Zber, analýza a klasifikácia sieťovej prevádzky modelmi strojového učenia

Cieľom tohto prakticky zameraného semináru je ukázať ako je možné vytvoriť analytický nástroj využitím databázy a strojového učenia.

Využité technológie

V rámci semináru sa bude pracovať s nasledujúcimi technológiami:

a. Tvorba desktopovej aplikácie využitím knižnice Tkinter.
b. Programovací jazyk Python a knižnica Scapy.
c. PostgreSQL databáza.
d. Scikit-Learn pre prácu so strojovým učením.
e. Sieťová infraštruktúra (Prepínače, Smerovače).
f. Wireshark pre kontrolu priebehu útokov.

Zbierka riešených úloh

Zbierka úloh pozostáva z ukážky vzorového riešenia úloh zameraných na precvičenie praktických zručností spájajúcich rôzne odvetvia informatiky (Počítačové siete, Bezpečnosť, Programovanie, Databázy a Strojové učenie).

Úloha 0: Vytvorenie základného projektu v IDE PyCharm

Najskôr je potrebné do Vášho systému nainštalovať podporu pre jazyk Python verzie 3.x.x (možné stiahnuť z: https://www.python.org/downloads/). Základná inštalácia Pythonu v sebe obsahuje základné vývojové prostredie IDLE. Vhodnejšie je ale pracovať s niektorým z dostupných IDE. V našom prípade sa bude pracovať s nástrojom PyCharm Community (možné stiahnuť z: https://www.jetbrains.com/pycharm/download/#section=windows).

Po nainštalovaní podpory pre jazyk a vývojového prostredia stačí pre vytvorenie nového projektu vyhľadať záložku **File -> Create Project**. Projektu stačí zadať názov a zvoliť jeho umiestnenie. Po vyplnení týchto údajov stačí stlačiť tlačidlo **Create**.

Po vytvorení projektu je možné upravovať vytvorený súbor s príponou **.py**, alebo je možné vytvárať vlastné pracovné súbory (kliknutím pravým tlačidlom myši na názov vytvoreného projektu a zvolením možnosti **New -> Python File**, ktorému je možné zadať ľubovoľný názov). Keďže je Python Interpretovaný jazyk, tak je do súboru možné priamo zadávať požadované príkazy s následným interpretovaním súboru stlačením tlačidla vývojového prostredia **RUN**. Pre lepšiu orientáciu bude ale v každom súbore vložená špeciálna metóda **main**, v ktorej budú zadávané príkazy a volané vytvorené obslužné funkcie. Vzhľad základného zdrojového kódu je znázornený na nasledujúcom výpise:

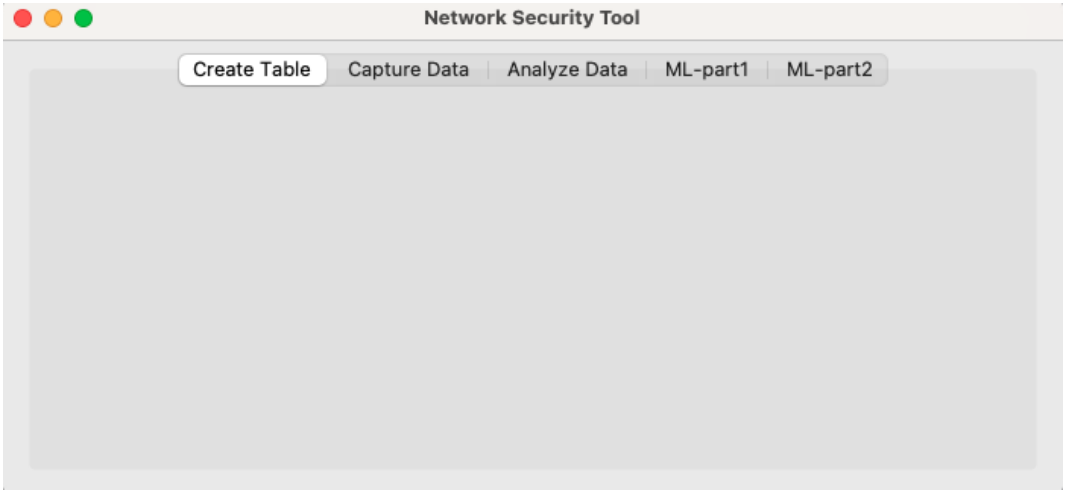
```
import library

def function():
    function body

if __name__ == '__main__':
    function()
```

Úloha 1: Vytvorenie používateľského rozhrania vytváranej aplikácie

Pre vytvorenie používateľského rozhrania aplikácie v jazyku python je možné použiť knižnicu tkinter. Vzhľadom k tomu, že aplikácia bude umožňovať analyzovať rôzne situácie, tak bude v sebe obsahovať viacero záložiek umožňujúcich ovládať jednotlivé funkcionality. Pre tento účel bude využitý objekt typu Notebook. Nasledujúci zdrojový kód znázorňuje vytvorenie spomínaného používateľského rozhrania:

```
from tkinter import ttk
import tkinter as tk

if __name__ == '__main__':
    root = tk.Tk()
    root.title("Network Security Tool")
    root.geometry("700x300")

    tabControl = ttk.Notebook(root)
    tab1 = ttk.Frame(tabControl)
    tab2 = ttk.Frame(tabControl)
    tab3 = ttk.Frame(tabControl)
    tab4 = ttk.Frame(tabControl)
    tab5 = ttk.Frame(tabControl)
    tabControl.add(tab1, text="Create Table")
    tabControl.add(tab2, text="Capture Data")
    tabControl.add(tab3, text="Analyze Data")
    tabControl.add(tab4, text="ML-part1")
    tabControl.add(tab5, text="ML-part2")
    tabControl.pack(expand=1, fill="both")

    root.mainloop()
```

Po spustení uvedeného zdrojového kódu sa zobrazí nasledujúce okno vytváranej aplikácie:



Samotné okno potrebuje obsahovať základné elementy ako sú text, používateľský vstup a tlačidlá, používateľské rozhranie obsahujúce všetky potrebné elementy pre interakciu s používateľom je možné vidieť na nasledujúcom zdrojovom kóde:

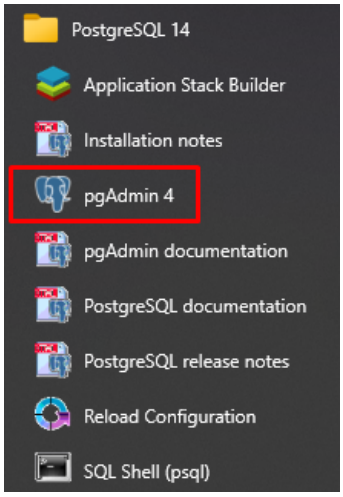
```
def t1_create_table():
     pass


def t2_ICMP_Packet_Capture():
     pass


def t3_insert_to_table(table_name, src_mac, dst_mac, src_ip,
dst_ip, icmp_type, payload):
     pass


def t4_ICMP_Packet_Analyze():
     pass


def t5_ML_DataSet():
     pass


def t6_ICMP_Packet_Clasification():
     pass


if __name__ == '__main__':
     root = tk.Tk()
     root.title("Network Security Tool")
     root.geometry("700x300")

     tabControl = ttk.Notebook(root)
     tab1 = ttk.Frame(tabControl)
     tab2 = ttk.Frame(tabControl)
     tab3 = ttk.Frame(tabControl)
     tab4 = ttk.Frame(tabControl)
     tab5 = ttk.Frame(tabControl)
     tabControl.add(tab1, text="Create Table")
     tabControl.add(tab2, text="Capture Data")
     tabControl.add(tab3, text="Analyze Data")
     tabControl.add(tab4, text="ML-part1")
     tabControl.add(tab5, text="ML-part2")
     tabControl.pack(expand=1, fill="both")

     table_name_value = tk.StringVar(root, "")
     ttk.Label(tab1, text="Creating Table for Storing
Data").grid(columnspan=2, row=0)
```

```
     ttk.Label(tab1, text="Set Table Name:").grid(column=0, row=1)
     ttk.Entry(tab1, textvariable=table_name_value).grid(column=1,
row=1)
     ttk.Button(tab1, text="Create Table",
command=t1_create_table).grid(column=0, row=4)

     number_of_icmp_msg_value = tk.StringVar(root, "")
     T2_table_name_value = tk.StringVar(root, "")
     ttk.Label(tab2, text="Capturing ICMP
messages").grid(columnspan=2, row=0)
     ttk.Label(tab2, text="Set Packet Count:").grid(column=0,
row=1)
     ttk.Entry(tab2,
textvariable=number_of_icmp_msg_value).grid(column=1, row=1)
     ttk.Label(tab2, text="Set Table Name:").grid(column=0, row=2)
     ttk.Entry(tab2,
textvariable=T2_table_name_value).grid(column=1, row=2)
     ttk.Button(tab2, text="Start Capture",
command=t2_ICMP_Packet_Capture).grid(column=0, row=3)

     T3_secret_string_value = tk.StringVar(root, "")
     T3_table_name_value = tk.StringVar(root, "")
     ttk.Label(tab3, text="Analyze ICMP
messages").grid(columnspan=2, row=0)
     ttk.Label(tab3, text="Set Secret Word to
Find:").grid(column=0, row=1)
     ttk.Entry(tab3,
textvariable=T3_secret_string_value).grid(column=1, row=1)
     ttk.Label(tab3, text="Set Table Name:").grid(column=0, row=2)
     ttk.Entry(tab3,
textvariable=T3_table_name_value).grid(column=1, row=2)
     ttk.Button(tab3, text="Analyze",
command=t4_ICMP_Packet_Analyze).grid(column=0, row=3)

     ttk.Label(tab4, text="DataSet Creation").grid(columnspan=2,
row=0)
     T4_table_name_value = tk.StringVar(root, "")
     T4_danger_string_value = tk.StringVar(root, "")
     T4_final_table_with_lable_value = tk.StringVar(root, "")
     ttk.Label(tab4, text="Set Table Name:").grid(column=0, row=1)
     ttk.Entry(tab4,
textvariable=T4_table_name_value).grid(column=1, row=1)
     ttk.Label(tab4, text="Set Danger Payload:").grid(column=0,
row=2)
     ttk.Entry(tab4,
textvariable=T4_danger_string_value).grid(column=1, row=2)
     ttk.Label(tab4, text="Set Final Table Name
(labeled):").grid(column=0, row=3)
     ttk.Entry(tab4,
textvariable=T4_final_table_with_lable_value).grid(column=1, row=3)
     ttk.Button(tab4, text="Create DataSet",
```

```
command=t5_ML_DataSet).grid(column=0, row=4)

      T5_src_mac_value = tk.StringVar(root, "")
      T5_dst_mac_value = tk.StringVar(root, "")
      T5_src_ip_value = tk.StringVar(root, "")
      T5_dst_ip_value = tk.StringVar(root, "")
      T5_icmp_type_value = tk.StringVar(root, "")
      T5_payload_value = tk.StringVar(root, "")
      T5_train_table_value = tk.StringVar(root, "")

      ttk.Label(tab5, text="Classification of ICMP messages using
ML").grid(columnspan=2, row=0)
      ttk.Label(tab5, text="Set Source MAC address:").grid(column=0,
row=1)
      ttk.Entry(tab5, textvariable=T5_src_mac_value).grid(column=1,
row=1)
      ttk.Label(tab5, text="Set Destination MAC
address:").grid(column=0, row=2)
      ttk.Entry(tab5, textvariable=T5_dst_mac_value).grid(column=1,
row=2)
      ttk.Label(tab5, text="Set Source IP address:").grid(column=0,
row=3)
      ttk.Entry(tab5, textvariable=T5_src_ip_value).grid(column=1,
row=3)
      ttk.Label(tab5, text="Set Destionation IP
address:").grid(column=0, row=4)
      ttk.Entry(tab5, textvariable=T5_dst_ip_value).grid(column=1,
row=4)
      ttk.Label(tab5, text="Set ICMP Type:").grid(column=0, row=5)
      ttk.Entry(tab5,
textvariable=T5_icmp_type_value).grid(column=1, row=5)
      ttk.Label(tab5, text="Set Payload:").grid(column=0, row=6)
      ttk.Entry(tab5, textvariable=T5_payload_value).grid(column=1,
row=6)
      ttk.Label(tab5, text="Set Train Table Name:").grid(column=0,
row=7)
      ttk.Entry(tab5,
textvariable=T5_train_table_value).grid(column=1, row=7)
      ttk.Button(tab5, text="Start Clasification",
command=t6_ICMP_Packet_Clasification).grid(column=0, row=8)

      root.mainloop()
```

Nasledujúca časť tohto seminára sa bude zaoberať ukážkou práce s PostgreSQL databázou a strojovým učením. Pre tento účel je preto potrebné najskôr nainštalovať PostgreSQL databázu, pre ktorej správu je možné stiahnuť nástroje z: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads

Po jej stiahnutí a inštalácií je možné v časti Start nájsť nasledujúce súbory, pričom cez označený súbor je možné využitím grafického rozhrania pracovať s databázou:

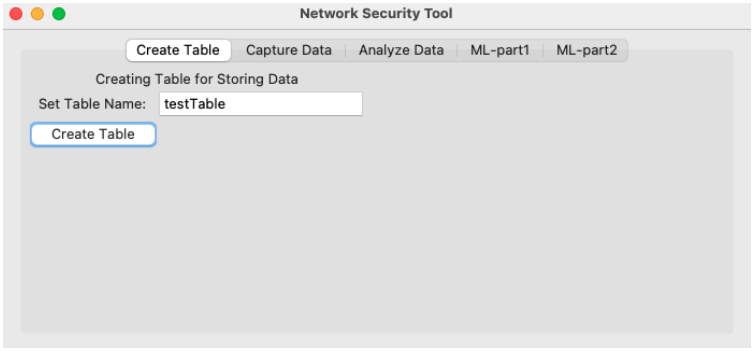


Prvotné spustenie bude vyžadovať nastavenie prihlasovacích údajov a vytvorenie pracovnej databázy.

