KRYCY Laboratorium 2 + Projekt 2

Budowa systemu analizy sieciowej + PoC

Mateusz Plichta, Kacper Średnicki, Karol Żelazowski

14 lutego 2025

Spis treści

1. Wprowadzenie

Niniejszy raport zawiera opis prac wykonanych w ramach Laboratorium nr 2 i Projektu nr 2 z przedmiotu Kryminalistyka Cyfrowa. W ramach zadania zbudowano system analizy sieciowej spełniający konkretne wymagania i potwierdzono poprawność jego działania realizując PoC. Kolejne sekcje tego raportu odnoszą się do konkretnych wymagań, które spełnia system.

2. Laboratorium 2: Funkcjonalności podstawowe

2.1. Analiza flow (A.1)

Pierwszym zadaniem było użycie biblioteki NFStream do wczytywania pliku pcap. `wget -O malicious_traffic_lab2.pcap https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-15/2013-09-28_capture-win19.pcap` To jest plik, który wykorzystaliśmy na potrzeby analizy flow. Po użyciu NFStream do załadowania pcapa używamy dataframe'u z biblioteki panda do analizy i statystyk. Poniżej fragment jednej z funkcji korzystyjącej ze streamera

```python
streamer = NFStreamer(source=r"malicious_traffic_lab2.pcap",
statistical_analysis = True)
df = load_df(streamer)
```

To jest natomiast format dataframe'u wykorzystywanego na przestrzeni laboratorium. Jest to część wartości, które można zebrać, wszystkie parametry widnieją w dokumentacji NFStream. `https://www.nfstream.org/docs/api#nflow`

```python
def load_df(streamer):
    df = pd.DataFrame([{
        "source_ip": flow.src_ip,
        "destination_ip": flow.dst_ip,
        "source_port": flow.src_port,
        "destination_port": flow.dst_port,
        "protocol": flow.protocol,
        "src_to_dst_bytes": flow.src2dst_bytes,
        "dst_to_src_bytes": flow.dst2src_bytes,
        "total_bytes": flow.bidirectional_bytes,
        "src_to_dst_packets": flow.src2dst_packets,
        "dst_to_src_packets": flow.dst2src_packets,
        "total_packets": flow.bidirectional_packets,
        "timestamp": pd.to_datetime(flow.bidirectional_first_seen_ms, unit='ms'),  # Dodanie kolumny czasu
    } for flow in streamer])
    return df
```

2.2. Analiza flow (A.2)

W kolejnym kroku wykonaliśmy podsumowanie statystyk flow. Wykonaliśmy 2 implementacje funkcji, zarówno takie podstawowe, jak i bardziej zaawansowane patrząc na statystyki pod różnym kątem.

```python
def analyze_flow(df):
    summary = {
        "Total Flows": len(df),
        "Total Bytes Transferred": df["total_bytes"].sum(),
```

```
        "Total Packets Transferred": df["total_packets"].sum(),
        "Top 5 Source IPs": df.groupby("source_ip")
        ["total_packets"].sum().nlargest(5),
        "Top 5 Destination IPs": df.groupby("destination_ip")
        ["total_packets"].sum().nlargest(5),
    }
    print("Podsumowanie statystyk flow:")
    for key, value in summary.items():
        print(f"{key}: {value}")
```

```python
def summary(df):
        # Podstawowe podsumowanie
    summary = {
        "Total Flows": len(df),
        "Total Bytes Transferred": df["total_bytes"].sum(),
        "Total Packets Transferred": df["total_packets"].sum(),
        "Average Bytes per Flow": df["total_bytes"].mean(),
        "Average Packets per Flow": df["total_packets"].mean(),
        "Top 5 Source IPs": df.groupby("source_ip")
        ["total_packets"].sum().nlargest(5),
        "Top 5 Destination IPs": df.groupby("destination_ip")
        ["total_packets"].sum().nlargest(5),
        "Top 5 Protocols (by flows)": df["protocol"].value_counts().nlargest(5),
    }
    print("Podsumowanie statystyk flow:")
    for key, value in summary.items():
        print(f"{key}: {value}")



        # Szczegółowa analiza
    print("\nSzczegółowa analiza:")

    # 1. Analiza średnich bajtów i pakietów na przepływ w zależności od protokołu
    protocol_stats = df.groupby("protocol").agg(
        average_bytes=("total_bytes", "mean"),
        average_packets=("total_packets", "mean"),
        total_flows=("protocol", "count")
    ).sort_values(by="total_flows", ascending=False)
    print("\nŚrednie bajty i pakiety na protokół:")
    print(protocol_stats)

    # 2. Analiza 10 największych przepływów pod względem liczby bajtów
    top_flows_by_bytes = df.nlargest(10, "total_bytes")[[
        "source_ip", "destination_ip", "source_port", "destination_port",
        "total_bytes"
    ]]
    print("\nTop 10 przepływów pod względem liczby bajtów:")
```

```python
    print(top_flows_by_bytes)

    # 4. Analiza adresów IP z największą liczbą przepływów
    top_ips_by_flows = df["source_ip"].value_counts().nlargest(5)
    print("\nTop 5 IP źródłowych z największą liczbą przepływów:")
    print(top_ips_by_flows)

    return summary
```

```
Podsumowanie statystyk flow:
Total Flows: 2237
Total Bytes Transferred: 20099729
Total Packets Transferred: 33022
Average Bytes per Flow: 8985.126955744301
Average Packets per Flow: 14.761734465802414
Top 5 Source IPs: source_ip
10.0.2.119                  17514
fe80::45f6:82cd:198c:6a1a    15505
10.0.2.2                        2
::                              1
```

