

# Graficzny interfejs manipulatora pakietów

[Wprowadzenie](#)

[Cel](#)

[Motywacja](#)

[Droga rozwoju](#)

[Narzędzia i technologie](#)

[Harmonogram](#)

[Opis projektu](#)

[Protokoły](#)

[IPTables](#)

[Scapy](#)

[PyQt](#)

[Schemat działania](#)

[Qt](#)

[TCP](#)

[UDP](#)

[Instrukcja obsługi](#)

[Cheat sheet](#)

[Przykłady](#)

# Wprowadzenie

Projekt realizowany w ramach zajęć “Podstawy Telekomunikacji” pod nadzorem mgr Przemysława Walkowiaka.

## Cel

Celem projektu było utworzenie interfejsu, który umożliwi edycje przechwyconych pakietów oraz ich odesłanie w spreparowanej formie. Domyślnym systemem, pod którym interfejs graficzny był testowany to Kali Linux. Z racji wykorzystywanych technologii, używanie programu pod systemem Windows jest niemożliwe - IPTables nie występuje na systemach rodziny Windows. Dodatkowo obsługę IPTables oparliśmy na wywoływaniu funkcji z powłoki systemu.

## Motywacja

Temat został wybrany ze względu na chęci rozszerzenia umiejętności programowania w języku Python oraz rozszerzenia znajomości systemu Linux. Dodatkowo dał nam możliwości analizy oraz nauki wykorzystywanych protokołów takich jak TCP, UDP, ICMP, IP w sposób praktycznie i namacalny.

## Droga rozwoju

Program, który zdajemy w pełni obsługuje protokoły UDP, TCP, IP oraz ICMP. W przyszłości można zaimplementować obsługę również innych protokołów takich jak Ethernet, ARP, oraz wielu innych, które są bardzo popularne.

Dodatkowo w programie zostały zawarte pewne szablony odpowiedzi. Na przykład możemy odpowiedzieć automatycznie na zapytanie TCP - “Three way handshake”, które może nas połączyć chociażby z gniazdkiem TCP klienta, który się łączy. Jest bardzo dużo różnych wzorców, które mogłyby być rozpoznane oraz zaimplementowane - dzięki temu można pomijać pakiety, które nie są dla nas istotne, a są standardem funkcjonowania danego protokołu. Dzięki takim funkcjom program nadawałby się doskonale do analizy programów oraz systemów. Ciekawą funkcjonalnością również byłoby wsparcie pełnego fuzzingu pakietów.

Poza szablonami programowymi przydatne mogłyby okazać się szablony predefiniowane za pomocą list. Wtedy pakiet, który przychodzi byłby dodawany do listy na nasze życzenie. Następnie byłby przez nas modyfikowany i zapisywany jako szablon na stałe. Wtedy każdy kolejny nadchodzący pakiet, byłby obsługiwany w ten sam sposób. Użytkownik określałby, które pola muszą być identyczne żeby został obsłużony w ten sposób. Pomysł ten, eliminuje manualną ingerencję użytkownika w strukturę pakietu oraz konieczność programowania automatycznych odpowiedzi.

# Narzędzia i technologie

Scapy - Jest to interaktywne narzędzie do manipulacji pakietami. Za jego pomocą można fałszować, dekodować pakiety z wielu protokołów, wysyłać je, przechwytywać, dopasowywać wymagania i wysyłać pakiet zwrotny. Scapy jest napisany w języku Python, najlepiej funkcjonuje w wersji 2.7. Strona internetowa narzędzia - <https://github.com/phaethon/scapy>.

Python - Jest to wysokopoziomowy, dynamiczny język programowania. Typy w pythonie są dynamiczne, zaś pamięć jest zarządzana automatycznie. Zyskał popularność dzięki rozbudowanym bibliotekom standardowym oraz prostocie. Strona internetowa języka - <https://www.python.org/>.

PyQt - Wieloplatformowe, bardzo rozbudowane narzędzie, które pozwoliło nam m.in. na budowę interfejsu graficznego programu. PyQt.Gui - przestrzeń nazw, w której znajdują się wszystkie interesujące nas komponenty. Zawiera szereg klas, które obsługują tworzenie okien, przycisków, suwaków, pól tekstowych, tabel i wielu innych. PyQt zawiera również szereg klas związanych z gniazdkami, plikami, etc. Jednakże nie było nam to potrzebne. Opis platformy programistycznej - <https://wiki.python.org/moin/PyQt>.

PyCharm - IDE pod którym tworzyliśmy projekt, w wersji darmowej "Community". Wybrane ze względu na przyjemny interfejs, uzupełnianie kodu, podkreślanie błędów i wiele innych funkcji wspomagających produktywność. Strona IDE - <https://www.jetbrains.com/pycharm/>.

IPTables - Jest to program sterujący filtrem pakietów. Wymaga jądra skompilowanego z ip\_tables. Funkcją wykorzystywaną była NFQUEUE, która razem ze Scapy tworzy moduł dzięki któremu można edytować pakiety. Może być również stosowany jako zaporę ogniową systemu. Dla nas służył jako filtr - każdy pakiet spełniający zdefiniowaną regułę trafiał do NFQUEUE. Artykuł opisujący użycie nfqueue, iptables oraz scapy - <https://5d4a.wordpress.com/2011/08/25/having-fun-with-nfqueue-and-scapy/>.

C++ Qt - Jest to odpowiednik silnika Qt dla języka C++. W nim napisana została prosta aplikacja, która korzysta z gniazdek TCP oraz UDP. Dzięki niej mogliśmy obserwować zmieniające się pakiety edytowane przez aplikację manipulatora.

Qt Creator - IDE dla silnika Qt. Przejrzyste narzędzie, która usprawnia pracę, dodatkowo wsparte przez debugger oraz kompilator Visual Studio 2015.

# Harmonogram

Poniżej w tabeli znajduje się opis prac wykonanych w trakcie powstawania projektu. Prace nie są rozdzielone na osobę - każda z funkcjonalności była opracowywana wspólnie (mniej lub bardziej).

Zawartość tabeli potwierdza historia projektu na GitHub,  
<https://github.com/bojakowsky/OnTheFlyPacketManipulator>.

Data	Zadanie
30.03	Prezentacja, praca wejścia
13.04	Utworzenie projektu, repozytorium, prostego GUI, skupienie się na obsłudze IPTables z kodu
27.04	Rozbudowa GUI - dodanie okna pozwalającego na dodawanie reguł dla IPTables, prosta tabela, podstawowa obsługa IPTables z kodu (wywołania na podstawie pól)
11.05	Pełne funkcjonowanie formy dodawania reguł dla protokołów ICMP, TCP oraz UDP, jak również obsługa ich z poziomu menadżera pakietów, testowanie oraz nauka obsługi otrzymywanych pakietów
25.05	Dodanie funkcji odzyskiwania reguł po wyłączeniu programu, rozdzielenie aplikacji okienkowej oraz menadżera pakietów na dwa osobne wątki, wypełnianie wstępne tabeli danymi pobranymi przez menadżera pakietów na podstawie zasobów współdzielonych, refaktoryzacja modułów
08.06	Dodanie automatycznego odświeżania tabeli na podstawie zasobów współdzielonych, dodanie przycisków usuwania elementów tabeli, wszystkich elementów tabeli. Dodaliśmy przycisk automatycznej odpowiedzi oraz przycisk odpowiedzi typu "Fuzz" (losowe dane są generowane), na dwuklik dodaliśmy formę edycji pakietu wraz z opcjami automatyzacji podmiany danych
po 08.06	Dodanie aplikacji w Qt C++ z obsługą gniazdek UDP i TCP, testowanie podmiany pakietów, poprawki

# Opis projektu

Aby w pełni zrozumieć działanie programu poniżej opisane są poszczególne części projektu, które składają się na całość. Począwszy od obsługiwanych protokołów, po strukturę projektu.

## Protokoły

W projekcie dodana została obsługa trzech protokołów - ICMP, UDP oraz TCP. Nie oznacza to jednak, że nie możemy przechwytywać innych pakietów. Oznacza to, że dodane zostały funkcje pozwalające na usprawnienie pracy z tymi protokołami.

Dla protokołu TCP usprawnieniem jest opcja automatycznego odsyłania pakietu, z wyszczególnieniem dwóch przypadków:

1. na przychodzącym pakiecie jest ustawiona flaga SYN - Three way handshake
2. dla każdego innego pakietu

Kod zaprezentowany na rys 1. przedstawia w jaki sposób odsyłane są automatycznie pakiety z flagą SYN. Adres IP wysyłającego jest zamieniany z odbiorcą, następnie porty są zamieniane. Na samym końcu wartość pola ACK wysyłającego jest ustawiona jako wartość pola odbiorcy SEQ, zaś wartość pola SEQ + 1 wysyłającego jest ustawiona na wartość pola ACK odbiorcy. Flaga jest ustawiana na wartość 'SA', co oznacza SYN-ACK, co potwierdza przyjęcie połączenia.

```
ip = IP()
tcp = TCP()
ip.src = pkt[IP].dst
ip.dst = pkt[IP].src

tcp.sport = pkt[TCP].dport
tcp.dport = pkt[TCP].sport

tcp.ack = pkt[TCP].seq + 1
tcp.seq = pkt[TCP].ack
tcp.flags = flag
send(ip / tcp)
```

Rys 1.: Three way handshake, protokół TCP.

W przeciwnym przypadku, przypadek 2. z listy, nie musimy podnosić numeru sekwencji, oraz dodawany jest "payload" do pakietu, tzn. warstwa bajtów. Zaprezentowany na rys. 2.

Flaga domyślnie jest ustawiana na wartość 'PA' - PSH-ACK. Co zazwyczaj spowoduje odesłanie pakietu.

```

ip = IP()
tcp = TCP()
ip.src = pkt[IP].dst
ip.dst = pkt[IP].src
tcp.ack = pkt[TCP].seq
tcp.seq = pkt[TCP].ack
tcp.sport = pkt[TCP].dport
tcp.dport = pkt[TCP].sport
tcp.flags = flag
data = pkt[TCP].payload
send(ip / tcp / data)

```

Rys 2.: Pozostałe pakiety TCP, bez flagi SYN.

