#!/usr/bin/env python3  
import asyncio  
import argparse  
import logging  
import sys  
import re  
from datetime import datetime  
from httptools import HttpRequestParser  
  
# ------------------ CONFIGURATION ------------------  
  
DEFAULT\_LISTEN\_PORT = 7070  
DEFAULT\_BACKEND\_HOST = '127.0.0.1'  
DEFAULT\_BACKEND\_PORT = 8080  
  
# ------------------ LOGGING SETUP ------------------  
  
logger = logging.getLogger("pywaf")  
logger.setLevel(logging.INFO)  
ch = logging.StreamHandler()  
formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')  
ch.setFormatter(formatter)  
logger.addHandler(ch)  
  
# ------------------ RULE ENGINE ------------------  
  
class DetectionRule:  
 def \_\_init\_\_(self, name, pattern, description, block\_on\_detect=True):  
 self.name = name  
 self.pattern = pattern  
 self.description = description  
 self.block\_on\_detect = block\_on\_detect  
  
 def match(self, headers, raw\_request):  
 return self.pattern(headers, raw\_request)  
  
def rule\_cl\_te(headers, raw\_request):  
 # Both Content-Length and Transfer-Encoding present  
 return 'content-length' in headers and 'transfer-encoding' in headers  
  
def rule\_te\_cl(headers, raw\_request):  
 # Both Transfer-Encoding and Content-Length present (same as above but for clarity)  
 return 'transfer-encoding' in headers and 'content-length' in headers  
  
def rule\_cl\_cl(headers, raw\_request):  
 # Duplicate Content-Length headers  
 return len([k for k in headers\_raw\_list(raw\_request) if k.lower() == 'content-length']) > 1  
  
def rule\_te\_te(headers, raw\_request):  
 # Duplicate Transfer-Encoding headers  
 return len([k for k in headers\_raw\_list(raw\_request) if k.lower() == 'transfer-encoding']) > 1  
  
def rule\_obfuscated\_headers(headers, raw\_request):  
 # Obfuscated headers: extra spaces, tabs, line folding, non-printable, null bytes  
 obf\_pattern = re.compile(rb'^(?:[^\r\n:]+)[ \t]+:', re.MULTILINE)  
 null\_bytes = b'\x00' in raw\_request  
 folding = re.search(rb'\r\n[ \t]+', raw\_request)  
 return bool(obf\_pattern.search(raw\_request) or null\_bytes or folding)  
  
def headers\_raw\_list(raw\_request):  
 # Extracts raw header names from the raw request bytes  
 headers = []  
 lines = raw\_request.split(b'\r\n')  
 for line in lines[1:]:  
 if not line or b':' not in line:  
 break  
 header = line.split(b':', 1)[^1\_0].decode('latin1', errors='replace')  
 headers.append(header)  
 return headers  
  
DETECTION\_RULES = [  
 DetectionRule("CL-TE", rule\_cl\_te, "Both Content-Length and Transfer-Encoding headers present"),  
 DetectionRule("TE-CL", rule\_te\_cl, "Both Transfer-Encoding and Content-Length headers present"),  
 DetectionRule("CL-CL", rule\_cl\_cl, "Duplicate Content-Length headers"),  
 DetectionRule("TE-TE", rule\_te\_te, "Duplicate Transfer-Encoding headers"),  
 DetectionRule("Obfuscated-Headers", rule\_obfuscated\_headers, "Obfuscated or malformed headers"),  
]  
  
# ------------------ CONFIGURATION HANDLING ------------------  
  
class WAFConfig:  
 def \_\_init\_\_(self, args):  
 self.listen\_port = args.listen\_port  
 self.backend\_host = args.backend\_host  
 self.backend\_port = args.backend\_port  
 self.log\_all = args.log\_all  
 self.block = args.block  
 self.rules = DETECTION\_RULES  
  
