение

Листинг 1

```
# ========================= Аспекты безопасности
==============================
```

```
# ------------- Простая аутентификация клиента. Реализация сервера
-------------


import hmac
import os

# HMAC - Keyed-Hashing for Message Authentication.
# Модуль hmac служит для вычисления хэш-функции с ключом от сообщения.
# Такой подход используется для аутентификации сообщений.
# Подробное описание алгоритма содержится в RFC 2104
(https://tools.ietf.org/html/rfc2104.html).

# Возможности данного модуля могут быть применены для аутентификации клиента.

#                              Общая схема работы
#                  СЕКРЕТ известен только клиенту и серверу (ключ)
# ----------------------
----------------------------
# |       Клиент         |                          |           Сервер
|
# |--------------------  |  запрос_аутентификации
|-----------------------------|
# |                      | ----------------------> |      Генерация rnd_msg
|
# |                      |         rnd_msg         |
|
# |                      | <---------------------  |
|
# |     Вычисление       |                          |         Вычисление
|
# |HMAC(СЕКРЕТ, rnd_msg)|                          |    HMAC(СЕКРЕТ, rnd_msg)
|
# |                      |                          |
|
# |                      |       HMAC_клиент       |
|
# |                      | ----------------------> |         Сравнение
|
# |                      |                          | HMAC_клиент == HMAC_сервер
|
# |                      |                          |            |  |
|
# |                      |       свой (доверие) <--------------да   нет
|
# |                      | <====================== |                 |--> чужой
|
# ------------------------
----------------------------
```

```
# hmac.new(key, msg=None, digestmod=None)
# key - byte-строка, представляющая ключ
# msg - сообщение, для которого нужно вычислить хэш
# digestmod - имя хэш-фукнции, которая будет применена для вычисления (sha-1,
sha-256, ...)


# -------------- Функция аутентификации клиента на сервере
--------------------
def server_authenticate(connection, secret_key):
    ''' Запрос аутентификаии клиента.
        connection - сетевое соединение (сокет);
        secret_key - ключ шифрования, известный клиенту и серверу
    '''
    # 1. Создаётся случайное послание и отсылается клиенту
    message = os.urandom(32)
    connection.send(message)

    # 2. Вычисляется HMAC-функция от послания с использованием секретного
ключа
    hash = hmac.new(secret_key, message)
    digest = hash.digest()

    # 3. Пришедший ответ от клиента сравнивается с локальным результатом HMAC
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest, response)


# Выдержка из официальной документации, которая советует для сравнения
хэш-сумм
# использовать криптобезопасную функцию hmac.compare_digest, а не оператор ==
# ------------------------------------------------------------------------
# Warning:
# When comparing the output of digest() to an externally-supplied digest
# during a verification routine, it is recommended to use
# the compare_digest() function instead of the == operator
# to reduce the vulnerability to timing attacks.
# ------------------------------------------------------------------------


# ---------------------- Эхо-сервер ----------------------------------------
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'our_secret_key'

def echo_handler(client_sock):
    ''' Эхо-обработка.
        Проводит аутентификацию клиента и отсылает его же запрос обратно (эхо).
    '''
    if not server_authenticate(client_sock, secret_key):
        client_sock.close()
        return
```

```
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)


def echo_server(address):
    ''' Эхо-сервер.
        "Слушает" указанный адрес и общается с клиентом через echo_handler.
    '''
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)
    while True:
        conn, addr = s.accept()
        echo_handler(conn)


echo_server(('', 9999))
```

Листинг 2

```
# ========================= Аспекты безопасности
==============================

# ------------- Простая аутентификация клиента. Реализация клиента
-------------

import hmac
import os

def client_authenticate(connection, secret_key):
    ''' Аутентификация клиента на удаленном сервисе.
        Параметр connection - сетевое соединение (сокет);
        secret_key - ключ шифрования, известный клиенту и серверу
    '''
    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)


# ------------------------------ Клиент
--------------------------------------

from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'our_secret_key'

sock = socket(AF_INET, SOCK_STREAM)
```

```
sock.connect(('localhost', 9999))

client_authenticate(sock, secret_key)

sock.send(b'Hello, my secure server!')
resp = sock.recv(1024)
print('Сервер ответил: ', resp.decode())
```

