Implementing Snort IDS to secure DVWA Against Common Cyber Attacks

-APRIL 2025

By： J Saiprakash [Student]

Github：https://github.com/prakash-js

Securing DVWA with Snort IDS/IPS

This is a self-initiated project where I focused on monitoring and detecting attacks against DVWA (Damn Vulnerable Web Application) using Snort IDS. I wrote custom Snort rules to identify threats like SQL Injection, XSS, port scanning, command injection, reverse shells, Denial of Service (DoS) attacks, and brute-force login attempts.

The environment was set up on a single Ubuntu VM, configured with a dual-interface network setup to better handle traffic and reduce false positives. This setup allowed me to gain practical, hands-on experience in writing and tuning Snort rules, and helped me understand how custom detection can be applied to simulate real-world attack scenarios in a controlled lab environment.

What is DVWA?

DVWA stands for Damn Vulnerable Web Application. It is a web application designed with known security vulnerabilities. It allows security professionals and enthusiasts to practice identifying and exploiting common web application weaknesses in a controlled and safe environment. DVWA provides a platform to study security concepts and test the effectiveness of security tools.

Deployed DVWA using the official GitHub repository: https://github.com/digininja/DVWA

What is Snort?

SNORT is a powerful open-source **Intrusion Detection System (IDS)** and **Intrusion Prevention System (IPS)** that provides real-time network traffic analysis and data packet logging. SNORT uses a rule-based language that combines anomaly, protocol, and signature inspection methods to detect potentially malicious activity.

IDS:

IDS is an passive monitoring solution for detecting possible malicious activities/patterns ,abnormal incidents, and policy violations.

It is responsible for generating alerts for each suspicious event.

IPS:

IPS is an active monitoring solution for detecting possible malicious activities/patterns ,abnormal incidents, and policy violations.

It is responsible for stopping/preventing/terminating the suspicious event as soon as the detection is performed.

What are Snort Rules?

Snort rules are user-defined instructions that tell the Snort Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) what kind of network traffic to watch for and how to respond when it is detected. These rules are written in a specific syntax that allows Snort to inspect packets in real-time and trigger alerts or block malicious traffic.

Basic Components of a Snort Rule

- **Action** – Defines what Snort should do when the rule is triggered. Common actions include alert, log, pass, drop, reject, etc.

- **Protocol** – Specifies the protocol the rule applies to, such as TCP, UDP, ICMP

- **Flags** – Used to match specific TCP flags in packets. This can help identify connection states.
  Examples: S (SYN), A (ACK), SA (SYN-ACK)

- **Source and Destination IP/Ports** – Indicates which IP addresses and ports the rule will inspect. Wildcards like **any** can be used.

- **content** – Searches for specific strings or payloads in packet data. Often used for detecting known attack patterns.
  Example: content:"/etc/passwd";

- **nocase** – Makes the content match case-insensitive. Useful when attackers try to bypass detection using casing variations.
  Example: content:"select"; nocase;

- **detection_filter** – Controls how frequently a rule can trigger from a specific source. Helps in detecting brute-force or scanning behavior while reducing false positives.
  Example: detection_filter:track by_src, count 5, seconds 60;

- **sid** – The Snort rule ID. It uniquely identifies the rule and is required for every rule.
  Example: sid:10000001;

- **rev** – The revision number of the rule. Used to keep track of rule updates.
  Example: rev:1;

For in-depth documentation, installation guides, and official rule writing tutorials, refer to the official Snort website: https://docs.snort.org/

Overview of Some Rules I Created and used:

In this section, I have explained a selection of key Snort rules I wrote during this project. Rather than listing every rule line-by-line, I've highlighted a few that demonstrate different detection techniques used to monitor and identify specific attack behaviors. Each rule is shown with its purpose, logic, and configuration choices. This helps provide insight into how these rules were built not just for detection, but to minimize false positives while simulating real-world attack scenarios.

All custom Snort rules written and used in this project to detect specific attacks are publicly available on my GitHub.

- **GitHub Repository**:
  https://github.com/prakashjs/project_with_snort_IDS

Rules Used for Port Scanning and DOS:

```
alert udp $EXTERNAL_NET any -> $HOME_NET !53 (detection_filter:track by_src, count 4, seconds 60; msg:"Potential UDP Port Scan Detected"; sid:10000001; rev:1;)

alert udp $EXTERNAL_NET any -> $HOME_NET !5353 (detection_filter:track by_src, count 4, seconds 60; msg:"Potential UDP Port Scan Detected"; sid:10000002; rev:1;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 0:79 (detection_filter:track by_src, count 4, seconds 60; msg:"Potential TCP Port Scan Detected"; sid:10000003; rev:1;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 81:65535 (detection_filter:track by_src, count 5, seconds 60; msg:"Potential TCP Port Scan Detected"; sid:10000004; rev:1;)
```

The above rules were used to detect port scanning activity by monitoring for connection attempts to any TCP or UDP port other than TCP 80 (HTTP) and UDP ports 53, 5353 (DNS and mDNS). If more than 4 connection attempts occur from the same source IP within 60 seconds, an alert is triggered with the message "Potential TCP/UDP Port Scan Detected."

I chose not to use any flags in these rules. While it's common to use flags like `S` (SYN), `A` (ACK), or `SA` (SYN-ACK) and others to identify the state of TCP handshakes, I focused instead on simple connection attempt detection across disallowed ports.

The same rule structure was also used to detect Denial of Service (DoS) attacks, such as TCP flood and UDP flood. To achieve this, I changed the destination port to any, and set the detection threshold to count 1000, seconds 1. This allowed the rule to trigger an alert when a source sends 1000 or more packets within 1 second, which is typical flood behavior.

Snort rule for Payload Based Attacks:

The following Snort rule format was used to detect various payload-based attacks, including **SQL Injection (SQLi), Cross-Site Scripting (XSS), Command Injection**, and **Path Traversal**:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80
(detection_filter: track by_src, count 3, seconds 20;flow:to_server;
msg:"<Potential attack detected>";content:"<payload>";nocase;sid:<ID>;rev:1;)
```

Each rule is designed to trigger an alert if three matching payloads are sent from the same source within 20 seconds, indicating potential automated or brute-force attack behavior.

The flow:to_server; keyword ensures that only client-to-server traffic is inspected, which is where these types of injection attacks are typically delivered.

The content:"<payload>"; field is replaced with actual known malicious patterns to detect specific attack types.

The nocase; option is used to make the detection case-insensitive, improving effectiveness since attackers often obfuscate payloads using different character cases.

This rule structure allows the detection of common web-based attack attempts by inspecting HTTP request bodies or parameters for signs of exploitation.

By keeping the rule generic and modifying only the msg and content fields, This approach made it easy to detect different types of attacks using the same basic rule format.

Snort rule for Revershell Detection:

```
alert tcp $HOME_NET !80 ->  $EXTERNAL_NET !80
(flow:established,to_client;msg:"Potential Reverse Connection
Detected";flags:PA;sid:1100001;)

alert tcp $EXTERNAL_NET  !80 ->  $HOME_NET !80 (msg:"Reverse
Connection Detected";content:"/bin/bash";nocase;sid:1100002;)
```

For reverse shell detection, I created a rule that monitors for traffic from server to client on all ports except port 80, and specifically looks for packets with the PA (PUSH-ACK) TCP flags. This is because, in a typical TCP-based reverse shell, once the connection is established and the attacker starts receiving command output, the server sends data with the PA flag set.

I avoided inspecting port 80 in the reverse shell detection rule to reduce false positives caused by normal web traffic. Additionally, I ran this rule in a separate Snort instance (terminal) to avoid conflict with the port scanning detection rules.

This is because, during a reverse shell connection, the initial TCP handshake and subsequent PA-flagged packets (used when sending commands or responses) can resemble port scanning behavior. In such cases, a single reverse shell session could mistakenly trigger one port scan alert and one or more reverse shell alerts, making it harder to clearly identify the activity. Running the rules separately helped isolate and better understand each type of alert.

For a few rules, I also used payload string matching in both directions — attacker to server and server to attacker. This was useful in detecting inputs like sudo -l as well as the responses, since the payload resides in the data section of the TCP segment, allowing bidirectional command and response flow during active sessions.

Snort rule for Brute Force Detection:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (
 detection_filter: track by_src, count 15, seconds 20;
content:"/DVWA/login.php"; http_uri;
content:"Login="; http_client_body;
msg:"BruteForce attack Detected";sid:10000091;)
```

This rule is designed to detect **brute-force login attempts** targeting the DVWA login page.

The condition content:"/DVWA/login.php"; http_uri; ensures the alert triggers only on HTTP requests to the login endpoint.

The second content:"Login="; http_client_body; checks the POST body to ensure the request contains login form data.

detection_filter: track by_src, count 15, seconds 20; triggers an alert if a source IP sends 15 login attempts within 20 seconds, which is typical brute-force behavior.

This helps detect repeated failed login attempts without triggering on normal user activity.

Whitelisting IPs Using Snort Threshold Configuration

In Snort, false positives can be reduced by whitelisting trusted IPs using the threshold.conf file. This is especially useful when traffic from sources like localhost, routers, or other trusted devices is mistakenly flagged as suspicious.

To reduce noise and avoid false positives in my local environment, I whitelisted several trusted IP router, host machine, and broadcast addresses.