# ------------------ HTTP PARSING ------------------  
  
class HTTPRequest:  
 def \_\_init\_\_(self):  
 self.method = None  
 self.url = None  
 self.headers = {}  
 self.body = b''  
 self.complete = False  
 self.raw = b''  
  
 def on\_url(self, url):  
 self.url = url  
  
 def on\_header(self, name, value):  
 self.headers[name.lower()] = value  
  
 def on\_headers\_complete(self):  
 pass  
  
 def on\_body(self, body):  
 self.body += body  
  
 def on\_message\_complete(self):  
 self.complete = True  
  
def parse\_http\_request(data):  
 req = HTTPRequest()  
 parser = HttpRequestParser(req)  
 try:  
 parser.feed\_data(data)  
 except Exception as e:  
 logger.warning(f"Failed to parse HTTP request: {e}")  
 return req  
  
def extract\_headers(raw\_request):  
 # Returns dict of headers from raw request bytes  
 headers = {}  
 lines = raw\_request.split(b'\r\n')  
 for line in lines[1:]:  
 if not line or b':' not in line:  
 break  
 k, v = line.split(b':', 1)  
 headers[k.strip().lower().decode('latin1')] = v.strip().decode('latin1', errors='replace')  
 return headers  
  
# ------------------ DETECTION ENGINE ------------------  
  
def detect\_attack(headers, raw\_request, rules):  
 detected = []  
 for rule in rules:  
 if rule.match(headers, raw\_request):  
 detected.append(rule)  
 return detected  
  
# ------------------ PROXY LOGIC ------------------  
  
class WAFProxy(asyncio.Protocol):  
 def \_\_init\_\_(self, config):  
 self.config = config  
 self.transport = None  
 self.peername = None  
 self.buffer = b''  
 self.headers = {}  
 self.request\_detected = False  
  
 def connection\_made(self, transport):  
 self.transport = transport  
 self.peername = transport.get\_extra\_info('peername')  
  
 def data\_received(self, data):  
 self.buffer += data  
 # Try to parse a complete HTTP request  
 if b'\r\n\r\n' not in self.buffer:  
 return # Wait for more data  
  
 # Parse headers and body  
 headers = extract\_headers(self.buffer)  
 req\_obj = parse\_http\_request(self.buffer)  
 detected\_rules = detect\_attack(headers, self.buffer, self.config.rules)  
 attack\_detected = len(detected\_rules) > 0  
  
 # Logging  
 now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 src\_ip = self.peername[^1\_0] if self.peername else 'unknown'  
 status = "SUSPICIOUS" if attack\_detected else "BENIGN"  
 logger.info(f"[{now}] {src\_ip} {status} Request: {req\_obj.method} {req\_obj.url}")  
  
 if self.config.log\_all or attack\_detected:  
 logger.info(f"Headers: {headers}")  
 if attack\_detected:  
 logger.warning(f"Attack Detected: {[r.name for r in detected\_rules]}")  
  
 # Terminal printout  
 print(f"\n[{now}] {src\_ip} {status} - {req\_obj.method} {req\_obj.url}")  
 if attack\_detected:  
 print(f" >> Detected: {[r.name for r in detected\_rules]}")  
  
 # Block or forward  
 if attack\_detected and self.config.block:  
 self.transport.write(b"HTTP/1.1 403 Forbidden\r\nContent-Length: 0\r\n\r\n")  
 self.transport.close()  
 return  
  
 # Forward to backend  
 asyncio.create\_task(self.forward\_to\_backend(self.buffer))  
  
 async def forward\_to\_backend(self, request\_data):  
 try:  
 reader, writer = await asyncio.open\_connection(  
 self.config.backend\_host, self.config.backend\_port  
 )  
 writer.write(request\_data)  
 await writer.drain()  
 response = await reader.read(65536)  
 self.transport.write(response)  
 self.transport.close()  
 writer.close()  
 await writer.wait\_closed()  
 except Exception as e:  
 logger.error(f"Error forwarding to backend: {e}")  
 self.transport.close()  
  
