

網站應用程式的攻防實錄：剖析現代 Web 威脅與智慧防禦機制

以一個擬真電商網站「ShopEasy」為例，深入探討常見漏洞的攻擊手法與 WAF（網站應用程式防火牆）的偵測原理。

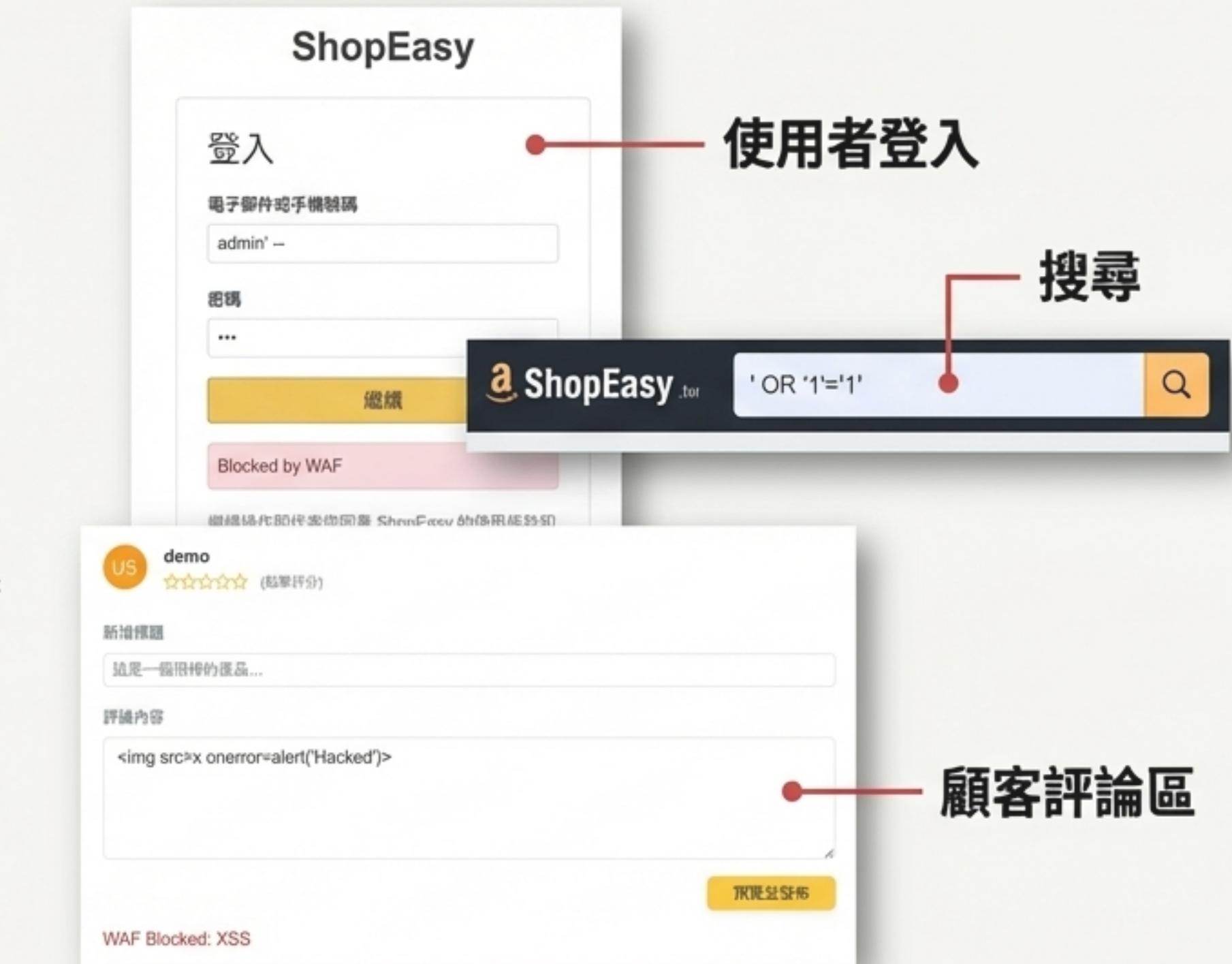


每個網站請求，都可能是一次試探或一場攻擊。本簡報將帶您深入數位戰場的前線，首先揭示攻擊者如何利用常見漏洞滲透系統，接著展示一個智慧型 WAF 如何抽絲剝繭，精準識別並攔截這些威脅。

