UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione



USO DI MODELLI LLM LOCALI PER MIGLIORARE TECNICHE DI DYNAMIC APPLICATION SECURITY TESTING (DAST)

USING LOCAL LLMS TO ENHANCE DYNAMIC APPLICATION SECURITY TESTING (DAST) TECHNIQUES

Relatore
Prof. Adriano Mancini

Laureando
Mattia Sbattella

Correlatore
Dott. Fabio Filipponi

ANNO ACCADEMICO 2025/2026

Sigma S.p.A.

Un ringraziamento particolare va a Sigma S.p.A., che ha reso possibile la collaborazione nell'ambito del tirocinio e la successiva realizzazione di questa tesi.



Abstract



Figura 1: ZAPilot logo.

Gli strumenti di *Dynamic Application Security Testing* (DAST), quali OWASP ZAP e Burp Suite, rappresentano uno standard consolidato per l'analisi di sicurezza delle applicazioni web. Nelle attività manuali, funzionalità come l'intercettazione del traffico, l'intruder e la manipolazione delle richieste consentono ad un analista esperto di individuare vulnerabilità complesse. Tuttavia, tale efficacia dipende in modo significativo dalle competenze dell'operatore e dal tempo disponibile. Le modalità automatizzate, pur permettendo di estendere e velocizzare l'analisi, incontrano invece limiti rilevanti nella comprensione del contesto applicativo, nella navigazione di flussi non lineari e nella compilazione di form che richiedono dati semanticamente coerenti.

Questa criticità emerge anche durante le sperimentazioni preliminari condotte su OWASP Juice Shop, un'applicazione intenzionalmente vulnerabile progettata per attività di formazione e valutazione della sicurezza OWASP Foundation [2026b]. In tale contesto, le scansioni automatiche eseguite con ZAP e Burp Suite non hanno prodotto vulnerabilità confermate: i risultati ottenuti consistevano prevalentemente in segnalazioni a bassa confidenza e falsi positivi. Inoltre, parti di applicazioni nascoste dietro form complesse non venivano nemmeno indicizzate.

La presente tesi propone ZAPilot (logo in Figura 1), un sistema progettato per affiancare OWASP ZAP mediante l'impiego di *Large Language Models* (LLMs). La scelta non è quella di sostituire gli strumenti DAST esistenti: ZAP e Burp Suite mantengono un ruolo

primario poiché le loro scansioni producono evidenze tecniche ripetibili, ispezionabili dall'analista e riconducibili a regole di rilevamento consolidate. Tali caratteristiche sono rilevanti anche nei contesti di audit e conformità, nei quali una risposta generata da un modello linguistico non può costituire, da sola, prova sufficiente dell'esistenza di una vulnerabilità.

L'architettura distingue una modalità *discovery*, dedicata alla navigazione e alla compilazione contestuale dei form, da una modalità *attack*, nella quale il modello propone input di test in relazione ai campi individuati.

L'impiego di modelli locali consente di mantenere all'interno dell'infrastruttura aziendale le richieste, i contenuti delle pagine e i dati inseriti nei form. Pur rinunciando in parte alle prestazioni dei modelli commerciali più avanzati, questa scelta riduce i costi di inferenza e rende l'approccio più adatto a scenari in cui riservatezza e controllo dei dati sono requisiti prioritari.

Il contributo della tesi consiste nel valutare se un LLM locale possa rendere più efficace l'esplorazione guidata da un DAST, senza alterarne il ruolo di strumento di verifica. In particolare, ZAPilot viene valutato su OWASP Juice Shop per misurare quante funzionalità aggiuntive riesce a raggiungere rispetto alla scansione automatica tradizionale e se tali percorsi permettono di esporre vulnerabilità precedentemente non rilevate.

