# ТЕХНИЧЕСКИЕ НАУКИ

УДК 004.43

## Утилита сетевого взаимодействия на Python

***Н.М. Кодацкий, В.В. Галушка***
Донской государственный технический университет (г. Ростов-на-Дону, Российская Федерация)

**Аннотация.** В настоящее время существует необходимость в эффективном использовании инструментов для решения задач сетевого взаимодействия на языке программирования Python, так как в среде разработки Python отсутствуют нативные средства работы с протоколами сетевых уровней и часто требуется написание кода с низкоуровневыми функциями для обеспечения сетевой связности. Цель данной статьи являлась задача разработки утилиты сетевого взаимодействия на языке программирования Python, которая позволит упростить процесс работы с протоколами сетевых уровней и повысить производительность программного обеспечения за счет оптимизации работы функций сетевого взаимодействия.

***Ключевые слова:*** python, sockets, TSP, UDP, сетевое взаимодействие.

## Network Interaction Utility in Python

***Nikita M Kodatskiy, Vasiliy V Galushka***
Don State Technical University (Rostov-on-Don, Russian Federation)

**Abstract.** Currently, there is a need to effectively use tools for solving network interaction problems in the Python programming language, since the Python development environment lacks native tools for working with network layer protocols and often requires writing code with low-level functions to ensure network connectivity. The work objective is to develop a network interaction utility in the Python programming language, which will simplify the process of working with network layer protocols and improve software performance by optimizing the operation of network communication functions.

**Keywords:** python, sockets, TSP, UDP, networking.

**Введение.** Любому специалисту по защите информации или системному администратору неизбежно приходится взаимодействовать с сетевыми коммуникациями. Корпоративная сеть всегда является самым привлекательным местом для злоумышленников. Через нее злоумышленникам доступны возможности сканирования системы, внедрение сетевых пакетов, анализ трафика и эксплуатация удаленных узлов. Часто системными администраторами в целях повышения безопасности вырезаются утилиты сетевого взаимодействия, такие как NetCat, Wireshark, Nmap и другие [1]. Такое решение способно ограничить возможности злоумышленников при их проникновении в корпоративные сети. Но в любой системе на базе Linux пользователю доступен интерпретатор Python [2]. В таком случае открывается целое множество инструментов для создания собственных утилит. Используя язык Python, программисту доступен модуль sockets. Данный модуль позволяет создавать клиентов и серверов, взаимодействующих по TCP и UDP. Грамотное использование модуля позволит произвести или поддержать несанкционированный доступ к атакуемой машине, что является прямой угрозой информационной безопасности предприятия [2, 3].

Разработка утилиты сетевого взаимодействия на Python актуальна для решения практических задач, таких как построение сетевых приложений, систем управления базами данных и прочих задач, связанных с сетевым взаимодействием [2]. Понимание принципов работы анализаторов сети может помочь сетевым администраторам в мониторинге инцидентов информационной безопасности и их перекрытию [4].

Целью работы является описание базового принципа работы с сетью с использованием sockets на языке Python. Результатом работы является разработанная программная оболочка для анализа сети предприятия.

**Архитектура клиент – сервер.** Технология «клиент-сервер» предназначена для распределения нагрузки между запрашивающими (клиентами) и теми устройствами, которые предоставляют ответы на запросы (сервера). Такими устройствами выступают программное обеспечение на одной или нескольких вычислительных машинах, которые взаимодействуют через сеть. Принципиальной разницы между клиентом и сервером нет, их различает выполняемая ими роль в процессе коммуникации [5].

Клиент — конечная рабочая станция, задачей которой является отправка запросов на получение определенной информации или решение какой-либо задачи серверу.

Сервер — это средство вычислительной техники, которое взаимодействует со всеми подключенными к нему клиентами и предоставляет им необходимые данные.

Сеть — структурная единица, осуществляющая сетевой обмен данными между отдельными устройствами с общими ресурсами.

Приложение — программное обеспечение, благодаря которому происходит обработка данных и обеспечение физического распределения между структурными элементами.

Сервер выполняет работу в многопользовательском режиме, автоматически настраивая приоритет по очереди запросов или других параметров. Такая архитектура обеспечивает быструю обработку запросов, чтобы приоритет или очереди становились незаметны множеству подключенных клиентов. С помощью сетевых протоколов осуществляется обмен данными между узлами сети [5].

