**WAF Architecture for HTTP Request Smuggling Detection**

WAF architecture follows a classic reverse proxy model, specifically designed to defend against HTTP Request Smuggling (HRS) by sitting between clients and your application’s frontend server. Here’s a detailed breakdown:

**1. Components**

* **Client (e.g., BurpSuite)**
  + Sends HTTP requests to the WAF.
* **WAF (Python Module)**
  + Listens on port 7070.
  + Parses and inspects every HTTP request for smuggling indicators.
  + Logs and blocks malicious requests.
  + Forwards clean requests to the frontend server.
* **Frontend Server (Go Reverse Proxy)**
  + Receives sanitized requests from the WAF.
  + Forwards them to the backend server.
* **Backend Server (Go HTTP Server)**
  + Processes the requests and generates responses.

**2. Data Flow**

[Client]  
 |  
 v  
[WAF (Python)]  
 |  
 |--[Detection & Logging]  
 | |-- If malicious: Block & Log  
 | |-- If clean: Forward  
 v  
[Frontend (Go)]  
 |  
 v  
[Backend (Go)]  
 |  
 v  
[Frontend (Go)]  
 |  
 v  
[WAF (Python)]  
 |  
 v  
[Client]

**3. WAF Internal Flow**

* **Request Reception:**  
  WAF receives the HTTP request from the client.
* **Parsing & Inspection:**  
  WAF parses headers and body, checking for:
  + Both Content-Length and Transfer-Encoding
  + Multiple or obfuscated headers
  + Malformed or conflicting header values
  + Chunked body format issues
  + Content-Length mismatches
* **Decision Point:**
  + If any smuggling indicator is found, the request is blocked and logged.
  + If the request is clean, it is forwarded to the frontend server.
* **Logging:**  
  All detection events and errors are logged for auditing.

**4. Architectural Principles**

* **Reverse Proxy Mode:**  
  The WAF acts as a reverse proxy, inspecting and controlling all inbound traffic before it reaches your application servers[[1]](#fn1)[[2]](#fn2).
* **Full Traffic Inspection:**  
  All HTTP requests are parsed and validated against a comprehensive set of smuggling detection rules[[1]](#fn1)[[2]](#fn2)[[3]](#fn3).
* **Integration:**  
  The WAF is transparent to both clients and backend infrastructure, requiring no changes to application code.
* **Extensibility:**  
  Detection logic is modular, allowing for easy updates as new smuggling techniques emerge.

**5. Security Rationale**

* **Why this architecture?**
  + HRS attacks exploit parsing discrepancies between front-end and back-end servers[[4]](#fn4)[[5]](#fn5).
  + By placing the WAF as the first line of defense, you ensure only sanitized, protocol-compliant requests reach your Go frontend and backend servers[[5]](#fn5)[[2]](#fn2).
  + Logging and blocking at the WAF layer provide visibility and rapid response to attacks[[1]](#fn1)[[2]](#fn2).

**Summary Table**

|  |  |
| --- | --- |
| Component | Role |
| Client | Sends HTTP requests |
| WAF (Python) | Inspects, detects, logs, and forwards/block |
| Frontend (Go) | Reverse proxy, logs, and forwards to backend |
| Backend (Go) | Processes requests and generates responses |

This architecture is robust, modular, and aligns with best practices for mitigating HTTP Request Smuggling in modern web applications[[4]](#fn4)[[5]](#fn5)[[1]](#fn1)[[2]](#fn2).

⁂

1. <https://www.radware.com/cyberpedia/application-security/waf-architecture/>

1. <https://onlinelibrary.wiley.com/doi/10.1155/2022/3121177>

1. <https://arxiv.org/html/2503.10846v1>

1. <https://portswigger.net/web-security/request-smuggling>

1. <https://www.vaadata.com/blog/what-is-http-request-smuggling-exploitations-and-security-best-practices/>