# SQL Injection

Protecting against SQL injection attacks is crucial for maintaining the security of your web applications. In this guide, I will provide a Lua script for HAProxy that detects potential SQL injection attempts based on common patterns in the request parameters. The script will block requests containing suspicious payloads and return an appropriate error message.

This script checks incoming request parameters for common SQL injection patterns.

Pattern Matching: The script defines a set of SQL injection patterns that it will look for in request parameters, headers, and the request body.

Request Inspection: It checks the request body, query parameters, and headers for these patterns.

Blocking: If any patterns are detected, it responds with a 403 Forbidden status and a message indicating that a potential SQL injection attack was detected.

# Custom Error Pages

Description: Serve custom error pages for specific backend errors, using Lua to generate the content dynamically based on the error condition.

Generate a personalized error message based on the client’s request details or the specific backend that failed.

# Custom Rate Limiting

Description: Implement custom rate-limiting logic based on various factors. HAProxy has basic rate-limiting, but Lua can be used to define more granular or dynamic policies.

Set a custom rate limit per user or API key.

Rate limit based on the time of day or geographic location.

Use different rate limits for specific request paths.

# Custom Redirect Logic

Use Lua to dynamically generate HTTP redirects based on request details.

Redirect users to different versions of the site based on headers or cookies (e.g., mobile vs. desktop version).

Generate SEO-friendly redirects for content migrations or rewrites.

These Lua script ideas range from improving security and performance to adding new features and improving observability.

Implementing custom redirect logic in HAProxy using Lua scripting can enhance your application by allowing you to define specific rules for redirecting users based on various criteria. Below, I will provide a Lua script that implements custom redirect logic and some curl commands to test it.

Inspect the incoming requested port

Redirect the request based on specific conditions (e.g., certain paths or query parameters).

Path and Query Inspection: The script inspects the request URI and query parameters to determine which redirect rules to apply.

Redirect Rules:

If the request URI is /old-path, it redirects to /new-path with a 301 Moved Permanently status.

If the request URI is /products and contains the query parameter category=sale, it redirects to /sale-products with a 302 Found status.

If the request URI is /old-about, it redirects to /about-us with a 307 Temporary Redirect s (preserving the HTTP method).

Whitelisting

Is a security feature that allows only specific IP addresses or ranges to access certain resources. In this example, we will write a Lua script for HAProxy that checks incoming requests against a predefined list of whitelisted IP addresses. If the request comes from a non-whitelisted IP, the script will block the request and respond with a 403 Forbidden status.

Define a list of whitelisted IP addresses.

Inspect incoming requests to check if the client's IP address is in the whitelist.

Block requests from non-whitelisted IPs.

Whitelisted IPs: Maintains a table of whitelisted IP addresses.

Client IP Retrieval: It retrieves the source IP address of the incoming request.

Whitelist Check: If the client's IP is not found in the whitelist, the script responds with a 403 Forbidden status and a custom error message.

# Blacklist

Implementing IP blacklisting is an important security measure that helps protect your application by denying access to requests from specific malicious or unwanted IP addresses. We will check incoming requests against a predefined list of blacklisted IP addresses. If a request comes from a blacklisted IP, the script will block the request and respond with a 403 Forbidden status.

Define a list of blacklisted IP addresses.

Inspect incoming requests to check if the client's IP address is in the blacklist.

Block requests from blacklisted IPs.

Client IP Retrieval: It retrieves the source IP address of the incoming request.

Blacklist Check: If the client's IP is found in the blacklist, the script responds with a 403 Forbidden status and a custom error message.

# Custom Request/Response Header Manipulation

Description: Modify HTTP headers dynamically on the fly. For example, you can strip sensitive headers, add new ones, or alter existing ones based on specific conditions (e.g., based on User-Agent, IP address, or URL path).

Strip X-Forwarded-For if it's coming from an untrusted network.

Add a custom X-RateLimit-Remaining header for clients based on IP-based rate-limiting logic.

# Authentication via External Systems

Implement custom authentication mechanisms where HAProxy communicates with an external system to verify credentials.

Verify user credentials against an OAuth server or LDAP.

Implement a custom Single Sign-On (SSO) flow where HAProxy handles login requests and forwards valid tokens to the backend.

# Geolocation-based Request Routing

Description: Use an external geolocation service or database to route traffic to specific backends based on the client's geographic location.

Route clients from Europe to a European data center, and clients from the U.S. to a U.S.-based server.

Serve different content (e.g., language-specific) based on the geolocation of the request.

# HAProxy Lua Script for Client Browser Detection

Inspect the User-Agent header to identify the browser.

Respond with a custom message based on the detected browser type (e.g., Chrome, Firefox, Safari).

User-Agent Inspection: The script retrieves the User-Agent header from the incoming request.

Browser Detection: It checks the User-Agent string for known browser identifiers (like "Chrome", "Firefox", "Safari", or "MSIE") and responds with a specific message for each browser type.

Fallback: If no User-Agent is found, it responds with a 400 Bad Request.