我們的案例研究：ShopEasy 電商平台

一個功能完整的電商網站，但也因此擁有多個潛在的攻擊入口點。我們的攻防演練將圍繞以下幾個核心互動功能展開：

-  • **使用者登入 & 搜尋 (User Login & Search)**：接收使用者輸入，直接與後端資料庫互動。
-  • **顧客評論區 (Customer Reviews)**：允許使用者提交內容，並將其顯示給其他訪客。
-  • **低價回報系統 (Price-match Reporting)**：接收外部 URL，由伺服器主動發起連線。
-  • **產品文件下載 (Document Download)**：根據參數讀取並提供伺服器上的檔案。



威脅一：SQL 注入 (SQL Injection) — 當資料變成了指令

情境與原理

情境: 在登入或搜尋欄位，攻擊者輸入的不是預期的文字，而是惡意的 SQL 指令片段。

原理: 後端程式碼（例如使用 f-string）直接拼接使用者輸入，導致資料庫將輸入的資料當作「程式碼」執行，扭曲了原始查詢邏輯。

攻擊實例

- `' OR '1'='1` 利用邏輯漏洞，使 WHERE 條件永遠為真，如同「萬能鑰匙」，可繞過驗證或撈出所有資料。
- `admin' --` 利用註解符號，讓密碼驗證的 SQL 語句失效，直接以 `admin` 身分登入。

惡意輸入

ShopEasy

登入

電子郵件或手機號碼

`' OR '1'='1`

密碼

...

繼續

Blocked by WAF

WAF 擋截



Request Blocked

Your request was flagged as suspicious by the Web Application Firewall.

Attack Type: SQLI
Severity: HIGH
Payload: param.q: ' OR '1'='1'

威脅二：跨站腳本攻擊 (XSS) – 在你的瀏覽器裡執行我的程式碼

情境與原理

****情境****: 在顧客評論區，攻擊者提交一段惡意 JavaScript 腳本。當其他使用者預覽這條評論時，腳本便在其瀏覽器上執行。

****原理****: 網站將使用者輸入的惡意腳本原封不動地「反射」回瀏覽器，瀏覽器誤以為是合法程式碼而執行。根本原因在於後端未對輸出進行 HTML 跳脫 (Escaping) 。

攻擊實例

- : 一個看似無害的圖片標籤，卻能透過 onerror 事件執行任意腳本。
- <svg/onload=alert(document.cookie)> : 更進階的攻擊，直接讀取使用者的 Cookie，可用於竊取登入狀態，劫持帳號。

惡意輸入

A screenshot of a web application's review section. The user profile shows 'US' and 'demo'. The review content field contains the following malicious code: . A red arrow points from this code to the 'WAF 擋截' panel.

WAF 擋截



惡意輸入

A screenshot of a web application's review section for 'ShopEasy'. The review content field contains the following malicious code: <svg/onload=alert(document.cookie)>. A red arrow points from this code to the 'WAF 擋截' panel.

WAF 擋截



威脅三：伺服器端請求偽造 (SSRF) — 把你的伺服器變成我的內網探測器

情境與原理

情境：在「發現更便宜的價格」功能中，使用者可以提交一個商品網址，系統會自動去抓取頁面內容。

原理：伺服器信任並直接訪問了使用者提供的 URL，但未限制其訪問範圍，導致攻擊者可以讓伺服器去訪問內部網路資源（例如 localhost 或內網 IP）。

惡意請求

發現更便宜的價格？
如果您在其他網站（如 PChome）看到更低價格，請貼上網址，系統會自動抓取頁面內容。

對方價格 (\$) : \$ 5000
商品網址 (URL) : [/api/fetch?url=http://localhost:8080/admin](http://localhost:8080/admin)

提交查詢

系統回應：
錯誤：Blocked by WAF (SSRF)

Debug Info

攻擊實例

- <http://localhost:8080/admin> : 讓伺服器訪問自己的管理後台。
- <file:///C:/Windows/win.ini> : 利用 file:/// 協定，直接讀取伺服器硬碟上的任意檔案。

