Научно-исследовательская работа (2 семестр)

Задание 2.2. ВСР

Фатьянов М.А.

**Основы кибербезопасности**

Роль шифрования и SSL/HTTPS: современные протоколы защищённой передачи (HTTPS) базируются на TLS/SSL, который обеспечивает три ключевых сервиса безопасности – аутентификацию сторон, шифрование и контроль целостности передаваемых данных. При установке HTTPS-соединения клиент проверяет сертификат сервера (аутентификация), после чего все данные шифруются до отправки по сети. Это препятствует перехвату и чтению трафика посторонними. Одновременно TLS гарантирует целостность: даже единичное изменение в переданных данных будет обнаружено. Иными словами, HTTPS превращает HTTP в «шифрованный HTTP» – данные проходят по порту 443 в зашифрованном виде, что защищает от атак «человек посередине» и подмены содержимого.

Симметричное (AES) и асимметричное (RSA) шифрование: при симметричном шифровании для шифрования и расшифровки используется один и тот же секретный ключ. Классическим примером является AES (Advanced Encryption Standard) – блочный алгоритм с размером блока 128 бит и ключами длиной 128, 192 или 256 бит. AES выбран стандартом NIST и широко используется для быстрого шифрования больших объёмов данных. Асимметричное шифрование (RSA) использует пару ключей – публичный и приватный. Публичный ключ можно открыть всем, чтобы зашифровывать сообщения, а приватный хранится в секрете у владельца и служит для расшифровки. В RSA (алгоритм Ривеста–Шамира–Адлемана) сообщения шифруются публичным ключом и могут быть расшифрованы только соответствующим приватным ключом. Из-за высокой вычислительной сложности RSA обычно применяют для передачи небольших объёмов данных (например, ключей сессии), а для шифрования «тяжёлых» потоков используют симметрию (например, сначала шифруют данные AES, а ключ AES передают через RSA).

Структура HTTPS-сервера и настройка SSL-сертификата: HTTPS-сервер – это веб-сервер, который слушает порт 443 и использует TLS для защиты соединений. Серверу необходим SSL-сертификат, содержащий его публичный ключ. При старте соединения сервер отправляет клиенту свой сертификат; клиент проверяет его подпись (у доверенного центра сертификации или самоподписанной авторитетной цепочки) и устанавливает общую сессию. Например, для тестового сервера можно создать самоподписанный сертификат утилитой OpenSSL:

openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem

. В Python-библиотеке ssl такие сертификаты подключаются к сокету через SSLContext. Пример настройки сервера на Python:

context = ssl.SSLContext(ssl.PROTOCOL\_TLS\_SERVER)

context.load\_cert\_chain('cert.pem', 'key.pem')

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

sock.bind(('localhost', 8443))

sock.listen(5)

with context.wrap\_socket(sock, server\_side=True) as ssock:

conn, addr = ssock.accept()

conn.sendall(b'Hello over HTTPS!')

. Этот сервер будет отвечать по HTTPS, шифруя трафик алгоритмами из SSL/ TLS. При настройке важно убедиться, что сертификат и приватный ключ правильно загружены и порт соответствует 443 (или 8443 для тестов).

Анализ сетевого трафика, Wireshark и Scapy: Wireshark – открытый и бесплатный протокол-анализатор, позволяющий захватывать и интерактивно просматривать сетевой трафик. С его помощью можно видеть заголовки пакетов TCP/IP, HTTP, TLS и т.д., анализировать, какие соединения устанавливаются (например, по портам 80/443). Wireshark умеет фильтровать и «дешифровать» трафик, если известны ключи сессии. Scapy – Python-библиотека для работы с пакетами на низком уровне. С помощью Scapy можно генерировать произвольные пакеты, снимать трафик (sniff), парсить полученные пакеты. Например, команда sniff(filter="tcp port 443", count=10) в Scapy соберёт первые 10 пакетов HTTPS-трафика. Scapy полезна для программного анализа и автоматизации задач, дополняя возможности графического Wireshark. Оба инструмента – бесплатные (Wireshark GPL, Scapy open-source) и хорошо документированы.

**Источники и ресурсы для самостоятельного изучения**

Официальная документация Python (модуль ssl для TLS/SSL-сокетов) и руководство по сети (модуль socket, http.server).

Статьи MDN Web Docs о безопасности веба: например, раздел *Transport Layer Security*

Официальная документация библиотеки cryptography (руководство по AES, RSA, Fernet).

Официальная документация OpenSSL – раздел *openssl-req* (генерация сертификатов).

Руководство пользователя Wireshark и Wiki (официальный сайт Wireshark) – общие сведения о захвате и анализе пакетов.

Документация Scapy на Read the Docs (введение, функции sniff, формирование пакетов).

Бесплатная книга *“Crypto 101”* Лауренса ван Хоутвена (введение в криптографию) и набор задач Cryptopals (практические криптозадачи).

Бесплатные онлайн-курсы по кибербезопасности: например, курс MIT 6.858 «Computer Systems Security» (лекции на YouTube), курсы от Coursera/edX по основам безопасности, видеоматериалы SharkFest (Wireshark) и др.