Indice

1 Introduzione **1**
- 1.1 Motivazioni . . . 1
  - 1.1.1 Un compromesso progettuale . . . 3
  - 1.1.2 Posizionamento di ZAPilot nel ciclo DevSecOps . . . 5
- 1.2 Stato dell'arte . . . 8
  - 1.2.1 LLM a supporto del penetration testing . . . 8
  - 1.2.2 AutoPentest-AI e influenze su ZAPilot . . . 8
  - 1.2.3 AWS Summit Milano 2026 e AWS Security Agent . . . 9
- 1.3 Obiettivi della tesi . . . 12

2 Metodologie & Strumenti **13**
- 2.1 Metodologie . . . 13
  - 2.1.1 SAST, DAST e IAST . . . 13
  - 2.1.2 OWASP Top Ten . . . 15
- 2.2 Strumenti . . . 17
  - 2.2.1 OWASP ZAP . . . 17
  - 2.2.2 OWASP Juice Shop . . . 17
  - 2.2.3 Podman . . . 18
  - 2.2.4 Redis . . . 19
  - 2.2.5 Playwright . . . 19
  - 2.2.6 Obsidian . . . 20
- 2.3 Modelli valutati . . . 21
  - 2.3.1 OpenAI GPT-OSS-120B . . . 21
  - 2.3.2 Google Gemma 4 31B IT . . . 21
  - 2.3.3 Qwen3-Coder-30B-A3B-Instruct . . . 22
  - 2.3.4 Modelli considerati ma non inclusi nella valutazione . . . 22

3 Progettazione **23**
3.1 Architettura generale e flusso operativo . . . 23
3.2 Autenticazione e bootstrap della sessione . . . 25
3.3 Queue Builder . . . 26
3.3.1 Initial Phase . . . 27
3.3.2 Re-discover . . . 28
3.3.3 Final Phase . . . 29
3.3.4 Discover . . . 29
3.3.5 Attacco . . . 31
3.4 Deterministic Extraction Layer . . . 32
3.4.1 Pipeline di elaborazione . . . 32
3.4.2 Modello canonico della superficie: normalizzazione . . . 32
3.4.3 Caratterizzazione della superficie . . . 33
3.4.4 Identità strutturale e identità logica . . . 33
3.4.5 Differenze nel dettaglio . . . 34
3.4.6 Riuso dei risultati e strategia di retest . . . 34
3.5 Browser Playwright . . . 35
3.5.1 Gestione del contesto browser: Browser Runtime . . . 36
3.5.2 Navigazione e osservazione . . . 36
3.5.3 Esecuzione delle azioni: Action Executor . . . 37
3.5.4 Estrazione del contesto: Browser Extractor . . . 37
3.6 LLM . . . 38
3.6.1 Base Client . . . 38
3.6.2 Analyst . . . 39
3.6.3 Planner . . . 40
3.6.4 Evaluator, LLM as a judge . . . 40
3.6.5 Wiki . . . 41
3.6.6 Navigator . . . 42
3.7 LLM Wiki come knowledge base persistente . . . 42
3.7.1 Tre paradigmi per la conoscenza . . . 43
3.7.2 Confronto tra gli approcci . . . 46
3.8 Memoria . . . 47
3.8.1 Memoria temporanea . . . 47
3.8.2 Memoria persistente . . . 48
3.8.3 Redis . . . 49
3.9 Orchestratore . . . 50
3.9.1 Modalità Attack . . . 50
3.9.2 Modalità Discovery . . . 53

4 Implementazione **56**
4.1 Bootstrap . . . 57
4.1.1 Caricamento e validazione del file di configurazione . . . 57
4.1.2 Caricamento e validazione del profilo di Auth . . . 58

4.1.3 Esecuzione della fase Auth del bootstrap . . . 58
4.2 Queue Builder . . . 61
4.2.1 Normalizzazione, scope e priorità degli URL . . . 62
4.2.2 Configurazione del Context ZAP . . . 62
4.2.3 Crawler Playwright . . . 63
4.3 Deterministic Extraction Layer . . . 63
4.3.1 Pipeline di assemblaggio . . . 64
4.3.2 Implementazione del modello canonico . . . 64
4.3.3 Classificazione ed estrazione delle feature . . . 65
4.3.4 Calcolo degli identificatori . . . 66
4.3.5 Persistenza, cache e retest . . . 67
4.4 Browser Playwright . . . 67
4.4.1 Action Executor . . . 67
4.5 LLM . . . 70
4.6 Wiki . . . 71
4.6.1 Creazione della LLM Wiki: ingest e merge . . . 72
4.6.2 Knowledge di base persistente . . . 72
4.6.3 Retrieval ibrido della LLM Wiki: query . . . 73
4.6.4 Integrità del knowledge graph: lint . . . 74
4.7 Memory . . . 75
4.7.1 Memoria di sessione . . . 76
4.7.2 Try exact replay . . . 77
4.7.3 Grafico persistente . . . 81
4.8 Orchestrator . . . 82
4.8.1 MaxCandidates e Budget . . . 83
4.8.2 Assenza di superficie d'attacco . . . 83
4.8.3 Influenza della cache e del replay . . . 83

