Name: César Rafael Sorgato Santos

Email: rafaelsorgato@hotmail.com

First, I opened the file to manually analyze the available data. After spending a few minutes familiarizing myself with the file, I noticed some unusual entries in the ClientRequestPath field. So, I decided to use regex and Python to extract all values with unusual entries:

import pandas as pd

import re

df = pd.read\_excel(r"C:\Users\rafae\Downloads\file.xlsx")

data = []

for index,column in df.iterrows():

    data.append(column["ClientRequestPath"])

itens = []

for item in data:

    if not re.match(r'^/[a-z0-9/]+$', item, re.IGNORECASE):

            if item not in itens:  
 itens.append(item)

df\_itens = pd.DataFrame(itens, columns=['ClientRequestPath'])

with pd.ExcelWriter(r"C:\Users\rafae\Downloads\file.xlsx", mode='a', if\_sheet\_exists='replace') as writer:

    df\_itens.to\_excel(writer, sheet\_name='clientpath', index=False)

29 lines were returned, representing important results that provide various relevant information:

/<img src='x' onerror='alert(1)'>

/admin.php?user=admin&password=admin

/reset-password?email=%27%20OR%201=1--

/cgi-bin/test-cgi

/../../../../windows/system32/cmd.exe

/favicon.ico

/../../../windows/win.ini

/shell.php?cmd=cat%20/etc/passwd

/<script>alert('XSS')</script>

/vulnerable.php?search=<svg onload=alert(1)>

/index.php?id=' OR '1'='1

/../../../../../../../../../../etc/shadow

/script.php?=<script>window.location='http://evil.com'</script>

/<iframe src='javascript:alert(1)'></iframe>

/"!--"<XSS>=&{()}

/.git/config

/config.php.bak

/login.jsp?user=admin'--

/<marquee><img src=1 onerror=alert(1)></marquee>

/wp-admin/setup-config.php

/robots.txt

/../../boot.ini

/%00%01%02%03%04%05%06%07

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

/eval?param=<%=Runtime.getRuntime().exec("calc.exe")%>

/<meta http-equiv='refresh' content='0url=javascript:alert(1)'>

/search?q=<script>alert(0)</script>

/phpinfo.php

/api/v1/users?search= DROP TABLE users--

## Risk 1 - Injection

Some examples found:

- /script>alert('XSS')</script> (Cross-site scripting)

- /marquee><img src=1 onerror=alert(1)></marquee> (Cross-site scripting)

- /api/v1/users?search= DROP TABLE users-- (SQL injection)

**Risk**: Malicious users can execute malicious scripts/queries and compromise security through unauthorized access, data loss, and the integrity and availability of the data.  
  
**Policy -**

**Objective**: Block and filter potentially malicious entries requested from the web.

Scope: All systems that provide web applications and APIs.

Responsibilities:

**Management**: Allocate resources and ensure compliance.

Development: Implement necessary resources.

Information Security: Respond to incidents, monitor and test filtering and blocking systems.

Users: Report any suspected vulnerabilities or malicious activity.

**Controls**:

Implementation of filtering systems.

Code reviews.

Routine testing of protection effectiveness.

Configuration adjustments.

**Implementation** -

Filtering System: There are numerous ways to block injection (WAF, HTML tags, code, etc.). I thought about training a model with an XSS [dataset](https://raw.githubusercontent.com/fmereani/Cross-Site-Scripting-XSS/refs/heads/master/XSSDataSets/Payloads.csv) to detect malicious inputs (keep in mind that it was only trained for XSS; other types of inputs can also be trained, but due to time constraints, I only focused on XSS).

import pandas as pd  
from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

import chardet

with open(r'C:\Users\rafae\Downloads\tools\scripts/xss.csv', 'rb') as f:

    result = chardet.detect(f.read())

df = pd.read\_csv(r'C:\Users\rafae\Downloads\tools\scripts/xss.csv', encoding=result['encoding'])

X = df['Payloads']

y = df['Class']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

vectorizer = TfidfVectorizer(stop\_words='english', max\_features=5000)

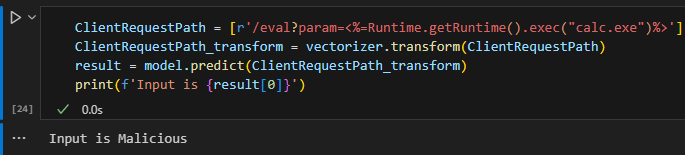
X\_train\_tfidf = vectorizer.fit\_transform(X\_train)