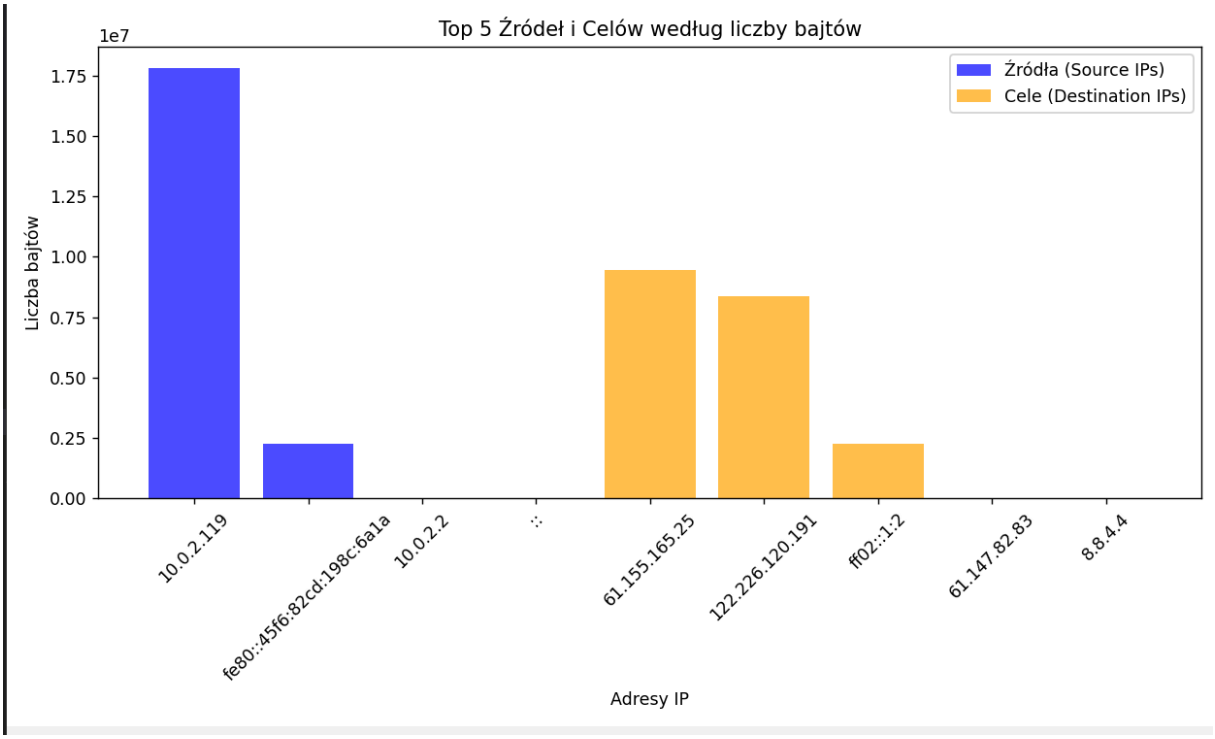
```
Top 5 Destination IPs: destination_ip
ff02::1:2          15498
122.226.120.191     8761
61.155.165.25       8707
8.8.4.4               13
202.108.23.135        10
Name: total_packets, dtype: int64
Top 5 Protocols (by flows): protocol
17    2227
6        4
58       3
1        3
Name: count, dtype: int64
```

```
Szczegółowa analiza:

Średnie bajty i pakiety na protokół:
          average_bytes  average_packets  total_flows
protocol
17         1.024091e+03         6.970813         2227
6          4.454482e+06      4371.750000            4
1          2.273333e+02         1.666667            3
58         1.560000e+02         2.000000            3

Top 10 przepływów pod względem liczby bajtów:
                    source_ip   destination_ip  source_port  destination_port  total_bytes
139                10.0.2.119    61.155.165.25        49161                80      9462820
39                 10.0.2.119  122.226.120.191        49159                80      8351760
17                 10.0.2.119     61.147.82.83        49158                80         2400
0    fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
4    fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
7    fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
10   fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
14   fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
19   fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
20   fe80::45f6:82cd:198c:6a1a        ff02::1:2          546               547         1029
```

numery protokołów zdefiniowane przez IANA (Internet Assigned Numbers Authority):

- 17: To jest numer protokołu dla UDP (User Datagram Protocol).
- 6: To jest numer protokołu dla TCP (Transmission Control Protocol).
- 1: To jest numer protokołu dla ICMP (Internet Control Message Protocol).
- 58: To jest numer protokołu dla ICMPv6 (Internet Control Message Protocol for IPv6).

Oprócz tego wprowadziliśmy jeszcze prostą wizualizację na wykresie podstawowych statystyk.



2.3. Detection as a Code (D.1)

2.3.1. Generowanie ruchu sieciowego

Ruch sieciowy jest generowany na potrzeby tego punktu przez funkcję *generate_traffic()*, która tworzy zestaw pakietów przy użyciu biblioteki *scapy*. Na potrzeby eksperymentu przygotowano symulację następujących scenariuszy:

- przesyłanie danych z adresu źródłowego 10.0.0.1 do 192.168.0.1 przez protokół UDP na port docelowy 3389;
- komunikacja z adresem IP z czarnej listy (122.226.120.191) przez protokół TCP na port 53.

Pakiety są tworzone dynamicznie w pętli, a następnie przechowywane w pamięci w postaci listy. Implementacja opisanej wyżej funkcji została przedstawiona poniżej:

```python
def generate_traffic():
    click.echo("Generowanie ruchu sieciowego...")

    packets = []

    for _ in range(10):
        packet = IP(src="10.0.0.1", dst="192.168.0.1") / UDP(sport=12345,
        dport=3389) / ("X" * 1000)
        packets.append(packet)

    packet = IP(src="10.0.0.3", dst="122.226.120.191") / TCP(sport=55555,
    dport=53) / b"GET / HTTP/1.1\r\nHost: studia.elka.pw.edu.pl\r\n\r\n"
    packets.append(packet)

    click.echo(f"Ruch sieciowy wygenerowany: {len(packets)} pakietów.")
    return packets
```

2.3.2. Implementacja reguł detekcyjnych

Reguły detekcyjne zostały zaimplementowane w dedykowanym pliku *detection_rules.py*, który zawiera klasę *DetectionRule* oraz funkcję *load_rules()*, zwracającą listę wszystkich zdefiniowanych reguł. Każda reguła jest definiowana jako instancja klasy *DetectionRule* i zawiera następujące elementy:

- nazwa reguły: informuje o jej funkcji, np. „Suspicious Port Usage”;
- opis reguły: szczegółowy opis, który wyjaśnia, co reguła analizuje;
- warunek detekcji: funkcja weryfikująca przepływy pod kątem anomalii, np. wyszukiwanie określonych adresów IP lub portów.

Skrypt zawarty w pliku *detection_rules.py* przedstawiono poniżej:

```python
class DetectionRule:

    def __init__(self, name, description, condition):
        self.name = name
        self.description = description
        self.condition = condition

    def detect(self, df):
        anomalies = self.condition(df)
        return anomalies

def load_rules():
    rules = [
        DetectionRule(
            name="Large Data Exfiltration",
            description="Wykrywa przepływy z dużą ilością przesłanych danych.",
            condition=lambda df: df[df['total_bytes'] > 8_000_000]
        ),
        DetectionRule(
            name="Suspicious Port Usage",
            description="Wykrywa użycie podejrzanych portów docelowych.",
            condition=lambda df: df[df['destination_port'].isin([23, 3389, 4444])]
        ),
        DetectionRule(
            name="Communication with Blacklisted IPs",
            description="Wykrywa komunikację z czarną listą IP.",
            condition=lambda df: df[df['destination_ip'].isin(['61.155.165.25',
            '122.226.120.191'])]
        )
    ]
    return rules
```

Jak widać powyżej zdefiniowano trzy reguły detekcyjne:

- Large Data Exfiltration - ta reguła identyfikuje przepływy, w których przesyłane są duże ilości danych (powyżej 8 MB), co może wskazywać na potencjalną kradzież danych (data exfiltration);
- Suspicious Port Usage - ta reguła wykrywa ruch kierowany na podejrzane porty, takie jak 23 (Telnet), 3389 (RDP), czy 4444 (często używany przez malware), co może wskazywać na nieautoryzowany dostęp lub próbę ataku;

- Communication with Blacklisted IPs - ta reguła identyfikuje ruch sieciowy związany z adresami IP zdefiniowanymi jako niebezpieczne lub potencjalnie złośliwe (61.155.165.25, 122.226.120.191), może być użyteczna w identyfikacji zagrożeń takich jak botnety czy połączenia z serwerami C2.

