Shebang and Imports

```
#!/usr/bin/env python3
import asyncio
import argparse
import logging
import sys
import re
from datetime import datetime
from httptools import HttpRequestParser
```

- **`#!/usr/bin/env python3`**
  This is a *shebang* line. It tells Unix-like systems to use Python 3 to run this script.

- **`import asyncio`**
  Imports the `asyncio` module for writing asynchronous code (e.g., handling many connections at once without blocking).
  Example:

  ```
  import asyncio
  async def hello():
      print("Hello, world!")
  asyncio.run(hello())
  ```

- **`import argparse`**
  Used for parsing command-line arguments.
  Example:

  ```
  import argparse
  parser = argparse.ArgumentParser()
  parser.add_argument('--name')
  args = parser.parse_args()
  print(args.name)
  ```

- **import logging**

  Provides a flexible logging system for status and error messages.

  Example:

  ```
  import logging
  logging.basicConfig(level=logging.INFO)
  logging.info("This is an info message")
  ```

- **import sys**

  Gives access to system-specific parameters and functions.

  Example:

  ```
  import sys
  print(sys.version)
  ```

- **import re**

  For regular expressions (pattern matching in text).

  Example:

  ```
  import re
  match = re.search(r'\d+', 'abc123')
  print(match.group())  # Output: 123
  ```

- **from datetime import datetime**

  Imports the `datetime` class to work with dates and times.

  Example:

  ```
  from datetime import datetime
  now = datetime.now()
  print(now)
  ```

- **from httptools import HttpRequestParser**

  Imports an HTTP request parser from the `httptools` library for parsing raw HTTP requests.

  Example:

  ```
  from httptools import HttpRequestParser
  # Used in the code for parsing HTTP requests.
  ```

Configuration

```
DEFAULT_LISTEN_PORT = 7070
DEFAULT_BACKEND_HOST = '127.0.0.1'
DEFAULT_BACKEND_PORT = 8080
```

Sets default configuration values for the proxy server:

- Listen on port 7070
- Forward to backend host 127.0.0.1 (localhost) on 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)
```

- Creates a logger named "pywaf".
- Sets log level to INFO.
- Adds a stream handler (prints logs to terminal).
- Formats log messages to include time, level, and message.

Example log output:

```
2025-06-19 07:00:00 INFO WAF listening on 0.0.0.0:7070, forwarding to 127.0.0.1:8080
```

Rule Engine

DetectionRule Class

```
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)
```

- Represents a security rule.
- Each rule has a name, a function (`pattern`) to check, a description, and whether to block if detected.
- `match()` runs the rule's function.

Rule Functions

Each function checks for a specific HTTP request anomaly:

```
def rule_cl_te(headers, raw_request):
    return 'content-length' in headers and 'transfer-encoding' in headers
```

- Detects if both `Content-Length` and `Transfer-Encoding` headers are present (a common attack vector).

```
def rule_te_cl(headers, raw_request):
    return 'transfer-encoding' in headers and 'content-length' in headers
```

- Same as above, written for clarity.

```
def rule_cl_cl(headers, raw_request):
    return len([k for k in headers_raw_list(raw_request) if k.lower() == 'content-length']) > 1
```

- Detects duplicate `Content-Length` headers.

```
def rule_te_te(headers, raw_request):
    return len([k for k in headers_raw_list(raw_request) if k.lower() == 'transfer-encoding']) > 1
```

- Detects duplicate `Transfer-Encoding` headers.

```
def rule_obfuscated_headers(headers, raw_request):
    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)
```

- Detects obfuscated or malformed headers (e.g., extra spaces, tabs, line folding, or null bytes).

Helper Function

```
def headers_raw_list(raw_request):
    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
```

- Extracts all header names from the raw HTTP request.

Rule List

```
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"),
]
```

- List of rules to check each HTTP request against.

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
```

- Reads configuration from command-line arguments and stores them.

HTTP Parsing

HTTPRequest Class

```
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
```

- Used by `HttpRequestParser` to store HTTP request data as it is parsed.

Parsing Functions

```
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
```

- Parses a raw HTTP request using `httptools`.

```
def extract_headers(raw_request):
    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
```

- Extracts headers from the raw request as a dictionary.

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
```

- Checks each rule against the request and returns a list of detected issues.

Proxy Logic

WAFProxy Class

```
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
        if b'\r\n\r\n' not in self.buffer:
            return  # Wait for more data

        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

        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:
```

```python
            logger.info(f"Headers: {headers}")
            if attack_detected:
                logger.warning(f"Attack Detected: {[r.name for r in detected_rules]}")

        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]}")

        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

        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()
```

- Handles each client connection.
- Receives data, parses it, checks for attacks, logs, and either blocks or forwards the request to the backend.

Main Functionality

```
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()
```

- Sets up command-line arguments for configuration.

```
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()
```

- Parses arguments, creates the server, and starts the event loop.

```
if __name__ == "__main__":
    main()
```

- Ensures the script runs only if executed directly (not imported).

Summary Table: Key Libraries/Modules

Library/Module	Purpose	Example Usage
asyncio	Asynchronous programming	Handling multiple connections
argparse	Command-line argument parsing	`--listen-port 8080`
logging	Logging events	`logging.info("Message")`
sys	System-specific parameters	`sys.version`
re	Regular expressions	`re.search(r'\d+', 'abc123')`
datetime	Date and time handling	`datetime.now()`
httptools	HTTP parsing	`HttpRequestParser(req)`

This code is a basic asynchronous Web Application Firewall (WAF) that listens for HTTP requests, checks them for suspicious patterns, logs details, and either blocks or forwards them to a backend server[1][2][3][4].

⁂

WAFProxy Class

The `WAFProxy` class is the core of the proxy server. It inherits from `asyncio.Protocol`, which means it handles network connections asynchronously.

Key Methods:

- **`__init__(self, config)`**:
  Initializes the proxy with configuration, sets up buffers for incoming data, and prepares to track the client connection.
- **`connection_made(self, transport)`**:
  Called when a client connects. Stores the transport (connection) object and the client's address.

- **data_received(self, data)**:
  Called whenever data arrives from the client. It buffers the data, waits until a full HTTP request is received (detected by `\r\n\r\n`), then:
  - Extracts headers and parses the HTTP request.
  - Runs detection rules to check for suspicious patterns.
  - Logs request details, including whether it is suspicious.
  - If an attack is detected and blocking is enabled, sends a 403 Forbidden response and closes the connection.
  - Otherwise, forwards the request to the backend server using an asynchronous task.
- **forward_to_backend(self, request_data)**:
  Asynchronously connects to the backend server, sends the request, reads the response, and relays it back to the client. Handles errors gracefully by logging and closing the connection[5][6].

Proxy Logic

The proxy logic is implemented within the `WAFProxy` class and its methods:

- **Accepts client connections** using asyncio's event-driven model.
- **Buffers incoming data** until a complete HTTP request is received.
- **Parses and inspects the request** for suspicious patterns (like HTTP smuggling or obfuscated headers).
- **Logs all activity** for monitoring and debugging.
- **Blocks or forwards** the request based on detection results and configuration.
- **Forwards requests** to the backend server and relays the response to the client[5][6].

This design allows the proxy to act as a gatekeeper, filtering out malicious or malformed requests before they reach the backend.

Main Functionality

The main functionality is set up in the `main()` function:

1. **Parses command-line arguments** (e.g., listening port, backend host/port, logging, blocking) using `argparse`.
2. **Initializes configuration** with these arguments.
3. **Creates an asyncio event loop** and starts the proxy server on the specified port.
4. **Runs the event loop**, handling connections and requests as they arrive.
5. **Gracefully shuts down** on keyboard interrupt, closing the server and event loop cleanly[5][6].

Detection Engine

The detection engine is responsible for identifying suspicious HTTP requests:

- **DetectionRule class**:
  Encapsulates a rule, including its name, detection function, description, and whether it should block on detection.

- **Rule functions**:
  Functions like `rule_cl_te`, `rule_cl_cl`, etc., check for:

  - Both `Content-Length` and `Transfer-Encoding` headers present (potential smuggling).
  - Duplicate headers.
  - Obfuscated or malformed headers (spaces, tabs, null bytes, line folding).

- **detect_attack(headers, raw_request, rules)**:
  Iterates over all rules, applies each to the request, and returns a list of detected rule violations[5][6].

Parsing Using httptools

The code uses the `httptools` library for robust HTTP parsing:

- **HTTPRequest class**:
  Implements callback methods (`on_url`, `on_header`, etc.) that `HttpRequestParser` calls as it parses the request.

- **parse_http_request(data)**:
  Creates an `HTTPRequest` object and a `HttpRequestParser` instance, then feeds the raw request data

to the parser.
This extracts the HTTP method, URL, headers, and body in a reliable way, even for edge cases or malformed requests.

- **extract_headers(raw_request)**:
  Separately parses headers from the raw request bytes for use by detection rules[5][6].

Summary Table

Component	Purpose
WAFProxy class	Handles client connections, buffers data, detects attacks, blocks/forwards requests
Proxy logic	Accepts, inspects, logs, and relays HTTP requests between client and backend
Main functionality	Sets up configuration, starts event loop, runs/shuts down server
Detection engine	Applies security rules to detect suspicious/malicious HTTP requests
httptools parsing	Robustly parses HTTP requests, extracting method, URL, headers, and body

This structure ensures suspicious requests are detected and either blocked or logged before reaching your backend, using efficient asynchronous code and robust parsing[5][6].

⁂

explain this:

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)

Here’s a line-by-line explanation of the logging setup code:

```
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)
```

Explanation

- **logger = logging.getLogger("pywaf")**
  This creates (or retrieves) a logger object named `"pywaf"`.
  - *Purpose*: Organizes logs under a specific name, so you can control logging for this component separately from others[7][8][9].
- **logger.setLevel(logging.INFO)**
  Sets the minimum level of messages that the logger will handle to `INFO`.
  - *Effect*: Only messages with level `INFO` or higher (e.g., `WARNING`, `ERROR`) will be processed. Lower-level messages like `DEBUG` will be ignored[7][8][9].
- **ch = logging.StreamHandler()**
  Creates a stream handler, which outputs log messages to the standard output (typically the console)[7][8][9].
- **formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')**
  Defines the format for log messages.
  - *%(asctime)s*: Timestamp of the log entry.
  - *%(levelname)s*: Log level (INFO, WARNING, etc.).
  - *%(message)s*: The actual log message[7][8][9].
- **ch.setFormatter(formatter)**
  Applies the formatter to the stream handler, so all messages handled by `ch` will use the specified format[7][8][9].
- **logger.addHandler(ch)**
  Adds the configured handler (`ch`) to the logger.
  - *Effect*: Log messages from `"pywaf"` will be output to the console in the specified format[7][8][9].

Example Output

When the logger is used like this:

```
logger.info("WAF started")
```

The output might look like:

```
2025-06-19 07:05:00 INFO WAF started
```

Why This Matters

- **Readability**: Log entries are timestamped and clearly labeled by severity.
- **Organization**: Using a named logger (`"pywaf"`) helps manage logs for different parts of a larger application.
- **Flexibility**: You can easily adjust log levels or redirect output (for example, to a file) by changing the handler or formatter settings[7][8][9].

This setup is a best practice for Python applications that need clear, manageable logging—especially in security or server software[9][10].

⁂

explain this:

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)
```

Here's a detailed explanation of the `DetectionRule` class and how it works, line by line:

```
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)
```

Line-by-Line Explanation

- **class DetectionRule:**
  This defines a new class called `DetectionRule`. A class in Python is a blueprint for creating objects (instances) that group together data and functions.