正常請求

發現更便宜的價格？
如果您在其他網站（如 PChome，看到更低價格，請貼上網址，系統機器人將自動查價並調整。

對方價格 (\$) : \$ 5000
商品網址 (URL) : <https://www.google.com>

提交查詢

系統回應：
狀態：系統已成功訪問該連結。正在分析價格 (\$000.0)...
內容：

```
<!doctype html><html itemscope="" itemtype="http://schema.org/NebPage" lang="zh-Tw"><head></head><body><img alt="Google logo" data-itemtype="Image" data-itemname="Google logo" data-itemurl="https://www.google.com/images/branding/googlelogo/1x/googlelog_standard_color_128dp.png" data-nonce="qC_FrV-QX2lio268o0eu2g" data-type="Image"/>(function(){var _g={KEI:"HFE3af3HMsHNuDKP1aS1Ac",KEXPI:'0,1384225,5,2845,17,16,2,17,36811522,25228661,152394,65160,39628,9139,4599,329,6225,1117,63648,22778,37789,28339,594,36458,28675,1222,1,4371,6292,11378,4881,2747,2,4,1,1164,1249,1677,18712,2385,4035,10588,18588:9,
```

威脅四：目錄遍歷 (Path Traversal) — 翻閱伺服器不該公開的檔案

情境與原理

情境：在「下載使用說明書」功能，系統預期使用者下載指定的說明書檔案。

原理：攻擊者利用 `../` 這個「回到上一層目錄」的符號，跳出原本被限制的資料夾，進而存取系統中的任意檔案。

漏洞成因：後端程式碼在讀取檔案時，未驗證請求的路徑是否在合法目錄內，也未過濾 `../` 等特殊字元。

攻擊實例： `/api/download?file=main.py` (讀取後端原始碼)，`/api/download?file=../../Mawin.ini` (讀取系統設定檔)，`/api/download?file=../../Windows/win.ini` (讀取系統設定檔)

惡意讀取 (源碼洩漏)

```
!<input type="button" value="←"/> <input type="button" value="→"/> <input type="button" value="C"/> ① 127.0.0.1:8080/api/download?file=main.py
```

```
import sqlite3
import os
import requests
from fastapi import FastAPI, Request, Response
from fastapi.responses import HTMLResponse, JSONResponse, RedirectResponse
from fastapi.templating import Jinja2Templates
from fastapi.staticfiles import StaticFiles
from pydantic import BaseModel
from pydantic import BaseHodel
from typing import Optional
from detector import detect_attack

app = FastAPI(title="Vulnerable Amazon-Clone")
```

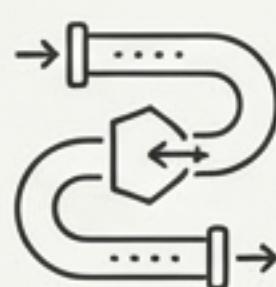
正常讀取

```
✓<input type="button" value="←"/> <input type="button" value="→"/> <input type="button" value="C"/> ① 127.0.0.1:8080/api/download?file=user_manual.txt
```

```
Welcome to ShopEasy! This is a secret manual.
```