# Cookie size

Vacolidating cookie size in HAProxy using Lua can be useful for ensuring that your application adheres to certain constraints, especially when dealing with session management or stateful applications. Below is a Lua script that checks the size of cookies in incoming requests and responds accordingly. Additionally,

Inspect the Cookie header of incoming requests.

Calculate the size of the cookies.

Respond with an error if the total size exceeds a specified limit (e.g., 4096 bytes).

# Inspects for the presence off Local File Inclusion

Local File Inclusion (LFI) is a type of security vulnerability that allows an attacker to include files on a server through the web browser. It is essential to inspect requests for patterns that may indicate LFI attempts, such as certain directory traversal sequences (e.g., ../, ..%2f, etc.).

I will provide a Lua script for HAProxy that inspects requests for potential LFI attempts and a set of curl commands to test this functionality.

Inspect the request path and parameters.

Check for common patterns that indicate LFI attempts.

# Inspects the request body for attempts to exploit RFI (Remote File Inclusion)

Remote File Inclusion (RFI) is a type of vulnerability that allows an attacker to include remote files through the web application. This can lead to serious security issues, such as executing malicious code on the server. To help mitigate this risk, we can write a Lua script for HAProxy that inspects the request body for patterns indicative of RFI attempts.

Below, I will provide a Lua script to detect potential RFI attempts and a set of curl commands to test this functionality.

Inspect the request body for common patterns that may indicate RFI attempts (e.g., URLs, specific file types).

Block requests that appear to be attempting RFI.

Cross-Site Scripting (XSS)

Vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. To help mitigate XSS risks, we can write a Lua script for HAProxy that inspects incoming requests for potential XSS patterns.

Inspect both the request body and query parameters for common XSS patterns (e.g., <script>, javascript:, etc.).

Block requests that appear to be attempting XSS.

Request Inspection: The script retrieves the request body and query parameters of incoming requests.

Pattern Checking: It checks both the query and body for common XSS attack patterns, such as: <script> tags javascript: URIs

Event handler attributes (e.g., onclick=, onerror=)

JavaScript functions like alert( and eval(

Blocking: If any of these patterns are detected, the script responds with a 403 Forbidden status and a custom error message.

# Admin path protection

Protecting administrative paths in a web application is crucial to prevent unauthorized access. Below, I will provide a Lua script for HAProxy that checks if the requested URL contains a sensitive admin path. If the request matches this path, the script will enforce authentication checks or block access entirely based on certain conditions.

Inspect the requested path.

Check if it matches an admin path (e.g., /admin).

Respond with a 403 Forbidden if the user is not authenticated, or simply allow the request if authenticated.

The script retrieves the requested URI of incoming requests.

Admin Path Check: It checks if the requested path matches the defined admin path (in this case, /admin).

If the request does not contain valid authentication, the script responds with a 403 Forbidden status and a custom error message.

# log4j\_rce

The Log4j Remote Code Execution (RCE) vulnerability, often referred to as "Log4Shell," exploits the Java logging library Log4j. Attackers can manipulate logging messages, allowing them to execute arbitrary code on vulnerable systems. To help mitigate this risk, we can create a Lua script for HAProxy that inspects incoming requests for patterns indicative of Log4j RCE attempts.

Inspect both the request headers and body for patterns that could indicate a Log4j RCE attempt.

Block requests that appear to be attempting to exploit the vulnerability.

It checks both headers and body for common Log4j RCE patterns that attackers might use, such as: JNDI lookups (${jndi:ldap://, ${jndi:rmi://, ${jndi:http://)

Environment variable injections (${env:)

Nested variable injections (${${::)

If any of these patterns are detected, the script responds with a 403 Forbidden status and a custom error message.

# Bot detection

Bot detection and protection are crucial for web applications to prevent malicious bots from scraping data, executing attacks, or causing denial-of-service. Below is a Lua script for HAProxy that implements a basic bot detection mechanism. The script will check user-agent strings and request rates to identify and block suspected bots.

Check the user-agent header for known bot signatures.

Block requests identified as bots and respond with a 403 Forbidden status.

Known Bots: The script contains a table of known bot user-agent signatures.

Bot Detection: It checks incoming requests against this table to identify bots.

Rate Limiting: The script tracks the number of requests from each IP address within a minute. If the request count exceeds the defined limit, it blocks further requests from that IP for the remainder of the minute.

Response Handling: If a bot is detected or rate limits are exceeded, the script sends a 403 Forbidden or 429 Too Many Requests response accordingly.

# Unit testing

With curl responses can be effectively done using a shell script or a testing framework in a programming language like Python. Here, I'll demonstrate how to write a shell script to automate the testing of curl responses against expected outcomes.

Make HTTP requests using curl.

Capture the HTTP response codes and body.

Compare the actual responses to the expected outcomes.

Report the results.

Function Definition: The check\_response function takes three parameters: the URL to test, the expected HTTP status code, and a string expected in the response body. It uses curl to make the request and captures both the HTTP status and the response body.

Response Validation: The script checks if the actual status code matches the expected status code and if the response body contains the expected string. It outputs whether the test passed or failed.

Cleanup: The script removes the temporary response file at the end.