Exploratory Data Analysis on Malware IoT Traffic

Dataset used: https://www.kaggle.com/datasets/agungpambudi/network-malware-detection-connection-analysis

This dataset describes network traffic which has been flagged by IoT malware detection systems. Important attributes for the data include the timestamp, uid (unique identifier), source IP address and port, destination IP address and port, and the label of the connection.

In this exercise, I perform the following:

- Download a dataset from Kaggle about network traffic and uploaded it to Colab.
- Organize the data into columns utilizing the file's unique delimitter (|).
- Perform basic cleaning tasks such as dropping null values and irrelevant columns.
- Explore the data by asking and answering 5 simple EDA questions.
- Discuss the results of the analysis and suggest potential next steps.

I initially import any necessary libraries and load all 12 of the files.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Loading files
df_1 = pd.read_csv("/content/CTU-IoT-Malware-Capture-1-1conn.log.labeled.csv")
df_2 = pd.read_csv("/content/CTU-IoT-Malware-Capture-20-1conn.log.labeled.csv")
df_3 = pd.read_csv("/content/CTU-IoT-Malware-Capture-21-1conn.log.labeled.csv")
df_4 = pd.read_csv("/content/CTU-IoT-Malware-Capture-3-1conn.log.labeled.csv")
df_5 = pd.read_csv("/content/CTU-IoT-Malware-Capture-34-1conn.log.labeled.csv")
df_6 = pd.read_csv("/content/CTU-IoT-Malware-Capture-35-1conn.log.labeled.csv")
df_7 = pd.read_csv("/content/CTU-IoT-Malware-Capture-42-1conn.log.labeled.csv")
df_8 = pd.read_csv("/content/CTU-IoT-Malware-Capture-44-1conn.log.labeled.csv")
df_9 = pd.read_csv("/content/CTU-IoT-Malware-Capture-48-1conn.log.labeled.csv")
df_10 = pd.read_csv("/content/CTU-IoT-Malware-Capture-60-1conn.log.labeled.csv")
df_11 = pd.read_csv("/content/CTU-IoT-Malware-Capture-8-1conn.log.labeled.csv")
df_12 = pd.read_csv("/content/CTU-IoT-Malware-Capture-9-1conn.log.labeled.csv")
```

Next, I look at samples from several of the datasets to ensure everything loaded properly. They are initially difficult to read and very cramped.

```
print(df_1.sample(10))
print(df_2.sample(10))
print(df_3.sample(10))
```

```
         ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|miss
896734  1526234439.023739|Cll7nV2tuI4OrjTab5|192.168.1...
166776  1525942582.007071|Ct4J9k3cnQDbpY3Uq8|192.168.1...
562     1525880001.056842|CQcf6wkGdOsUIIZrj|87.18.20.3...
607186  1526111240.009333|Ct3S462jXplw7OBhqf|192.168.1...
6321    1525882122.020218|COqf5mJH4F4ChbTS1|192.168.10...
590879  1526104785.99969|Cav45Q3KChmskf7Fg4|192.168.10...
291096  1525989920.022084|CbvNqG2Ji3hWxlDste|192.168.1...
199024  1525954942.042392|CmHpWf1JFSndenSAO1|192.168.1...
646916  1526127222.003587|C2q6Vp16Ikuz5K1Zma|192.168.1...
527663  1526080143.022516|C9tKtoRuzoo63xfU2|192.168.10...
      ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|missed
1456  1538511440.549519|Cm5sc44pOimtUxHlF2|192.168.1...
2239  1538534917.550741|C5zNNT33RKN97Ozpii|192.168.1...
3087  1538559991.550287|C5RSk8vixRM8WJWLb|192.168.10...
932   1538495831.551009|CZ5cGQOWmiyEkhOPa|192.168.10...
1888  1538522980.548459|CRWht03rqNjbsI0qAi|192.168.1...
12    1538478864.657195|CE1jSk2DkfDD8Rz3Ch|192.168.1...
2712  1538557185.548839|CLkJwe1esJjqSGerCd|192.168.1...
1892  1538523237.547834|Cf4a9qNp55FgFHN3|192.168.100...
2062  1538530289.550990|CmbNp71ntVpmPzHvUc|192.168.1...
1580  1538512542.557450|CZSCbb3i4tGZEEGgLa|192.168.1...
      ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|missed
1287  1538598907.658338|ChbKUA4sT67t1WGMTa|192.168.1...
1787  1538602199.708768|CPLYDiu5lDeWyKNlj|192.168.10...
2662  1538615249.677118|CQcGcf3g0IdFp9Zt26|192.168.1...
3143  1538636956.798863|CCfsLs2yNLnKI8dLM3|192.168.1...
1997  1538603528.686001|CT1qpO2dIYqwaa6Qji|192.168.1...
215   1538574376.657881|CeIxNx1BiFfVE69IT8|192.168.1...
2348  1538613286.665555|CzDfM8DbQDLa07vGh|192.168.10...
832   1538596211.660903|CWSz7Y1CpsYltz7Atk|192.168.1...
```

