**OWASP ZAP Security Assessment Report**

**Website Security Scan Overview**

A security scan was conducted on the website using the OWASP ZAP tool. The following vulnerabilities were detected:

**Summary of Alerts**

|  |  |
| --- | --- |
| Alerts | Total |
| High Priority Alerts | 0 |
| Medium Priority Alerts | 4 |
| Low Priority Alerts | 3 |
| Informational Priority Alerts | 3 |

**Medium Priority Alerts**

**1. Absence of Anti-CSRF Tokens**

* **URL:** http://127.0.0.1:5000/login
* **Risk Level:** Medium
* **Confidence:** Low
* **Description:** No Anti-CSRF tokens were found in an HTML submission form. This makes the application vulnerable to Cross-Site Request Forgery (CSRF) attacks, where malicious actors can force users to perform unintended actions.
* **Code Issue:**
  + <form action="https://validator.w3.org/check" method="post">
* **Recommended Fix:** Implement CSRF protection by using a CSRF token in forms.

**2. HTTP to HTTPS Insecure Transition in Form Post**

* **URL:** http://127.0.0.1:5000/login
* **Risk Level:** Medium
* **Confidence:** Medium
* **Description:** A form is posted from an HTTP page to an HTTPS destination. This can be exploited via a Man-in-the-Middle (MITM) attack, replacing or altering the form submission.
* **Recommended Fix:** Enforce HTTPS site-wide and ensure all forms are submitted securely.

**3. Missing Anti-clickjacking Header**

* **URL:** http://127.0.0.1:5000/login
* **Risk Level:** Medium
* **Confidence:** Medium
* **Description:** The response does not include X-Frame-Options or Content-Security-Policy headers to prevent Clickjacking attacks.
* **Recommended Fix:** Implement the following HTTP headers:
* @app.after\_request
* def set\_headers(response):
* response.headers["X-Frame-Options"] = "DENY"
* response.headers["Content-Security-Policy"] = "frame-ancestors 'none';"

return response

**4. Content Security Policy (CSP) Header Not Set**

* **URL:** http://127.0.0.1:5000/robots.txt
* **Risk Level:** Medium
* **Confidence:** High
* **Description:** The absence of a CSP header increases the risk of Cross-Site Scripting (XSS) and data injection attacks.
* **Recommended Fix:** Implement CSP headers with restrictive policies:

response.headers["Content-Security-Policy"] = "default-src 'self'; script-src 'self';"

**Low Priority Alerts**

**1. Cookie Without SameSite Attribute**

* **URL:** http://127.0.0.1:5000/register
* **Risk Level:** Low
* **Confidence:** Medium
* **Description:** Cookies are missing the SameSite attribute, making them susceptible to CSRF attacks.
* **Recommended Fix:**

response.set\_cookie("session", value, samesite="Strict", secure=True, httponly=True)

**2. Server Leaks Version Information via 'Server' Header**

* **URL:** http://127.0.0.1:5000/static/styles.css
* **Risk Level:** Low
* **Confidence:** High
* **Description:** The web server exposes version information, which can aid attackers in identifying vulnerabilities.
* **Recommended Fix:** Disable the Server header in Flask:
* @app.after\_request
* def remove\_server\_header(response):
* response.headers.pop("Server", None)

return response

**3. X-Content-Type-Options Header Missing**

* **URL:** http://127.0.0.1:5000/static/styles.css
* **Risk Level:** Low
* **Confidence:** Medium
* **Description:** The absence of X-Content-Type-Options: nosniff allows MIME-type sniffing, which can lead to content security risks.
* **Recommended Fix:**

response.headers["X-Content-Type-Options"] = "nosniff"

**Cross-Site Scripting (XSS) Vulnerability Check**

* **Findings:** No XSS vulnerabilities detected. User input is sanitized before rendering.
* **Testing Payload:** <script>alert('XSS')</script>
* **Application Response:** Redirects to an apology template (apology.html).

**SQL Injection Vulnerability Check**

* **Findings:** The application uses parameterized queries, mitigating SQL injection risks.
* **Example Code Review:**

rows = db.execute("SELECT \* FROM users WHERE username = ?", request.form.get("username"))

* **Recommendations:** Continue using parameterized queries and avoid string concatenation in SQL.

**Authentication and Password Security Review**

* **Findings:** Passwords are securely hashed using werkzeug.security.
* **Example Code:**
* from werkzeug.security import check\_password\_hash, generate\_password\_hash

hashPass = generate\_password\_hash(password)

* **Recommendation:** Consider implementing Multi-Factor Authentication (MFA) for enhanced security.