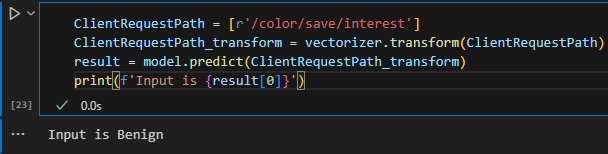
X\_test\_tfidf = vectorizer.transform(X\_test)

model = MultinomialNB()

model.fit(X\_train\_tfidf, y\_train)

Once trained, it would only be necessary to call the model, passing the ClientRequestPath as a parameter.





If there are very large amounts of requests, the code can be optimized for performance by adding a pre-validation step before prediction, such as validating for special characters like “\*!@$<>#%”. If none of these or other symbols are present, it is not necessary to perform the prediction.

**Routine Testing**: There are several solutions available for testing malicious inputs, one example is XSStrike, which automates XSS testing. In this case, we only need to download the software and apply the necessary tests. This will not be discussed in detail as it is not part of the challenge, but it is provided as an extra:



## Risk 2 - **Misconfiguration Risk**

Some examples found:

/api/v1/users (API versioning available for use, allow only the current version)

/wp-admin/setup-config.php (Exposure of File)

/../../../windows/win.ini (Directory Traversal Attack)

/config.php.bak (Exposure of File)

Risk: Malicious users can compromise security through unauthorized access, data loss, and the integrity and availability of the data.

**Policy** -

**Objective**: Review, maintain, and adjust web application/server configurations to ensure they are secure.

Scope: All systems that provide web applications and APIs.

**Responsibilities**:

**Management**: Allocate resources and ensure compliance.

Development: Implement necessary resources and configure systems.

Information Security: Respond to incidents, monitor, and test web system configurations.

Users: Report any suspected vulnerabilities or malicious activity.

**Controls**:

Implementation of filtering systems.

Code reviews.

Routine testing of protection effectiveness.

Configuration adjustments.

**Implementation -**

1 - Manually configure applications according to the needs and usage, for example: For the accessed path "/cgi-bin/test-cgi", it is necessary to know which server is hosting it (e.g., Apache) and understand if CGI is being used, in order to restrict access or disable it if not in use. The same applies to other cases, such as: /config.php.bak and /wp-admin/setup-config.php.

2 - Detect (Directory Traversal Attack)

The ideal solution for this step is to fix misconfigurations; however, if it is necessary to perform the detection through scripts, various [wordlists](https://github.com/Karanxa/Bug-Bounty-Wordlists/tree/main) can be used together to detect pre-mapped misconfiguration patterns (the example below will include both directory traversal attack examples and the misconfiguration cases from step 1):

In this example, two lists will be used ([config.txt](https://github.com/Karanxa/Bug-Bounty-Wordlists/blob/main/config.tx) and [php\_files\_with\_path.txt](https://github.com/Karanxa/Bug-Bounty-Wordlists/blob/main/php_files_with_path.txt)), but these lists will depend on the applications used in the environment.

The code validates the two types of requests that may arrive (encoded and non-encoded).

import urllib.parse

import pandas as pd

miss\_patterns = [line.strip() for line in open(r'C:\Users\rafae\Downloads\tools\scripts/leaky-misconfigs.txt')]

def Filter(ClientRequestPath):

    if any(character in ClientRequestPath for character in["%3B" , "%3C" , r"%3E" , "%0A" , "%0D" , "%20" , "%00"]):

        return "BAD REQUEST"

    else:

        ClientRequestPath = urllib.parse.unquote(ClientRequestPath)

        if("../") in ClientRequestPath:

            return "BAD REQUEST"

        elif any(pattern in ClientRequestPath for pattern in miss\_patterns):

            return "BAD REQUEST"

        else:

            return "PASS"

print(Filter('/../../../windows/win.ini'))

print(Filter(f'%2F..%2F..%2F..%2Fwindows%2Fwin.ini'))

print(Filter('/color/save/interest'))

print(Filter('/wp-admin/setup-config.php'))

print(Filter('/.git/config'))

**Resultados**:

BAD REQUEST

BAD REQUEST

PASS

BAD REQUEST

BAD REQUEST

## Risco 3 - Suspect user-agents

Some examples found:

Mozilla/5.0 (compatible MSIE 5.0 Windows 98 Win 9x 4.90 Trident/5.1)

Mozilla/5.0 (Android 1.0 Mobile rv:56.0) Gecko/56.0 Firefox/56.0

Mozilla/5.0 (iPod U CPU iPhone OS 3\_2 like Mac OS X eu-FR) AppleWebKit/532.20.4 (KHTML, like Gecko) Version/3.0.5 Mobile/8B116 Safari/6532.20.4

Risk: Malicious users are altering user-agent data, which may be used for automation tasks such as brute-force attacks.