2.3.3. Analiza ruchu i detekcja z wykorzystaniem reguł

Przygotowano funkcję *detect_anomalies_with_rules*, odpowiadającą za wykrywanie anomalii w przepływach sieciowych na podstawie zdefiniowanych reguł. Funkcja:

- wczytuje reguły detekcyjne za pomocą *load_rules()*;
- iteruje przez każdą regułę, stosując jej warunek do danych w formie tabelarycznej (DataFrame);
- zwraca listę alertów zawierających: nazwę reguły, opis reguły, zidentyfikowane anomalie.

Implementacja opisanej wyżej funkcji została przedstawiona poniżej:

```python
def detect_anomalies_with_rules(df):
    rules = load_rules()
    alerts = []
    for rule in rules:
        anomalies = rule.detect(df)
        if not anomalies.empty:
            alerts.append((rule.name, rule.description, anomalies))
    return alerts
```

Powyższa funkcja jest wykorzystywana w funkcji *analyze_traffic* analizującej wygenerowane pakiety, przekształcając je na przepływy, a następnie stosując reguły detekcyjne:

- tworzy przepływy z pakietów (flows) z kluczowymi informacjami, takimi jak: adres źródłowy, port, protokół, liczba bajtów itp;
- przekazuje przepływy jako DataFrame do funkcji *detect_anomalies_with_rules*;
- wyświetla wyniki detekcji w formie alertów lub informuje o braku anomalii.

Istotny fragment funkcji *analyze_traffic* przedstawiono poniżej:

```python
    flows = []
    for packet in packets:
        if IP in packet:
            flow = {
                "source_ip": packet[IP].src,
                "destination_ip": packet[IP].dst,
                "source_port": packet[TCP].sport if TCP in packet else
                packet[UDP].sport,
                "destination_port": packet[TCP].dport if TCP in packet else
                packet[UDP].dport,
                "protocol": "TCP" if TCP in packet else "UDP",
                "total_bytes": len(packet),
                "total_packets": 1,
            }
            flows.append(flow)

    df = pd.DataFrame(flows)
    alerts = detect_anomalies_with_rules(df)
```

Dodatkowo przygotowano też funkcję dokonującą detekcji na podstawie reguł, jednak nie analizującą ruchu wygenerowanego za pomocą *scapy*, a na podstawie wczytanego pliku *pcap*. Istotny fragment tej funkcji został przedstawiona poniżej:

```python
def exec_detect_rules():
    streamer = NFStreamer(source=r"malicious_traffic_lab2.pcap",
    statistical_analysis=True)
    df = load_df(streamer)
    alerts = detect_anomalies_with_rules(df)
```

2.3.4. PoC

W ramach Proof of Concept przeprowadzono analizę ruchu opisanego w Sekcja 2.3.1, bazując na regułach detekcyjnych opisanych w Sekcja 2.3.2. Zgodnie z przewidywaniami wygenerowane zostały dwa alerty (Rys. 1).

```
Generowanie ruchu sieciowego...
Ruch sieciowy wygenerowany: 11 pakietów.
Rozpoczęcie analizy ruchu...

WYKRYTE ALERTY:
ALERT!

Reguła: Suspicious Port Usage
Opis: Wykrywa użycie podejrzanych portów docelowych.
Wykryte anomalie:
  source_ip destination_ip  source_port  destination_port protocol  total_bytes  total_packets
0  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
1  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
2  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
3  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
4  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
5  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
6  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
7  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
8  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
9  10.0.0.1    192.168.0.1        12345              3389      UDP         1028              1
ALERT!

Reguła: Communication with Blacklisted IPs
Opis: Wykrywa komunikację z czarną listą IP.
Wykryte anomalie:
   source_ip   destination_ip  source_port  destination_port protocol  total_bytes  total_packets
10  10.0.0.3  122.226.120.191        55555                53      TCP           87              1
```

Rys. 1: Alerty wywołane przez ruch wygenerowany za pomocą *scapy*

Przeprowadzono również detekcję z ruchu, zarejestrowanego w pliku *malicious_traffic_lab2.pcap*, pobranym z https://mcfp.felk.cvut.cz/publicDatasets/. Również wywołane zostały dwa alerty, inne niż w poprzednim przykładzie (Rys. 2).

```
ALERTY WYKRYTE:

Reguła: Large Data Exfiltration
Opis: Wykrywa przepływy z dużą ilością przesłanych danych.
Wykryte anomalie:
     source_ip   destination_ip  source_port  destination_port  protocol  src_to_dst_bytes  dst_to_src_bytes  total_bytes  src_to_dst_packets  dst_to_src_packets  total_packets            timestamp
50   10.0.2.119  122.226.120.191        49159                80         6            108634           8243126      8351760                2010                6751           8761 1970-01-01 00:35:09.831
145  10.0.2.119    61.155.165.25        49161                80         6            128199           9334621      9462820                2371                6336           8707 1970-01-01 00:49:07.365

Reguła: Communication with Blacklisted IPs
Opis: Wykrywa komunikację z czarną listą IP.
Wykryte anomalie:
     source_ip   destination_ip  source_port  destination_port  protocol  src_to_dst_bytes  dst_to_src_bytes  total_bytes  src_to_dst_packets  dst_to_src_packets  total_packets            timestamp
50   10.0.2.119  122.226.120.191        49159                80         6            108634           8243126      8351760                2010                6751           8761 1970-01-01 00:35:09.831
145  10.0.2.119    61.155.165.25        49161                80         6            128199           9334621      9462820                2371                6336           8707 1970-01-01 00:49:07.365
```

Rys. 2: Alerty wywołane przez ruch zarejestrowany w pliku *pcap*

2.4. Wizualizacja (V.1)

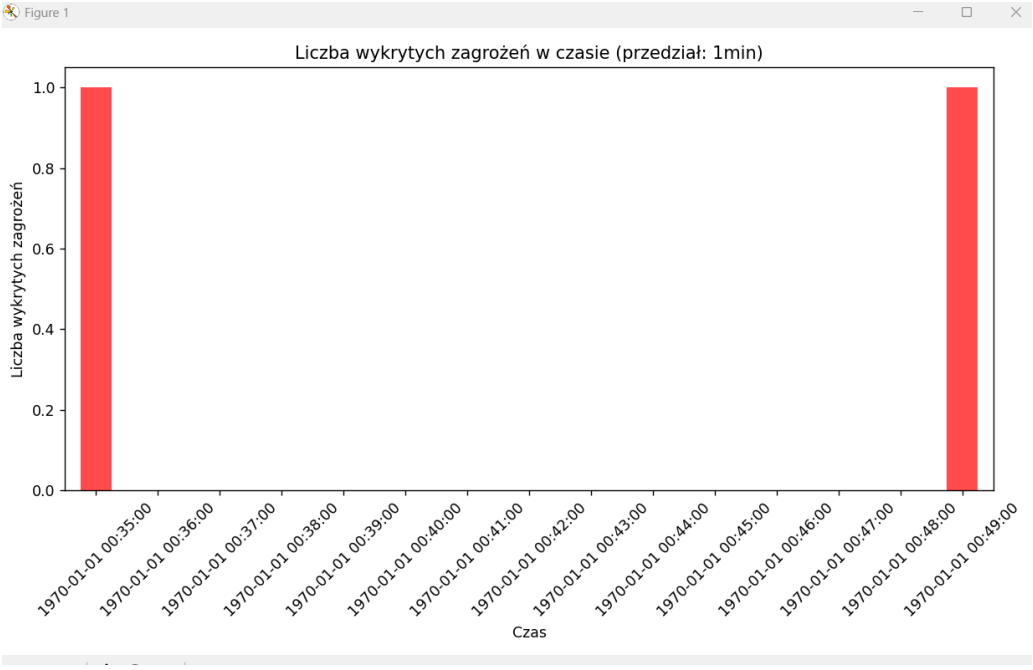
Zaimplementowano funkcjonalność wizualizacji liczby zagrożeń wykrytych za pomocą modułu DaaC z punktu D.2 (opisanego w Sekcja 3.1) w czasie za pomocą wykresu słupkowego. Zmodyfikowano w tym celu funkcję *detect_sigama_rule* opisaną w Sekcja 3.1, dodając do niej następujący fragment, odpowiedzialny za konwersję timestamp'u ramki danych na DateTime:

```python
detected_flows['timestamp'] = pd.to_datetime(detected_flows['timestamp'])
return detected_flows
```

Następnie przygotowano funkcję *visualize_threats_over_time* odpowiedzialną za grupowanie liczby zagrożeń w przedziałach czasowych oraz sporządzanie wykresu:

```python
def visualize_threats_over_time(detected_flows, interval='1min'):
    if detected_flows.empty:
        click.echo("Brak wykrytych zagrożeń do wizualizacji.")
        return

    # Grupowanie liczby zagrożeń w przedziałach czasowych
    threats_over_time =
    detected_flows.set_index('timestamp').resample(interval).size()

    # Tworzenie wykresu
    plt.figure(figsize=(10, 6))
    threats_over_time.plot(kind='bar', color='red', alpha=0.7)
    plt.xlabel("Czas")
    plt.ylabel("Liczba wykrytych zagrożeń")
    plt.title(f"Liczba wykrytych zagrożeń w czasie (przedział: {interval})")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
```

Sporządzono wykres liczby zagrożeń wykrytych w ramach PoC w sekcji D.2 (Sekcja 3.1) w czasie i przedstawiono go poniżej (Rys. 3).



Rys. 3: Wykres liczby zagrożeń wykrytych w ramach detekcji w punkcie D.2 w czasie

3. Projekt 2: Funkcjonalności zaawansowane

3.1. Detection as a Code (D.2)

Po analizie pliku, który był wykorzystywany również w trakcie laboratorium. Doszliśmy do poniższych wniosków: Szczególnie zauważalne jest 10.0.2.119 przesyłające: 9,462,820 bajtów do

61.155.165.25, 8,351,760 bajtów do 122.226.120.191 Obie te komunikacje wyróżniają się znaczną ilością przesyłanych danych. To może wskazywać na exfiltrację danych (np. przesyłanie skradzionych informacji).

Oba adresy IP pochodzą z chin 61.155.165.25 (61.155.160.0/21) AS 140292 (CHINATELECOM Jiangsu province Suzhou 5G network) 122.226.120.191 (122.226.120.0/24) AS 136190 (JINHUA, ZHEJIANG Province, P.R.China.)

Adresy takie jak fe80::45f6:82cd:198c:6a1a i ff02::1:2 wskazują na komunikację lokalną IPv6, co samo w sobie nie jest podejrzane, a ich wielokrotne występowanie z identycznymi bajtami/pakietami sugeruje ruch rozgłoszeniowy.

W związku z tym stworzyliśmy poniższą zasadę sigma w formacie yml.

```yml
title: Large Data Transfer on Port 80
id: 12345
logsource:
  product: network
  service: netflow  # Możesz dostosować to do odpowiedniego źródła danych
detection:
  selection:
    destination_port: 80
    bytes_sent|gte: 1000000  # Poprawiona składnia porównania
    dst_ip:
      - "122.226.120.191"  # Known suspicious IP
      - "61.155.165.25"    # Known suspicious IP
  condition: selection
fields:
  - source_ip
  - destination_ip
  - total_bytes
  - total_packets
```

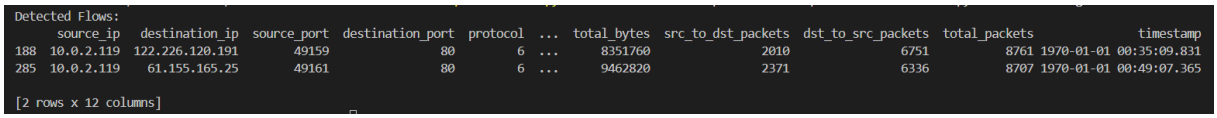
- destination_port: 80: Wskazuje ruch HTTP, często wykorzystywany do nieautoryzowanych transferów danych.
- bytes_sent|gte: 1000000: (Greater Than or Equal To) Określa próg ilości przesłanych danych (1 MB). Wartość ta może być regulowana w zależności od środowiska.
- dst_ip: Lista podejrzanych adresów IP. Wskazuje na wcześniejszą wiedzę o złośliwej aktywności z tych adresów.

Użyliśmy biblioteki yaml do parsowania zasady sigma w pythonie. Wykorzystując wartości sparsowane z pliku yml oraz porównanie ich z dataframem otrzymaliśmy prostu detection as a code.

```python
def detect_sigma_rule(df):
    with open(r'sigma_rule.yml', 'r') as file:
        sigma_rule = yaml.safe_load(file)


    selection = sigma_rule['detection']['selection']
    dst_ips = selection['dst_ip']
    bytes_sent_gte = selection['bytes_sent|gte']
    destination_port = selection['destination_port']
```

```
    detected_flows = df[
        (df['destination_port'] == destination_port) &
        (df['total_bytes'] >= bytes_sent_gte) &
        (df['destination_ip'].isin(dst_ips))
    ]

    print("Detected Flows:")
    print(detected_flows)
```

Wynik funkcji:



3.2. Machine Learning (ML.1)

Głównym zadaniem podpunktów związanych z machine learning'iem było stworzenie modelu, który klasyfikowałby flow na podstawie jego różnych cech, takich jak: port źródłowy ruchu, port docelowy, użyty protokół komunikacyjny, czasu trwania, ilości pakietów. Zadaniem modelu jest określenie czy dany ruch jest związany ze złośliwym działaniem, czy jest to normalny ruch.

3.2.1. Ekstrakcja odpowiednich danych z flow

Aby przygotować plik `PCAP` do analizy używamy NFStream'a. Następnie wyciągamy z niego parametry, które potrzebne będą nam do analizy ruchu. W celu wytrenowania modelu oznaczamy ruch też etykietą, gdzie 0 oznacza ruch normalny, natomiast 1 oznacza ruch złośliwy. Wykorzystujemy do tego funkcję `extract_features_with_nfstream`, która przyjmuje dwa argumenty:

- `pcap_file` - plik pcap z ruchem
- `label` - oznaczenie czy podawany ruch jest złośliwy - 1, czy normalny - 0

```python
def extract_features_with_nfstream(pcap_file, label):
    streamer = NFStreamer(source=pcap_file).to_pandas()

    # Wybieramy istotne cechy
    features = streamer[[
        'src_port', 'dst_port', 'protocol', 'bidirectional_duration_ms',
        'bidirectional_packets', 'bidirectional_bytes'
    ]].copy()

    features['label'] = label  # 0: normalny, 1: anomalny
    return features
```

Następnie za pomocą funkcji `load_data` klasyfikujemy ruch na podstawie, którego będziemy trenować model.

```python
def load_data():
    # Ekstrakcja cech dla normalnego i złośliwego ruchu
    normal_data = extract_features_with_nfstream("normal_traffic_lab2.pcap",
    label=0)  # 0: normalny ruch
    malicious_data = extract_features_with_nfstream("mal_traffic_lab2.pcap",
    label=1)  # 1: złośliwy ruch
```

```
    return normal_data, malicious_data
```

3.2.2. Trenowanie modelu

Następnie za pomocą uzyskanych danych trenujemy model, który opiera się na algorytmie losowego lasu decyzyjnego z biblioteki `scikit-learn`. Uzyskane dane dzielimy na dane trenujące i na dane sprawdzające poprawność działania algorytmu w stosunku 7 : 3. Następnie na podstawie danych trenujących trenujemy model. Na końcu przedstawiamy raport klasyfikacji, który opisuje z jaką precyzją nasz model jest w stanie klasyfikować ruch. Raport zawiera:

- precyzję modelu, która określa jaka część przewidzianych jako pozytywne klasyfikacji jest faktycznie pozytywna
- czułość modelu, która określa jaka część rzeczywistych pozytywnych przypadków została poprawnie sklasyfikowana
- F1-socre - średnia harmoniczna precyzji i czułości
- support - liczba próbek w każdej klasie
- macro average - średnia arytmetyczna dla każdej klasy
- weighted average - średnia ważona uwzględniająca rozkład klas

Zwracana jest także dokładność modelu. Na podstawie danego modelu tworzymy jego macierz konfuzji z wykorzystaniem biblioteki `scikit-learn`.

```python
def train_model(normal_data, malicious_data):
    data = pd.concat([normal_data, malicious_data], ignore_index=True)
    print("Rozmiar zbioru danych:", data.shape)
    print(data.head())
    X = data.drop('label', axis=1)
    y = data['label']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)
    clf = RandomForestClassifier(n_estimators=100, random_state=42)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    y_pred_proba = clf.predict_proba(X_test)[:, 1]
    # Raport wyników
    print("Raport klasyfikacji:\n", classification_report(y_test, y_pred))
    accuracy = clf.score(X_test, y_test)
    print("Dokładność modelu: {:.2f}%".format(accuracy * 100))
    cm = confusion_matrix(y_test, y_pred)
    return y_pred, y_test, y_pred_proba, X_train, y_train, X_test, cm
```

```
Rozmiar zbioru danych: (184394, 7)
   src_port  dst_port  protocol  bidirectional_duration_ms  bidirectional_packets  bidirectional_bytes  label
0         0         0        58                       8015                      3                  210      0
1     52382        53        17                         33                      2                  258      0
2        67        68        17                       8280                      4                 2360      0
3     56750        53        17                         29                      2                  386      0
4     64807        53        17                         33                      2                  371      0
Raport klasyfikacji:
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      1403
           1       1.00      1.00      1.00     53916

    accuracy                           1.00     55319
   macro avg       0.99      0.98      0.99     55319
weighted avg       1.00      1.00      1.00     55319

Dokładność modelu: 99.89%
```

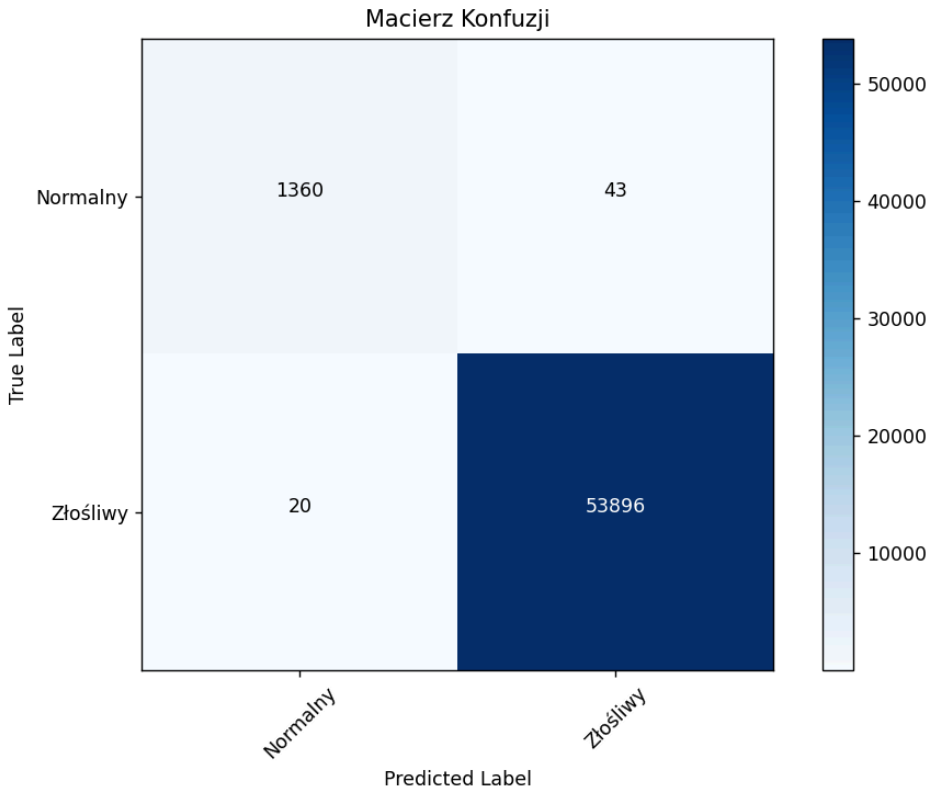
Rys. 4: Rozmiar danych podawanych do trenowania i oceniania modelu oraz raport klasyfikacyjny modelu.

3.3. Machine Learning (ML.2)

3.3.1. Wizualizacja miar jakości

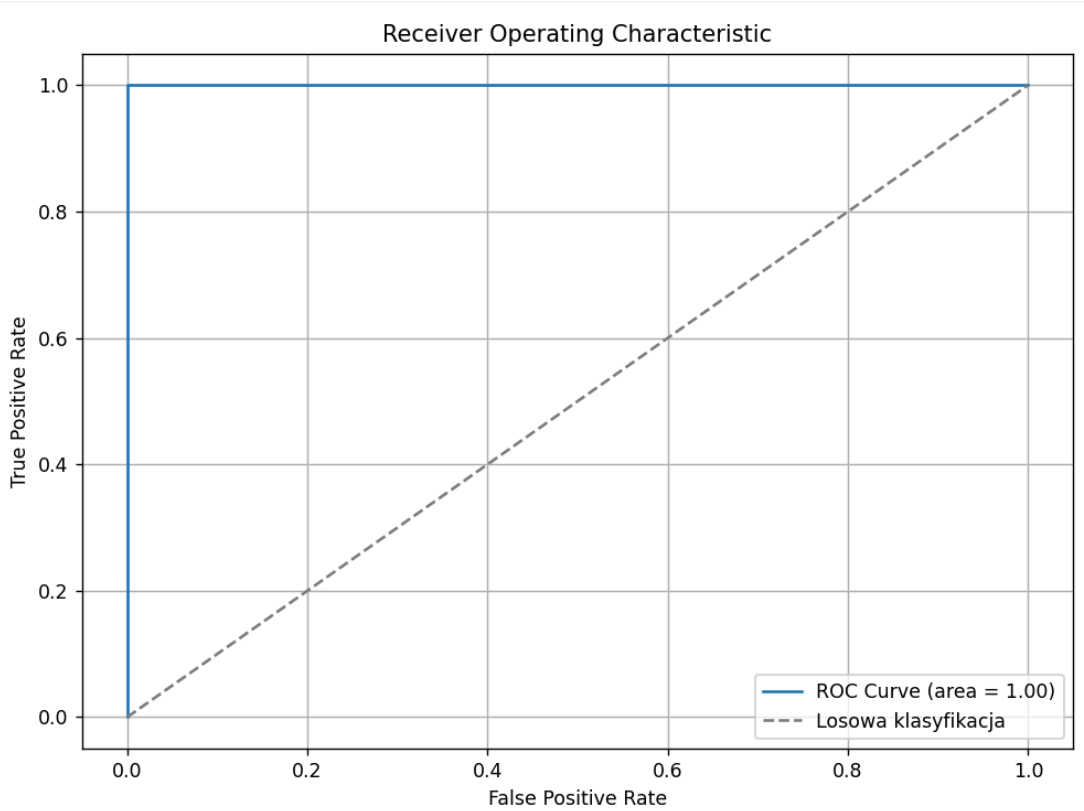
Aby zwizualizować miary jakości korzystamy z macierzy konfuzji modelu oraz krzywej ROC.

```python
ef plot_confusion_matrix(cm, class_names):
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Macierz Konfuzji')
    plt.colorbar()

    # Dodanie osi z nazwami klas
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)

    # Dodanie liczb w macierzy
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > cm.max() / 2. else "black")

    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.show()
```



Rys. 5: Macierz konfuzji modelu

```python
def curve_ROC(y_test, y_pred_proba):
    fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
    roc_auc = auc(fpr, tpr)
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f'ROC Curve (area = {roc_auc:.2f})'.format(roc_auc))
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--', label='Losowa
    klasyfikacja')


    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```



Rys. 6: Krzywa ROC

3.3.2. Tuning hiperparametów

Aby znaleźć lepsze parametry dla naszego modelu korzystamy z narzędzia `GridSearchCV` na podstawie parametru F1. Funkcja pochodzi z biblioteki `scikit-learn`.

```python
def search_params(X_train, y_train):
    param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]
    }
    grid_search = GridSearchCV(RandomForestClassifier(random_state=42), param_grid,
    cv=3, scoring='f1')
    grid_search.fit(X_train, y_train)
    print("Najlepsze parametry:", grid_search.best_params_)
    best_params = grid_search['best_params_']
    return best_params['max_depth'], best_params['min_samples_split'],
    best_params['n_estimators']
```

Na podstawie parametrów znalezionych przez funkcję `search_params` używamy funkcji `better_model` aby zbudować lepszy model.

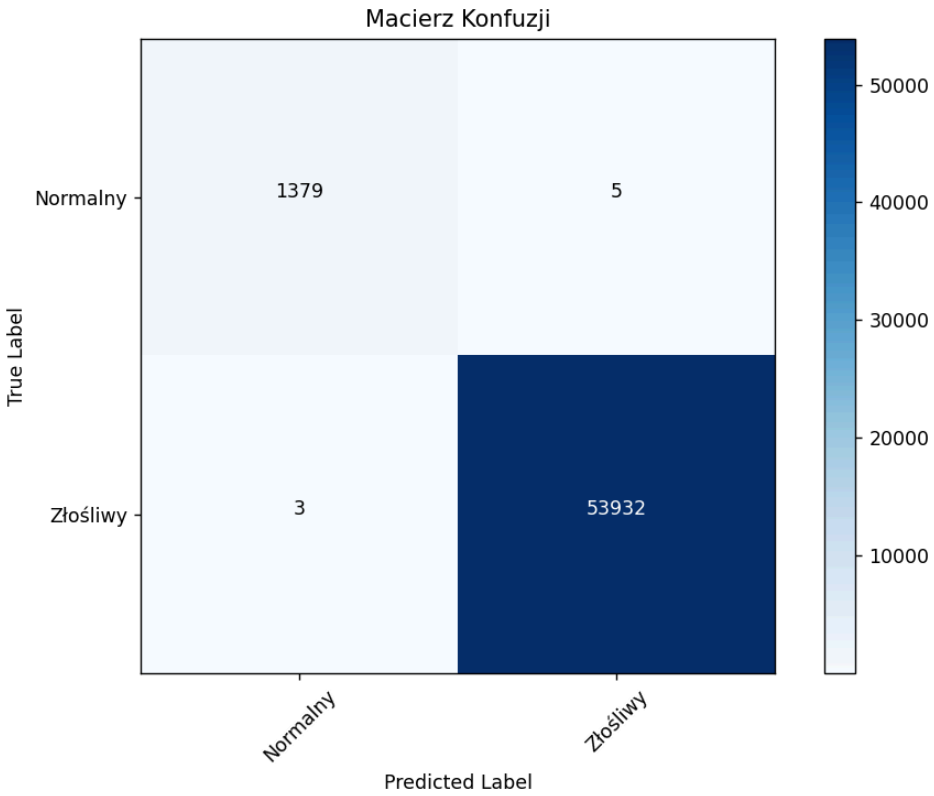
```python
def better_model(max_depth, min_samples_split, n_estimators, X_train,
y_train):
    # Tworzenie modelu z najlepszymi parametrami
    best_rf = RandomForestClassifier(
        n_estimators=n_estimators,
```