Istnieje też opcja wysyłania pakietów automatycznie generowanych - fuzz. Służy to do testów automatycznych, zaimplementowany dla protokołów TCP, UDP oraz ICMP, przykład dla protokołu TCP na rys. 3. Uwagę przykuwa funkcja fuzz()

```

ip = IP()
tcp = TCP()
ip.src = pkt[IP].dst
ip.dst = pkt[IP].src
tcp.sport = pkt[TCP].dport
tcp.dport = pkt[TCP].sport
send(ip/fuzz(tcp))

```

Rys. 3.: Przykład testu typu fuzz na pakiecie TCP.

Dla pakietów UDP sytuacja jest uproszczona - sytuacja automatycznego odsyłania pakietu przedstawiona na rys. 4. Dla wysyłanego pakietu adres dostawcy oraz odbiorcy są zamieniane. Port docelowy jest zwiększany o 1 (na potrzeby analizy z klientem napisanym w Qt C++), port źródła zostaje niezmieniony. Dodatkowo przepisywany jest payload.

```

ip = IP()
udp = UDP()
ip.src = pkt[IP].dst
ip.dst = pkt[IP].src
udp.sport = pkt[UDP].sport
udp.dport = pkt[UDP].dport+1
print(pkt[UDP].payload)
data = pkt[UDP].payload
send(ip / udp / data)

```

Rys. 4.: Automatyczna obsługa pakietu UDP.

W przypadku pakietu ICMP całość zaprezentowana jest na rys. 5. Interesującymi polami są code oraz type. Type oznacza typ pakietu ICMP, wiele z tych typów posiada własne kody, dokładna rozpiska (RFC 792) - <http://www.nthelp.com/icmp.html>.

```
ip = IP()
icmp = ICMP()
ip.src = pkt[IP].dst
ip.dst = pkt[IP].src
icmp.type = 0
icmp.code = 0
icmp.id = pkt[ICMP].id
icmp.seq = pkt[ICMP].seq
data = pkt[ICMP].payload
send(ip / icmp / data, verbose=0)
```

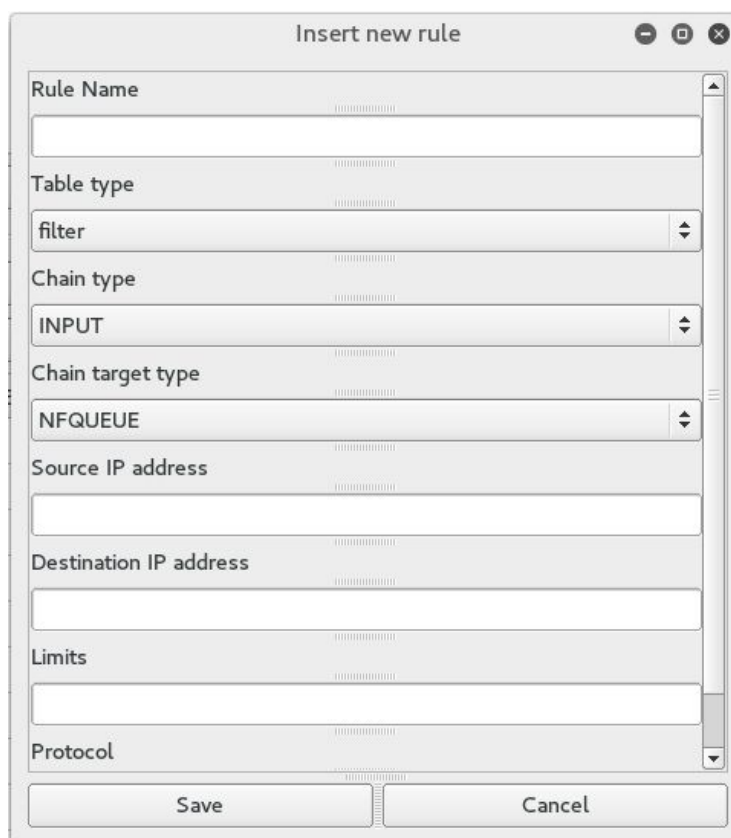
Rys. 5.: Automatyczna obsługa pakietu ICMP.

# IPTables

Sterowanie filtrami pakietów umożliwia nam umieszczenie odpowiedniego pakietu w kolejce NFQUEUE, z której korzysta biblioteka Scapy.

Aby wiedzieć jak korzystać z programu należy dokładnie rozumieć użycie programu IPTables. Ponieważ dane wprowadzane do formy są wywoływane z poziomu powłoki za pomocą właśnie tego narzędzia. Dokładna dokumentacja - <http://linux.die.net/man/8/iptables>.

Jednakże krótko opiszę poszczególne komponenty, które zostały zaimplementowane w formie dodawania reguł. Na rys. 6. zaprezentowana jest forma dodawania reguł.



Rys. 6.: Forma dodawania reguł

Spoglądając na rys. 6. szybki opis pól i ich znaczenie:

1. Table type - typ tabeli
  - a. filter - domyślna tabela, gdzie trafiają reguły
  - b. nat - tabela dla pakietów ustanawiających połączenia
  - c. mangle - tabela dla wyspecjalizowanych zmian w pakietach
  - d. raw - tabela z najwyższym priorytetem, pakiety trafiają tutaj od razu po przesłaniu
2. Chain type - Typ łańcucha, w którym mogą znaleźć się pakiety, różne typy są dostępne dla różnych typów tabel
  - a. input - wywoływana dla nadchodzących pakietów przeznaczonych dla lokalnej maszyny - dla typów tabeli filter, mangle
  - b. forward - wywoływana dla pakietów tworzonych lokalnie, przeznaczonych dla pakietów opuszczających lokalną maszynę - dla typów tabeli filter, mangle



- c. output - wywoływana dla pakietów, które są trasowane przez lokalną maszynę, ale nie są dla niej przeznaczone - dla typów tabeli filter, nat, mangle, raw
  - d. prerouting - wywoływane dla pakietów z zewnątrz przed tym jak rozpoczną być trasowane - dla typów tabeli nat, mangle, raw
  - e. postrouting - dla pakietów opuszczających lokalną maszynę, po trasowaniu - dla typów tabeli nat, mangle
3. Chain target - można określić to jako politykę określonego łańcucha (co się z nim stanie)
    - a. accept - akceptujemy przyjęcie pakietu, nie jest przetwarzany przez kod programu
    - b. drop - porzucamy przyjęcie pakietu, nie jest przetwarzany przez kod programu - wysyłający nie jest informowany
    - c. reject - odrzucamy przyjęcie pakietu, nie jest przetwarzany przez kod programu - wysyłający jest informowany
    - d. nfqueue - pakiet trafia do kolejki nfqueue, utworzonej za pomocą Scapy - my decydujemy co się z nim stanie
  4. Source address - adres IPv4, z którego przyjdzie pakiet
  5. Destination address - adres IPv4, do którego trafi pakiet
  6. Limits - określa liczbę pakietów jaka zostanie przyjęta w pewnym czasie np. 1/s (1 na sekundę), 5/h (5 na godzinę), 3/m (3 na minutę)
  7. Protocol - filtr protokołu dla reguły
    - a. None - niesprecyzowany, czyli wszystkie protokoły
    - b. ICMP - tylko protokół ICMP
    - c. UDP - tylko protokół UDP
    - d. TCP - tylko protokół TCP

Następnie w zależności od wybranego protokołu, pojawiają się pola nowe pola do wypełnienia.

Prezentuje się to następująco:

1. None - brak nowych pól
2. UDP
  - a. Source port - port źródłowy datagramu
  - b. Destination port - port docelowy datagramu
3. ICMP
  - a. ICMP type - wspomniany wcześniej w opisie protokołu (RFC 792)
4. TCP
  - a. Source port - port źródłowy datagramu
  - b. Destination port - port docelowy datagramu
  - c. Considered flags, Matched flags - wyjaśnię na przykładzie
    - i. Założmy, że wartości "considered flags" to: SYN,ACK,FIN,RST
    - ii. Założmy, że "matched flags" to: SYN

Oznacza to, że zostaną dopasowane pakiety tylko takie z ustawioną flagą SYN i nieustawioną flagą ACK,FIN,RST (wprowadzając dane nie stosujemy spacji)

## Scapy

Klasa PacketManager w pełni wykorzystuje funkcje z biblioteki Scapy, na niej opiera się cała logika działania manipulatora pakietów. Spójrzmy na rys. 7., w funkcji run\_manager odbywa się tworzenie kolejki NFQUEUE, na kolejce ustawiany jest callback, który wywołuje metodę process. Warto zauważyć, że kolejka jest tworzona na pozycji 0.

```
class PacketManager(object):

    def __init__(self, queue, queueRaw):
        print("PacketManager initialized.")
        self.queue = queue
        self.queueRaw = queueRaw

    def run_manager(self):
        q = nfqueue.queue()
        q.open()
        q.bind(socket.AF_INET)
        q.set_callback(self.process)
        q.create_queue(0)
        try:
            print("NFQUEUE ran, socket binded.")
            q.try_run()
        except:
            print(sys.exc_info()[0])
            print("NFQUEUE closed, socket unbinded.")
            q.unbind(socket.AF_INET)
            q.close()
```

Rys. 7.: Część kodu klasy PacketManager.