Využitím uvedeného nástroja je možné pracovať priamo s tabuľkami, ktoré je možné nájsť v stromovej štruktúre zvýraznenej v ľavej časti **Servers -> PostgreSQL -> Databases -> [DB_name] -> Schemas -> Tables**

Úloha 2: Vytvorenie tabuľky v databáze

Prvou úlohou je vytvorenie tabuľky v databáze. Grafické rozhranie aplikácie v tomto prípade bude potrebovať načítať názov tabuľky. Do tabuľky budú následne ukladané dáta o dátovom toku ICMP správ a teda vytvorená tabuľa bude obsahovať polia pre [**src/dst MAC, src/dst IP, ICMP type, Payload**]. Používateľské rozhranie aplikácie je znázornené na nasledujúcom obrázku:



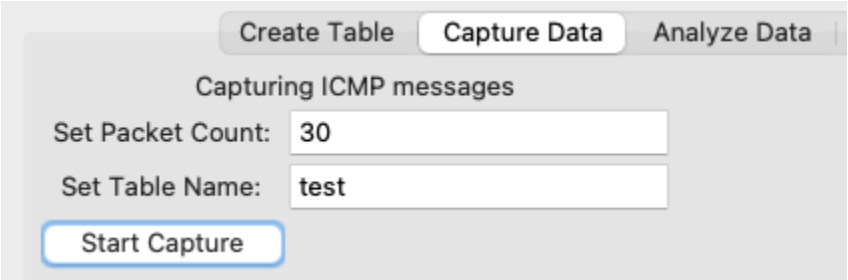
Funkcia pre vytvorenie tabuľky je znázornená v nasledujúcom výpise:

```
def t1_create_table():
      global connection, cursor
      name = table_name_value.get()
      try:
      connection = psycopg2.connect(user="postgres",
                                    password="DBpass123!",
                                    host="127.0.0.1",
                                    port="5432",
                                    database="test_DB")
      cursor = connection.cursor()

      create_table_query = """CREATE TABLE """ + name + """ (
                msg_id SERIAL PRIMARY KEY,
                src_mac VARCHAR(17) NOT NULL,
                dst_mac VARCHAR(17) NOT NULL,
                src_ip VARCHAR(15) NOT NULL,
                dst_ip VARCHAR(15) NOT NULL,
                icmp_type VARCHAR(1) NOT NULL,
                payload VARCHAR(32) NOT NULL
            )"""
      cursor.execute(create_table_query)
      connection.commit()

      except (Exception, psycopg2.Error) as error:
      print("Error while connecting to PostgreSQL", error)
      finally:
      if connection:
            cursor.close()
            connection.close()
            print("PostgreSQL connection is closed")
```

V rámci funkcie sa najskôr realizuje pripojenie k databáze využitím autentifikačných údajov. Následne sa vytvorí SQL dopyt pre vytvorenie tabuľky, ktorý sa následne vykoná využitím metód execute a comit. Posledným krokom je uzavretie spojenia s databázou. Úspešné vytvorenie tabuľky je možné vidieť priamo v grafickom rozhraní kliknutím na novo vytvorenú tabuľku pravým tlačidlom a voľbou **View/Edit data -> All Rows**, čo je znázornené na nasledujúcom obrázku:

Úloha 3: Vloženie dát do tabuľky

Po vytvorení tabuľky je možné začať do nej vkladať dáta. Pre tento účel najskôr vytvoríme funkciu pre odchytávania ICMP správ, z ktorých budú následne získané, pre nás zaujímavé charakteristiky, ktoré budú vložené do tabuľky v databáze. Používateľské rozhranie by malo obsahovať vstup pre identifikovanie názvu tabuľky, do ktorej budú dáta vkladané a množstvo dát, ktoré chceme odchytiť. Ukážka používateľského rozhrania je znázornená na nasledujúcom obrázku:



Pre odchytenie dát je možné použiť funkciu uvedenú v nasledujúcom výpise:

```
def t2_ICMP_Packet_Capture():
      table = T2_table_name_value.get()
      packets = sniff(filter="icmp",
count=int(number_of_icmp_msg_value.get()))

      for packet in packets:
      src_mac = packet.getlayer(Ether).src
      dst_mac = packet.getlayer(Ether).dst
      src_ip = packet.getlayer(IP).src
      dst_ip = packet.getlayer(IP).dst
      icmp_type = packet.getlayer(ICMP).type
      payload = packet.getlayer(Raw).load.decode()
      print(src_mac, dst_mac, src_ip, dst_ip, icmp_type, payload)
      t3_insert_to_table(table, src_mac, dst_mac, src_ip, dst_ip,
icmp_type, payload)
```

Uvedená funkcia najskôr získa názov cieľovej tabuľky a začne odchytávať požadované množstvo ICMP správ. Po odchytení budú z každého odchyteného paketu získané požadované údaje, ktoré sú následne vložené do volania funkcie pre zápis záznamu do databázy.

Funkcia pre zápis dát do databázy je znázornená v nasledujúcom výpise:

```
def t3_insert_to_table(table_name, src_mac, dst_mac, src_ip,
dst_ip, icmp_type, payload):
      global connection, cursor
      try:
      connection = psycopg2.connect(user="postgres",
                                    password="DBpass123!",
                                    host="127.0.0.1",
                                    port="5432",
                                    database="test_DB")
      cursor = connection.cursor()
```

```
      # Executing a SQL query to insert data into  table
      insert_query = """INSERT INTO """ + table_name + """
            (src_mac, dst_mac, src_ip, dst_ip, icmp_type, payload)
            VALUES (%s, %s, %s, %s, %s, %s)
            """
      value_tuple = (src_mac, dst_mac, src_ip, dst_ip, icmp_type,
payload)
      cursor.execute(insert_query, value_tuple)
      connection.commit()
      print("1 Record inserted successfully")
      # Fetch result
      cursor.execute("SELECT * from " + table_name)
      record = cursor.fetchall()

      except (Exception, psycopg2.Error) as error:
      print("Error while connecting to PostgreSQL", error)
      finally:
      if connection:
            cursor.close()
            connection.close()
            print("PostgreSQL connection is closed")
```