```
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        random_state=42
    )

    # Trenowanie modelu na danych treningowych
    best_rf.fit(X_train, y_train)
    return best_rf
```

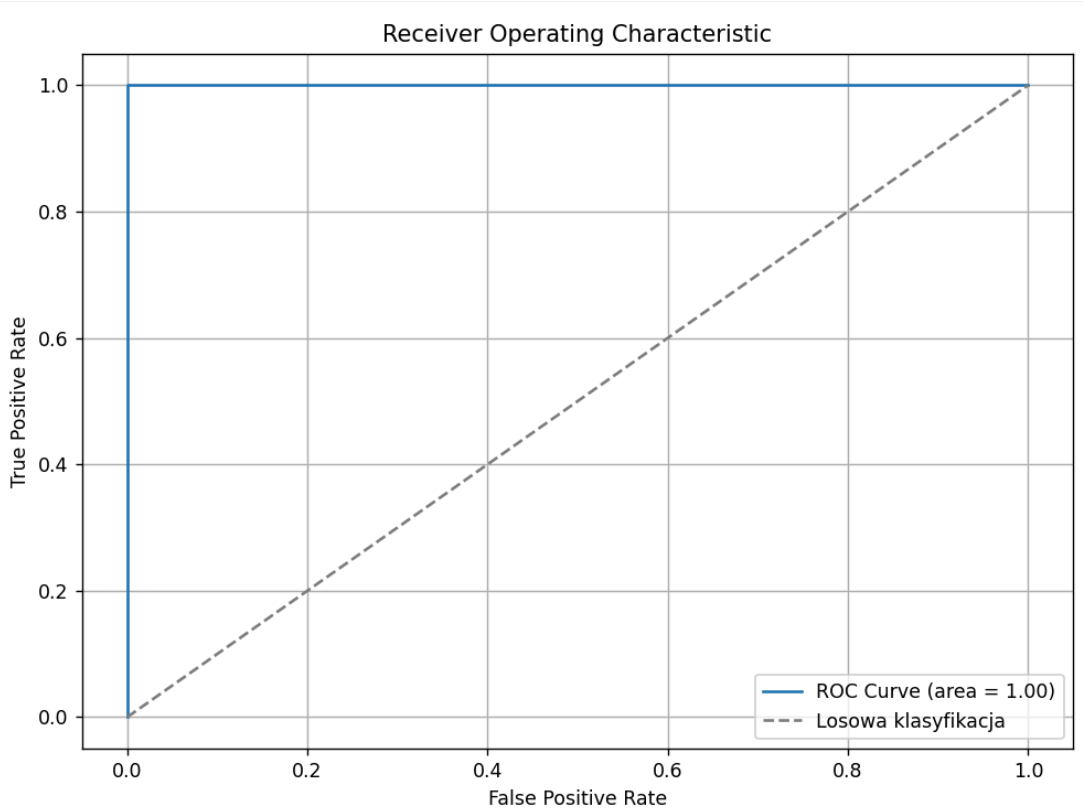
Dalej używamy funkcję `accuracy_grade_matrix_ROC` aby zwizualizować dokładność nowego modelu za pomocą macierzy konfuzji i krzywej ROC.

```
Najlepsze parametry: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 200}
Dokładność modelu: 99.99%
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1384
           1       1.00      1.00      1.00     53935

    accuracy                           1.00     55319
   macro avg       1.00      1.00      1.00     55319
weighted avg       1.00      1.00      1.00     55319
```

Rys. 7: Najlepsze parametry i raport klasyfikacji nowego modelu



Rys. 8: Macierz kofnuzji nowego modelu



Rys. 9: Krzywa ROC nowego modelu

3.4. Enrichment (E.1)

3.4.1. Pobranie danych geograficznych

Funkcja *get_ip_info(ip_address)* umożliwia pozyskanie szczegółowych informacji o danym adresie IP za pomocą zewnętrznego API: http://ip-api.com/. Zwraca dane dotyczące geolokalizacji oraz dostawcy usług internetowych:

- country: kraj, z którego pochodzi adres IP;
- regionName: nazwa regionu pochodzenia adresu;
- city: miasto pochodzenia adresu;
- lat: szerokość geograficzna;
- lon: długość geograficzna;
- isp: dostawca usług internetowych.

Funkcja wykorzystuje bibliotekę *requests* do wykonania żądania GET. Odpowiedź jest przetwarzana, a dane są ekstraktowane w formie słownika:

```python
def get_ip_info(ip_address):
    url = f"http://ip-api.com/json/{ip_address}?
    fields=country,regionName,city,lat,lon,isp"
    try:
        response = requests.get(url)
        if response.status_code == 200:
            data = response.json()
            return {
                "IP": ip_address,
                "Country": data.get("country"),
```

```
                "Region": data.get("regionName"),
                "City": data.get("city"),
                "Latitude": data.get("lat"),
                "Longitude": data.get("lon"),
                "ISP": data.get("isp"),
            }
        else:
            return {"IP": ip_address, "Error": f"HTTP {response.status_code}"}
    except Exception as e:
        return {"IP": ip_address, "Error": str(e)}
```

3.4.2. Zawarcie danych geograficznych w raporcie

Do funkcji *detect_sigma_rule* opisanej w Sekcja 3.1 dodano przedstawiony poniżej fragment kodu, odpowiedzialny za pobranie danych geograficznych o każdym adresie IP wykrytym w ramach detekcji oraz przedstawienie tych danych w raporcie:

```python
    for ip in detected_flows["destination_ip"].unique():
        info = get_ip_info(ip)
        print(f"\nInformacje o {ip}:")
        for key, value in info.items():
            print(f"{key}: {value}")
```

Rozbudowany przez funkcjonalność pobierania danych geograficznych o adresach IP raport z detekcji przedstawiono poniżej (Rys. 10).

