

Projekt ADAC

Wykrywanie SQL Injection z wykorzystaniem uczenia maszynowego

Wojciech Drabik, Jarosław Smardzewski, Joanna Wysmułek, Weronika Zipser

Politechnika Warszawska, Cyberbezpieczeństwo, Studia II stopnia, Semestr 1

17 kwietnia 2024

Spis treści

1. Wstęp	3
1.1. Cel pracy	3
2. Przegląd literatury	4
3. Zebranie danych	5
3.1. Konfiguracja systemu IDS	5
3.2. Automatyzacja zapytań SQLi	8
3.3. Bot generujący poprawne zapytania	9
4. Przygotowanie zbioru danych	12
4.1. Preprocessing zebranych zapytań	13
4.2. Analiza danych	15
5. Przeprowadzenie uczenia maszynowego	21
6. Wyniki	25
7. Wnioski	27

1. Wstęp

Realizacja projektu z przedmiotu ADAC dotyczyła przeprowadzenia wszystkich etapów procesu uczenia maszynowego modelu. Projekt składał się z następujących etapów:

- Zdefiniowanie celu projektu,
- Przegląd literatury,
- Zebranie danych,
- Preprocessing i analiza danych,
- Stworzenie modelu z wykorzystaniem uczenia maszynowego,
- Analiza otrzymanych wyników i wnioski.

Grupa projektowa otrzymała dowolność w wyborze tematyki i problemu, którego dotyczyć będzie tworzony model. Nałożonym ograniczeniem był wymóg samodzielnego zebrania danych, które zostaną wykorzystane jako zbiór trenujący i testowy w fazie uczenia modelu. Do sprawozdania dodane zostało archiwum zawierające logi wykorzystane w procesie uczenia oraz Jupyter Notebook z kodem realizującym wszystkie etapy uczenia maszynowego.

1.1. Cel pracy

Celem pracy przyjętym przez grupę projektową jest **"Wykrywanie SQL Injection z wykorzystaniem uczenia maszynowego"**.

Projekt koncentruje się na 3 głównych elementach:

- Zaprojektowanie i implementacja architektury monitoringu pozwalającej na rejestrowanie zapytań HTTP, w tym zapytań zawierających SQLi,
- Analiza zebranych zapytań HTTP pod kątem zdefiniowania paramentów charakterystycznych dla zapytań zawierających SQLi,
- Stworzenie efektywnego modelu potrafiącego odróżnić niegroźne zapytania od zapytań zawierających SQLi.

Na podstawie wyników uczenia maszynowego zaproponowany zostanie mechanizm, o który może zostać wzbogacony, wykorzystany na wcześniejszym etapie, system monitoringu. Zaproponowany mechanizm ma za zadanie zwiększyć skuteczność wykrywania zapytań SQLi, co przełoży się na zwiększenie poziomu bezpieczeństwa chronionego systemu.

2. Przegląd literatury

Jako wiedza teoretyczna przygotowująca do realizacji projektu posłużyły głównie trzy publikacje:

- A. Joshi and V. Geetha, "SQL Injection detection using machine learning," 2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (IC-CICCT), Kanyakumari, India, 2014, pp. 1111-1115, DOI: 10.1109/ICCICCT.2014.6993127.
- Mishra Sonali, "SQL Injection Detection Using Machine Learning" (2019). Master's Projects. 727. DOI: 10.31979/etd.j5dj-ngvb
- Ding Chen et al SQL Injection Attack Detection and Prevention Techniques Using Deep Learning 2021 J. Phys.: Conf. Ser. 1757 012055 DOI: 10.1088/1742-6596/1757/1/012055

Tematyka wymienionych powyżej dokumentów jest zbliżona do tej realizowanej w ramach projektu. Publikacje pozwoliły na określenie zbioru cech parametrów zapytania HTTP, które mogą wskazywać na występowanie SQLi. Szczegóły dotyczące cech, które wykorzystane zostały jako atrybuty rekordów użytych w ML, zawarte zostały w **Rozdziale 4.1**.

Publikacje zawierają również opis metod użytych w procesie uczenia maszynowego, jak również pozwoliły zdefiniować zbiór metryk pozwalających na ocenę rezultatów stworzonego modelu.

Główną cechą wyróżniającą realizowany projekt na tle wymienionych publikacji jest sposób pozyskiwania i rejestrowania zapytań HTTP. Szczegóły zaprojektowanej architektury zostały zawarte w **Rozdziale 3**.

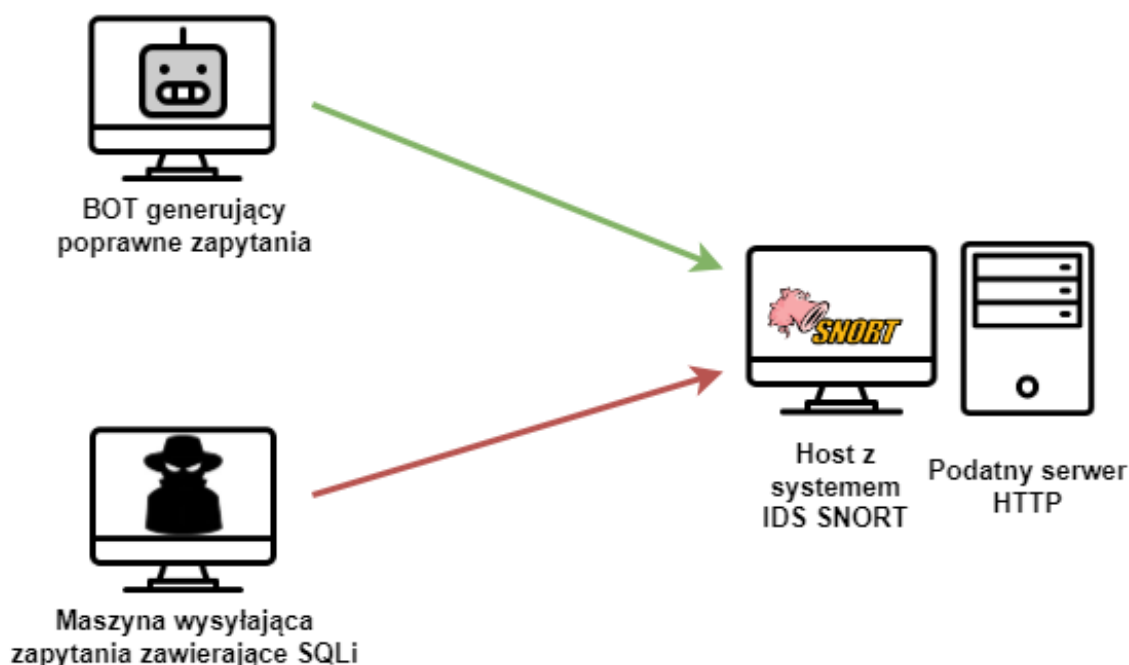
3. Zebranie danych

Dane potrzebne do przeprowadzenia uczenia maszynowego zostały zebrane w specjalnie przygotowanym do tego celu środowisku.

Środowisko składa się z następujących elementów:

- Bot realizujący poprawne zapytania HTTP,
- Maszyna z podatnym serwerem WWW i systemem IDS **Snort**,
- Maszyna przeprowadzająca ataki SQLi,

Wzajemne relacje elementów systemu są pokazane na **Rysunku 1**.



Rys. 1: Architektura laboratoryjna wykorzystana w trakcie realizacji projektu

W ramach przeprowadzonego procesu zbierania danych pozyskane zostało 21403 próbek ruchu sieciowego.

3.1. Konfiguracja systemu IDS

System monitoringu i rejestracji zapytań wysyłanych przez użytkownika stworzony został na bazie narzędzia open source: **Snort**. Narzędzie jest rozwiązaniem IDS (Intrusion Detection System) pozwalającym na definiowanie reguł filtracji ruchu oraz udostępniającym użytkownikowi szereg modułów (określanych jako Preprocessors) pozwalających na pogłębioną analizę pakietów sieciowych i

rejestrację w logach dodatkowych informacji o zapytaniach.

Moduły dokonujące analizy pakietów zostały skonfigurowane w następujący sposób:

```
config paf_max: 16000
preprocessor frag3_global
preprocessor frag3_engine
preprocessor stream5_global: \
    max_tcp 8192, track_tcp yes, track_udp yes, track_icmp yes
preprocessor stream5_icmp
preprocessor stream5_tcp: \
    ports both 80
preprocessor stream5_udp: \
    ignore_any_rules
preprocessor http_inspect: global memcap 327710 iis_unicode_map
    ↪ /usr/src/snort-2.9.20/etc/unicode.map 1252
preprocessor http_inspect_server: server default \
    profile all ports { 80 8080 8180 } oversize_dir_length 500 \
    log_uri log_hostname
preprocessor rpc_decode: 111 32771
preprocessor bo
output unified2: filename log
include $RULE_PATH/sql.rules
```

Kod 1: Konfiguracja narzędzia Snort

Zaimportowane moduły mają za zadanie:

- Rekonstruować zdefragmentowane pakiety TCP,
- Scrapować zrekonstruowane pakiety pod kątem pełnego adresu URL zapytania oraz adresu IP nadawcy pakietu,
- zapisywać uzyskane informacje w specjalnym formacie binarnym.

W celu rejestracji potencjalnych zapytań SQLi zdefiniowany został zbiór reguł filtrujących ruch na endpointach zawierających parametry z badaną podatnością:

```
alert tcp any any <> ip_addr (msg:"SQL_detection"; sid:201;content:"GET";
    ↳ http_method;content:"products.php"; http_uri;)
alert tcp any any <> ip_addr (msg:"SQL_detection"; sid:202;content:"GET";
    ↳ http_method;content:"details.php"; http_uri;)
alert tcp any any <> ip_addr (msg:"SQL_detection"; sid:203;content:"POST";
    ↳ http_method; content:"login.php"; http_uri;)
```

Kod 2: Reguły filtracji pakietów zaimplementowane w ramach narzędzia Snort

Uruchomione narzędzie zapisywało w formacie binarnym informacje o wszystkich zapytaniach wysyłanych na podatne endpointy.

(Event)

```
sensor id: 0 event id: 1 event second: 1680093070 event microsecond:
    ↳ 911037
sig id: 202 gen id: 1 revision: 0 classification: 0
priority: 0 ip source: 10.0.2.8 ip destination: 10.0.2.31
src port: 48884 dest port: 80 protocol: 6 impact_flag: 0 blocked: 0
```

Packet

```
sensor id: 0 event id: 1 event second: 1680093070
packet second: 1680093070 packet microsecond: 911037
linktype: 1 packet_length: 335
[ 0] 08 00 27 7A F9 38 08 00 27 7C 8E 8E 08 00 45 00 ..'z.8..'|....E.
[ 16] 01 41 20 70 40 00 40 06 01 21 0A 00 02 08 0A 00 .A p@.@...!.....
[ 32] 02 1F BE F4 00 50 4B CF 4D 4D 05 51 7C 4A 80 18 .....PK.MM.Q|J..
[ 48] 01 F6 D0 9C 00 00 01 01 08 0A E4 C0 23 3F 00 19 .....#?...
[ 64] 37 05 47 45 54 20 2F 64 65 74 61 69 6C 73 2E 70 7.GET /details.p
[ 80] 68 70 3F 70 72 6F 64 3D 35 26 74 79 70 65 3D 32 hp?prod=5&type=2
[ 96] 20 48 54 54 50 2F 31 2E 31 0D 0A 43 61 63 68 65 HTTP/1.1..Cache
[ 112] 2D 43 6F 6E 74 72 6F 6C 3A 20 6E 6F 2D 63 61 63 -Control: no-cac
[ 128] 68 65 0D 0A 55 73 65 72 2D 41 67 65 6E 74 3A 20 he..User-Agent:
[ 144] 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 58 31 31 Mozilla/5.0 (X11
[ 160] 3B 20 55 3B 20 4C 69 6E 75 78 20 69 36 38 36 3B ; U; Linux i686;
```

```
[ 176] 20 72 75 2D 52 55 3B 20 72 76 3A 31 2E 39 2E 32 ru-RU; rv:1.9.2
[ 192] 61 31 70 72 65 29 20 47 65 63 6B 6F 2F 32 30 30 a1pre) Gecko/200
[ 208] 39 30 34 30 35 20 55 62 75 6E 74 75 2F 39 2E 30 90405 Ubuntu/9.0
[ 224] 34 20 28 6A 61 75 6E 74 79 29 20 46 69 72 65 66 4 (jaunty) Firef
[ 240] 6F 78 2F 33 2E 36 61 31 70 72 65 0D 0A 48 6F 73 ox/3.6a1pre..Hos
[ 256] 74 3A 20 31 30 2E 30 2E 32 2E 33 31 0D 0A 41 63 t: 10.0.2.31..Ac
[ 272] 63 65 70 74 3A 20 2A 2F 2A 0D 0A 41 63 63 65 70 cept: /*/*..Accep
[ 288] 74 2D 45 6E 63 6F 64 69 6E 67 3A 20 67 7A 69 70 t-Encoding: gzip
[ 304] 2C 64 65 66 6C 61 74 65 0D 0A 43 6F 6E 6E 65 63 ,deflate..Connec
[ 320] 74 69 6F 6E 3A 20 63 6C 6F 73 65 0D 0A 0D 0A tion: close....
```

```
(ExtraDataHdr)
```

```
event type: 4 event length: 58
```

```
(ExtraData)
```

```
sensor id: 0 event id: 1 event second: 1680093070
```

```
type: 9 datatype: 1 bloblength: 34 HTTP URI: /details.php?pr
```

```
↪ od=5&type=2
```

```
(ExtraDataHdr)
```

```
event type: 4 event length: 41
```

```
(ExtraData)
```

```
sensor id: 0 event id: 1 event second: 1680093070
```

```
type: 10 datatype: 1 bloblength: 17 HTTP Hostname: 10.0.2.31
```

Kod 3: Pojedynczy rekord wygenerowany na podstawie zarejestrowanego połączenia

3.2. Automatyzacja zapytań SQLi

Realizując projekt przyjęte zostało założenie, że atakujący nie przygotowują własnych ataków SQLi, a zamiast tego posiłkują się dostępnymi na rynku skanerami podatności. Ponadto skanery podatności umożliwiają przeprowadzenie znacznie bardziej różnorodnych ataków, niż miałyby to miejsce w sytuacji tworzenia własnych niebezpiecznych zapytań. Wykonany został zatem przegląd dostępnych na rynku skanerów podatności.

Wyróżnić należy następujące narzędzia:

- SQLMap
- JSQI Injection Tool
- DSSS
- Explo
- BBQSQL
- Leviathan

Do przeprowadzenia testowego ataku SQLi zostały wykorzystane narzędzia wyróżnione kolorem zielonym: **SQLMap**, **JSQI Injection Tool** i **DSSS**. Narzędzia te zostały wybrane ze względu na dużą ilość wykonywanych zapytań i różnorodność wykorzystywanych technik. Rozległy zbiór danych jest niezwykle istotny by zagwarantować stworzenie skutecznego modelu.

Wykorzystane skanery wygenerowały w sumie 20403 zapytań, które obejmowały następujące kategorie ataków SQLi:

- **Classic SQLi** - klasyczny rodzaj ataku, w którym atakujący może wykorzystać kanał komunikacji do jednoczesnego wysłania zapytania i otrzymania na nie odpowiedzi,
- **Error-based SQLi** - atak polegający na wiadomościach błędu zwracanych przez bazę danych pozwalających na poznanie struktury atakowanej bazy,
- **Union-based SQLi** - atak polegający na wykorzystaniu operatora UNION umożliwiającego połączenie dwóch zapytań SELECT,
- **Blind SQLi** - atak, w którym atakujący nie widzi bezpośredniego rezultatu wykonanego polecenia. Atakujący obserwuje zachowanie systemu na podstawie reakcji na szereg wysyłanych payloadów,
- **Boolean-based Blind SQLi** - atak typu Blind SQLi polegający na wysyłaniu wyrażeń logicznych. W zależności od prawdziwości wysłanego zapytania, system wysyła inną odpowiedź, co pozwala ocenić, czy informacje zawarte w wyrażeniu logicznym były poprawne,
- **Time-based Blind SQLi** - atak typu Blind SQLi polegający na wysyłaniu zapytań zawierających operator sleep(). Jeśli po wysłaniu zapytania wystąpi opóźnienie otrzymania odpowiedzi, atakujący wie, że wstrzyknięte zapytanie jest poprawne i zostało wykonane.

3.3. Bot generujący poprawne zapytania

Do symulacji poprawnych zapytań HTTP został utworzony bot, napisany w języku **Python**.

Program spełniał poniższe funkcje:

- Wysyłanie zapytań na punkty końcowe: **product** i **details**,
- Wypełnienie pola **param** losowym ciągiem znaków **ASCII** o długości od 2 do 30 znaków,
- Generowanie zapytań co 0.5 sekundy,
- Wysyłanie 1000 zapytań do serwera WWW.

```
import requests
import random
import time
import string
ADDRESS = '192.168.43.6'
PORT = '80'
SLEEP = 0.5
ENDPOINTS = ['products.php','details.php']
ALPHABET = list(string.ascii_uppercase)
RUNTIME = 1000
while(RUNTIME >= 0):
    target = ENDPOINTS[random.randint(0, len(ENDPOINTS) - 1)]
    param = ''
    if (target == 'products.php'):
        param = 'type='
        for x in range(0, random.randint(2, 30)):
            param = param + random.choice(ALPHABET)
    elif(target == 'details.php'):
        param = 'type='
        for x in range(0, random.randint(2, 30)):
            param = param + random.choice(ALPHABET)
        param = param + '&prod='
        for x in range(0, random.randint(2, 30)):
            param = param + random.choice(ALPHABET)
        response = requests.get('http://' + ADDRESS + ":" + PORT + "/" + target +
                                ↪ '?' + param )
    if(response.status_code == 200):
        RUNTIME = RUNTIME - 1
        time.sleep(SLEEP)
print(RUNTIME)
```

```
print('http://' + ADDRESS + ":" + PORT + "/" + target + '?' + param)
```

Kod 4: Skrypt generujący poprawne zapytania

4. Przygotowanie zbioru danych

Korzystając z logów wygenerowanych przez narzędzie **Snort** został przeprowadzony scraping. Miał on na celu obróbkę zgromadzonych danych do postaci **Pandas DataFrame**. Scraping logów został przeprowadzony z wykorzystaniem skryptu w języku Python:

```
import re
import pandas as pd

captured_records = []
class captured_record:
    def __init__(self, url, event_time, hostname):
        self.url = url
        self.event_time = event_time
        self.hostname = hostname

    def print_record(self):
        print("Record_content")
        print(self.url)
        print(self.event_time)
        print(self.hostname)

def load_snort_result(filename):
    with open(filename, "r") as file:
        lines = file.readlines()
    for line in lines:
        line = line.rstrip()
        if(re.search("event_second", line) and re.search("event_microsecond",
            ↪ line) ):
            event_seconds, event_microseconds = line.split("event_second:↪
            ↪ ").[1].split("_event_microsecond:↪")
            event_time_tmp = float(event_seconds) + float(event_microseconds) *
            ↪ pow(10, -6)
        if(re.search("HTTP_URI", line)):
            url_tmp = line.split("HTTP_URI:↪")[1]
        if(re.search("HTTP_Hostname", line)):
```

```
hostname_tmp = line.split("HTTP_Hostname:")[1]
captured_records.append(captured_record(url_tmp, event_time_tmp,
    ↪ hostname_tmp))
captured_records_columns = ["full_url", "event_time", "hostname", "injection"]
captured_records_df = pd.DataFrame(columns=captured_records_columns)

load_snort_result("sqli_log.txt")
for record in captured_records:
    row = pd.DataFrame([[record.url, record.event_time, record.hostname, 1]],
    ↪ columns=captured_records_columns)
    captured_records_df = pd.concat([captured_records_df, row],
    ↪ ignore_index=True)

captured_records = []
load_snort_result("no_sqli_log.txt")
for record in captured_records:
    row = pd.DataFrame([[record.url, record.event_time, record.hostname, 0]],
    ↪ columns=captured_records_columns)
    captured_records_df = pd.concat([captured_records_df, row],
    ↪ ignore_index=True)

captured_records_df.to_csv('sql_df.csv', index=False)
```

Kod 5: Skrypt wykorzystany do scrapingu danych

4.1. Preprocessing zebranych zapytań

Początkowy zbiór danych składał się z następujących kolumn:

- full_url
- event_time
- hostname
- injection

Zawierał on **21403** rekordy.

Następnie kolumna **full_url** została zdekodowana z kodowania HTML na utf-8. W związku z faktem, że każdy url może zawierać tylko znaki ASCII, a każdy znak specjalny musi być poprzedzony znakiem ucieczki **%** i odpowiadającą mu wartością dziesiętną w kodzie ASCII, konwersja jest wy-

magana, aby we właściwy sposób przeszukać zebrane dane. Obrobione dane zostały umieszczone w kolumnie **parsed_url**.

Kolejnym krokiem było zliczenie wystąpień słów kluczowych SQL w analizowanym zapytaniu. Ich wystąpienie świadczyłoby o potencjalnym ataku w danym zapytaniu.

Słowa kluczowe zostały uzyskane metodą web-scrapingu strony https://www.w3schools.com/sql/sql_ref_keywords.asp. Uzyskana lista słów kluczowych została następnie wykorzystana do analizy przechwyconych zapisów url. Liczba wystąpień słów kluczowych SQL w zapytaniu została następnie zapisana w kolumnie **sql_keyword**.

W celu zbadania długości przechwyconych zapytań została utworzona kolumna **params_length**. Została ona uzupełniona całkowitą długością przesyłanych parametrów znajdujących po znaku **?**.

Kolejnym elementem poddanym analizie była częstotliwość zapytań generowanych przez hosta. W tym celu przeprowadzona została analiza częstotliwościowa w interwałach **10** sekund. Metryka została wygenerowana za pomocą następującej formuły:

$$\frac{request_volume}{period} \quad (1)$$

Gdzie request_volume to ilość odebranych zapytań, a period to wyszukiwany interwał.

Przedostanią dodaną metryką była metryka **special_chars**. W celu jej utworzenia przesłane zapytania zostały przeanalizowane pod kątem występowania znaków specjalnych z listy:

-, ', ", :, ;, #, / *, *, /, ,, ., (,)

Metryka została wypełniona dla każdego rekordu zgodnie z ilością występujących znaków specjalnych.

Ostatnią dodaną metryką była matryka **values_to_length**, która została wypełniona za pomocą następującej formuły:

$$\frac{sql_keyword + special_chars}{params_length} * 100 \quad (2)$$

Utworzona metryka została zapisana w formie procentowego stosunku sumy znaków specjalnych i słów kluczowych sql do długości zapytania.

Po procesie pre-processingu zbiór danych prezentował się następująco:

- **full_url** - zapisany url z ataku,
- **event_time** - czas zdarzenia,
- **hostname** - ip hosta źródłowego,
- **injection** - czy był to atak,
- **parsed_url** - zdekodowany do postaci czytelnej string url,
- **sql_keyword** - ilość słów kluczowych w zapytaniu,
- **params_length** - długość parametrów w zapytaniu,
- **request_volume** - częstotliwość zapytań w oknie czasowym,
- **special_chars** - ilość znaków specjalnych w zapytaniu,
- **values_to_length** - stosunek sumy znaków specjalnych i słów kluczowych sql do długości zapytania.

4.2. Analiza danych

W pierwszym kroku zostały usunięte wszystkie duplikaty, które mogłyby zaburzyć proces uczenia maszynowego poprzez dodanie sztucznych wag do poszczególnych rekordów. Został również rozwiązany problem uszkodzonych rekordów zawierających wartości Nan. Wstępnie uporządkowany zbiór danych został opisany z wykorzystaniem funkcji **describe()**.

	sql_keyword	params_length	request_volume	special_chars	values_to_length
count	5625.000000	5625.000000	5625.000000	5625.000000	5625.000000
mean	3.388622	77.059733	94.800978	4.422933	12.896231
std	1.879575	56.744467	26.172713	0.996691	5.606127
min	0.000000	1.000000	0.100000	1.000000	0.451807
25%	2.000000	41.000000	93.600000	4.000000	8.823529
50%	4.000000	59.000000	97.800000	5.000000	12.500000
75%	4.000000	101.000000	106.800000	5.000000	15.686275
max	9.000000	664.000000	115.300000	11.000000	100.000000

Rys. 2: Wstępny opis zbioru danych

Zebrane dane zostały następnie przeanalizowane pod kątem rozkładu próbek. W szczególności wzięte pod uwagę zostały wartości znacznie odbiegające od rozkładu normalnego (outliers) oraz korelacja pomiędzy poszczególnymi metrykami.

```

Q1 = sql_records_df.quantile(0.25)
Q3 = sql_records_df.quantile(0.75)
IQR = Q3 - Q1
LB = Q1 - 1.5 * IQR
UB = Q3 + 1.5 * IQR

outliers_LB = (sql_records_df[IQR.index] < LB).sum()
outliers_UB = (sql_records_df[IQR.index] > UB).sum()
out_of_bond = (outliers_LB + outliers_UB) * 100 / len(sql_records_df.index)

outliers = pd.DataFrame({'LB': LB, 'UB': UB, 'LB_outliers': outliers_LB,
                        'UB_outliers': outliers_UB, 'Out_of_bond_percentage' :
                        ↪ out_of_bond})

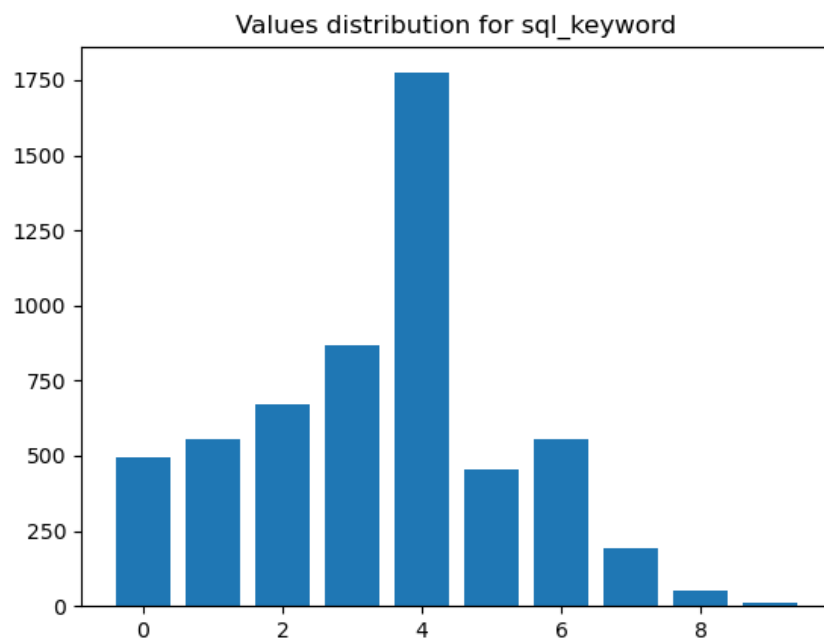
```

Kod 6: Analiza rozkładu wartości dla poszczególnych metryk

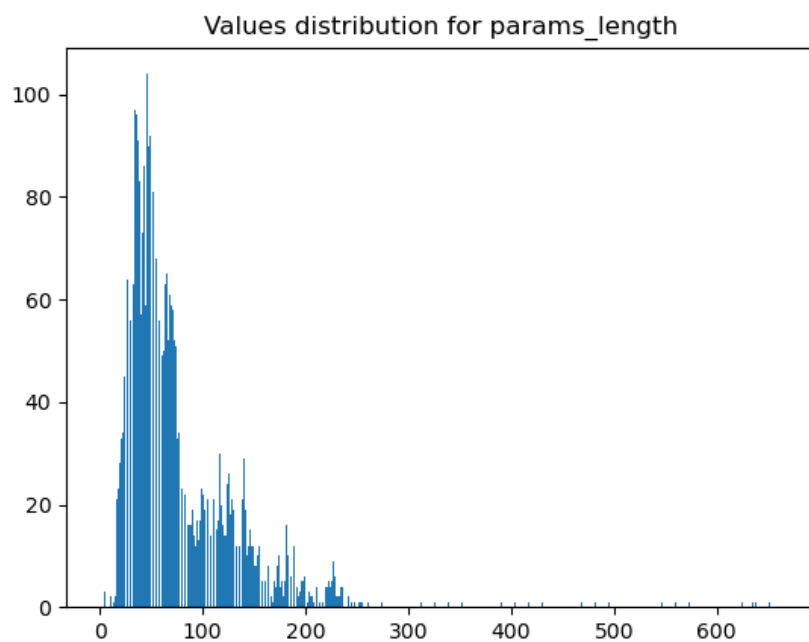
	LB	UB	LB_outliers	UB_outliers	Out of bond percentage
sql_keyword	-1.000000	7.000000	0	62	1.102222
params_length	-49.000000	191.000000	0	196	3.484444
request_volume	73.800000	126.600000	518	0	9.208889
special_chars	2.500000	6.500000	198	24	3.946667
values_to_length	-1.470588	25.980392	0	120	2.133333

Rys. 3: Wynik analiza rozkładu wartości dla poszczególnych metryk

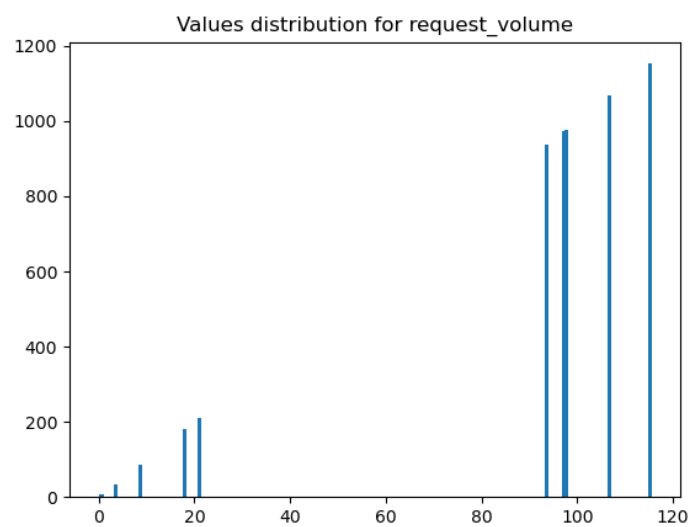
Wyniki otrzymane dla parametru **request_volume** okazały się być dość niepokojące. Analiza danych wykazała duży odsetek danych leżących w **LB_outlier**. W celu bardziej dokładnego zbadania analizowanych danych stworzone zostały wykresy rozkładu wartości dla poszczególnych metryk.



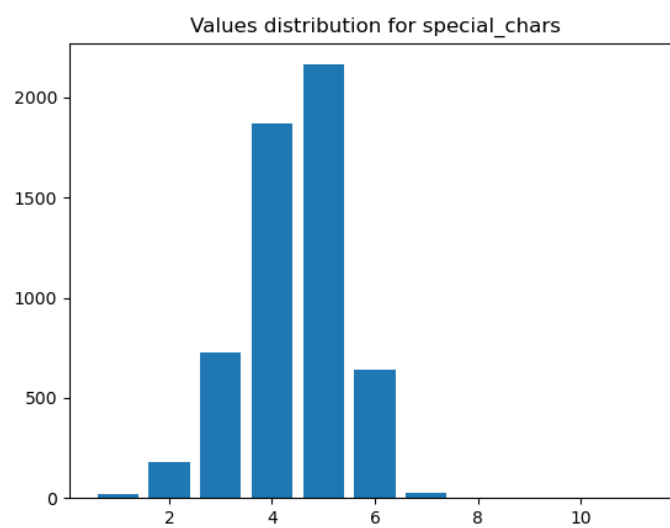
Rys. 4: Rozkład wartości dla metryki sql_keyword



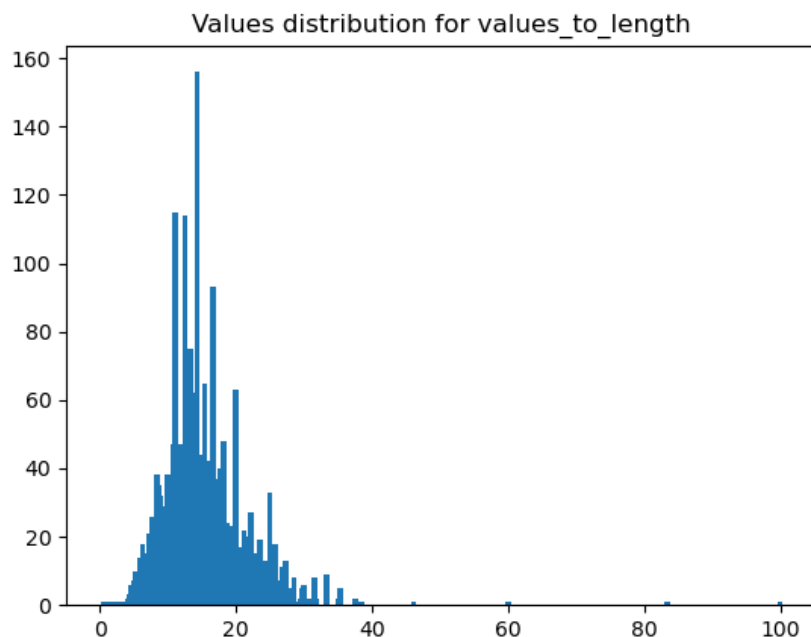
Rys. 5: Rozkład wartości dla metryki params_length



Rys. 6: Rozkład wartości dla metryki request_volume



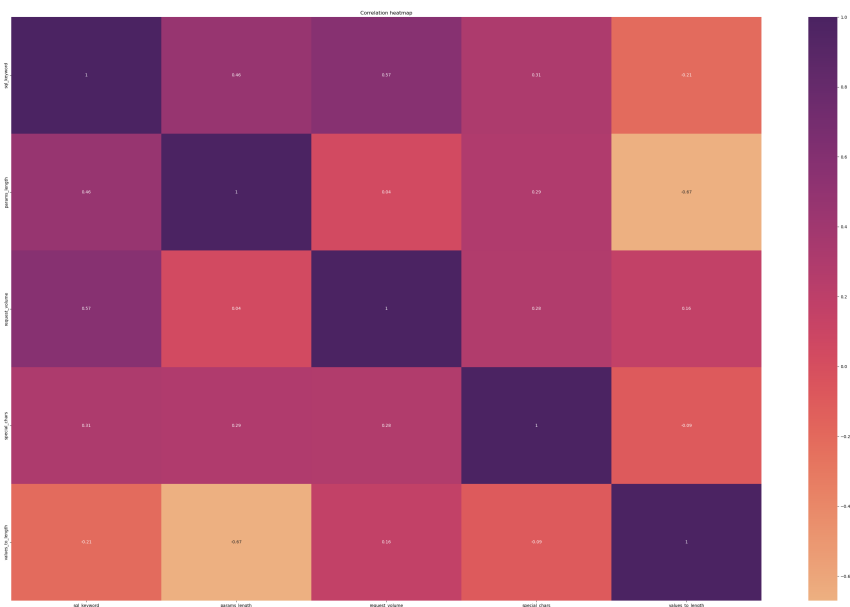
Rys. 7: Rozkład wartości dla metryki special_chars



Rys. 8: Rozkład wartości dla metryki values_to_length

Ponownie analiza wykazała nietopowy rozkład wartości dla metryki **request_volume** zaprezentowany na **Rysunku 6**. Należy zaznaczyć, że choć otrzymany rozkład jest nietopowy w kontekście pozostałych metryk, nie jest on wynikiem błędu. Rozkład ten wynika z charakterystyki niektórych z ataków SQLi. Ataki czasowe wiążą się z koniecznością dłuższego czasu oczekiwania na odpowiedź serwera, co zmniejsza ilość zapytań wysyłanych na minutę. Zatem zarejestrowane wartości znacznie odbiegające od normy są niezwykle istotne, gdyż pozwolą modelowi uczącemu na wykrywanie wcześniej wymienionych ataków.

Ostatnim przeprowadzonym elementem analizy danych było wykrycie korelacji pomiędzy metrykami. Zbyt wysoka lub niska wartość tego parametru oznaczałaby, że model uczący opiera się na metrykach nie mających wpływu na końcową efektywność predykcji.



Rys. 9: Macierz korelacji

Żadna z uzyskanych wartości nie przekroczyła progu 0.7, gdzie najwyższa korelacja miała wartość 0.67. Można zatem stwierdzić, że, choć dostrzegalne są pewne zależności między metrykami, to nie są one na tyle wysokie, by wykluczyć którąś z nich w procesie uczenia maszynowego modelu.

5. Przeprowadzenie uczenia maszynowego

Proces uczenia maszynowego został zrealizowany z użyciem algorytmu **N Nearest Neighbors**. Jest to algorytm opierający się na modelu głosowania bezwzględnego. Model na podstawie n najbliższych sąsiadów od analizowanego rekordu przeprowadza głosowanie. Na podstawie głosowania podejmowana jest decyzja, jaka klasa zostanie przyporządkowana badanemu rekordowi.

W celu zwiększenia efektywności uczenia została wykorzystana metoda **k-fold validation**. Polega ona na podziale zbioru danych na k podzbiorów. Jeden z podzbiorów zostanie wykorzystany jako zbiór testowy, a pozostałe jako trenujące. Proces podziału zostaje wykonany k krotnie. Procedura ta ma zapewnić wykrycie przetrenowania modelu i weryfikację, czy wykorzystywany zbiór jest na tyle różnorodny, by zapewnić wysoką efektywność predykcji bez względu na podany zbiór testujący.

Przygotowany został skrypt mający pozwolić na ustalenie najbardziej optymalnych wartości parametrów n i k , które zostaną użyte w finalnej wersji uczenia modelu.

```
round = 1
ml_results_columns = ["folds", "neighbors", "accuracy"]
ml_results = pd.DataFrame(columns=ml_results_columns)

for folds in range(2,11):
    kfold = KFold(n_splits=folds, shuffle=True, random_state=1)
    for train, test in kfold.split(np_dataset):
        X_train = np_dataset[train][:, :-1]
        y_train = np_dataset[train][:, -1]
        X_test = np_dataset[test][:, :-1]
        y_test = np_dataset[test][:, -1]
        neighbors = range(2,6)
        for i in neighbors:
            knn = KNeighborsClassifier(n_neighbors=i)
            knn.fit(X_train, y_train)
            y_pred = knn.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
            row = pd.DataFrame([[folds, i, accuracy]],
                               ↪ columns=ml_results_columns)
            ml_results = pd.concat([ml_results, row], ignore_index=True)
        round = round + 1
```

```
tmp_columns = ["folds", "neighbors", "accuracy"]
tmp_df = pd.DataFrame(columns=tmp_columns)
for folds in range(2,11):
    for i in range(2,6):
        tmp = ml_results.loc[(ml_results['neighbors'] == i) &
            ↪ (ml_results['folds'] == folds)]
        tmp = tmp.sum()['accuracy']/len(tmp)
        row = pd.DataFrame([[folds, i, tmp]], columns=tmp_columns)
        tmp_df = pd.concat([tmp_df, row], ignore_index=True)
```

Kod 7: Procedura określenia parametrów n i k

```
Accuracies comparison
For 2 folds --- 3 neighbors ---- 0.9990214732119579 accuracy
For 3 folds --- 3 neighbors ---- 0.998369130874713 accuracy
For 4 folds --- 3 neighbors ---- 0.9993475778308033 accuracy
For 5 folds --- 3 neighbors ---- 0.99836867862969 accuracy
For 6 folds --- 3 neighbors ---- 0.998695208833671 accuracy
For 7 folds --- 3 neighbors ---- 0.9988582611126642 accuracy
For 8 folds --- 3 neighbors ---- 0.99918428586504 accuracy
For 9 folds --- 3 neighbors ---- 0.9988578887257302 accuracy
For 10 folds --- 3 neighbors ---- 0.9986949429037522 accuracy
Max accuracy
For 4 folds --- 3 neighbors ---- 0.9993475778308033 accuracy
```

Rys. 10: Znalezienie paramentów zapewniających największą dokładność

Po weryfikacji dla jakiej liczby sąsiadów i w jakich warunkach dokładność jest najwyższa okazało się, że zachodzi to dla **4 podgrup** i **3 sąsiadów**.

