

**import http.server**

import socketserver  
import requests  
import re  
import logging  
from io import BytesIO

FORWARD\_URL = '<http://localhost:7070>' # Go frontend  
LISTEN\_PORT = 7070

**Configure logging**

logging.basicConfig(filename='waf\_smuggling.log', [level=logging.INFO](http://level=logging.INFO), format='%(asctime)s %(message)s')

**Regex patterns for obfuscated headers**

TE\_OBFUSCATED = re.compile(r'transfer[\s\_-]*encoding\s*:', re.IGNORECASE)  
CL\_OBFUSCATED = re.compile(r'content[\s\_-]*length\s*:', re.IGNORECASE)

def detect\_smuggling(headers, body):  
"""  
Returns (is\_smuggling\_detected, reason)  
"""  
header\_names = [k.lower() for k in headers.keys()]  
te\_headers = [k for k in headers.keys() if k.lower() == 'transfer-encoding']  
cl\_headers = [k for k in headers.keys() if k.lower() == 'content-length']

# 1. CL.TE or TE.CL: Both headers present  
if te\_headers and cl\_headers:  
 te\_value = headers[te\_headers[0]].lower()  
 cl\_value = headers[cl\_headers[0]]  
 if 'chunked' in te\_value:  
 # Check for CL.TE or TE.CL  
 return True, f'Both Content-Length ({cl\_value}) and Transfer-Encoding ({te\_value}) headers present.'  
  
# 2. Multiple TE or CL headers (TE.TE, CL.CL)  
if len(te\_headers) > 1 or len(cl\_headers) > 1:  
 return True, 'Multiple Transfer-Encoding or Content-Length headers detected.'  
  
# 3. Obfuscated TE/CL headers (e.g., Transfer\_Encoding, Transfer-Encoding with whitespace)  
for raw\_header in headers:  
 if TE\_OBFUSCATED.match(raw\_header):  
 return True, f'Obfuscated Transfer-Encoding header: {raw\_header}'  
 if CL\_OBFUSCATED.match(raw\_header):  
 return True, f'Obfuscated Content-Length header: {raw\_header}'  
  
# 4. Invalid/malformed TE or CL values  
if te\_headers:  
 te\_value = headers[te\_headers[0]].strip().lower()  
 if te\_value not in ['chunked', 'identity', '']:  
 return True, f'Unusual Transfer-Encoding value: {te\_value}'  
 # Check for chunked body format if chunked  
 if te\_value == 'chunked' and not body.endswith(b'0\r\n\r\n'):  
 return True, 'Transfer-Encoding chunked without proper chunked body ending.'  
  
if cl\_headers:  
 try:  
 cl\_value = int(headers[cl\_headers[0]].strip())  
 if cl\_value != len(body):  
 return True, f'Content-Length mismatch: header={cl\_value}, actual={len(body)}'  
 except Exception:  
 return True, f'Invalid Content-Length value: {headers[cl\_headers[0]]}'  
  
# 5. Duplicate or conflicting headers (e.g., both Transfer-Encoding: chunked, identity)  
if te\_headers:  
 te\_value = headers[te\_headers[0]].lower()  
 if ',' in te\_value and 'chunked' in te\_value and 'identity' in te\_value:  
 return True, f'Conflicting Transfer-Encoding values: {te\_value}'  
  
# 6. Non-standard line endings or header delimiters (CRLF injection)  
# Not directly visible via headers dict, but can be checked in raw request parsing (advanced)  
  
return False, ''

class WAFRequestHandler(http.server.BaseHTTPRequestHandler):  
def do\_POST(self):  
self.handle\_request()  
def do\_GET(self):  
self.handle\_request()  
def do\_PUT(self):  
self.handle\_request()  
def do\_DELETE(self):  
self.handle\_request()  
def do\_OPTIONS(self):  
self.handle\_request()

def handle\_request(self):  
 # Read request headers and body  
 content\_length = int(self.headers.get('Content-Length', 0))  
 body = self.rfile.read(content\_length) if content\_length else b''  
  