- **def __init__(self, name, pattern, description, block_on_detect=True):**
  This is the constructor method. It runs whenever a new `DetectionRule` object is created.

  - `name`: A string, the name of the rule (e.g., "CL-TE").
  - `pattern`: A function that takes headers and the raw request as input and returns `True` if the rule matches (i.e., if something suspicious is detected).
  - `description`: A string describing what the rule checks for.
  - `block_on_detect`: A boolean (default `True`). If `True`, the request should be blocked if this rule matches.

- **self.name = name**
  Stores the rule's name in the object.

- **self.pattern = pattern**
  Stores the detection function in the object.

- **self.description = description**
  Stores the human-readable description.

- **self.block_on_detect = block_on_detect**
  Stores whether to block the request if this rule matches.

- **def match(self, headers, raw_request):**
  This method checks if the rule matches a given HTTP request.

  - `headers`: The parsed headers of the HTTP request.
  - `raw_request`: The raw bytes of the HTTP request.

- **return self.pattern(headers, raw_request)**
  Calls the pattern function with the headers and raw request. Returns `True` if the rule is triggered, `False` otherwise.

Example Usage

Suppose you want to detect requests that have both `Content-Length` and `Transfer-Encoding` headers (a common HTTP smuggling vector):

```
def rule_cl_te(headers, raw_request):
    return 'content-length' in headers and 'transfer-encoding' in headers

# Create a DetectionRule instance for this rule:
cl_te_rule = DetectionRule(
    name="CL-TE",
    pattern=rule_cl_te,
    description="Both Content-Length and Transfer-Encoding headers present"
)

# Example headers:
headers = {'content-length': '10', 'transfer-encoding': 'chunked'}
raw_request = b"POST / HTTP/1.1\r\nContent-Length: 10\r\nTransfer-Encoding:
chunked\r\n\r\n"

# Check if the rule matches:
if cl_te_rule.match(headers, raw_request):
```

```
    print("Attack detected!")
```

This would print `Attack detected!` because both headers are present, and the rule's pattern function returns `True`[11][12].

Why Use This Pattern?

- **Flexibility**: Each rule can have a different detection function.
- **Organization**: Rules are objects with names and descriptions, making them easy to manage, log, and report.
- **Extensibility**: You can add new rules by simply writing a new function and creating a new `DetectionRule` instance.

In summary, the `DetectionRule` class is a reusable way to define, manage, and check security rules for HTTP requests in your Web Application Firewall[11][12].

⁂

explain this with example:

def rule_cl_te(headers, raw_request):
return 'content-length' in headers and 'transfer-encoding' in headers

def rule_te_cl(headers, raw_request):
return 'transfer-encoding' in headers and 'content-length' in headers

Here’s an explanation of the two functions:

```
def rule_cl_te(headers, raw_request):
    return 'content-length' in headers and 'transfer-encoding' in headers

def rule_te_cl(headers, raw_request):
    return 'transfer-encoding' in headers and 'content-length' in headers
```

What Do These Functions Do?

Both functions are **detection rules** used in a Web Application Firewall (WAF) to spot a potentially dangerous situation in HTTP requests:

The presence of both the Content-Length and Transfer-Encoding headers in a single HTTP request.

This is a classic indicator of an HTTP Request Smuggling attack, where ambiguity between these headers can be exploited to bypass security controls or poison caches[13][14][15].

How They Work

- Both functions accept two arguments:
    - `headers`: a dictionary of HTTP headers (with header names as keys, usually in lowercase).
    - `raw_request`: the raw HTTP request bytes (not used in these two functions, but present for consistency).
- They return `True` if **both** headers are present in the request, otherwise `False`.

Why Are There Two Functions?

They do the same check but are named differently for clarity and to match different rule naming conventions. This can help with logging and reporting, making it clear which rule was triggered[13][14].

Example

Suppose you receive this HTTP request:

```
POST / HTTP/1.1
Host: example.com
Content-Length: 10
Transfer-Encoding: chunked

[body]
```

When parsed, the headers dictionary will look like:

```
headers = {
    'host': 'example.com',
    'content-length': '10',
    'transfer-encoding': 'chunked'
}
```

Now, calling either detection rule:

```
print(rule_cl_te(headers, b""))  # Outputs: True
print(rule_te_cl(headers, b""))  # Outputs: True
```

Result:
Both functions return `True` because both `content-length` and `transfer-encoding` are present in the headers.

Why Is This Dangerous?

If both headers are present, different servers or proxies may interpret the request differently—some using `Content-Length`, others using `Transfer-Encoding`. This ambiguity can be abused by attackers to "smuggle" malicious requests past security controls, making this a critical check in web security[13][14][15].

Summary Table

Function	What it Checks	Example Output
rule_cl_te	Both `content-length` and `transfer-encoding` in headers	True
rule_te_cl	Both `transfer-encoding` and `content-length` in headers	True

In short:
These functions are simple but vital checks to detect a dangerous combination of HTTP headers that can lead to HTTP Request Smuggling attacks[13][14][15].

References:

⁂

explaint his with example for regex and what all patterns will match in this regex?

def rule_obfuscated_headers(headers, raw_request):

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)

Let's break down the function and regex, then show examples of what will and won't match.

```
def rule_obfuscated_headers(headers, raw_request):
    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)
```

What Does This Function Do?

This detection rule checks if an HTTP request contains **obfuscated or malformed headers**. It looks for three things:

1. **Obfuscated header lines:** Headers with extra spaces or tabs before the colon.
2. **Null bytes:** Presence of the `\x00` byte, which is non-printable and suspicious.
3. **Header line folding:** Lines that start with whitespace after a newline (an old, now-unsupported way to split headers).

If any of these are found, the function returns `True` (suspicious).

The Regex: `rb'^(?:[^\r\n:]+)[ \t]+:'` (with `re.MULTILINE`)

Explanation

- `^` — Start of a line (because of `re.MULTILINE`, this is after every `\n`, not just the start of the string)[16].
- `(?:[^\r\n:]+)` — One or more characters that are NOT carriage return (`\r`), newline (`\n`), or colon (`:`). This matches the header name.
- `[ \t]+` — One or more spaces or tabs (the "obfuscation" part: normal headers should not have spaces/tabs here).
- `:` — A literal colon, which separates the header name from its value.

So:
This matches any header line where there are extra spaces or tabs between the header name and the colon.

What Will Match?

- `X-Test : value` (multiple spaces before colon)
- `Header\t\t:` (tabs before colon)
- `User-Agent : value` (single space before colon, which is technically allowed by old specs but is suspicious)
- `Foo :bar`
- `A\t: b` (tab before colon)

What Won't Match?

- `X-Test: value` (no space before colon)
- `Host:example.com`
- `Header:value`
- `Foo: bar` (space after colon is fine, not before)

Example Inputs and Matches

Header Line	Match?	Why?

`X-Test : value`	Yes	Multiple spaces before colon
`User-Agent : value`	Yes	Single space before colon
`Foo\t\t:bar`	Yes	Tabs before colon
`Host: example.com`	No	No space before colon
`Header:value`	No	No space before colon
`Foo: bar`	No	No space before colon

What About Null Bytes and Folding?

- **Null bytes:** If the raw request contains `\x00` anywhere, it's flagged.
  - Example: `GET / HTTP/1.1\r\nHost:\x00example.com\r\n\r\n` → Match (null byte present)
- **Folding:** If any header line starts with whitespace after a newline (e.g., `\r\n\tvalue`), it's flagged.
  - Example:

```
X-Test: value\r\n\tfolded-value
```

This is flagged as folding.

Summary Table

Pattern/Condition	Example (raw header)	Match?	Reason
Spaces before colon	`Foo : bar`	Yes	Obfuscated
Tabs before colon	`Foo\t: bar`	Yes	Obfuscated
No space before colon	`Foo: bar`	No	Normal
Null byte	`Host:\x00example.com`	Yes	Null byte present
Header folding	`X-Test: value\r\n\tfolded-value`	Yes	Folding detected

Why Is This Important?

Obfuscated headers can be used by attackers to bypass security filters, confuse HTTP parsers, or exploit vulnerabilities. Detecting them is crucial for web security, especially in a Web Application Firewall[17][18][19].

In short:
This rule uses a regex to catch headers with extra spaces/tabs before the colon, and also flags null bytes or folded headers as suspicious—helping detect HTTP header obfuscation attacks[19][17].

⁂

explain this with example:

def rule_cl_cl(headers, raw_request):
return len([k for k in headers_raw_list(raw_request) if k.lower() == 'content-length']) > 1

def rule_te_te(headers, raw_request):
return len([k for k in headers_raw_list(raw_request) if k.lower() == 'transfer-encoding']) > 1

Here's a detailed explanation of these two functions, including how they work and examples:

```
def rule_cl_cl(headers, raw_request):
    return len([k for k in headers_raw_list(raw_request) if k.lower() == 'content-length']) > 1

def rule_te_te(headers, raw_request):
    return len([k for k in headers_raw_list(raw_request) if k.lower() == 'transfer-encoding']) > 1
```

What Do These Functions Do?

- **rule_cl_cl**: Checks if the HTTP request contains **more than one Content-Length header** (i.e., duplicate `Content-Length` headers).

- **rule_te_te**: Checks if the HTTP request contains **more than one Transfer-Encoding header** (i.e., duplicate `Transfer-Encoding` headers).

If duplicates are found, the function returns `True` (meaning the request is suspicious or malformed); otherwise, it returns `False`[20][21].

How Do They Work?

Both functions use a helper called `headers_raw_list(raw_request)`, which extracts all header names from the raw HTTP request (as they appear, including duplicates).

- They create a list of all header names (case-insensitive).
- They count how many times the target header (either `content-length` or `transfer-encoding`) appears.
- If the count is greater than 1, they return `True`.

Example: Duplicate Content-Length

Suppose you receive this HTTP request (raw bytes, simplified for clarity):

```
POST / HTTP/1.1\r\n
Host: example.com\r\n
Content-Length: 10\r\n
Content-Length: 20\r\n
\r\n
[some body]
```

The `headers_raw_list(raw_request)` function will extract:

```
['Host', 'Content-Length', 'Content-Length']
```

Now, calling the function:

```
rule_cl_cl({}, raw_request)  # returns True
```

Because `Content-Length` appears **twice**.

Example: Duplicate Transfer-Encoding

Suppose you receive this HTTP request:

```
POST / HTTP/1.1\r\n
Host: example.com\r\n
Transfer-Encoding: chunked\r\n
Transfer-Encoding: gzip\r\n
\r\n
[some body]
```

Extracted headers:

```
['Host', 'Transfer-Encoding', 'Transfer-Encoding']
```

Now, calling the function:

```
rule_te_te({}, raw_request)  # returns True
```

Because `Transfer-Encoding` appears **twice**.