Metoda process przedstawiona na Rys. 8. wybiera dane z pakietu dodanego do kolejki NFQUEUE, następnie rozpakowuje je za pomocą metody IP(), pakiet jest porzucany (NF\_DROP), po to by obsłużyć go z poziomu kodu. Istotne są dwie listy - queue oraz queueRaw. Lista queue zawiera dane, które będą wyświetlane w tabeli, zaś w queueRaw jest utrzymywany nierozpakowany pakiet, tak aby móc go wykorzystać w dowolnym momencie. Warto dodać, że obie te listy są zasobem dzielonym przez dwa procesy.

```

def process(self, i, payload):
    data = payload.get_data()
    pkt = IP(data)
    proto = pkt.proto

    #print(str(pkt).encode("HEX"))
    # Dropping the packet
    payload.set_verdict(nfqueue.NF_DROP)

    # Some console logging
    print(pkt.summary())

    #Add to multiprocessing queue the data (displayed in table)
    layers = build_packet_layer(pkt)
    dictToDisplay = {}
    for layer in layers:
        dictToDisplay[layer.name] = layer.fields
    self.queue.append(dictToDisplay)

    #Origin not formatted data holded in queueRaw, also multiprocess resource
    self.queueRaw.append(data)

```

Rys. 8.: Metoda “process”

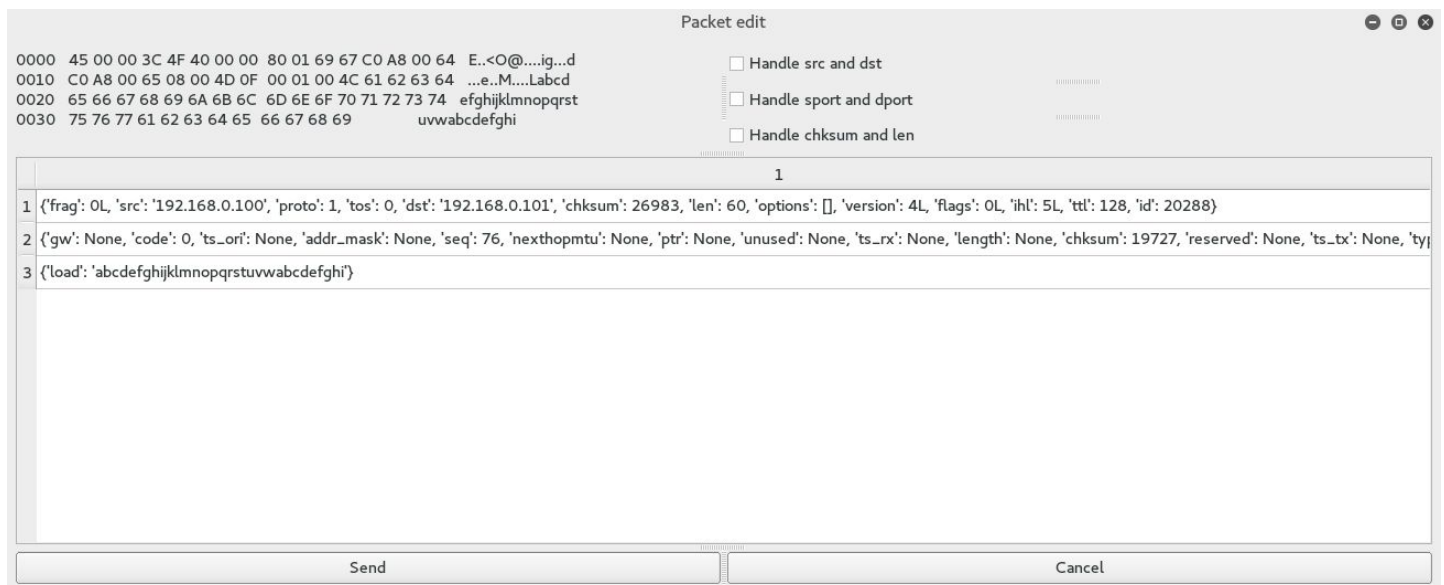
Najbardziej istotną metodą, która odsyła zmodyfikowany przez nas pakiet, w sposób półautomatyczny lub manualny jest “send\_packet\_based\_on\_layers”, widoczny na rysunku 9. Jej definicja znajduje się poza klasą PacketManager, jednakże w tym samym pliku. Klasa ta na podstawie każdej warstwy pakietu (każdego wiersza w tabeli edycji pakietu) buduje pakiet, który zostanie odesłany. Argumenty, które są przekazywane to odpowiednio:

- layersNew - nowe warstwy (pobrana bezpośrednio z tabeli)
- raw - oryginał pakietu, którego warstwy zostały nadpisane
- handleSrcAndDst - zamienia miejscami wartości pól src i dst
- handlePorts - w zależności od protokołu modyfikuje bądź zamienia miejscami wartości pól sport i dport (source port, destination port)
- handleChksumAndLen - oblicza za nas długość oraz sumę kontrolną pakietu

```
def send_packet_based_on_layers(layersNew, raw, handleSrcAndDst, handlePorts, handleChksumAndLen):
    pkt = get_packet_from_raw(raw)
    counter = 0
    while True:
        lay = pkt.getlayer(counter)
        if (lay == None):
            send_raw_packet_back(pkt, handleSrcAndDst, handlePorts, handleChksumAndLen)
            break
        else:
            for key, value in lay.fields.iteritems():
                newLay = eval(str(layersNew[counter]))
                lay.fields[key] = newLay[key]
                if "load" in key:
                    pkt[counter - 1].payload = newLay[key]
            counter = counter + 1
```

Rys. 9.: Metoda obsługująca odesłanie pakietu zmodyfikowanego widoku edycji pakietu

Aby zobrazować omawiane wyżej funkcje spójrzmy na rys. 10. W górnym lewym rogu znajduje się hexdump oryginalnej wiadomości. Na prawo znajdują się trzy pola, które można zaznaczyć. Są to wspomniane trzy ostatnie argumenty z metody “send\_packet\_based\_on\_layers”. Argument pierwszy layersNew - to każdy z wierszy złożony w jeden pakiet. Edytujemy pakiet przez bezpośrednie zmiany wartości. Jeżeli wykonamy coś źle - pakiet nie zostanie wysłany, a w konsoli pojawią się błędy.

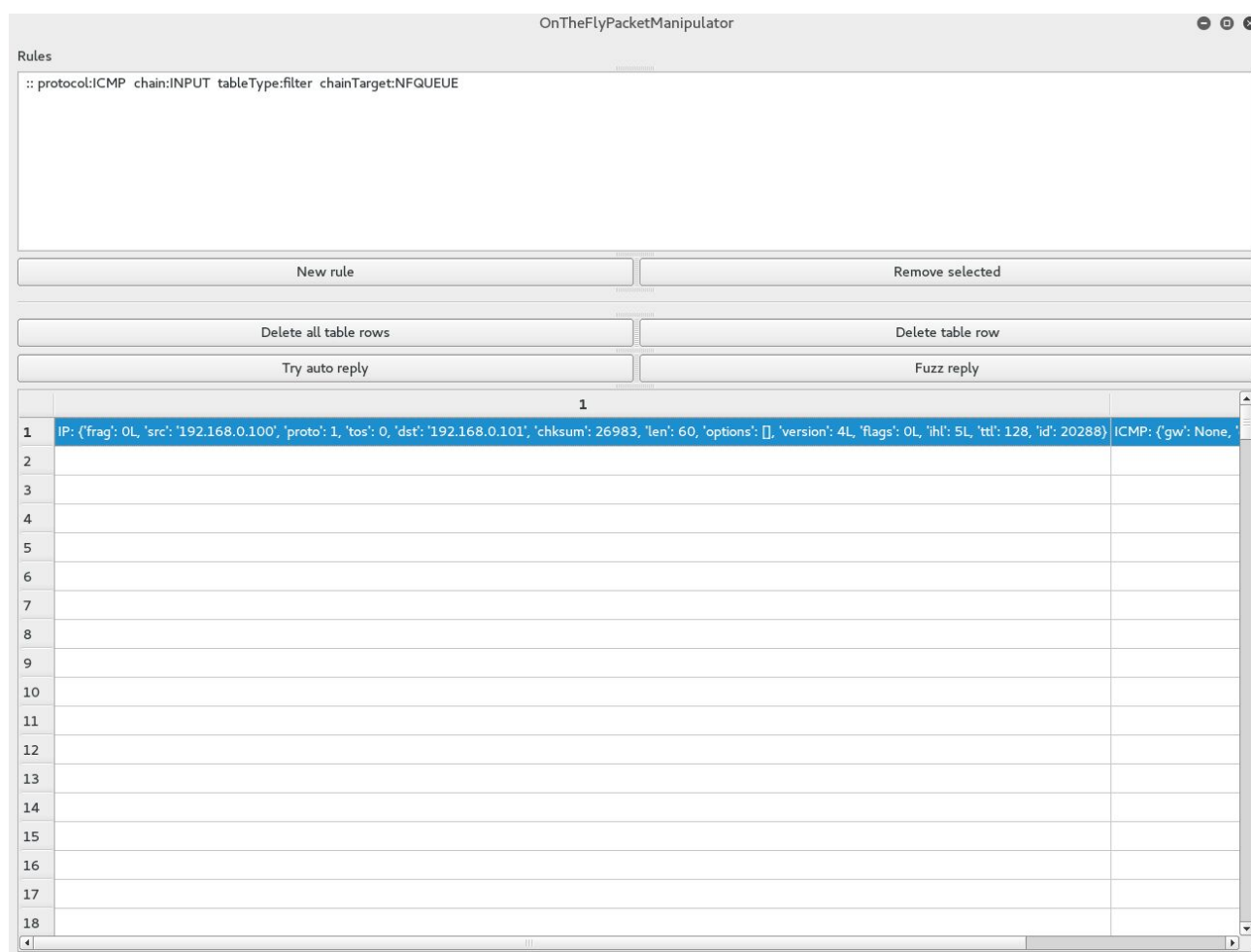


Rys. 10.: Okno edycji pakietu

# PyQt

Na całość aplikacji składa się kilka okien oraz komponentów. Główne okno przedstawione jest na rys. 11. Na samej górze w sekcji “rules” znajdują się reguły, które zostały dodane - dodawanie odbywa się przez naciśnięcie przycisku “New rule”, co otwiera okno, które zostało już wcześniej omówione. “Remove selected” usuwa jeden wpis z sekcji “rules”.