 # Detect smuggling  
 detected, reason = detect\_smuggling(self.headers, body)  
 if detected:  
 log\_entry = f"[WAF] HTTP Request Smuggling detected: {reason} | Path: {self.path} | Headers: {dict(self.headers)}"  
 logging.warning(log\_entry)  
 self.send\_response(400)  
 self.end\_headers()  
 self.wfile.write(b"WAF: HTTP Request Smuggling detected.\n")  
 return  
  
 # Forward request to Go frontend  
 try:  
 # Prepare headers for forwarding  
 forward\_headers = {k: v for k, v in self.headers.items()}  
 resp = requests.request(  
 method=self.command,  
 url=FORWARD\_URL + self.path,  
 headers=forward\_headers,  
 data=body,  
 allow\_redirects=False,  
 timeout=10  
 )  
 self.send\_response(resp.status\_code)  
 for k, v in resp.headers.items():  
 self.send\_header(k, v)  
 self.end\_headers()  
 self.wfile.write(resp.content)  
 except Exception as e:  
 logging.error(f"[WAF] Error forwarding request: {e}")  
 self.send\_response(502)  
 self.end\_headers()  
 self.wfile.write(b"WAF: Error forwarding request.\n")

if **name** == '**main**':  
with socketserver.ThreadingTCPServer(('', LISTEN\_PORT), WAFRequestHandler) as httpd:  
print(f'[WAF] Listening on port {LISTEN\_PORT}...')  
httpd.serve\_forever()

explain this code line by line, all the built in functions and datatypes and everything explain line by line, give small exampleas for the necessary

This Python script implements a simple Web Application Firewall (WAF) proxy to detect and block HTTP Request Smuggling attacks. It listens for HTTP requests, analyzes them for suspicious patterns, logs incidents, and forwards safe requests to a backend server. Below is a detailed, line-by-line explanation of the code, including built-in functions, datatypes, and relevant examples.

**Imports**

import http.server  
import socketserver  
import requests  
import re  
import logging  
from io import BytesIO

* import http.server: Imports Python’s built-in HTTP server classes for handling HTTP requests.
* import socketserver: Imports classes for creating network servers.
* import requests: Imports the popular third-party library for making HTTP requests.
* import re: Imports the regular expression module for pattern matching.
* import logging: Imports the logging module for writing logs to files or the console.
* from io import BytesIO: Imports an in-memory bytes buffer, useful for manipulating binary data.

**Example:**

import re  
pattern = re.compile(r'\d+')  
print(pattern.match('123')) # Output: <re.Match object; span=(0, 3), match='123'>

**Configuration**

FORWARD\_URL = 'http://localhost:7070' # Go frontend  
LISTEN\_PORT = 7070

* FORWARD\_URL: The URL to which safe requests are forwarded (the backend server).
* LISTEN\_PORT: The port number on which this WAF proxy will listen for incoming HTTP requests.

**Logging Setup**

logging.basicConfig(filename='waf\_smuggling.log', level=logging.INFO, format='%(asctime)s %(message)s')

* Configures logging to write INFO-level and higher messages to waf\_smuggling.log.
* %(asctime)s %(message)s specifies the log message format (timestamp + message).

**Regular Expressions for Obfuscated Headers**

TE\_OBFUSCATED = re.compile(r'transfer[\s\_\-]\*encoding\s\*:', re.IGNORECASE)  
CL\_OBFUSCATED = re.compile(r'content[\s\_\-]\*length\s\*:', re.IGNORECASE)

* TE\_OBFUSCATED and CL\_OBFUSCATED are regex patterns to match obfuscated Transfer-Encoding and Content-Length headers (e.g., with spaces, underscores, or dashes).
* re.IGNORECASE makes the match case-insensitive.

**Example:**

TE\_OBFUSCATED.match('Transfer\_Encoding:') # Output: <re.Match object ...>

**Smuggling Detection Function**

def detect\_smuggling(headers, body):  
 """  
 Returns (is\_smuggling\_detected, reason)  
 """

* Defines a function to check for HTTP request smuggling indicators.
* Returns a tuple: (True/False, reason string).