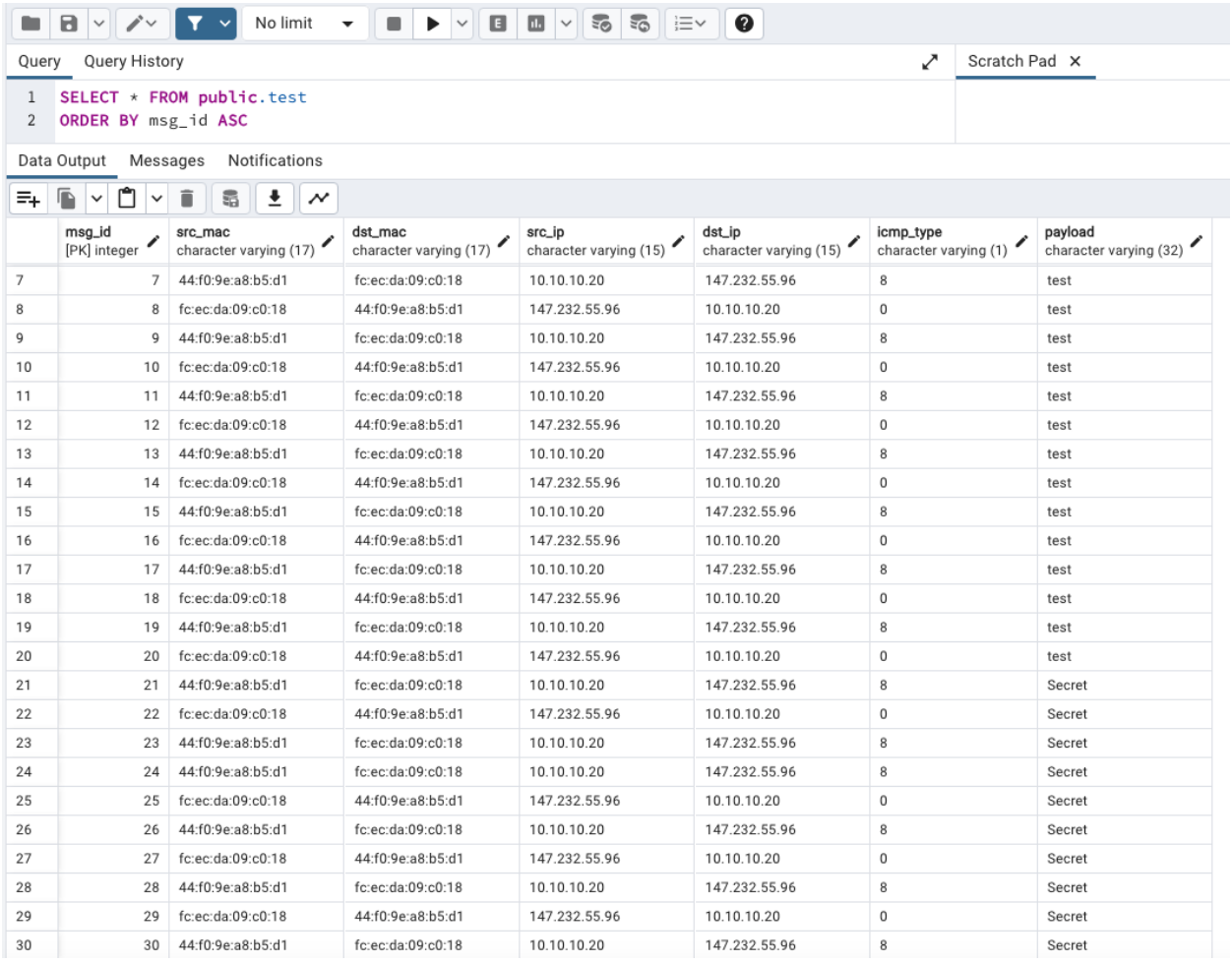
Podobne ako pri vytváraní tabuľky, aj tu najskôr dôjde k nadviazaniu spojenia. Následne sa vytvorí SQL dotaz pre vloženie dát do tabuľky, kde %s označíme množinu hodnôt, ktorú chceme vložiť. Následne vytvoríme tuple, do ktorého budú vložené reálne hodnoty. Následne metóde execute vložíme ako argumenty vytvorený SQL dotaz a hodnoty, ktoré chceme zapísať. Pre kontrolu úspešnosti zapísania záznamu je možné skúsiť dané záznamy z tabuľky vyčítať využitím selektu (tento krok nie je nutný). Nakoniec je spojenie s databázou uzavrené.

Pre generovanie ICMP správ si vytvorme na to určenú aplikáciu, ktorej zdrojový kód môže byť nasledovný:

```
def create_ping():
      for i in range(0, 10):
      icmp = IP(dst='cnl.sk') / ICMP() / 'test'
      resp = sr1(icmp, timeout=2)

      for i in range(0, 10):
      icmp = IP(dst='cnl.sk') / ICMP() / 'Secret'
      resp = sr1(icmp, timeout=2)

      for i in range(0, 10):
      icmp = IP(dst='cnl.sk') / ICMP() / 'test'
      resp = sr1(icmp, timeout=2)

if __name__ == '__main__':
      create_ping()
```

Teraz spustite aplikáciu na odchytávanie 30 ICMP správ a následne týchto 30 správ vygenerujte vyššie uvedeným kódom.

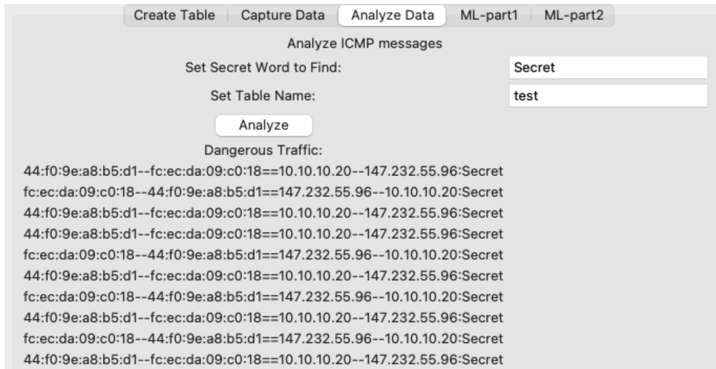
Overenie úspešnosti zápisu záznamov do tabuľky je možné zobraziť aj v grafickom rozhraní rovnakým spôsobom ako v úlohe 2 čo znázorňuje nasledujúci obrázok:



	msg_id [PK] integer	src_mac character varying (17)	dst_mac character varying (17)	src_ip character varying (15)	dst_ip character varying (15)	icmp_type character varying (1)	payload character varying (32)
7	7	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
8	8	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
9	9	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
10	10	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
11	11	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
12	12	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
13	13	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
14	14	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
15	15	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
16	16	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
17	17	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
18	18	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
19	19	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test
20	20	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test
21	21	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret
22	22	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	Secret
23	23	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret
24	24	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret
25	25	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	Secret
26	26	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret
27	27	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	Secret
28	28	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret
29	29	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	Secret
30	30	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret

Úloha 4: Statické vyhľadávanie nebezpečnej komunikácie v tabuľke

Nebezpečnú komunikáciu (obsahujúcu citlivé dáta) je možné vyhľadávať manuálnym prehľadávaním záznamov v tabuľke a čítaním payloadu. Pre tento účel je možné využiť nasledujúce používateľské rozhranie, ktoré od používateľa vyžiada hľadané citlivé slovo, a názov tabuľky, v ktorej chceme vyhľadávať:



Ak niektorý zo záznamov obsahoval citlivé údaje, tak sa záznam o tejto komunikácií vypíše do používateľského rozhrania aplikácie. Týmto sme vytvorili jednoduchý detekčný nástroj. Funkcia pre uvedený proces je znázornená v nasledujúcom výpise:

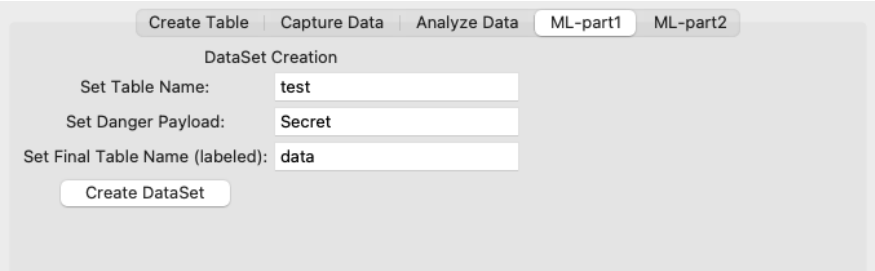
```
def t4_ICMP_Packet_Analyze():
      global connection, cursor
      try:
      connection = psycopg2.connect(user="postgres",
                                        password="DBpass123!",
                                        host="127.0.0.1",
                                        port="5432",
                                        database="test_DB")
      cursor = connection.cursor()
      cursor.execute("SELECT * from " + T3_table_name_value.get())
      records = cursor.fetchall()
      index = 5
      for record in records:
            if record[6].find(T3_secret_string_value.get()) != -1:
                  ttk.Label(tab3, text="Dangerous
Traffic:").grid(column=0, row=4)
                  ttk.Label(tab3,
text=f"{record[1]}--{record[2]}=={record[3]}--{record[4]}:{record[6
]}").grid(column=0,

      row=index)
                  print(record[3], record[4])
                  index += 1
      except (Exception, psycopg2.Error) as error:
      print("Error while connecting to PostgreSQL", error)
      finally:
      if connection:
            cursor.close()
```

```
        connection.close()
        print("PostgreSQL connection is closed")
```

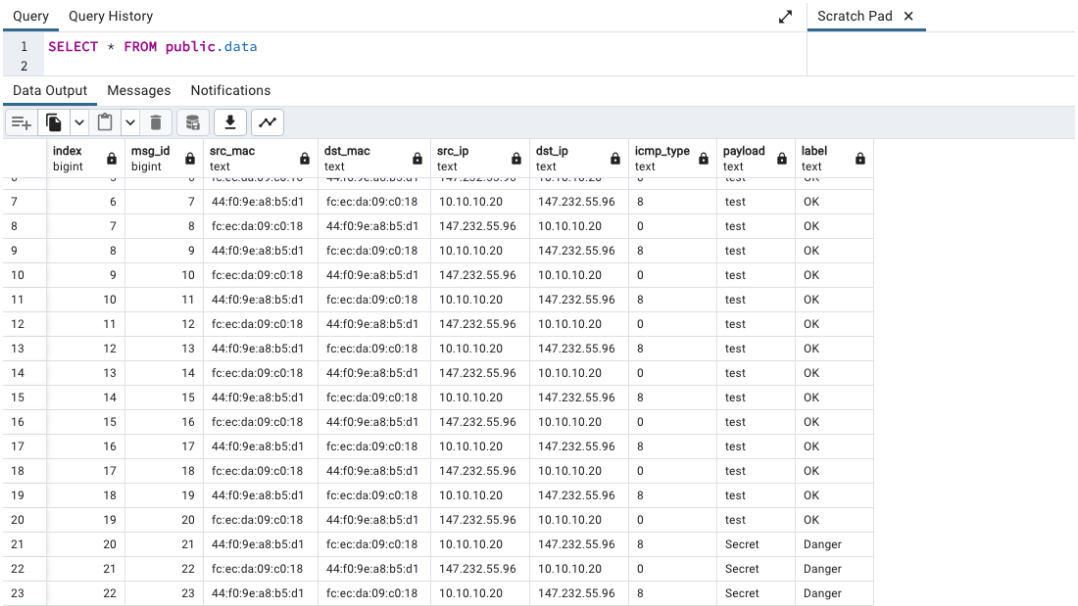
V tomto prípade sa stačí pripojiť do databázy, vytvoriť SQL dotaz (selekt na danú tabuľku) a získať všetky záznamy z tabuľky. Následne stačí prejsť každým záznamom, v ktorom v stĺpci payload vyhľadáme hľadaný string. Ak ho daný záznam obsahuje, tak sú do okna aplikácie vypísané informácie o celej komunikácií. Nakoniec je štandardne zatvorené spojenie s databázou.

Úloha 5: Vytvorenie dátovej sady pre neskoršiu analýzu strojovým učením

Pre jednoduchosť využijeme v tejto úlohe štandardnú metódu strojového učenia a to **Učenie s učiteľom**. V tejto metóde je potrebné vytvoriť označenú dátovú sadu, v ktorej bude povedané, ktoré dáta strojového učenia predstavujú hrozbu, a ktoré sú bežné. Najskôr teda vykonáme štandardnú statickú analýzu, ktorej výsledkom bude vytvorenie nového záznamu obohateného o jeden nový stĺpec, v ktorom sa bude nachádzať značka. Používateľské rozhranie pre tento účel vyžaduje získať informáciu o pôvodnej tabuľke, reťazci, na základe ktorého označíme záznamy za nebezpečné a nakoniec názov novej tabuľky ktorú vytvoríme a vložíme do nej novo vytvorené záznamy. Možné používateľské rozhranie znázorňuje nasledujúci obrázok:



Novo vytvorená tabuľka je zobrazená v nasledujúcom obrázku:



index bigint	msg_id bigint	src_mac text	dst_mac text	src_ip text	dst_ip text	icmp_type text	payload text	label text
7	6	7	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
8	7	8	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
9	8	9	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
10	9	10	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
11	10	11	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
12	11	12	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
13	12	13	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
14	13	14	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
15	14	15	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
16	15	16	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
17	16	17	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
18	17	18	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
19	18	19	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	test	OK
20	19	20	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	test	OK
21	20	21	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret	Danger
22	21	22	fc:ec:da:09:c0:18	44:f0:9e:a8:b5:d1	147.232.55.96	10.10.10.20	0	Secret	Danger
23	22	23	44:f0:9e:a8:b5:d1	fc:ec:da:09:c0:18	10.10.10.20	147.232.55.96	8	Secret	Danger