```
2243  1538612691.676344|CR8bdp3HfZFZiJdpoj|192.168.1...
1439  1538599845.657468|Ca06414BZnC3JnsUq3|192.168.1...
```

The next step is to perform basic EDA to gain understanding of the data.

In this step, I learned that even though the datasets hold the information for every column, this information is instead stored in a single column with the datatype "object". This leads to difficulties reading and filtering the data.

There is also high variance in file size, ranging from 237 rows to over 10 million.

```
# Obtaining the shape of the data (rows, columns)
print(df_1.shape)
print(df_2.shape)
print(df_3.shape)
print(df_4.shape)
print(df_5.shape)
print(df_6.shape)
print(df_7.shape)
print(df_8.shape)
print(df_9.shape)
print(df_10.shape)
print(df_11.shape)
print(df_12.shape)

# Obtaining a basic description of the data (unique and common values)
print(df_1.describe())
print(df_2.describe())
print(df_3.describe())

# Obtaining info about data (datatype)
print(df_1.info())
print(df_2.info())
print(df_3.info())
```

```
(1008748, 1)
(3209, 1)
(3286, 1)
(156103, 1)
(23145, 1)
(10447787, 1)
(4426, 1)
(237, 1)
(3394338, 1)
(3581028, 1)
(10403, 1)
(6378293, 1)
       ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|miss
count                                            1008748
unique                                           1008748
top     1525879831.015811|CUmrqr4svHuSXJy5z7|192.168.1...
freq                                                   1
       ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|miss
count                                               3209
unique                                              3209
top     1538478769.600293|CSQG794riQ4XnzTxP2|192.168.1...
freq                                                   1
       ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|miss
count                                               3286
unique                                              3286
top     1538572953.710599|Cu3Tieri43IPsyBO3|192.168.10...
freq                                                   1
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1008748 entries, 0 to 1008747
Data columns (total 1 columns):
 #   Column
---  ------
 0   ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|missed
dtypes: object(1)
memory usage: 7.7+ MB
None
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3209 entries, 0 to 3208
Data columns (total 1 columns):
 #   Column
---  ------
 0   ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|missed
dtypes: object(1)
memory usage: 25.2+ KB
None
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3286 entries, 0 to 3285
Data columns (total 1 columns):
 #   Column
---  ------
 0   ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|missed
dtypes: object(1)
memory usage: 25.8+ KB
None
```

The next step is to finally organize the data so that it properly fits into columns.

(This took me quite some time and was very difficult for me to eventually get right, so I was quite pleased to when it eventually worked.)

I decide to work with 1 dataframe instead of the initial 12 here.