```
Detected Flows:
     source_ip   destination_ip  source_port  destination_port  protocol  src_to_dst_bytes  dst_to_src_bytes  total_bytes  src_to_dst_packets  dst_to_src_packets  total_packets            timestamp
42   10.0.2.119  122.226.120.191        49159                80         6            108634           8243126      8351760                2010                6751           8761 1970-01-01 00:35:09.831
156  10.0.2.119   61.155.165.25        49161                80         6            128199           9334621      9462820                2371                6336           8707 1970-01-01 00:49:07.365

Informacje o 122.226.120.191:
IP: 122.226.120.191
Country: China
Region: Zhejiang
City: Jinhua
Latitude: 29.0792
Longitude: 119.647
ISP: China Telecom

Informacje o 61.155.165.25:
IP: 61.155.165.25
Country: China
Region: Jiangsu
City: Nanjing
Latitude: 32.0607
Longitude: 118.763
ISP: China Telecom
```

Rys. 10: Raport z detekcji rozbudowany o dane geograficzne związane z adresami IP

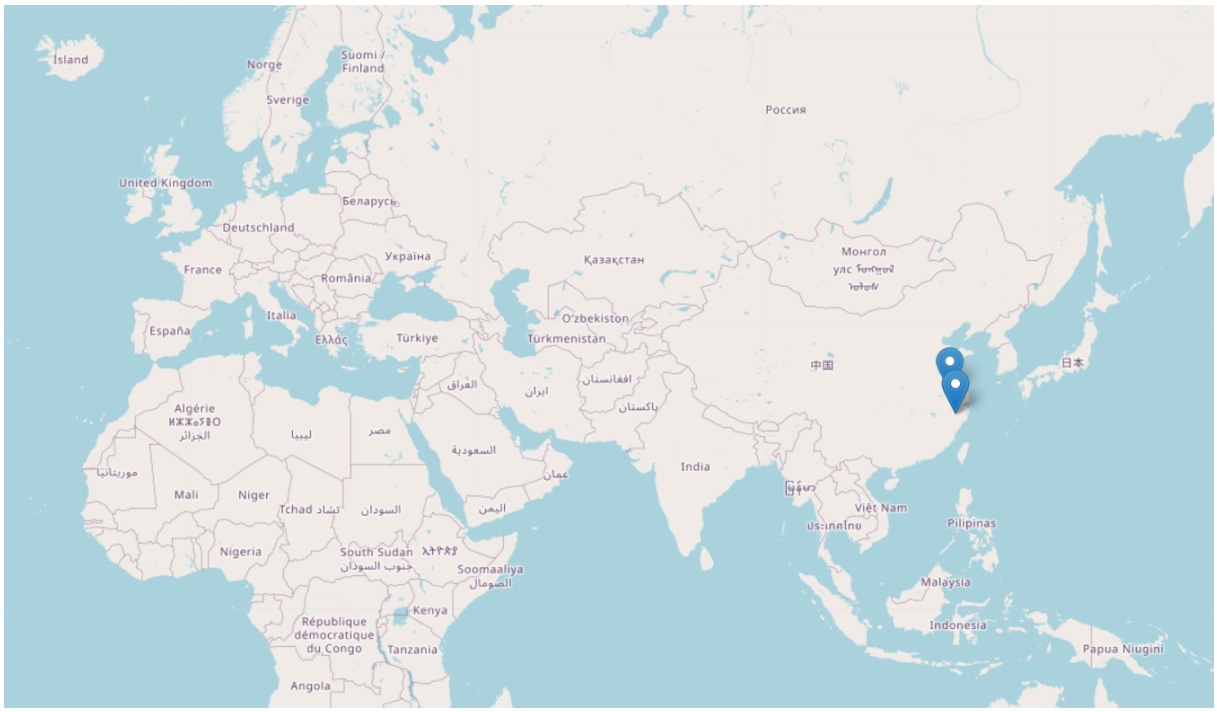
3.5. Wizualizacja (V.2)

Wykorzystując api do pobrania danych geograficznych o adresach ip oraz bibliotekę folium do generowanie mapy wykonaliśmy prostą wizualizację

```python
def get_ip_coordinates(ip_address):
    url = f"http://ip-api.com/json/{ip_address}?fields=lat,lon"
    try:
        response = requests.get(url)
        if response.status_code == 200:
            data = response.json()
            return {
                "Latitude": float(data.get("lat", 0.0)),  # Ensure lat is a float
                "Longitude": float(data.get("lon", 0.0)),  # Ensure lon is a float
            }
```

```
        else:
            return {"Error": f"HTTP {response.status_code}"}
    except Exception as e:
        return {"Error": str(e)}

def generate_map(ip_list):

    ip_coordinates = []
    for ip in ip_list:
        coords = get_ip_coordinates(ip)
        if "Error" not in coords:
            ip_coordinates.append({
                "ip": ip,
                "latitude": coords["Latitude"],
                "longitude": coords["Longitude"]
            })
        else:
            print(f"Error for IP {ip}: {coords['Error']}")


    map = folium.Map(location=[20, 0], zoom_start=2)

    for loc in ip_coordinates:
        folium.Marker(
            location=[loc["latitude"], loc["longitude"]],
            popup=f"IP: {loc['ip']}"
        ).add_to(map)


    map.save("ip_locations_map.html")
    print("The map has been saved as ip_locations_map.html.")
```

A oto efekt.



4. Implementacja interfejsu CLI

Za pomocą biblioteki *click* i odpowiednich dekoratorów, zaimplementowano interfejs CLI, umożliwiający użytkownikowi wygodne korzystanie z poszczególnych funkcjonalności systemu. Poniżej wymieniono przygotowane opcje udostępniane przez interfejs terminala:

- exec-save - zapisuje dane z formatu pcap do pliku, domyślna nazwa pliku output.txt, możliwość zmiany nazwy z poziomu cli, może być przydatne do efektywnego przeszukiwania pliku tekstowe na przykład za pomocą polecenia | grep
- exec-summary - wypisuje podsumowanie pliku pcap
- exec-visualize - rysuje dwa wykresy oparte na podsumowaniu, jeden z nich odnosi się do pakietów przesyłanych a drugi do bajtów przesyłanych
- exec-detect-sigma - wykonuje parsowanie pliku yml do pythona oraz sprawdzanie zasady sigma z plikiem pcap oraz alertowanie
- *analyze-traffic* - przeprowadzenie detekcji na ruchu generowanym przez *scapy*, na podstawie reguł (D.1)
- *detect-rules* - przeprowadzenie detekcji na ruchu zarejestrowanym w pliku *pcap*, na podstawie reguł (D.1)
- *visualize-threats* - sporządzenie wykresu liczby wykrytych zagrożeń w czasie (V.1)
- *detect-sigma-ip* - wykonanie detekcji sigma z uwzględnieniem danych geograficznych o adresach IP w raporcie (E.1)
- *exec_ml* - utworzenie modelu do detekcji złośliwego ruchu sieciowego (ML.1)
- *exec_tuning* - ulepszenie modelu po przez redukcje fałszywych pozytywów, wizualziacja miar jakości (ML.2)
- exec-map - wykorzystując bibliotekę folium generuje mapę w pliku html

5. Podsumowanie

W ramach projektu i laboratorium, zgodnie z założeniami, stworzono prototypowy system analizy sieciowej w konwencji Proof of Concept (PoC), który realizuje kluczowe funkcje analizy ruchu sieciowego na poziomie przepływów. System wykorzystuje zasady Detection as a Code, stosując metody zarówno detekcji regułowej jak i Machine Learning.