5 Risultati **86**
5.1 Fase di discovery . . . 86
5.1.1 Senza ZAPilot . . . 86
5.1.2 Con ZAPilot . . . 87
5.2 Fase di attack . . . 88
5.2.1 Senza ZAPilot . . . 88
5.2.2 Con ZAPilot . . . 88
5.2.3 Riassunto della fase di attack . . . 95
5.3 Confronto sperimentale tra i modelli linguistici . . . 95
5.3.1 Risultati per singola scansione . . . 97
5.3.2 Confronto aggregato per modello . . . 97
5.3.3 Stima del costo con un modello commerciale . . . 98
5.3.4 Incidenza delle chiamate LLM . . . 99
5.3.5 Analisi per ruolo della pipeline . . . 99
5.3.6 Affidabilità, retry e caching . . . 100

5.3.7 Efficacia nell'individuazione delle vulnerabilità 101
5.3.8 Discussione dei risultati 103
5.4 Altre macchine testate 103
5.4.1 DVWA 104
5.4.2 Applicazioni Sigma SpA 107
5.5 L'importanza del Reasoning 107
5.6 Un caso particolare: prompt injection 108
5.7 Possibile bias conoscitivo dei modelli 110

6 Conclusioni **114**
6.1 Conclusioni 114
6.2 Sviluppi futuri 116

Appendice **119**

Bibliografia **124**

Elenco delle figure

1 ZAPilot logo. 2

1.1 Report generato da Burp Suite Professional durante l'analisi di OWASP Juice Shop. 2
1.2 Confronto qualitativo tra DAST, ZAPilot, soluzioni agentiche e team di penetration testing rispetto a tempi, costi, confidenzialità, capacità operative, potenza computazionale e interpretabilità. 5
1.3 Architettura AWS security agent. 10

2.1 Confronto concettuale tra SAST, DAST e IAST. 14
2.2 Logo dello strumento. Fonte: ZAP Development Team [2026]. 17
2.3 Logo dello strumento. Fonte: OWASP Foundation [2026c]. 17
2.4 Logo dello strumento. Fonte: Podman [2026]. 18
2.5 Logo dello strumento. Fonte: Redis [2026]. 19
2.6 Logo dello strumento. Fonte: Microsoft [2026]. 19
2.7 Logo dello strumento. Fonte: Obsidian [2026]. 20

3.1 Architettura generale di ZAPilot e principali fasi del flusso operativo. . . 23
3.2 Flusso generale delle attività di discovery e attack coordinate dall'Orchestrator. 24
3.3 Fase di bootstrap. 25
3.4 Struttura del Queue Builder e ciclo di elaborazione con l'Orchestrator. . 27
3.5 Initial Phase: discover e filtro costruiscono la coda iniziale, mentre l'attacco produce le segnalazioni di sicurezza. 28
3.6 Re-discover: aggiornamento incrementale della coda dopo l'elaborazione di un nodo. 29
3.7 Final Phase: esecuzione conclusiva dell'attacco dopo l'esaurimento della coda. 29