Различают 4 типа клиент-серверной архитектуры:

1. Одноуровневая, в ней ПО распределяется по отдельным рабочим станциям, работающей с одним сервером. Дополнительные ПО не используются, а сервер всего лишь предоставляет сведения на запросы. Такая архитектура проста, но сложна в администрировании при большом количестве клиентов, также необходимы средства синхронизации.

2. Двухуровневая содержит прикладные программы на выделенном сервере программного обеспечения. Интерфейсы взаимодействия (программы-клиенты) находятся на рабочих станциях. Различают как тонкий клиент и толстый сервер, так и толстый клиент и тонкий сервер. Такая архитектура удобна и проста в модификации, масштабировании и является высокопроизводительной. При большом количестве клиентов следует увеличивать и мощности сервера.

3. Трехуровневая предназначена для обслуживания сервера приложений, включает в себя множество разновидностей серверного аппаратного обеспечения. Связь клиента с базой данных происходит через специальное промежуточное ПО сервера приложений. Такой подход позволяет добиться целостности потока, хорошего уровня защиты базы данных от несанкционированного доступа, но наличие промежуточного ПО значительно усложняет структуру взаимодействия.

4. Многоуровневая состоит из ряда серверов приложений, использующих результаты данных друг друга и сторонних серверов. Такая система обеспечивает повышенную гибкость, но значительно усложняет архитектуру и процесс взаимодействия.

**Конечная точка связи.** Перед тем как сервер отработает запрос клиента, обеим сторонам необходимо выполнить ряд подготовительных действий [3]. Для этого необходимо создать конечную точку связи (сокет). Через сокет сервер отслеживает запрос от клиента. Существует множество сетевых точек связи для обработки различных типов сообщений.

Unix сокеты — вид связи между двумя процессами, отображаемый в виде файла. Файл может использоваться для быстрого установления соединения между процессами без накладных расходов [3]. Unix Domain Sockets — сокеты для локальной сети, находящиеся на одном компьютере. Они используются для передачи данных между программами без использования сетевых интерфейсов. Файлы сокетов Unix не записывают данные на диск, они хранятся в памяти ядра. Они являются ссылкой на сокет и дают разрешения файловой системе для управления доступом.

Поскольку сокеты являются ничем иным как просто каналом внутри ядра, они используют транспортные протоколы для передачи данных (TCP и UDP). Другие протоколы, такие как FTP, SMTP и RDP, используются на более высоком уровне. Они предоставляют инфраструктуру для передачи данных. Сокеты на основе TCP называются потоковыми сокетами, куда все данные будут поступать по порядку, а на основе UDP — это сокеты для дейтаграмм, для которых порядок (или даже доставка) не гарантируется. Необработанные (raw) сокеты не

имеют ограничений и используются для реализации записей программ, например, Wireshark [4]. При любом подключении используются сокеты, а обычно подключение происходит к <IP-host_addres>:<port> из удаленной системы. Когда речь идет о подключение через Rest API, сокеты могут используются для ускорения соединения [4].

**Использование Python**. Для создания сокета в Python используется функция socket(socket_family, socket_type, protocol=0, где socket_family, которая может принимать значения AF_UNIX или AF_INET, socket_type - SOCK_STREAM или SOCK_DGRAM, а протокол обычно опускается и по умолчанию считается 0. Приведу алгоритмы создания TCP-клиента и TCP-сервера [2].

1. TCP-клиент. Для начала подключим библиотеку для работы с сокетами. Определим интересующий хост и порт. После чего создадим сокет с параметрами семейства сокета и его типом. AF_INET значит, что мы будем использовать стандартный адрес IPv4 или сетевое имя, а SOCK_STREAM, что мы используем протокол TCP для передачи данных. Затем осуществим подключение к серверу, отправив ему GET-запрос с данными в байтах. В результате получим ответ от сервера, распечатав это сообщение в консоли. По завершению сеанса сокет необходимо закрыть.

Такой алгоритм действий допускает, что соединение всегда остается стабильным и сервер ждет от клиента отправку данных, после чего своевременно их высылает клиенту. Программисты редко добавляют тонкости работы с блокирующими сокетами, исключениями и т. д. в инструменты, написанные для сбора данных или эксплуатации удаленных компьютеров. Часто такие тонкости в задачах администрирования и тестирования на уязвимости не пригождаются [3].