```
# Rename the existing column to 'column_name'
df_1 = df_1.rename(columns={df_1.columns[0]: 'column_name'})
# Organize the data into correct columns using the delimitter to separate values
df_1[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_sta

#df_2 = df_2.rename(columns={df_2.columns[0]: 'column_name'})
#df_2[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_st

#df_3 = df_3.rename(columns={df_3.columns[0]: 'column_name'})
#df_3[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_st
```

```
df_1.head()
```

	column_name	ts	uid	id.orig_h	id.orig_p	id.resp_h	id
0	1525879831.015811|CUmrqr4svHuSXJy5z7|192.168.1...	1525879831.015811	CUmrqr4svHuSXJy5z7	192.168.100.103	51524	65.127.233.163	
1	1525879831.025055|CH98aB3s1kJeq6SFOc|192.168.1...	1525879831.025055	CH98aB3s1kJeq6SFOc	192.168.100.103	56305	63.150.16.171	
2	1525879831.045045|C3GBTkINvXNjVGtN5|192.168.10...	1525879831.045045	C3GBTkINvXNjVGtN5	192.168.100.103	41101	111.40.23.49	
3	1525879832.016240|CDe43c1PtgynajGI6|192.168.10...	1525879832.016240	CDe43c1PtgynajGI6	192.168.100.103	60905	131.174.215.147	
4	1525879832.024985|CJaDcG3MZzvf1YVYI4|192.168.1...	1525879832.024985	CJaDcG3MZzvf1YVYI4	192.168.100.103	44301	91.42.47.63	

5 rows × 24 columns

Continuing the data preparation, the next step is to drop the original messy column and then begin the cleaning process.

```
# Dropping the original non-organized column
df_1.drop('column_name', axis=1, inplace=True)
```

```
# Checking for shapes and nulls
df_1.shape
df_1.describe()
df_1.info()

# Drops null values
df_1.dropna(inplace=True)
df_1.shape
df_1.info()
df_1.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 212448 entries, 0 to 1008685
Data columns (total 24 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   ts              212448 non-null  datetime64[ns]
 1   uid             212448 non-null  object
 2   id.orig_h       212448 non-null  object
 3   id.orig_p       212448 non-null  Int64
 4   id.resp_h       212448 non-null  object
 5   id.resp_p       212448 non-null  Int64
 6   proto           212448 non-null  object
 7   service         212448 non-null  object
 8   duration        212448 non-null  float64
 9   orig_bytes      212448 non-null  Int64
 10  resp_bytes      212448 non-null  Int64
 11  conn_state      212448 non-null  object
 12  local_orig      212448 non-null  object
 13  local_resp      212448 non-null  object
 14  missed_bytes    212448 non-null  Int64
 15  history         212448 non-null  object
 16  orig_pkts       212448 non-null  Int64
 17  orig_ip_bytes   212448 non-null  Int64
 18  resp_pkts       212448 non-null  Int64
 19  resp_ip_bytes   212448 non-null  Int64
 20  tunnel_parents  212448 non-null  object
 21  label           212448 non-null  object
 22  detailed_label  212448 non-null  object
 23  hour            212448 non-null  int32
dtypes: Int64(9), datetime64[ns](1), float64(1), int32(1), object(12)
memory usage: 41.5+ MB
```