Листинг 3

```
# ========================== Аспекты безопасности
==============================
# ------------- Модуль PyCrypto для криптографических функций в Питоне
--------

# ------------------------- Шифрование сообщений
------------------------------


# Библиотека PyCrypto реализует криптографические примитивы и функции на
Питоне.
# Однако данная библиотека не обновляется с 2014 года.
# PyCryptodome (PyCryptoDomeEx) - это fork библиотеки PyCrypto, развивается.
# Код проекта: https://github.com/Legrandin/pycryptodome

# Установка:  pip install pycryptodome

# PyCryptodome совместима по API с PyCrypto,
# PyCryptoDomeEx - дополняет/изменяет исходный API.


import os
from binascii import hexlify
from Cryptodome.Cipher import AES

# Для шифрования данных в PyCryptodome есть поддержка нескольких алгоритмов:
#   - блочные шифры: AES, DES, 3DES, Blowfish
#   - поточные шифры: Salsa20, ChaCha20

plaintext = b'The rain in Spain'


def padding_text(text):
    ''' Выравнивание сообщения до длины, кратной 16 байтам.
        В данном случае исходное сообщение дополняется пробелами.
    '''
    pad_len = (16 - len(text) % 16) % 16
    return text + b' ' * pad_len


def _encrypt(plaintext, key):
```

```
    ''' Шифрование сообщения plaintext ключом key.

        Атрибут iv - вектор инициализации для алгоритма шифрования.
        Если не задаётся явно при создании объекта-шифра, генерируется случайно.
        Его следует добавить в качестве префикса к финальному шифру,
        чтобы была возможность правильно расшифровать сообщение.
    '''
    cipher = AES.new(key, AES.MODE_CBC)
    ciphertext = cipher.iv + cipher.encrypt(plaintext)
    return ciphertext


def _decrypt(ciphertext, key):
    ''' Расшифровка шифра ciphertext ключом key

        Вектор инициализации берётся из исходного шифра.
        Его длина для большинства режимов шифрования всегда 16 байт.
        Расшифровываться будет оставшаяся часть шифра.
    '''
    cipher = AES.new(key, AES.MODE_CBC, iv=ciphertext[:16])
    msg = cipher.decrypt(ciphertext[16:])
    return msg


# Осуществим шифрование сообщения алгоритмом AES
# key (строка байтов) - секретный ключ для симметричного шифрования.
# Ключ должен быть длиной 16 (AES-128), 24 (AES-192) или 32 (AES-256) байта.
key = b'Super Secret Key'

# Длина сообщения должна быть кратна 16, поэтому выполним выравнивание.
plaintext = padding_text(plaintext)

# Выполним шифрование
cipher = _encrypt(plaintext, key)
print(hexlify(cipher))

# Выполним расшифрование
msg = _decrypt(cipher, key)
print(msg)
```

Листинг 4

```
# ========================= Аспекты безопасности
==============================
# ---------------------------- SQL-инъекции
----------------------------------

import sqlite3
import os

db_file = 'strong.sqlite3'
```

```
auth_data = {'admin':'21232f297a57a5a743894a0e4a801fc3',
             'user': 'ee11cbb19052e40b07aac0ca060c23ee',
             'guest':'084e0343a0486ff05530df6c705c8bb4'}

sql_create_user = """CREATE TABLE IF NOT EXISTS USER
               (id INTEGER PRIMARY KEY,
                login TEXT,
                password TEXT);"""

sql_insert = """INSERT INTO USER (login, password) VALUES (?, ?);
                """

def create_db(db_file):
    ''' Создание БД для демонстрации
    '''
    if os.path.exists(db_file):
        os.remove(db_file)

    conn = sqlite3.connect(db_file)
    curr = conn.cursor()

    curr.execute(sql_create_user)

    for login, password in auth_data.items():
        curr.execute(sql_insert, (login, password))

    conn.commit()
    conn.close()


def sql_injection_1(user_id):
    ''' Пример простой SQL-инъекции
    '''
    # Строка подобного рода уязвима к SQL-инъекциям:
    week_select_1 = "SELECT * FROM USER WHERE id = "
    week_select_1_2 = "SELECT * FROM USER WHERE id = {}"

    conn = sqlite3.connect(db_file)
    curr = conn.cursor()
    curr.execute(week_select_1 + (user_id))
    res = curr.fetchall()
    print('Результат первой уязвимой строки с запросом: ')
    print(res)

    curr.execute(week_select_1_2.format(user_id))
    res = curr.fetchall()
    print('Результат первой модифицированной строки с запросом: ')
    print(res)
    conn.close()
```

```
def sql_injection_2(user_id, user_login):
    ''' Попытка защититься, экранировав кавычками отдельные параметры запроса
    '''
    # Однако такая строка тоже может быть уязвима:
    week_select_2 = 'SELECT * FROM USER WHERE id = "{}" AND login = "{}"'

    conn = sqlite3.connect(db_file)
    curr = conn.cursor()
    curr.execute(week_select_2.format(user_id, user_login))
    res = curr.fetchall()
    print('Результат второй уязвимой строки с запросом: ')
    print(res)
    conn.close()


def sql_injection_3(user_id):
    ''' Простая SQL-инъекция с большими возможностями
    '''
    week_select_3 = "SELECT * FROM USER WHERE id = {}"

    conn = sqlite3.connect(db_file)
    curr = conn.cursor()
    curr.execute(week_select_3.format(user_id))
    res = curr.fetchall()
    print('Результат третьей уязвимой строки с запросом: ')
    print(res)

    conn.close()


# Сначала создадим БД для демонстрации:
create_db(db_file)

# Логическое выражение 1=1 всегда является истинным,
# поэтому SELECT выберет все данные из таблицы:
sql_injection_1('1 OR 1=1')

# В примере ниже будет сформирован запрос:
# SELECT * FROM Users WHERE id ="" or ""="" AND login ="" or ""=""
# Выражение or ""="" всегда истинно, поэтому запрос вернёт все записи из
таблицы
sql_injection_2('" or ""="', '" or ""="')


# Для некоторых СУБД может быть выполнен следующий запрос:
sql_injection_3('1; DROP TABLE USER;')
# Будет сформирован запрос:
# SELECT * FROM USER WHERE id = 0; DROP TABLE USER;
# который приведёт к удалению таблицы USER


# ----------------------------- Выводы
```

```
----------------------------------------
# Для защиты от SQL-инъекций стоит:
# 1. Использовать параметры в SQL-запросах, например:
#     week_select_1 = "SELECT * FROM USER WHERE id = ?;"
#     curr.execute(week_select_1, user_id)
# 2. Проводить фильтрацию пользовательского ввода до передачи в SQL-запрос
```