3.8 Pipeline di discovery unificata: ZAP Classic Spider, ZAP AJAX Spider e Playwright DOM crawl. 30
3.9 Pipeline di attacco: Active Scan di ZAP, Gitleaks e Sensitive Path Check. 31
3.10 Pipeline del *Deterministic Extraction Layer*. 32
3.11 Struttura del Browser Layer basato su Playwright. 36
3.12 Architettura dei componenti basati su LLM e suddivisione delle relative responsabilità e ruoli. 38
3.13 Flusso di interazione tra i vari LLM in fase di attacco. 39
3.14 Sistema RAG. 43
3.15 Sistema agentico. 45
3.16 Memorie utilizzate. 48
3.17 Flusso coordinato dall'Orchestrator in modalità Attack. 51
3.18 Flusso di gestione del cache hit. 52
3.19 Flusso di exact replay di un attacco confermato. 52
3.20 Flusso coordinato dall'Orchestrator in modalità Discovery. 54

4.1 Architettura generale di containerizzazione. 56
4.2 Esempio di risoluzione dei locator e compilazione controllata dei campi da parte di `ActionExecutor` , in particolare il campo `field_email` e `field_password` 69
4.3 Errore iniziale sulla paginazione e controllo introdotto per circoscrivere il submit della barra di ricerca. 70
4.4 Obsidian graph. 75
4.5 Conservazione e riutilizzo dei valori confermati dalla memoria di sessione durante la discovery. 77
4.6 Grafo generato in fase di attacco. 82

5.1 Input senza contesto di ZAP. 87
5.2 Sequenza delle interazioni eseguite da ZAPilot durante la fase di discovery. 88
5.3 Confronto della superficie dell'applicazione raggiunta prima (a) e dopo (b) l'esplorazione guidata. 89
5.4 Attacco eseguito con la sola scansione automatica di ZAP, senza l'utilizzo di ZAPilot. Trova solo una possibile SQL injection con confidenza low. . 90
5.5 Evidenze relative alla SQL injection utilizzata per aggirare l'autenticazione. 91
5.6 Segnalazione di un presunto bypass del CAPTCHA, risultata un falso positivo in seguito alla verifica. 92
5.7 Segnalazione del redirect verso un URL esterno controllato tramite il parametro `to` 93
5.8 Confronto delle vulnerabilità dell'applicazione trovate prima (a) e dopo (b) la fase di attacco con ZAPilot. 96
5.9 Confronto degli alert prodotti dai tre modelli nelle scansioni considerate. Il conteggio complessivo include anche segnalazioni generate da ZAP e deve pertanto essere interpretato insieme alle evidenze relative alle vulnerabilità assistite dal modello. 102

5.10 Potenziali vulnerabilità individuate durante la valutazione di ZAPilot su DVWA. 105
5.11 Sintesi delle vulnerabilità individuate da Burp Suite Professional durante la scansione di DVWA. 106
5.12 Esempio di reasoning utilizzato per risolvere la challenge basata su regole semantiche. 107
5.13 Istruzione malevola inserita nel contenuto della pagina durante la fase di *discovery*. 108
5.14 Prova dell'indirect prompt injection avvenuta correttamente. 109
5.15 Tentativo di accesso di Gemma-4-31B-IT con l'indirizzo corretto e una password errata. 111
5.16 Accesso ottenuto da Gemma-4-31B-IT mediante credenziali deboli durante uno dei tentativi effettuati. 111
5.17 Indicazione della password `admin123` nel file `authentication.md` della sintesi wiki passata al planner. 112

Elenco delle tabelle

1.1 Confronto tra l'impiego di OWASP ZAP senza e con l'integrazione di ZAPilot nel ciclo DevSecOps. 7

2.1 Categorie dell'OWASP Top Ten 2025. 16

3.1 Confronto tra State Hash e Surface Signature 35
3.2 Confronto tra RAG, agenti con memoria e LLM Wiki. 46

4.1 Comportamento dell'Orchestrator in funzione di cache, persistenza e replay cross-version. 84

5.1 Confronto complessivo tra scansione tradizionale e scansione assistita. . 95
5.2 Risultati aggregati per singola scansione. 97
5.3 Confronto aggregato tra modelli. 97
5.4 Costo ipotetico delle scansioni applicando il listino standard di Claude Fable 5. 98
5.5 Incidenza temporale delle chiamate LLM sul tempo totale di scansione. . 99
5.6 Numero di chiamate LLM per ruolo e per scansione. 100

CAPITOLO 1

Introduzione

Questo capitolo introduce le motivazioni alla base di ZAPilot e inquadra il progetto nel contesto del ciclo DevSecOps. A partire dai limiti delle verifiche di sicurezza eseguite soltanto al termine dello sviluppo, viene analizzato il ruolo degli strumenti DAST e la difficoltà di automatizzare l'esplorazione di applicazioni caratterizzate da form complessi, autenticazione e workflow dipendenti dal contesto. Il capitolo presenta il posizionamento di ZAPilot, ne confronta l'approccio con le più recenti soluzioni di penetration testing agentico e definisce gli obiettivi perseguiti nel lavoro di tesi.