```
<class 'pandas.core.frame.DataFrame'>
Index: 212448 entries, 0 to 1008685
Data columns (total 24 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   ts              212448 non-null  datetime64[ns]
 1   uid             212448 non-null  object
 2   id.orig_h       212448 non-null  object
 3   id.orig_p       212448 non-null  Int64
 4   id.resp_h       212448 non-null  object
 5   id.resp_p       212448 non-null  Int64
 6   proto           212448 non-null  object
 7   service         212448 non-null  object
 8   duration        212448 non-null  float64
 9   orig_bytes      212448 non-null  Int64
 10  resp_bytes      212448 non-null  Int64
 11  conn_state      212448 non-null  object
 12  local_orig      212448 non-null  object
 13  local_resp      212448 non-null  object
 14  missed_bytes    212448 non-null  Int64
 15  history         212448 non-null  object
 16  orig_pkts       212448 non-null  Int64
 17  orig_ip_bytes   212448 non-null  Int64
 18  resp_pkts       212448 non-null  Int64
 19  resp_ip_bytes   212448 non-null  Int64
 20  tunnel_parents  212448 non-null  object
 21  label           212448 non-null  object
 22  detailed_label  212448 non-null  object
 23  hour            212448 non-null  int32
dtypes: Int64(9), datetime64[ns](1), float64(1), int32(1), object(12)
memory usage: 41.5+ MB
```


	ts	id.orig_p	id.resp_p	duration	orig_bytes	resp_bytes	missed_bytes	orig_pkts	orig_ip_bytes	resp_
count	212448	212448.0	212448.0	212448.000000	212448.0	212448.0	212448.0	212448.0	212448.0	212
mean	2018-05-11 19:44:16.420595968	45089.159098	5163.134362	3.203882	4.754095	11.856228	0.0	3.35644	197.463939	0.67
min	2018-05-09 15:30:31.015810966	3.0	0.0	0.000002	0.0	0.0	0.0	0.0	0.0	
25%	2018-05-10 16:32:38.761027328	39022.0	23.0	2.998560	0.0	0.0	0.0	3.0	180.0	
50%	2018-05-11 17:49:09.504612864	46371.5	2323.0	2.998797	0.0	0.0	0.0	3.0	180.0	
75%	2018-05-12	53668.0	8080.0	2.999042	0.0	0.0	0.0	3.0	180.0	

```
# Manaully changing datatypes of all columns

# Timestamp of connection
```

```
df_1['ts'] = pd.to_datetime(df_1['ts'], unit ='s')
# Unique ID
df_1['uid'] = df_1['uid'].astype(str)
# Source IP address
df_1['id.orig_h'] = df_1['id.orig_h'].astype(str)
# Source port used
df_1['id.orig_p'] = pd.to_numeric(df_1['id.orig_p'], errors='coerce').astype('Int64')
# Destination IP address
df_1['id.resp_h'] = df_1['id.resp_h'].astype(str)
# Destination port
df_1['id.resp_p'] = pd.to_numeric(df_1['id.resp_p'], errors='coerce').astype('Int64')
# Network protocol used
df_1['proto'] = df_1['proto'].astype(str)
# Service used in connection
df_1['service'] = df_1['service'].astype(str)
# Duration of connection
df_1['duration'] = df_1['duration'].replace('-', np.nan).astype(float)
# Number of bytes sent from source to destination
df_1['orig_bytes'] = pd.to_numeric(df_1['orig_bytes'], errors='coerce').astype('Int64')
# Number of bytes sent back from destination to source
df_1['resp_bytes'] = pd.to_numeric(df_1['resp_bytes'], errors='coerce').astype('Int64')
# State of the connection
df_1['conn_state'] = df_1['conn_state'].astype(str)
# Whether origin of connection is local (blank)
df_1['local_orig'] = df_1['local_orig'].astype(str)
# Whether connection is local (Blank)
df_1['local_resp'] = df_1['local_resp'].astype(str)
# Number of missed bytes
df_1['missed_bytes'] = pd.to_numeric(df_1['missed_bytes'], errors='coerce').astype('Int64')
# History of connection states
df_1['history'] = df_1['history'].astype(str)
# Number of packets sent from source to destination
df_1['orig_pkts'] = pd.to_numeric(df_1['orig_pkts'], errors='coerce').astype('Int64')
# Number of IP bytes sent back from source to destination
df_1['orig_ip_bytes'] = pd.to_numeric(df_1['orig_ip_bytes'], errors='coerce').astype('Int64')
# Number of packets sent from dest to source
df_1['resp_pkts'] = pd.to_numeric(df_1['resp_pkts'], errors='coerce').astype('Int64')
# Number of IP bytes sent back from source to dest
df_1['resp_ip_bytes'] = pd.to_numeric(df_1['resp_ip_bytes'], errors='coerce').astype('Int64')
# Whether connection is part of tunnel
df_1['tunnel_parents'] = df_1['tunnel_parents'].astype(str)
# Basic label for connection (malicious/benign)
df_1['label'] = df_1['label'].astype(str)
# Detailed description for connection
df_1['detailed_label'] = df_1['detailed_label'].astype(str)
```