Листинг 5

```
# ========================= Аспекты безопасности
==============================
# --------------------- Примеры работы с функцией eval
------------------------

# Будьте осторожны с функцией eval:
eval("__import__('os').system('echo evil_eval BU-ga-ga')")


# Можно попробовать обезопасить себя, почистив __builtins__:
eval("__import__('os').system('echo evil_eval coming again')",
{'__builtins__':{}})


# Но это тоже можно обойти...
# Будьте аккуратны, код ниже
# приведёт к некорректному завершению работы интерпретатора
s = """
(lambda fc=(
    lambda n: [
        c for c in
            ().__class__.__bases__[0].__subclasses__()
            if c.__name__ == n
        ][0]
    ):
    fc("function")(
        fc("code")(
            0,0,0,0,0,b"BOOOOM",(),(),(),"","",0,b""
        ),{}
    )()
)()
"""
eval(s, {'__builtins__':{}})
```

Листинг 6

```
# ========================= Аспекты безопасности
==============================
# ---------------------- Примеры работы с модулем pickle
----------------------
```

```python
# Выдержки из официальной документации, которые говорят,
# что модуль является не вполне безопасным решением для передачи объектов по
сети.

#
------------------------------------------------------------------------------
# https://docs.python.org/3.6/library/pickle.html
# Warning! The pickle module is not intended to be secure against erroneous
# or maliciously constructed data. Never unpickle data received
# from an untrusted or unauthenticated source.

# https://docs.python.org/3.6/library/multiprocessing.html
# Warning
# The Connection.recv() method automatically unpickles the data it receives,
# which can be a security risk unless you can trust the process which sent the
message.
# Therefore, unless the connection object was produced using Pipe()
# you should only use the recv() and send() methods after performing
# some sort of authentication.
#
------------------------------------------------------------------------------


# Модуль pickle служит для сохранения Python-объектов
(сериализация/десериализация)
import pickle

# Однако при десериализации не проверяется содержимое внутренностей объекта.
# Строка ниже выполнит системную функцию echo:
pickle.loads(b"cos\nsystem\n(S'echo I am Evil Pickle-module!''\ntR.")


#
------------------------------------------------------------------------------
# А что, если передать pickle-объект по сети? Хорошая идея!

import subprocess
import socket

# Другой вариант - создать свой класс,
# метод __reduce__ которого должен будет осуществлять десериализацию
class EvilPayload:
    """ Функция __reduce__ будет выполнена при распаковке объекта
    """
    def __reduce__(self):
        """ Запустим на машине клиента безобидный Notepad (или другой редактор)
        """
        import os
        os.system("echo You've been hacked by Evil Pickle!!! > evil_msg.txt")
```

```
        # with open('evil_msg.txt', 'w') as f:
        #     f.write("You've been hacked by Evil Pickle!!!")

        return (subprocess.Popen, (('notepad','evil_msg.txt'),))


# Реализуем простой сокет-сервер для демонстрации примера.
# Клиентское приложение находится в файле evil_pickle_client.py
def evil_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(("localhost", 9999))
    print('Зловещий сервер запущен...')
    sock.listen()
    conn, addr = sock.accept()
    print('К нам попался клиент', addr)

    print('Отправляем ему "троянца"...')
    # Отсылаем опасный объект "доверчивому" клиенту
    conn.send(pickle.dumps(EvilPayload()))

evil_server()


# Некоторые рекомендации по безопасному использованию модуля pickle
# 1. По возможности, шифруйте сетевой трафик (SSL/TLS).
# 2. Если шифрование невозможно, пользуйтесь электронной подписью для
подтверждения данных.
# 3. Если pickle-данные сохраняются на диск, убедитесь, что только доверенные
процессы могут менять эти данные.
# 4. По возможности, избегайте модуля pickle. Воспользуйтесь, например, JSON.
```

Листинг 7

```
# ========================= Аспекты безопасности
==============================

# ----- Простой сокет-клиент для демонстрации работы с pickle-данными
---------

import pickle
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("localhost", 9999))

# Получаем опасное сообщение
message = sock.recv(1024)

# Распаковываем, "радуемся" - нас взломали...
pickle.loads(message)
```

```
sock.close()
```