Why Whitelist IPs?

Local Environment: When running Snort on localhost or within a private network, trusted devices (e.g., routers or nameserver or internal network devices) might trigger false alerts. Whitelisting these IPs ensures that Snort doesn't flag legitimate traffic.

Public-Facing Systems: For servers exposed to the internet, Snort can be configured to ignore alerts from trusted external IPs, such as known partners or peer-to-peer connections.

How to Whitelist IPs:

Locate the threshold.conf file (**/etc/snort/threshold.conf**)

Add the following entry to whitelist an IP or network range:

suppress gen_id 1, sig_id <SID>, track by_src, ip_src <trusted_IP_or_network>

Adjust the SID and IP as necessary.

Using Two Interfaces for Better Traffic Handling

I configured two network interfaces from different subnets:

One for Web Server traffic

One for System/OS-level traffic

Even with whitelist rules, it's difficult to filter out every background system traffic (like Ubuntu package updates). These can create false positives in Snort detection.

To solve this, I used two interfaces with different metric values:

Web Server Interface → Metric: `400`

System Interface → Metric: 100

What is Metric?

Metric decides the **priority of a route** lower metric = higher priority.

How to Set Interface Metric?

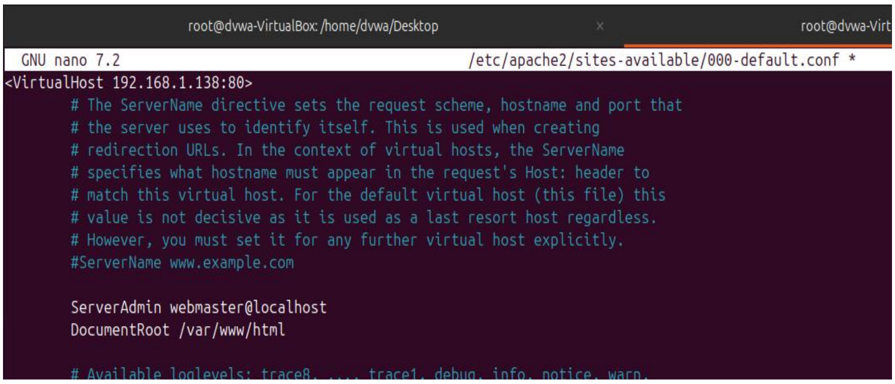
I used ip route to update the metric values temporarily and made the change **persistent** by configuring a custom systemd service.

Alternatively, we can simply use the default interface with a higher metric value to prioritize web server traffic.

Apache Configuration Update

To ensure Apache listens only on the **web-facing interface** (higher metric), update:

I updated the Apache configuration file located at /etc/apache2/sites-available/000-default.conf by changing the first line from <VirtualHost *:80> to <Web-Server-IP:80> to ensure Apache listens only on the desired interface with the higher metric.



```
 GNU nano 7.2                                          /etc/apache2/sites-available/000-default.conf *
<VirtualHost 192.168.1.138:80>
        # The ServerName directive sets the request scheme, hostname and port that
        # the server uses to identify itself. This is used when creating
        # redirection URLs. In the context of virtual hosts, the ServerName
        # specifies what hostname must appear in the request's Host: header to
        # match this virtual host. For the default virtual host (this file) this
        # value is not decisive as it is used as a last resort host regardless.
        # However, you must set it for any further virtual host explicitly.
        #ServerName www.example.com

        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/html

        # Available loglevels: trace8, ..., trace1, debug, info, notice, warn,
```

DVWA with Snort IDS Environment Setup:

- DVWA is deployed on an Ubuntu virtual machine.
- Web Server: Apache2 (apache2-bin version 2.4.58-1ubuntu8.5)
- Server IP & Port: 192.168.1.138 on port 80
- Snort is installed and configured on the same VM with custom rules.
- The VM is configured with two network interfaces:
  One in Bridged Adapter mode
  One in NAT Network mode

Attack Simulation:

To simulate real-world threats, various common web application attacks are launched from a Kali Linux attacker machine :

Attacker machine IPs : 192.168.1.100 and 192.168.1.119

- Port Scanning
- SQL Injection
- Cross-Site Scripting (XSS)
- Command Injection
- Path Traversal
- Reverse Shell Attempt (Reverse Connection Detection)
- DOS (Denial of Service) Attack Detection
- Brute Force Attack

To simulate real-world attack scenarios, we will use publicly available payloads sourced from trusted security forums, cheat sheets, and educational resources. These payloads are crafted to test for common web application vulnerabilities.

Objective:

The main objective is to trigger alerts in Snort when any of these attacks are attempted against DVWA.

Note : Snort is capable of much more , this project focuses on writing custom rules to demonstrate detection capabilities for a selected set of common threats.

Port Scanning:

Here, I aim to **detect unauthorized port scanning attempts**. The logic is simple: Only allow expected traffic on **TCP port 80 (HTTP)** — any attempt to connect to other ports will be flagged.

Rule Behavior:

Accepts incoming connections only to port **80, 443**.

Any connection attempt to other **non-allowed ports** is treated as **potential port scanning**.

Snort will **generate an alert** when this behavior is detected.

Testing Port Scanning Detection with Kali Linux :

To test my custom Snort rules for port scanning, I used Nmap from a Kali Linux machine to simulate an attack

TCP Port Scan Test:

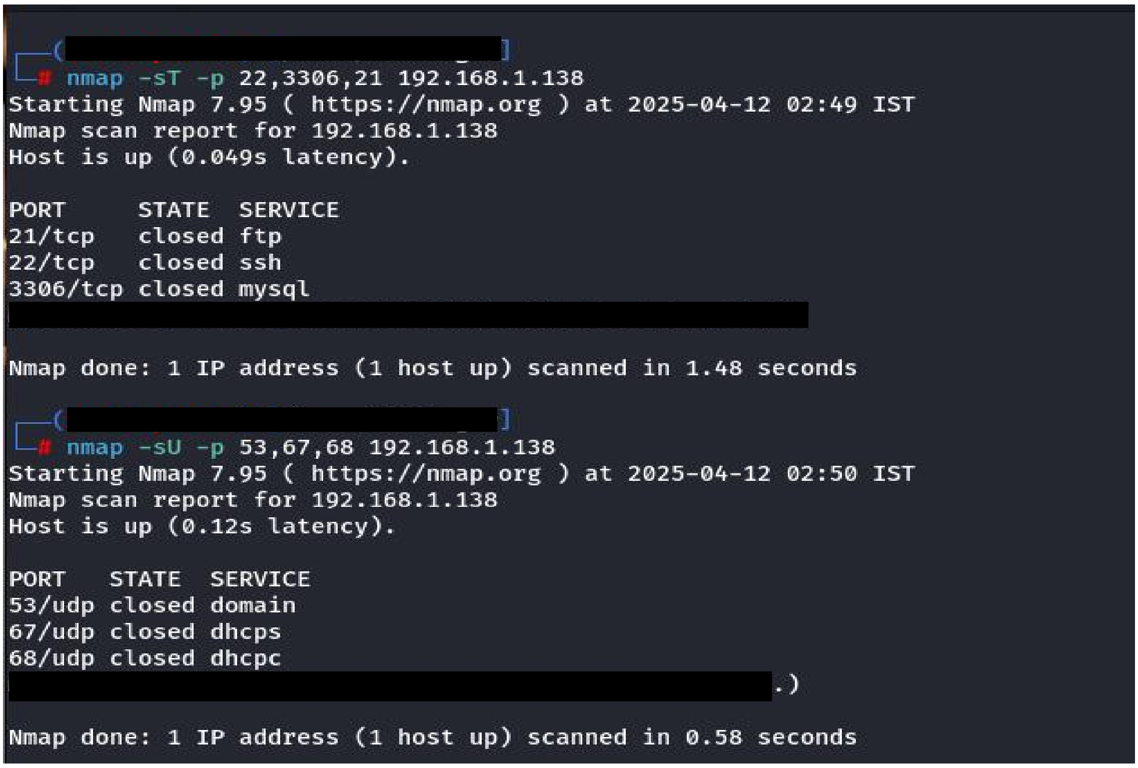
```
nmap -sT -p 22,21,3306  <Target-IP>
```

This command performs a TCP connect scan on ports 22 (SSH), 21 (FTP), and 3306 (MySQL).Since these ports are not allowed (only port 80 is permitted), Snort detected thisbehavior and triggered an alert.