2. TCP-сервер. Используя язык Python для написания сервера, алгоритм создания сокетов примерно такой же, как и на клиенте. Написание TCP-сервера может быть полезно в задачах реализации командных оболочек или прокси-серверов [2]. Такой сервер логично будет реализовать в многопоточном режиме.

Подгрузим необходимые модули и зададим IP-адрес и порт, который будет прослушивать сервер. Напишем основной код, в котором сервер будет выполнять прослушивание подключений, указав, что отложенных соединений должно быть не больше какого-то ограниченного лимита (объема очереди). При переполнении этой очереди поступающие запросы будут откидываться или игнорироваться (в зависимости от операционной системы), пока не освободится новое место в буфере. В бесконечном цикле сервер ожидает входящие соединения. При подключении клиента, мы получаем клиентский сокет и подробности об удаленном соединении. Создаем поток, указывающий на функцию *handleclient*, передаем функции клиентское соединение. Главный цикл сервера освобождается для обработки следующего входящего соединения. Функция *handleclient* выполняет вызов *recv()*, затем возвращает клиенту сообщение. Данное взаимодействие может иметь следующий вид, как на рис. 1.

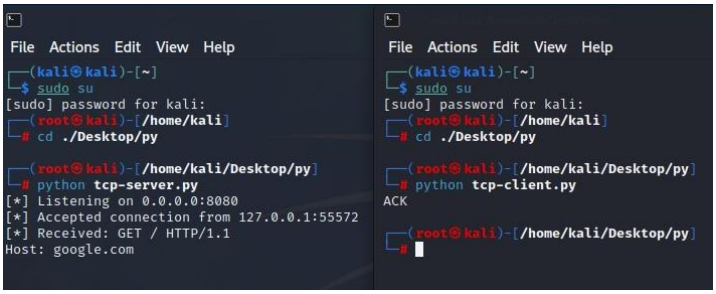


Рис. 1. Вывод при подключении клиента к серверу

**Разработанная утилита сетевого взаимодействия**. Далее приведу структуру создания программной оболочки для сетевого взаимодействия. Такой инструмент всегда может быть написан злоумышленником, когда он попадет внутрь корпоративной сети и не обнаружит известных программ для сетевого администрирования [3].

Для создания утилиты на Python хорошим решением будет использовать объектно-ориентированный подход. Для этого опишем два класса: *Server*, представляющий из себя TCP-сервер и *Client* для реализации клиента.

Для начала разберем более подробно класс *Client.* В нем проинициализируем объект экземпляра данного класса такими полями: адрес и порт целевого хоста; новым экземпляром сокета, потоком для чтения данных, флагом для определения текущего состояния соединения, и флаг для индикации того, что хост был отключен. Опишем метод *run()*. В нем будем выполнять следующие действия:

– устанавливать соединение с указанным хостом и портом;

– запускать новый поток, читающий данные из сокета;

– в цикле с прерыванием на флаг выполнять чтение пользовательского ввода, отправки сообщений на сервер и проверки наличия отправленных данных. Если отправленных данных не было, то выводить ошибку об завершении соединения и закрытию сокета;

– обработать ошибки;

– по окончанию действий закрыть сокет и завершить вторичный поток чтения данных.

Добавим также закрытый метод *__reader()* в отдельном потоке, который будет выполнять следующие действия:

– в бесконечном цикле получать данные из сокета;

– использовать функцию *select.select* для проверки готовности сокета к чтению;

– если данные получены, они будут добавляться в строковый буфер;

– если сокет закрыт, устанавливается флаг *self.__stop*;

– если буфер содержит данные, они выводятся на экран;

– при возникновении ошибки выводится сообщение об ошибке;

– если *self.__target_disconnect* установлен, выводится предупреждение о завершении работы удаленной машины и вызывается прерывание входного потока.

Разберем логику работы класса *Server*. Для его инициализации опишем конструктор со следующими полями: новый экземпляр сокета, адрес и порт целевого хоста, объект переданных обработчиков, флаг текущего состояния соединения. Создадим метод *listen()*, который запускает сервер на заданном хосте и порту и прослушивает соединения, обрабатывая входящие запросы следующим образом:

– выводится сообщение о начале прослушивания на указанном адресе и порту;

– устанавливается параметр SO_REUSEADDR для сокета, чтобы обеспечить быстрое повторное использование адреса и порта после закрытия сокета;

– привязывается сокет к указанному адресу и порту;

– устанавливается максимальная длина очереди ожидающих соединений в 5;

– в бесконечном цикле принимаются новые входящие соединения;

– для каждого входящего соединения создается отдельный поток, который обрабатывает входящие данные и отправляет ответы;

– при возникновении ошибки выводится сообщение об ошибке;

– при получении прерывания клавиатуры выводится сообщение о завершении работы сервера;

– в конце закрывается сокет.