**Header Extraction**

header\_names = [k.lower() for k in headers.keys()]  
te\_headers = [k for k in headers.keys() if k.lower() == 'transfer-encoding']  
cl\_headers = [k for k in headers.keys() if k.lower() == 'content-length']

* header\_names: Lowercase list of all header names.
* te\_headers, cl\_headers: Lists of header keys matching Transfer-Encoding and Content-Length, respectively.

**Example:**

headers = {'Transfer-Encoding': 'chunked', 'Content-Length': '10'}  
te\_headers = [k for k in headers if k.lower() == 'transfer-encoding'] # ['Transfer-Encoding']

**Smuggling Checks**

**1. Both CL and TE headers present**

if te\_headers and cl\_headers:  
 te\_value = headers[te\_headers[^1\_0]].lower()  
 cl\_value = headers[cl\_headers[^1\_0]]  
 if 'chunked' in te\_value:  
 return True, f'Both Content-Length ({cl\_value}) and Transfer-Encoding ({te\_value}) headers present.'

* If both headers exist and Transfer-Encoding contains "chunked", this is a classic smuggling vector.

**2. Multiple TE or CL headers**

if len(te\_headers) > 1 or len(cl\_headers) > 1:  
 return True, 'Multiple Transfer-Encoding or Content-Length headers detected.'

* Multiple headers with the same name can confuse servers.

**3. Obfuscated TE/CL headers**

for raw\_header in headers:  
 if TE\_OBFUSCATED.match(raw\_header):  
 return True, f'Obfuscated Transfer-Encoding header: {raw\_header}'  
 if CL\_OBFUSCATED.match(raw\_header):  
 return True, f'Obfuscated Content-Length header: {raw\_header}'

* Checks for headers with unusual formatting (e.g., Transfer\_Encoding:).

**4. Invalid/malformed TE or CL values**

if te\_headers:  
 te\_value = headers[te\_headers[^1\_0]].strip().lower()  
 if te\_value not in ['chunked', 'identity', '']:  
 return True, f'Unusual Transfer-Encoding value: {te\_value}'  
 if te\_value == 'chunked' and not body.endswith(b'0\r\n\r\n'):  
 return True, 'Transfer-Encoding chunked without proper chunked body ending.'  
  
if cl\_headers:  
 try:  
 cl\_value = int(headers[cl\_headers[^1\_0]].strip())  
 if cl\_value != len(body):  
 return True, f'Content-Length mismatch: header={cl\_value}, actual={len(body)}'  
 except Exception:  
 return True, f'Invalid Content-Length value: {headers[cl\_headers[^1\_0]]}'

* Ensures Transfer-Encoding is valid and, if chunked, that the body ends properly.
* Checks Content-Length is a valid integer and matches the actual body length.

**5. Conflicting TE values**

if te\_headers:  
 te\_value = headers[te\_headers[^1\_0]].lower()  
 if ',' in te\_value and 'chunked' in te\_value and 'identity' in te\_value:  
 return True, f'Conflicting Transfer-Encoding values: {te\_value}'

* Detects multiple, conflicting values in the Transfer-Encoding header.

**6. Non-standard line endings**

* Not directly checked here, but mentioned for completeness.

**Return if safe:**

return False, ''

* No smuggling detected.

**WAF HTTP Request Handler Class**

class WAFRequestHandler(http.server.BaseHTTPRequestHandler):

* Inherits from BaseHTTPRequestHandler to handle HTTP requests.

**HTTP Methods**

def do\_POST(self): self.handle\_request()  
def do\_GET(self): self.handle\_request()  
def do\_PUT(self): self.handle\_request()  
def do\_DELETE(self): self.handle\_request()  
def do\_OPTIONS(self): self.handle\_request()

* For each HTTP method, calls handle\_request().

**Main Handler**

def handle\_request(self):  
 # Read request headers and body  
 content\_length = int(self.headers.get('Content-Length', 0))  
 body = self.rfile.read(content\_length) if content\_length else b''

* Reads the request body based on the Content-Length header.
* self.headers is a dictionary-like object with HTTP headers.
* self.rfile.read(n) reads n bytes from the request stream.