Example: No Duplicates

```
POST / HTTP/1.1\r\n
Host: example.com\r\n
Content-Length: 10\r\n
\r\n
[some body]
```

Extracted headers:

```
['Host', 'Content-Length']
```

Now, calling the function:

```
rule_cl_cl({}, raw_request)  # returns False
rule_te_te({}, raw_request)  # returns False
```

Why Is This Important?

- **Duplicate Content-Length or Transfer-Encoding headers are forbidden by the HTTP standard.**
- Attackers may use duplicate headers to "smuggle" malicious requests, confuse proxies, or bypass security controls[22][23][24].
- Detecting and blocking such requests is a critical security measure for web servers and proxies.

Summary Table

Raw Headers Example	rule_cl_cl	rule_te_te	Reason
2x Content-Length	True	False	Duplicate CL
2x Transfer-Encoding	False	True	Duplicate TE
1x Content-Length, 1x Transfer-Encoding	False	False	No duplicates
3x Content-Length	True	False	Duplicate CL
1x Host only	False	False	No target headers

In summary:
These functions detect duplicate `Content-Length` or `Transfer-Encoding` headers in HTTP requests, which is a sign of malformed or potentially malicious traffic[20][21][22][23][24].

References:

⁂

explain this function with examples:

def headers_raw_list(raw_request):

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)[0].decode('latin1', errors='replace')

headers.append(header)

return headers

Here's a detailed explanation of the function:

```
def headers_raw_list(raw_request):
    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)[^8_0].decode('latin1', errors='replace')
        headers.append(header)
    return headers
```

What Does This Function Do?

- **Purpose:**
  Extracts the names of all HTTP headers from the raw HTTP request bytes, preserving duplicates and their order.

- **Why is this useful?**
  Some attacks (like HTTP request smuggling) rely on duplicate headers, so it's important to see all header names as they appear in the raw request, not just a parsed dictionary (which would overwrite duplicates)[25][26].

How Does It Work?

1. **Splits the raw request into lines** using `\r\n` as the delimiter.
2. **Skips the request line** (e.g., `GET / HTTP/1.1`) by starting from `lines[1:]`.
3. **For each header line:**
   - If the line is empty or doesn't contain a colon (`:`), it stops (end of headers).
   - Otherwise, splits the line at the first colon to get the header name.
   - Decodes the header name from bytes to a string using `'latin1'` encoding.
   - Adds the header name to the `headers` list.
4. **Returns the list of header names** (as strings), in the order they appeared.

Example 1: Normal Headers

Input:

```
raw_request = (
    b"POST / HTTP/1.1\r\n"
    b"Host: example.com\r\n"
    b"Content-Length: 10\r\n"
    b"User-Agent: curl/7.68.0\r\n"
    b"\r\n"
    b"body"
)
```

What happens:

- Splits into lines:
  - `POST / HTTP/1.1`
  - `Host: example.com`
  - `Content-Length: 10`
  - `User-Agent: curl/7.68.0`
  - (empty line)
  - `body`

- Skips the first line.
- For each header line, extracts header name before the colon.

Output:

```
['Host', 'Content-Length', 'User-Agent']
```

Example 2: Duplicate Headers

Input:

```
raw_request = (
    b"GET / HTTP/1.1\r\n"
    b"Foo: 1\r\n"
    b"Bar: 2\r\n"
    b"Foo: 3\r\n"
    b"\r\n"
)
```

Output:

```
['Foo', 'Bar', 'Foo']
```

Notice that both `Foo` headers are preserved, which is important for detecting duplicates[25].

Example 3: Malformed or Non-Header Lines

Input:

```
raw_request = (
    b"GET / HTTP/1.1\r\n"
    b"X-Test: value\r\n"
    b"BrokenHeader\r\n"
    b"Another: header\r\n"
    b"\r\n"
```

```
)
```

- The function stops at the first line without a colon (`BrokenHeader`), so it only extracts `X-Test`.

Output:

```
['X-Test']
```

Key Points

- Only lines with a colon are considered headers.
- Stops parsing at the first empty line or line without a colon (end of headers).
- Decodes header names using `'latin1'` to handle all valid HTTP bytes.
- Preserves duplicates and order.

In summary:
This function extracts all header names from a raw HTTP request, as they appear, which is essential for detecting duplicate or malformed headers in web security tools[25][26][27].

References:

⁂

explain this , and explain how this is matched, and how these functions are calling other related functions and all in detaails.

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"),
]

Here's a detailed breakdown of the code block and how the detection rules are created, matched, and used in the program:

```
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"),
]
```

What is DETECTION_RULES?

- **DETECTION_RULES** is a list of `DetectionRule` objects.
- Each `DetectionRule` represents a specific security check (rule) for incoming HTTP requests.
- The rules are used by the detection engine to inspect every incoming HTTP request and identify suspicious or malicious patterns[28][29].

How is a DetectionRule defined?

```
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)
```

- **name**: A string identifier for the rule (e.g., "CL-TE").
- **pattern**: A function that takes the parsed headers and the raw HTTP request, returning `True` if the rule matches (i.e., the suspicious pattern is found).
- **description**: Human-readable explanation of what the rule checks.
- **block_on_detect**: If `True`, the request should be blocked if this rule matches (default is `True`).
- The `match()` method calls the `pattern` function with the given headers and raw request[28][29].

How are these rules matched?

1. Rule Functions

Each rule in `DETECTION_RULES` is associated with a function:

- **rule_cl_te**: Returns `True` if both `Content-Length` and `Transfer-Encoding` headers are present.
- **rule_te_cl**: Same as above, but named for clarity and reporting.
- **rule_cl_cl**: Returns `True` if there are duplicate `Content-Length` headers (using `headers_raw_list`).
- **rule_te_te**: Returns `True` if there are duplicate `Transfer-Encoding` headers (using `headers_raw_list`).
- **rule_obfuscated_headers**: Returns `True` if any header is obfuscated (extra spaces, tabs, line folding, null bytes, etc.)[28][29].

2. How are rules called?

When a request is received, the following happens:

- The proxy receives the raw HTTP request.
- The headers are extracted using `extract_headers(raw_request)`.
- The detection engine calls `detect_attack(headers, raw_request, rules)`:

```
def detect_attack(headers, raw_request, rules):
    detected = []
    for rule in rules:
```

```
        if rule.match(headers, raw_request):
            detected.append(rule)
    return detected
```

- For each rule in `DETECTION_RULES`, the `match()` method is called, which in turn calls the rule's pattern function.
- If the pattern function returns `True`, that rule is added to the list of detected (matched) rules[28][29].

3. Example: How a rule is matched

Suppose this HTTP request is received:

```
POST / HTTP/1.1\r\n
Host: example.com\r\n
Content-Length: 10\r\n
Transfer-Encoding: chunked\r\n
\r\n
```

- `extract_headers()` will produce:
  `{'host': 'example.com', 'content-length': '10', 'transfer-encoding': 'chunked'}`
- The detection engine will check each rule:
  - `rule_cl_te(headers, raw_request)` → `True` (both headers present)
  - `rule_te_cl(headers, raw_request)` → `True`
  - `rule_cl_cl(headers, raw_request)` → `False` (no duplicate Content-Length)
  - `rule_te_te(headers, raw_request)` → `False` (no duplicate Transfer-Encoding)
  - `rule_obfuscated_headers(headers, raw_request)` → `False` (no obfuscation)
- The detected rules will be `["CL-TE", "TE-CL"]`.

How do rule functions call other functions?

- **rule_cl_cl** and **rule_te_te** both call `headers_raw_list(raw_request)`, which returns a list of all header names as they appear in the raw request (preserving duplicates).
- **rule_obfuscated_headers** uses regex and checks for null bytes and line folding in the raw request.

- Each rule function is focused on a specific pattern, but can use helper functions as needed[28][29].

How does this fit into the full detection flow?

1. **Request received** by the proxy.
2. **Headers and raw request** are passed to the detection engine.
3. **Each DetectionRule** in `DETECTION_RULES` is checked using its `match()` method.
4. **If any rules match**, the request is flagged as suspicious, logged, and (depending on config) possibly blocked.
5. **If no rules match**, the request is forwarded to the backend server[28][29].

Visual Overview

Rule Name	Function	What it Detects
CL-TE	rule_cl_te	Both Content-Length and Transfer-Encoding headers present
TE-CL	rule_te_cl	Both Transfer-Encoding and Content-Length headers present
CL-CL	rule_cl_cl	Duplicate Content-Length headers
TE-TE	rule_te_te	Duplicate Transfer-Encoding headers
Obfuscated-Headers	rule_obfuscated_headers	Obfuscated or malformed headers (spaces, tabs, folding, etc)

Example: Full Call Chain

1. **Proxy receives data** →
2. `data_received()` parses headers and calls `detect_attack(headers, raw_request, config.rules)` →
3. For each rule in `DETECTION_RULES`, `rule.match(headers, raw_request)` is called →
4. `match()` calls the rule function (e.g., `rule_cl_te`) →
5. If the function returns `True`, the rule is added to detected rules →

6. Detected rules are logged and may cause the request to be blocked[28][29].

Summary:

- `DETECTION_RULES` is a list of security rules, each defined by a name, a detection function, and a description.
- When a request comes in, each rule's function is called with the headers and raw request.
- If any rule matches (returns `True`), the request is flagged as suspicious.
- Some rules use helper functions (like `headers_raw_list`) to analyze the raw request for duplicates or obfuscation.
- This modular design makes it easy to add, remove, or modify rules for HTTP request inspection[28][29][30][31].

References:

⁂

explain this fully, explain ehat is self, args, init and everything with example:

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

Let's break down the `WAFConfig` class and its constructor, explaining every element and showing how it's used in the context of your Web Application Firewall (WAF) project.

The Code

```
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
```

What is a Class?

A **class** in Python is a blueprint for creating objects. It groups together data (attributes) and functions (methods) that operate on that data.

Example:

```
class Dog:
    def __init__(self, name):
        self.name = name

my_dog = Dog("Rex")
print(my_dog.name)  # Output: Rex
```

What is `self`?

- `self` is a reference to the current object (instance) of the class.
- It allows you to set or access attributes and methods on *that* specific object.
- Every method in a class (except static methods) must have `self` as its first parameter.

What is `__init__`?