Następne przyciski dotyczą wierszy w tabeli. Każda z kolumn tabeli przedstawia pewną warstwę pakietu, wiersze w tym widoku są nieedytowalne, aby to zrobić należy nacisnąć dwukrotnie na wierszu (otwierany jest wtedy okno edycji pakietu). “Delete all table rows” usuwa wszystkie wiersze tabeli, “delete table row”, tylko jeden - aktualnie zaznaczony. Widoczne są również opcje “Try auto reply” oraz “Fuzz reply”. Zostały już wcześniej przedstawione ich funkcje - naciśnięcie ich nie powoduje otwarcia okna edycji pakietów. Opis kodu GUI jest nieistotny z punktu założeń funkcjonalnych - wyjaśnienie znajdziemy w dokumentacji PyQt.



Rys. 11.: Główne okno programu

Tabela jest odświeżana za pomocą “refreshera”, który realizuje swoje zadanie w pół sekundowych interwałach, implementacja tego rozwiązania została przedstawiona na Rys. 12. Wywoływana metoda to “packetQueueRefresher”.

```
def __init__(self, packetQueue, packetQueueRaw):
    super(MainView, self).__init__()
    self.table = MyTable(255, 12)
    self.table.setSelectionBehavior(QtGui.QAbstractItemView.SelectRows)
    self.table.setEditTriggers(QtGui.QAbstractItemView.NoEditTriggers)
    self.initUI()
    self.packetQueue = packetQueue
    self.packetQueueRaw = packetQueueRaw;
    self.timer = QTimer()
    self.timer.timeout.connect(self.packetQueueRefresher)
    self.timer.start(500)
    self.packetEditWindow = None
    self.insertRuleWindow = None
    self.table.cellDoubleClicked.connect(self.rowClicked)
```

Rys. 12.: Implementacja timera odpowiedzialnego za wywołanie odświeżenia tabeli w widoku głównym

Odświeżenie danych polega na deserializacji obiektu z zasobu międzyprocesowego, sposób realizacji takiego działania widnieje na Rys. 13.

```
def packetQueueRefresher(self):
    j = 0
    for queList in self.packetQueue:
        i = 0
        for key, value in queList.iteritems():
            newItem = QtGui.QTableWidgetItem(key + ": " + str(value))
            self.table.setItem(j, i, newItem)
            i = i + 1
        j = j + 1

    if len(self.packetQueue) > 0:
        self.table.resizeColumnsToContents()
```

Rys. 13.: Odświeżanie listy w widoku głównym

Aby zadbać o komfort użytkownika, przy uruchamianiu programu wykonywany jest backup zawartości IPTables użytkownika, dzięki czemu nie straci on dodanych przed uruchomieniem programu reguł. Wykonywanie backupu zaprezentowane zostało na Rys. 14. Po wykonaniu backupu zawartość IPTables jest czyszczona.

```
def main():
    try:
        call("iptables-save > iptables-backup", shell=True)
        call("iptables -t filter --flush", shell=True)
        call("iptables -t nat --flush", shell=True)
        call("iptables -t raw --flush", shell=True)
        call("iptables -t mangle --flush", shell=True)
```

Rys. 14.: Wykonywanie backupu zawartości IPTables

Przy zakończeniu programu wszystkie dodane przez niego reguły do IPTables zostają usunięte z listy a następnie wykonywane jest przywrócenie backupu tabeli IPTables do stanu, w którym znajdowała się przed uruchomieniem programu, kod został przedstawiony poniżej na Rys. 15..

```
finally:
    print("Retoring ip tables...")
    call("iptables-restore < iptables-backup", shell=True)
    print("Resotred.")

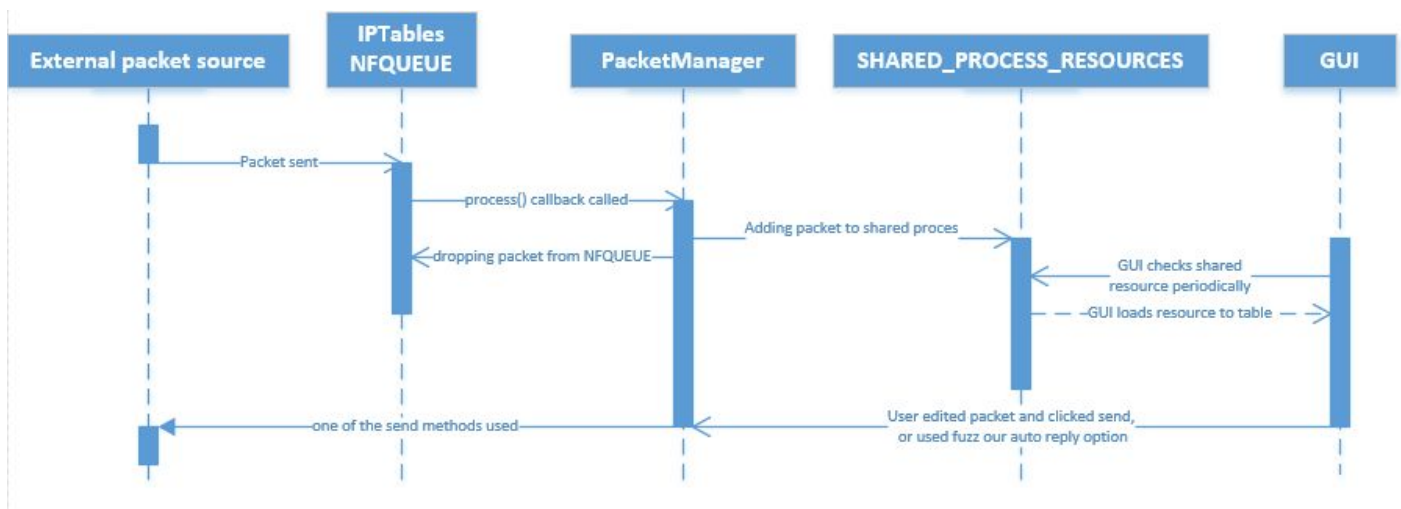
    print("Cleaning up.")
    call("pkill -f ApplicationMain.py", shell=True) #making sure scapy nfqueue has been closed
    print("Bye!")
```

Rys. 15.: Przywracanie zawartości IPTables po zakończeniu programu

## Schemat działania

Rys. 12. prezentuje poglądowy diagram sekwencji, który przedstawia przepływ danych przez aplikację. Na początku otrzymujemy pakiet z zewnętrznego źródła - jeżeli reguła została dodana w IPTables to zostaje przekazany menadżerowi pakietów. Ten dodaje go do wspólnych zasobów dla procesów, oraz ustawia na nim werdykt, by go porzucić. GUI w okresie 500ms odświeża tabelę, sprawdzając czy zaszły zmiany w zasobach. Użytkownik otwiera jeden z pakietów edytuje oraz odsyła lub korzysta z opcji automatycznej odpowiedzi, czy też testowania pakietu (fuzz), pakiet trafia w zmienionej formie do pewnego źródła - bazując na zawartości pakietu.





Rys. 12.: Poglądowy diagram sekwencji przepływu pakietu

Najbardziej istotna część programu znajduje się w pliku “ApplicationMain”. Kod pliku przedstawiony jest na rys. 13. Metoda “main()” jest główną metodą całej aplikacji. W niej na początku wykonywana jest kopia zapasowa zawartości iptables. Następnie cała zawartość iptables jest czyszczona. Kolejno tworzone są dwie listy, które są zasobem współdzielonym dla procesu packetManagerProcess, który jest demonem oraz dla procesu appProcess. Na samym końcu zawartość iptables jest przywracana do stanu sprzed uruchomienia programu oraz upewniamy się, że program został wyłączony całkowicie.



```

def runApp(queue, queueRaw):
    mv = MainView(queue, queueRaw)
    sys.exit(app.exec_())

def runPacketManager(queue, queueRaw):
    pm = PacketManager(queue, queueRaw)
    pm.run_manager()

def main():
    try:
        call("iptables-save > iptables-backup", shell=True)
        call("iptables -t filter --flush", shell=True)
        call("iptables -t nat --flush", shell=True)
        call("iptables -t raw --flush", shell=True)
        call("iptables -t mangle --flush", shell=True)

        queue = multiprocessing.Manager().list()
        queueRaw = multiprocessing.Manager().list()
        packetManagerProcess = multiprocessing.Process(target=runPacketManager, args=(queue, queueRaw, ))
        packetManagerProcess.daemon = True
        packetManagerProcess.start()

        appProcess = multiprocessing.Process(target=runApp, args=(queue, queueRaw))
        appProcess.daemon = False
        appProcess.start()

        appProcess.join()
        packetManagerProcess.terminate()
    finally:
        print("Retoring ip tables...")
        call("iptables-restore < iptables-backup", shell=True)
        print("Resotred.")

        print("Cleaning up.")
        call("pkill -f ApplicationMain.py", shell=True) #making sure scapy nfqueue has been closed
        print("Bye!")

```

Rys. 13.: Główna funkcja programu, powoływanie procesów

## Qt

Aby lepiej zrozumieć działanie przesyłanych pakietów postanowiliśmy utworzyć prostą aplikację, które wysyła pakiety (TCP) oraz datagramy (UDP). W tym celu skorzystaliśmy z C++ oraz silnika Qt.