UDP Port Scan Test:

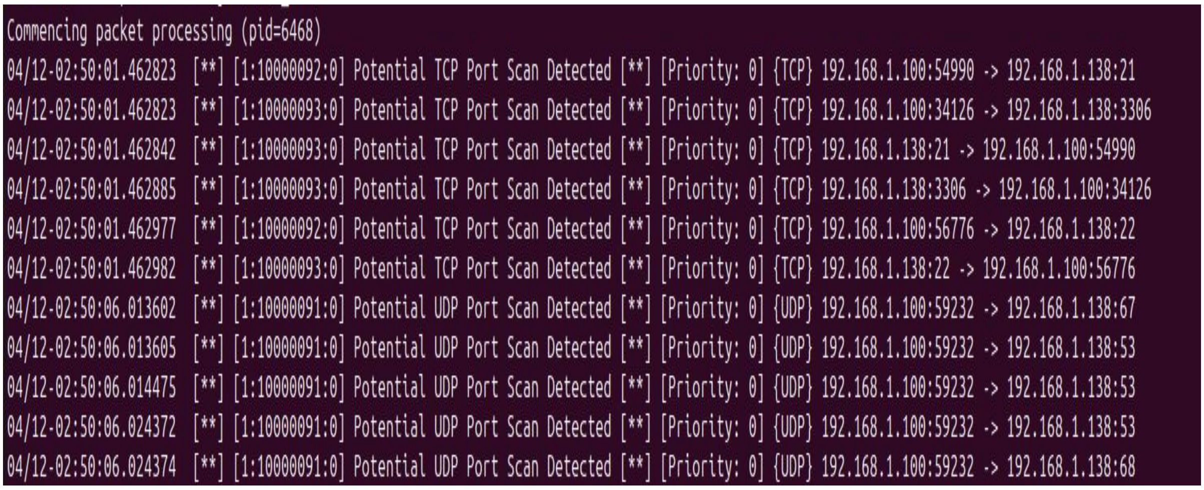
```
nmap -sU -p 53,67,65 <Target-IP>
```

This performs a **UDP scan** on DNS (53), DHCP (67), and an arbitrary port (65). Again, Snort successfully raised an alert.



```
┌──([redacted]]
└─# nmap -sT -p 22,3306,21 192.168.1.138
Starting Nmap 7.95 ( https://nmap.org ) at 2025-04-12 02:49 IST
Nmap scan report for 192.168.1.138
Host is up (0.049s latency).

PORT     STATE  SERVICE
21/tcp   closed ftp
22/tcp   closed ssh
3306/tcp closed mysql
[redacted]

Nmap done: 1 IP address (1 host up) scanned in 1.48 seconds

┌──([redacted]]
└─# nmap -sU -p 53,67,68 192.168.1.138
Starting Nmap 7.95 ( https://nmap.org ) at 2025-04-12 02:50 IST
Nmap scan report for 192.168.1.138
Host is up (0.12s latency).

PORT   STATE  SERVICE
53/udp closed domain
67/udp closed dhcps
68/udp closed dhcpc
[redacted].)

Nmap done: 1 IP address (1 host up) scanned in 0.58 seconds
```

TCP & UDP Port Scanning performed using Kali Linux



```
Commencing packet processing (pid=6468)
04/12-02:50:01.462823  [**] [1:10000092:0] Potential TCP Port Scan Detected [**] [Priority: 0] {TCP} 192.168.1.100:54990 -> 192.168.1.138:21
04/12-02:50:01.462823  [**] [1:10000093:0] Potential TCP Port Scan Detected [**] [Priority: 0] {TCP} 192.168.1.100:34126 -> 192.168.1.138:3306
04/12-02:50:01.462842  [**] [1:10000093:0] Potential TCP Port Scan Detected [**] [Priority: 0] {TCP} 192.168.1.138:21 -> 192.168.1.100:54990
04/12-02:50:01.462885  [**] [1:10000093:0] Potential TCP Port Scan Detected [**] [Priority: 0] {TCP} 192.168.1.138:3306 -> 192.168.1.100:34126
04/12-02:50:01.462977  [**] [1:10000092:0] Potential TCP Port Scan Detected [**] [Priority: 0] {TCP} 192.168.1.100:56776 -> 192.168.1.138:22
04/12-02:50:01.462982  [**] [1:10000093:0] Potential TCP Port Scan Detected [**] [Priority: 0] {TCP} 192.168.1.138:22 -> 192.168.1.100:56776
04/12-02:50:06.013602  [**] [1:10000091:0] Potential UDP Port Scan Detected [**] [Priority: 0] {UDP} 192.168.1.100:59232 -> 192.168.1.138:67
04/12-02:50:06.013605  [**] [1:10000091:0] Potential UDP Port Scan Detected [**] [Priority: 0] {UDP} 192.168.1.100:59232 -> 192.168.1.138:53
04/12-02:50:06.014475  [**] [1:10000091:0] Potential UDP Port Scan Detected [**] [Priority: 0] {UDP} 192.168.1.100:59232 -> 192.168.1.138:53
04/12-02:50:06.024372  [**] [1:10000091:0] Potential UDP Port Scan Detected [**] [Priority: 0] {UDP} 192.168.1.100:59232 -> 192.168.1.138:53
04/12-02:50:06.024374  [**] [1:10000091:0] Potential UDP Port Scan Detected [**] [Priority: 0] {UDP} 192.168.1.100:59232 -> 192.168.1.138:68
```

Snort alert triggered unauthorized port scanning attempts

Port Scanning Conclusion:

In this phase, I successfully demonstrated how Snort can detect unauthorized port scanning attempts.

By allowing only TCP traffic on port 80 and flagging connections to other ports, Snort was able to raise alerts for suspicious activity.

Furthermore, Snort rules can be made even more specific allowing you to detect or block traffic based on port numbers, IP addresses, or even TCP flags like SYN, ACK, or SYN-ACK. This flexibility makes Snort a powerful tool for detecting early-stage reconnaissance and intrusion attempts.

SQL Injection

To detect SQL Injection (SQLi) attempts, I created custom Snort rules that look for specific SQL related characters and keywords.

These include:

Special characters: ', ", --, =, +

SQL keywords: SELECT, CASE, OR, AND

Their URL-encoded equivalents (e.g., %27, %22, %2D%2D, etc.)

Whenever an attacker tries to inject these patterns into a request either directly or using encoded payload **Snort triggers an alert**, identifying the possible SQLi attempt.

Testing SQL Injection Detection with Common payload :

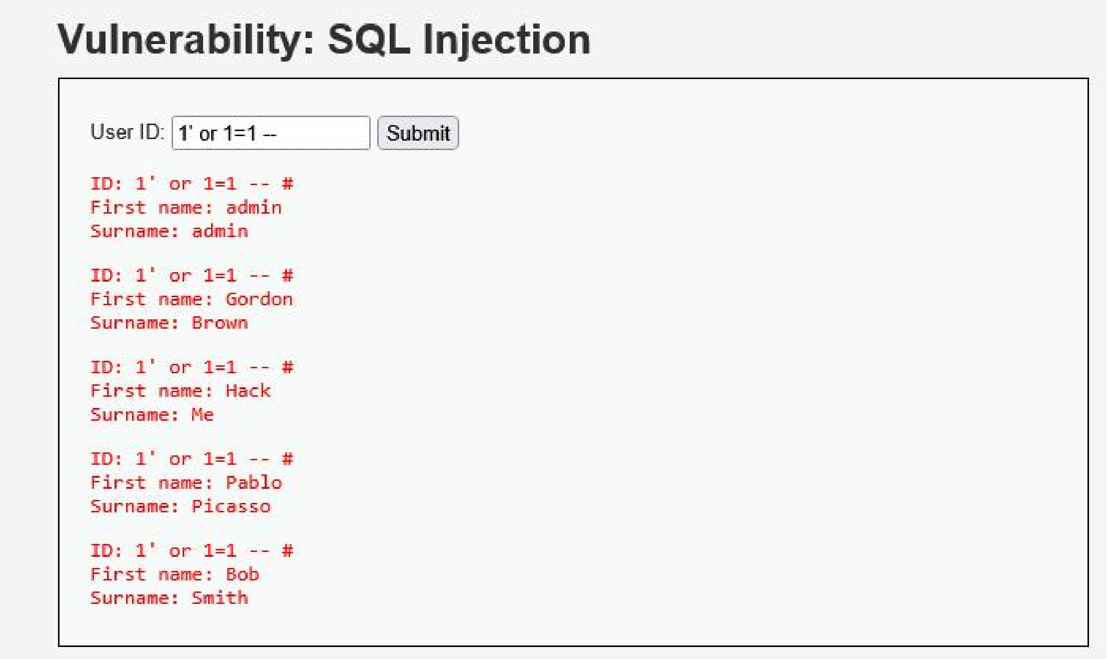
In this section, I demonstrate the detection of a classic SQL Injection payload:

' OR 1=1 --

This payload is widely used to bypass login forms or manipulate SQL queries. It works by injecting a condition that always evaluates to true, potentially granting unauthorized access.

I inserted this payload into a vulnerable input field on the DVWA application running on the target machine.