1.1 Motivazioni

Nel ciclo di sviluppo del software, le attività di verifica della sicurezza vengono frequentemente considerate come una fase conclusiva, da svolgere soltanto in prossimità del rilascio di una nuova versione dell'applicazione. Tale impostazione risulta particolarmente problematica: quando le tempistiche di progetto si riducono o emergono ritardi nelle fasi di sviluppo, i test di sicurezza sono spesso ridimensionati, posticipati o, nei casi più critici, completamente omessi. Il tempo apparentemente risparmiato durante lo sviluppo può tuttavia tradursi in costi significativamente superiori in caso di compromissione dell'applicazione in ambiente produttivo. Una vulnerabilità non individuata prima del rilascio può infatti comportare interruzioni del servizio, perdita o esposizione di dati, danni reputazionali e costi di remediation più elevati rispetto a quelli richiesti da un controllo preventivo.

Un ulteriore elemento di criticità riguarda le modalità con cui tali verifiche vengono eseguite. I test funzionali e di sicurezza sono spesso svolti da sviluppatori, da personale con competenze tecniche elevate oppure direttamente da membri del team che hanno realizzato l'applicazione. Sebbene tali figure possiedano una conoscenza approfondita dell'architettura e dei requisiti del sistema, questa conoscenza può introdurre un bias significativo: chi ha progettato o implementato una funzionalità tende a conoscere già i

percorsi corretti di utilizzo, i valori da inserire nei campi e le assunzioni implicite alla base del workflow applicativo. Di conseguenza, alcune condizioni anomale, percorsi inattesi o comportamenti tipici di un utente esterno possono rimanere inesplorati.

Per affrontare questi problemi si ricorre a strumenti automatici per l'analisi delle vulnerabilità, in particolare agli strumenti DAST. Questi permettono di eseguire controlli ripetibili, ridurre l'intervento manuale e integrare le verifiche di sicurezza nel processo di sviluppo e rilascio. L'automazione comporta tuttavia un inevitabile compromesso tra scalabilità e capacità di adattarsi al contesto. Uno scanner automatico deve essere abbastanza generale da operare su applicazioni differenti, ma proprio questa generalità ne limita la capacità di interpretare correttamente ogni specifica superficie di attacco.

La figura 1.1 mostra, a titolo di esempio, il report prodotto da Burp Suite Professional durante l'analisi di OWASP Juice Shop. Sebbene si tratti di uno strumento commerciale avanzato e ampiamente utilizzato, una scansione di questo tipo non riesce ad individuare automaticamente le vulnerabilità presenti nell'applicazione target, progettata appositamente come applicazione vulnerabile. La potenza dello strumento non elimina quindi i limiti intrinseci dell'esplorazione automatica né la necessità di integrare approcci differenti.



Figura 1.1: Report generato da Burp Suite Professional durante l'analisi di OWASP Juice Shop.

Risulta irrealistico pensare ad uno strumento automatico perfettamente efficace in ogni contesto applicativo. È in questo scenario che si colloca ZAPilot, progettato come componente intelligente di orchestrazione e di supporto all'esplorazione e agli attacchi basati sul contesto. Il sistema sfrutta le capacità di ragionamento contestuale dei modelli

linguistici per affiancare gli strumenti DAST tradizionali, adattare i tentativi di attacco al tipo di form presente nella pagina, superare i blocchi causati da elementi interattivi difficili da interpretare e contribuire così all'ottimizzazione del ciclo DevSecOps.