**MFA Implementation Steps**

1. Install necessary packages: pip install pyotp qrcode pillow
2. Generate MFA secret per user:
3. secret = pyotp.random\_base32()

db.execute("UPDATE users SET mfa\_secret = ? WHERE id = ?", secret, user\_id)

1. Validate MFA during login:
2. totp = pyotp.TOTP(user\_mfa\_secret)
3. if not totp.verify(request.form.get("mfa\_code")):

return apology("Invalid MFA Code", 403)

**Security Misconfiguration Fixes**

1. **Secure HTTP Headers:** Use Flask-Talisman to enforce security headers.
2. **HTTPS Enforcement:** Redirect HTTP requests to HTTPS.
3. **Debug Mode Disabled:** Ensure debug=True is not used in production.
4. **Keep Dependencies Updated:** Use pip list --outdated and update packages regularly.

**Conclusion and Recommendations**

**Immediate Fixes Required:**

* Implement CSRF tokens in forms.
* Enforce HTTPS for form submissions.
* Add Clickjacking protection headers.
* Set Content Security Policy (CSP) headers.

**Best Practices to Follow:**

* Maintain secure coding practices.
* Regularly scan and update dependencies.
* Implement Multi-Factor Authentication (MFA).
* Follow OWASP Top 10 security guidelines.

By implementing these recommendations, the security of the website will be significantly improved against common web vulnerabilities.

# Security risk assessment report

|  |
| --- |
| **Part 1: Select up to three hardening tools and methods to implement** |
| 1. Employees Share Passwords 2. Admin Password for Database is Set to Default 3. Firewalls Do Not Filter Traffic 4. Multifactor Authentication (MFA) is Not Used |
|

|  |
| --- |
| **Part 2: Explain your recommendations** |
| **1. Employees Share Passwords**   * **Risk:** Increases the likelihood of unauthorized access. * **Recommended Fix:** Implement NIST password policies to ensure:   + Unique, strong passwords per user.   + Password expiration and change policies.   + Employee education on password security.   **2. Admin Password for Database is Set to Default**   * **Risk:** Default passwords make the database highly vulnerable. * **Recommended Fix:**   + Change the default password immediately.   + Use a strong, randomly generated password.   + Restrict database access to authorized personnel.   **3. Firewalls Do Not Filter Traffic**   * **Risk:** Allows unrestricted inbound and outbound network traffic. * **Recommended Fix:** Implement an **Intrusion Prevention System (IPS)**:   + Configure firewalls with strict access control rules.   + Monitor traffic for anomalies and suspicious behavior.   + Block unauthorized IPs and malicious activities.   **4. Multifactor Authentication (MFA) is Not Used**   * **Risk:** Increases vulnerability to brute force and credential stuffing attacks. * **Recommended Fix:** Implement MFA using the following approach:   + Install necessary packages:   pip install pyotp qrcode pillow   * + Generate an MFA secret per user:   secret = pyotp.random\_base32()  db.execute("UPDATE users SET mfa\_secret = ? WHERE id = ?", secret, user\_id)   * + Validate MFA during login:   totp = pyotp.TOTP(user\_mfa\_secret)  if not totp.verify(request.form.get("mfa\_code")):  return apology("Invalid MFA Code", 403) |

**Examine alerts, logs, and rules with Suricata**

**Task 1. Examine a custom rule in Suricata**

The /home/analyst directory contains a custom.rules file that defines the network traffic rules, which Suricata captures.

In this task, I’ll explore the composition of the Suricata rule defined in the custom.rules file.

* Use the cat command to display the rule in the custom.rules file:

cat custom.rules

The command returns the rule as the output in the shell:

alert http $HOME\_NET any -> $EXTERNAL\_NET any (msg:"GET on wire"; flow:established,to\_server; content:"GET"; http\_method; sid:12345; rev:3;)