**Example:**

body = self.rfile.read(10) # Reads 10 bytes from the request body

**Smuggling Detection and Logging**

detected, reason = detect\_smuggling(self.headers, body)  
if detected:  
 log\_entry = f"[WAF] HTTP Request Smuggling detected: {reason} | Path: {self.path} | Headers: {dict(self.headers)}"  
 logging.warning(log\_entry)  
 self.send\_response(400)  
 self.end\_headers()  
 self.wfile.write(b"WAF: HTTP Request Smuggling detected.\n")  
 return

* Calls detect\_smuggling().
* If detected, logs the incident, sends a 400 Bad Request, and writes a warning to the client.

**Forwarding Safe Requests**

try:  
 forward\_headers = {k: v for k, v in self.headers.items()}  
 resp = requests.request(  
 method=self.command,  
 url=FORWARD\_URL + self.path,  
 headers=forward\_headers,  
 data=body,  
 allow\_redirects=False,  
 timeout=10  
 )  
 self.send\_response(resp.status\_code)  
 for k, v in resp.headers.items():  
 self.send\_header(k, v)  
 self.end\_headers()  
 self.wfile.write(resp.content)  
except Exception as e:  
 logging.error(f"[WAF] Error forwarding request: {e}")  
 self.send\_response(502)  
 self.end\_headers()  
 self.wfile.write(b"WAF: Error forwarding request.\n")

* Forwards the request to the backend using the requests library.
* Sends the backend’s response status, headers, and body back to the client.
* On error, logs the error, sends a 502 Bad Gateway, and writes an error message.

**Server Startup**

if \_\_name\_\_ == '\_\_main\_\_':  
 with socketserver.ThreadingTCPServer(('', LISTEN\_PORT), WAFRequestHandler) as httpd:  
 print(f'[WAF] Listening on port {LISTEN\_PORT}...')  
 httpd.serve\_forever()

* Starts the server using ThreadingTCPServer for handling multiple connections.
* Binds to all interfaces on LISTEN\_PORT.
* serve\_forever() keeps the server running.

**Summary Table: Key Built-ins and Types**

|  |  |  |
| --- | --- | --- |
| Element | Type/Function | Example/Role |
| dict | Built-in type | {'a': 1, 'b': 2} |
| list | Built-in type | [^1\_1][^1\_2] |
| str.lower() | String method | 'ABC'.lower() # 'abc' |
| re.compile(pattern, flags) | Regex compilation | re.compile(r'\d+', re.IGNORECASE) |
| logging.basicConfig() | Logging config | Sets up logging |
| requests.request() | HTTP request | Sends HTTP request |
| self.rfile.read(n) | Read bytes from stream | Reads request body |
| self.send\_response(code) | Send HTTP status | Sends response code to client |
| self.send\_header(k, v) | Send HTTP header | Sends header to client |
| self.end\_headers() | End headers section | Marks end of HTTP headers |
| self.wfile.write(data) | Write response body | Writes data to client |

This script is a practical example of a Python-based WAF proxy, using standard library modules and the requests library to inspect and forward HTTP traffic, with a focus on detecting HTTP Request Smuggling attacks—an area you are interested in[[1]](#fn1)[[2]](#fn2).

⁂

**TE\_OBFUSCATED = re.compile(r'transfer[\s\_-]*encoding\s*:', re.IGNORECASE)**

CL\_OBFUSCATED = re.compile(r'content[\s\_-]*length\s*:', re.IGNORECASE)

how to modify this function if i need to add much more obfuscated headers.

To make your code easily extensible for detecting many more obfuscated headers, you should generalize the pattern creation process. Instead of hardcoding each regex, use a list of header names and dynamically generate regex patterns for each. This makes it simple to add new headers to check for obfuscation.