防線建立：智慧型 WAF 的偵測邏輯

面對多樣化的攻擊手法，一個有效的防禦系統不能只靠單一策略。接下來，我們將拆解 WAF 的核心偵測引擎，看它如何應對上述威脅。



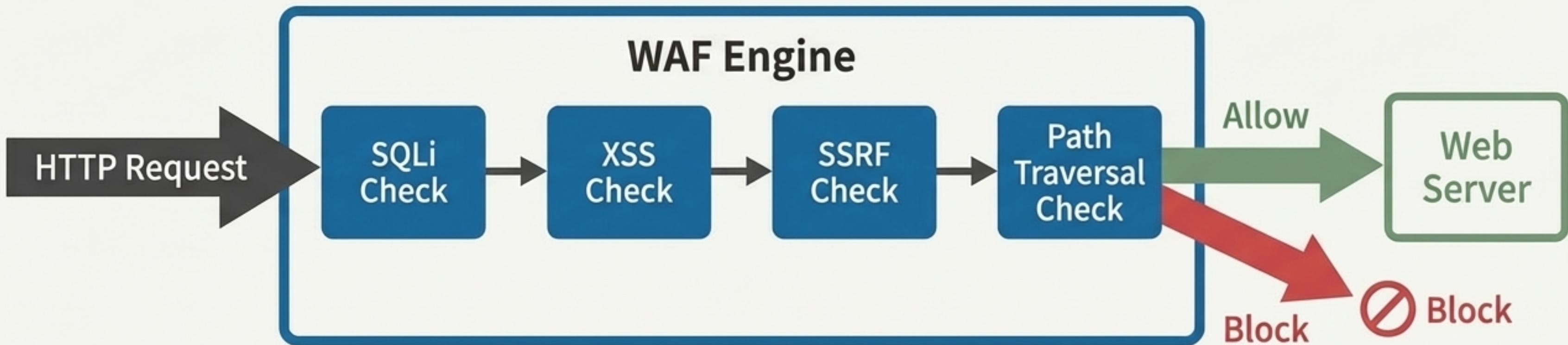
管線式設計 (Pipeline Design)

請求依序通過多個檢測模組，高風險攻擊優先處理，兼顧效能與安全。



多層次策略 (Multi-layered Strategy)

結合關鍵字比對、行為分析與情境判斷，建立深度防禦。



防禦第一步：正規化 (Normalization) — 讓偽裝無所遁形

核心問題

攻擊者經常使用 URL 編碼（例如 `'%2e%2e%2f'` 代表 `'../'`）來隱藏惡意指令，試圖繞過簡單的字串比對。

解決方案：`'_collect_fields'` 函式

1. **收集 (Collect)**：將請求中所有可能包含 payload 的欄位 (URL, Params, Body, User-Agent) 集中起來。
2. **解碼 (Decode)**：對所有收集到的值進行 URL 解碼 (unquote)，還原其真實內容。
3. **攤平 (Flatten)**：將所有欄位整理成一個扁平的字典結構，方便後續模組進行統一檢查。

```
def _collect_fields(input_data: dict) -> Dict[str, str]:  
    pieces: Dict[str, str] = {}  
  
    # URL、方法、User-Agent  
    raw_url = _to_str(input_data.get("url", ""))  
    decoded_url = unquote(raw_url) ← URL 解碼  
    pieces["url"] = decoded_url  
  
    pieces["http_method"] = _to_str(input_data.get("http_method", ""))  
    pieces["user_agent"] = _to_str(input_data.get("user_agent", ""))  
  
    # params 可能是 GET query string 的參數  
    params = input_data.get("params", {}) or {}  
    for k, v in params.items():  
        raw = _to_str(v)  
        decoded = unquote(raw) ← # 參數也先解碼一次 - 參數也先解碼一次  
        pieces[f"param.{k}"] = decoded  
  
    # body 是 POST/PUT 的內容  
    body = input_data.get("body", {}) or {}  
    for k, v in body.items():  
        raw = _to_str(v)  
        decoded = unquote(raw) # body 內容也先解碼  
        pieces[f"body.{k}"] = decoded  
  
    return pieces
```

防禦策略 I：關鍵字模式匹配 — 攻擊特徵的「黑名單」

原理

`_find_pattern` 函式會遍歷所有經過正規化的欄位，檢查是否包含預先定義好的攻擊關鍵字（Patterns）。

應用場景

- **SQL Injection**：偵測 ` or 1=1`、`union select`、`--` 等 SQL 語法關鍵字。
- **XSS**：偵測 `<script>`、`onerror=`、`onload=`、`javascript:` 等腳本相關字串。
- **Path Traversal**：偵測 `../`、`/etc/passwd` 等路徑遍歷特徵。

攻擊特徵定義

```
DEFAULT_RULES = {
    "SQLI_PATTERNS": [
        # 僅存 or 1=1 類型
        " or 1=1",
        " or '1'='1",
        "' or '1'='1",
        "\\" or \"1\"=\\"1\",
        " or '1'='1",
        " or 1=1--",
        " or 1=1#",
        " or 1=1/o",
        " union select",
        "--",
        ";--",
        "/n",
        "%",
        "# toe-based / function 類型 SQL
        "sleep(",

        "pg_sleep('",
        "waitfor delay",
    ],
    "XSS_PATTERNS": [
        "<script>",           # <script>...</script>
        "onerror=",            # tag srcss onerror...
        "onload=",             # onload 事件
        "javascript:",         # href="javascript:alert(1)"
        "alert()",              # alert(1)
        # 更多需覽事件 handler
        "onclick=",
        "onmouseover=",
    ],
}
```