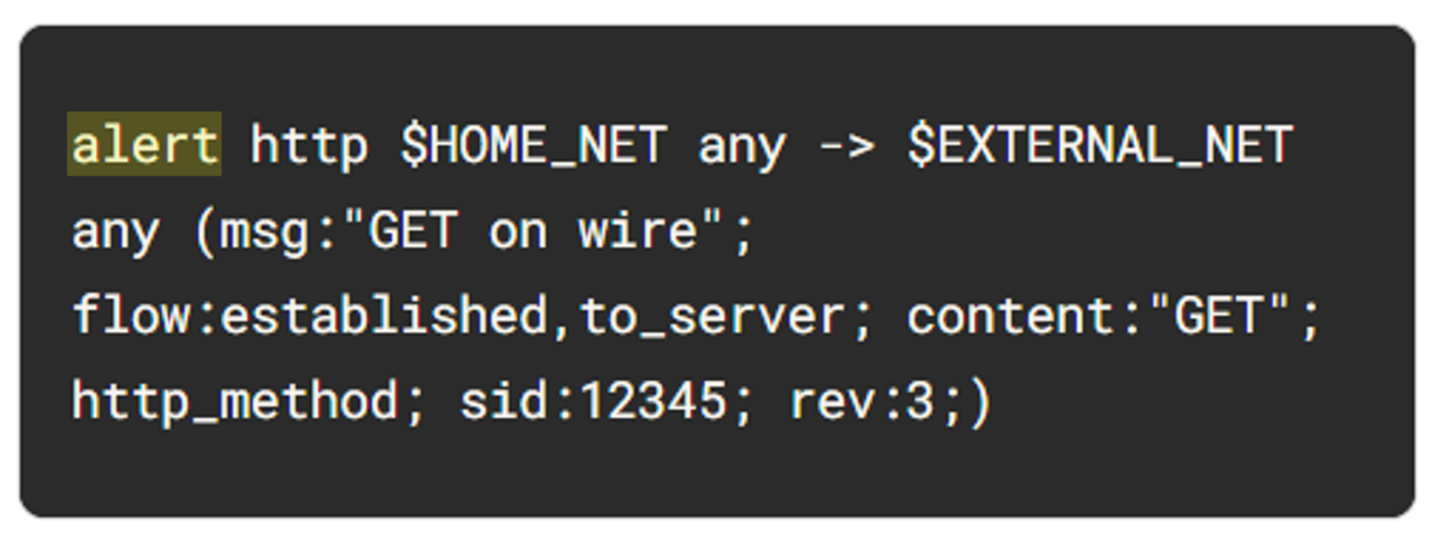
**Terminal Code:**

analyst@c6cfc5a176f1:~$ cat custom.rules

alert http $HOME\_NET any -> $EXTERNAL\_NET any (msg:"GET on wire"; flow:established,to\_server; content:"GET"; http\_method; sid:12345; rev:3;)

This rule consists of three components: an **action**, a **header**, and **rule options**.

Let's examine each component in more detail.

**Action**

The **action** is the first part of the signature. It determines the action to take if all conditions are met.

Actions differ across network intrusion detection system (NIDS) rule languages, but some common actions are alert, drop, pass, and reject.

Using our example, the file contains a single alert as the action. The alert keyword instructs to alert on selected network traffic. The IDS will inspect the traffic packets and send out an alert in case it matches.

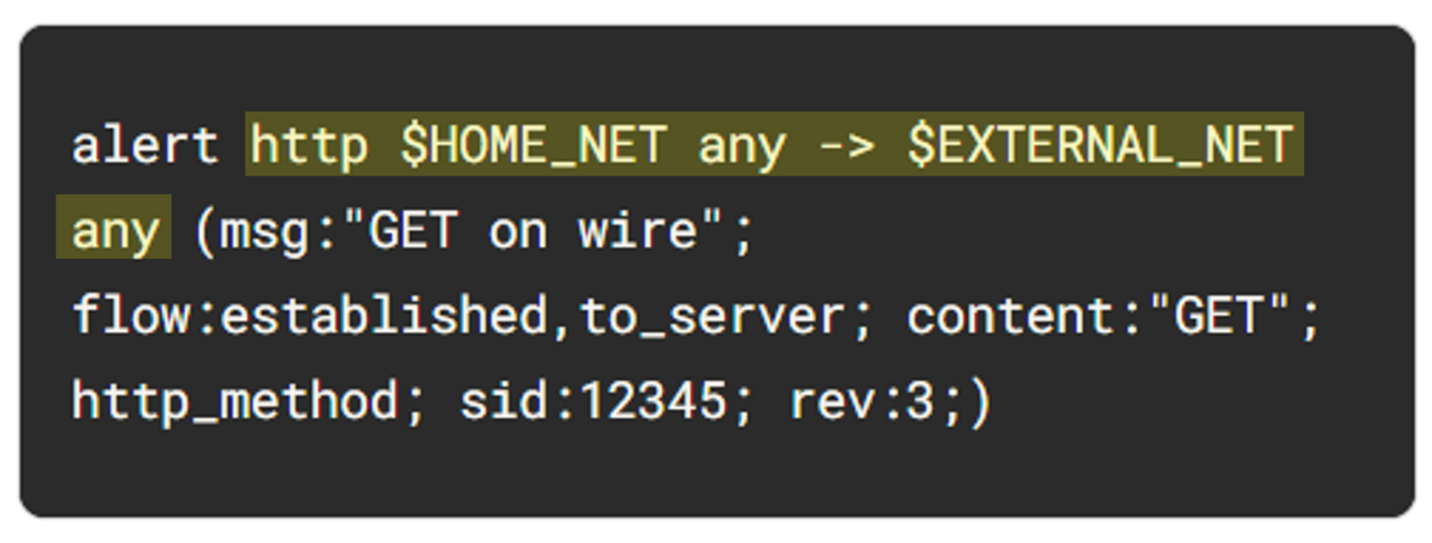
Note that the drop action also generates an alert, but it drops the traffic. A drop action only occurs when Suricata runs in IPS mode.