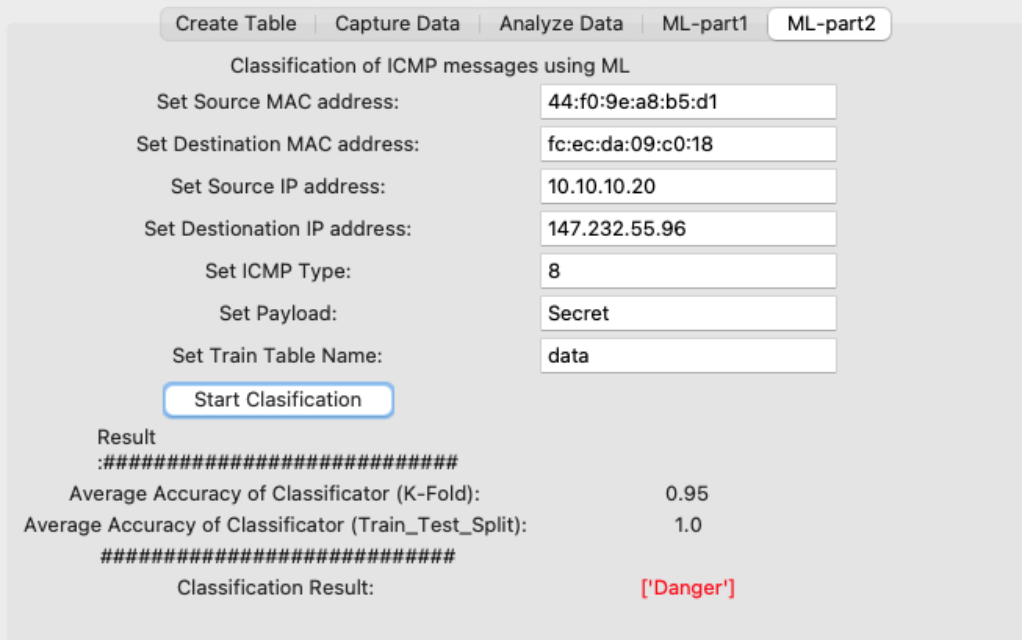
Funkcia, ktorá danú funkcionalitu realizuje sa dá vytvoriť z funkcií implementovaných v predchádzajúcich úlohách. Jej finálnu podobu znázorňuje nasledujúci výpis:

```
def t5_ML_DataSet():
     connection = psycopg2.connect(user="postgres",
                                   password="DBpass123!",
                                   host="127.0.0.1",
                                   port="5432",
                                   database="test_DB")
     df = pd.read_sql("select * from " + T4_table_name_value.get(),
con=connection)
     print(df)
     mark = []
     cursor = connection.cursor()
     cursor.execute("SELECT * from " + T4_table_name_value.get())
     records = cursor.fetchall()
     for record in records:
     if record[6].find(T4_danger_string_value.get()) != -1:
          mark.append("Danger")
     else:
          mark.append("OK")

     df2 = df.assign(label=mark)
     engine =
create_engine("postgresql://postgres:DBpass123!@localhost:5432/test
_DB")
     df2.to_sql(T4_final_table_with_lable_value.get(), engine,
if_exists='replace')
     print(df2)
```

Uvedená funkcia najskôr vytvorí spojenie s databázou. Následne využitím knižnice **pandas** vykonáme nad databázou selekt pre získanie všetkých záznamov a ich uloženie do dátovej štruktúry dataFrame. Ďalej je vytvorené prázdne pole pre ukladanie značiek pre každý analyzovaný záznam. Pre jednoduchosť vykonáme totožný selekt na vyčítanie záznamov z tabuľky. Vo for cykle sa prejde každým záznamom, a ak obsahuje nebezpečný string, tak sa do poľa značiek uloží hodnota „Danger“, v opačnom prípade hodnota „OK“. Posledným krokom je vytvorenie novej pandas dataFrame, do ktorej sa uloží pôvodne načítaná tabuľka obohatená o nový stĺpec label, do ktorého sa vložia záznamy z vytvoreného poľa značiek. Po vytvorení novej verzie dataFrame stačí uvedený dataFrame zapísať do novej tabuľky v databáze. V našom prípade po každom volaní bude stále stará tabuľka prepísaná (spôsobené príznakom „replace“). Táto novo vytvorená tabuľka bude slúžiť ako vstup pre spracovanie strojovým učením – učenie s učiteľom.

Úloha 6: Trénovanie modelu a klasifikácia nebezpečnej prevádzky

V rámci tejto úlohy budú demonštrované rôzne procesy potrebné pre trénovanie modelu strojového učenia s následnou klasifikáciou. Pozornosť bude venovaná hlavne procesu predspracovania dát (kódovanie = transformácia reťazcov do číselnej podoby), trénovaniu, vyhodnoteniu úspešnosti vytvoreného modelu a samotnú klasifikáciu novo získaných dát. Nové dáta je možné získať štandardným odchytávaním dát zo sieťovej karty, no pre demonštráciu a jednoduchosť zadá používateľ informácie o dátovom toku priamo do aplikácie, ktorá následne vypíše presnosť natrénovaného modelu a samotnú predikciu. Nasledujúci obrázok znázorňuje možné používateľské rozhranie:



Keďže procesov potrebných pre ukážku strojového učenia je viac, tak aj ich vysvetlenie bude ukázané postupne s podrobnejším vysvetlením úryvkov zdrojového kódu.

```
def t6_ICMP_Packet_Clasification():
      global connection
      ttk.Label(tab5, text="                          ",
background="white").grid(column=1, row=10)
      ttk.Label(tab5, text="                          ",
background="white").grid(column=1, row=11)
      ttk.Label(tab5, text="                          ",
background="white").grid(column=1, row=13)
      try:
      connection = psycopg2.connect(user="postgres",
                                    password="DBpass123!",
                                    host="127.0.0.1",
                                    port="5432",
                                    database="test_DB")
      df = pd.read_sql("select * from " +
T5_train_table_value.get(), con=connection)
```

```
     my_test_json = {
          "src_mac": [T5_src_mac_value.get()],
          "dst_mac": [T5_dst_mac_value.get()],
          "src_ip": [T5_src_ip_value.get()],
          "dst_ip": [T5_dst_ip_value.get()],
          "icmp_type": [T5_icmp_type_value.get()],
          "payload": [T5_payload_value.get()]
     }
     my_test_df = pd.DataFrame(my_test_json)
```

Prvotným krokom je vytvorenie spojenia s databázou a získanie obsahu tabuľky s označenými dátami. Následne si do premennej typu slovník uložíme dáta, ktoré zadal používateľ do databázy a prevedieme ich do formátu pandas dataFrame (potrebné pre spracovanie modelom strojového učenia).

Ďalším krokom je vytvorenie dátovej sady, ktorá bude obsahovať všetky dátové toky (vrátane 1 riadka zadaného od používateľa) bez značky. Takto vytvorené dáta (X časť) je potrebné previesť do podoby čísel (pôvodné dáta sú v podobe reťazcov). Tento proces sa nazýva kódovanie dát, na čo je možné použiť encoder. Časť kódu vykonávajúci uvedenú funkcionalitu je znázornený v nasledujúcom výpise:

```
x_matrix = df.iloc[:, 2:8]
     x_matrix = pd.concat([x_matrix, my_test_df],
ignore_index=True, axis=0)

     column_header = ["src_mac", "dst_mac", "src_ip", "dst_ip",
"payload"]
     for column in column_header:
          enc = LabelEncoder()
          label_encoder = enc.fit(x_matrix[column])
          x_matrix[column] =
label_encoder.transform(x_matrix[column])

     my_test_df = x_matrix.tail(1)
     x_matrix = x_matrix.iloc[:-1, :]

     y_vector = np.array(df.label)
```

Uvedený zdrojový kód je možné interpretovať nasledovne. Najskôr z načítaných dát (tabuľka) zoberieme všetky riadky a stĺpce od 2 po 7. K uvedeným dátam pridáme nový riadok zadaný používateľom. Ďalej sa vytvorí pole s názvami stĺpcov. Cez for cyklus sa prejde každým stĺpcom, kde sa zakaždým vytvorí nová inštancia enkódera, ktorý sa najskôr natrénuje na stĺpci a následne sa vykoná transformácia dát z reťazca do podoby číselnej hodnoty. Po prevedení dát do čísel znova spätne odoberieme posledný riadok (používateľom zadaný záznam), ktorý uchováme v samostatnej premennej (my_test_df). Tiež vytvoríme premennú y_vector, ktorý bude obsahovať stĺpec značiek.