全面搜索邏輯

```
# 在所有欄位搜關鍵字
def _find_pattern(pieces: Dict[str, str], patterns: List[str]) -> Tuple[bool, str, str]:
    """在所有欄位裡面找有沒有出現任一 pattern。
    回傳：
        (是否命中，該欄位名稱，該欄位原始內容)
    游標判斷回傳 (False, "", "")。
    """
    for field_name, raw_value in pieces.items():
        value_lower = raw_value.lower()
        for p in patterns:
            if p.lower() in value_lower:
                return True, field_name, raw_value
    return False, "", ""
```

防禦策略 II：情境感知分析 — 拆解 SSRF 攻擊

挑戰

SSRF 的 payload 本身就是一個合法的 URL，單純的關鍵字匹配無法辨識其惡意性。

偵測流程：`_check_ssrf` 函式

1. **候選者提取**：從請求的各個部分找出所有看起來像 URL 的字符串。
2. **URL 解析**：對每個候選 URL 進行解析，提取其主機名稱 (Hostname)。
3. **風險判斷**：呼叫 `_is_private_or_metadata_ip` 函式，判斷主機是否為內網 IP (如 `127.0.0.1`、`10.x.x.x`) 或雲端服務的 metadata IP。若是，則判定為 SSRF 攻擊。