**Лабораторная работа (практическое задание)**

1. HTTPS-сервер на Python: написать HTTP-сервер на Python (например, с помощью http.server или socket), «обернуть» его SSL/TLS-контекстом с самоподписанным сертификатом. То есть сгенерировать ключ и сертификат OpenSSL (например, openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem), а затем в коде загрузить их через ssl.SSLContext или ssl.wrap\_socket. Сервер должен принимать соединения по HTTPS и, например, возвращать простой текстовый ответ.
2. Шифрование/дешифровка строки (AES и RSA): используя библиотеку cryptography, реализовать шифрование и расшифровку. Для AES можно использовать высокоуровневый рецепт Fernet (который под капотом использует AES-CBC+HMAC), пример использования:

from cryptography.fernet import Fernet

key = Fernet.generate\_key()

cipher = Fernet(key)

token = cipher.encrypt(b"секретное сообщение")

print(cipher.decrypt(token)) # раскодирует обратно

. Для RSA – сгенерировать пару ключей (public/private) через rsa.generate\_private\_key, затем зашифровать короткое сообщение публичным ключом и расшифровать приватным (с помощью паддинга OAEP). Пример генерации ключа:

from cryptography.hazmat.primitives.asymmetric import rsa

private\_key = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048)

public\_key = private\_key.public\_key()

, пример шифрования/дешифровки с OAEP:

from cryptography.hazmat.primitives import padding, hashes

ciphertext = public\_key.encrypt(b"msg", padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None))

plaintext = private\_key.decrypt(ciphertext, padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None))

1. Захват и анализ трафика (Wireshark + Scapy): запустить HTTPS-сервер (п.1) и клиентское соединение к нему. С помощью Wireshark начать захват на локальном интерфейсе, применить фильтр по порту 8443 (или 443) – увидеть пакеты TCP и TLS. Отметить пакеты ClientHello, ServerHello, записать использованные алгоритмы шифрования. Попытаться посмотреть содержимое – без ключа дешифрования оно не видно. Затем использовать Scapy в режиме сниффинга. Например:

from scapy.all import sniff

packets = sniff(filter="tcp port 8443", count=5)

for pkt in packets:

print(pkt.summary())

– это покажет заголовки захваченных пакетов HTTPS. Цель – понять, как анализируется зашифрованный трафик и какие метаданные при этом видны.

**Инструкции и методические рекомендации**

Настройка сертификата: при генерации самоподписанного сертификата студенты должны указать все поля (CN – например, localhost). Если сертификат не совпадёт с именем хоста, браузер/клиент выдаст предупреждение или ошибку. Для простоты тестирования можно снимать проверку сертификата на клиентской стороне. При использовании Python-скрипта важно передавать правильные пути к файлам cert.pem и key.pem в SSLContext.load\_cert\_chain().

Код HTTPS-сервера: при обёртывании сокета через ssl.wrap\_socket или SSLContext важно указать server\_side=True. Учителю следует напомнить, что после wrap\_socket сам сокет меняет интерфейс – операции accept() и чтение/запись производятся уже на SSL-сокете. Частая ошибка – забыть вызвать sock.listen() или accept(). Также надо убедиться, что скрипт запущен с достаточными правами (привилегия на привязку к порту и т.д.). При тестировании можно использовать порт >1024 (например, 8443), чтобы не требовать админских прав.

Шифрование AES/RSA: при шифровании в AES (Fernet) обращайте внимание, что метод encrypt и decrypt работают с байтовыми строками (типа b"..."). Если шифруем текст, нужно его конвертировать в байты. В случае RSA важно совпадение схемы padding при шифровании и дешифровании – здесь применяется OAEP с SHA-256. Задайте студентам ключ фиксированного размера (минимум 2048 бит) и продемонстрируйте, что слишком короткие сообщения (больше чем модуль RSA) не шифруются. Показать пример: если попробовать шифровать очень длинную строку RSA, библиотека выдаст ошибку о переполнении.

Работа с Wireshark: объяснить студентам, как выбрать правильный сетевой интерфейс и как использовать строку фильтра (напр., tcp.port == 8443 или tls) для поиска нужного трафика. Подчеркнуть, что без приватных ключей TLS-сессии данные останутся зашифрованными (будет видно только заголовки TCP и часть TLS handshake). Частая путаница – интерпретация «шифрованных» пакетов как мусора; обратить внимание, что в Wireshark они помечены как TLS Record. Также напомнить о возможных проблемах: если у Wireshark не установлены нужные библиотечные пакеты (WinPcap/Npcap), он не сможет перехватывать пакеты на Windows.

Scapy: при запуске скриптов сниффинга (sniff) в Scapy обычно требуются повышенные привилегии (особенно на Windows/Linux – запускать от администратора или с sudo). Если студенты получают пустые результаты – проверить параметры фильтра (они чувствительны к синтаксису BPF) и права. Также Scapy видит только тот трафик, который проходит через выбранный интерфейс; на локальном соединении иногда требуется захват трафика «собственного компьютера» (loopback), что не всегда просто сделать.