**Policy** -

**Objective**: Block suspicious user-agents.

Scope: All systems providing web applications and APIs.

**Responsibilities**:

Management: Allocate resources and ensure compliance.

Development: Implement the necessary resources.

Information Security: Respond to incidents, monitor, and test blocking systems.

Users: Report any suspected vulnerabilities or malicious activity.

**Controls**:

Implementation of blocking systems.

Routine testing of protection effectiveness.

**Implementation** -

Blocking System

Regarding user-agent blocking, it is a complicated task because it may affect regular users. Therefore, it will depend on the business rules. The script below focuses only on extremely suspicious user-agents, as they contain very old versions that are no longer in common circulation in the market and are incompatible with modern web pages. (Almost all user-agents in the document appear to have been generated manually, but I will ignore this and proceed assuming that most of them are reliable). In the code below, I considered outdated versions of certain operating systems to classify them as suspicious, with searches being done using regex.

import re

def Filter(ClientRequestUserAgent):

    if re.match(r'.+(Android [1-3]\.|Windows (9(8|5)|NT)|iPad OS [1-5]\_|iPod)', ClientRequestUserAgent):

        return "SUSPICIOUS"

    else:

        return "Not suspicious"

print(Filter("Mozilla/5.0 (iPad CPU iPad OS 1\_1\_5 like Mac OS X) AppleWebKit/536.1 (KHTML, like Gecko) CriOS/60.0.896.0 Mobile/31N563 Safari/536.1"))

print(Filter("Mozilla/5.0 (Android 12.1 Mobile rv:38.0) Gecko/38.0 Firefox/38.0"))

print(Filter("Mozilla/5.0 (Android 1.1 Mobile rv:33.0) Gecko/33.0 Firefox/33.0"))

print(Filter("Opera/9.99.(Windows NT 5.0 or-IN) Presto/2.9.172 Version/10.00"))

RETORNOS:  
SUSPICIOUS

Not suspicious

SUSPICIOUS

SUSPICIOUS

## Risk 4 - Lack of SSL/TLS

Some examples found:

thomas.biz - http

pena.com - http

sosa.com - http

valdez-clarke.com - http

marshall-odonnell.net - http

Risk: Malicious users can intercept unencrypted data traveling across the network.

**Policy** -

**Objective**: Implement SSL/TLS.

Scope: All systems providing web applications and APIs.

**Responsibilities**:

Management: Allocate resources and ensure compliance.

Development: Implement the necessary resources.

Information Security: Respond to incidents and monitor certificates.

Users: Report any expired certificates.

**Controls**:

Implement SSL/TLS.

Monitoring.

Blocking system.

**Implementation** -

Blocking System

Regarding blocking HTTP requests, it will depend on the business rules. Some companies opt not to use encryption due to cost reduction or because encryption is unnecessary based on the nature of the page.

The code below specifies which HTTP methods are not allowed in "http" requests.

BlockedMethods = ["GET","POST"]

def Filter(ClientRequestScheme, Method):

    if 'http' == ClientRequestScheme:

        if Method in BlockedMethods:

            return "HTTP BLOCKED"

return "PASS"

print(Filter("http","GET"))

print(Filter("http","POST"))

print(Filter("http","PATCH"))

print(Filter("https","POST"))

RETORNO:  
HTTP BLOCKED

HTTP BLOCKED

PASS

PASS

Other policies could be implemented, such as blocking ClientDeviceType = Desktop when the user-agent points to a mobile device; blocking access with an IP from country X while the ClientCountry points to country Y (e.g., 113.50.114.50 = CN | ClientCountry = IN). The same applies to country validation between IP/ClientCountry with ASN (Autonomous System Number), since they are fixed regions. However, I've run out of time, so I will submit with the data I managed to gather.