The pass action allows the traffic to pass through the network interface. The pass rule can be used to override other rules. An exception to a drop rule can be made with a pass rule. For example, the following rule has an identical signature to the previous example, except that it singles out a specific IP address to allow only traffic from that address to pass:

pass http 172.17.0.77 any -> $EXTERNAL\_NET any (msg:"BAD USER-AGENT";flow:established,to\_server;content:!”Mozilla/5.0”; http\_user\_agent; sid: 12365; rev:1;)

The reject action does not allow the traffic to pass. Instead, a TCP reset packet will be sent, and Suricata will drop the matching packet. A TCP reset packet tells computers to stop sending messages to each other.

**Header**



The next part of the signature is the **header**. The header defines the signature’s network traffic, which includes attributes such as protocols, source and destination IP addresses, source and destination ports, and traffic direction.

The next field after the action keyword is the protocol field. In our example, the protocol is http, which determines that the rule applies only to HTTP traffic.

The parameters to the protocol http field are $HOME\_NET any -> $EXTERNAL\_NET any. The arrow indicates the direction of the traffic coming from the $HOME\_NET and going to the destination IP address $EXTERNAL\_NET.

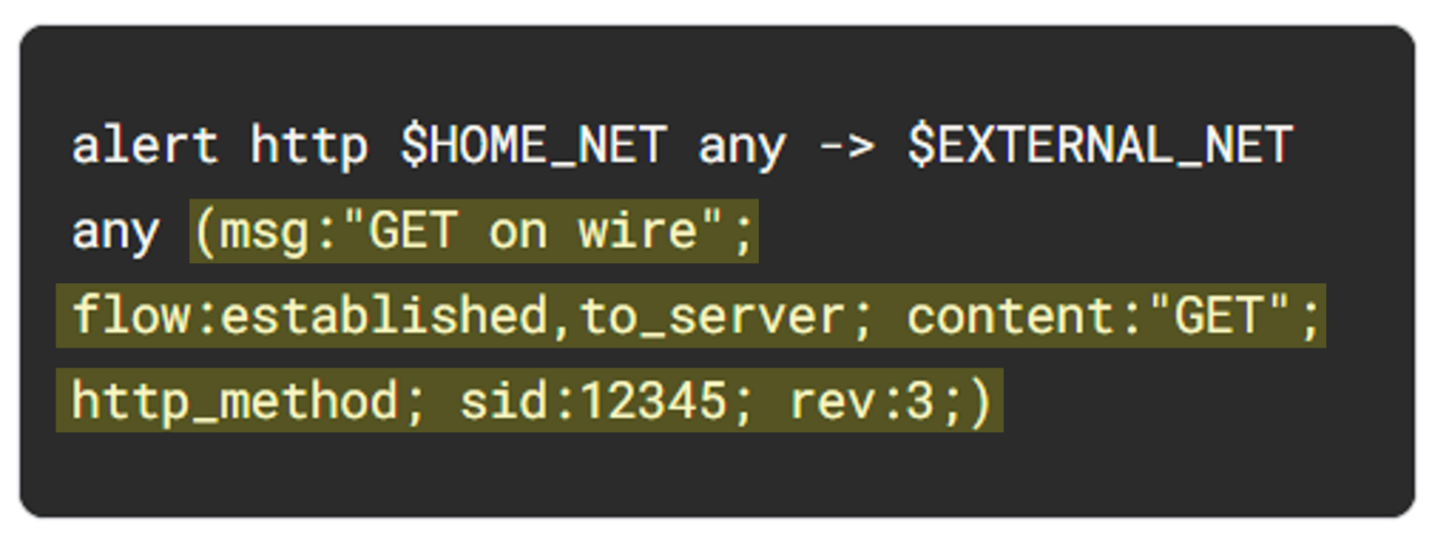
$HOME\_NET is a Suricata variable defined in /etc/suricata/suricata.yaml that you can use in your rule definitions as a placeholder for your local or home network to identify traffic that connects to or from systems within your organization.