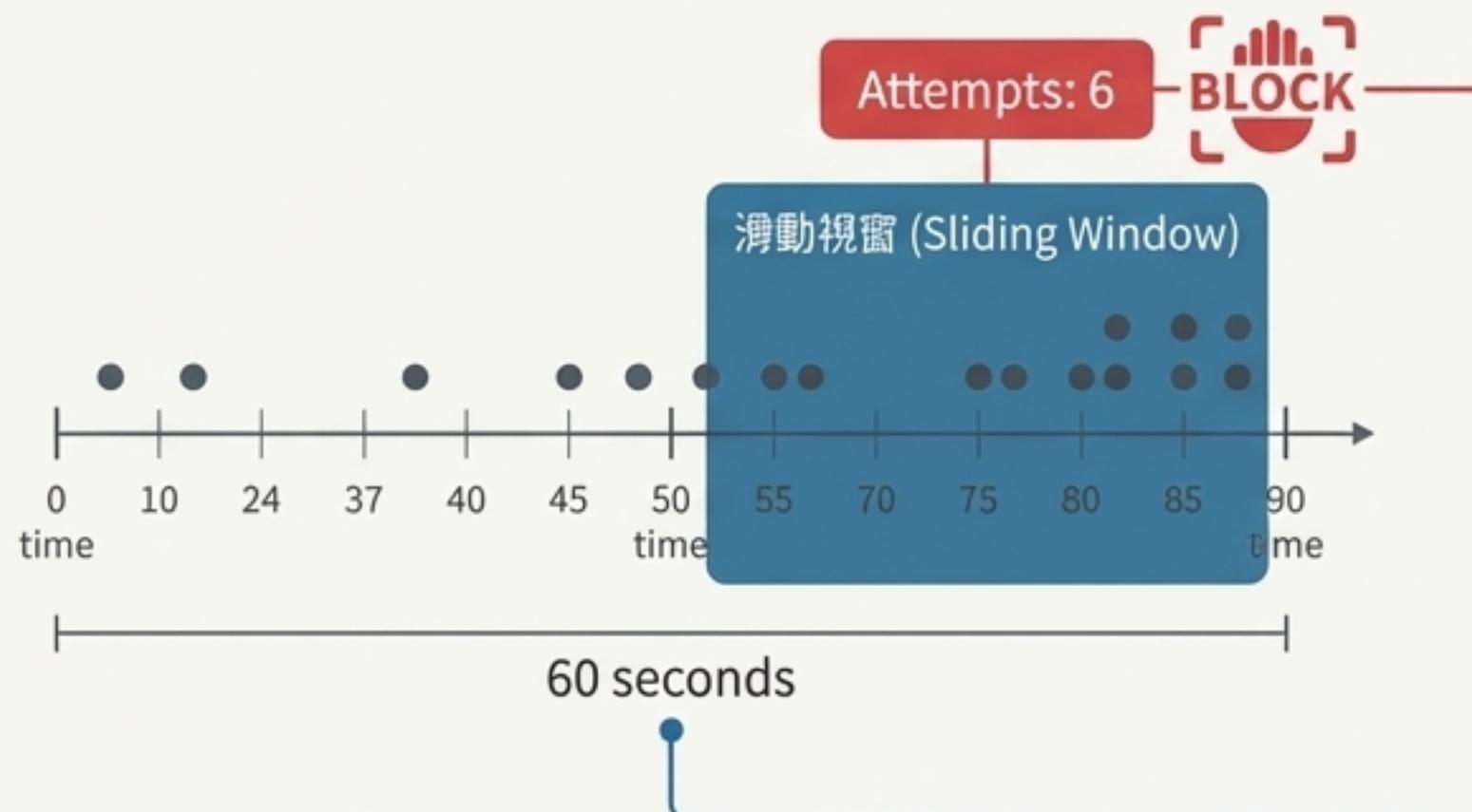
```
def _check_ssrf(input_data: dict) -> Tuple[bool, str]:  
    candidates: List[str] = []  
  
    # 先把原始 URL 也算進可能的 payload  
    raw_url = unquote(_to_str(input_data.get("url", "")))  
    candidates.append(raw_url)  
  
    # params / body 標的內容  
    params = input_data.get("params", {}) or {}  
    body = input_data.get("body", {}) or {}  
  
    for d in (params, body):  
        for _, v in d.items():  
            s = unquote(_to_str(v))  
            candidates.append(s)  
  
    1 → for value in candidates:  
        v = value.strip()  
        if not v:  
            continue  
  
        # 選出看起來像 http/https 的 URL  
        lower = v.lower()  
        url_str = None  
        if lower.startswith("http://") or lower.startswith("https://"):  
            url_str = v  
        else:  
            idx = lower.find("http://")  
            if idx == -1:  
                idx = lower.find("https://")  
            if idx != -1:  
                url_str = v[idx:]  
        if not url_str:  
            continue  
        try:  
            parsed = urlparse(url_str) ← 2  
        except Exception:  
            continue  
        host = parsed.hostname  
        if not host:  
            continue  
        if _is_private_or_metadata_ip(host): ← 3  
            # 記錄 SSRF 風險  
            return True, url_str  
  
    return False, ""
```

防禦策略 III：行為分析 – 識別暴力破解 (Brute Force)

思維轉變：

有些攻擊看的不是「內容」，而是「頻率」。
暴力破解就是短時間內發動大量登入嘗試。

偵測演算法：滑動視窗 (Sliding Window)



```
def _check_bruteforce(input_data: dict) -> Tuple[bool, str]:  
    ip = _to_str(input_data.get("ip_address", ""))  
    url = _to_str(input_data.get("url", "")).lower()  
    method = _to_str(input_data.get("http_method", "")).upper()  
  
    # 不是 login 相關的就不算登入嘗試  
    if "login" not in url:  
        return False, ""  
  
    # 只統計 POST /login (可以視情況調整)  
    if method != "POST":  
        return False, ""  
  
    now = time.time()  
    attempts = _LOGIN_ATTEMPTS.get(ip, [])  
    attempts.append(now)  
  
    # 移除超過時間窗的舊紀錄  
    cutoff = now - BRUTE_FORCE_WINDOW_SECONDS  
    attempts = [t for t in attempts if t >= cutoff]  
    _LOGIN_ATTEMPTS[ip] = attempts  
  
    if len(attempts) >= BRUTE_FORCE_THRESHOLD:  
        info = f"{ip} tried login {len(attempts)} times in {BRUTE_FORCE_WINDOW_SECONDS}  
        return True, info  
  
    return False, ""
```

防禦策略 IV：威脅情報 – 揪出可疑的使用者代理 (Suspicious UA)