## TCP

Na rys. 14. zaprezentowany jest kod wykorzystywany do testowania pakietów protokołu TCP. Slot `printData()` informuje nas jaką wiadomość odbierzemy od manipulatora, zaś sloty `connected()` oraz `disconnected()` poinformują nas, gdy się połączymy bądź rozłączymy.

```
socket = new QTcpSocket();
connect(socket, SIGNAL(connected()), this, SLOT(connected()));
connect(socket, SIGNAL(disconnected()), this, SLOT(disconnected()));
connect(socket, SIGNAL(readyRead()), this, SLOT(printData()));

socket->connectToHost("192.168.0.101", 1000);
socket->waitForConnected(45000);
if (socket->state() == QTcpSocket::ConnectedState){
    socket->write("ALOHA MAN");
}
```

Rys. 14.: Kod gniazda TCP

## UDP

Na rys. 15. zaprezentowany jest kod wykorzystywany do testowania pakietów protokołu UDP. Slot `printUdpData()` odbiera oraz odsyła dane do manipulatora. Nasłuchiwanie odbywa się na porcie 1000, zaś wysyłanie na porcie 999.

```
udpSocket = new QUdpSocket();
connect(udpSocket, SIGNAL(readyRead()), this, SLOT(printUDPData()));

QHostAddress address("192.168.0.100");
udpSocket->bind(address, 1000);

udpSocketSender = new QUdpSocket();
udpSocketSender->writeDatagram("Jeden dwa trzy!", QHostAddress("192.168.0.101"), 999);
```

Rys. 15.: Kod gniazda UDP

# Instrukcja obsługi

Wszystkie wskazówki dotyczące programu można wyczytać na poprzednich stronach dokumentacji.

Aby uruchomić program należy mieć zainstalowane:

- System linux z jądrem skompilowanym wraz z ip\_tables
- Python 2.7
- Scapy
- NFQUEUE

Następnie uruchamiamy terminal, przechodzimy do folderu z projektem i wpisujemy:

```
python ApplicationManager.py
```

Wszystkie błędy aplikacji logowane są w konsoli.

## Cheat sheet

Zamieszczone zostaje tutaj kilka tabel, spisów, które usprawnią manipulowanie pakietami, bez konieczności posiadania dokumentacji.

Wartości HEX flag protokołu TCP	
0x00 NULL	0x80 CWR
0x01 FIN	0x81 FIN-CWR
0x02 SYN	0x82 SYN-CWR
0x03 FIN-SYN	0x83 FIN-SYN-CWR
0x08 PSH	0x88 PSH-CWR
0x09 FIN-PSH	0x89 FIN-PSH-CWR
0x0A SYN-PSH	0x8A SYN-PSH-CWR
0x0B FIN-SYN-PSH	0x8B FIN-SYN-PSH-CWR
0x10 ACK	0x90 ACK-CWR
0x11 FIN-ACK	0x91 FIN-ACK-CWR
0x12 SYN-ACK	0x92 SYN-ACK-CWR
0x13 FIN-SYN-ACK	0x93 FIN-SYN-ACK-CWR
0x18 PSH-ACK	0x98 PSH-ACK-CWR
0x19 FIN-PSH-ACK	0x99 FIN-PSH-ACK-CWR
0x1A SYN-PSH-ACK	0x9A SYN-PSH-ACK-CWR
0x1B FIN-SYN-PSH-ACK	0x9B FIN-SYN-PSH-ACK-CWR
0x40 ECE	0xC0 ECE-CWR
0x41 FIN-ECE	0xC1 FIN-ECE-CWR
0x42 SYN-ECE	0xC2 SYN-ECE-CWR
0x43 FIN-SYN-ECE	0xC3 FIN-SYN-ECE-CWR
0x48 PSH-ECE	0xC8 PSH-ECE-CWR
0x49 FIN-PSH-ECE	0xC9 FIN-PSH-ECE-CWR
0x4A SYN-PSH-ECE	0xCA SYN-PSH-ECE-CWR
0x4B FIN-SYN-PSH-ECE	0xCB FIN-SYN-PSH-ECE-CWR
0x50 ACK-ECE	0xD0 ACK-ECE-CWR
0x51 FIN-ACK-ECE	0xD1 FIN-ACK-ECE-CWR

0x52 SYN-ACK-ECE	0xD2 SYN-ACK-ECE-CWR
0x53 FIN-SYN-ACK-ECE	0xD3 FIN-SYN-ACK-ECE-CWR
0x58 PSH-ACK-ECE	0xD8 PSH-ACK-ECE-CWR
0x59 FIN-PSH-ACK-ECE	0xD9 FIN-PSH-ACK-ECE-CWR
0x5A SYN-PSH-ACK-ECE	0xDA SYN-PSH-ACK-ECE-CWR
0x5B FIN-SYN-PSH-ACK-ECE	0xDB FIN-SYN-PSH-ACK-ECE-CWR

Typy ICMP	
0	Echo Reply
1	Unassigned
2	Unassigned
3	Destination Unreachable
4	Source Quench
5	Redirect
6	Alternate Host Address
7	Unassigned
8	Echo
9	Router Advertisement
10	Router Selection
11	Time Exceeded
12	Parameter Problem
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply
17	Address Mask Request
18	Address Mask Reply
19	Reserved (for Security)
20-29	Reserved (for Robustness Experiment)
30	Traceroute
31	Datagram Conversion Error
32	Mobile Host Redirect
33	IPv6 Where-Are-You
34	IPv6 I-Am-Here
35	Mobile Registration Request
36	Mobile Registration Reply
37	Domain Name Request
38	Domain Name Reply
39	SKIP
40	Photuris
41-255	Reserved

### Nagłówek UDP

+	Bity 0 – 7	8 – 15	16 – 23	24 – 31
0	Adres źródłowy			
32	Adres docelowy			
64	Zera	Protokół	Długość UDP	
96	Port źródłowy		Port docelowy	
128	Długość		Suma kontrolna	
160	Dane			

- **Długość** - 16-bitowe pola specyfikują długość w bajtach całego datagramu: nagłówek i dane. Minimalna długość to 8 bajtów i jest to długość nagłówka. Wielkość pola ustala teoretyczny limit 65527 bajtów, dla danych przenoszonych przez pojedynczy datagram UDP.
- **Suma kontrolna** - 16 bitowe pole, które jest użyte do sprawdzania poprawności nagłówka oraz danych. Pole jest opcjonalne. Ponieważ IP nie wylicza sumy kontrolnej dla danych, suma kontrolna UDP jest jedyną gwarancją, że dane nie zostały uszkodzone.

Źródło: [https://pl.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://pl.wikipedia.org/wiki/User_Datagram_Protocol)

### Nagłówek IP

+	Bity 0 - 3	4 - 7	8 - 15	16 - 18	19 - 31
0	Wersja	Długość nagłówka	Typ usługi	Całkowita długość	
32	Numer identyfikacyjny			Flagi	Przesunięcie
64	Czas życia		Protokół warstwy wyższej	Suma kontrolna nagłówka	
96	Adres źródłowy IP				
128	Adres docelowy IP				
160	Opcje IP			Wypełnienie	
192	Dane				

- **Wersja** (4 bity) - (ang. Version) pole opisujące wersję protokołu, jednoznacznie definiujące format nagłówka.
- **Długość nagłówka** (4 bity) - (ang. Internet Header Length) długość nagłówka IP wyrażona w 32-bitowych słowach; minimalny, poprawny nagłówek ma długość co najmniej 5.
- **Typ usługi** (8 bitów) - (ang. Type of Services) pole wskazujące jaka jest pożądana wartość QoS dla danych przesyłanych w pakiecie. Na podstawie tego pola, routery ustawiają odpowiednie wartości transmisji.
- **Całkowita długość pakietu** (16 bitów) - (ang. Total Length) długość całego datagramu IP (nagłówek oraz dane); maksymalna długość datagramu wynosi  $2^{16} - 1 = 65535$ . Minimalna wielkość datagramu jaką musi obsłużyć każdy host wynosi 576 bajtów, dłuższe pakiety mogą być dzielone na mniejsze (fragmentacja).
- **Numer identyfikacyjny** (16 bitów) - (ang. Identification) numer identyfikacyjny, wykorzystywany podczas fragmentacji do określenia przynależności pofragmentowanych datagramów
- **Flagi** (3 bity) - (ang. Flag) flagi wykorzystywane podczas fragmentacji datagramów. Zawierają dwa używane pola: DF, które wskazuje, czy pakiet może być fragmentowany oraz MF, które wskazuje, czy za danym datagramem znajdują się kolejne fragmenty.
- **Przesunięcie** (13 bitów) - (ang. Fragment Offset) w przypadku fragmentu większego datagramu pole to określa miejsce danych w oryginalnym datagramie; wyrażone w jednostkach ośmiooktetowych
- **Czas życia** (8 bitów) - (ang. Time to live) czas życia datagramu. Zgodnie ze standardem liczba przeskoków przez jaką datagram znajduje się w obiegu. Jest zmniejszana za każdym razem, gdy datagram jest przetwarzany w routerze - jeżeli czas przetwarzania jest równy 0, datagram jest usuwany z sieci (nie przekazywany dalej) o czym nadawca usuniętego pakietu jest informowany zwrotnie z wykorzystaniem protokołu ICMP. Istnienie tej wartości jest konieczne, zapobiega krążeniu pakietów (patrz Burza broadcastowa) w sieci.
- **Protokół warstwy wyższej** (8 bitów) - (ang. Protocol) informacja o protokole warstwy wyższej, który jest przenoszony w polu danych datagramu IP.



- **Suma kontrolna nagłówka** (16 bitów) - (ang. Header Checksum) suma kontrolna nagłówka pakietu, pozwalająca stwierdzić czy został on poprawnie przesłany, sprawdzana i aktualizowana przy każdym przetwarzaniu nagłówka.
- **Adres źródłowy** (32 bity) i adres docelowy (32 bity) - (ang. Source/Destination IP Address) pola adresów nadawcy i odbiorcy datagramu IP.
- **Opcje** (32 bity) - (ang. Options) niewymagane pole opcji, opisujące dodatkowe zachowanie pakietów IP
- **Wypełnienie** - (ang. Padding) - opcjonalne pole wypełniające nagłówek tak, aby jego wielkość była wielokrotnością 32, wypełnione zerami.

Źródło: <https://pl.wikipedia.org/wiki/IPv4>

### Nagłówek TCP

Offset	Oktet	0								1								2								3							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Port nadawcy																Port odbiorcy															
4	32	Numer sekwencyjny																															
8	64	Numer potwierdzenia (jeżeli flaga ACK jest ustawiona)																															
12	96	Długość nagłówka				Zarezerwowane			N S	C W R	E C R	U R C	A C S	P S S	R S Y	S Y I	Szerokość okna																
16	128	Suma kontrolna																Wskaźnik priorytetu (jeżeli flaga URG jest ustawiona)															
20	160	Opcje (jeżeli długość nagłówka > 5, to pole jest uzupełniane "0")																															
...	...	...																															