In this lab $HOME\_NET is defined as the 172.21.224.0/20 subnet.

The word any means that Suricata catches traffic from any port defined in the $HOME\_NET network.

So far, we know that this signature triggers an alert when it detects any http traffic leaving the home network and going to the external network.

**Rule options**



The many available **rule options** allow you to customize signatures with additional parameters. Configuring rule options helps narrow down network traffic so you can find exactly what you’re looking for. As in our example, rule options are typically enclosed in a pair of parentheses and separated by semicolons.

Let's further examine the rule options in our example:

* The msg: option provides the alert text. In this case, the alert will print out the text “GET on wire”, which specifies why the alert was triggered.
* The flow:established,to\_server option determines that packets from the client to the server should be matched. (In this instance, a server is defined as the device responding to the initial SYN packet with a SYN-ACK packet.)
* The content:"GET" option tells Suricata to look for the word GET in the content of the http.method portion of the packet.
* The sid:12345 (signature ID) option is a unique numerical value that identifies the rule.
* The rev:3 option indicates the signature's revision which is used to identify the signature's version. Here, the revision version is 3.

To summarize, this signature triggers an alert whenever Suricata observes the text GET as the HTTP method in an HTTP packet from the home network going to the external network.

**Task 2. Trigger a custom rule in Suricata**

Now I am familiar with the composition of the custom Suricata rule, I must trigger this rule and examine the alert logs that Suricata generates.

1. List the files in the /var/log/suricata folder:

ls -l /var/log/suricata

**Terminal Code:**

analyst@c6cfc5a176f1:~$ ls -l /var/log/suricata

total 0

Note that before running Suricata, there are no files in the /var/log/suricata directory.

1. Run suricata using the custom.rules and sample.pcap files:

sudo suricata -r sample.pcap -S custom.rules -k none

This command starts the Suricata application and processes the sample.pcap file using the rules in the custom.rules file. It returns an output stating how many packets were processed by Suricata.

**Terminal Code:**

analyst@c6cfc5a176f1:~$ sudo suricata -r sample.pcap -S custom.rules -k none

18/2/2025 -- 06:32:14 - <Notice> - This is Suricata version 4.1.2 RELEASE

18/2/2025 -- 06:32:15 - <Notice> - all 2 packet processing threads, 4 management threads initialized, engine started.

18/2/2025 -- 06:32:15 - <Notice> - Signal Received. Stopping engine.

18/2/2025 -- 06:32:17 - <Notice> - Pcap-file module read 1 files, 200 packets, 54238 bytes

Now I’ll further examine the options in the command:

* The -r sample.pcap option specifies an input file to mimic network traffic. In this case, the sample.pcap file.
* The -S custom.rules option instructs Suricata to use the rules defined in the custom.rules file.
* The -k none option instructs Suricata to disable all checksum checks.

Suricata adds a new alert line to the /var/log/suricata/fast.log file when all the conditions in any of the rules are met.

1. List the files in the /var/log/suricata folder again:

ls -l /var/log/suricata

**Terminal Code:**

analyst@c6cfc5a176f1:~$ ls -l /var/log/suricata

total 16

-rw-r--r-- 1 root root 1430 Feb 18 06:32 eve.json

-rw-r--r-- 1 root root 292 Feb 18 06:32 fast.log

-rw-r--r-- 1 root root 2911 Feb 18 06:32 stats.log

-rw-r--r-- 1 root root 353 Feb 18 06:32 suricata.log

Note that after running Suricata, there are now four files in the /var/log/suricata directory, including the fast.log and eve.json files. You'll examine these files in more detail.

1. Use the cat command to display the fast.log file generated by Suricata:

cat /var/log/suricata/fast.log

The output returns alert entries in the log:

11/23/2022-12:38:34.624866 [\*\*] [1:12345:3] GET on wire [\*\*] [Classification: (null)] [Priority: 3] {TCP} 172.21.224.2:49652 -> 142.250.1.139:80

11/23/2022-12:38:58.958203 [\*\*] [1:12345:3] GET on wire [\*\*] [Classification: (null)] [Priority: 3] {TCP} 172.21.224.2:58494 -> 142.250.1.139:80

Each line or entry in the fast.log file corresponds to an alert generated by Suricata when it processes a packet that meets the conditions of an alert generating rule. Each alert line includes the message that identifies the rule that triggered the alert, as well as the source, destination, and direction of the traffic.