# ------------------ MAIN ------------------  
  
def parse\_args():  
 parser = argparse.ArgumentParser(description="Async Python WAF for HTTP Request Smuggling Detection")  
 parser.add\_argument('--listen-port', type=int, default=DEFAULT\_LISTEN\_PORT, help='Port to listen on (default: 7070)')  
 parser.add\_argument('--backend-host', type=str, default=DEFAULT\_BACKEND\_HOST, help='Backend server host (default: 127.0.0.1)')  
 parser.add\_argument('--backend-port', type=int, default=DEFAULT\_BACKEND\_PORT, help='Backend server port (default: 8080)')  
 parser.add\_argument('--log-all', action='store\_true', help='Log all requests, not just attacks')  
 parser.add\_argument('--block', action='store\_true', help='Block detected attacks (otherwise forward)')  
 return parser.parse\_args()  
  
def main():  
 args = parse\_args()  
 config = WAFConfig(args)  
 loop = asyncio.get\_event\_loop()  
 server\_coro = loop.create\_server(lambda: WAFProxy(config), '0.0.0.0', config.listen\_port)  
 server = loop.run\_until\_complete(server\_coro)  
 logger.info(f"WAF listening on 0.0.0.0:{config.listen\_port}, forwarding to {config.backend\_host}:{config.backend\_port}")  
 try:  
 loop.run\_forever()  
 except KeyboardInterrupt:  
 logger.info("WAF shutting down...")  
 finally:  
 server.close()  
 loop.run\_until\_complete(server.wait\_closed())  
 loop.close()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

* **Reverse Proxy**: Listens on a configurable port, forwards to backend, preserves original headers/body.
* **Detection Engine**: Modular, rule-based detection for CL-TE, TE-CL, CL-CL, TE-TE, and header obfuscation[[1]](#fn1).
* **Runtime Options**: CLI flags for logging, blocking, ports, backend.
* **Logging**: Timestamp, source IP, request status, headers, and attack details.
* **Extensible**: Add/remove/update rules in the DETECTION\_RULES list.
* **Async**: High-performance, asyncio-based, suitable for research/test environments.
* **Well-Commented**: Code is organized and annotated for clarity.

python waf.py --listen-port 7070 --backend-host 127.0.0.1 --backend-port 8080 --log-all --block

* To only log attacks and not block: omit --block.
* To log all requests: add --log-all.
* Change backend/listen ports as needed.

This WAF is suitable for research and testing in environments focused on HTTP Request Smuggling and header obfuscation[[2]](#fn2)[[3]](#fn3). For more advanced detection or production use, further enhancements (such as full HTTP/1.1 pipelining support, config file parsing, or integration with security event systems) can be added as needed.

⁂

**Threaded**

**Features**

* **Threaded Server:** Handles multiple requests concurrently using Python's socketserver.ThreadingMixIn
* **Logging:** All requests are logged to a file (.log) with detailed information.

**Full Working Code**

import socketserver  
import threading  
import logging  
from datetime import datetime  
  
# Configure logging to file  
logging.basicConfig(  
 filename='waf\_requests.log',  
 level=logging.INFO,  
 format='%(asctime)s %(message)s',  
 datefmt='%Y-%m-%d %H:%M:%S'  
)  
  
def detect\_hrs(raw\_request):  
 # Extract header names (case-insensitive)  
 headers = []  
 lines = raw\_request.split('\r\n')  
 for line in lines[1:]:  
 if not line or ':' not in line:  
 break  
 header\_name = line.split(':', 1)[^2\_0].strip().lower()  
 headers.append(header\_name)  
 cl\_count = headers.count('content-length')  
 te\_count = headers.count('transfer-encoding')  
 # HRS detection rules  
 if 'content-length' in headers and 'transfer-encoding' in headers:  
 return True  
 if cl\_count > 1 or te\_count > 1:  
 return True  
 return False  
  
class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):  
 def handle(self):  
 try:  
 data = b''  
 self.request.settimeout(1.0)  
 while b'\r\n\r\n' not in data:  
 chunk = self.request.recv(1024)  
 if not chunk:  
 break  
 data += chunk  
 except Exception:  
 pass  
 raw\_request = data.decode('latin1', errors='replace')  
 hrs\_detected = detect\_hrs(raw\_request)  
 if hrs\_detected:  
 status\_code = 403  
 response\_body = 'HTTP/1.1 403 Forbidden\r\nContent-Length: 0\r\n\r\n'  
 else:  
 status\_code = 200  
 response\_body = 'HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n'  
 self.request.sendall(response\_body.encode('latin1'))  
 log\_msg = f'Status: {status\_code}, HRS Detected: {hrs\_detected}, Request from: {self.client\_address[^2\_0]}'  
 logging.info(log\_msg)  
 now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 print(f'{now} | Status: {status\_code} | HRS Detected: {hrs\_detected}')  
  