- `__init__` is a special method called the **constructor**.
- It runs automatically when you create a new instance of the class.
- It is used to initialize the object's attributes with values.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 30)
print(p.name)  # Output: Alice
print(p.age)   # Output: 30
```

What is `args`?

- `args` is a parameter to the `__init__` method.
- In this context, `args` is an object that holds configuration values, typically produced by `argparse` (Python's command-line argument parsing library).
- `args` will have attributes like `listen_port`, `backend_host`, etc., based on what the user provided on the command line.

Example:

Suppose you run your script with:

```
python waf.py --listen-port 9000 --backend-host 10.0.0.2 --block
```

Then `args.listen_port` is `9000`, `args.backend_host` is `"10.0.0.2"`, and `args.block` is `True`.

What Does Each Line in `__init__` Do?

- `self.listen_port = args.listen_port`

  Saves the listening port for the WAF proxy (e.g., 7070 or 9000).

- `self.backend_host = args.backend_host`

  Saves the backend server's hostname or IP (e.g., "127.0.0.1" or "10.0.0.2").

- `self.backend_port = args.backend_port`

  Saves the backend server's port (e.g., 8080).

- `self.log_all = args.log_all`

  Whether to log all requests or only suspicious ones (Boolean).

- `self.block = args.block`

  Whether to block detected attacks (Boolean).

- `self.rules = DETECTION_RULES`

  Assigns the list of detection rules (like CL-TE, TE-CL, etc.) to this config object.

How is This Used in the Program?

1. **Parsing Arguments:**

   The script uses `argparse` to parse command-line arguments and stores them in `args`:

   ```
   args = parse_args()
   ```

2. **Creating the Config Object:**

   The config object is created and initialized with these arguments:

   ```
   config = WAFConfig(args)
   ```

3. **Using the Config:**

   The `config` object is passed to other parts of the program, such as the proxy server, so they know what settings to use:

   ```
   server_coro = loop.create_server(lambda: WAFProxy(config), '0.0.0.0',
   config.listen_port)
   ```

   Inside `WAFProxy`, you might see:

```
self.config.listen_port
self.config.block
self.config.rules
```

This lets the proxy know which port to listen on, whether to block attacks, and which rules to use for detection[32][33].

Example: Full Flow

Suppose you run:

```
python waf.py --listen-port 9000 --backend-host 10.0.0.2 --backend-port 8081 --log-all --block
```

- `args.listen_port = 9000`
- `args.backend_host = "10.0.0.2"`
- `args.backend_port = 8081`
- `args.log_all = True`
- `args.block = True`

So, after:

```
config = WAFConfig(args)
```

You get:

```
config.listen_port == 9000
config.backend_host == "10.0.0.2"
config.backend_port == 8081
config.log_all == True
config.block == True
config.rules == DETECTION_RULES
```

Summary Table

Term	Meaning
class	Blueprint for creating objects with data and functions
self	Refers to the current instance of the class
init	Constructor method, initializes object attributes
args	Object holding configuration (from command-line arguments)
.listen_port, .backend_host, etc.	Attributes set on the config object
DETECTION_RULES	List of all detection rules to use for inspecting HTTP requests

In summary:
The `WAFConfig` class is a container for all the configuration settings your WAF needs. It is initialized with the user's command-line arguments, and its attributes are used throughout the program to control how the WAF operates[32][33].

References:

⁂

explain this whole code, pass,self.,what it does and where is the call going, and where the parameter wiill go and come, with exaples:

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
```

Here's a detailed, step-by-step explanation of the `HTTPRequest` class, including what `self`, `pass`, and each method does, how parameters flow, and how this class interacts with the rest of the code—with examples.

The HTTPRequest Class

```
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
```

What is `self`?

- `self` refers to the instance of the class currently being used.
- It allows each object (instance) to keep its own data.
- Example:

```
req1 = HTTPRequest()
req2 = HTTPRequest()
req1.url = "/foo"
req2.url = "/bar"
# req1 and req2 keep their own data separate
```

What is `__init__`?

- `__init__` is the constructor method, called automatically when a new object is created.
- It sets up the initial state of the object (attributes like `self.method`, `self.url`, etc.).
- Example:

```
req = HTTPRequest()
print(req.method)  # None
```

What does `pass` mean?

- `pass` is a placeholder statement in Python.

- It does nothing and is used when a statement is required syntactically but you don't want any code to run.
- In `on_headers_complete`, it means "do nothing for now."

What does each method do?

1. on_url(self, url)

- Called when the HTTP parser finds the URL (path) in the request.
- Sets `self.url` to the given value.
- Example:

```
req = HTTPRequest()
req.on_url("/index.html")
print(req.url)  # "/index.html"
```

2. on_header(self, name, value)

- Called for each header found in the request.
- Adds the header to the `self.headers` dictionary, using a lowercase key for case-insensitivity.
- Example:

```
req.on_header("Host", "example.com")
req.on_header("Content-Length", "10")
print(req.headers)  # {'host': 'example.com', 'content-length': '10'}
```

3. on_headers_complete(self)

- Called when all headers have been parsed.
- Here, it does nothing (`pass`), but you could add code if you want to act at this point.

4. on_body(self, body)

- Called for each chunk of the request body.
- Appends the chunk to `self.body`.

- Example:

```
req.on_body(b"Hello")
req.on_body(b" World")
print(req.body)  # b"Hello World"
```

5. on_message_complete(self)

- Called when the entire HTTP message has been parsed.
- Sets `self.complete` to `True`.
- Example:

```
req.on_message_complete()
print(req.complete)  # True
```

How are these methods called?

These methods are called automatically by the `httptools.HttpRequestParser`, which is used in the `parse_http_request` function:

```
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
```

- When you call `parser.feed_data(data)`, the `httptools` parser reads the HTTP request data and calls the appropriate methods on `req` (`HTTPRequest` instance):
  - When it finds the URL, it calls `req.on_url(url)`
  - For each header, it calls `req.on_header(name, value)`
  - When headers are done, it calls `req.on_headers_complete()`
  - For each body chunk, it calls `req.on_body(body)`

- When the whole message is parsed, it calls `req.on_message_complete()`

Example: Full Flow

Suppose you have this raw HTTP request:

```
POST /submit HTTP/1.1\r\n
Host: example.com\r\n
Content-Length: 5\r\n
\r\n
hello
```

Step-by-step:

1. `parse_http_request(data)` is called with the above bytes.
2. `HTTPRequest()` creates a new object (`req`).
3. `HttpRequestParser(req)` is created.
4. `parser.feed_data(data)` parses:
   - Calls `req.on_url("/submit")`
   - Calls `req.on_header("Host", "example.com")`
   - Calls `req.on_header("Content-Length", "5")`
   - Calls `req.on_headers_complete()`
   - Calls `req.on_body(b"hello")`
   - Calls `req.on_message_complete()`
5. After parsing:
   - `req.url == "/submit"`
   - `req.headers == {"host": "example.com", "content-length": "5"}`
   - `req.body == b"hello"`
   - `req.complete == True`

Where do parameters go and come from?

- The `httptools` parser reads the HTTP bytes and calls the methods on your `HTTPRequest` object, passing the relevant data (like header name and value, body chunk, etc.) as parameters.
- You don't call these methods directly; they're called by the parser as it processes the request.

Summary Table

Method	Called by	Parameter(s)	What it does
`__init__`	Python	none	Sets up empty/default attributes
`on_url(url)`	HttpRequestParser	url (string/bytes)	Sets `self.url`
`on_header(name, value)`	HttpRequestParser	name, value	Adds to `self.headers`
`on_headers_complete()`	HttpRequestParser	none	(Does nothing here)
`on_body(body)`	HttpRequestParser	body (bytes)	Appends to `self.body`
`on_message_complete()`	HttpRequestParser	none	Sets `self.complete = True`

Why is this design used?

- **Separation of concerns:** The parser handles the protocol, your class just stores the results.
- **Flexibility:** You can extend `HTTPRequest` to do more (logging, validation, etc.) at each stage.
- **Integration:** Fits perfectly with `httptools`, which expects these callback methods.

In summary:
The `HTTPRequest` class acts as a container for parsed HTTP request data, with methods that are called by the HTTP parser as it processes each part of the request. The use of `self` ensures each instance keeps its own data, and parameters like `url`, `name`, `value`, and `body` are provided by the parser as it decodes the request, filling in the attributes of the object for later use in detection, logging, or forwarding[34][35].

⁂

explain the b' ' with examples, what is b and all, alternatives , everything:

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
```

Here's a comprehensive explanation of what `b''` means in Python, how it's used in your code, alternatives, and why it matters—especially in the context of HTTP parsing and web security tools.

What does b'' mean in Python?

- `b''` is a **bytes literal** in Python.
- The `b` prefix tells Python that the contents are **raw bytes**, not a string of Unicode characters.

- Example:

```
b'hello'  # This is a bytes object, not a string
'hello'   # This is a (Unicode) string object
```

Why use bytes (b'') instead of strings?

- **HTTP requests and responses are fundamentally sequences of bytes**, not text.
- When you read data from a network socket (as in your WAF proxy), you get bytes, not strings.
- Using bytes ensures you can handle any data, even non-printable or non-UTF-8 content, without errors.
- Many protocols (including HTTP) require precise handling of byte sequences, such as for delimiters (`b'\r\n'`), header parsing, or detecting null bytes (`b'\x00'`).

Examples of bytes usage in your code

1. Initializing empty bytes

```
self.body = b''
self.raw = b''
```

- Starts with an empty bytes object, ready to accumulate raw HTTP data.

2. Splitting and searching in bytes

```
lines = raw_request.split(b'\r\n')
if b'\r\n\r\n' not in self.buffer:
    return  # Wait for more data
```

- Splits the raw HTTP request into lines using the byte sequence for CRLF.
- Checks for the end of headers in a raw HTTP request.

3. Regex with bytes

```
obf_pattern = re.compile(rb'^(?:[^\r\n:]+)[ \t]+:', re.MULTILINE)
```

- The `rb''` prefix means a **raw bytes regex pattern**—required when matching against bytes, not strings.

4. Appending bytes

```
def on_body(self, body):
    self.body += body
```

- Appends incoming body data (as bytes) to the `self.body` attribute.

What happens if you use a string instead of bytes?

- Network data is not always valid UTF-8 or ASCII; using strings can cause decode errors.
- String operations (like `split('\r\n')`) won't work on bytes objects; you must use `b'\r\n'`.
- Regex patterns for strings (`r''`) won't work on bytes; you must use `rb''`.

How to convert between bytes and strings

- **Bytes to string:**

  ```
  b'hello'.decode('utf-8')  # 'hello'
  ```

- **String to bytes:**

  ```
  'hello'.encode('utf-8')   # b'hello'
  ```

- For HTTP headers, `'latin1'` is often used because it can represent all byte values 0–255 without error:

  ```
  b'\xff'.decode('latin1')  # 'ÿ'
  ```

Alternatives and when to use them

- Use **strings** (`''`) for text you know is Unicode and not coming from raw network data.
- Use **bytes** (`b''`) for:
    - Reading/writing files in binary mode
    - Network communication (sockets, HTTP parsing)
    - Binary protocols

Example: Why bytes are needed in HTTP parsing

Suppose you receive this raw HTTP request (as bytes):

```
raw = b'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n'
```

- To split headers, you must use `raw.split(b'\r\n')`.
- To search for the end of headers, you check `b'\r\n\r\n' in raw`.

If you tried to use strings:

```
raw = 'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n'  # This is a string, not bytes
```

- This would only work if all data is valid UTF-8 and you decoded it first.
- If the data contains binary or non-UTF-8 bytes, decoding would fail.

Summary Table

Literal	Type	Example Use Case
`'abc'`	str	Human-readable text, config files
`b'abc'`	bytes	Raw network data, HTTP parsing
`rb'abc'`	bytes	Raw bytes regex patterns

In your code

- `self.body = b''` and `self.raw = b''` ensure that HTTP body and raw data are safely handled as bytes.
- All HTTP parsing, header extraction, and regex checks are performed on bytes to avoid encoding issues and to match the true protocol format.

In summary:

- `b''` is a bytes literal, essential for handling raw network data in Python, especially for HTTP parsing and security tools.
- Always use bytes (`b''`) when working with sockets, binary data, or protocols like HTTP.
- Convert to/from strings only when necessary, and use the correct encoding (`utf-8`, `latin1`, etc.) for your context[36][37][38][39][40].

⁂

explain how this works, with example, what is try, except, exception, and how it works here, logger, warning, req, everything with example:

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

Here's a detailed explanation of the function and all its parts:

```
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
```

What Does This Function Do?

- **Purpose:**
  It parses raw HTTP request data (as bytes) into a structured `HTTPRequest` object, using the `httptools` library.
  If parsing fails, it logs a warning but still returns the (possibly incomplete) `HTTPRequest` object.

Step-by-Step Breakdown

1. `req = HTTPRequest()`

- **Creates a new instance** of the `HTTPRequest` class.
- This object will hold the parsed HTTP method, URL, headers, body, etc.
- Example:

```
req = HTTPRequest()
# req.method, req.url, req.headers, etc. are all empty or None at this point.
```

2. `parser = HttpRequestParser(req)`

- **Creates a new HTTP parser** from the `httptools` library.
- The parser is given the `req` object. As it parses the HTTP data, it will call methods on `req` (like `on_url`, `on_header`, etc.) to fill in the request details.
- Example:

```
parser = HttpRequestParser(req)
```

3. try: ... except Exception as e: ...

- **Purpose:**
  This is Python's error-handling mechanism.
- **try block:**
  Runs code that might raise an error (an "exception").
- **except Exception as e:**
  If any error occurs inside the `try` block, the code inside `except` runs instead of crashing the program.
- **Exception** is a built-in Python class for errors. `as e` gives you the actual error object so you can print/log it.

Example:

```
try:
    x = 1 / 0
except Exception as e:
    print(f"Error: {e}")  # Output: Error: division by zero
```

4. parser.feed_data(data)

- **Feeds the raw HTTP data** to the parser.
- As the parser processes the data, it calls methods on the `req` object to fill in its fields (method, url, headers, body, etc.).
- If the data is malformed or parsing fails, an exception will be raised.

Example:

```
raw = b"GET / HTTP/1.1\r\nHost: example.com\r\n\r\n"
parser.feed_data(raw)
# This will cause req.on_url, req.on_header, etc. to be called.
```

5. except Exception as e: logger.warning(...)

- If parsing fails (for example, the data is not a valid HTTP request), this block runs.
- **logger.warning(...)** logs a warning message using the configured logger.

- The message includes the error details (`{e}`).

Example:

If the data is malformed:

```
raw = b"BAD DATA"
parser.feed_data(raw)  # Might raise an exception
# Logs: 2025-06-19 08:00:00 WARNING Failed to parse HTTP request: [error details]
```

The logger is set up earlier in the code to print warnings with timestamps and severity levels[41][42].

6. `return req`

- Returns the `HTTPRequest` object, whether parsing succeeded or failed.
- If parsing failed, the object may have incomplete or empty fields.

How Is This Used?

- In the proxy logic, when a full HTTP request is received, the code calls:

  ```
  req_obj = parse_http_request(self.buffer)
  ```

- This gives a structured object (`req_obj`) representing the HTTP request, which can then be inspected, logged, or checked for attacks[41][42].

Example Flow

Suppose the raw HTTP data is:

```
data = b"POST /login HTTP/1.1\r\nHost: site.com\r\nContent-Length: 4\r\n\r\nuser"
req = parse_http_request(data)
print(req.url)           # '/login'
print(req.headers)       # {'host': 'site.com', 'content-length': '4'}
print(req.body)          # b'user'
print(req.complete)      # True
```

If the data is malformed:

```
data = b"BAD REQUEST"
req = parse_http_request(data)
# Logs a warning, req fields may be empty or incomplete
```

Summary Table

Line	What it does
`req = HTTPRequest()`	Creates an empty HTTPRequest object
`parser = HttpRequestParser(req)`	Sets up the parser, linking it to req
`try: parser.feed_data(data)`	Tries to parse the data, filling req via callbacks
`except Exception as e:`	If parsing fails, catch the error
`logger.warning(...)`	Logs the error using the logger
`return req`	Returns the (possibly incomplete) HTTPRequest object

Why Use Try/Except Here?

- **Prevents the whole program from crashing** if a single HTTP request is malformed.
- **Logs the error** for debugging and monitoring.
- **Continues processing** other requests.

What is logger.warning?

- `logger` is a logging object set up at the start of the program[41][42].
- `logger.warning(...)` records a warning-level message, usually printed to the console with a timestamp and severity.

In summary:

This function safely parses raw HTTP request data into a structured object, handles any errors gracefully by logging them, and always returns an HTTPRequest object for further inspection or processing[41][42].

References:

⁂

explain how this all works, in low level, and how to modify if needed, lines, split, and how all this works here, strip, decode, what is latin, :

def extract_headers(raw_request):

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

Here is a detailed, low-level explanation of how the `extract_headers` function works, how and why it uses `split`, `strip`, `decode`, and `'latin1'`, and how you could modify it if needed.

The Function

```
def extract_headers(raw_request):
    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
```

Step-by-Step: How It Works

1. **Splitting the Raw Request into Lines**

   ```
   lines = raw_request.split(b'\r\n')
   ```

   - `raw_request` is a bytes object representing the full HTTP request.
   - `split(b'\r\n')` breaks the request into lines, using the HTTP standard line ending (CRLF).
   - Example:

     ```
     raw_request = (b"GET / HTTP/1.1\r\n"
                    b"Host: example.com\r\n"
                    b"Content-Length: 10\r\n"
                    b"\r\n"
                    b"body")
     lines = [
         b"GET / HTTP/1.1",
         b"Host: example.com",
         b"Content-Length: 10",
         b"",
         b"body"
     ]
     ```

2. **Skipping the Request Line**

   ```
   for line in lines[1:]:
   ```

   - The first line (`lines`) is the request line (e.g., `GET / HTTP/1.1`), not a header.
   - The loop starts from `lines[^14_1]` to process only headers.

3. **Looping Through Header Lines**

```
if not line or b':' not in line:
    break
```

- Stops at the first empty line (`b""`), which indicates the end of headers.
- Also stops if a line does not contain a colon (`:`), which is not a valid header line.

4. **Splitting Each Header into Name and Value**

```
k, v = line.split(b':', 1)
```

- Splits the line at the first colon into a header name (`k`) and value (`v`).
- Example: `b"Host: example.com"` → `k = b"Host"`, `v = b" example.com"`

5. **Cleaning and Decoding the Header Name and Value**

```
headers[k.strip().lower().decode('latin1')] = v.strip().decode('latin1',
errors='replace')
```

- **strip()**: Removes leading and trailing whitespace (spaces, tabs) from the name and value.
  - Example: `b" Host "` → `b"Host"`
- **lower()**: Converts the header name to lowercase for case-insensitive matching.
  - Example: `"Host"` → `"host"`
- **decode('latin1')**: Converts bytes to a string using the Latin-1 (ISO-8859-1) encoding.
  - HTTP/1.x headers are defined as being Latin-1 encoded, which means all byte values 0–255 are valid and mapped directly to Unicode code points[43][44].
  - Using `'latin1'` avoids errors and preserves all possible byte values, even non-ASCII ones.
- **errors='replace'**: If a decoding error occurs (rare with `'latin1'`), it replaces problematic bytes with a replacement character.
- The cleaned, decoded, lowercase header name is used as the dictionary key; the cleaned, decoded value is the value.

6. **Returning the Headers Dictionary**

```
return headers
```

- Returns a dictionary mapping header names (as lowercase strings) to their values.

Example: What Does It Produce?

Suppose:

```
raw_request = (
    b"POST /submit HTTP/1.1\r\n"
    b"Host: example.com\r\n"
    b"Content-Length: 5\r\n"
    b"User-Agent: curl/7.68.0\r\n"
    b"\r\n"
    b"hello"
)
```

Then:

```
headers = extract_headers(raw_request)
print(headers)
# Output:
# {
#     'host': 'example.com',
#     'content-length': '5',
#     'user-agent': 'curl/7.68.0'
# }
```

Why Use latin1 for Decoding?

- HTTP/1.x headers are defined as ISO-8859-1 (Latin-1) encoded, which allows all byte values 0–255[43][44].
- Using `'utf-8'` could cause errors if the header contains non-UTF-8 bytes.

- `'latin1'` is a safe choice for HTTP header parsing, as it guarantees no decode errors and preserves all data[43][44].

How to Modify or Extend

- **To support duplicate headers:**
  Change `headers` from a dict to a list of tuples, or store values as lists.
- **To ignore case but preserve original case:**
  Store both the original and lowercase versions.
- **To use a different encoding:**
  Change `'latin1'` to `'utf-8'`, but beware of decode errors.
- **To process headers differently:**
  Add more logic inside the loop, such as logging, validation, or filtering.

What If You Used Strings Instead of Bytes?

- If you split on `'\r\n'` instead of `b'\r\n'`, you'd get errors or incorrect results if the data isn't valid UTF-8.
- You'd have to decode the entire request up front, risking decode errors and possibly losing non-ASCII data.

Summary Table

Step	What It Does	Example Input	Example Output
split	Splits raw bytes into lines at `b'\r\n'`	b'Host: x\r\nFoo: y'	[b'Host: x', b'Foo: y']
strip	Removes leading/trailing whitespace from name and value	b' Host '	b'Host'
lower	Makes header names lowercase for case-insensitive matching	b'HOST'	b'host'

decode('latin1')	Converts bytes to string, preserving all possible byte values	b'foo'	'foo'
dict assignment	Stores header name/value in a dictionary	'host': 'example.com'	{'host': 'example.com'}

References

- [paste.txt][45]
- [python: http header must be encodable in latin1][43]
- [Python: Converting from ISO-8859-1/latin1 to UTF-8][44]
- [programming.http_parsing][46]
- [programming.web_security][47]

In summary:
`extract_headers` robustly parses HTTP headers from raw bytes, using low-level operations (`split`, `strip`, `decode`) and the `'latin1'` encoding to ensure all possible header data is handled safely and correctly, as required by the HTTP standard. This makes it reliable for web security and proxy tools[45][43][44].

⁂

how this works:

def detect_attack(headers, raw_request, rules):

detected = []

for rule in rules:

if rule.match(headers, raw_request):

detected.append(rule)

return detected

The function `detect_attack` is a core part of the detection engine in your Python-based Web Application Firewall (WAF). Here's a detailed, low-level explanation of how it works, how it interacts with other components, and how you might modify or extend it:

The Code

```
def detect_attack(headers, raw_request, rules):
    detected = []
    for rule in rules:
        if rule.match(headers, raw_request):
            detected.append(rule)
    return detected
```

Step-by-Step: How It Works

1. **Inputs:**
    - `headers`: A dictionary of HTTP headers (e.g., `{'host': 'example.com', 'content-length': '10'}`).
    - `raw_request`: The raw HTTP request as bytes (e.g., `b'POST / HTTP/1.1\r\nHost: ...'`).
    - `rules`: A list of `DetectionRule` objects—each representing a security check (see below).
2. **Initialize an empty list:**

    ```
    detected = []
    ```

    - This list will collect all rules that are triggered (matched) by the request.
3. **Loop through each rule:**

    ```
    for rule in rules:
    ```

    - Each `rule` is an instance of the `DetectionRule` class.
4. **Check if the rule matches:**