1.1.1 Un compromesso progettuale

La scelta di affiancare un modello linguistico ad uno strumento DAST, anziché realizzare un agente autonomo incaricato dell'intero processo di penetration testing, nasce dalla valutazione dei compromessi tra tempi, costi, confidenzialità, capacità operative, potenza computazionale ed interpretabilità. ZAPilot non sostituisce gli strumenti di analisi dinamica, ma ne estende le possibilità operative nei passaggi che richiedono una comprensione del contesto. Il modello interpreta la pagina e lo stato dell'applicazione, interagisce con l'interfaccia e può formulare azioni e input di attacco coerenti con il contesto osservato. Il DAST continua invece a svolgere le attività sistematiche per le quali è stato progettato, quali il crawling, l'invio delle richieste e la verifica delle vulnerabilità sugli endpoint raggiunti.

Dal punto di vista dei tempi, questo approccio conserva i vantaggi propri dell'automazione. Un'esecuzione DAST può essere avviata con maggiore frequenza e richiede generalmente tempi inferiori rispetto ad un penetration test manuale completo. Il supporto del modello mira inoltre ad evitare che la scansione si arresti davanti a form o workflow non interpretabili dal crawler e ad orientare le verifiche in funzione delle informazioni presenti nella pagina, senza introdurre la complessità di un agente chiamato a pianificare e condurre autonomamente ogni fase del test. L'obiettivo non è raggiungere la profondità di analisi di un professionista, bensì ampliare in modo sostenibile sia la superficie esaminabile sia la capacità di adattare le azioni di test al contesto applicativo.

Anche il contenimento dei costi ha orientato la progettazione. L'impiego di modelli eseguiti localmente evita i costi ricorrenti associati ai servizi basati sul consumo di token e rende più prevedibile il costo delle scansioni ripetute. Questa scelta offre inoltre un vantaggio rilevante in termini di confidenzialità: contenuti dei form, richieste e risposte, credenziali di test, token di sessione ed endpoint interni possono rimanere nell'infrastruttura che ospita il sistema, senza essere trasmessi ad un fornitore esterno.

Il vantaggio economico e la maggiore tutela dei dati comportano tuttavia una minore potenza disponibile. I modelli utilizzabili su un'infrastruttura locale sono generalmente più limitati oppure richiedono risorse hardware considerevoli per garantire prestazioni confrontabili con quelle dei modelli ospitati su piattaforme remote. Questo vincolo diventa ancora più evidente in un'architettura pienamente agentica, nella quale più agenti devono operare in parallelo, mantenere contesti estesi e condividere la memoria dello stato del test. Parallelismo, lunghezza del contesto e gestione della memoria aumentano infatti sia il fabbisogno computazionale sia i tempi di inferenza.

Ne deriva una capacità operativa intenzionalmente più circoscritta rispetto a quella delle piattaforme agentiche complete. ZAPilot non cerca di sostituire il penetration tester, di coordinare numerosi strumenti offensivi o di costruire in autonomia catene di attacco complesse, ma concentra le risorse disponibili su un insieme circoscritto di

compiti, lasciando allo scanner le verifiche automatiche che sa già eseguire in modo ripetibile. La limitazione delle capacità non rappresenta pertanto un effetto accidentale, ma una decisione progettuale che consente di bilanciare rapidità, sostenibilità economica, protezione dei dati e requisiti computazionali.

Questa specializzazione favorisce inoltre la spiegabilità del comportamento del sistema. In un agente generalista, l'elevato numero di strumenti, azioni e strategie disponibili amplia lo spazio delle decisioni possibili e rende più difficile ricostruire le ragioni che hanno portato alla scelta di uno specifico percorso operativo. In ZAPilot, al contrario, il modello interviene entro un perimetro ristretto e con uno scopo chiaramente definito in cui la separazione tra le decisioni del modello e le attività deterministiche svolte dal DAST rende più semplice attribuire le azioni ai rispettivi componenti, esaminare gli input e gli output del modello e comprendere il percorso che ha condotto al raggiungimento di un nuovo endpoint o all'esecuzione di una specifica verifica. Tale vantaggio non elimina la natura probabilistica del modello linguistico, ma limita il numero di decisioni autonome da analizzare e rende il processo complessivo più tracciabile rispetto a quello di un agente dotato di un insieme più ampio di possibilità operative.