**Example Approach**

**1. Define a list of header names you want to check:**

OBFUSCATED\_HEADER\_NAMES = [  
 'transfer-encoding',  
 'content-length',  
 'x-forwarded-for',  
 'host',  
 # Add more header names here as needed  
]

**2. Create a function to generate regex patterns for obfuscated headers:**

import re  
  
def make\_obfuscated\_header\_regex(header\_name):  
 # Replace dash with a pattern that matches dash, underscore, or whitespace, zero or more times  
 pattern = re.sub(r'-', r'[\s\_\-]\*', header\_name)  
 # Add optional whitespace before colon  
 return re.compile(rf'{pattern}\s\*:', re.IGNORECASE)

**3. Build a list of compiled regex patterns:**

OBFUSCATED\_HEADER\_PATTERNS = [make\_obfuscated\_header\_regex(name) for name in OBFUSCATED\_HEADER\_NAMES]

**4. Use these patterns in your detection logic:**

def detect\_smuggling(headers, body):  
 for raw\_header in headers:  
 for pattern in OBFUSCATED\_HEADER\_PATTERNS:  
 if pattern.match(raw\_header):  
 return True, f'Obfuscated header detected: {raw\_header}'  
 # ...rest of your logic...  
 return False, ''

**How This Helps**

* **Easy extensibility:** Just append new header names to OBFUSCATED\_HEADER\_NAMES.
* **Consistent detection:** All headers are checked using the same flexible obfuscation logic.
* **Maintainability:** No need to manually write a regex for every new header.

**Example:**  
If you add "x-forwarded-for" to the list, the regex will match obfuscated forms like x\_forwarded\_for:, x forwarded-for:, etc., helping you catch more smuggling attempts[[3]](#fn3).

This approach is particularly useful for web security work, such as HTTP Request Smuggling detection, which you are interested in[[3]](#fn3).

⁂

**TE\_OBFUSCATED = re.compile(r'transfer[\s\_-]*encoding\s*:', re.IGNORECASE)**

CL\_OBFUSCATED = re.compile(r'content[\s\_-]*length\s*:', re.IGNORECASE)

explain this

These two lines create regular expression (regex) patterns to detect obfuscated HTTP headers that are commonly targeted in HTTP Request Smuggling attacks—a topic you are interested in[[4]](#fn4).

**Explanation of Each Line**

TE\_OBFUSCATED = re.compile(r'transfer[\s\_\-]\*encoding\s\*:', re.IGNORECASE)

* **Purpose:** Matches any variant of the Transfer-Encoding: header, even if it is obfuscated with spaces, underscores, or dashes.
* **Breakdown:**
  + transfer: Matches the word "transfer".
  + [\s\_\-]\*: Matches zero or more spaces (\s), underscores (\_), or dashes (-). This allows for "transfer encoding", "transfer\_encoding", "transfer-encoding", or any combination (e.g., "transfer\_ -encoding").
  + encoding: Matches the word "encoding".
  + \s\*: Matches zero or more whitespace characters before the colon.
  + :: Matches the colon that ends the header name.
  + re.IGNORECASE: Makes the match case-insensitive, so it will match "Transfer-Encoding:", "transfer-encoding:", etc.

**Examples matched:**

* Transfer-Encoding:
* Transfer\_Encoding:
* transfer -encoding :
* TRANSFER\_ENCODING:

CL\_OBFUSCATED = re.compile(r'content[\s\_\-]\*length\s\*:', re.IGNORECASE)

* **Purpose:** Matches any variant of the Content-Length: header, with similar obfuscation.
* **Breakdown:**
  + content: Matches the word "content".
  + [\s\_\-]\*: Matches zero or more spaces, underscores, or dashes.
  + length: Matches the word "length".
  + \s\*: Matches optional whitespace before the colon.
  + :: Matches the colon.
  + re.IGNORECASE: Case-insensitive.

**Examples matched:**

* Content-Length:
* Content\_Length:
* content -length :
* CONTENT\_LENGTH:

**Why Use This?**

Attackers may try to bypass security filters by inserting spaces, underscores, or dashes into header names. These regex patterns help detect such obfuscated headers, which is crucial in web security and HTTP Request Smuggling prevention[[4]](#fn4).