- **Port nadawcy** – 16-bitowy numer identyfikujący [port](#) nadawcy.
- **Port odbiorcy** – 16-bitowy numer identyfikujący [port](#) odbiorcy.
- **Numer sekwencyjny** – 32-bitowy identyfikator określający miejsce pakietu danych w pliku przed fragmentacją (dzięki niemu, można "poskładać" plik z poszczególnych pakietów).
- **Numer potwierdzenia** – 32-bitowy numer będący potwierdzeniem otrzymania pakietu przez odbiorcę, co pozwala na synchronizację nadawanie-potwierdzenie.
- **Długość nagłówka** – 4-bitowa liczba, która oznacza liczbę 32-bitowych wierszy nagłówka, co jest niezbędne przy określaniu miejsca rozpoczęcia danych. Dlatego też nagłówki mogą mieć tylko taką długość, która jest wielokrotnością 32 bitów.
- **Zarezerwowane** – 3-bitowy ciąg zer, zarezerwowany dla ewentualnego przyszłego użytku.
- **Flagi** – 9-bitowa informacja/polecenie dotyczące bieżącego pakietu. Poszczególne flagi oznaczają:
  - NS – (ang. Nonce Sum) jednobitowa suma wartości flag ECN (ECN Echo, Congestion Window Reduced, Nonce Sum) weryfikująca ich integralność
  - CWR – (ang. Congestion Window Reduced) flaga potwierdzająca odebranie powiadomienia przez nadawcę, umożliwia odbiorcy zaprzestanie wysyłania echa.
  - ECE – (ang. ECN-Echo) flaga ustawiana przez odbiorcę w momencie otrzymania pakietu z ustawioną flagą CE
  - URG – informuje o istotności pola "Priorytet"
  - ACK – informuje o istotności pola "Numer potwierdzenia"
  - PSH – wymusza przesłanie pakietu

- RST – resetuje połączenie (wymagane ponowne uzgodnienie sekwencji)
- SYN – synchronizuje kolejne numery sekwencyjne
- FIN – oznacza zakończenie przekazu danych
- **Szerokość okna** – 16-bitowa informacja o tym, ile danych może aktualnie przyjąć odbiorca. Wartość 0 wskazuje na oczekiwanie na segment z innym numerem tego pola. Jest to mechanizm zabezpieczający komputer nadawcy przed zbyt dużym napływem danych.
- **Suma kontrolna** – 16-bitowa liczba, będąca wynikiem działań na bitach całego pakietu, pozwalająca na sprawdzenie tego pakietu pod względem poprawności danych. Obliczana jest z całego nagłówka TCP z wyzerowanymi polami sumy kontrolnej oraz ostatnich ośmiu pól nagłówka IP stanowiących adresy nadawcy i odbiorcy pakietu.
- **Wskaźnik priorytetu** – jeżeli flaga URG jest włączona, informuje o ważności pakietu.
- **Opcje** – czyli ewentualne dodatkowe informacje i polecenia:
  - 0 – koniec listy opcji
  - 1 – brak działania
  - 2 – ustawia maksymalną długość segmentu

W przypadku opcji 2 to tzw. Uzupełnienie, które dopełnia zerami długość segmentu do wielokrotności 32 bitów (patrz: informacja o polu "Długość nagłówka")

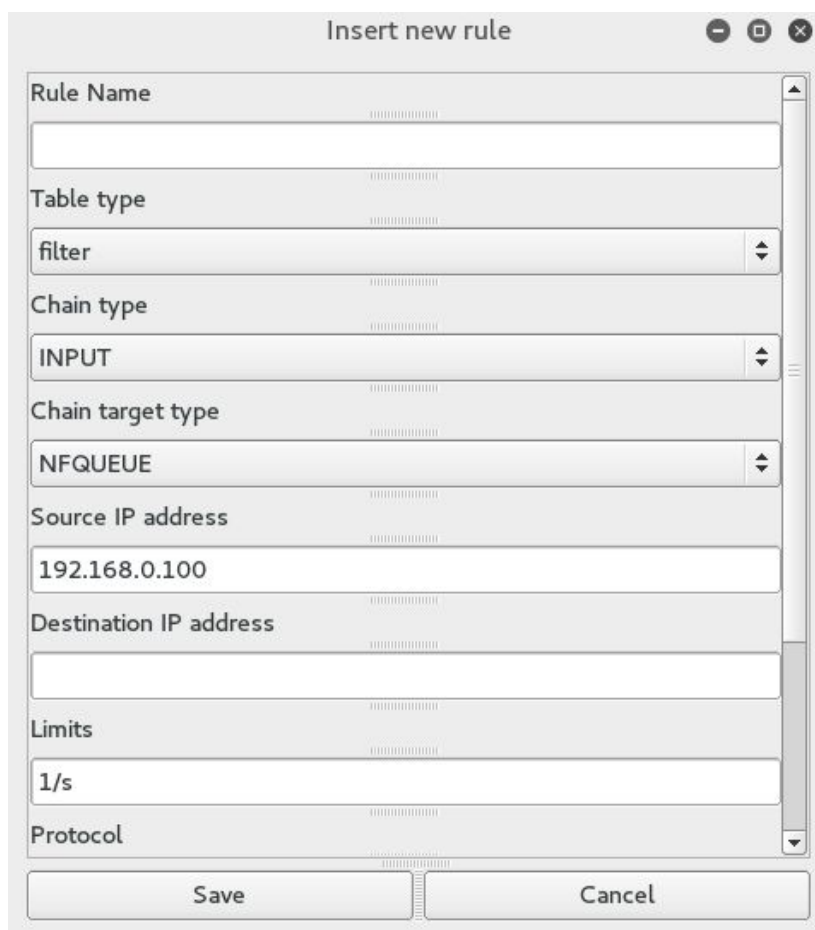
Źródło: [https://pl.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://pl.wikipedia.org/wiki/Transmission_Control_Protocol)



# Przykłady

## Przechwycenie i podmiana komunikatu UDP

Na rysunku 16. widzimy okno dodawania nowej reguły. Chain target type ustawiamy na wartość NFQUEUE - sprawia to, że pakiet zostanie obsłużony przez kod programu. Podajemy adres z jakiego chcemy otrzymywać pakiety oraz ograniczamy obsługiwane pakiety do jednego na sekundę.



Insert new rule

Rule Name

Table type

filter

Chain type

INPUT

Chain target type

NFQUEUE

Source IP address

192.168.0.100

Destination IP address

Limits

1/s

Protocol

Save Cancel

Rys. 16.: Okno dodawania nowej reguły - UDP.

W dalszej części, na rysunku 17. definiujemy protokół - jest to UDP. Przyjęte zostaną tylko pakiety z zakresu 999 do 1000.



Protocol

UDP

Source Port

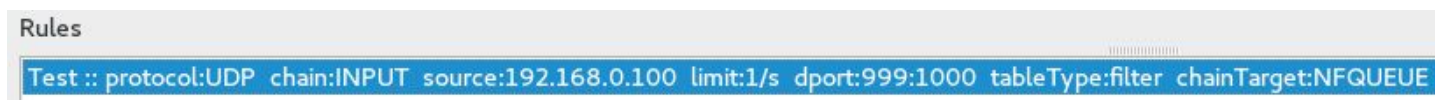
Destination Port

999:1000

Save Cancel

Rys. 17.: Okno dodawania nowej reguły - UDP - następna część.

Na rysunku 18. widzimy charakterystykę dodanej reguły na liście reguł. Znajduje się tutaj pełna informacja o dodanej regule - nazwa reguły czysto subiektywna. Ma na celu pomóc nam się odnaleźć w dodawanych regułach. Warto przypomnieć, że reguły można składać. Tzn. jedna może przyjmować UDP z pewnego zakresu, druga odrzucać z pewnego zakresu. Jedna może przechwytywać pakiety, które są trasowane, a porzucać te, które są tworzone na naszym komputerze etc.



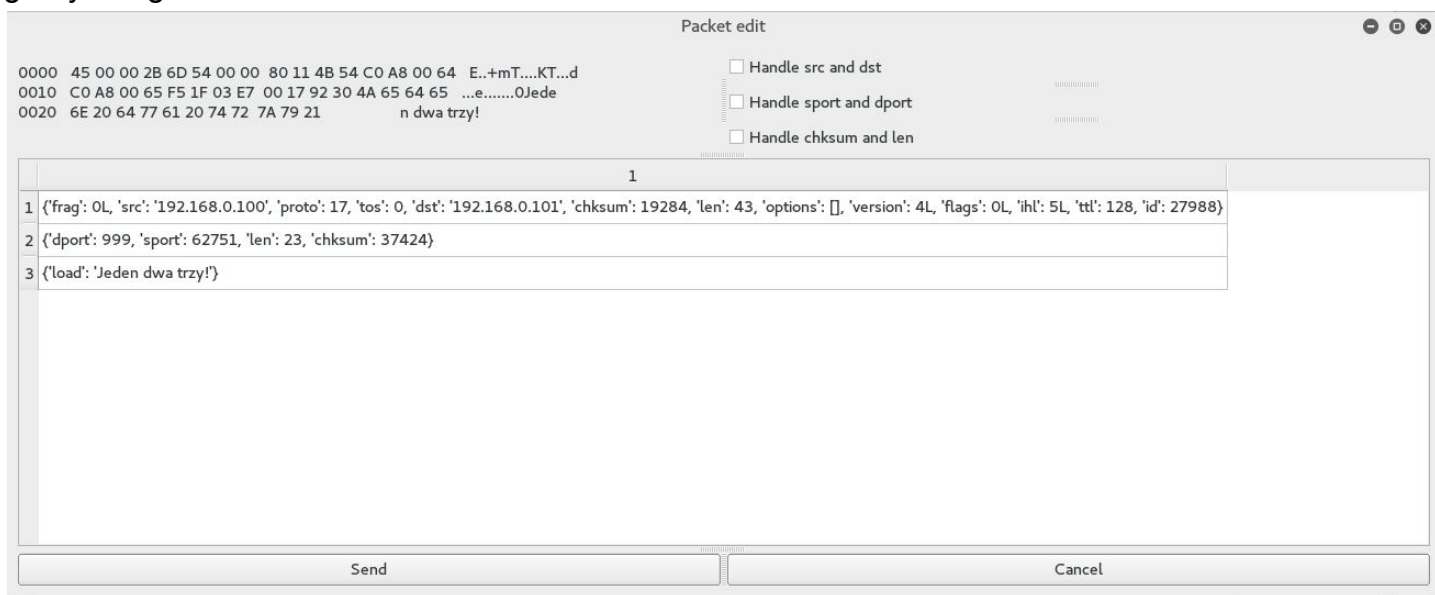
Rys. 18.: Widok dodanej reguły na liście.