原理

正常的網站訪客使用瀏覽器（如 Chrome、Safari），而駭客在進行自動化掃描時，常使用特定的工具，這些工具會在請求的 User-Agent 標頭留下特徵。

WAF 動作

_check_suspicious_ua 函式會比對 User-Agent 字串與可疑工具列表。命中時，這通常是一個低至中風險的輕級告警，提醒管理者「有自動化工具正在掃描你的網站」。

 正常 UA (✓)	 可疑 UA (✗)
 Chrome	sqlmap
 Safari	nmap
 Edge	curl
 Firefox	python-requests
	BurpSuite

```
def _check_suspicious_ua(input_data: dict) -> Tuple[bool, str]:  
    ua = _to_str(input_data.get("user_agent", "")).lower()  
    if not ua:  
        return False, ""  
  
    → for pattern in RULES.get("SUSPICIOUS_UA_PATTERNS", []):  
        p = pattern.lower()  
        if p and p in ua:  
            return True, ua  
    return False, ""
```

總指揮：`detect_attack` 主偵測流程

• 設計模式：Pipeline（管線）設計

- 請求資料進入後，會依序通過各個偵測模組。
- **效能優化**：將危害程度最高、特徵最明顯的 SQL Injection 檢查放在最前面。一旦命中，立即回傳結果並阻擋，無需執行後續檢查，提升反應速度。

偵測順序

```
def detect_attack(input_data: dict) -> dict:  
    # 預設結果 (沒有攻擊)  
    result = {  
        "is_attack": False,  
        "attack_type": "NONE",  
        "severity": "LOW",  
        "payload": "",  
        "should_block": False,  # 先預設 False，最後再由 _apply_block_flag 決定  
  
        "ip_address": _to_str(input_data.get("ip_address", "")),  
        "timestamp": _now_tw(),  
    }  
  
    # 把所有欄位收集起來 (url / params / body / user_agent)  
    pieces = _collect_fields(input_data) # (1)  
  
    # 檢查 SQL Injection  
    hit, field, value = _find_pattern(pieces, RULES["SQLI_PATTERNS"]) # (2)  
    if hit:  
        result["is_attack"] = True  
        result["attack_type"] = "SQLI"  
        result["severity"] = "HIGH"  
        result["payload"] = f"{field}: {value}"  
        return _apply_block_flag(result)  
  
    # ... (後續檢測邏輯)
```

<IMAGE_0>

• 動態規則加載：

- WAF 啟動時會嘗試從 `rules.json` 檔案載入最新的攻擊特徵與設定。
- **容錯機制**：如果 `rules.json` 不存在或格式錯誤，系統會自動退回使用內建的 DEFAULT_RULES，確保 WAF 核心功能永不中斷。

動態與容錯

```
def _load_rules_from_file(filename: str = "rules.json") -> None:  
    global RULES, MODE, BRUTE_FORCE_WINDOW_SECONDS, BRUTE_FORCE_THRESHOLD  
    # ... (範例代碼)  
  
    if not os.path.exists(filename):  
        # 找不到檔案就維持預設規則與模式  
        return  
  
    try:  
        with open(filename, "r", encoding="utf-8") as f: # 打開檔案，讓它自動關掉。  
            data = json.load(f) 將 JSON 內容轉成 Python dict，存在 data 裡。  
            # ... (規則載入與 MODE 調整邏輯)  
  
    except Exception:  
        RULES = DEFAULT_RULES.copy()  
        NODE = "LOG_ONLY" # 有問題就直接忽略，維持預設  
  
    # 啟動時优先試著載入一次  
    _load_rules_from_file()  
    <IMAGE_1>
```

<IMAGE_1>

結論：從被動攔截到主動防禦的思維

核心洞察

1. *攻擊無孔不入*：任何使用者輸入點都可能是潛在的漏洞。
2. *防禦需要深度*：一個強固的 WAF 結合了多種偵測策略。
3. *彈性是關鍵*：透過可配置的規則(`rules.json`)，防禦系統必須能夠快速適應新的威脅。

最終訊息

安全不是一個終點，而是一個動態的過程。真正的韌性來自於深刻理解攻擊者的思維，並以此建構出能夠持續學習與適應的智慧防禦體系。