**Terminal Code:**

analyst@c6cfc5a176f1:~$ cat /var/log/suricata/fast.log

11/23/2022-12:38:34.624866 [\*\*] [1:12345:3] GET on wire [\*\*] [Classification: (null)] [Priority: 3] {TCP} 172.21.224.2:49652 -> 142.250.1.139:80

11/23/2022-12:38:58.958203 [\*\*] [1:12345:3] GET on wire [\*\*] [Classification: (null)] [Priority: 3] {TCP} 172.21.224.2:58494 -> 142.250.1.102:80

**Task 3. Examine eve.json output**

In this task, I must examine the additional output that Suricata generates in the eve.json file.

As previously mentioned, this file is located in the /var/log/suricata/ directory.

The eve.json file is the standard and main Suricata log file and contains a lot more data than the fast.log file. This data is stored in a JSON format, which makes it much more useful for analysis and processing by other applications.

1. Use the cat command to display the entries in the eve.json file:

cat /var/log/suricata/eve.json

The output returns the raw content of the file. I have noticed that there is a lot of data returned that is not easy to understand in this format.

**Terminal Code:**

analyst@c6cfc5a176f1:~$ cat /var/log/suricata/eve.json

{"timestamp":"2022-11-23T12:38:34.624866+0000","flow\_id":7862278060181,"pcap\_cnt":70,"event\_type":"alert","src\_ip":"172.21.224.2","src\_port":49652,"dest\_ip":"142.250.1.139","dest\_port":80,"proto":"TCP","tx\_id":0,"alert":{"action":"allowed","gid":1,"signature\_id":12345,"rev":3,"signature":"GET on wire","category":"","severity":3},"http":{"hostname":"opensource.google.com","url":"\/","http\_user\_agent":"curl\/7.74.0","http\_content\_type":"text\/html","http\_method":"GET","protocol":"HTTP\/1.1","status":301,"redirect":"https:\/\/opensource.google\/","length":223},"app\_proto":"http","flow":{"pkts\_toserver":4,"pkts\_toclient":3,"bytes\_toserver":357,"bytes\_toclient":788,"start":"2022-11-23T12:38:34.620693+0000"}}

{"timestamp":"2022-11-23T12:38:58.958203+0000","flow\_id":1529124663497972,"pcap\_cnt":151,"event\_type":"alert","src\_ip":"172.21.224.2","src\_port":58494,"dest\_ip":"142.250.1.102","dest\_port":80,"proto":"TCP","tx\_id":0,"alert":{"action":"allowed","gid":1,"signature\_id":12345,"rev":3,"signature":"GET on wire","category":"","severity":3},"http":{"hostname":"opensource.google.com","url":"\/","http\_user\_agent":"curl\/7.74.0","http\_content\_type":"text\/html","http\_method":"GET","protocol":"HTTP\/1.1","status":301,"redirect":"https:\/\/opensource.google\/","length":223},"app\_proto":"http","flow":{"pkts\_toserver":4,"pkts\_toclient":3,"bytes\_toserver":357,"bytes\_toclient":797,"start":"2022-11-23T12:38:58.955636+0000"}}

1. Use the jq command to display the entries in an improved format:

jq . /var/log/suricata/eve.json | less

1. Press **Q** to exit the less command and to return to the command-line prompt.

Note how much easier it is to read the output now as opposed to the cat command output.

**Terminal Code:**

analyst@c6cfc5a176f1:~$ jq . /var/log/suricata/eve.json | less

{

"timestamp": "2022-11-23T12:38:34.624866+0000",

"flow\_id": 7862278060181,

"pcap\_cnt": 70,

"event\_type": "alert",

"src\_ip": "172.21.224.2",

"src\_port": 49652,

"dest\_ip": "142.250.1.139",

"dest\_port": 80,

"proto": "TCP",

"tx\_id": 0,

"alert": {

"action": "allowed",

"gid": 1,

"signature\_id": 12345,

"rev": 3,

"signature": "GET on wire",

"category": "",

"severity": 3

},

"http": {

"hostname": "opensource.google.com",

"url": "/",

"http\_user\_agent": "curl/7.74.0",

"http\_content\_type": "text/html",

"http\_method": "GET",

**Conclusion**

I’ve completed this activity to use Suricata to trigger alerts on network traffic.

Which is based on some practical experience in running Suricata to

* create custom rules and run them in Suricata,
* monitor traffic captured in a packet capture file, and
* examine the fast.log and eve.json output.