La figura 1.2 mette a confronto approcci differenti mediante un radar chart a sei assi, corrispondenti a tempi, costi, confidenzialità, capacità operative, potenza computazionale e interpretabilità. Il grafico propone una lettura qualitativa generale: non esiste una configurazione ottimale in assoluto, poiché ogni soluzione privilegia alcune proprietà a discapito di altre. Un DAST tradizionale favorisce ripetibilità e contenimento delle risorse, una soluzione pienamente agentica amplia l'autonomia e le capacità operative sostenendo però costi e requisiti computazionali maggiori, mentre un team umano offre flessibilità e interpretabilità elevate a fronte di tempi e costi superiori. Le valutazioni riportate non costituiscono misure assolute, ma sintetizzano le tendenze e i compromessi tipici delle diverse categorie considerate.

All'interno di questo quadro, ZAPilot occupa una posizione intermedia. Rispetto al solo DAST introduce comprensione del contesto, interazione guidata e generazione di tentativi di attacco pertinenti alla pagina analizzata; rispetto a un agente generalista conserva invece un perimetro operativo più limitato. Il suo posizionamento deriva quindi dalla ricerca di un equilibrio tra i sei fattori, non dalla massimizzazione di una singola dimensione.

Nel grafico in figura 1.2 non compare volutamente una dimensione relativa alla capacità di individuare vulnerabilità. ZAPilot nasce infatti per colmare un limite circoscritto degli strumenti DAST, ampliandone le possibilità operative. Una maggiore copertura della superficie applicativa e la formulazione di tentativi di attacco contestualmente pertinenti possono aumentare le opportunità di sottoporre endpoint e funzionalità a verifiche significative, ma non implicano né consentono di garantire che vengano individuate tutte le vulnerabilità presenti. Una simile garanzia sarebbe in generale irrealistica: l'esito di un'analisi dipende dal perimetro considerato, dallo stato dell'applicazione, dalle classi di vulnerabilità ricercate, dalle tecniche impiegate e dai limiti degli strumenti utilizzati. Inoltre, l'assenza di segnalazioni non costituisce una prova dell'assenza di vulnerabilità. La capacità di rilevamento viene pertanto valutata sperimentalmente nei

casi di studio successivi e non viene trattata come una proprietà assoluta o come una dimensione direttamente confrontabile nel compromesso progettuale.