```
df_1.sample(100)
```

	ts	uid	id.orig_h	id.orig_p	id.resp_h	id.resp_p	proto	service	duration	orig_by
58566	2018-05-09 21:30:19.012273073	CvwPA31rpmx5lPj619	192.168.100.103	46861	212.222.174.108	23	tcp	-	2.999039	
1006172	2018-05-14 07:06:36.031743050	CqdOCA9YEStldlzz9	192.168.100.103	56524	38.228.142.202	23271	tcp	-	2.998995	
949252	2018-05-14 00:17:45.054860115	CdeVtq2dxyVo7SnSid	192.168.100.103	35017	92.102.52.159	23	tcp	-	NaN	<
454437	2018-05-11 15:22:14.012839079	Cj9WC63cRNuFUZDXm8	192.168.100.103	48907	35.234.77.213	23	tcp	-	2.998792	
641307	2018-05-12 11:35:35.042100906	Cprhlb2UcHiGcmvsGg	192.168.100.103	34972	140.43.84.23	23	tcp	-	NaN	<
...	...	...	...	...	...	...	...	...	...	
191964	2018-05-10 11:37:18.013542891	CT1AZP1kYl7z04nOF4	192.168.100.103	49865	221.218.2.161	2323	tcp	-	2.999308	
294775	2018-05-10 22:28:03.011280060	CJdSEs1w05vnRrC5c3	192.168.100.103	43763	187.203.21.32	24157	udp	-	NaN	<
719320	2018-05-12 20:46:36.024605989	C5fUQE4vOutRN2vdTb	192.168.100.103	51186	161.49.156.221	8080	tcp	-	NaN	<
977372	2018-05-14 03:39:39.040369034	CLpK0s4JloiAw5qbLc	192.168.100.103	41550	216.167.144.248	2323	tcp	-	NaN	<
732800	2018-05-12 22:23:29.025087118	CKJMSl4oI1y6UEO1fg	192.168.100.103	59448	48.131.96.251	23	tcp	-	NaN	<

100 rows × 23 columns

Exploring the Data

1) How many connections were labeled 'Malicious' and how many were labeled 'Benign'?

To answer this, I filter tbe connections using a 'for' loop and print the results. This is to have a better understanding of the scope of how much malicious content was recorded.

```
# variables to store counts
m_count = 0
b_count = 0

# Iterates through the label column
for label in df_1['label']:
  if label == 'Benign':
    # adds 1 to the benign count
    b_count += 1
  elif label == 'Malicious':
    # adds 1 to the malicious count
    m_count += 1

# Print the counts
print("Benign Count:", b_count)
print("Malicious Count:", m_count)
```

```
Benign Count: 469275
Malicious Count: 539473
```

The results are roughly even, although there are slightly more malicious connections than benign.

2) Which ports are most commonly present within records of malicious connections?

To answer this, I explore which ports are most commonly associated with the malicious label in the dataset. This is to gain insight into which ports may have been involved in a par

```
# Filters through the label column for malicious connections
mal_df = df_1[df_1['label'] == 'Malicious']

# Collecting frequency of each origin and destination port
o_val_counts = mal_df['id.orig_p'].value_counts()
```

```
d_val_counts = mal_df['id.resp_p'].value_counts()

# Printing top 5 results
print(o_val_counts.head(5))
print(d_val_counts.head(5))
```

```
id.orig_p
54605    19
45334    18
54239    17
40176    17
59180    17
Name: count, dtype: Int64
id.resp_p
23      95561
8080    48289
2323    30299
9527    15142
Name: count, dtype: Int64
```

The top origin port was port 54605 with 19 malicious connections associated with it.

The top destination port was port 23 with 95,561 malicious connections associated with it. I assume this is because port 23 is a common port for attacks or was involved in a particular attack(s) that took place.

3) What is the timeframe of the collected records?

To answer this, I initially determine the timestamps of the first and last recorded file. This is to better understand the scope of the timeframe of collected files.