Ďalším krokom je rozdelenie dát na trénovacie a testovacie (zvyčajne v pomere 70:30), na ktorých sa overí, do akej miery je model spoľahlivý pre dátovú sadu, s ktorou sa pracuje. Vyhodnotenie je možné vďaka tomu, že k trénovacím aj testovacím dátam poznáme aj výslednú značku. Na trénovacích dátach model natrénujeme (obsahujú aj X-dáta aj Y-značky). Následne sa vykoná predikcia na testovacích dátach. Model po klasifikácií vygeneruje predikované značky, ktoré je možné porovnať s reálnymi značkami testovacích dát. Ďalší spôsob vyhodnotenia modelu je využitie cross_validácie, ktorá využíva rozdelenie dát do viacerých „foldov“ a trénovanie a predikcia sa deje na rôznej kombinácií dát. Zdrojový kód pre rozdelenie dát, trénovanie klasifikačného modelu (Random forest) a vyhodnotenie presnosti je znázornené na nasledujúcom výpise:

```
X_train, X_test, y_train, y_test = train_test_split(x_matrix,
y_vector, test_size=0.25, random_state=0)

     rf = RandomForestClassifier(criterion="entropy", max_depth=5,
n_estimators=5)
     scores = model_selection.cross_val_score(rf, X_train, y_train,
cv=5, n_jobs=-1)
     # vypis testovani jednotlivych foldov
     print("\nPriemerne score najlepsieho modelu: " +
str(np.mean(scores)))

     # predikcia a vratenie vysledneho vektora pre testovacie data
     rf.fit(X_train, y_train)
     final_predict = rf.predict(X_test)
     tts_acc = (y_test == final_predict).mean()
     print("Total Correct:\t", (y_test == final_predict).sum())
```

Posledným krokom je ukázať natrénovanému modelu nové (používateľom zadané) dáta, na ktorých sa vykoná predikcia a vygeneruje sa značka. Táto značka sa následne vypíše do používateľského rozhrania spolu s presnosťou predikcie, čo znázorňuje nasledujúci výpis:

```
# predikcia na novej vzorke
     my_value = rf.predict(my_test_df)
     print("Predikcia je:", my_value)
     ttk.Label(tab5,
text="Result\n:############################").grid(column=0, row=9)
     ttk.Label(tab5, text="Average Accuracy of Classificator
(K-Fold): ").grid(column=0, row=10)
     ttk.Label(tab5, text=f"{str(np.mean(scores))}").grid(column=1,
row=10)
     ttk.Label(tab5, text="Average Accuracy of Classificator
(Train_Test_Split): ").grid(column=0, row=11)
     ttk.Label(tab5, text=f"{tts_acc}").grid(column=1, row=11)
     ttk.Label(tab5,
text="############################").grid(column=0, row=12)
     ttk.Label(tab5, text="Classification Result: ").grid(column=0,
row=13)
     if my_value == "OK":
```

```
          ttk.Label(tab5, text=f"{my_value}", background="green",
foreground="white").grid(column=1, row=13)
     else:
          ttk.Label(tab5, text=f"{my_value}", background="red",
foreground="red").grid(column=1, row=13)


     except (Exception, psycopg2.Error) as error:
     print("Error while connecting to PostgreSQL", error)
     finally:
     if connection:
          print("PostgreSQL connection is closed")
```

Bonus: Tvorba vlastnej dátovej jednotky bez použitia knižnice Scapy

Tvorba vlastnej dátovej jednotky a jej následné odoslanie vyžaduje väčšie oprávnenia, a preto je potrebné pre túto úlohu pracovať v operačnom systéme Linux a spúšťať skript so sudo právami. Pracovať sa bude so Socket-om. Výpis zobrazuje základnú štruktúru tvorby dátovej jednotky:

```
import socket import binascii
if __name__ == '__main__':
    s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
    s.bind(("enp0s3", 0))

    ethernet_header = build_ethernet_header("11:22:33:11:22:33",
"aa:bb:cc:aa:bb:cc", "ipv4")

    icmp_header = build_icmp_echo_request_header("TEST")
    ipv4_header = build_ipv4_header("10.20.30.40", "22.22.22.22",
"icmp", icmp_header)

    packet = ethernet_header + ipv4_header + icmp_header

    s.send(packet)
```

Ako je možné vidieť, tak najskôr sa vytvorí objekt pre socket, ktorému nastavíme typ na RAW a spojíme ho so sieťovým rozhraním, cez ktoré budeme chcieť komunikovať. Ďalej sa postupne vytvoria Ethernetová hlavička, IP hlavička, ICMP hlavička s payload-om. Nakoniec sa vytvorené hlavičky zreťazia a výsledný vytvorený paket sa odošle využitím metódy send vytvoreného socketu.

Tvorba jednotlivých hlavičiek bola implementovaná v podobe samostatných funkcií, ktoré dostávajú ako argument isté požadované argumenty ako MAC/IP/Payload.

Najskôr sa vytvorí Ethernetová hlavička, ktorej tvorba je uvedená v nasledujúcom výpise:

```
def build_ethernet_header(dst_mac, src_mac, protocol_type):
     dst_mac_byte = binascii.unhexlify(dst_mac.replace(":", ""))
     src_mac_byte = binascii.unhexlify(src_mac.replace(":", ""))

     pt_options = {
     "ipv4": b'\x08\x00',
     "ipv6": b'\x86\xdd',
     "arp": b'\x08\x06'
     }
     pt_byte = pt_options[protocol_type]

     created_ethernet_header = dst_mac_byte + src_mac_byte +
pt_byte
     return created_ethernet_header
```

Vytváranie hlavičiek je možné postupným spájaním dát v bytes type. Potrebné je teda podľa štandardu vytvoriť dáta pre vyplnenie jednotlivých polí hlavičiek. Uvedená funkcia najskôr odstráni z MAC adries dvojbodku a prevedie do byte podoby. Následne sa vytvorí premenná pre uchovanie hodnoty pre typ protokolu vyššej vrstvy. Vo výsledku stačí spojiť hodnoty pre cieľovú a zdrojovú MAC adresu a typ. Funkcia na konci vráti vytvorenú dátovú jednotku.