```
if rule.match(headers, raw_request):
    detected.append(rule)
```

- `rule.match(headers, raw_request)` calls the `match` method of the `DetectionRule` class, which in turn calls the rule's pattern function (e.g., `rule_cl_te`, `rule_cl_cl`, etc.) with the headers and raw request as arguments.
- If the rule's pattern function returns `True`, it means the suspicious pattern was detected, and the rule is added to the `detected` list.

5. **Return all detected rules:**

```
return detected
```

- The function returns a list of all rules that matched the request. If the list is empty, no suspicious patterns were found.

How Does This Fit Into the WAF?

- When a request is received, the proxy parses the headers and raw request.
- It then calls `detect_attack(headers, raw_request, self.config.rules)` (where `self.config.rules` is typically `DETECTION_RULES`).
- The result is a list of detected attacks (rules that matched).
- If any rules are detected, the request is flagged as suspicious and may be blocked or logged in detail[48][49].

Example: End-to-End Flow

Suppose you receive this HTTP request:

```
POST / HTTP/1.1\r\n
Host: example.com\r\n
Content-Length: 10\r\n
Transfer-Encoding: chunked\r\n
```

```
\r\n
```

- `headers` will be:

  ```
  {'host': 'example.com', 'content-length': '10', 'transfer-encoding': 'chunked'}
  ```

- `raw_request` is the raw bytes of the request.
- `rules` is the list of `DetectionRule` objects.

When `detect_attack(headers, raw_request, rules)` is called:

- `rule_cl_te.match(headers, raw_request)` returns `True` (both `Content-Length` and `Transfer-Encoding` present).
- `rule_te_cl.match(headers, raw_request)` also returns `True` (same check for clarity).
- The other rules may return `False` (no duplicates, no obfuscation).
- The function returns a list with the two matching rules.

What is a DetectionRule and How Does `match` Work?

From the code:

```
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)
```

- Each rule has a name, a pattern function, and a description.
- The `match` method simply calls the pattern function with the headers and raw request.

How to Modify or Extend

- **Add a new rule:**
  Write a new pattern function (e.g., `def rule_new(headers, raw_request): ...`) and add a new `DetectionRule` to the rules list.
- **Change rule logic:**
  Edit the corresponding pattern function.
- **Change detection behavior:**
  You could modify `detect_attack` to stop at the first match, or to collect additional information about each match.

Visual Flow

Step	What Happens
Input	headers, raw_request, rules
For each rule	Call rule.match(headers, raw_request)
If match	Add rule to detected list
Return	List of all matched rules

Summary

- `detect_attack` checks a request against each detection rule and returns all rules that match (i.e., all detected attacks).
- It is central to the WAF's ability to inspect and classify incoming HTTP requests for suspicious or dangerous patterns[48][49][50].
- You can easily add, remove, or modify rules to change what your WAF detects.

References:

⁂

explain this class, each line with alternatives and examples, and what memory call it will do, which all functions it will call, and what affects when it calls, and what value it gives, and what value it gets.:

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
    if b'\r\n\r\n' not in self.buffer:
        return  # Wait for more data

    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

    now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    src_ip = self.peername[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]}")

    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]}")

    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

    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()
```

Here's a detailed, line-by-line explanation of the `WAFProxy` class, including what each line does, what values are passed and returned, which functions are called (and when), alternatives, and the effects of each call. This will help you understand the low-level flow of the proxy logic in your Python-based Web Application Firewall[51][52][53][54].

The Code

```
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
        if b'\r\n\r\n' not in self.buffer:
            return  # Wait for more data

        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

        now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        src_ip = self.peername[^16_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]}")

        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]}")

        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
```

```
        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()
```

Line-by-Line Explanation

Class Definition

```
class WAFProxy(asyncio.Protocol):
```

- Inherits from `asyncio.Protocol`, making this class suitable for asynchronous network communication.
- Used as the handler for each incoming client connection.

`__init__` Method

```
def __init__(self, config):
    self.config = config
    self.transport = None
    self.peername = None
```

```
    self.buffer = b''
    self.headers = {}
    self.request_detected = False
```

- **config**: Receives a `WAFConfig` object containing all settings (ports, backend, rules, etc.).
- **self.transport**: Will hold the connection object to the client.
- **self.peername**: Will store the client's address (IP, port).
- **self.buffer**: Accumulates incoming bytes until a full HTTP request is received.
- **self.headers**: Placeholder for parsed headers (not strictly needed here, but could be used for state).
- **self.request_detected**: Tracks if a suspicious request was detected for this connection.

Alternatives:

- You could initialize `self.headers` only when needed, or use a list if you want to support duplicate headers.

connection_made Method

```
def connection_made(self, transport):
    self.transport = transport
    self.peername = transport.get_extra_info('peername')
```

- Called automatically when a new client connects.
- **transport**: The connection object (lets you send/receive data).
- **peername**: Gets the client's IP and port for logging and auditing.

data_received Method

```
def data_received(self, data):
    self.buffer += data
    if b'\r\n\r\n' not in self.buffer:
        return  # Wait for more data
```

- Called whenever new data arrives from the client.
- **data**: The newly received bytes.
- Accumulates data in `self.buffer` until the end of HTTP headers is detected (`b'\r\n\r\n'`).
- If not enough data yet, returns and waits for more.

Parsing and Detection

```
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
```

- **extract_headers(self.buffer)**: Parses all HTTP headers from the raw bytes and returns a dictionary.
- **parse_http_request(self.buffer)**: Uses `httptools` to parse the full HTTP request into an `HTTPRequest` object (method, url, headers, body).
- **detect_attack(...)**: Checks the request against all detection rules. Returns a list of matched (triggered) rules.
- **attack_detected**: Boolean, `True` if any rules matched.

Alternatives:

- You could parse only headers first, then parse the body if needed for certain rules.

Logging and Output

```
now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
src_ip = self.peername[^16_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}")
```

- Gets the current time, client IP, and sets status based on detection.
- Logs a summary of the request.

```
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]}")
```

- If logging all requests or an attack is detected, logs headers and (if applicable) the names of triggered rules.

```
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]}")
```

- Prints a summary to the terminal for real-time monitoring.

Blocking or Forwarding

```
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
```

- If an attack is detected and blocking is enabled, sends a 403 Forbidden response to the client and closes the connection.

```
asyncio.create_task(self.forward_to_backend(self.buffer))
```

- If not blocked, creates an asynchronous task to forward the request to the backend server.

forward_to_backend Method

```
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()
```

- Asynchronously connects to the backend server (host and port from config).
- Forwards the full HTTP request.
- Waits for the backend response (up to 65536 bytes).
- Sends the backend response back to the original client, then closes both connections.
- Logs and closes the client connection if any error occurs.

Alternatives:

- You could stream the request/response in chunks for large bodies.
- You could add retry logic or more detailed error handling.

Function Calls and Data Flow

Who calls what?

- The asyncio event loop calls `connection_made` and `data_received` as data arrives.
- `data_received` calls:
    - `extract_headers` (parses headers)
    - `parse_http_request` (parses full request)
    - `detect_attack` (checks all rules)
    - `forward_to_backend` (forwards request if not blocked)

What values are passed and returned?

- `self.buffer` accumulates data from the client.

- Parsed headers and request objects are passed to detection and logging.
- The backend host/port and rules come from `self.config`.
- If an attack is detected, a 403 response is sent; otherwise, the backend's response is relayed.

What memory/context does it use?

- Uses `self.config` for all settings (from command-line args).
- Uses `self.peername` for logging client info.
- Uses `self.buffer` to handle partial/complete requests.

Example Flow

1. **Client connects:**
   → `connection_made` stores the transport and peername.
2. **Client sends data:**
   → `data_received` adds it to the buffer.
   → If a complete HTTP request is present, parses headers and request, checks for attacks.
3. **If attack detected and blocking enabled:**
   → Sends 403 Forbidden, closes connection.
4. **If no attack or not blocking:**
   → Forwards request to backend, relays response, closes connection.

How to Modify

- **To log more details:**
  Add more fields to the log or print statements.
- **To support HTTP pipelining:**
  Adjust buffer handling to process multiple requests per connection.
- **To handle large bodies:**
  Stream data in chunks rather than reading all at once.

- **To add new detection rules:**
  Add new functions and append to `DETECTION_RULES` in config.

Summary Table

Line/Method	What it does	Calls/Effects
`__init__`	Initializes state	Receives config, sets up buffer, etc.
`connection_made`	Stores transport and peername	-
`data_received`	Buffers data, parses, detects, logs, blocks/forwards	Calls extract_headers, parse_http_request, detect_attack, forward_to_backend
`forward_to_backend`	Forwards request, relays response, handles errors	Uses asyncio, closes connections

In summary:
`WAFProxy` is the main class handling each client connection in your WAF. It buffers incoming data, parses and inspects requests, logs activity, blocks or forwards traffic based on detection results, and manages all network communication asynchronously. All its behavior is driven by configuration and detection rules, making it flexible and extensible for web security tasks[51][55][54][52][53][56].

⁂

explain this main functionality, parser, what is add.argument parser.parser.args(), args, type, --, :

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()

Here's a detailed explanation of the main argument parsing functionality, including how the parser works, what each line does, what `add_argument` means, how `type`, `--`, and `args` work, and how this fits into the overall WAF application:

The Code

```
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()
```

What is argparse and the parser?

- **argparse** is a standard Python library for parsing command-line arguments.
- **ArgumentParser** is the main class; it creates a parser object that understands what arguments your script accepts.
- The parser handles user input from the command line, checks types, applies defaults, and provides help messages.

Line-by-Line Breakdown

1. Create the parser

```
parser = argparse.ArgumentParser(description="Async Python WAF for HTTP Request
Smuggling Detection")
```

- Creates a new argument parser object.
- The `description` appears in the help message (`python script.py --help`).

2. Add arguments

Each `add_argument` defines a command-line option the user can provide.

Example:

```
parser.add_argument('--listen-port', type=int, default=DEFAULT_LISTEN_PORT, help='Port
to listen on (default: 7070)')
```

- `--listen-port`: The name of the command-line option. The `--` prefix means it's an optional argument (not a positional one).
- `type=int`: The value must be an integer (e.g., `--listen-port 8081`).
- `default=DEFAULT_LISTEN_PORT`: If the user doesn't provide it, use the default value (from earlier in the code, typically 7070).
- `help`: The help text shown in `--help`.

Other arguments:

- `--backend-host`: String, default is `127.0.0.1`.
- `--backend-port`: Integer, default is 8080.
- `--log-all`: Boolean flag (if present, sets to `True`), logs all requests.
- `--block`: Boolean flag (if present, sets to `True`), blocks detected attacks.

Flags with `action='store_true'`

- For `--log-all` and `--block`, you don't provide a value; just including the flag sets the value to `True`.
- Example: `python script.py --log-all` sets `args.log_all` to `True`.