⁂

**explain the full code that i gave at begining. i need to modify the pattern matching and all , so where to modify for**

1. adding mode pattern matching
2. changing the attack type
3. change what to do if attack detected
4. action after attack detected or no attack detected
5. if i need to modify or normalize the header , sanitize it to normal header form where to modify and what to modify
6. log for both clean request and attack request.

give me every details and modified codes and functions.

Here’s a comprehensive breakdown of your WAF proxy code, with detailed explanations and guidance on how and where to modify it for your specific needs in pattern matching, attack type handling, response actions, header normalization, and logging. This is tailored for your interests in web security and HTTP Request Smuggling[[5]](#fn5)[[6]](#fn6).

**1. Adding More Pattern Matching**

**Where to Modify:**  
The pattern matching for obfuscated headers is currently defined by the regex variables (TE\_OBFUSCATED, CL\_OBFUSCATED) and their usage in the detect\_smuggling function.

**How to Extend:**

* Use a list of header names and dynamically generate regex patterns.
* Replace the static regex variables with a more flexible system.

**Modified Code Example:**

import re  
  
# List of header names to check for obfuscation  
OBFUSCATED\_HEADER\_NAMES = [  
 'transfer-encoding',  
 'content-length',  
 # Add more header names as needed  
]  
  
def make\_obfuscated\_header\_regex(header\_name):  
 pattern = re.sub(r'-', r'[\s\_\-]\*', header\_name)  
 return re.compile(rf'{pattern}\s\*:', re.IGNORECASE)  
  
OBFUSCATED\_HEADER\_PATTERNS = [make\_obfuscated\_header\_regex(name) for name in OBFUSCATED\_HEADER\_NAMES]

**Update the detection logic:**

for raw\_header in headers:  
 for pattern in OBFUSCATED\_HEADER\_PATTERNS:  
 if pattern.match(raw\_header):  
 return True, f'Obfuscated header detected: {raw\_header}'

**2. Changing the Attack Type**

**Where to Modify:**

* The detect\_smuggling function contains the logic for detecting specific attack types.
* To detect new attack types, add new checks in this function.

**How to Extend:**

* Add new conditions for other HTTP attacks (e.g., header injection, CRLF injection).

**Example:**

def detect\_smuggling(headers, body):  
 # Existing checks...  
  
 # Example: Detect CRLF injection in headers  
 for key, value in headers.items():  
 if '\r' in key or '\n' in key or '\r' in value or '\n' in value:  
 return True, f'CRLF injection detected in header: {key}'  
  
 # Add more attack checks as needed  
 return False, ''

**3. Change What to Do If Attack Detected**

**Where to Modify:**

* In the handle\_request method, after detected, reason = detect\_smuggling(self.headers, body).

**How to Extend:**

* Modify the block that handles detected attacks to change the response, log differently, or trigger an alert.

**Example:**

if detected:  
 log\_entry = f"[WAF] HTTP Request Smuggling detected: {reason} | Path: {self.path} | Headers: {dict(self.headers)}"  
 logging.warning(log\_entry)  
 # Custom action: send alert, block IP, etc.  
 self.send\_response(403) # Forbid instead of Bad Request  
 self.end\_headers()  
 self.wfile.write(b"WAF: Attack detected and blocked.\n")  
 return

**4. Action After Attack Detected or No Attack Detected**

**Where to Modify:**

* Still in handle\_request.

**How to Extend:**

* After attack detected: You can add code to notify admins, block IPs, etc.
* After no attack: You can log clean requests, modify response, etc.

**Example:**

if detected:  
 # ...existing block...  
 return  
else:  
 # Log clean request  
 logging.info(f"[WAF] Clean request: {self.command} {self.path} | Headers: {dict(self.headers)}")  
 # Proceed to forward the request

**5. Normalize or Sanitize Headers**

**Where to Modify:**

* Before forwarding the request, after smuggling detection but before calling requests.request.