W międzyczasie uruchomiliśmy prosty program oparty o gniazda sieciowe (powyżej wspomniany w dokumentacji). Wysłałem on nam wiadomość za pomocą protokołu UDP. Wiersz jest wyświetlony w tabeli. W każdej kolumnie znajduje się kolejna warstwa pakietu - przewijając okno na prawo możemy przejrzeć następne warstwy.

1	
1	IP: {'frag': 0L, 'src': '192.168.0.100', 'proto': 17, 'tos': 0, 'dst': '192.168.0.101', 'chksum': 19284, 'len': 43, 'options': [], 'version': 4L, 'flags': 0L, 'ihl': 5L, 'ttl': 128, 'id': 27988} UDP: {'dport': 999,

Rys. 19.: Przechwycony pakiet UDP w wierszu tabeli.

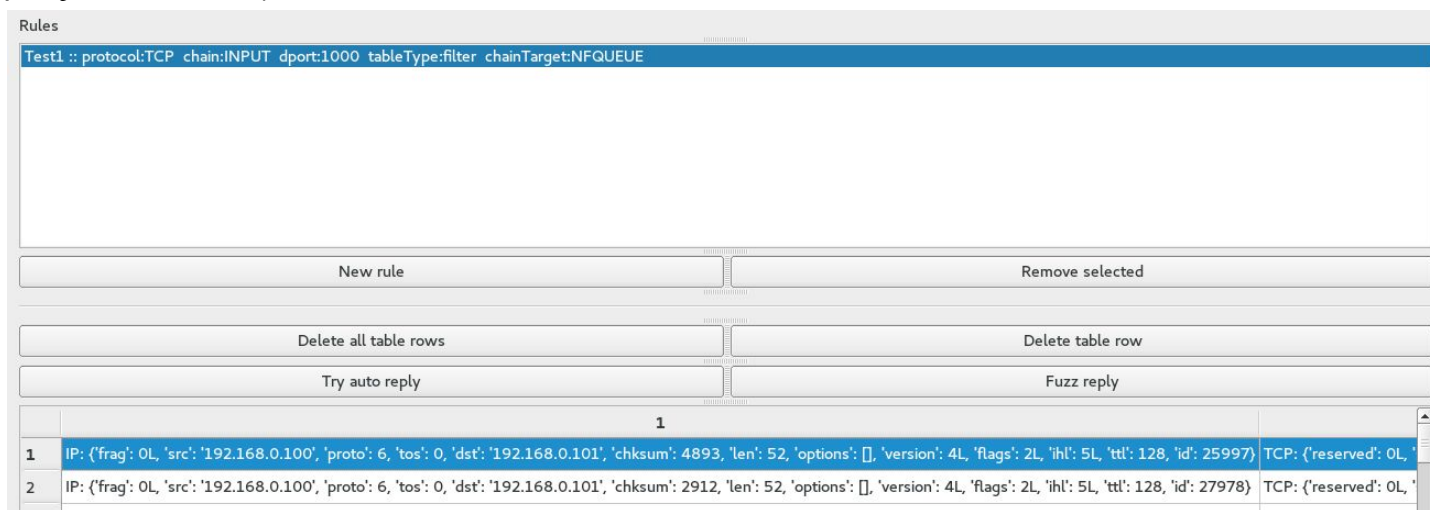
Na rysunku 20. widzimy okno edycji pakietu. Warstwy zamiast kolumnami są tutaj wyświetlone we wierszach. I tak od góry: warstwa IP, UDP, Payload. W górnym prawym rogu można dostrzec trzy przyciski. Pozwalają one na przyspieszenie pracy podmiiany manualnej pakietu. Każde z pól jest w pełni konfigurowalne. W tym przypadku, aby poprawnie odesłać pakiet UDP, dport należy ustawić na wartość 1000 (port na którym nasłuchuje listener socket). Należy zamienić src oraz dst w warstwie IP. Do obliczenia sumy kontrolnej oraz długości pakietu chcemy skorzystać z przycisku w prawym górnym rogu.



Rys. 20.: Okno edycji pakietu UDP.

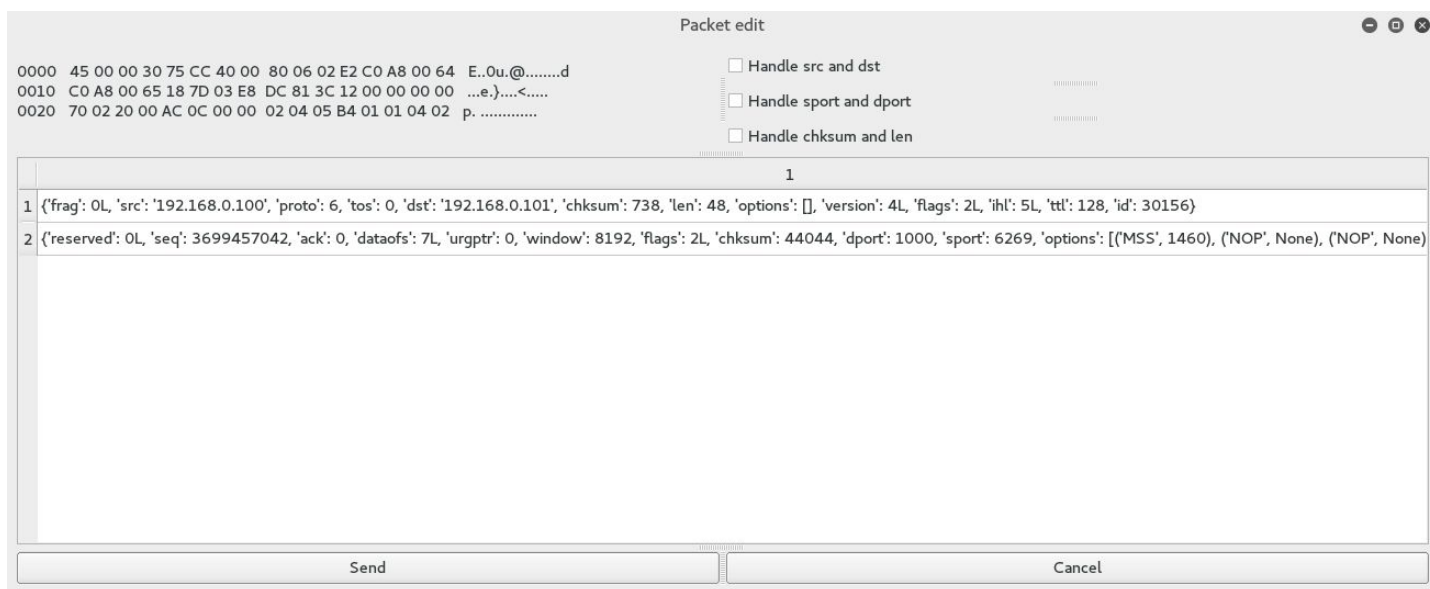
## Przechwycenie i podmiana komunikatu TCP

Na rysunku 21. widzimy nową regułę, na wzór poprzedniej. Dport ustawiony jest na 1000, zaś protokół, który teraz przechwytujemy to TCP. Poniżej pojawiły się dwa pakiety. Po pobieżnej analizie można dostrzec, że są to pakiety odpowiedzialne za Three Way Handshake (nawiązywanie połączenia w TCP).



Rys. 21.: Główne okno programu, konfiguracja pod TCP.

Na rys. 22. znajdziemy potwierdzenie - w warstwie TCP możemy dostrzec flagę ustawioną na wartość 2. Kiedy przejdziemy do tabelki **“Wartości HEX flag protokołu TCP”** i porównamy wartości, okaże się, że flaga 2 to flaga synchronizacji (SYN). Oznacza to, że gniazdko oczekuje akceptacji (SYN-ACK). Tym razem skorzystamy z opcji automatycznej - używamy przycisku **“Try auto reply”**.



Rys. 22.: TCP, pakiet z flagą SYN (Three way handshake).

Na rys. 23. możemy już zaobserwować odebrany kolejny pakiet. Gniazdko zaraz po nawiązaniu połączenia wysyła wiadomość powitalną. Sytuacja ma się dokładnie tak sam w przypadku edytowania UDP jak i TCP. Wszystko jest edytowalne - jednakże warto zajrzeć do działu “Protokoły” by dowiedzieć się jak TCP obsługuje porty, a jak to robi UDP, etc.