Добавим закрытый метод *__handle()*, который обрабатывает входящие сообщения следующим образом:

– установка флага *close*;

– для каждого обработчика, переданного в конструкторе объекта, вызывается инициализационный метод;

– в бесконечном цикле читаются данные из сокета;

– используется функция *select.select* для проверки готовности сокета к чтению;

– полученные данные добавляются в массив байтов *raw_buffer*;

– если в *raw_buffer* есть данные, для каждого обработчика вызывается метод *handle_msg*, который обрабатывает входящие данные. Если после обработки данных необходимо завершить соединение, устанавливается флаг *close* в значение True;

– при возникновении ошибки выводится сообщение об ошибке;

– при получении ошибки *BrokenPipeError* выводится сообщение о закрытии соединения;

– в любом случае соединение закрывается.

Далее остается только реализовать основную функцию, которая будет просматривать переданные аргументы и запускать соответствующий класс. Добавить интерфейс пользователя консольного приложения.

Проведем тестирование разработанной утилиты. Через командную оболочку запустим разработанный скрипт с аргументом –help и увидим информацию о использовании разработанного инструмента (рис. 2).

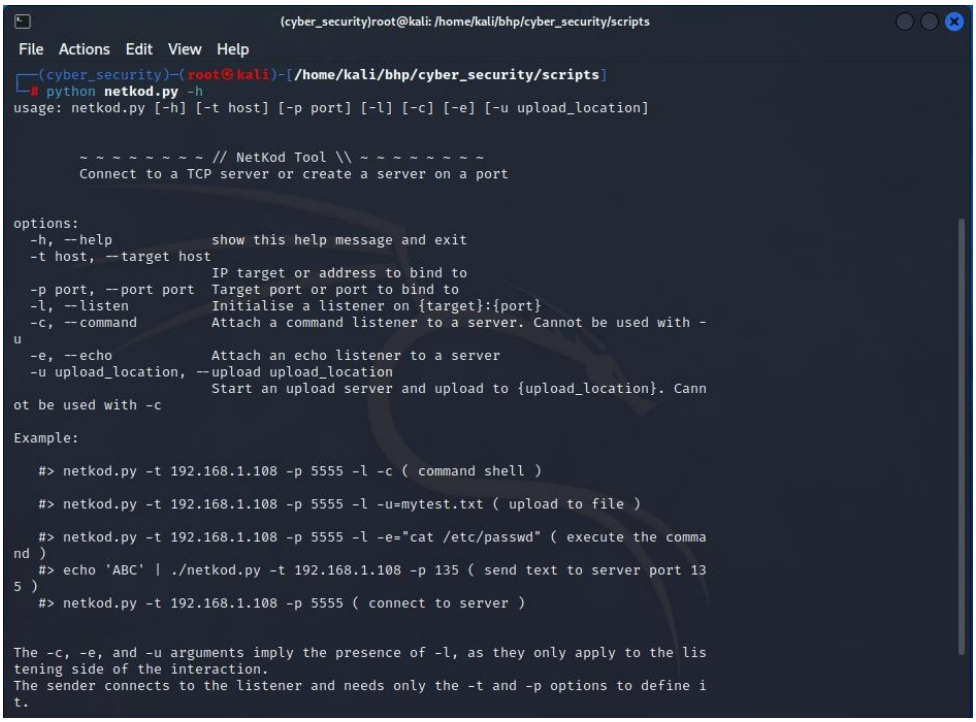


Рис. 2. Вывод информации об инструменте

Теперь запустим слушатель с использованием локального IP-адреса и порта 1234, чтобы предоставить доступ к оболочке (рис. 3):

# python netkod.py -t 127.0.0.1 -p 1234

После чего откроем новый терминал в системе и запустим netcat на том же хосте и порте используя команду:

# nc 127.0.0.1 1234

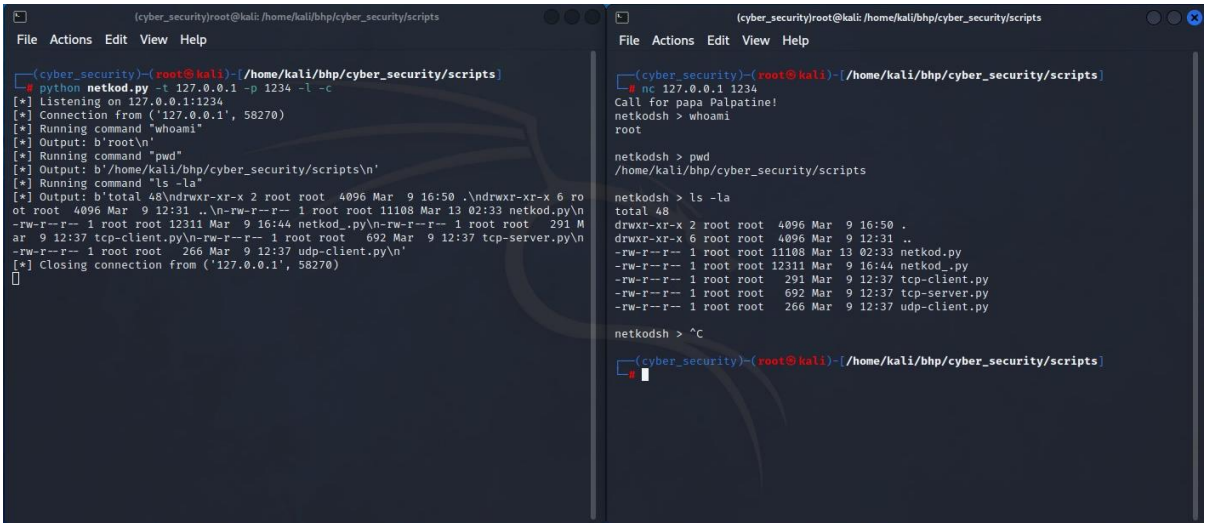


Рис. 3. Перехват трафика через слушатель TCP-сервера

**Заключение.** Один из важных аспектов информационной безопасности, который важно понимать, что сама безопасность — сложная тема для обычных пользователей [1]. На средства проникновения и тестирования уязвимостей обращают особое внимание, поскольку они практически не предоставляют ограничений в возможностях работы с сетью предприятия. Уязвимости присутствуют в каждой системе, а единственным способом их сокращения является привлечение специалистов по тестированию на проникновения для обнаружения слабых мест системы и их перекрытию [5]. Таким образом, данную утилиту можно использовать для установления тестовых соединений, отладки сетевых устройств и приложений, а также передачи файлов, что будет полезно при администрировании сети.

**Библиографический список**

1. Penetration Testing using Linux Tools: Attacks and Defense Strategies. *International Journal of Engineering Research & Technology (IJERT).* 2016;5(12):153–158. URL: https://www.ijert.org/research/penetration-testing-using-linux-tools-attacks-and-defense-strategies-IJERTV5IS120166.pdf (дата обращения 20.02.2023).
2. Rejah Rehim. Pentesting with Python and Linux: Strategies and Techniques. Packt Publishing; 2017. 226 p.
3. Michael W.L. Linux Pentesting Cookbook: Practical Techniques for Securing Your Network. Packt Publishing; 2017. 216 p.
4. Introduction to Pentest on Linux. URL: https://hub.packtpub.com/introduction-penetration-testing-and-kali-linux/ (accessed 20.02.2023).

*Об авторах:*

**Кодацкий Никита Максимович**, магистрант кафедры «Вычислительные системы и информационная безопасность» Донского государственного технического университета (344029, РФ, г. Ростов-на-Дону, ул. Страны Советов, 1), nickitadatsky@gmail.com

**Галушка Василий Викторович**, доцент кафедры «Вычислительные системы и информационная безопасность», кандидат технических наук Донского государственного технического университета (344029, РФ, г. Ростов-на-Дону, ул. Страны Советов, 1), galushkavv@yandex.ru

*About Authors:*

**Nikita M Kodatskiy**, Master's degree student of the Computer Systems and Information Security Department, Don State Technical University (1, Strany Sovetov str., Rostov-on-Don, 344029, RF), nickitadatsky@gmail.com

**Vasiliy V Galushka**, associate professor of the Computer Systems and Information Security Department, Don State Technical University (1, Strany Sovetov str., Rostov-on-Don, 344029, RF), Csnd. Sci. (Eng.), galushkavv@yandex.ru