Thanks to the custom Snort rule I created, the IDS successfully detected the SQL Injection attempt and generated a real-time alert

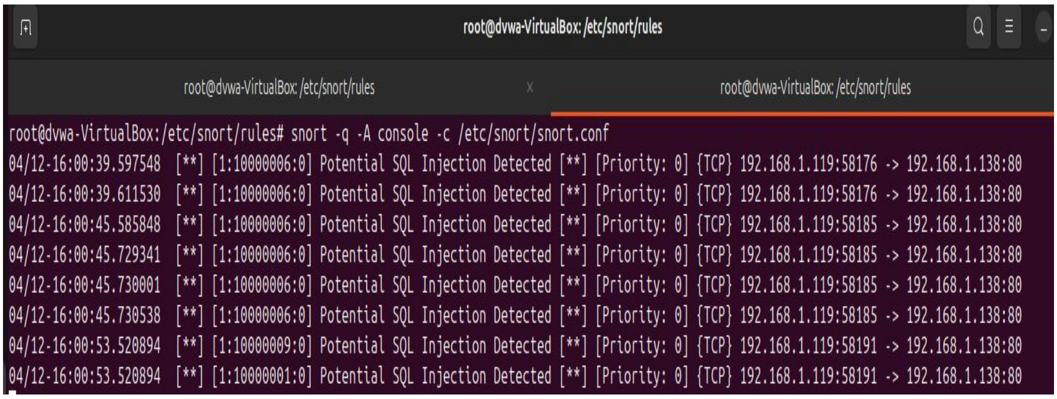


Vulnerability: SQL Injection

User ID: 1' or 1=1 --

```
ID: 1' or 1=1 -- #
First name: admin
Surname: admin

ID: 1' or 1=1 -- #
First name: Gordon
Surname: Brown

ID: 1' or 1=1 -- #
First name: Hack
Surname: Me

ID: 1' or 1=1 -- #
First name: Pablo
Surname: Picasso

ID: 1' or 1=1 -- #
First name: Bob
Surname: Smith
```

SQL Injection payload (' OR '1=1 --)



root@dvwa-VirtualBox: /etc/snort/rules
root@dvwa-VirtualBox: /etc/snort/rules
root@dvwa-VirtualBox: /etc/snort/rules

```
root@dvwa-VirtualBox:/etc/snort/rules# snort -q -A console -c /etc/snort/snort.conf
04/12-16:00:39.597548  [**] [1:10000006:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58176 -> 192.168.1.138:80
04/12-16:00:39.611530  [**] [1:10000006:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58176 -> 192.168.1.138:80
04/12-16:00:45.585848  [**] [1:10000006:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58185 -> 192.168.1.138:80
04/12-16:00:45.729341  [**] [1:10000006:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58185 -> 192.168.1.138:80
04/12-16:00:45.730001  [**] [1:10000006:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58185 -> 192.168.1.138:80
04/12-16:00:45.730538  [**] [1:10000006:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58185 -> 192.168.1.138:80
04/12-16:00:53.520894  [**] [1:10000009:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58191 -> 192.168.1.138:80
04/12-16:00:53.520894  [**] [1:10000001:0] Potential SQL Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:58191 -> 192.168.1.138:80
```

Snort alert triggered for SQL payload

SQL Injection Conclusion:

By matching common SQLi characters and keywords, Snort is able to detect injection attempts at the network level. That said, if we extend Snort's rule set to include more variations and complications often used in real-world SQLi payloads, it can improve its chances of catching more advanced attacks. Still, since Snort is signature-based, it can be bypassed by newer or obfuscated injection methods that don't match any predefined patterns.

it serves as an effective first-line defense when combined with secure coding practices on the application side.

Cross-Site Scripting (XSS)

To identify XSS attacks, I created custom Snort rules that inspect incoming requests for suspicious JavaScript and HTML elements. The rules are designed to catch both direct and encoded payloads commonly used in XSS attacks.

The detection logic includes:

Special characters: <, >, ;, (,)

JavaScript keywords: alert, prompt, console.log, img, script

URL-encoded equivalents: %3C, %3E, %28, %29, etc.

Whenever a request contains one or more of these patterns, Snort immediately generates an alert, signaling a potential Cross-Site Scripting attempt.

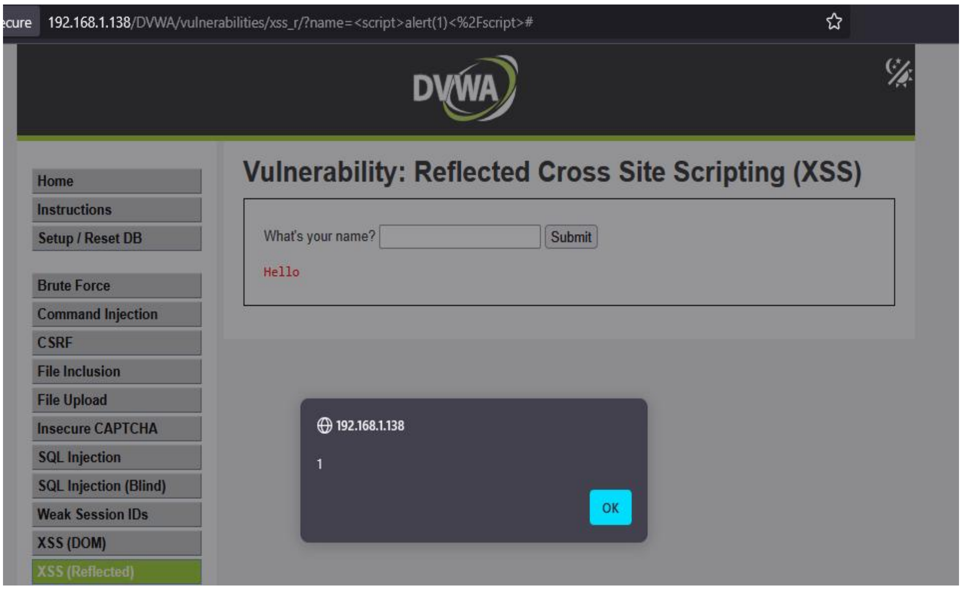
Testing XSS Detection with Common payload :

If a web application executes this code and displays an alert box, it indicates a vulnerability to XSS. In a real-world scenario, an attacker could modify this payload to steal session cookies, perform phishing attacks, or run malicious scripts.

<script>alert(1)</script>

I tested this by injecting the payload into a vulnerable input field on DVWA (Damn Vulnerable Web Application), which is running on the target machine.

Thanks to my custom Snort rule, the system successfully detected this attempt and triggered an alert, showcasing how even basic payloads can be flagged.



ecure 192.168.1.138/DVWA/vulnerabilities/xss_r/?name=<script>alert(1)<%2Fscript>#

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? Submit

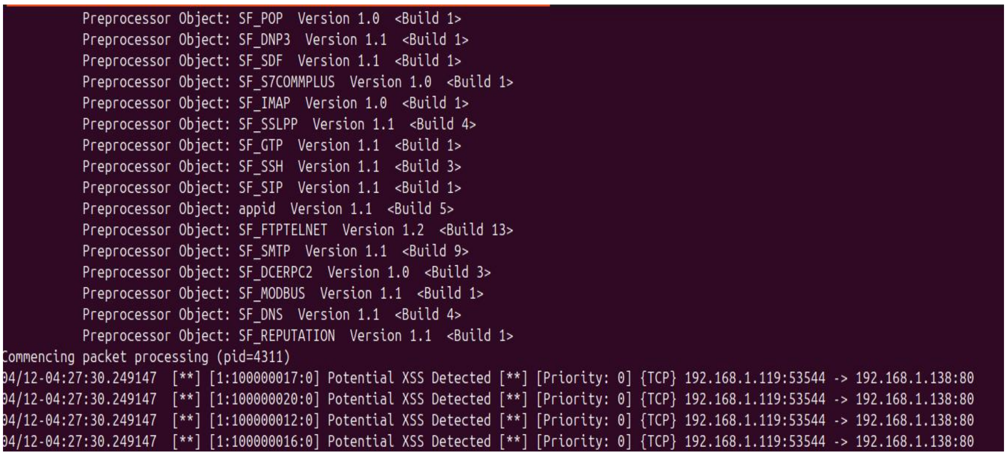
Hello

192.168.1.138

1

OK

XSS Payload Inserted into DVWA Input Field



```
          Preprocessor Object: SF_POP  Version 1.0  <Build 1>
          Preprocessor Object: SF_DNP3  Version 1.1  <Build 1>
          Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
          Preprocessor Object: SF_S7COMMPLUS  Version 1.0  <Build 1>
          Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
          Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
          Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
          Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
          Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
          Preprocessor Object: appid  Version 1.1  <Build 5>
          Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
          Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
          Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
          Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
          Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
          Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
Commencing packet processing (pid=4311)
04/12-04:27:30.249147  [**] [1:100000017:0] Potential XSS Detected [**] [Priority: 0] {TCP} 192.168.1.119:53544 -> 192.168.1.138:80
04/12-04:27:30.249147  [**] [1:100000020:0] Potential XSS Detected [**] [Priority: 0] {TCP} 192.168.1.119:53544 -> 192.168.1.138:80
04/12-04:27:30.249147  [**] [1:100000012:0] Potential XSS Detected [**] [Priority: 0] {TCP} 192.168.1.119:53544 -> 192.168.1.138:80
04/12-04:27:30.249147  [**] [1:100000016:0] Potential XSS Detected [**] [Priority: 0] {TCP} 192.168.1.119:53544 -> 192.168.1.138:80
```

Snort Alert Triggered by XSS

Conclusion: XSS Detection Phase

In this phase, I successfully demonstrated how custom Snort rules can be used to detect Cross-Site Scripting (XSS) attempts in a web application.

By targeting key JavaScript keywords, HTML special characters, and their URL-encoded versions, Snort was able to detect some basic payloads like

<script>alert(1)</script>.

With the help of Snort I was able to effectively detect an early stage attacks.

Command Injection

To identify Command Injection attempts, I developed custom Snort rules that analyze incoming requests for suspicious elements and system commands typically used in such attacks.

Detection Logic Includes:

Special Characters: |,&&, &

Common System Commands: ls, ping, whoami, echo, pwd

URL-Encoded Equivalents: such as %7C (for |), %26 (for &), etc.