3. Parse the arguments

```
return parser.parse_args()
```

- This reads the actual command-line arguments, parses them, and returns an object (`args`) where each argument is an attribute.
- Example: If you run `python script.py --listen-port 9090 --block`, then:
    - `args.listen_port == 9090`
    - `args.block == True`
    - `args.backend_host == '127.0.0.1'` (default)

How is this used in the main program?

```
def main():
    args = parse_args()
    config = WAFConfig(args)
    # ... rest of the server setup ...
```

- `args` is passed to the configuration class, which stores all settings for use throughout the program.

What does `--` mean in arguments?

- The `--` prefix makes an argument optional and keyword-based (not positional).
- Example: `--listen-port 8081` vs. positional `8081`.

What does `type` do?

- Ensures the value is converted to the specified type (e.g., `int` for port numbers).

- If the user provides an invalid value (`--listen-port abc`), the parser will show an error.

What does `default` do?

- If the argument is not provided, uses the specified default value.

What does `help` do?

- Provides a description for each argument in the help message.

Example Usage

Command:

```
python waf.py --listen-port 9000 --backend-host 10.0.0.2 --block
```

Parsed values:

- `args.listen_port == 9000`
- `args.backend_host == "10.0.0.2"`
- `args.block == True`
- `args.backend_port == 8080` (default)
- `args.log_all == False` (default)

Alternatives

- You could use positional arguments (no `--`), but that's less flexible.
- You could use other libraries like `click` or `docopt` for more complex parsing, but `argparse` is standard and sufficient for most needs.

Summary Table

Argument	Type	Default	Example Usage	Result in `args`
--listen-port	int	7070	--listen-port 9000	args.listen_port = 9000
--backend-host	str	127.0.0.1	--backend-host 10.0.0.2	args.backend_host = ...
--backend-port	int	8080	--backend-port 8000	args.backend_port = 8000
--log-all	flag	False	--log-all	args.log_all = True
--block	flag	False	--block	args.block = True

In summary:

- `argparse.ArgumentParser()` creates a parser for command-line arguments.
- `add_argument()` defines what arguments your script accepts, their types, defaults, and help text.
- `parser.parse_args()` parses the actual command-line input and returns an object (`args`) with all values.
- This makes your script flexible, user-friendly, and easy to configure from the command line[57][58].

⁂

how to add or modify these if needed:

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()

To **add or modify command-line arguments** in your `parse_args` function using `argparse`, you just need to adjust the calls to `parser.add_argument()` in the function. Here's a full explanation of how it works, how to add or change arguments, and examples for each case:

How the Argument Parser Works

- `argparse.ArgumentParser()` creates a parser object that understands command-line options[59][60].
- `add_argument()` tells the parser what arguments your script accepts, their type, default value, and help text[59][61][62][63][60].
- `parser.parse_args()` reads the actual command-line input and returns a Namespace object (`args`) with all values as attributes.

How to Add a New Argument

Suppose you want to add an option to enable debug logging:

Add this line in `parse_args`:

```
parser.add_argument('--debug', action='store_true', help='Enable debug logging')
```

Now, if the user runs:

```
python waf.py --debug
```

Then `args.debug` will be `True`.

How to Modify an Existing Argument

Suppose you want to change the default backend port from 8080 to 9000:

Change this line:

```
parser.add_argument('--backend-port', type=int, default=9000, help='Backend server port (default: 9000)')
```

Now, if the user doesn't specify `--backend-port`, it will use 9000.

How to Add a Positional Argument

Suppose you want the user to specify a required config file:

Add this line:

```
parser.add_argument('config_file', type=str, help='Path to the configuration file')
```

Now, the user must run:

```
python waf.py myconfig.yaml
```

and `args.config_file` will be `"myconfig.yaml"`.

How to Change the Type or Add Choices

Suppose you want to restrict the log level to specific values:

Add this line:

```
parser.add_argument('--log-level', type=str, choices=['DEBUG', 'INFO', 'WARNING', 'ERROR'], default='INFO', help='Set the logging level')
```

Now, running:

```
python waf.py --log-level DEBUG
```

will set `args.log_level` to `"DEBUG"`. If the user tries an invalid value, argparse will show an error.

Example: Full Modified parse_args Function

```
def parse_args():
    parser = argparse.ArgumentParser(description="Async Python WAF for HTTP Request Smuggling Detection")
    parser.add_argument('--listen-port', type=int, default=7070, help='Port to listen on (default: 7070)')
    parser.add_argument('--backend-host', type=str, default='127.0.0.1', help='Backend server host (default: 127.0.0.1)')
    parser.add_argument('--backend-port', type=int, default=8080, 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)')
    parser.add_argument('--debug', action='store_true', help='Enable debug logging')  # New argument
    parser.add_argument('--log-level', type=str, choices=['DEBUG', 'INFO', 'WARNING', 'ERROR'], default='INFO', help='Set the logging level')  # New argument
    parser.add_argument('config_file', type=str, help='Path to the configuration file')  # New positional argument
    return parser.parse_args()
```

How to Remove an Argument

Just delete or comment out the corresponding `add_argument` line.

How to Access the Values

After calling `args = parse_args()`, you access values as attributes:

```
if args.debug:
    print("Debug mode is on")
print("Backend port:", args.backend_port)
print("Config file:", args.config_file)
```

Quick Reference Table

What you want to do	How to do it (example)
Add a flag	`parser.add_argument('--debug', action='store_true', help='Enable debug')`
Add an option with a value	`parser.add_argument('--timeout', type=int, default=30, help='Timeout in seconds')`
Add a positional argument	`parser.add_argument('input_file', type=str, help='Input file path')`
Restrict choices	`parser.add_argument('--mode', choices=['fast', 'slow'], default='fast', help='Operation mode')`
Change default	Change `default=...` in `add_argument`
Remove an argument	Delete or comment out its `add_argument` line

Summary

- **To add an argument:** Use `parser.add_argument()` with the desired options[59][62][63][60].
- **To modify an argument:** Change its parameters (type, default, help, etc.).
- **To remove an argument:** Delete its line.
- **To access argument values:** Use `args.argument_name` after parsing.

This makes your CLI flexible and easy to extend for new features or configuration options[59][61][62][63][60].

References:

⁂

explain this, config, WAFConfigs, parse_-args, 0.0.0.0, server, try,except, finally, loop.run, everything with example,:

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()

Here's a detailed, line-by-line explanation of the `main()` function, covering what each part does, how configuration and argument parsing works, what `0.0.0.0` means, how the server is started and stopped, and how exceptions and cleanup are handled—with practical examples and context.

The Code

```
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()
```

Step-by-Step Explanation

1. Argument Parsing

```
args = parse_args()
```

- Calls the `parse_args()` function, which uses `argparse` to read command-line arguments (like `--listen-port`, `--backend-host`, etc.)[66].
- Returns an `args` object where each argument is an attribute (e.g., `args.listen_port`).

Example:
If you run:

```
python waf.py --listen-port 9000 --block
```

Then:

- `args.listen_port == 9000`
- `args.block == True`

2. Configuration Object

```
config = WAFConfig(args)
```

- Creates a `WAFConfig` object, passing in the parsed arguments.
- The `WAFConfig` class stores all the main settings (listen port, backend host/port, logging preferences, detection rules, etc.)[66].

Example:

- `config.listen_port` is set to `args.listen_port`
- `config.block` is set to `args.block`

3. Get the Event Loop

```
loop = asyncio.get_event_loop()
```

- Acquires the main asyncio event loop, which is responsible for running asynchronous tasks and handling network events[67].

4. Create the Server Coroutine

```
server_coro = loop.create_server(lambda: WAFProxy(config), '0.0.0.0',
config.listen_port)
```

- Prepares a coroutine to create a TCP server.
- `lambda: WAFProxy(config)` means each new client connection will be handled by a new `WAFProxy` instance, using the current configuration.
- `'0.0.0.0'` means the server will listen on all available network interfaces (all IP addresses on the machine). This makes the server accessible from any network interface, not just localhost[66].
- `config.listen_port` is the port number to listen on (e.g., 7070 or whatever was set).

5. Start the Server

```
server = loop.run_until_complete(server_coro)
```

- Runs the coroutine to completion, actually starting the server and binding the port.
- `server` is now the running server object.

6. Log Startup

```
logger.info(f"WAF listening on 0.0.0.0:{config.listen_port}, forwarding to
{config.backend_host}:{config.backend_port}")
```

- Logs an informational message showing where the proxy is listening and where it will forward requests.

7. Run the Event Loop Forever

```
try:
    loop.run_forever()
```

- Starts the asyncio event loop, which will now handle all network events, client connections, and background tasks.
- This loop keeps running until the process is stopped (e.g., with Ctrl+C).

8. Handle Shutdown (Keyboard Interrupt)

```
except KeyboardInterrupt:
    logger.info("WAF shutting down...")
```

- If the user presses Ctrl+C (KeyboardInterrupt), this block runs.
- Logs a shutdown message.

9. Cleanup (Finally Block)

```
finally:
    server.close()
    loop.run_until_complete(server.wait_closed())
    loop.close()
```

- **server.close()**: Initiates server shutdown (stops accepting new connections).
- **loop.run_until_complete(server.wait_closed())**: Waits for the server to finish closing all connections.
- **loop.close()**: Closes the event loop, releasing all resources.

This ensures a clean and graceful shutdown, so no connections are left hanging[68].

Practical Example

Suppose you run:

```
python waf.py --listen-port 9000 --backend-host 10.0.0.2 --block
```

- The proxy listens on all interfaces, port 9000.
- Forwards requests to 10.0.0.2:8080 (default backend port).
- Blocks detected attacks.
- You can stop the server with Ctrl+C, and it will shut down gracefully.

Summary Table

Line/Step	What it Does	Example Value/Effect
`parse_args()`	Parses command-line arguments	`args.listen_port == 9000`
`WAFConfig(args)`	Creates config object with all settings	`config.block == True`
`get_event_loop()`	Gets the main asyncio event loop	
`create_server()`	Prepares a coroutine to start the server	Listens on 0.0.0.0:9000

`run_until_complete()`	Starts the server and binds port	
`run_forever()`	Runs the event loop forever (handles connections)	
`except KeyboardInterrupt`	Handles Ctrl+C shutdown	Logs shutdown message
`finally`	Closes server and event loop gracefully	Ensures no resource leaks

Key Concepts

- **0.0.0.0**: Listen on all network interfaces (public and private IPs).
- **try/except/finally**: Ensures the server can be stopped gracefully, logging shutdown and cleaning up resources.
- **parse_args/WAFConfig**: Modularizes configuration, making it easy to change how the proxy runs.
- **loop.run_forever()**: Keeps the server running, handling all asynchronous events and connections.

In summary:
This `main()` function sets up configuration, starts the event-driven proxy server on all interfaces, handles connections and detection, and ensures clean shutdown on exit—all using Python's `asyncio` for efficient, scalable network programming[66][67][68].

References:

⁂

if name == "main":

```
main()
```

The line:

```
if __name__ == "__main__":
    main()
```

is a standard Python idiom used to control how and when certain parts of your code are executed. Here's a detailed explanation of what it means, how it works, and why it's important, with examples relevant to your WAF code.