Rules

Test1 :: protocol:TCP chain:INPUT dport:1000 tableType:filter chainTarget:NFQUEUE

Packet edit

0000 45 00 00 31 11 4F 40 00 80 06 67 5E C0 A8 00 64 E..1.O@...g^...d

0010 C0 A8 00 65 18 80 03 E8 29 A6 CC 98 00 00 00 01 ...e....).....

0020 50 18 FF 70 AE 9A 00 00 41 4C 4F 48 41 20 4D 41 P..p....ALPHA MA

0030 4E N

☐ Handle src and dst

☐ Handle sport and dport

☐ Handle chksum and len

1

1 {'frag': 0L, 'src': '192.168.0.100', 'proto': 6, 'tos': 0, 'dst': '192.168.0.101', 'chksum': 26462, 'len': 49, 'options': [], 'version': 4L, 'flags': 2L, 'ihl': 5L, 'ttl': 128, 'id': 4431}

2 {'reserved': 0L, 'seq': 698797208, 'ack': 1, 'dataofs': 5L, 'urgptr': 0, 'window': 65392, 'flags': 24L, 'chksum': 44698, 'dport': 1000, 'sport': 6272, 'options': []}

3 {'load': 'ALPHA MAN'}

Send

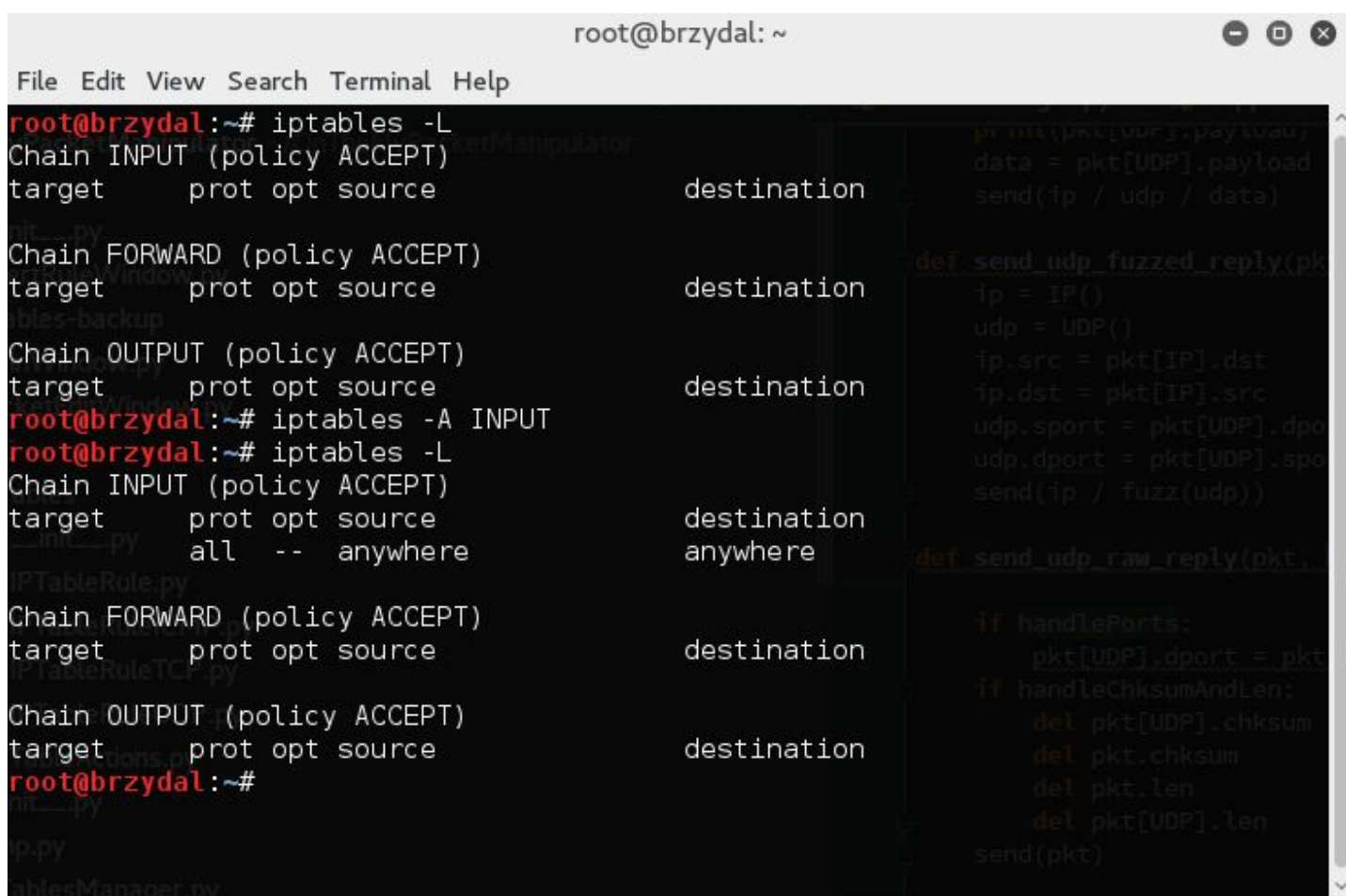
Cancel

Rys. 23.: Pakiet po nawiązaniu połączenia z gniazdem TCP.

# Różnica

Dla osób zupełnie niezaznajomionych z systemem Linux czy narzędziem Scapy oraz IPTables. Powyżej w dokumencie zostały opisane osiągnięcia tego projektu. Poniżej przedstawiam proste, całkowicie manualne wykorzystanie IPTables oraz Scapy. Czyli - jak to wygląda z poziomu konsoli. Po zaawansowane informacje - odsyłam do dokumentacji zarówno IPTables oraz Scapy.

Na rysunku 24. możemy zobaczyć jak wygląda IPTables w oryginale. Wszystko to zostało przepisane do interfejsu graficznego, obsłużona została spora część wszystkich parametrów IPTables. Na rysunku przedstawiono kolejno: wyświetlenie pustej tabeli, dodanie wpisu, który akceptuje wszystko, wyświetlenie ponownie - wraz z dodanym plikiem.

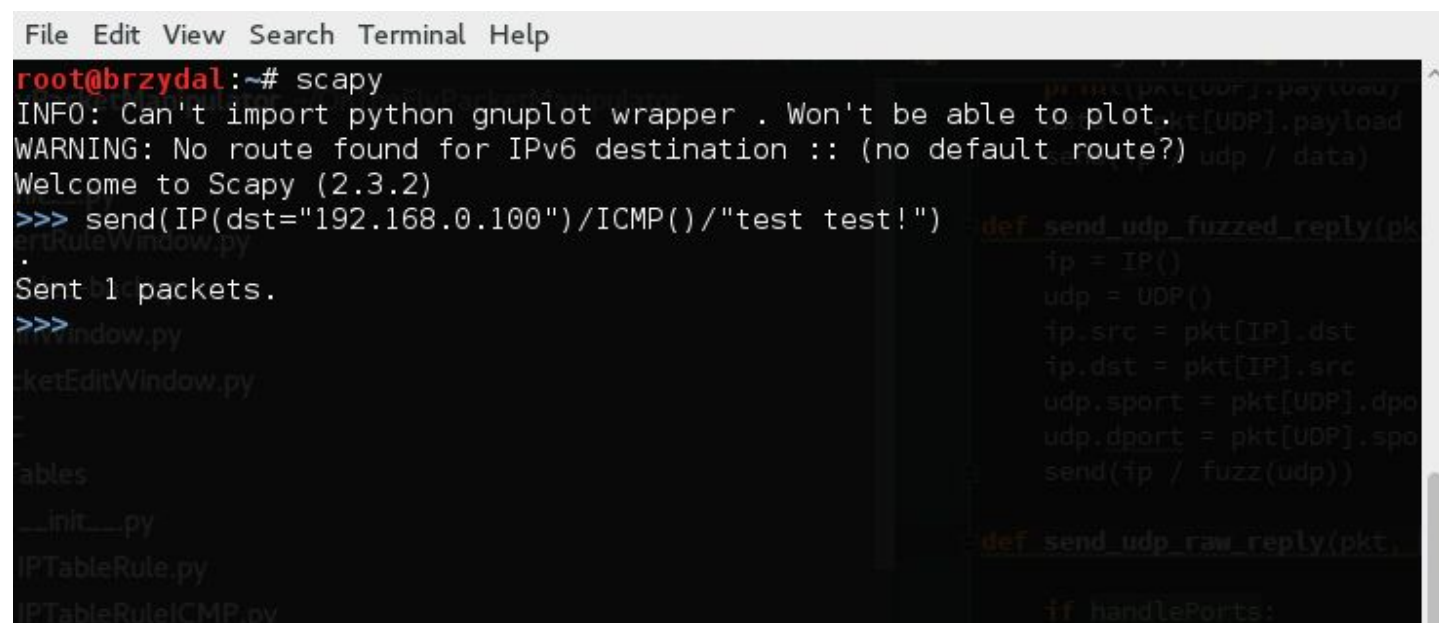


```
root@brzydal: ~
File Edit View Search Terminal Help
root@brzydal:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
root@brzydal:~# iptables -A INPUT
root@brzydal:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
--in--    all  --  anywhere              anywhere
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
root@brzydal:~#
def send_udp_fuzzed_reply(pkt):
    print(pkt[UDP].payload)
    data = pkt[UDP].payload
    send(ip / udp / data)

def send_udp_raw_reply(pkt):
    if handlePorts:
        pkt[UDP].dport = pkt
    if handleChecksumAndLen:
        del pkt[UDP].checksum
        del pkt.checksum
        del pkt.len
        del pkt[UDP].len
    send(pkt)
```

Rys. 24.: Obsługa IPTables z poziomu konsoli.

Na rysunku 25. możemy zobaczyć proste wywołanie jednej z metod scapy. Możemy zobaczyć jak budowane i składane są kolejne warstwy pakietu. Najpierw warstwa IP, na to jest nakładana warstwa ICMP, na samym końcu Payload - czysty tekst.



```
File Edit View Search Terminal Help
root@brzydal:~# scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot.
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.3.2)
>>> send(IP(dst="192.168.0.100")/ICMP()/"test test!")
Sent 1 packets.
>>>
```

```
def send_udp_fuzzed_reply(pkt):
    ip = IP()
    udp = UDP()
    ip.src = pkt[IP].dst
    ip.dst = pkt[IP].src
    udp.sport = pkt[UDP].dport
    udp.dport = pkt[UDP].sport
    send(ip / fuzz(udp))

def send_udp_raw_reply(pkt):
    if handlePorts:
```

Rys. 25.: Obsługa scapy z poziomu konsoli.