These rules are crafted to detect both direct and encoded payloads, increasing the accuracy of detection.

Whenever a request contains one or more of these patterns, Snort immediately triggers an alert, flagging a potential Command Injection attempt.

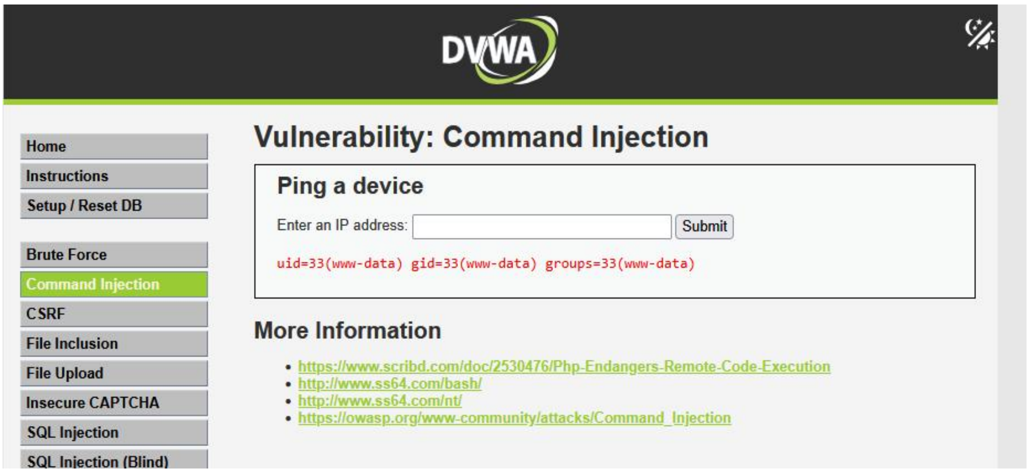
Testing Command Injection Detection with Common Payload:

In this phase, I tested the effectiveness of my custom Snort rule by injecting a simple yet powerful command:

id

This command is commonly used in Command Injection attacks to retrieve user identity information from the target system. I entered the payload into a vulnerable input field in DVWA. Upon submission, Snort successfully detected the presence of the id command and generated an alert, indicating a potential command injection attempt.

This demonstrates how well Snort can identify even basic system commands when they're used in suspicious contexts.



Vulnerability: Command Injection

Ping a device

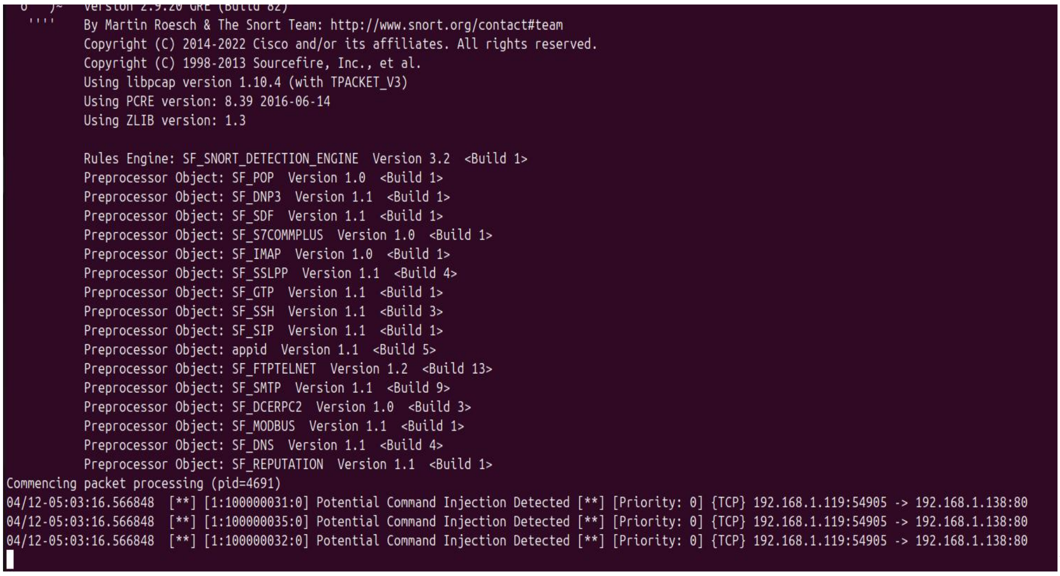
Enter an IP address:

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

More Information

- https://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
- http://www.ss64.com/bash/
- http://www.ss64.com/nt/
- https://owasp.org/www-community/attacks/Command_Injection

Command Injection payload (id) Injected into DVWA



```
  o" )~   Version 2.9.20 GRE (Build 82)
   ''''    By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
           Copyright (C) 2014-2022 Cisco and/or its affiliates. All rights reserved.
           Copyright (C) 1998-2013 Sourcefire, Inc., et al.
           Using libpcap version 1.10.4 (with TPACKET_V3)
           Using PCRE version: 8.39 2016-06-14
           Using ZLIB version: 1.3

           Rules Engine: SF_SNORT_DETECTION_ENGINE  Version 3.2  <Build 1>
           Preprocessor Object: SF_POP  Version 1.0  <Build 1>
           Preprocessor Object: SF_DNP3  Version 1.1  <Build 1>
           Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
           Preprocessor Object: SF_S7COMMPLUS  Version 1.0  <Build 1>
           Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
           Preprocessor Object: appid  Version 1.1  <Build 5>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
           Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
           Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
           Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
           Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
           Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
Commencing packet processing (pid=4691)
04/12-05:03:16.566848  [**] [1:100000031:0] Potential Command Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:54905 -> 192.168.1.138:80
04/12-05:03:16.566848  [**] [1:100000035:0] Potential Command Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:54905 -> 192.168.1.138:80
04/12-05:03:16.566848  [**] [1:100000032:0] Potential Command Injection Detected [**] [Priority: 0] {TCP} 192.168.1.119:54905 -> 192.168.1.138:80
```

rt

Alert Triggered by XSS Attempt

Conclusion: Command Injection Detection

In this phase, I successfully demonstrated how Snort can detect Command Injection attempts by analyzing suspicious patterns and system command usage within HTTP requests.

By crafting custom rules that monitor for:

Special characters like | and &

Common system commands such as id, whoami, ls, and ping

Their URL-encoded equivalent

Snort was able to effectively trigger alerts whenever these patterns appeared, indicating a potential command injection attempt.

This showcases how Snort can be used to build a lightweight, flexible layer of detection and prevention, even for sophisticated input based attacks.

Path Traversal

To identify Path Traversal attempts, I developed custom Snort rules that analyze incoming requests for suspicious elements and system commands typically used in such attacks.

Detection Logic Includes:

Special Characters: ../,/../

Common System Commands: /etc/passwd, /etc/hosts,/

Null value : %00

URL-Encoded Equivalents: ..%2F,..%252F,%2F..%2F,%252F..%252F

These rules are crafted to detect both direct and encoded payloads, increasing the accuracy of detection.

Whenever a request contains one or more of these patterns, Snort immediately triggers an alert that potential Path Traversal attempt.

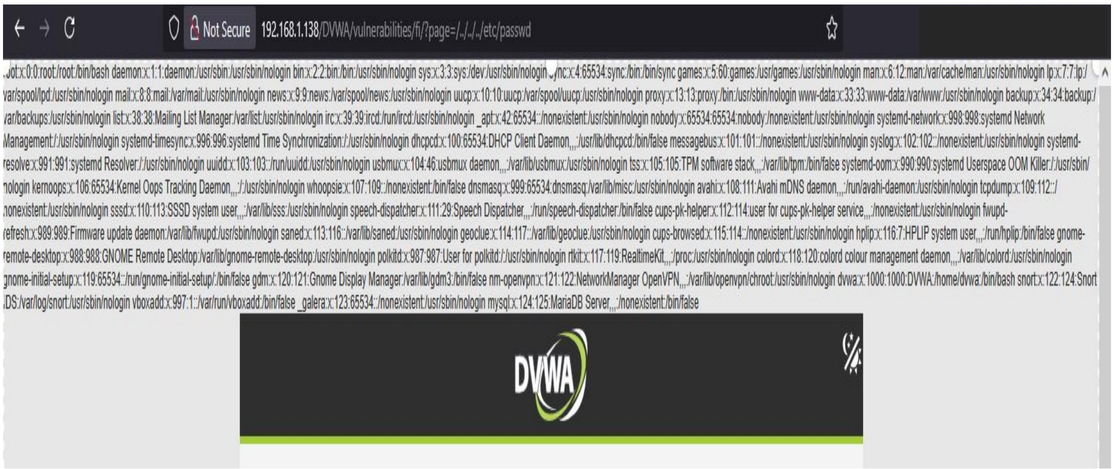
Testing Path Traversal Detection with Common Payload:

In this test, I used a classic Path Traversal payload:

/../../../../etc/passwd

This payload attempts to access the system file /etc/passwd by navigating up the directory structure. I entered this into a vulnerable input field within the DVWA application.