**How to Extend:**

* Normalize header names (e.g., remove extra spaces, convert underscores/dashes to standard form).

**Example Function:**

def normalize\_headers(headers):  
 normalized = {}  
 for k, v in headers.items():  
 # Remove spaces, convert underscores/dashes to dashes, lowercase  
 key = re.sub(r'[\s\_]+', '-', k).strip().lower()  
 normalized[key] = v  
 return normalized

**Usage:**

forward\_headers = normalize\_headers(self.headers)

**6. Log for Both Clean and Attack Requests**

**Where to Modify:**

* In handle\_request, after both attack detection and clean forwarding.

**How to Extend:**

* Use logging.info for clean, logging.warning or logging.error for attacks.

**Example:**

if detected:  
 logging.warning(f"[WAF] Attack detected: {reason} | Path: {self.path} | Headers: {dict(self.headers)}")  
 # ...response...  
else:  
 logging.info(f"[WAF] Clean request: {self.command} {self.path} | Headers: {dict(self.headers)}")

**Full Example of Modified Key Sections**

import re  
import logging  
  
OBFUSCATED\_HEADER\_NAMES = [  
 'transfer-encoding',  
 'content-length',  
 # Add more as needed  
]  
  
def make\_obfuscated\_header\_regex(header\_name):  
 pattern = re.sub(r'-', r'[\s\_\-]\*', header\_name)  
 return re.compile(rf'{pattern}\s\*:', re.IGNORECASE)  
  
OBFUSCATED\_HEADER\_PATTERNS = [make\_obfuscated\_header\_regex(name) for name in OBFUSCATED\_HEADER\_NAMES]  
  
def normalize\_headers(headers):  
 normalized = {}  
 for k, v in headers.items():  
 key = re.sub(r'[\s\_]+', '-', k).strip().lower()  
 normalized[key] = v  
 return normalized  
  
def detect\_smuggling(headers, body):  
 for raw\_header in headers:  
 for pattern in OBFUSCATED\_HEADER\_PATTERNS:  
 if pattern.match(raw\_header):  
 return True, f'Obfuscated header detected: {raw\_header}'  
  
 # Add more attack detection logic here...  
  
 return False, ''  
  
class WAFRequestHandler(http.server.BaseHTTPRequestHandler):  
 def handle\_request(self):  
 content\_length = int(self.headers.get('Content-Length', 0))  
 body = self.rfile.read(content\_length) if content\_length else b''  
  
 detected, reason = detect\_smuggling(self.headers, body)  
 if detected:  
 logging.warning(f"[WAF] Attack detected: {reason} | Path: {self.path} | Headers: {dict(self.headers)}")  
 self.send\_response(403)  
 self.end\_headers()  
 self.wfile.write(b"WAF: Attack detected and blocked.\n")  
 return  
 else:  
 logging.info(f"[WAF] Clean request: {self.command} {self.path} | Headers: {dict(self.headers)}")  
  
 # Normalize headers before forwarding  
 forward\_headers = normalize\_headers(self.headers)  
 # ...forward request as before...

**Summary Table**

|  |  |  |
| --- | --- | --- |
| Requirement | Where to Modify | What to Modify/Add |
| Add pattern matching | Regex creation & detection | Add to OBFUSCATED\_HEADER\_NAMES and regex logic |
| Change attack type | detect\_smuggling function | Add new detection logic |
| Change detection action | handle\_request | Change response code, logging, notification, etc. |
| Action after detection | handle\_request | Add code after detection/clean check |
| Normalize headers | Before forwarding | Add normalize\_headers function and use it |
| Log all requests | handle\_request | Add logging.info for clean, logging.warning for attack |

This structure allows you to quickly adapt your WAF for new attack patterns, response strategies, and logging requirements, supporting your ongoing cybersecurity and web security projects[[5]](#fn5)[[6]](#fn6)[[7]](#fn7).

⁂

1. interests.web\_security

1. projects.cybersecurity

1. interests.web\_security

1. interests.web\_security

1. interests.web\_security

1. projects.cybersecurity

1. interests.cybersecurity