## PASTA worksheet

|  |  |
| --- | --- |
| **Stages** | **Sneaker company** |
| **I. Define business and security objectives** | Make **2-3 notes** of specific business requirements that will be analyzed.   * *Will the app process transactions?* * *Does it do a lot of back-end processing?* * *Are there industry regulations that need to be considered?*   *This Finance app is developed for simple and seamlessly connects sellers and buyers (Traders). Where the sellers can buy cryptocurrencies by their accounts by using login credentials and others.*  *The process of the transaction seamlessly throws a big security concern.*  *Yes, this app is doing a lot of back-end processes. In this app, users just interact with the application interface and choose the sneakers they want to buy further listing the available items and processing transactions and many other functions are performed by the back-end.*  *Industries need to focus on securing the PII of users because they use their accounts for purchasing the company products so, the company needs to focus on security threats and conduct threat modeling techniques to find out proactive vulnerabilities which cause risk to user personal data.* |
| **II. Define the technical scope** | List oftechnologies used by the application:   * *Application programming interface (API)* * *Public key infrastructure (PKI)* * *SHA-256* * *SQL*   Write **2-3 sentences** (40-60 words) that describe why you choose to prioritize that technology over the others.  The technologies that are mentioned above are the best suited for this application.  An API is a set of rules that define how software components interact with each other. By using API we don’t need to perform functionality of n application from scratch.  PKI is an encryption framework that secures the exchange of online information. We can secure user login credentials more efficiently by using symmetric and asymmetric techniques.  SHA-256 is a commonly used hash function that takes an input of any length and produces a digest of 256 bits. By using the hashing function and salting technique we can secure users passwords and prevent threat attacks from users accounts.  SQL is a programming language used to create, interact with, and request information from a database. For example, the mobile app uses SQL to store information about the sneakers that are for sale, as well as the sellers who are selling them. It also uses SQL to access that data during a purchase. |
| **III. Decompose application** | [Sample data flow diagram](https://docs.google.com/presentation/d/1ol7y79popTFfNHM-90ES-H-i1Lpd0YNvPShxBlXozjg/template/preview?resourcekey=0-DZAkf7Vzh2PXsP-j3oXV-g) |
| **IV. Threat analysis** | List **2 types of threats** in the PASTA worksheet that are risks to the information being handled by the application.   * *What are the internal threats?* * *What are the external threats?*   *The internal threats could be application users who use weak passwords for accessing their accounts making is easy for attackers to access their accounts.*  *The external threat could be SQL injection which is exploited by threat actors in the database. This database is performing flawful operations.* |
| **V. Vulnerability analysis** | List **2 vulnerabilities** in the PASTA worksheet that could be exploited.   * *Could there be things wrong with the codebase?* * *Could there be weaknesses in the database?* * *Could there be flaws in the network?*   *There could be a wrong use of PKI in applications which will be vulnerable in the future for organizations.*  *Yes, there could be a weakness in the database. The malicious actor uses SQL injection in the database which will be a risk for organizations.*  *If there any vulnerability in API and organizations aren’t focused on hashing techniques, It will be a cause for flaws in the network.* |
| **VI. Attack modeling** | [Sample attack tree diagram](https://docs.google.com/presentation/d/1FmWLyHgmq9XQoVuMxOym2PHO8IuedCkan4moYnI-EJ0/template/preview?usp=sharing&resourcekey=0-zYPY7AhPJdcClXamlAfOag) |
| **VII. Risk analysis and impact** | List **4 security controls** that you’ve learned about that can reduce risk.  When performing threat modeling, multiple methods can be used, such as  STRIDE  PASTA  Trike  VAST |

**Write-up for Solved picoCTF Challenges**

**Challenge 1: interencdec.py**

**Description:**

This challenge involved decoding a base64-encoded string and then decrypting the result using a Caesar cipher.

**Steps to Solve:**

1. **Base64 Decoding:** The encoded flag was first base64-decoded twice.
2. **Caesar Cipher Decryption:** The output was an encrypted string that had been shifted using a Caesar cipher. A brute-force approach was applied by shifting the text from 0 to 25 positions.
3. **Correct Flag Identification:** By iterating through all possible shifts, the correct plaintext was found at shift 7.

**Code:**  
import base64

# Step 1: Base64 decode

encoded\_flag = "YidkM0JxZGtwQlRYdHFhR3g2YUhsZmF6TnFlVGwzWVROclh6ZzJhMnd6TW1zeWZRPT0nCg=="

# Decode the base64 twice

decoded\_once = base64.b64decode(encoded\_flag).decode('utf-8')

# Split by single quotes and extract the second part

extracted\_part = decoded\_once.split("'")[1]

# Decode the extracted part again with base64

decoded\_twice = base64.b64decode(extracted\_part).decode('utf-8')

# Step 2: Caesar cipher decryption function

def caesar\_decrypt(ciphertext, shift):

    decrypted = []

    for char in ciphertext:

        if char.isalpha():

            shift\_val = ord('a') if char.islower() else ord('A')

            decrypted.append(chr((ord(char) - shift\_val - shift) % 26 + shift\_val))

        else:

            decrypted.append(char)

    return ''.join(decrypted)

# Try all possible Caesar cipher shifts (0-25)

for shift in range(26):

    print(f"Shift {shift}: {caesar\_decrypt(decoded\_twice, shift)}")

**Flag:**

picoCTF{caesar\_d3cr9pt3d\_86de32d2}

**Challenge 2: MOD26.py**

**Description:**

This challenge involved decrypting a message encoded using ROT13.