Pri tomto spôsobe tvorby paketu je vhodné po implementácií každého ďalšieho poľa odoslať dátovú jednotku a overiť v nástroji Wireshark, či rozoznáva zatiaľ vytvorené polia. Ďalšou implementovanou funkciou je funkcia pre tvorbu IP hlavičky, ktorá je uvedená v nasledujúcom výpise:

```
def build_ipv4_header(src_ip, dst_ip, protocol, packet_payload):
     version_and_hl_byte = b'\x45'  # version 4 je pre IPv4, HL =
dlzka hlavicky v pocte 32 bit. slov = 5
     tos_byte = b'\x00'  # ToS = Type of Service suvisi s kvalitou
sluzieb
     identification_byte = b'\x00\x00'
     flags_byte = b'\x40\x00'

     tmp_ttl = 13  # vzpocet celkovej dlzky paketu v bajtoch
     ttl_byte = tmp_ttl.to_bytes(1, 'big')  # konverzia celkovej
dlzky do hex na 16bit

     protocol_options = {
     "icmp": 1,
     "tcp": 6,
     "udp": 17
     }
     tmp_protocol = protocol_options[protocol]
     protocol_byte = tmp_protocol.to_bytes(1, 'big')

     tmp_total_length = 20 + len(packet_payload)  # vzpocet
celkovej dlzky paketu v bajtoch
     total_length_byte = tmp_total_length.to_bytes(2, 'big')  #
konverzia celkovej dlzky do hex na 16bit

     header_checksum_byte = b'\x00\x00'

     src_ip_byte = socket.inet_aton(src_ip)
     dst_ip_byte = socket.inet_aton(dst_ip)

     created_ipv4_header = version_and_hl_byte + tos_byte +
total_length_byte + identification_byte + flags_byte + \
                           ttl_byte + protocol_byte +
header_checksum_byte + src_ip_byte + dst_ip_byte

     checksum = compute_header_checksum(created_ipv4_header, 160)
     header_checksum_byte = binascii.unhexlify(hex(int(checksum,
2))[2:])

     created_ipv4_header = version_and_hl_byte + tos_byte +
```

```
total_length_byte + identification_byte + flags_byte + \
                            ttl_byte + protocol_byte +
header_checksum_byte + src_ip_byte + dst_ip_byte

      return created_ipv4_header
```

Ako je možné vidieť aj táto funkcia len postupne vytvára premenné, do ktorých vkladá dáta pre jednotlivé polia hlavičky IP paketu. Netreba zabudnúť na to, že do daných polí je potrebné ukladať dáta v byte formáte, čo znázorňujú používané metódy ako to_bytes, socket.inet_aton a pod. V kóde sa nachádza aj funkcia compute_header_checksum, ktorej implementácia bude uvedená neskôr. Jej úlohou je výpočet kontrolného súčtu. Pri tvorbe IP hlavičky je potrebné najskôr vložiť do poľa kontrolného súčtu samé 0, následne vypočítať kontrolný súčet a nakoniec spätne vložiť do poľa pre kontrolný súčet hodnotu vypočítaného kontrolného súčtu.

Poslednou hlavičkou, ktorú je potrebné vytvoriť je hlavička protokolu ICMP, ktorej tvorba je uvedená v nasledujúcom výpise:

```
def build_icmp_echo_request_header(msg):
      type_byte = b'\x08'
      code_byte = b'\x00'
      checksum_byte = b'\x00\x00'
      identifier_byte = b'\x00\x00'
      sequence_number_byte = b'\x00\x00'
      payload = msg.encode()
      created_icmp_echo_request_header = type_byte + code_byte +
checksum_byte + identifier_byte + sequence_number_byte + \
                                        payload

      checksum =
compute_header_checksum(created_icmp_echo_request_header,
len(created_icmp_echo_request_header)*8)
      checksum_byte = binascii.unhexlify(hex(int(checksum, 2))[2:])

      created_icmp_echo_request_header = type_byte + code_byte +
checksum_byte + identifier_byte + sequence_number_byte + \
                                        payload

      return created_icmp_echo_request_header
```

Aj v tomto prípade stačí postupne vkladať potrebné hodnoty podľa štandardu. Najzložitejšou časťou je výpočet kontrolného súčtu, na výpočet ktorého je možné použiť rovnakú funkciu ako pri výpočte kontrolného súčtu IP hlavičky. Pozrime sa teda na implementáciu funkcie pre výpočet kontrolného súčtu, ktorá je uvedená v nasledujúcom výpise:

```
def compute_header_checksum(created_header, bit_size):
      hex_val = created_header.hex()
```

```
      binary = bin(int(hex_val, 16))[2:].zfill(bit_size)

      header_bin_array = []
      for i in range(int(len(binary) / 16)):
      tmp_str = binary[16 * i:16 * i + 16]
      header_bin_array.append(tmp_str)

      checksum = header_bin_array[0]
      for i in range(len(header_bin_array)-1):
      checksum = compute_checksum(checksum+header_bin_array[i+1],
16)

      # Calculating the complement of sum
      final_checksum = ''
      for i in checksum:
      if i == '1':
            final_checksum += '0'
      else:
            final_checksum += '1'
      return final_checksum
```

Proces výpočtu kontrolného súčtu je tiež možné vyhľadať v štandardoch. Vytvorenú hlavičku je najskôr potrebné previesť do 16 sústavy s následným prevodom do dvojkovej s doplnením 0 pre možnosť rozdelenia dát na 16 bitové slová. Týmto vznikne jeden dlhý prúd núl a jednotiek (celá hlavička zapísaná v dvojkovej sústave). Ďalším krokom je vytvorenie poľa, do ktorého budú ukladané 16 bitové slová z hlavičky. Pre tento účel sa použije cyklus for, ktorý postupne prejde celým obsahom hlavičky, rozdelí ho po 16 bitoch a vloží do vytvoreného poľa. Následne ďalší for cyklus bude postupne brať z uvedeného poľa 16 bitové slová a robiť na nich binárny súčet, ktorý počíta funkcia **compute_checksum**, ktorá v každej iterácií dostane ako argument pôvodný kontrolný 16b súčet a pripočíta k nemu ďalších 16 bitov.

Finálny kontrolný súčet je v podobe komplementu (invertovaná podoba všetkých bitov). Poslednou funkciou je funkcia na spočítanie dvojice k-bitových čísel, ktorej tvar je uvedený v nasledujúcom výpise:

```
def compute_checksum(sent_message, k):
      # Dividing sent message in packets of k bits.
      c1 = sent_message[0:k]
      c2 = sent_message[k:2 * k]

      # Calculating the binary sum of packets
      calc_sum = bin(int(c1, 2) + int(c2, 2))[2:]  # konverzia do 10
sustavy, scitanie a prevod do binarneho tvaru
      # Adding the overflow bits
      if len(calc_sum) > k:
      x = len(calc_sum) - k
      calc_sum = bin(int(calc_sum[0:x], 2) + int(calc_sum[x:],
2))[2:]
```

```
    if len(calc_sum) < k:
    calc_sum = '0' * (k - len(calc_sum)) + calc_sum

    return calc_sum
```

Ako je možné vidieť, tak najskôr dôjde k rozdeleniu 32 bitového čísla na dve polovice. Tieto čísla sa následne vzájomne v binárnej sústave spočítajú. Ak dôjde k presahu, tak sa hodnota presahujúceho bitu pripočíta k pôvodným 16 bitom súčtu. Funkcia vo výsledku vráti 16b hodnotu súčtu dvoch čísel.

Nasledujúci obrázok znázorňuje vytvorenú dátovú jednotku odchytenú nástrojom Wireshark, ktorý potvrdzuje úspešnosť implementácie vytvoreného PDU a výpočtu kontrolných súčtov: