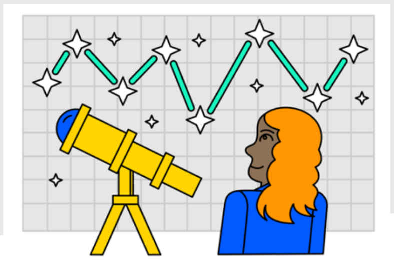
Threat Hunting Project: Detecting Anomalies in Computer Network Communication



Yovel Hadari | BSc Information Systems and Cybersecurity Student and graduate of Hakeriot – Mentoriot program | February 2025

Introduction

This project focuses on analyzing a dataset of network traffic logs and packets to identify anomalies and potential threats. The primary goal is to detect and characterize Denial of Service (DoS) attacks within the network traffic data. The analysis utilizes Python libraries such as pandas, numpy, matplotlib, scipy, and seaborn for data manipulation, statistical analysis, and visualization.

About the Project

This project was conducted as an independent project within the Mentoriot program in Hakeriot. Mentoriot is a cybersecurity program that focuses on providing students with practical experience in various areas of cybersecurity. Mentoriot, as part of Hakeriot, is designed to empowering women to enter the cybersecurity workforce.

Project Objectives

- Develop a network traffic analysis framework to detect anomalies and potential threats.
- Identify behavioral patterns indicative of cyberattacks.
- Design a flexible detection algorithm capable of analyzing CSV-based network data.
- Generate insights and correlations to enhance threat intelligence.
- Refine detection mechanisms using real-world attack datasets to improve accuracy.

Data and Methodology

The dataset used for this analysis is the "DAPT 2020" dataset[1], which contains network traffic data collected over five days, with each day representing three months in a real-world scenario. The dataset is designed to help researchers understand anomalies and identify Advanced Persistent Threat (APT) attacks in their early stages.

The analysis focuses on detecting three specific types of DoS attacks: SYN Flood, ACK Flood, and Reflection Attack. The methodology involves the following steps:

1. Data Aggregation: Grouping data by destination IP and 5-minute time slots to identify patterns and trends.
2. Outlier Detection: Using statistical methods like the Interquartile Range (IQR) and Z-score to identify outliers in various network traffic features.
3. Attack Classification: Developing rules and thresholds to classify potential DoS attacks based on the identified outliers and patterns.
4. Visualization: Creating various plots and graphs to visualize the network traffic data, anomalies, and attack patterns.

Denial of Service (DoS) Attacks

Denial of Service (DoS) attacks are a type of cyberattack where the attacker attempts to disrupt or disable services or resources by overwhelming them with traffic. These attacks can originate from a single source or multiple sources (DDoS - Distributed Denial of Service). In the context of APT attacks, DoS attacks can be used as a distraction technique to divert the attention of security teams while the attacker executes a deeper intrusion.

When traffic passes through a proxy server, all addresses converge to a single address, making it appear as if the traffic originates from the same IP address, regardless of whether it is a DoS or DDoS attack. This makes it difficult to distinguish between the two types of attacks from an external perspective.

SYN Flood Attack

A SYN Flood attack is a type of DoS attack that exploits the TCP handshake process. The attacker sends a large number of SYN packets to initiate TCP connections but does not complete the handshake by sending the corresponding ACK packets. This behavior causes

[1] https://www.kaggle.com/datasets/sowmyamyneni/dapt2020

the server to exhaust its resources waiting for the completion of the handshake, leading to service disruption.

Code Implementation

The code for this project is implemented in Python using Jupyter Notebook. It utilizes various libraries such as Pandas, Matplotlib, Seaborn, Numpy, and Scipy for data analysis and visualization. The code includes functions for data aggregation, outlier detection, attack classification, and visualization.

I will present some of the functions I created, and details about the rest can be found in the Readme.md file attached to the Git repository. It's worth noting that the code details I'll present are partial descriptions, meaning that if the explanation is about the function as a whole, the example will be of a part of it for demonstration purposes. The implementation is in Jupyter Notebook.

Data structures used:

- Dictionaries (`dict`) for attack classification and storage
- Lists (`list`) for handling multiple source IPs, ports, and protocols
- DataFrames (Pandas) for efficient tabular data processing

Data Grouping by Destination and Time Slots

A function was implemented to group the data by destination IP and time slots. This function takes a DataFrame (`df`) and groups it by `Dst IP`, dividing the time into 5-minute slots. A key was added to link further aggregations to it. In the aggregation, I included summaries of all the features that I knew would be essential and relevant for finding anomalies according to the definition of each of the three attacks described:

```
def group_data(df):
    # Merge the summarized data and detailed data using the 'group_id' key.
    # The 'group_id' serves as the unique identifier connecting the
summarized statistics
    # with the detailed information about the sources (Src IP, Src Port,
Protocol):
    df['group_DosDetect_id'] = df.groupby(['Dst IP',
pd.Grouper(freq='5min')]).ngroup()
    # - 'grouped_summary': Contains aggregate statistics (e.g., counts,
averages) per Dst IP and time interval.
    grouped_summary = df.groupby(['group_DosDetect_id', 'Dst IP',
pd.Grouper(freq='5min')]).agg(
        # Aggregate summarized statistics per group
        CountRequests=('Src IP', 'count'),
        CountSrc_uniq=('Src IP', 'nunique'),  # Count unique source IPs
        Flow_Packets_s_avg=('Flow Packets/s', 'mean'),
        Flow_Bytes_s_avg=('Flow Bytes/s', 'mean'),
        SYN_count_sum=('SYN Flag Count', 'sum'),
        ACK_count_sum=('ACK Flag Count', 'sum'),
        Bwd_sum=('Total Bwd packets', 'sum'),
        Fwd_sum=('Total Fwd Packet', 'sum'),
    ).reset_index()

    # Calculating ratios separately to avoid referencing within the
aggregation function
    grouped_summary['SYN_ACK_Ratio'] = grouped_summary['SYN_count_sum']
/((grouped_summary['ACK_count_sum']) + 1)
    grouped_summary['ACK_SYN_Ratio'] = grouped_summary['ACK_count_sum'] /
(grouped_summary['SYN_count_sum'] + 1)
    grouped_summary['Bwd_Fwd_Ratio'] = grouped_summary['Bwd_sum'] /
(grouped_summary['Fwd_sum'] + 1)

    # Format 'HourTime' as a datetime column for sorting
    grouped_summary['HourTime'] =
grouped_summary['Datetime'].dt.floor('30min').dt.strftime('%H:%M')
```

I created a key identifier for them. These columns are later sent for Thresholds examination, which will be detailed later in this report. You can see that I created a column to examine the correlation between syn flags to ack flags, and BWD and FWD packet movement to consider them in anomaly characteristics according to the patterns characterizing each attack accordingly.

In addition, I created an aggregation linked to the same key, for source `(Src)` data, the port it used, and the protocol. This detail is relevant to the research, and this implementation is more correct from a relational perspective:

```
grouped_src_details = df.groupby('group_DosDetect_id').apply(
    lambda group: pd.Series({
        'Src Details': group[
            ['Src IP', 'Src Port', 'SrcPort_categorical', 'Protocol',
'Protocol_categorical']].to_dict('records'),
        'SrcIP_uniq': group['Src IP'].unique().tolist(),
        'SrcPort_uniq': group['Src Port'].unique().tolist(),
        'SrcPort_categorical':
group['SrcPort_categorical'].unique().tolist(),
        'Protocol_uniq': group['Protocol'].unique().tolist(),
        'Protocol_categorical':
group['Protocol_categorical'].unique().tolist(),

        # count of port types
        'Well_Known_Port_Count': sum(group['SrcPort_categorical'] == 'Well-
Known Ports'),
        'Registered_Port_Count': sum(group['SrcPort_categorical'] ==
'Registered Ports'),
        'Dynamic_Private_Port_Count': sum(group['SrcPort_categorical'] ==
'Dynamic/Private Ports'),

        # count of protocol types
        'Protocol_TCP_Count': sum(group['Protocol_categorical'] == 'TCP'),
        'Protocol_UDP_Count': sum(group['Protocol_categorical'] == 'UDP'),
        'Protocol_ICMP_Count': sum(group['Protocol_categorical'] == 'ICMP'),
        'Other_Protocol_Count': sum(group['Protocol_categorical'] ==
'Other'),
        })
).reset_index()
```

- `Src Details`: Contains dictionaries with source IP address, source port, and protocol for each group to check their set in each interaction.
- `SrcPort_uniq`: Unique ports within the group.
- `SrcPort_categorical`: Groups source ports into predefined categories (Well-Known, Registered, Dynamic/Private).
- `Protocol_uniq`: Unique protocol numbers within the group.
- `Protocol_categorical`: Maps protocol numbers to human-readable names.
- `Well_Known_Port_Count/Registered_Port_Count/Dynamic_Private_Port_Count`: Count of ports per group type.
- `Protocol_TCP_Count/Protocol_UDP_Count/Protocol_ICMP_Count/Other_Protocol_Count`: Count of protocols per group type.

Function for Creating Threshold and Examining Outliers

I sent all those columns to a function that checks their Thresholds statistically. I chose the method that uses the interquartile range. In the process, I saw that there was communication with an unusual number of requests, specifically 5 requests `(CountRequests)`, while there is also data with around 1000 requests. As mentioned, both do not represent the same benchmark, and upon deeper investigation, I saw that most values for this metric do not exceed the hundreds, and there is indeed a significant jump to the thousands. Therefore, I added a value aimed at normalization. To do this, I used the same method for examining Outliers, but it worked on the Zscore normalization method, which allows creating an identical benchmark for values or units of measure or different distributions by converting them to a common scale. This actually happens by examining their distance from the average in units of standard deviations, and allows determining the degree of "outlier" of a certain value comparable. It should be noted that attention should be paid to defining the threshold when examining times, for example, if there are jumps in defining the "normal" data, as this is also relevant to the client's settings. Take, for example, a company server that does not provide service on holidays; we will likely see a significant decrease in its communication behavior during these times. In our case, I did not address this but created a general threshold for all the data.

```python
def detect_zscore_outliers_iqr(df, column):
    """Detects outliers for a threshold in a DataFrame column using the IQR
method."""
    # Using ZScore to normalize the data - considering more robust or
tailored approaches to detect outliers.
    zscore = stats.zscore(df[column])
    df[f'Normalized_{column}'] = zscore
    iqr = 0.5
    lower_bound = 0.25 - 1.5 * iqr
    upper_bound = 0.75 + 1.5 * iqr
    outlier_dict = {
        "dict_name": "outlier_dict",
        "column": column,
        "zscore": zscore,
        "lower_bound": lower_bound,
        "upper_bound": upper_bound,
        "iqr": iqr,
        "std": df[column].std()
    }
    return outlier_dict


def detect_outliers_iqr(df, column):
    # Normalized Outlier
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    iqr = Q3 - Q1
    lower_bound = Q1 - 1.5 * iqr
    upper_bound = Q3 + 1.5 * iqr
    outlier_dict = {
        "dict_name": "outlier_dict",
        # "df_pointer": point_to_df,
        "column": column,
        "lower_bound": lower_bound,
        "upper_bound": upper_bound,
        "iqr": iqr,
        "std": df[column].std()
    }
    return outlier_dict
```

Port Type Translation

I created a function to examine the port type and return a translation for its type:

```
def categorize_ports(port):
    """
    Categorize ports into Well-Known Ports, Registered Ports, and
Dynamic/Private Ports.
    - Well-Known Ports: 0-1023
    - Registered Ports: 1024-49151
    - Dynamic/Private Ports: 49152-65535
    """
    if port < 1024:
        return 'Well-Known Ports'
    elif 1024 <= port <= 49151:
        return 'Registered Ports'
    else:
        return 'Dynamic/Private Ports'
    # Define protocol categories
```

Protocol Type Translation

I created global variables for mapping protocol types and a get function for implementing the translation:

```
# Global dictionary for protocol mappings
PROTOCOL_MAPPING = {
    1: 'ICMP',  # Internet Control Message Protocol
    6: 'TCP',  # Transmission Control Protocol
    17: 'UDP',  # User Datagram Protocol
    50: 'ESP',  # Encapsulating Security Payload
    51: 'AH',  # Authentication Header
    8: 'EGP',  # Exterior Gateway Protocol
    # Add other protocol mappings as needed
}

def get_protocol_name(protocol_number):
    # Function to get the protocol name
    """Return the human-readable name for a given protocol number."""
    return PROTOCOL_MAPPING.get(protocol_number, 'Other')
```

Suspect Attack Examination

A function was created to examine and characterize the existence of a suspected attack. If an attack is suspected, the entire aggregated row created in the grouping function is added to a dictionary under a value that will be returned according to the key in the function's input. This is done along with the type of attack and the relevant data ratio for its characterization. In the case of SYN Flood, the ratio is the SYN-ACK ratio. There is also consideration for "Normal Data" that can be added at the client's discretion. If the set of conditions that create the definition of a suspected attack are met, there will be an additional examination of each of them against its threshold in the "normal" data. It was defined that if an upper outlier exists in one of those features, there is a suspicion of an attack. This situation may be considered strict or lenient depending on the existing client;

this should be taken into account and adjusted accordingly in this client-oriented perspective. Here is a breakdown of only the condition for creating a SYN Flood attack:

```
def classify_attack(row, df_filename, dict_dos_attacks, key,
normal_data=None):
    # SYN Flood Attack
    if row['Outlier_Normalized_SYN_count_sum_U/L'] == "upper_bound" and \
            row['Outlier_SYN_ACK_Ratio_U/L'] == "upper_bound" and \
            row['Outlier_Flow_Packets_s_avg_U/L'] == "upper_bound":
        flag = True

        if not normal_data.empty:
            columns_to_check = ['Normalized_SYN_count_sum',
'Normalized_SYN_ACK_Ratio', 'Flow_Packets_s_avg']
            flag = False
            for column in columns_to_check:
                # Adjusted condition to check against normal thresholds
                if "Normalized" in column:
                    if row[column] > detect_zscore_outliers_iqr(normal_data,
column)['upper_bound']:
                        flag = True
                        break
                else:
                    if row[column] > detect_outliers_iqr(normal_data,
column)['upper_bound']:
                        flag = False
                        break  # Assuming any single outlier condition
triggers the classification
        if flag:
            attack_type = 'SYN_Flood'
            column = 'SYN_ACK_Ratio'
            Ratio = row['SYN_ACK_Ratio']
    # Inserts to Dict of Dos
    if attack_type!= 'Normal':
        dict_dos_attacks[key] = {
            "Attack_id": f"{key}-{row.group_DosDetect_id}",
            "group_DosDetect_id": row.group_DosDetect_id,
            "Source_dfName": df_filename,
            "attack_type": attack_type,
            f"{column}": Ratio,
            "Dst IP": row['Dst IP'],
            "CountRequests": row['CountRequests'],
            "CountSrc_uniq": row['CountSrc_uniq'],  # Count unique source IPs
            "Datetime": row['Datetime'],
            "HourTime": row['HourTime'],
            # Calculate statistics for SYN Flag Count
            "SYN_count_sum": row.SYN_count_sum,
            # Calculate statistics for ACK Flag Count
            "ACK_count_sum": row.ACK_count_sum,
            # Calculate statistics for Fwd Packet Flag Count
            "Fwd_sum": row.Fwd_sum,
            # Calculate statistics for Bwd Packet Flag Count
            "Bwd_sum": row.Bwd_sum,
            # Avg Flow Packets/ Bytes
            "Flow_Packets_s_avg": row.Flow_Packets_s_avg,
            "Flow_Bytes_s_avg": row.Flow_Bytes_s_avg,
            # Source details
            "Src Details": row['Src Details'],
            "SrcIP_uniq": row['SrcIP_uniq'],
            "SrcPort_uniq": row.SrcPort_uniq,
            "SrcPort_categorical": row.SrcPort_categorical,
            "Protocol_uniq": row.Protocol_uniq,
            "Protocol_categorical": row.Protocol_categorical,
            # Port categorial count
            'Well_Known_Port_Count': row.Well_Known_Port_Count,
            'Registered_Port_Count': row.Registered_Port_Count,
            'Dynamic_Private_Port_Count': row.Dynamic_Private_Port_Count,
            # Protocol categorial count
            'Protocol_TCP_Count': row.Protocol_TCP_Count,
            'Protocol_UDP_Count': row.Protocol_UDP_Count,
            'Protocol_ICMP_Count': row.Protocol_TCP_Count,
            'Other_Protocol_Count': row.Other_Protocol_Count,
        }
```

Analysis: Function for Creating a Graph to Examine Outliers

This shows a graph of the column data in the DataFrame that we want to examine, and a horizontal line of the median, upper bound, and lower bound data. You can also display a check for other data such as ranges with standard deviation, etc. (marked as comments in the code). If the user chooses to examine the data against the normal data or against an attack from the dictionary, or separately from them, they can do so. This function can also be implemented in the following function using a boxplot in the `boxGraphs_for_outliers()` func.

```
def graph_for_outlierCol(df, outlier_dict, normData=None,
outlierNormal_dict=None, attack=None):
    column = outlier_dict['column']
    # Ensure the column exists in the DataFrame
    if column in df.columns:
        # Ensure 'HourTime' is properly formatted as datetime for sorting and
plotting
        if not pd.api.types.is_datetime64_any_dtype(df['HourTime']):
            df['HourTime'] =
pd.to_datetime(df['Datetime']).dt.floor('30min').dt.strftime('%H:%M')
        # Sort grouped data by 'Dst IP' and 'HourTime' to ensure the correct
order
        df = df.sort_values(by='Dst IP')
        df = df.sort_values(by='HourTime')
        # Calculate the necessary statistics for the column
        col_mid = df[column].median()
        # col_std = df[column].std()
        lower_bound = outlier_dict['lower_bound']
        upper_bound = outlier_dict['upper_bound']
        # Plot the data with horizontal lines for thresholds
        plt.figure(figsize=(18, 10))
        sns.lineplot(data=df, x='HourTime', y=column, marker='o', color='y',
label='Analysed Data')
        # Add horizontal lines to indicate statistical thresholds
        plt.axhline(y=col_mid, color='blue', linestyle='dashed', linewidth=2,
label='Median')
        # plt.axhline(y=col_mid - col_std, color='sky blue',
linestyle='dashed', linewidth=2, label='-1 STD')
        # plt.axhline(y=col_mid + col_std, color='sky blue',
linestyle='dashed', linewidth=2,
```

Code Considerations

The code is designed to provide a general approach to network traffic analysis and DoS attack detection. However, it requires adjustments and customization depending on the specific characteristics of the data set being analyzed. Different data sets may have different features and structures, especially those captured through sniffing, which may include additional information or vary depending on the server and sniffing methods used.

Suspected DoS Attack

During the execution of the code on the DAPT 2020 dataset, a suspected DoS attack of the SYN Flood type was identified. This diagnosis can be further supported by analyzing the output data, including graphs, tables, and dictionaries. The analysis focuses on identifying the patterns that characterize the attack and the insights derived from it.

Relational Approach

A key identifier was created, and an aggregation linked to the same identifier was generated for source (Src) data, including the port and protocol used. This detailed information is relevant to the research, and this implementation is more appropriate from a relational perspective.

Data Normalization

During the analysis, it was observed that there is communication with a considered unusual number of requests, specifically 5 requests (CountRequests), while there is also data with around 1000 requests. As mentioned, both do not represent the same benchmark, and upon deeper investigation, it was found that most values for this metric do not exceed the hundreds, and there is indeed a significant jump to the thousands. Therefore, a value aimed at normalization was added.

Translation for Human Readability

Protocols and ports were translated to their human-readable names to improve the interpretability of the analysis.

Suspect Attack Examination

A function was created to examine and characterize the existence of a suspected attack. If an attack is suspected, the entire aggregated row created in the grouping function is added to a dictionary under a value that will be returned according to the key in the function's input. This is done along with the type of attack and the relevant data ratio for its characterization. In the case of SYN Flood, the ratio is the SYN-ACK ratio. There is also consideration for "Normal Data" that can be added at the client's discretion. If the set of conditions that create the definition of a suspected attack are met, there will be an additional examination of each of them against its threshold in the "normal" data. It was defined that if an upper outlier exists in one of those features, there is a suspicion of an attack. This situation may be considered strict or lenient depending on the existing client; this should be taken into account and adjusted accordingly in this client-oriented perspective.

Normal Data Reference

The analysis also includes a reference to data defined as "normal" to provide a baseline for comparison and identify deviations that may indicate an attack.

Attack Classification

Attack classification involves defining rules and thresholds to classify potential DoS attacks based on the identified outliers and patterns. The code includes conditions to identify specific attack types like SYN Flood based on the characteristics of the network traffic.

Visualization

Visualization plays a crucial role in understanding the network traffic data and identifying anomalies. The code includes functions to create various plots and graphs, such as line plots, scatter plots, and box plots, to visualize the network traffic data, anomalies, and attack patterns.

Results

The analysis revealed a suspected SYN Flood attack within the dataset. The attack is characterized by a high SYN to ACK ratio, indicating incomplete handshake patterns. The attack traffic shows a significant spike in SYN packets at 15:15, deviating from normal levels.

```
SYN_Flood: 1
Reflection_Attack: 0
ACK_Flood: 0
```

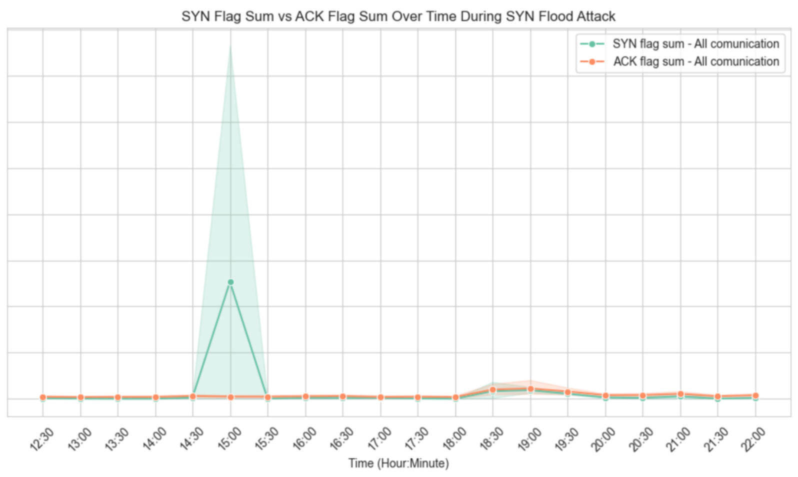
CountRequests	Syn_count_sum	ACK_count_sum
1261	1260	1

Key Observations:

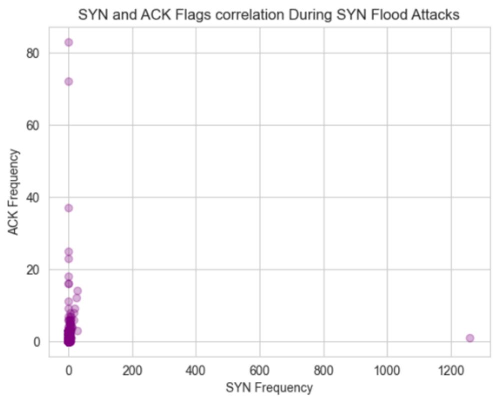
- High SYN to ACK ratio, indicating an incomplete handshake pattern.
- Normal traffic exhibits balanced SYN-ACK pairs.
- Attack traffic shows unmatched SYN packets.
- Unusual spike detected in SYN packet count at 15:15.
- Compared with normal traffic data, SYN count above 1200 sum, significantly deviating from normal levels.

Analysis of SYN Flood Attack:

- There is an unusual point of SYN flag above 1200, while the rest of the SYN and ACK flag data are close to the x-axis, with another relative increase, especially at 19:00. This is characteristic of a SYN attack.



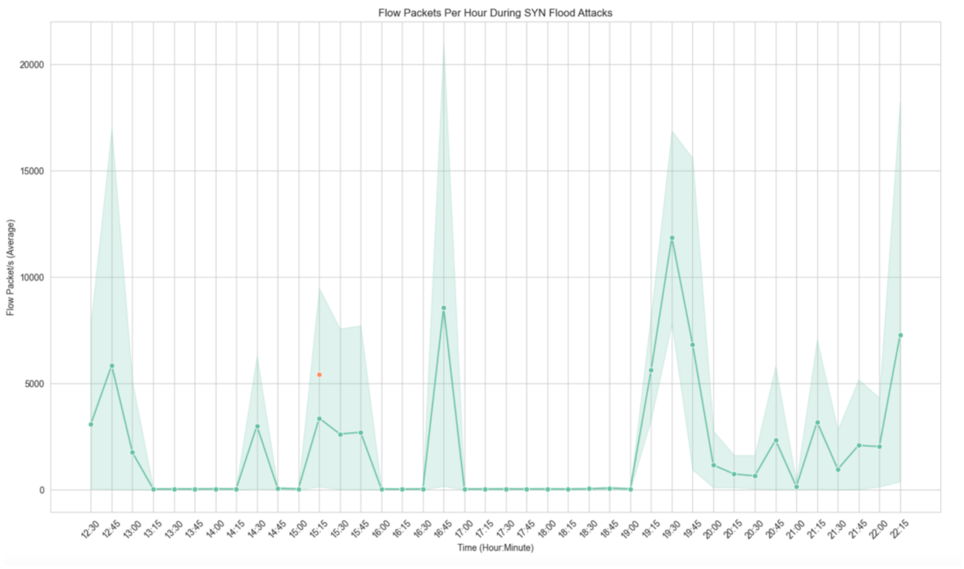
- The scatter plot shows no correlation between SYN and ACK flags in this communication. The SYN values range widely, while ACK values range lower, with a large gap from those of SYN. The wide range of SYN includes an outlier around 1200 that is not accompanied by a similar increase in the ACK value, suggesting a SYN Flood, where an attacker sends large quantities of SYN requests but does not complete the TCP Handshake process, with no significant increase in ACK occurrences.

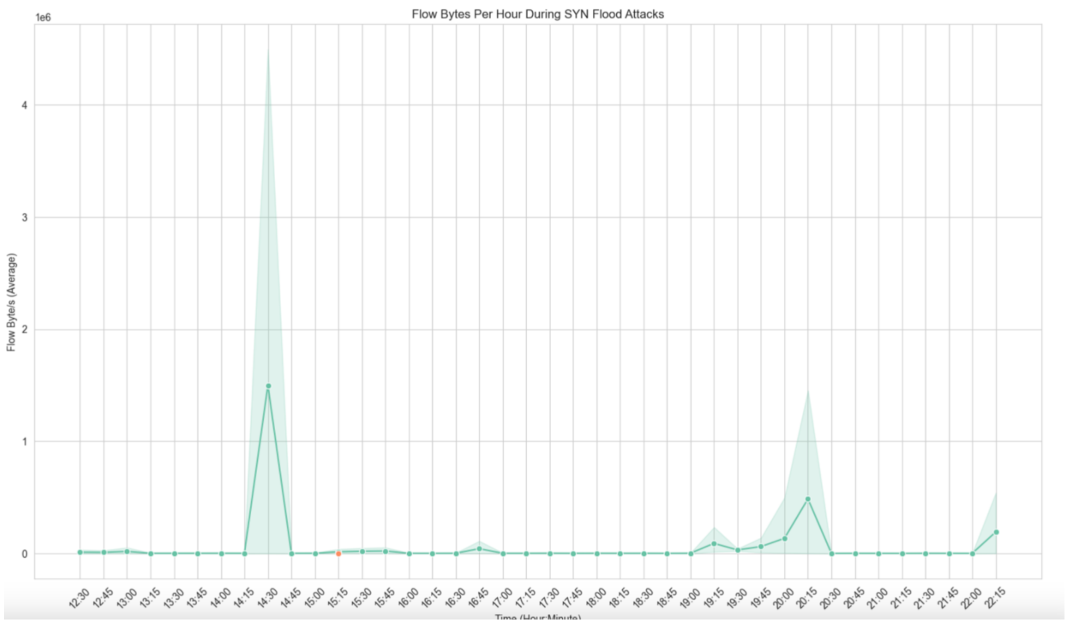


Analysis of Packet to Byte flow:

- High Packet to Byte flow, indicating a lot of Syn Packets but no Payload.

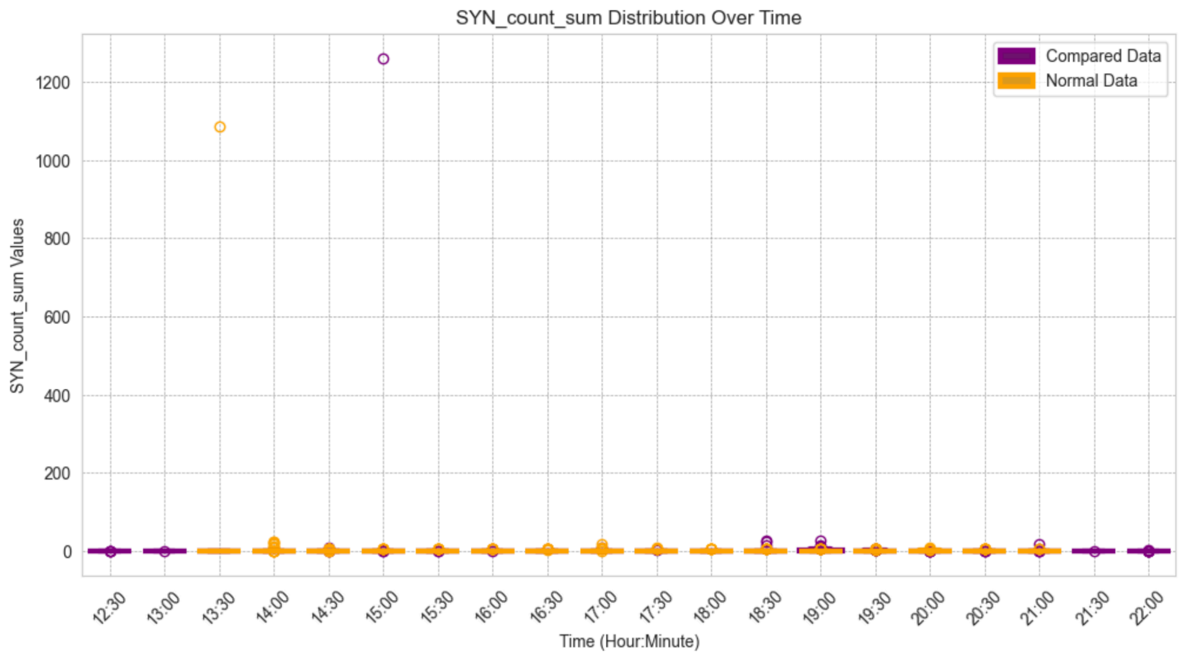
Flow Packets	Flow Bytes
5411	0

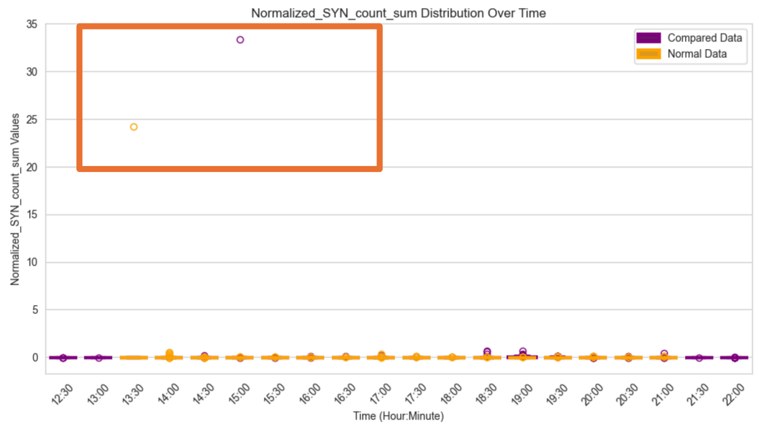




- The payload level is significantly lower than its usual behavior throughout the described communication, and its behavior jumps downward as it progresses. This behavior is characteristic of exceptional cases where there is a jump or drop in a particular feature in communication. In addition, it can be seen that at the time of the attack, 15:15, the number of packets is significantly larger than the Payload at this moment.

Comparison with Normal Traffic





More then 10 precents above ths higest point in the Normal Data.

- Mean, std, Quarters, and max: Bytes > Packets
- Attack scenario shows sudden bursts followed by system slowdowns

Statistic of Normal Data:

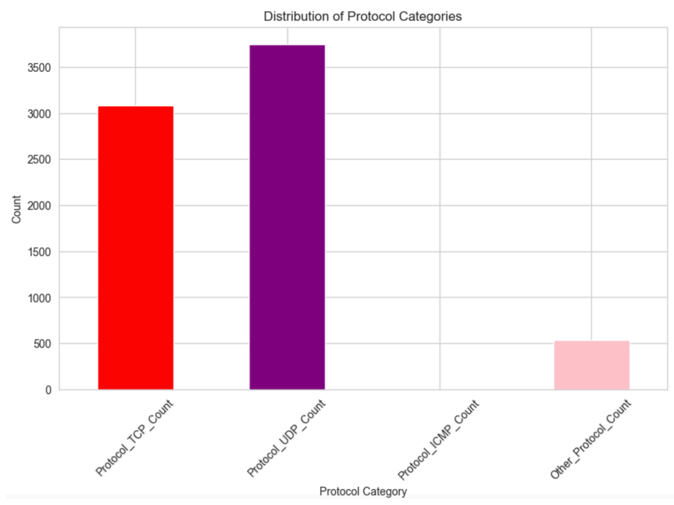
	Flow_Packets_s_avg	Flow_Bytes_s_avg
count	589	589
mean	1,625.344004	45,330.44
std	6,368.88894	742,403.20
min	0.025	0.00
25%	0.198407	0.00
50%	0.747536	36.68
75%	2.207692	187.92
max	131,166.0729	14,484,450.00

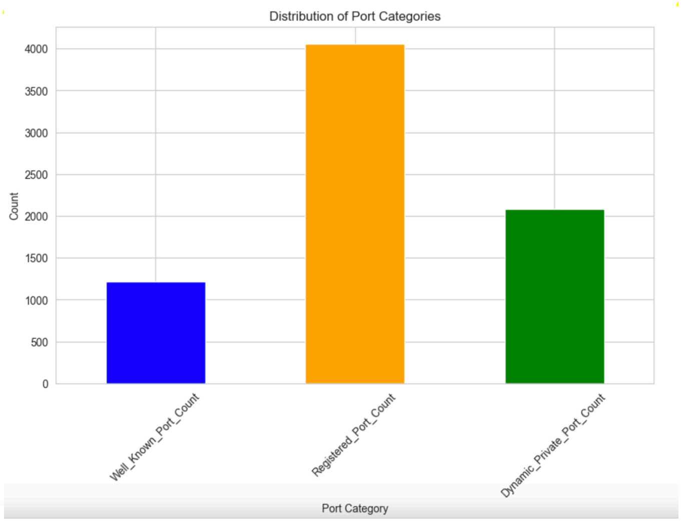
We can see Byte value are higher.

Compared Data:

CountRequests	Syn_count_sum	ACK_count_sum	SYN_ACK_Ratio
1261	1260	1	630
Flow_Packets_s_avg	**Flow_Bytes_s_avg**	**Protocol**	**Src Port**
5411	0	TCP, Other	[Registered Ports, Well-Known Ports]

Ports and Protocols:





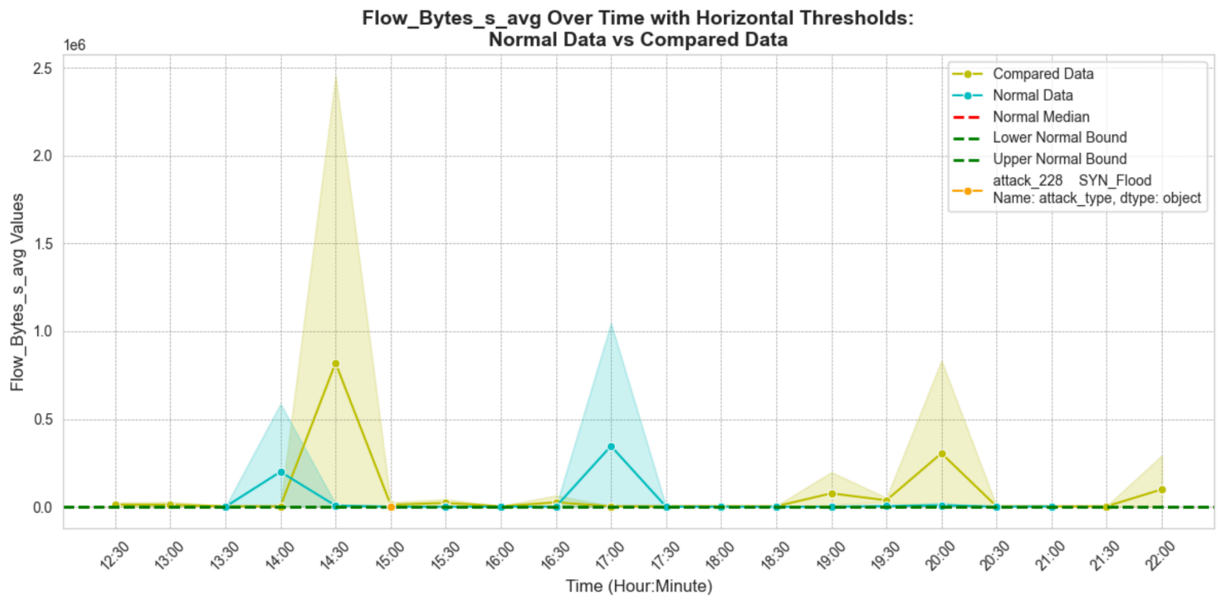
The vast majority of traffic (1261 ports) is on Registered ports, only one port is Well-Known, and there are no Dynamic/Private ports - indicating a distinct attack pattern based on sources. In terms of attack characteristics and target selection, Registered ports (1024-49151) are used by specific applications such as databases, remote management services, VPNs, and custom protocols. An attack focused on these ports indicates an attempt to harm critical services that do not operate on standard ports.

Compared Data:

Protocol_TCP_Count	Protocol_UDP_Count	Protocol_ICMP_Count	Other_Protocol_Count
1261	0	1261	0

Well_Known_Port_Count	Registered_Port_Count	Dynamic_Private_Port_Count
1	1261	0

Feature Spike and Decay Patter:



A jump in `Byte flow` observed in the graph before the identified attack time could be due to a general jump in one of the features preceding the attack. This could occur for several reasons, such as penetration testing or scanning in the pre-attack phase, or a legitimate traffic change. Comparing this behavior to the "normal" data can help determine the nature of the jump. An unusual increase relative to the normal data strengthens the possibility of an attack.

The graph also exhibits a "Decay Pattern" or "Tapering Off Effect," characterized by jumps that gradually decrease. This pattern may indicate a gradual failure of the attack as the attacker's resources are depleted or blocks increase. In the context of DoS attacks, this could be "Attack Exhaustion," where the load decreases after a peak.

A pre-attack spike may indicate reconnaissance or probing activity.

Key Takeaways

- SYN Flood is a critical DoS attack that exploits the TCP handshake mechanism.

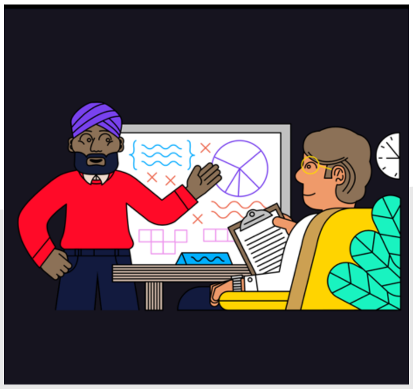
- The analysis detected significant anomalies in SYN packet volume and handshake patterns.
- Proactive mitigation strategies are crucial for network resilience and security.

Future Work

- Enhancing detection using AI-based anomaly detection techniques and unsupervized Data Science methods.
- Improving response strategies for customer service and incident handling.

Conclusion

This project successfully developed a framework for analyzing network traffic data and detecting potential DoS attacks. The analysis identified a SYN Flood attack within the dataset, highlighting the importance of proactive threat hunting and anomaly detection in maintaining network security. Future work will focus on refining the detection mechanisms and incorporating more advanced techniques to improve accuracy and response capabilities.



Sources and related content

1. https://www.kaggle.com/datasets/sowmyamyneni/dapt2020
2. https://www.splunk.com/en_us/blog/learn/ttp-tactics-techniques-procedures.html
3. https://purplesec.us/learn/prevent-syn-flood-attack/
4. https://www.paloaltonetworks.com/blog/security-operations/6-questions-you-must-ask-for-a-successful-incident-response/
5. Microsoft SecurityDocumentation
6. https://developers.cloudflare.com/fundamentals/basic-tasks/protect-your-origin-server/
7. https://www.konfidas.com/cybersecurity-briefs/cybersecuritybrief05012023
8. https://shushan.co.il/port-list-הסבר-על-פורטים
9. Cloudflare - How Proxies Hide Attack Origins Cloudflare Security Blog
10. OWASP - DDoS Prevention Cheat Sheet OWASP
11. Palo Alto Networks - Understanding DDoS and Proxy Attacks Palo Alto Networks
12. https://owasp.org/www-community/attacks/Command_Injection
13. https://purplesec.us/learn/prevent-syn-flood-attack/