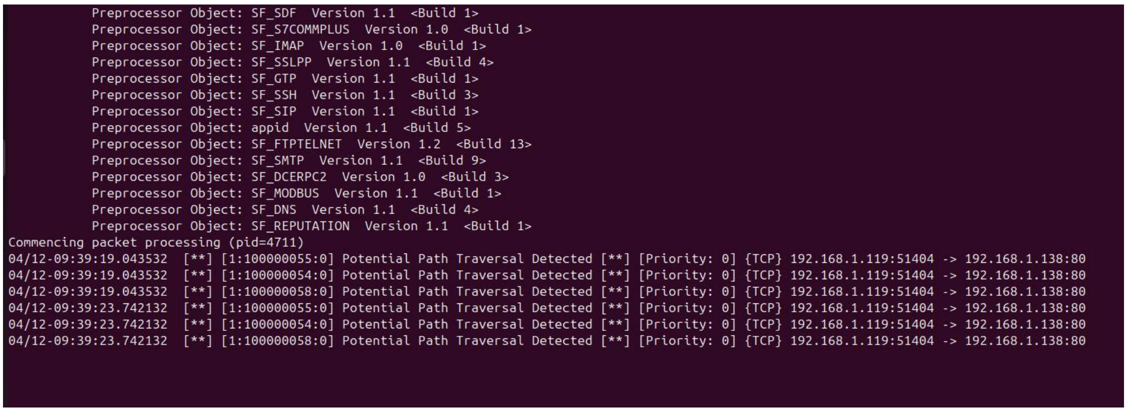
Snort detected the malicious pattern and triggered an alert, confirming that the custom rule for Path Traversal is working as intended.



```
192.168.1.138/DVWA/vulnerabilities/fi/?page=/../../../etc/passwd
```

```
ot:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin ync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/
var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/
var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin _apt:x:42:65534::/nonexistent:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin systemd-network:x:998:998:systemd Network
Management:/:/usr/sbin/nologin systemd-timesync:x:996:996:systemd Time Synchronization:/:/usr/sbin/nologin dhcpcd:x:100:65534:DHCP Client Daemon,,,:/usr/lib/dhcpcd:/bin/false messagebus:x:101:101::/nonexistent:/usr/sbin/nologin syslog:x:102:102::/nonexistent:/usr/sbin/nologin systemd-
resolve:x:991:991:systemd Resolver:/:/usr/sbin/nologin uuidd:x:103:103::/run/uuidd:/usr/sbin/nologin usbmux:x:104:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin tss:x:105:105:TPM software stack,,,:/var/lib/tpm:/bin/false systemd-oom:x:990:990:systemd Userspace OOM Killer:/:/usr/sbin/
nologin kernoops:x:106:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin whoopsie:x:107:109::/nonexistent:/bin/false dnsmasq:x:999:65534:dnsmasq:/var/lib/misc:/usr/sbin/nologin avahi:x:108:111:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin tcpdump:x:109:112::/
nonexistent:/usr/sbin/nologin sssd:x:110:113:SSSD system user,,,:/var/lib/sss:/usr/sbin/nologin speech-dispatcher:x:111:29:Speech Dispatcher,,,:/run/speech-dispatcher:/bin/false cups-pk-helper:x:112:114:user for cups-pk-helper service,,,:/nonexistent:/usr/sbin/nologin fwupd-
refresh:x:989:989:Firmware update daemon:/var/lib/fwupd:/usr/sbin/nologin saned:x:113:116::/var/lib/saned:/usr/sbin/nologin geoclue:x:114:117::/var/lib/geoclue:/usr/sbin/nologin cups-browsed:x:115:114::/nonexistent:/usr/sbin/nologin hplip:x:116:7:HPLIP system user,,,:/run/hplip:/bin/false gnome-
remote-desktop:x:988:988:GNOME Remote Desktop:/var/lib/gnome-remote-desktop:/usr/sbin/nologin polkitd:x:987:987:User for polkitd:/:/usr/sbin/nologin rtkit:x:117:119:RealtimeKit,,,:/proc:/usr/sbin/nologin colord:x:118:120:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
gnome-initial-setup:x:119:65534::/run/gnome-initial-setup/:/bin/false gdm:x:120:121:Gnome Display Manager:/var/lib/gdm3:/bin/false nm-openvpn:x:121:122:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin dvwa:x:1000:1000:DVWA:/home/dvwa:/bin/bash snort:x:122:124:Snort
IDS:/var/log/snort:/usr/sbin/nologin vboxadd:x:997:1::/var/run/vboxadd:/bin/false _galera:x:123:65534::/nonexistent:/usr/sbin/nologin mysql:x:124:125:MariaDB Server,,,:/nonexistent:/bin/false
```

Payload Injection Attempt (/../../../etc/passwd)



```
           Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
           Preprocessor Object: SF_S7COMMPLUS  Version 1.0  <Build 1>
           Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
           Preprocessor Object: appid  Version 1.1  <Build 5>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
           Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
           Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
           Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
           Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
           Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
Commencing packet processing (pid=4711)
04/12-09:39:19.043532  [**] [1:100000055:0] Potential Path Traversal Detected [**] [Priority: 0] {TCP} 192.168.1.119:51404 -> 192.168.1.138:80
04/12-09:39:19.043532  [**] [1:100000054:0] Potential Path Traversal Detected [**] [Priority: 0] {TCP} 192.168.1.119:51404 -> 192.168.1.138:80
04/12-09:39:19.043532  [**] [1:100000058:0] Potential Path Traversal Detected [**] [Priority: 0] {TCP} 192.168.1.119:51404 -> 192.168.1.138:80
04/12-09:39:23.742132  [**] [1:100000055:0] Potential Path Traversal Detected [**] [Priority: 0] {TCP} 192.168.1.119:51404 -> 192.168.1.138:80
04/12-09:39:23.742132  [**] [1:100000054:0] Potential Path Traversal Detected [**] [Priority: 0] {TCP} 192.168.1.119:51404 -> 192.168.1.138:80
04/12-09:39:23.742132  [**] [1:100000058:0] Potential Path Traversal Detected [**] [Priority: 0] {TCP} 192.168.1.119:51404 -> 192.168.1.138:80
```

Snort alert triggered for Path Traversal

Conclusion: Path Traversal Detection

In this phase, I successfully demonstrated how Snort can detect Path Traversal attempts by analyzing suspicious patterns and system command usage within HTTP requests.

This shows Snort was able to effectively trigger alerts whenever these patterns appeared, indicating a potential Path Traversal attempt.

Reverse Shell Attempt (Reverse Connection Detection)

In this scenario, I tested Snort's ability to detect a reverse shell attempt by simulating an attack using the PentestMonkey PHP reverse shell script.

Setup Details:

A **listener** was started on port 9001 using Kali Linux (`nc -lvp 9001`).

The reverse shell PHP script was uploaded to the DVWA application via the File Upload vulnerability section.

Once executed, the script initiated a reverse connection from the DVWA server to the attacker machine, attempting to give shell access.

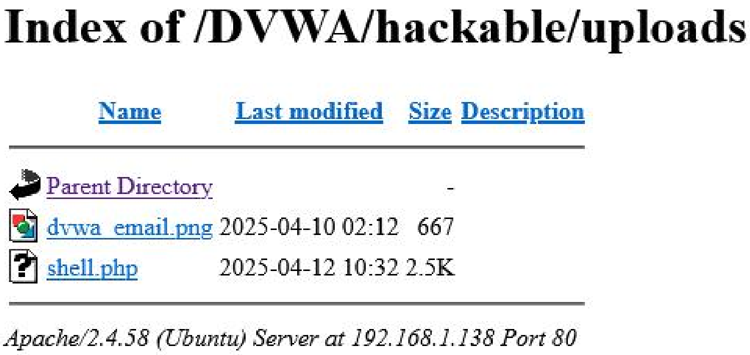
Detected Reverse Shell connection with Common Payload:

Payload content like /bin/bash, often seen in reverse shell behavior, which provides a full-featured interactive shell with.

Snort successfully detected and alerted the reverse shell activity, validating the effectiveness of the custom detection rules.

This indicates that even basic reverse shell patterns can be effectively flagged using lightweight, custom Snort rules.

To avoid conflicts between reverse shell and port scan alerts, I ran Snort with reverse shell rules alone in a separate tab, allowing for more focused detection of reverse shell activity without interference from port scan alerts. This approach isolates the two types of traffic, minimizing false positives.