Dla wyliczonych parametrów został przeprowadzony proces uczenia maszynowego modelu.

```
test_vector = []
predicted_vector = []

#init cross validation
kfold = KFold(n_splits=4, shuffle=True, random_state=1)
#split into folds
for train, test in kfold.split(np_dataset):
    X_train = np_dataset[train][:, :-1]
    y_train = np_dataset[train][:, -1]

    X_test = np_dataset[test][:, :-1]
    y_test = np_dataset[test][:, -1]

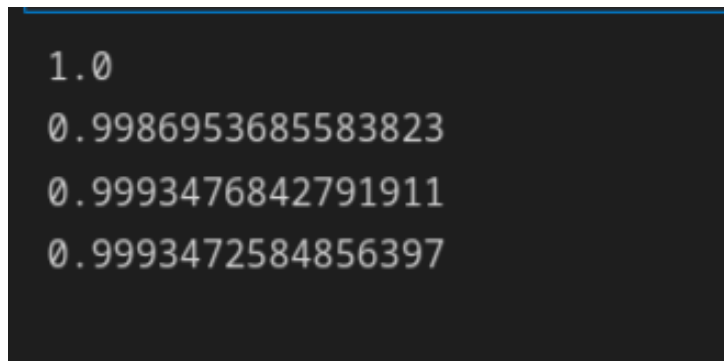
    # create model and train it
    knn = KNeighborsClassifier(n_neighbors=3)
    knn.fit(X_train, y_train)

    # Test results
    y_pred = knn.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(accuracy)

    test_vector = [*test_vector, *y_test]
    predicted_vector = [*predicted_vector, *y_pred]
```

Kod 8: Proces uczenia maszynowego

Przeprowadzone testy dały następujące rezultaty odnośnie skuteczności modelu:

A terminal window with a dark background and light gray text. It displays four lines of numerical values representing model accuracy. The first line is '1.0', the second is '0.9986953685583823', the third is '0.9993476842791911', and the fourth is '0.9993472584856397'.

```
1.0
0.9986953685583823
0.9993476842791911
0.9993472584856397
```

Rys. 11: Rezultaty uczenia maszynowego dla optymalnych parametrów

6. Wyniki

Dla uzyskanych wyników uczenia maszynowego wykorzystane zostały dodatkowe metryki oceny jakości:

- **strata logarytmiczna** - wskazuje, jak blisko jest prawdopodobieństwo predykcji w stosunku do odpowiedniej wartości rzeczywistej/prawdziwej. Im bardziej przewidywane prawdopodobieństwo odbiega od rzeczywistej wartości, tym wyższa jest wartość wskaźnika.
- **macierz pomyłek** - wizualizuje i podsumowuje wydajność algorytmu klasyfikacji (zawiera wartości TP, TN, FP, FN).
- **obszar pod krzywą** - znajduje obszar kształtów, które nie są poprawnie zdefiniowane.
- **wskaźnik wyników prawdziwie pozytywnych, prawdziwie negatywnych, fałszywie pozytywnych oraz fałszywie negatywnych** - wykorzystywane w macierzy pomyłek
- **wynik f1** - średnia harmoniczna precyzji i wycofania, wskaźnik oceny uczenia maszynowego, który mierzy dokładność modelu. Metryka dokładności oblicza, ile razy model dokonał poprawnej prognozy w całym zbiorze danych.
- **średni błąd bezwzględny** - mierzy średnią wielkość błędów w zestawie prognoz, bez uwzględniania ich kierunku.
- **błąd średniokwadratowy** - obliczany jest za pomocą średniej błędów podniesionych do kwadratu z danych w odniesieniu do funkcji. Większa wartość wskazuje, że punkty danych są szeroko rozproszone wokół momentu centralnego (średniej), podczas gdy mniejsza wartość - przeciwnie.

	Metric	Value
1	Accuracy	0.999347
2	Logarithmic Loss	0.0015111
3	Confusion Matrix Accuracy	0.999347
4	Area Under Curve	0.999989
5	F1 Score	0.999642
6	Mean Absolute Error	0.000652742
7	Mean Squared Error	0.000652742

Rys. 12: Podsumowanie wyników w ramach obliczanych metryk

Na podstawie wyliczonych wartości z otrzymanych wektorów testowych oraz przewidywanych, można stwierdzić, że poziom błędów jest niewielki (zbliżony do wartości 0,01 %), a dokładność modelu jest zbliżona do 99.9 %.

7. Wnioski

Przeprowadzony w ramach projektu proces uczenia maszynowego pozwolił na utworzenie dość dokładnego modelu detekcji ataku jakim jest **SQL Injection**. Ze względu na powszechność wykorzystania zautomatyzowanych skanerów w atakach SQLi otrzymane próbki są dość zbliżone do tych, które można uzyskać analizując rzeczywisty atak. Odpowiednia analiza i preprocessing danych pozwoliły na utworzenie dobrego zbioru do przeprowadzania uczenia maszynowego. W rezultacie utworzony model działa z dość dużą dokładnością, bliską 99.9%. Wysoka dokładność może wynikać z dość prostego problemu, jakim jest atak SQLi, ze względu na konieczność występowania słów kluczowych, a przekazywane w nim parametry często znacząco odbiegają od charakterystycznych dla prawidłowego ruchu.

Równie skuteczna co wyuczony model detekcja SQLi, mogłaby zostać zrealizowana poprzez wprowadzenie kilku reguł do silnika detekcji takiego jako Snort. Wprowadzenie reguł oferowałoby detekcję o zbliżonym poziomie skuteczności przy znacznie mniejszym nakładzie środków.

Wykorzystanie uczenia maszynowego w detekcji SQLi ma też swoją wadę. Model wykrywa atak na podstawie analizy cech charakterystycznych. Złośliwe zapytanie można skonstruować w sposób upodabniający je do poprawnego zapytania. Ze względu na swoją unikalność, model nie będzie posiadał podobnych zapytań w swojej bazie przykładów uczących i nie będzie potrafił poprawnie zidentyfikować ataku.