**Steps to Solve:**

1. **ROT13 Decryption:** The message was encrypted using ROT13, a simple letter substitution cipher that shifts characters by 13 places.
2. **Decoding with Python:** The codecs.decode function was used to decrypt the string.

**Code:**

import codecs

# Encrypted string

encrypted\_string = "cvpbPGS{arkg\_gvzr\_V'yy\_gel\_2\_ebhaqf\_bs\_ebg13\_GYpXOHqX}"

# ROT13 decryption using codecs

decrypted\_string = codecs.decode(encrypted\_string, 'rot\_13')

# Print the result

print(f"Decrypted string: {decrypted\_string}")

**Flag:**

picoCTF{next\_time\_I'll\_try\_2\_rounds\_of\_rot13\_TLcKBUdK}

**Challenge 3: TheNumbers.py**

**Description:**

This challenge involved decrypting a sequence of numbers mapped to letters.

**Steps to Solve:**

1. **Mapping Numbers to Letters:** The numbers were mapped to their corresponding letters in the alphabet (A=1, B=2, ..., Z=26).
2. **Reconstructing the Flag:** The outside and inside parts of the flag were reconstructed separately and then combined.

**Code:**

# Function to map numbers to letters

def number\_to\_letter(number\_sequence):

    return ''.join([chr(num + 64) for num in number\_sequence])

# Encrypted sequence (outside curly braces)

outside\_braces = [16, 9, 3, 15, 3, 20, 6]

# Encrypted sequence (inside curly braces)

inside\_braces = [20, 8, 5, 14, 21, 13, 2, 5, 18, 19, 13, 1, 19, 15, 14]

# Convert numbers to letters

outside\_decrypted = number\_to\_letter(outside\_braces)

inside\_decrypted = number\_to\_letter(inside\_braces)

# Construct the flag

flag = f"{outside\_decrypted}{{{inside\_decrypted}}}"

# Output the decrypted flag

print(f"Decrypted flag: {flag}")

**Flag:**

picoCTF{THENUMBERSMASON}

**Challenge 4: 13.py**

**Description:**

This challenge was another ROT13 decryption task.

**Steps to Solve:**

1. **ROT13 Decryption:** The encrypted message was decoded using the codecs.decode function.
2. **Retrieving the Flag:** The output was the correctly decrypted flag.

**Code:**

import codecs

# Encrypted string

encrypted\_string = "cvpbPGS{abg\_gbb\_onq\_bs\_n\_ceboyrz}"

# ROT13 decryption using codecs

decrypted\_string = codecs.decode(encrypted\_string, 'rot\_13')

# Print the result

print(f"Decrypted string: {decrypted\_string}")

**Flag:**

picoCTF{not\_too\_bad\_of\_a\_problem}

**Challenge 5: Basic-mod1.py**

**Description:**

This challenge involved decrypting a modular arithmetic-based encoding scheme.

**Steps to Solve:**

1. **Modulo Operation:** Each number in the encrypted list was reduced modulo 37.
2. **Character Mapping:** The resulting values were mapped to a custom alphabet consisting of lowercase letters, digits, and an underscore.
3. **Flag Construction:** The final string was built by iterating through the list and applying the decoding logic.

**Code:**

import string

alphabet = string.ascii\_lowercase

alphabet += "0123456789\_"

flag\_enc = [350, 63, 353, 198, 114, 369, 346, 184, 202, 322, 94, 235, 114, 110, 185, 188, 225, 212, 366, 374, 261, 213]

flag = ""

for c in flag\_enc:

    pos = c % 37

    flag += alphabet[pos]

print(flag)

**Flag:**

r0und\_n\_r0und\_add17ec2

**Conclusion:**

These challenges covered a variety of cryptographic techniques, including Base64 encoding/decoding, Caesar cipher decryption, ROT13 encryption, number-to-letter mapping, and modular arithmetic decryption. Each challenge required logical thinking and a fundamental understanding of encryption techniques to retrieve the correct flag.  
  