```
first_record = df_1['ts'].min()
last_record = df_1['ts'].max()

print(first_record)
print(last_record)
```

```
2018-05-09 15:30:31.015810966
2018-05-14 07:24:41.031404972
```

The initial record was logged on May 9, 2018, at roughly 3:30pm.

The last record was logged on May 14, 2018, at roughly 7:25am.

Thus, these records hold roughly 5 days worth of network traffic logs.

4) What hours of day have the highest number of malicious connections?

To answer this, I explore the counts of attacks for different times of the day. This is to determine if there is or was a particular pattern or common time noteworthy in the data.

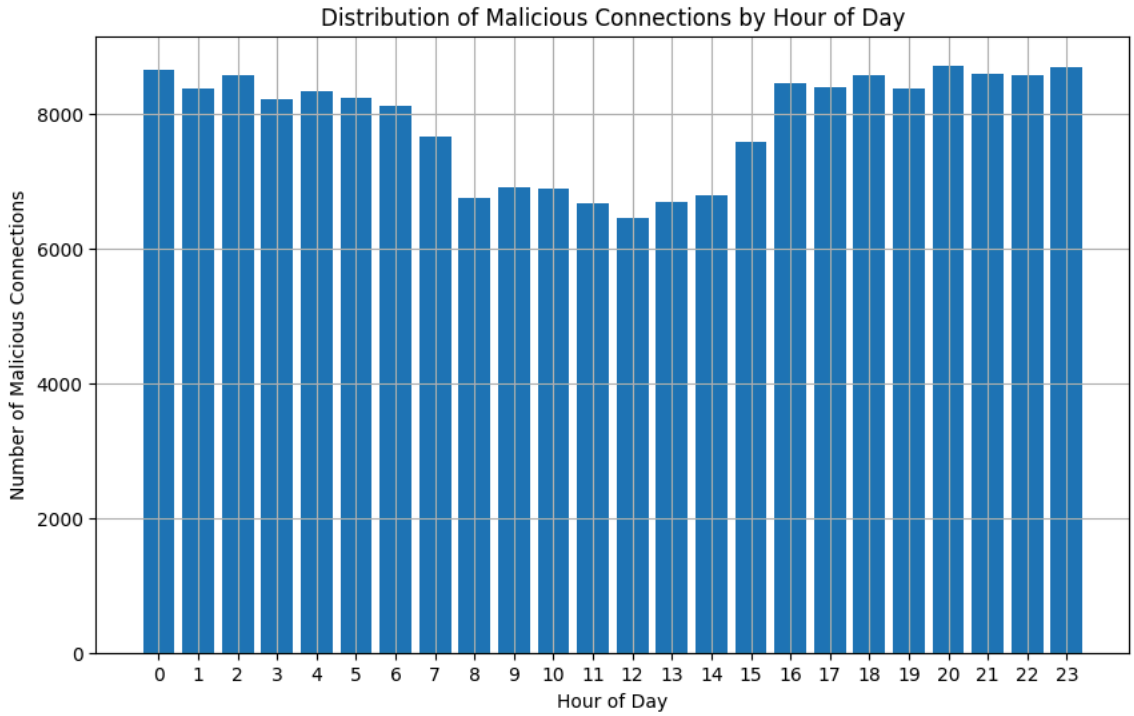
I then organize the counts of malicious files according to hour in the day to determine if a pattern in the results.