class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):  
 allow\_reuse\_address = True  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 HOST, PORT = '0.0.0.0', 7070  
 with ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler) as server:  
 print(f'WAF running on {HOST}:{PORT} (threaded)')  
 try:  
 server.serve\_forever()  
 except KeyboardInterrupt:  
 print('\nShutting down WAF...')  
 server.shutdown()  
 server.server\_close()

⁂

**Sample Code**

pip install httptools aiohttp

import asyncio  
import logging  
import re  
from datetime import datetime  
from httptools import HttpRequestParser  
import aiohttp  
  
FORWARD\_URL = 'http://localhost:7070'  
LISTEN\_PORT = 7070  
LOG\_FILE = 'waf\_smuggling.log'  
  
# Configure logging  
logging.basicConfig(filename=LOG\_FILE, level=logging.INFO, format='%(asctime)s %(message)s')  
  
# Regex patterns for obfuscated headers  
TE\_OBFUSCATED = re.compile(rb'transfer[\s\_\-]\*encoding\s\*:', re.IGNORECASE)  
CL\_OBFUSCATED = re.compile(rb'content[\s\_\-]\*length\s\*:', re.IGNORECASE)  
  
class HTTPRequest:  
 def \_\_init\_\_(self):  
 self.method = None  
 self.url = None  
 self.headers = []  
 self.headers\_dict = {}  
 self.body = b''  
 self.complete = False  
  
 def on\_url(self, url):  
 self.url = url.decode('latin1', errors='replace')  
  
 def on\_header(self, name, value):  
 n = name.decode('latin1', errors='replace')  
 v = value.decode('latin1', errors='replace')  
 self.headers.append((n, v))  
 self.headers\_dict.setdefault(n.lower(), []).append(v)  
  
 def on\_headers\_complete(self):  
 pass  
  
 def on\_body(self, body):  
 self.body += body  
  
 def on\_message\_complete(self):  
 self.complete = True  
  
def detect\_smuggling(req: HTTPRequest, raw: bytes):  
 headers = req.headers\_dict  
 header\_names = [k.lower() for k, \_ in req.headers]  
  
 # 1. CL-TE or TE-CL: Both headers present  
 has\_cl = 'content-length' in headers  
 has\_te = 'transfer-encoding' in headers  
 if has\_cl and has\_te:  
 te\_values = ','.join(headers['transfer-encoding']).lower()  
 cl\_values = ','.join(headers['content-length'])  
 if 'chunked' in te\_values:  
 return True, f'Both Content-Length ({cl\_values}) and Transfer-Encoding ({te\_values}) headers present.'  
  
 # 2. Duplicate headers (CL-CL, TE-TE)  
 if has\_cl and len(headers['content-length']) > 1:  
 return True, f'Duplicate Content-Length headers: {headers["content-length"]}'  
 if has\_te and len(headers['transfer-encoding']) > 1:  
 return True, f'Duplicate Transfer-Encoding headers: {headers["transfer-encoding"]}'  
  
 # 3. Obfuscated headers (raw scan)  
 for line in raw.split(b'\r\n'):  
 if TE\_OBFUSCATED.match(line):  
 return True, f'Obfuscated Transfer-Encoding header: {line.decode("latin1", errors="replace")}'  
 if CL\_OBFUSCATED.match(line):  
 return True, f'Obfuscated Content-Length header: {line.decode("latin1", errors="replace")}'  
  
 # 4. Malformed TE or CL values  
 if has\_te:  
 for v in headers['transfer-encoding']:  
 val = v.strip().lower()  
 if val not in ['chunked', 'identity', '']:  
 return True, f'Unusual Transfer-Encoding value: {val}'  
 if val == 'chunked' and not req.body.endswith(b'0\r\n\r\n'):  
 return True, 'Transfer-Encoding chunked without proper chunked body ending.'  
  