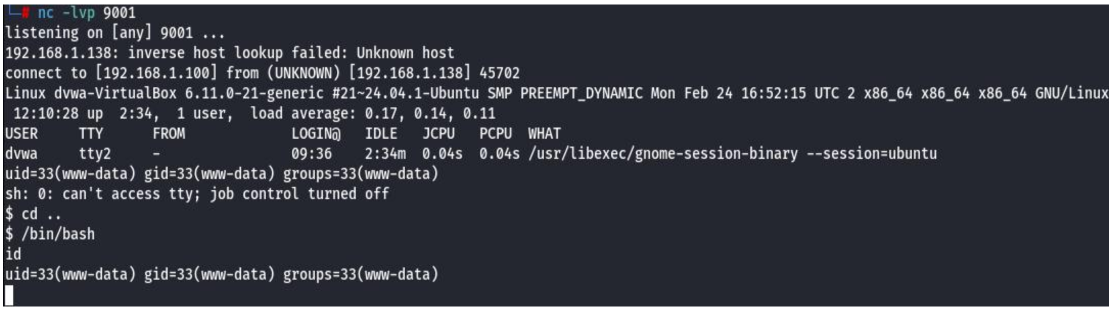


Index of /DVWA/hackable/uploads

Name	Last modified	Size	Description
Parent Directory		-	
dvwa_email.png	2025-04-10 02:12	667	
shell.php	2025-04-12 10:32	2.5K	

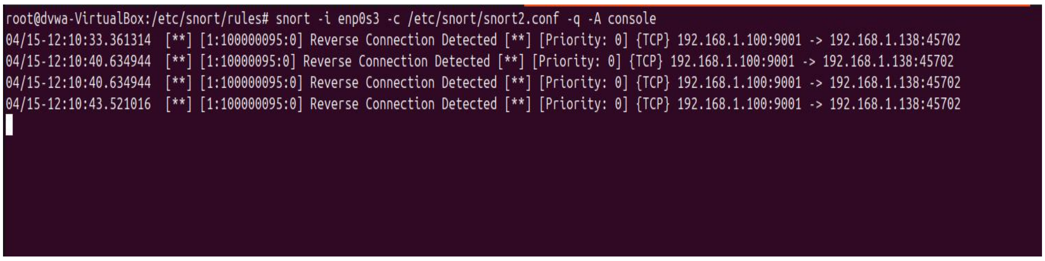
Apache/2.4.58 (Ubuntu) Server at 192.168.1.138 Port 80

Uploaded Revershell



```
└─# nc -lvp 9001
listening on [any] 9001 ...
192.168.1.138: inverse host lookup failed: Unknown host
connect to [192.168.1.100] from (UNKNOWN) [192.168.1.138] 45702
Linux dvwa-VirtualBox 6.11.0-21-generic #21~24.04.1-Ubuntu SMP PREEMPT_DYNAMIC Mon Feb 24 16:52:15 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
 12:10:28 up  2:34,  1 user,  load average: 0.17, 0.14, 0.11
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
dvwa     tty2     -                09:36    2:34m  0.04s  0.04s /usr/libexec/gnome-session-binary --session=ubuntu
uid=33(www-data) gid=33(www-data) groups=33(www-data)
sh: 0: can't access tty; job control turned off
$ cd ..
$ /bin/bash
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Reverse Connection from webserver



```
root@dvwa-VirtualBox:/etc/snort/rules# snort -i enp0s3 -c /etc/snort/snort2.conf -q -A console
04/15-12:10:33.361314  [**] [1:100000095:0] Reverse Connection Detected [**] [Priority: 0] {TCP} 192.168.1.100:9001 -> 192.168.1.138:45702
04/15-12:10:40.634944  [**] [1:10000095:0] Reverse Connection Detected [**] [Priority: 0] {TCP} 192.168.1.100:9001 -> 192.168.1.138:45702
04/15-12:10:40.634944  [**] [1:100000095:0] Reverse Connection Detected [**] [Priority: 0] {TCP} 192.168.1.100:9001 -> 192.168.1.138:45702
04/15-12:10:43.521016  [**] [1:100000095:0] Reverse Connection Detected [**] [Priority: 0] {TCP} 192.168.1.100:9001 -> 192.168.1.138:45702
```

Snort alert triggered for Reverse shell command Execution

Conclusion – Reverse Shell Detection

In this phase, I demonstrated how Snort can effectively detect reverse shell attempts using custom rules that analyze suspicious payloads, such as /bin/bash, commonly associated with reverse shell behavior. The alert generation confirmed that the network-level monitoring and detection mechanisms are capable of identifying even minimalistic reverse shell patterns, enhancing overall system security.

DoS (Denial of Service) Attack Detection

To identify potential Denial of Service (DoS) attacks, I developed custom Snort rules that monitor abnormal traffic rates targeting the web server. These rules are designed to detect both volume-based floods and protocol-specific abuse.

To better understand Snort's behavior under high traffic, I categorized 1000 packets per second as a "Flood" and 2000 packets per second as a "Very Intense Flood." These thresholds helped simulate different attack intensities and analyze Snort's detection and response under stress.

Detection Logic Includes:

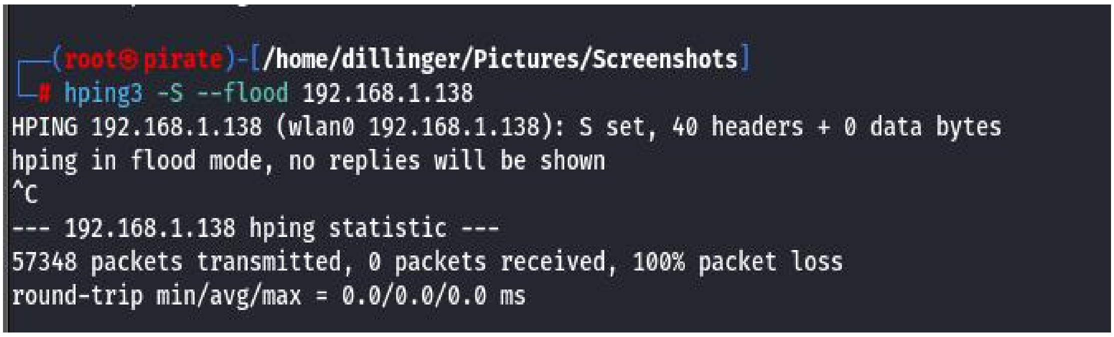
Rate Limiting Logic: Uses detection filter to track packet counts per source IP in a specific time
(e.g., 1000 packets in 1 second).

TCP Flood Detection: Alerts on rapid bursts of incoming TCP packets , helping to identify abnormal or malicious traffic beyond regular web service activity.

ICMP Flood Detection: Captures excessive ping (echo-request) traffic.

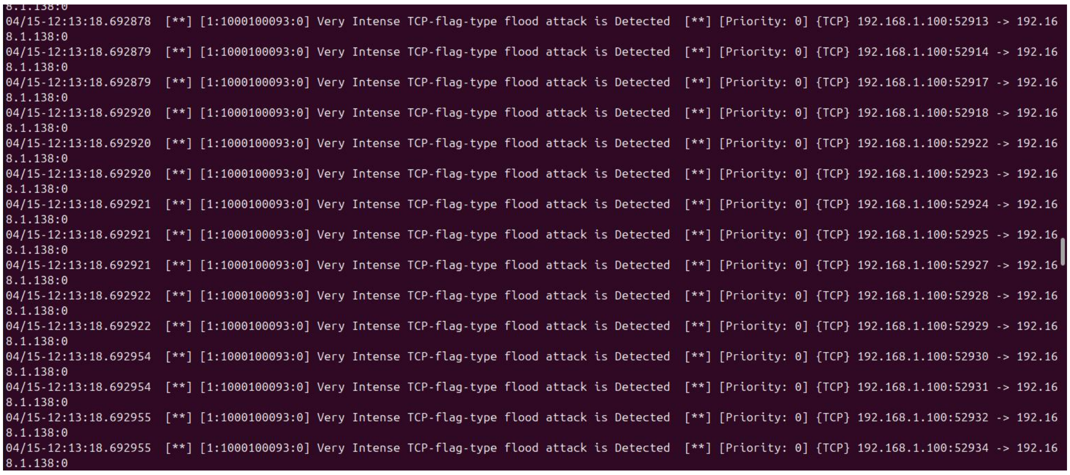
UDP Flood Detection: Alerts on rapid bursts of incoming UDP packets , helping to identify abnormal or malicious traffic beyond regular web service activity.

These rules help differentiate between normal traffic bursts and suspicious DoS attempts, with specific thresholds set to avoid false positives.



```
┌──(root㉿pirate)-[/home/dillinger/Pictures/Screenshots]
└─# hping3 -S --flood 192.168.1.138
HPING 192.168.1.138 (wlan0 192.168.1.138): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.1.138 hping statistic ---
57348 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

TCP SYN Flood triggered using hping3



```
8.1.138:0
04/15-12:13:18.692878  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52913 -> 192.16
8.1.138:0
04/15-12:13:18.692879  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52914 -> 192.16
8.1.138:0
04/15-12:13:18.692879  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52917 -> 192.16
8.1.138:0
04/15-12:13:18.692920  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52918 -> 192.16
8.1.138:0
04/15-12:13:18.692920  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52922 -> 192.16
8.1.138:0
04/15-12:13:18.692920  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52923 -> 192.16
8.1.138:0
04/15-12:13:18.692921  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52924 -> 192.16
8.1.138:0
04/15-12:13:18.692921  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52925 -> 192.16
8.1.138:0
04/15-12:13:18.692921  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52927 -> 192.16
8.1.138:0
04/15-12:13:18.692922  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52928 -> 192.16
8.1.138:0
04/15-12:13:18.692922  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52929 -> 192.16
8.1.138:0
04/15-12:13:18.692954  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52930 -> 192.16
8.1.138:0
04/15-12:13:18.692954  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52931 -> 192.16
8.1.138:0
04/15-12:13:18.692955  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52932 -> 192.16
8.1.138:0
04/15-12:13:18.692955  [**] [1:1000100093:0] Very Intense TCP-flag-type flood attack is Detected  [**] [Priority: 0] {TCP} 192.168.1.100:52934 -> 192.16
8.1.138:0
```

Snort detected TCP flood attack based on high packet rate

Conclusion - DOS

The implemented Snort rules effectively detected various types of DoS attempts by analyzing traffic rate patterns and specific protocol behaviors. By fine-tuning detection filters, I was able to reduce false positives while maintaining strong coverage against volume-based attacks.