What does if __name__ == "__main__": mean?

- Every Python file (module) has a special built-in variable called `__name__`.
- If the file is being run directly (like `python waf.py`), then `__name__` is set to `"__main__"`.
- If the file is being **imported** by another Python file (e.g., `import waf`), then `__name__` is set to the module's name (e.g., `"waf"`).

So:

- The code inside this `if` block will **only run if the file is executed directly**, not if it's imported as a module elsewhere.

What happens in your code?

```
if __name__ == "__main__":
    main()
```

- When you run your script (`python waf.py`), `__name__` is `"__main__"`, so `main()` is called.
- If another script does `import waf`, the code inside this block does **not** run automatically.

Why is this important?

- **Prevents code from running unintentionally** when importing modules.
- **Makes your code reusable:** You can import your classes and functions elsewhere without starting the server or running the main logic.

- **Standard practice** in Python for scripts and libraries.

Example

Suppose your file is named `waf.py`.
If you run:

```
python waf.py --listen-port 9000 --block
```

- `__name__` is `"__main__"`, so `main()` runs, which starts your WAF server.

If you do:

```
import waf
```

- `__name__` is `"waf"`, so `main()` is **not** called.
- You can now use `waf.WAFProxy`, `waf.DETECTION_RULES`, etc., in your own code.

How does main() work in this context?

When `main()` is called, it:

1. **Parses command-line arguments** using `parse_args()` (e.g., `--listen-port`, `--block`)[69].
2. **Creates a config object** (`WAFConfig(args)`) to hold all settings[69].
3. **Gets the asyncio event loop** to manage asynchronous tasks[69].
4. **Creates and starts the server** on `0.0.0.0` (all interfaces) and the specified port[69].
5. **Logs startup info**.
6. **Runs the event loop forever** (handles connections and proxying)[69].
7. **Handles shutdown** gracefully if you press Ctrl+C (KeyboardInterrupt)[69].
8. **Cleans up resources** in the `finally` block (closes server and event loop)[69].

Visual Flow

How the file is used	What happens?
Run directly (`python waf.py`)	`main()` runs, server starts
Imported (`import waf`)	`main()` does NOT run automatically

Summary Table

Code	When it runs	What it does
`if __name__ == "__main__": main()`	Only when script is run directly	Starts the main WAF server logic
(no such block)	Always runs on import	Can cause unwanted side effects

Why is this a best practice?

- Keeps your code modular and reusable.
- Prevents accidental execution of main routines when importing.
- Makes it easy to write tests or use parts of your code in other projects.

In summary:
`if __name__ == "__main__": main()` ensures your WAF server only starts when the script is run directly, not when it's imported. This is essential for clean, modular, and reusable Python code[69][70].

References:

⁂

give full overview explanation of whole code.and where to modify for logs, terminal output, args

modification or addition, modifications, parsing logic, detection logic deletion and addition.

Here's a comprehensive overview of your Python WAF (Web Application Firewall) proxy code, including how each part works, and **where/how to modify** for logs, terminal output, argument handling, parsing, and detection logic[71][72][73][74].

Full Code Structure and Flow

1. Imports and Constants

- Imports core modules: `asyncio` (async networking), `argparse` (CLI arguments), `logging` (logs), `re` (regex), `datetime`, and `httptools` (HTTP parsing).
- Sets default values for listening port and backend server.

2. Logging Setup

- Configures a logger named `"pywaf"` to print messages with timestamps, log level, and message content[75].
- **Modify logging**:
    - Change log format or level in the logging setup section.
    - Add file logging by adding a `FileHandler` to the logger.

3. Detection Rule Engine

- **DetectionRule** class: Each rule has a name, function, description, and block flag.
- **Rule functions**:
    - `rule_cl_te`, `rule_te_cl`: Detect both `Content-Length` and `Transfer-Encoding` headers.
    - `rule_cl_cl`, `rule_te_te`: Detect duplicate headers.

- `rule_obfuscated_headers`: Detects obfuscated headers (spaces, tabs, folding, null bytes) using regex[73].

- **DETECTION_RULES**: List of all rules used for request inspection.

To add/delete/modify detection logic:

- **Add a new rule**: Write a new function, then add a `DetectionRule` to `DETECTION_RULES`.
- **Remove a rule**: Delete it from `DETECTION_RULES`.
- **Change a rule**: Edit its function.

4. Configuration Handling

- **WAFConfig** class: Stores all config values (from CLI args), including rules.
- **Modify config**:
    - To add new config options, add them to `WAFConfig` and the argument parser.

5. HTTP Parsing

- **HTTPRequest** class: Stores parsed HTTP request data (method, URL, headers, body, etc.).
- **parse_http_request**: Uses `httptools` to parse raw HTTP requests, filling the `HTTPRequest` object[74].
- **extract_headers**: Parses headers from raw bytes for rule checks.

To modify parsing logic:

- Change or extend the `HTTPRequest` class or `parse_http_request` function.
- Adjust `extract_headers` for custom header handling or encoding.

6. Detection Engine

- **detect_attack**: Runs all detection rules on a request and returns a list of matched rules.
- **Modify detection engine**:

    - Change its logic to stop at first match, or collect more info as needed.

7. Proxy Logic

- **WAFProxy (asyncio.Protocol)**: Handles each client connection.
    - Buffers incoming data until a full HTTP request is received.
    - Parses headers and request.
    - Runs detection rules.
    - Logs request info and detection results.
    - Prints a summary to the terminal.
    - Blocks malicious requests (sends 403) if configured.
    - Otherwise, forwards the request to the backend and relays the response.

To modify terminal output:

- Change the `print(...)` statements in `data_received`.

To modify backend forwarding:

- Edit `forward_to_backend` (e.g., to support HTTPS or advanced load balancing).

8. Argument Parsing

- **parse_args**: Uses `argparse` to define and parse command-line arguments.
    - `--listen-port`, `--backend-host`, `--backend-port`, `--log-all`, `--block`.

To add or modify arguments:

- Add or change lines in `parse_args` with `parser.add_argument(...)`.
- Update `WAFConfig` and other code to use the new arguments.

9. Main Functionality

- **main**:
  - Parses arguments.
  - Creates config.
  - Sets up the asyncio event loop.
  - Starts the WAF server on `0.0.0.0` (all interfaces) and the chosen port.
  - Logs startup info.
  - Runs the server until interrupted.
  - Cleans up and shuts down gracefully on exit.

10. Entry Point

- `if __name__ == "__main__": main()`
  - Ensures the server only starts when the script is run directly, not when imported.

Where to Modify for...

Feature	Where to Modify
Logs (file, format, level)	Logging setup section at the top (`logger`, `formatter`, etc.)
Terminal output	`print(...)` statements in `WAFProxy.data_received`
Argument handling	`parse_args` function (add/change/remove `parser.add_argument(...)` lines)
Detection logic	Rule functions (e.g., `rule_cl_te`), `DETECTION_RULES` list, and `detect_attack`
Parsing logic	`HTTPRequest` class, `parse_http_request`, `extract_headers`
Blocking/forwarding	In `WAFProxy.data_received` (block: send 403, else: call `forward_to_backend`)
Backend connection	`WAFProxy.forward_to_backend`

Example: Adding a New CLI Argument

To add a `--debug` flag:

1. In `parse_args`:

```
parser.add_argument('--debug', action='store_true', help='Enable debug logging')
```

2. In logging setup:

```
if args.debug:
    logger.setLevel(logging.DEBUG)
```

3. Pass `args.debug` through `WAFConfig` if needed.

Example: Adding a Detection Rule

1. Write a new function:

```
def rule_custom(headers, raw_request):
    return 'x-custom-header' in headers
```

2. Add to `DETECTION_RULES`:

```
DetectionRule("Custom-Header", rule_custom, "Detects X-Custom-Header presence"),
```

Example: Changing Terminal Output

Edit the `print(...)` lines in `WAFProxy.data_received` to show more or less info.

Summary Table

Component	Purpose	How to Modify/Add/Delete
Logging	Log events, warnings, attacks	Change logger setup, add handlers
Terminal Output	Show real-time info on screen	Edit `print(...)` in `data_received`
CLI Arguments	User config at startup	Edit `parse_args`

Detection Logic	What attacks are detected	Edit rule functions, `DETECTION_RULES`
HTTP Parsing	Understand raw HTTP requests	Edit `HTTPRequest`, `parse_http_request`
Proxy/Forwarding	Handle/block/forward requests	Edit `WAFProxy` methods

In summary:
Your code is a modular, asynchronous Python WAF proxy that inspects HTTP requests for suspicious patterns using customizable rules, logs and prints events, and can be easily extended or modified for new logging, detection, argument, or parsing needs.
To modify any part, locate the relevant section (logs, arguments, rules, parsing, etc.) and adjust the code as shown above.[71][72][75][73][74]

⁂

1. paste.txt
2. paste.txt
3. projects.cybersecurity
4. programming.log_management
5. paste.txt
6. paste.txt
7. paste.txt
8. paste.txt
9. programming.log_management
10. projects.cybersecurity
11. paste.txt
12. paste.txt
13. paste.txt
14. paste.txt

15. interests.web_security
16. https://docs.python.org/3/library/re.html
17. interests.web_security
18. projects.cybersecurity
19. paste.txt
20. paste.txt
21. paste.txt
22. https://stackoverflow.com/questions/4280153/detect-duplicated-header-in-httpresponse
23. https://github.com/haproxy/haproxy/issues/2721
24. https://stackoverflow.com/questions/78401005/duplicate-header-transfer-encoding-in-response-causes-client-exception-java
25. paste.txt
26. programming.web_security
27. interests.web_security
28. paste.txt
29. paste.txt
30. programming.web_security
31. interests.web_security
32. paste.txt
33. paste.txt
34. paste.txt
35. paste.txt
36. paste.txt
37. paste.txt

38. programming.web_security
39. interests.web_security
40. projects.cybersecurity
41. paste.txt
42. paste.txt
43. https://gist.github.com/kl09/b7160aaf4526e4a34c730f4f280c2522
44. https://stackoverflow.com/questions/6539881/python-converting-from-iso-8859-1-latin1-to-utf-8
45. paste.txt
46. programming.http_parsing
47. programming.web_security
48. paste.txt
49. paste.txt
50. programming.web_security
51. paste.txt
52. projects.cybersecurity
53. programming.http_parsing
54. programming.web_security
55. paste.txt
56. interests.web_security
57. paste.txt
58. paste.txt
59. https://docs.python.org/3/library/argparse.html
60. https://www.cherryservers.com/blog/how-to-use-python-argparse
61. https://docs.python.org/3/howto/argparse.html

62. https://realpython.com/command-line-interfaces-python-argparse/

63. https://www.cs.unb.ca/~bremner/teaching/cs2613/books/python3-doc/library/argparse.html

64. paste.txt

65. paste.txt

66. paste.txt

67. https://docs.python.org/3/library/asyncio-eventloop.html

68. https://www.reddit.com/r/learnpython/comments/c3eh0u/asyncio_how_to_cleanly_exit_event_loop_with_an/

69. paste.txt

70. paste.txt

71. paste.txt

72. paste.txt

73. programming.web_security

74. programming.http_parsing

75. programming.log_management