 if has\_cl:  
 for v in headers['content-length']:  
 try:  
 cl\_val = int(v.strip())  
 if cl\_val != len(req.body):  
 return True, f'Content-Length mismatch: header={cl\_val}, actual={len(req.body)}'  
 except Exception:  
 return True, f'Invalid Content-Length value: {v}'  
  
 # 5. Conflicting TE values  
 if has\_te:  
 for v in headers['transfer-encoding']:  
 val = v.lower()  
 if ',' in val and 'chunked' in val and 'identity' in val:  
 return True, f'Conflicting Transfer-Encoding values: {val}'  
  
 return False, ''  
  
class WAFProtocol(asyncio.Protocol):  
 def \_\_init\_\_(self, backend\_url):  
 self.transport = None  
 self.backend\_url = backend\_url  
 self.buffer = b''  
 self.peername = None  
  
 def connection\_made(self, transport):  
 self.transport = transport  
 self.peername = transport.get\_extra\_info('peername')  
  
 def data\_received(self, data):  
 self.buffer += data  
 if b'\r\n\r\n' not in self.buffer:  
 return # Wait for full headers  
  
 # Parse HTTP request  
 req = HTTPRequest()  
 parser = HttpRequestParser(req)  
 try:  
 parser.feed\_data(self.buffer)  
 except Exception as e:  
 self.respond(400, b'Bad Request')  
 return  
  
 detected, reason = detect\_smuggling(req, self.buffer)  
 status\_code = 403 if detected else 200  
 now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 src\_ip = self.peername[^3\_0] if self.peername else 'unknown'  
 log\_entry = {  
 'time': now,  
 'src\_ip': src\_ip,  
 'status': status\_code,  
 'smuggling': detected,  
 'reason': reason,  
 'method': req.method,  
 'url': req.url,  
 'headers': req.headers,  
 }  
 # Log every request  
 logging.info(str(log\_entry))  
 # Print minimal terminal output  
 print(f"{now} | Status: {status\_code} | HRS Detected: {detected}")  
  
 if detected:  
 self.respond(403, b"WAF: HTTP Request Smuggling detected.\n")  
 return  
  
 # Forward to backend  
 asyncio.create\_task(self.forward\_request(self.buffer))  
  
 async def forward\_request(self, raw\_request):  
 try:  
 # Parse method and path for forwarding  
 req = HTTPRequest()  
 parser = HttpRequestParser(req)  
 parser.feed\_data(raw\_request)  
 url = self.backend\_url + (req.url or '/')  
 headers = {}  
 for k, v in req.headers:  
 headers.setdefault(k, v)  
 async with aiohttp.ClientSession() as session:  
 async with session.request(  
 method=req.method or 'GET',  
 url=url,  
 headers=headers,  
 data=req.body,  
 allow\_redirects=False,  
 timeout=10  
 ) as resp:  
 body = await resp.read()  
 response = (  
 f"HTTP/1.1 {resp.status} {resp.reason}\r\n" +  
 ''.join(f"{k}: {v}\r\n" for k, v in resp.headers.items()) +  
 "\r\n"  
 ).encode('latin1') + body  
 self.transport.write(response)  
 except Exception as e:  
 self.respond(502, b"WAF: Error forwarding request.\n")  
 finally:  
 self.transport.close()  
  
 def respond(self, code, body):  
 response = (  
 f"HTTP/1.1 {code} {'Forbidden' if code == 403 else 'OK'}\r\n"  
 f"Content-Length: {len(body)}\r\n"  
 f"Connection: close\r\n\r\n"  
 ).encode('latin1') + body  
 self.transport.write(response)  
 self.transport.close()  
  
async def main():  
 loop = asyncio.get\_running\_loop()  
 server = await loop.create\_server(  
 lambda: WAFProtocol(FORWARD\_URL),  
 '0.0.0.0', LISTEN\_PORT  
 )  
 print(f'[WAF] Listening on port {LISTEN\_PORT}...')  
 async with server:  
 await server.serve\_forever()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 asyncio.run(main())

1. pip install httptools aiohttp