



Kryminalistyka cyfrowa

Laboratorium 2 i Projekt 2

Stanisław Ciszkiewicz
Jakub Kusznier

Spis treści

1. Wstęp	3
1.1. Cel	3
1.2. Wymagania	3
2. Opis rozwiązania	5
2.1. Budowa systemu	5
2.2. Interfejs użytkownika	5
2.2.1. Analiza plików pcap	6
2.2.2. Analiza dynamiczna na interfejsie	6
2.2.3. Konfiguracja modelu ML	6
2.2.4. Mapa	6
3. Wymagania laboratoryjne	7
3.1. Analiza flow A1	7
3.1.1. Cel	7
3.1.2. Realizacja	7
3.2. Analiza flow A2	7
3.2.1. Cel	7
3.2.2. Realizacja	8
3.3. Detection as a Code D1	9
3.3.1. Cel	9
3.3.2. Realizacja	9
3.3.2.1. Detect_long_connection	9
3.3.2.2. Detect_dos_attack	10
3.3.2.3. Detect_blacklisted_IP	11
3.4. Wizualizacja V1	12
3.4.1. Cel	12
3.4.2. Realizacja	13
4. Wymagania projektowe	13
4.1. Detection as a Code (D2)	13
4.1.1. Cel	13
4.1.2. Realizacja	13
4.2. Machine Learning (ML1)	14
4.2.1. Cel	14
4.2.2. Realizacja	14
4.3. Machine Learning (ML2)	16
4.3.1. Cel	16
4.3.2. Realizacja	16
4.4. Machine Learning (ML3)	17
4.4.1. Cel:	17
4.4.2. Realizacja	17
4.5. Enrichment (E1) i Wizualizacja (V2)	19
4.5.1. Cel	19
4.5.2. Realizacja	19

1. Wstęp

1.1. Cel

Celem laboratorium 2 i projektu 2 z przedmiotu KRYCY było zbudowanie systemu analizy sieciowej. Naszym zadaniem było stworzenie rozwiązania w języku python do detekcji zagrożeń na bazie reguł (Detection as a Code) oraz zintegrować je również z rozwiązaniami uczenia maszynowego. Narzędzie powinno zwracać raport z wykonania zadań zleconych przez użytkownika w formacie pozwalającym na dalszą dokładniejszą analizę podejrzanych próbek.

1.2. Wymagania

Sprawozdanie z wykonanego zadania chcielibyśmy przedstawić prezentując kolejne funkcjonalności wymagane w poleceniu projektowo/laboratoryjnym.

Wymagania laboratoryjne:

ID	Kategoria	Opis wymagania	Typ	Proponowany sposób udowodnienia / dodatkowe komentarze
A.1	Analiza flow	Wczytywanie plików PCAP przy użyciu NFStream.	Must-have	Implementacja/inne wizualizacje w raporcie z narzędzia.
A.2	Analiza flow	Dla wczytanych przepływów wyświetlanie podsumowania statystyk flow, takich jak podsumowanie ilości przesłanych pakietów pomiędzy danymi hostami.	Must-have	Musi znajdować się w raporcie generowanym przez narzędzie.
D.1	Detection as a Code	Implementacja reguł detekcyjnych w Pythonie, np. jako funkcje w pliku detection_rules.py.	Must-have	Napisanie przykładowej reguły i symulacja przy wykorzystaniu tcpreplay lub scapy wywołania alertu.
V.1	Wizualizacja	Wykres liczby wykrytych zagrożeń w czasie (np. wykres słupkowy).	Must-have	Alternatywa: Zamiast wykresów – podsumowanie w tabeli tekstowej lub inna wizualizacja w raporcie z narzędzia.

Wymagania projektowe:

ID	Kategoria	Opis wymagania	Typ	Proponowany sposób udowodnienia / dodatkowe komentarze
D.2	Detection as a Code	Wczytywanie reguł w formacie Sigma (np. z użyciem PySigma).	Must-have	Demonstracja wczytywania reguły Sigma, przetwarzania jej w Pythonie i wywołania detekcji zgodnej z tą regułą.
ML.1	Machine Learning	Klasyfikacja flow na podstawie cech, takich jak czas trwania, liczba pakietów, protokół (np. z użyciem scikit-learn).	Must-have	Raport generowany przez narzędzie zawiera output z modelu, np. w postaci pewności zwróconej przez model lub wizualizacji działania modelu.
ML.2	Machine Learning	Redukcja liczby fałszywych pozytywów (FPR) za pomocą oceny jakości modelu i tuningu hiperparametrów.	Must-have	Liczenie metryk takich jak FPR, TPR lub wizualizacja macierzy konfuzji dla testowanego przypadku.
ML.3	Machine Learning	Narzędzie powinno pozwalać na trenowanie zawartego w nim modelu ML za pomocą nowych danych dostarczonych przez użytkownika (np. przez CLI).	Nice-to-have	Demonstracja możliwości wgrania nowych danych w formacie CSV przez użytkownika, ich przetwarzania i ponownego trenowania modelu.
E.1	Enrichment	Pobieranie podstawowych informacji o IP/domenach, np. z geopy lub innych źródeł Threat Intelligence przy użyciu API.	Must-have	Enrichment widoczny w raporcie generowanym przez narzędzie.
V.2	Wizualizacja	Mapa geograficzna przedstawiająca lokalizację adresów IP wykrytych jako podejrzane.	Nice-to-have	Wizualizacja lokalizacji IP na mapie, np. przy użyciu bibliotek folium lub plotly.

2. Opis rozwiązania

2.1. Budowa systemu

W ramach implementacji postawiliśmy na proste w obsłudze (zrozumiałe dla każdego użytkownika) rozwiązanie terminalowe. Użytkownik uruchamia system poprzez wywołanie pythonowego skryptu nids_app.py odpowiedzialnego za interfejs terminalowy z instrukcjami dla użytkownika oraz wywołanie pozostałych komponentów.

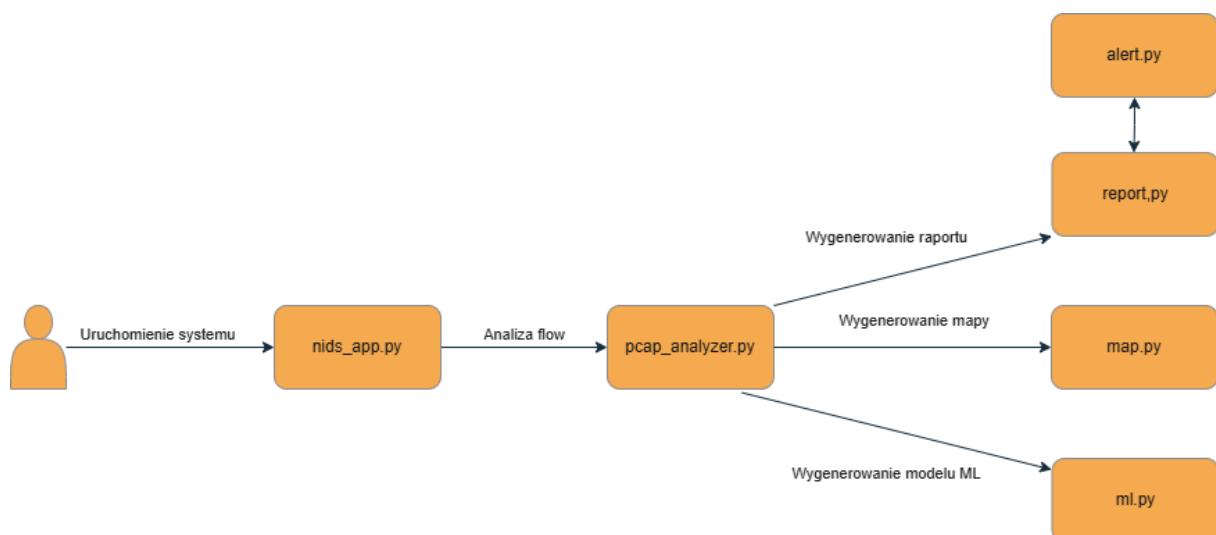
Mózgiem całego systemu jest klasa Pcap_Analyzer, która jest odpowiedzialna za analizę flow (zawiera m.in. reguły detekcyjne), odpowiada za przechwycenie danych o flow.

Po przeprowadzeniu odpowiednich operacji przez klasę PcapAnalyzer tworzony jest obiekt z klasy Report - odpowiedzialny za tworzenie raportu w formacie json.

Zaimplementowana została również klasa Alert, która odpowiedzialna jest za tworzenie struktury alertu przekazywanego do obiektu z klasy Report.

Zaimplementowaliśmy również klasę ML (Machine Learning) odpowiedzialną za budowę i obsługę modelu uczenia maszynowego używanego w naszym rozwiążaniu do klasyfikacji ruchu.

Jako rozwinięcie zaproponowaliśmy również klasę Map, której zadaniem jest tworzenie mapy z lokalizacjami podejrzanych adresów IP.



Rysunek 1: Budowa zbudowanego systemu

2.2. Interfejs użytkownika

Postawiliśmy na prosty, intuicyjny interfejs, dzięki któremu użytkownik naszego systemu może zlecać konkretne działania. Po uruchomieniu pliku nids_app.py w terminalu ukazuje się menu z następującymi opcjami:

```
Witamy w systemie analizy sieciowej. Wybierz z listy co chciałbyś zrobić
1. Analiza plików pcap.
2. Analiza dynamiczna na interfejsach.
3. Konfiguracja modelu ML.
4. Mapa lokalizacji podejrzanych adresów IP.
5. Wyjście
Wybieram opcję: ■
```

Rysunek 2: Menu główne

2.2.1. Analiza plików pcap

Opcja odpowiedzialna za analizę plików pcap, po jej wyborze następuje wybór spośród trzech możliwości:

- raport z danymi wszystkich flow z pcap
- raport z danymi podejrzanych flow znajdujących się w pliku pcap (na bazie detection rules)
- klasyfikacja podejrzanych flow na bazie modelu ML

```
Wybierz z listy, jakie zadanie chciałbyś zlecić systemowi:  
1. Otrzymanie raportu z danymi wszystkich flow z pcap.  
2. Otrzymanie raportu z podejrzanymi flow + alerty bezpieczeństwa (detection rules).  
3. Otrzymanie raportu z podejrzanymi flow przy użyciu modelu ML.  
4. Cofnij do menu głównego.  
Wybieram opcję: [ ]
```

Rysunek 3: Menu - analiza plików pcap

Po wyborze któregoś z opcji użytkownik proszony jest o podanie nazwy badanego pliku i umieszczenie go w katalogu resources projektu.

2.2.2. Analiza dynamiczna na interfejsie

Druga opcja dotyczy analizy ruchu live na interfejsie sieciowym. Opcje są analogiczne co do przypadku dla plików, natomiast po wyborze któregoś z nich użytkownik proszony jest o wpisanie nazwy interfejsu sieciowego, na którym chce przeprowadzić analizę.

```
Wybierz z listy, jakie zadanie chciałbyś zlecić systemowi:  
1. Otrzymanie raportu z analizy pakietów live na interfejsie.  
2. Otrzymanie raportu podejrzanych pakietów live na interfejsie (detection rules).  
3. Otrzymanie raportu podejrzanych pakietów live na interfejsie (ML).  
4. Cofnij do menu głównego.  
Wybieram opcję: [ ]
```

Rysunek 4: Menu - analiza dynamiczna na interfejsach

2.2.3. Konfiguracja modelu ML

Wybór tej opcji pozwala na przejście do panelu odpowiedzialnego za konfigurację modelu ML służącego do analizy plików pcap i ruchu live. Po wyborze tej opcji możemy wykonać następujące działania:

- stworzenie nowego modelu (wprowadzenie przez użytkownika własnych zbiorów danych treningowych), następnie używanie tego modelu do klasyfikacji ruchu,
- dotrenowanie modelu nowymi danymi - użytkownik również wprowadza nowe pliki (normal i malicious), natomiast w tym przypadku model jest dotrenowywany nowymi danymi, tj. nowe dane łączone są z danymi pierwotnymi i na nowo przeprowadzany jest proces uczenia modelu,
- przeanalizowanie jak dobrze obecny model radzi sobie z danymi - użytkownik wprowadza dwa pliki (normal i malicious), wynik ukazuje mu jak dobry jest model dla konkretnego przypadku (wskaźnik jakościowy)

```
Wybierz z listy, jakie zadanie chciałbyś zlecić systemowi:  
1. Wygenerowanie własnego modelu drzewa decyzyjnego do klasyfikacji flow.  
2. Dotrenowanie modelu nowymi danymi.  
3. Przeanalizowanie, jak dobrze obecny model klasyfikuje dane.  
4. Cofnij do menu głównego.  
Wybieram opcję: [ ]
```

Rysunek 5: Menu - konfiguracja modelu ml

2.2.4. Mapa

Ostatnia (czwarta) opcja w głównym menu, odpowiedzialna za zwrócenie mapy z lokalizacją podejrzanych adresów IP znajdujących się w pliku PCAP. Mapa zapisywana jest w pliku map.html w katalogu report.

3. Wymagania laboratoryjne

3.1. Analiza flow A1

3.1.1. Cel

Wymaganie zakładało wczytanie danych z plików PCAP przy pomocy NFStream oraz pokazanie uzyskanego wyniku w raporcie.

3.1.2. Realizacja

Po uruchomieniu systemu wybieramy opcję 1 (analiza plików pcap) i następnie ponownie opcję 1 („Otrzymanie raportu z danymi wszystkich flow z pcap”). Zostajemy poproszeni o podanie nazwy pliku pcap, dla którego chcemy przeprowadzić analizę (plik musi znajdować się w katalogu resources projektu).

```
Wybierz z listy, jakie zadanie chciałbyś zlecić systemowi:  
1. Otrzymanie raportu z danymi wszystkich flow z pcap.  
2. Otrzymanie raportu z podejrzanymi flow + alerty bezpieczeństwa (detection rules).  
3. Otrzymanie raportu z podejrzanymi flow przy użyciu modelu ML.  
4. Cofnij do menu głównego.  
Wybieram opcję: 1  
Umieść plik pcap w katalogu resources i podaj jego nazwę: normal_traffic.pcap  
Raport został zapisany - znajdziesz go w katalogu report w pliku report.json.
```

Rysunek 6: Zapis raportu z flow do pliku

Samo wczytywanie plików z poziomu kodu zostało rozwiązane przy pomocy biblioteki NFStream. Obiekt klasy PcapAnalyzer tworzony jest poprzez podanie następujących atrybutów: normal_pcap_file, mal_pcap_file i live_interface (pozostałe dwa zostaną omówione później). Następnie przy pomocy biblioteki NFStream zawartości plików zapisywane są do odpowiednich zmiennych, jako atrybuty obiektu.

Sytuacja ulega zmianie, gdy podczas tworzenia obiektu podany został argument live_interface – wtedy zarówno do atrybutu normal_stream i mal_stream przypisywany jest ruch z interfejsu.

```
class PcapAnalyzer:  
    def __init__(self, normal_pcap_file=None, mal_pcap_file=None, live_interface=None, retrain_norm_pcap=None, retrain_mal_pcap=None) -> None:  
        self.report = Report()  
        self.map = Map()  
        self.normal_stream = None  
        self.mal_stream = None  
        self.live_interface=live_interface  
        self.retrain_norm_stream=None  
        self.retrain_mal_stream=None  
  
        if retrain_norm_pcap:  
            self.retrain_norm_stream=NFStreamer(source=retrain_norm_pcap, statistical_analysis=True)  
  
        if retrain_mal_pcap:  
            self.retrain_mal_stream=NFStreamer(source=retrain_mal_pcap, statistical_analysis=True)  
  
        if normal_pcap_file:  
            self.normal_stream = NFStreamer(source=normal_pcap_file, statistical_analysis=True)  
  
        if mal_pcap_file:  
            self.mal_stream = NFStreamer(source=mal_pcap_file, statistical_analysis=True)  
  
        if live_interface:  
            self.normal_stream = NFStreamer(source=live_interface, statistical_analysis=True, idle_timeout=1, active_timeout=1 )  
            self.mal_stream = NFStreamer(source=live_interface, statistical_analysis=True, idle_timeout=1, active_timeout=1 )  
  
        self.ml =ML(self.normal_stream,self.mal_stream)
```

Rysunek 7: Konstruktor klasy PcapAnalyzer, załadowanie pliku PCAP (A1)

3.2. Analiza flow A2

3.2.1. Cel

Dla wczytanych przepływów z wymagania A1 należało wyświetlić podsumowanie statystyk flow. Wygenerowane podsumowanie powinno zostać zawarte w raporcie z narzędzia.

3.2.2. Realizacja

Kontynuujemy scenariusz z wymagania A1. Wykonana operacja skutkuje komunikatem o poprawnym wykonaniu zadania i zapisaniu danych z raportu do pliku report.json. Za konstrukcję raportu odpowiada klasa Report.

Po otworzeniu pliku report.json naszym oczom ukazuje się spis wszystkich pakietów z pliku PCAP wraz z ich najważniejszymi danymi (src IP, dst IP, src port, dst port, protocol etc.).

```
{
  "report_id": "be55317e-3b8a-4ee8-9d8d-8fae4b52a755",
  "section 1": "Flows data",
  "flows": [
    {
      "id": 0,
      "src_ip": "::",
      "dst_ip": "ff02::1:ffel:81le",
      "src_port": 0,
      "dst_port": 0,
      "protocol": 58,
      "bidirectional_bytes": 156,
      "bidirectional_packets": 2
    },
    {
      "id": 1,
      "src_ip": "::",
      "dst_ip": "ff02::1:ff68:ef0a",
      "src_port": 0,
      "dst_port": 0,
      "protocol": 58,
      "bidirectional_bytes": 78,
      "bidirectional_packets": 1
    },
    {
      "id": 2,
      "src_ip": "::",
      "dst_ip": "ff02::1:ff00:2c6",
      "src_port": 0,
      "dst_port": 0,
      "protocol": 58,
      "bidirectional_bytes": 78,
      "bidirectional_packets": 1
    }
  ]
}
```

Rysunek 8: Podstawowe dane flow z pliku PCAP (**A2**)

Dodatkowo, w ramach spełnienia wymagania A2 w raporcie możemy odnaleźć spis ilości występowania flow dla konkretnej pary adresów IP.

```
"section 2": "IP Communications Count",
"ip_communications": {
  ":: -> ff02::1:ffel:81le": 1,
  ":: -> ff02::1:ff68:ef0a": 1,
  ":: -> ff02::1:ff00:2c6": 1,
  "fe80::981:b1fd:8de1:81le -> ff02::1:ffel:81le": 2,
  "fe80::981:b1fd:8de1:81le -> ff02::1:ff68:ef0a": 2,
  "fe80::981:b1fd:8de1:81le -> ff02::1:ff00:2c6": 1,
  "fe80::981:b1fd:8de1:81le -> ff02::1:ff00:7f6": 1,
  ":: -> ff02::1:ff00:7f6": 1,
  "fe80::294d:3ea3:b53c:c77f -> ff02::1:ff00:7f6": 2,
  "fe80::695d:d39c:6fe8:612 -> ff02::1:ff00:280": 2,
  "fe80:::4a8:b1f0:5819:1034 -> ff02::1:ff0b:2224": 2,
  "fe80:::4a8:b1f0:5819:1034 -> ff02::1:ff00:fa9": 2,
  "fe80::294d:3ea3:b53c:c77f -> ff02::1:f4f4:1942": 2,
  "fe80::695d:d39c:6fe8:612 -> ff02::1:ff0e:612": 2,
  "fe80::4c7b:f8cd:4e3e:dd3c -> ff02::1:ff00:3a7": 2,
  "fe80::4c7b:f8cd:4e3e:dd3c -> ff02::1:ff3e:dd3c": 2,
  "fe80::4c7b:f8cd:4e3e:dd3c -> ff02::1:ff92:eff9": 2,
  "192.168.1.2 -> 239.255.255.250": 3,
  "fe80::5440:d617:fa87:1f07 -> ff02::1:ff98:fae8": 2,
  "fe80::5440:d617:fa87:1f07 -> ff02::1:ff87:1f07": 2,
  "fe80::695d:d39c:6fe8:612 -> ff02::1:ffa7:09b4": 2,
  "fe80::5440:d617:fa87:1f07 -> ff02::1:ff00:99e": 2,
  "fe80::294d:3ea3:b53c:c77f -> ff02::1:ff3c:c77f": 2,
  "fe80:::4a8:b1f0:5819:1034 -> ff02::1:ff19:1034": 2,
  "fe80::981:b1fd:8de1:81le -> ff02::1:ff00:c5c": 2,
  ":: -> ff02::1:ff00:c5c": 1,
  "fd2d:ab8c:225:0:d0eb:2d62:d499:23c8 -> ff02::1:ff00:1": 2,
  "fe80::294d:3ea3:b53c:c77f -> ff02::1:ff00:f72": 1,
  "192.168.1.125 -> 239.255.255.250": 2,
  "fd2d:ab8c:225::1 -> fd2d:ab8c:225:0:65fd:f7c4:fa68:ef0a": 1,
  "fd2d:ab8c:225:0:65fd:f7c4:fa68:ef0a -> fd2d:ab8c:225::1": 7,
  ...
}
```

Rysunek 9: Podsumowanie ilości flow w pliku PCAP dla konkretnej pary adresów IP (**A2**)

Analogiczne operacje możemy wykonać dla dynamicznego (live) przechwytywania pakietów na interfejsie sieciowym – uruchamiamy z głównego menu opcję 2 i następnie opcję 1 podając

nazwę interfejsu. Ze względu na przechwytywanie live, chcąc zakończenia analizy użytkownik musi wyrazić w takiej sytuacji używając kombinacji klawiszowej CTRL+C.

3.3. Detection as a Code D1

3.3.1. Cel

Celem tego wymagania było napisanie w języku Python reguł detekcyjnych pozwalających na wykrywanie zagrożeń. Program analizując pakiety (znalezione w pliku pcap, bądź live) stara się znaleźć takie, które spełniają regułę (tzn. uważane są za podejrzane). Po ocenie przez system, że dane flow można uznać za podejrzane generowany jest alert bezpieczeństwa.

3.3.2. Realizacja

Zdecydowaliśmy się na zaimplementowanie następujących trzech reguł detekcyjnych:

- detect_long_connection - reguła odpowiedzialna za wykrywanie długotrwałych połączeń (może świadczyć np. o złośliwej komunikacji z serwerem C&C, bądź spadku wydajności serwera)
- detect_dos_attack - wykrywanie czy nie jest przekroczony limit pakietów w danym flow oraz czy komunikacja nie trwa zbyt długo (może świadczyć o potencjalnym ataku DOS)
- detect_blacklisted_IP - reguła sprawdzająca czy przechwycone flow nie pochodzi od adresu IP uznanego za podejrzany, tj. zapisanego w pliku blacklist.txt

W celu wywołania tego modułu uruchomiana jest funkcja dla klasy PcapAnalyzer find_suspicious_flows, która odpowiedzialna jest za przeanalizowanie każdego flow w badanym streamie.

```
def find_suspicious_flows(self):  
    l_f_counter=0  
    dos_counter=0  
    l_c_counter=0  
    for flow in self.mal_stream:  
        l_f_counter=self.detect_blacklisted_ip(flow,l_f_counter)  
        l_c_counter=self.detect_long_connection(flow,l_c_counter)  
        dos_counter=self.detect_dos_attack(flow,dos_counter)  
        suspicious_json = self.report.save_suspicious_flows()  
        with open("report/suspicious_report.json", "w") as f:  
            f.write(suspicious_json)  
        self.report.plot_threat_distribution(l_f_counter, l_c_counter, dos_counter)
```

Rysunek 10: Funkcja odpowiedzialna za znalezienie podejrzanych flow

Wykrycie spełnienia którejkolwiek z reguł skutkuje zwiększeniem counter'a (potrzebny do utworzenia wykresów) oraz wywołaniem funkcji create_alert dla obiektu report (tworzony jest nowy alert, dopisywane dane flow).

```
def create_alert(self, flow_id ,message,flow_data):  
    alert = Alert(flow_id, message)  
    self.generated_alerts.append(alert)  
    self.suspicious_flows.append(flow_data)
```

Rysunek 11: Funkcja odpowiedzialna za znalezienie utworzenie alertu

Po zakończeniu analizy ubierane alerty oraz dane o podejrzanych pakietach zapisywane są do pliku suspicious_report.json znajdującego się w katalogu resources.

3.3.2.1. Detect_long_connection

Reguła detekcyjna odpowiedzialna za wykrywanie podejrzanie długich połączeń (tutaj wartość ustalona na ponad 65 sekund). Oczywiście, jest to tylko przykładowa wartość, która została ustalona na potrzebę testów, w realnym zastosowaniu powinna zostać dopasowana do realiów badanego ruchu. W przypadku wykrycia podejrzanie długiego flow tworzona jest wiadomość, przy pomocy funkcji get_flow_data zbierane są odpowiednie dane o flow, a następnie następuje wywo-

łanie wcześniej wspomnianej funkcji `create_alert`. Zwiększany jest również `counter` (potrzebne do tworzenia wykresów (V1)).

```
def detect_long_connection(self, flow, counter):
    if flow.bidirectional_duration_ms > 65000:
        message = f"Long connection detected from {flow.src_ip} (duration: {flow.bidirectional_duration_ms} ms)"
        flow_data=self.get_flow_data(flow)
        self.report.create_alert(flow.id, message, flow_data)
        counter+=1
    return(counter)
```

Rysunek 12: Funkcja `detect_long_connection`

Testowanie tej reguły ograniczyliśmy do wykonania analizy pliku pcap (ze względu na czas jaki zajmowało wyłapanie reguły na interfejsie). Jak możemy zauważyc, uruchomienie analizy dla pliku `mixed_traffic.pcap` zwraca nam dane pakietów znalezionych przez regułę oraz alerty bezpieczeństwa.

```
"suspicious_flows": [
    {
    },
    {
        "id": 12,
        "src_ip": "fe80::1dfe:6c38:93c9:c808",
        "dst_ip": "ff02::1:2",
        "src_port": 546,
        "dst_port": 547,
        "protocol": 17,
        "bidirectional_bytes": 1022,
        "bidirectional_packets": 7
    }
],
```

Rysunek 13: Podstawowe dane podejrzanego flow (**D1**)

```
"alerts": [
    ],
    "12": [
        {
            "flow_id": 12,
            "alert": "Long connection detected from fe80::1dfe:6c38:93c9:c808 (duration: 65330 ms)"
        }
],
```

Rysunek 14: Rodzaj wykrytego podejrzanej zachowania (**D1**)

3.3.2.2. Detect_dos_attack

Reguła ukazująca potencjalny sposób wykrywania ataku Denial of Service. Sprawdzana jest ilość wymienionych pakietów w czasie oraz czas w jakim to nastąpiło (oczywiście tutaj wartości również są testowe, przystosowane do wykrycia przy pomocy dostępnych plików pcap - wielkości powinny zostać docelowo dostosowane do badanego ruchu). Reszta operacji jest analogiczna jak w przypadku pozostałych reguł - tworzony jest alert oraz zwiększany `counter` wykrytych zagrożeń.

```
def detect_dos_attack(self, flow, counter):
    if flow.bidirectional_packets > 50 and flow.bidirectional_duration_ms < 40000:
        message = f"Potential DoS attack detected from {flow.src_ip} with {flow.bidirectional_packets} packets in {flow.bidirectional_duration_ms} ms"
        flow_data=self.get_flow_data(flow)
        self.report.create_alert(flow.id, message, flow_data)
        counter+=1
    return counter
```

Rysunek 15: Funkcja `detect_dos_attack`

W ramach testowania tej reguły uruchomiliśmy `tcpReplay` na interfejsie z przygotowanym plikiem `mixed.pcap`.

```

root@stacis:/home/ubuntu/Desktop/lab/krycylab/resources# tcpreplay -i enp0s3 mixed.pcap
Actual: 225 packets (181839 bytes) sent in 155.28 seconds
Rated: 1171.0 Bps, 0.009 Mbps, 1.44 pps
Flows: 14 flows, 0.09 fps, 225 flow packets, 0 non-flow
Statistics for network device: enp0s3
    Successful packets:      225
    Failed packets:          0
    Truncated packets:       0
    Retried packets (ENOBUFS): 0
    Retried packets (EAGAIN): 0

```

Rysunek 16: Uruchomienie tcpreplay na interfejsie enp0s3 z plikiem mixed.pcap

Po przesłaniu wszystkich pakietów i zakończeniu analizy przeszliśmy do pliku suspicious_report.json, w którym odnaleźliśmy dane wszystkich podejrzanych pakietów oraz wylistowane alerty bezpieczeństwa.

```

"suspicious_flows": [
  {
    "id": 31,
    "src_ip": "10.0.2.106",
    "dst_ip": "64.12.107.131",
    "src_port": 49159,
    "dst_port": 80,
    "protocol": 6,
    "bidirectional_bytes": 37709,
    "bidirectional_packets": 48
  },
  {
    "id": 34,
    "src_ip": "64.12.107.131",
    "dst_ip": "10.0.2.106",
    "src_port": 80,
    "dst_port": 49159,
    "protocol": 6,
    "bidirectional_bytes": 97412,
    "bidirectional_packets": 101
  }
]

```

Rysunek 17: Dane podejrzanego flow (DOS) w pliku suspicious_report.json (D1)

```

"alerts": [
  {
    "flow_id": 34,
    "alert": "Potential DoS attack detected from 64.12.107.131 with 101 packets in 575 ms"
  }
]

```

Rysunek 18: Alert DOS dla flow w pliku suspicious_report.json (D1)

3.3.2.3. Detect_blacklisted_IP

Reguła odpowiedzialna za wykrywanie flow pochodzących od określonego adresu źródłowego, uznanego za podejrzany (wpisanego na listę w pliku blacklist.txt). Po wykryciu flow pochodzącego z adresu IP z blackisty generowany jest alert i zwiększany counter odnotowanych zagrożeń. W naszej realizacji na liście znajduje się jeden adres IP (dla testu), natomiast docelowo użytkownik ma możliwość uzupełniania jej o własne, podejrzane adresy.

```

def detect_blacklisted_ip(self, flow, counter):
    with open("resources/blacklist.txt", 'r') as f:
        blacklist=set(line.strip() for line in f)
    if str(flow.src_ip) in blacklist:
        message = f"Potential attack from suspicious ip: {flow.src_ip}"
        flow_data = self.get_flow_data(flow)
        self.report.create_alert(flow.id, message, flow_data)
        counter += 1
    return counter

```

Rysunek 19: Funkcja detect_blacklisted_ip

Sposób testowania tej reguły był bardzo podobny do reguły detect DOS - uruchomiony został tcpreplay z plikiem mixed.pcap oraz nasłuchiwanie live na naszym narzędziu. Dzięki temu otrzymaliśmy dane podejrzanych flow oraz alerty dotyczące podejrzanej adresu IP w pliku suspicious_report.json.

```

1  {
2      "report_id": "21f4c543-628c-476a-983d-4e2123ec6a94",
3      "suspicious_flows": [
4          {
5              "id": 5,
6              "src_ip": "10.0.2.106",
7              "dst_ip": "8.8.4.4",
8              "src_port": 62105,
9              "dst_port": 53,
10             "protocol": 17,
11             "bidirectional_bytes": 76,
12             "bidirectional_packets": 1
13         },
14         {
15             "id": 7,
16             "src_ip": "10.0.2.106",
17             "dst_ip": "8.8.8.8",
18             "src_port": 62364,
19             "dst_port": 53,
20             "protocol": 17,
21             "bidirectional_bytes": 76,
22             "bidirectional_packets": 1
23         },
24         {
25             "id": 8,
26             "src_ip": "10.0.2.106",
27             "dst_ip": "8.8.4.4",
28             "src_port": 62364,
29             "dst_port": 53,
30             "protocol": 17,
31             "bidirectional_bytes": 76,
32             "bidirectional_packets": 1
33         }

```

Rysunek 20: Dane podejrzanego flow (Black List) w pliku suspicious_report.json (D1)

```

"alerts": [
    {
        "flow_id": 5,
        "alert": "Potential attack from suspicious ip: 10.0.2.106"
    },
    {
        "flow_id": 7,
        "alert": "Potential attack from suspicious ip: 10.0.2.106"
    },
    {
        "flow_id": 8,
        "alert": "Potential attack from suspicious ip: 10.0.2.106"
    }
]

```

Rysunek 21: Alerty BlackListed IP dla flow w pliku suspicious_report.json (D1)

3.4. Wizualizacja V1

3.4.1. Cel

Celem tego kryterium było zwizualizowanie (w formie wykresu lub tabelki) znalezionych zagrożeń.

3.4.2. Realizacja

Po wykonaniu wyszukania zagrożeń przy pomocy reguł detekcyjnych tworzony jest wykres kołowy przedstawiający rodzaj wykrytych zagrożenia na przestrzeni czasu zarówno w przypadku trybu z przechwytywaniem ruchu oraz w trybie z analizą przechwyconego wcześniej pakietu.



Rysunek 22: Wykres kołowy dzielący analizowany ruch na rodzaj zagrożeń

4. Wymagania projektowe

4.1. Detection as a Code (D2)

4.1.1. Cel

Celem użycia PySigma jest wsparcie automatyzacji i standaryzacji procesu wykrywania zagrożeń w danych sieciowych. Dzięki PySigma możemy efektywnie wykorzystać reguły detekcji zapisane w formacie Sigma, co pozwala między innymi na automatyzację detekcji zagrożeń, standaryzowanie reguł oraz wspieranie analizy sieciowej.

PySigma jest biblioteką Python, która umożliwia przetwarzanie reguł zapisanych w formacie Sigma. Pozwala ona na odczytywanie, przetwarzanie i konwersję reguł Sigma do konkretnych języków zapytań (tzw. backendów), takich jak np. zapytania do Elasticsearch, Splunk czy innych systemów analizy logów.

Jest ona szczególnie istotna ze względu na standaryzację. Sigma jest językiem niezależnym od platformy, co pozwala na łatwą współpracę z wieloma systemami analizy logów. Może również pracować z różnymi backendami, co zwiększa wszechstronność aplikacji.

4.1.2. Realizacja

Niestety, realizacja tego wymagania nie powiodła się - mieliśmy problem z zaimplementowaniem reguły przy pomocy PySigma (problem ze znalezieniem odpowiednich materiałów źródłowych do tematu). Ze względu na ograniczony czas realizacji zadania zdecydowaliśmy się pominięcie tego wymagania, jednak zastąpienia go pozostałymi wymaganiami nice-to-have oraz dodaniem większej ilości standardowych reguł detekcyjnych (3).

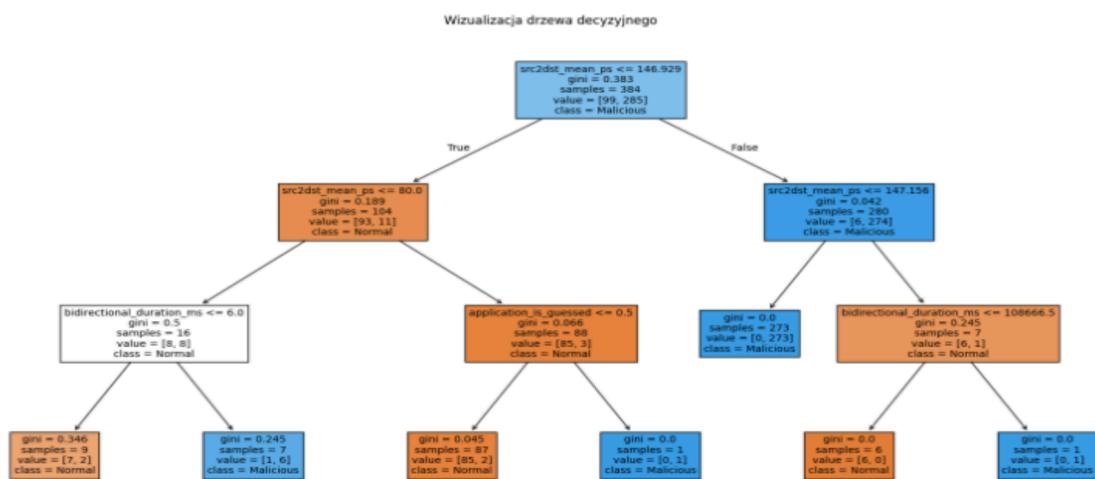
4.2. Machine Learning (ML1)

4.2.1. Cel

Celem tego kryterium jest klasyfikowanie flow przy pomocy modelu Machine Learning (drzewo decyzyjne). Dobrym rozwiązaniem jest też pokazanie użytkownikowi modelu, ewentualnie macieży błędu. Sklasyfikowane dane powinny trafić do raportu.

4.2.2. Realizacja

W ramach naszego systemu wdrożyliśmy model (drzewo decyzyjne), który jest dostępny w ramach naszego systemu, zaraz po jego uruchomieniu (na bazie zaszytych w naszym rozwiążaniu plików normal_traffic.pcap i malicious_traffic.pcap). Model jest przechowywany w katalogu, natomiast jednocześnie użytkownik zawsze może podejrzeć wygląd modelu w pliku decision_tree.png w katalogu report.



Rysunek 23: Wygenerowany model

Jednocześnie po wejściu w opcje konfiguracji ML (opcja 3 w głównym menu) użytkownik może wybrać opcję 1, która pozwala na wygenerowanie własnego modelu drzewa decyzyjnego (na bazie własnych plików).

```
Wybierz z listy, jakie zadanie chciałbyś zlecić systemowi:  
1. Wygenerowanie własnego modelu drzewa decyzyjnego do klasyfikacji flow.  
2. Dotrenowanie modelu nowymi danymi.  
3. Przeanalizowanie, jak dobrze obecny model klasyfikuje dane.  
4. Cofnij do menu głównego.  
Wybieram opcję: 1
```

Rysunek 24: Menu konfiguracyjne z opcjami dotyczącymi modelu ML

Za tworzenie, trenowanie i testowanie modelu odpowiada funkcja `train_and_evaluate_decision_tree` z klasy ML. Pobiera ona dane, przygotowuje je - funkcja `prepare_data`(tagując dane normalne etykietą 0 i podejrzane etykietą 1), następnie przechodzi do podziału zbioru na trenujący i testowy i dokonuje stworzenia modelu. Dodatkowo, obliczana jest dokładność modelu i tworzona jest macierz błędu(również wyświetlana użytkownikowi i dostępna w katalogu resources).

```

def train_and_evaluate_decision_tree(self):
    X_train, X_test, y_train, y_test = self.prepare_data()
    self.tree_model = DecisionTreeClassifier(max_depth=3, criterion='gini', random_state=42)
    self.tree_model.fit(X_train, y_train)

    predictions = self.tree_model.predict(X_test)
    self.accuracy = accuracy_score(y_test, predictions)
    self.conf_matrix = confusion_matrix(y_test, predictions)

    dump(self.tree_model, 'decision_tree_model.joblib')

```

Rysunek 25: Funkcja train_and_evaluate_decision_tree

W przypadku chęci sklasyfikowania ruchu na bazie modelu dostępnego w urządzeniu wybieramy opcję 3 dla analizy pakietów lub analizy live („Otrzymanie raportu z podejrzanych flow przy użyciu ML”). W celu weryfikacji uruchomiliśmy tcpreplay na interfejsie z plikiem mixed.pcap.

```

root@stacis:/home/ubuntu/Desktop/lab/krycyLab/resources# tcpreplay -i enp0s3 mixed.pcap
Actual: 225 packets (181839 bytes) sent in 155.28 seconds
Rated: 1170.9 Bps, 0.009 Mbps, 1.44 pps
Flows: 14 flows, 0.09 fps, 225 flow packets, 0 non-flow
Statistics for network device: enp0s3
    Successful packets:      225
    Failed packets:          0
    Truncated packets:       0
    Retried packets (ENOBUFS): 0
    Retried packets (EAGAIN): 0

```

Rysunek 26: Generowanie ruchu do przechwycenia przy pomocy narzędzia **tcpreplay**

Po przesłaniu wszystkich pakietów na interfejs, efekt możemy zobaczyć w pliku suspicious_reports.json, w którym znajdują się wszystkie flow uznane przez model ML za podejrzane (dane każdego flow + alert o podejrzany flow).

```

{
  "report_id": "93934511-da9a-44f4-a124-c9a4dd4c3a4e",
  "suspicious_flows": [
    {
      "id": 11,
      "src_ip": "10.0.2.106",
      "dst_ip": "213.155.158.83",
      "src_port": 49158,
      "dst_port": 80,
      "protocol": 6,
      "bidirectional_bytes": 44344,
      "bidirectional_packets": 49
    },
    {
      "id": 15,
      "src_ip": "10.0.2.106",
      "dst_ip": "224.0.0.252",
      "src_port": 57874,
      "dst_port": 5355,
      "protocol": 17,
      "bidirectional_bytes": 128,
      "bidirectional_packets": 2
    },
    {
      "id": 17,
      "src_ip": "10.0.2.106",
      "dst_ip": "224.0.0.252",
      "src_port": 51605,
      "dst_port": 5355,
      "protocol": 17,
      "bidirectional_bytes": 128,
      "bidirectional_packets": 2
    }
  ]
}

```

Rysunek 27: Dane podejrzanego flow (Black List) w pliku suspicious_report.json

```

"alerts": {
    "11": [
        {
            "flow_id": 11,
            "alert": "Potential malicious flow from 10.0.2.106 to 213.155.158.83 with id: 11."
        }
    ],
    "15": [
        {
            "flow_id": 15,
            "alert": "Potential malicious flow from 10.0.2.106 to 224.0.0.252 with id: 15."
        }
    ],
    "17": [
        {
            "flow_id": 17,
            "alert": "Potential malicious flow from 10.0.2.106 to 224.0.0.252 with id: 17."
        }
    ]
}

```

Rysunek 28: Alerty BlackListed IP dla flow w pliku suspicious_report.json

4.3. Machine Learning (ML2)

4.3.1. Cel

Narzędzie powinno dawać możliwość wizualizacji jakości modelu (macierz pomyłek, dokładność).

4.3.2. Realizacja

W naszym rozwiążaniu użytkownik ma dostęp do obecnej macierzy błędów w pliku confusion_matrix.png w katalogu report (macierz wygenerowana dla modelu zaszytego w programie lub modelu stworzonego przez użytkownika).



Rysunek 29: Macierz błędów modelu

Dodatkowo za każdym razem, gdy użytkownik tworzy nowy model jego oczom pokazuje się informacja o dokładności modelu.

```

Umieśc plik pcap o nazwie 'normal_traffic.pcap' z pozytywnymi próbками w katalogu resources:
Umieśc plik pcap o nazwie 'malicious_traffic.pcap' z złośliwymi próbками w katalogu resources:
Naciśnij Enter, kiedy umieścisz pliki w katalogu resources...
Dokładność modelu to: 0.9515151515151515

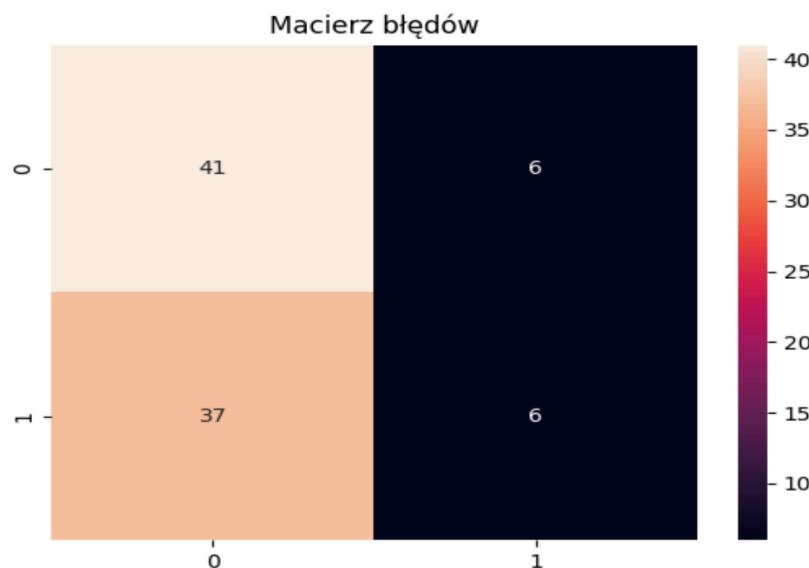
```

Rysunek 30: Wyliczanie dokładności modelu

Aby użytkownik mógł zobaczyć, czy aktualny model w dobry sposób klasyfikuje jego zbiór danych (sytuacja, w której użytkownik dysponuje zbiorem flow pozytywnych i podejrzanych) użytkownik

przed wygenerowaniem własnego modelu (opcja 1), może sprawdzić jak dobrze obecny model klasyfikuje jego zbiór danych. W tym celu należy wybrać w menu konfiguracji ML opcję 3 („Przeanalizowanie jak dobrze obecny model klasyfikuje dane”).

Po wyborze tej opcji użytkownik proszony jest o umieszczenie swoich plików w katalogu resources i podanie ich nazwy, a następnie oblicza dokładność modelu dla danych oraz przedstawia macierz błędów.



Rysunek 31: Otrzymana macierz błędów

```
Umieść plik pcap z pozytywnymi próbками w katalogu resources i podaj jego nazwę: retrain_norm.pcap  
Umieść plik pcap z złośliwymi próbками w katalogu resources i podaj jego nazwę: retrain_mal.pcap  
Naciśnij Enter, kiedy umieścisz pliki w katalogu resources...  
Dokładność modelu to: 0.5222222222223  
Model przetestowany na nowych danych.
```

Rysunek 32: Liczenie dokładności modelu

Jak możemy zauważyć dane dla plików retrain_norm.pcap i retrain_mal.pcap są poprawnie klasyfikowane w jedynie **52%** (co może sugerować konieczność albo wygenerowania nowego modelu (opcja 1) albo dotrenowania go nowymi danymi (opcja 2, zostanie omówiona w kolejnym rozdziale)).

4.4. Machine Learning (ML3)

4.4.1. Cel:

Jako dodatkowe zadanie (nice-to-have) należało dodać możliwość dotrenowania modelu nowymi danymi (jest to opcja 2 w menu konfiguracji ML).

4.4.2. Realizacja

Użytkownik proszony jest o podanie dwóch plików (pozytywnego i podejrzanego). Następnie program łączy nowe pliki ze starymi i przeprowadza proces ponownej nauki modelu na zwiększonym zbiorze.

Funkcja do ponownego trenowania modelu wygląda bardzo podobnie do funkcji trenowania, jednak różni się sposób przygotowania danych (tam łączone są nowe i stare pliki, aby trening oparty był o większą ilość danych).

```

def retrain_model(self, new_normal_stream, new_malicious_stream):
    X_train, X_test, y_train, y_test = self.prepare_data_for_retrain(new_normal_stream, new_malicious_stream)
    self.tree_model = DecisionTreeClassifier(max_depth=3, criterion='gini', random_state=42)
    self.tree_model.fit(X_train, y_train)

    predictions = self.tree_model.predict(X_test)
    self.accuracy = accuracy_score(y_test, predictions)
    self.conf_matrix = confusion_matrix(y_test, predictions)

    dump(self.tree_model, 'decision_tree_model.joblib')

```

Rysunek 33: Funkcja retrain_model

Przypominając sobie scenariusz z poprzedniego rozdziału (przetestowana jakość modelu na nowych danych to około **52%**) wykonaliśmy proces dotrenowania modelu danymi, na których użytkownik uprzednio testował model(retrain_norm.pcap i retrain_mal.pcap).

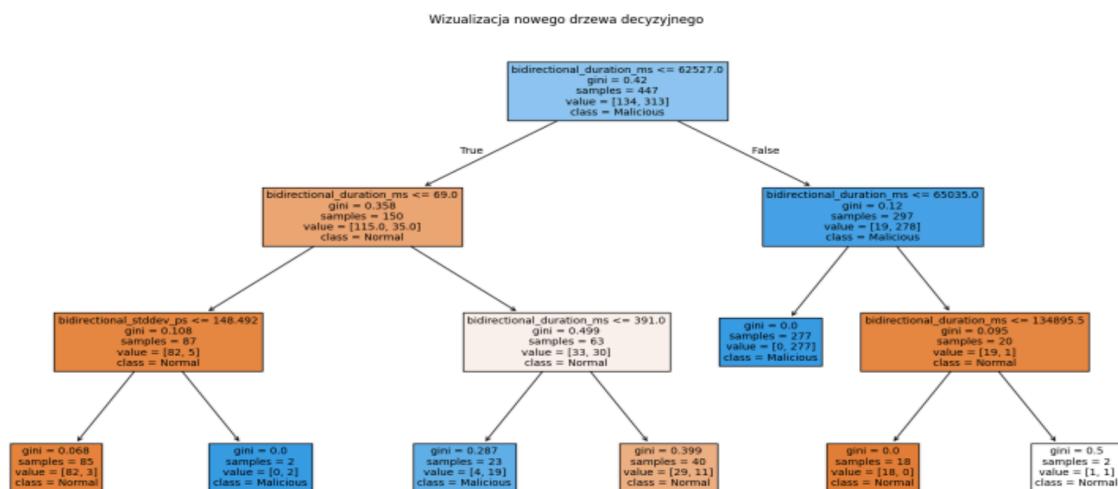
Po utworzeniu nowego modelu (dotrenowaniu go nowymi danymi) możemy sprawdzić ponownie jak dobrze obczony model klasyfikuje próbki dostarczone przez użytkownika. W tym celu uruchamiamy opcję 3 w panelu konfiguracji ML i uzyskujemy nową dokładność na poziomie **82 %**.

```

Wybieram opcję: 3
Umieść plik pcap z pozytywnymi próbami w katalogu resources i podaj jego nazwę: retrain_norm.pcap
Umieść plik pcap z złośliwymi próbami w katalogu resources i podaj jego nazwę: retrain_mal.pcap
Naciśnij Enter, kiedy umieścisz pliki w katalogu resources...
Dokładność modelu to: 0.8222222222222222

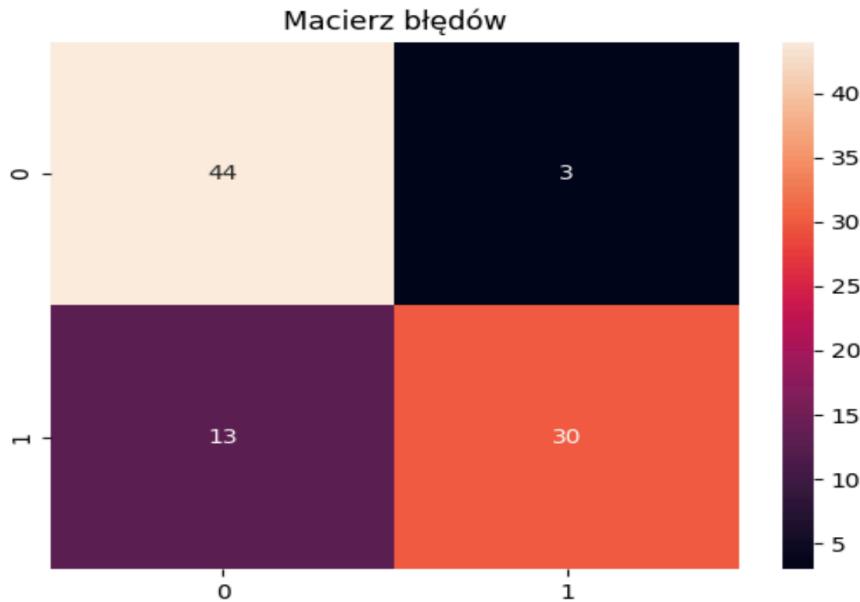
```

Rysunek 34: Liczenie dokładności modelu



Rysunek 35: Drzewo modelu po dotrenowaniu

Dodatkowo o zwiększeniu poprawności klasyfikowania nowych danych przez użytkownika świadczy nowa macierz błędów (znaczący spadek ilości true-negatives i false-positives).



Rysunek 36: Macierz błędów po dotrenowaniu dla danych wprowadzonych przez użytkownika

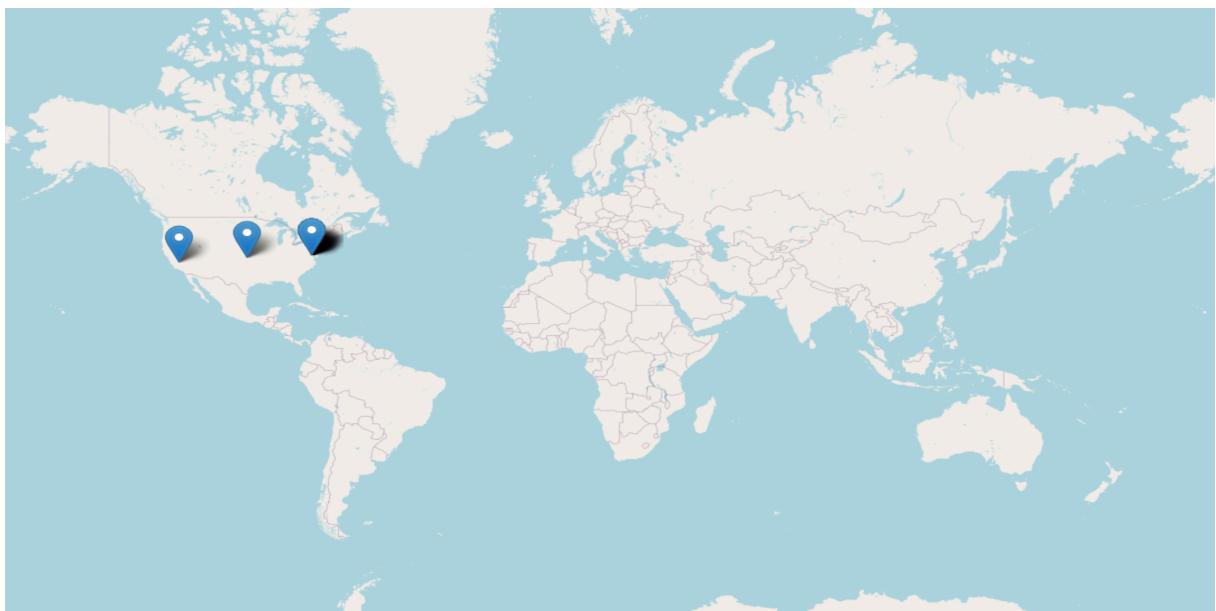
4.5. Enrichment (E1) i Wizualizacja (V2)

4.5.1. Cel

Narzędzie musi pobierać podstawowe informacje o IP/domenach przy użyciu API oraz jako „Nice-to-have” może tworzyć mapkę geograficzną przedstawiającą lokalizację adresów IP wykrytych jako podejrzane.

4.5.2. Realizacja

Ze względu powiązania punktów Enrichment i Wizualizacja zdecydowaliśmy się na ich połączenie. W efekcie aplikacja została wyposażona w funkcję tworzenia mapki z wykrytymi adresami IP. Działa ona wykorzystując bibliotekę **folium**, która na podstawie dostępnych danych lokalizuje źródło przechwyconego ruchu. Jak możemy zauważyć na poniższym rysunku, przykładowe dane wykorzystywane do testowania modelu mają swoje źródło w 3 lokalizacjach na terenie Stanów Zjednoczonych.



Rysunek 37: Lokalizowanie źródła analizowanego ruchu