```
# Creates an hour column
mal_df['hour'] = mal_df['ts'].dt.hour

# Filters for malicious connections and groups by frequency during each hour of day
mal_hourly_counts = mal_df.groupby('hour').size()

# Prints results
print(mal_hourly_counts)
print(mal_hourly_counts.sort_values(ascending=False))
```

```
hour
0     8660
1     8383
2     8579
3     8222
4     8331
5     8243
6     8110
7     7653
8     6748
9     6909
10    6890
11    6679
12    6446
```

```
12    6446
13    6698
14    6797
15    7578
16    8460
17    8392
18    8572
19    8374
20    8716
21    8588
22    8574
23    8689
dtype: int64
hour
20    8716
23    8689
0     8660
21    8588
2     8579
22    8574
18    8572
16    8460
17    8392
1     8383
19    8374
4     8331
5     8243
3     8222
6     8110
7     7653
15    7578
9     6909
10    6890
14    6797
8     6748
13    6698
11    6679
12    6446
dtype: int64
<ipython-input-52-8d5785896c61>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-
```

```
# creates histogram for frequency of malicious records during each hour of day
plt.figure(figsize=(10, 6))
plt.bar(mal_hourly_counts.index, mal_hourly_counts.values)
plt.xlabel("Hour of Day")
plt.ylabel("Number of Malicious Connections")
plt.title("Distribution of Malicious Connections by Hour of Day")
# Sets x-axis ticks to represent each hour
plt.xticks(range(24))
plt.grid(True)
plt.show()
```



The hour of day with the highest amount of malicious connections is the 20th hour, between 8-9pm.

It seems that more malicious connections have been logged during the evening and early morning, which lines up as being the opposite of common working hours.

5) What are the most common source and destination IP addresses?

To answer this, I explore the most common source and enpoint IPs, cross referenced with several other variables. This is to determine which, if any, IP addresses are associated with one another or possess a common description.

```
# Checks if there are common values for detailed labels to sort by
df_1.groupby('detailed_label').size()
```

detailed_label	0
-	23157
PartOfAHorizontalPortScan	189291

dtype: int64

```
# Calculates the most common origin IP and destination IP addresses
orig_IPs = df_1.groupby('id.orig_h').size().sort_values(ascending=False)
resp_IPs = df_1.groupby('id.resp_h').size().sort_values(ascending=False)
# Prints the results
print(orig_IPs.head(10))
print(resp_IPs.head(10))

# Cross referencing which origin and destination IPs are most commonly associated together
grouped_df = df_1.groupby(['id.orig_h', 'id.resp_h']).size().reset_index(name='count')
sort_df = grouped_df.sort_values(by='count', ascending=False)
# Prints the results
print(sort_df.head(10))

# Cross referencing which detailed labels are matched with the most origin IP addresses
grouped_df = df_1.groupby(['id.orig_h', 'detailed_label']).size().reset_index(name='count')
sort_df = grouped_df.sort_values(by='count', ascending=False)
# Prints the results
print(sort_df.head(10))
sort_df
```

```
id.orig_h
192.168.100.103    210739
192.168.100.1         949
70.45.29.240           10
210.206.154.134         9
201.81.12.29            7
146.94.254.33           6
81.130.230.46           6
118.163.192.88          6
221.5.224.77            5
125.125.23.137          4
dtype: int64
id.resp_h
147.231.100.5      3849
192.168.100.103    1709
89.221.214.130      995
37.187.104.44       938
213.239.154.12      785
210.206.154.134     127
221.5.224.77        124
175.196.5.46        124
118.163.192.88      124
200.170.160.2       122
dtype: int64
              id.orig_h        id.resp_h  count
49175   192.168.100.103    147.231.100.5   3849
184682  192.168.100.103   89.221.214.130    995
263       192.168.100.1  192.168.100.103    949
133804  192.168.100.103    37.187.104.44    938
116772  192.168.100.103   213.239.154.12    785
113812  192.168.100.103  210.206.154.134    127
```