Brute Force Attack:

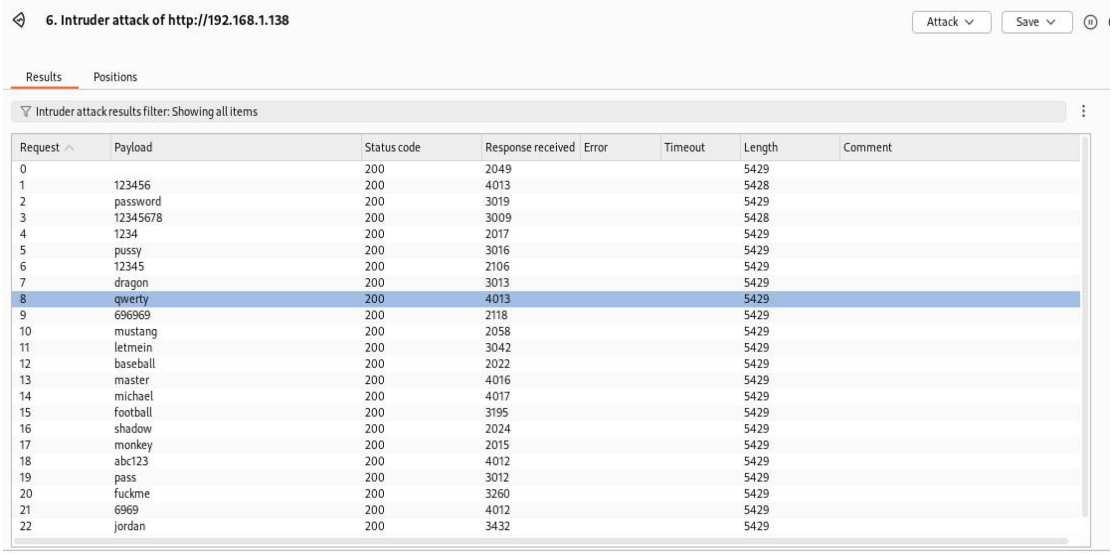
To detect potential Brute Force attacks against the web application's login page, I implemented custom Snort rules that monitor repeated login attempts from the same source IP within a short time frame. These rules are designed to detect attempts to guess credentials through repeated HTTP requests.

Detection Logic Includes:

Rate Limiting Logic: Uses detection_filter to track HTTP POST attempts to the login page from a single IP. For example, more than **15 attempts in 20 seconds** raises an alert.

Login Form Pattern Match: The rule looks for requests to /DVWA/login.php and checks for login credentials in the HTTP client body (e.g., presence of the parameter "Login=").

These rules help to successfully detected the brute force pattern during testing using automated login scripts. The alert was triggered as expected once the defined threshold was met.

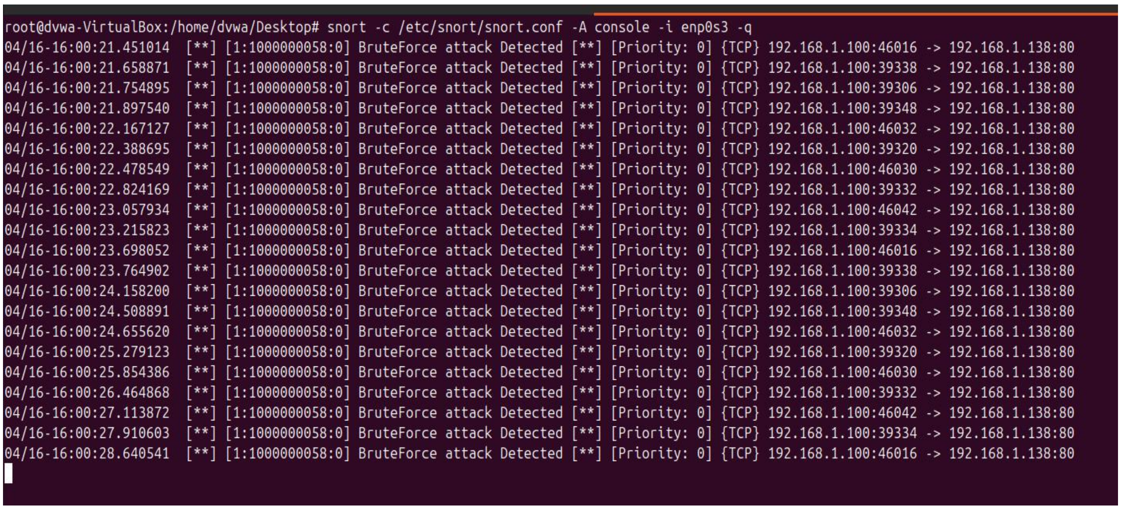


6. Intruder attack of http://192.168.1.138

Intruder attack results filter: Showing all items

Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
0		200	2049			5429	
1	123456	200	4013			5428	
2	password	200	3019			5429	
3	12345678	200	3009			5428	
4	1234	200	2017			5429	
5	pussy	200	3016			5429	
6	12345	200	2106			5429	
7	dragon	200	3013			5429	
8	qwerty	200	4013			5429	
9	696969	200	2118			5429	
10	mustang	200	2058			5429	
11	letmein	200	3042			5429	
12	baseball	200	2022			5429	
13	master	200	4016			5429	
14	michael	200	4017			5429	
15	football	200	3195			5429	
16	shadow	200	2024			5429	
17	monkey	200	2015			5429	
18	abc123	200	4012			5429	
19	pass	200	3012			5429	
20	fuckme	200	3260			5429	
21	6969	200	4012			5429	
22	jordan	200	3432			5429	

Brute Force Attack using Burp Suite



```
root@dvwa-VirtualBox:/home/dvwa/Desktop# snort -c /etc/snort/snort.conf -A console -i enp0s3 -q
04/16-16:00:21.451014  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46016 -> 192.168.1.138:80
04/16-16:00:21.658871  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39338 -> 192.168.1.138:80
04/16-16:00:21.754895  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39306 -> 192.168.1.138:80
04/16-16:00:21.897540  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39348 -> 192.168.1.138:80
04/16-16:00:22.167127  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46032 -> 192.168.1.138:80
04/16-16:00:22.388695  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39320 -> 192.168.1.138:80
04/16-16:00:22.478549  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46030 -> 192.168.1.138:80
04/16-16:00:22.824169  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39332 -> 192.168.1.138:80
04/16-16:00:23.057934  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46042 -> 192.168.1.138:80
04/16-16:00:23.215823  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39334 -> 192.168.1.138:80
04/16-16:00:23.698052  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46016 -> 192.168.1.138:80
04/16-16:00:23.764902  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39338 -> 192.168.1.138:80
04/16-16:00:24.158200  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39306 -> 192.168.1.138:80
04/16-16:00:24.508891  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39348 -> 192.168.1.138:80
04/16-16:00:24.655620  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46032 -> 192.168.1.138:80
04/16-16:00:25.279123  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39320 -> 192.168.1.138:80
04/16-16:00:25.854386  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46030 -> 192.168.1.138:80
04/16-16:00:26.464868  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39332 -> 192.168.1.138:80
04/16-16:00:27.113872  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46042 -> 192.168.1.138:80
04/16-16:00:27.910603  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:39334 -> 192.168.1.138:80
04/16-16:00:28.640541  [**] [1:1000000058:0] BruteForce attack Detected [**] [Priority: 0] {TCP} 192.168.1.100:46016 -> 192.168.1.138:80
```

Snort detected Brute force attack based on high packet rate

Conclusion - Brute Force Attack

This test confirms Snort’s strength in detecting web application-layer brute force attempts. With fine-tuned rule customization, it's possible to catch malicious login floods early. This helps to secure weak points like login forms from automated attacks and highlights Snort’s adaptability in defending against multiple threats.

Conclusion: Enhancing Security with Snort IDS

Through this project, I tested a variety of common attacks using minimal payloads, and observed how effectively Snort can detect malicious traffic. Snort proved to be a highly powerful Intrusion Detection System (IDS), especially when fine tuned with custom rules tailored to specific attack patterns.

One of Snort’s greatest strengths lies in its flexibility we can write and customize rules as we needed, enabling detection capabilities based on project requirements.

We can even use common payloads found on platforms like GitHub, and directly import them into Snort rules via the content keyword to expand detection coverage easily.

Although this project demonstrates only a few basic examples, it highlights the potential of Snort in real-world network defense.

Additionally, running Snort in multiple instances allowed me to isolate specific rule sets and focus on particular attacks without interference, improving detection accuracy and reducing noise.

While Snort and other IDS/IPS or firewall solutions play a critical role in detecting and preventing many types of attacks, they alone are not enough to secure a system. Effective security requires a combination of secure coding practices, proper system and network configuration, and correct implementation of security controls.

Additionally, some vulnerabilities such as Insecure Direct Object References (IDOR) involve logical flaws or access control issues that cannot be detected by traditional IDS/IPS, since they don’t involve malicious payloads or suspicious traffic patterns.
Therefore, a layered security approach is essential, combining network-level detection with secure application development and regular testing.