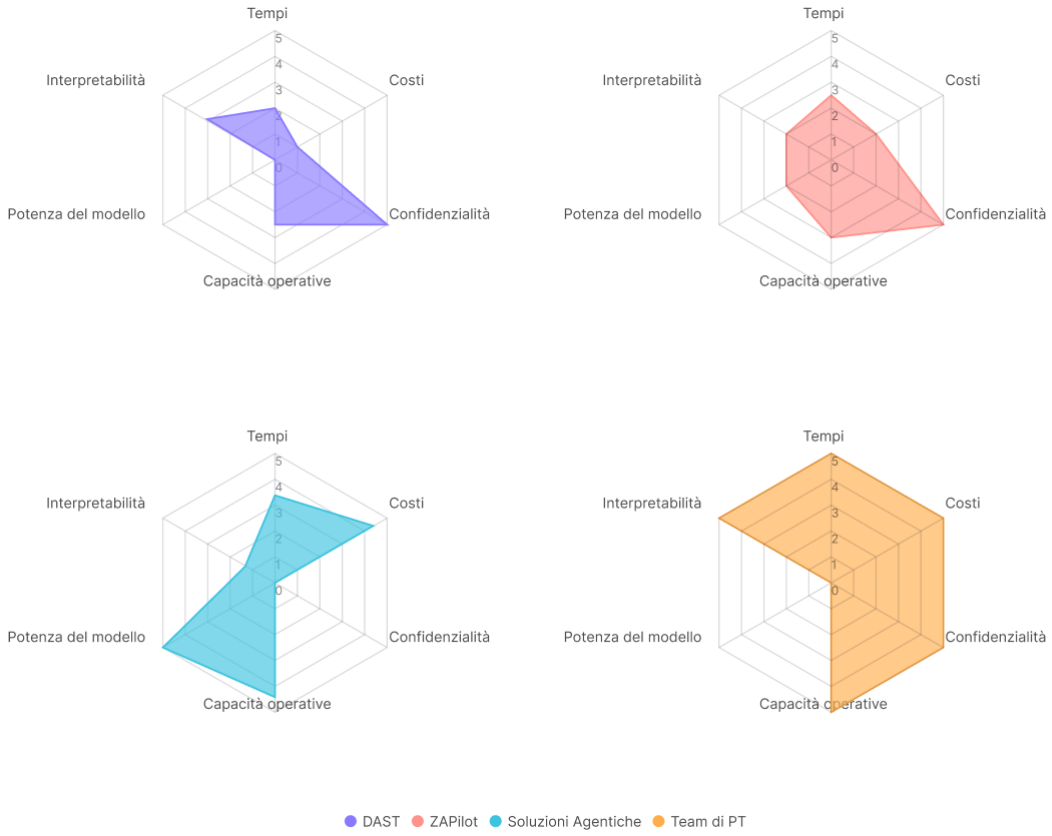


Figura 1.2: Confronto qualitativo tra DAST, ZAPilot, soluzioni agentiche e team di penetration testing rispetto a tempi, costi, confidenzialità, capacità operative, potenza computazionale e interpretabilità.

1.1.2 Posizionamento di ZAPilot nel ciclo DevSecOps

L'integrazione della sicurezza nel ciclo DevSecOps risponde alla crescente velocità con cui le applicazioni software vengono sviluppate, modificate e rilasciate. La diffusione degli strumenti automatici di sviluppo consente alle organizzazioni di distribuire nuove versioni del software con frequenza sempre maggiore. Durante la sessione dell'AWS Summit è stato riportato che circa il 60% delle organizzazioni aggiorna le proprie applicazioni web almeno una volta alla settimana; considerando un orizzonte mensile, tale percentuale raggiunge il 75%. Nella medesima presentazione, un penetration test manuale è stato indicato come un'attività dalla durata tipica compresa tra tre e sei settimane, con un costo variabile tra 5.000 e 50.000 dollari per applicazione (Amazon Web Services [2026]).

Questo divario temporale mostra come il problema principale non sia l'assenza di strumenti di sicurezza, ma la difficoltà di mantenere la frequenza dei controlli allineata a quella dei rilasci. Un penetration test condotto esclusivamente al termine dello sviluppo può trasformarsi in un collo di bottiglia per la distribuzione del software. Analogamente, l'esecuzione di scansioni DAST tradizionali durante lo sviluppo può non raggiungere gli obiettivi desiderati qualora l'applicazione presenti workflow complessi, elementi dinamici o form che richiedano dati semanticamente coerenti.

ZAPilot si inserisce proprio in questo punto del ciclo DevSecOps. In assenza del sistema proposto, OWASP ZAP può essere eseguito periodicamente negli ambienti di sviluppo o test, contribuendo ad automatizzare una parte delle attività normalmente associate al penetration testing. Tuttavia, quando una nuova versione dell'applicazione introduce un form complesso, una procedura di registrazione, un flusso di autenticazione o una sequenza di interazioni che richiede dati consistenti con il contesto, il crawler può non riuscire a proseguire autonomamente. In tali circostanze, la scansione DAST viene necessariamente limitata alla sola porzione dell'applicazione già scoperta o direttamente esposta.

L'integrazione di ZAPilot estende questo paradigma operativo. ZAPilot assume il ruolo di orchestratore intelligente, mentre OWASP ZAP viene utilizzato come proxy e come motore per le attività di crawling, analisi e verifica delle vulnerabilità. Quando durante la navigazione viene rilevato un form complesso o un altro elemento interattivo potenzialmente bloccante, ZAPilot può utilizzare un modello linguistico per interpretare il contesto della pagina, identificare lo scopo dei campi e proporre valori coerenti che consentano di proseguire nel workflow applicativo.

Le pagine, le funzionalità e gli endpoint raggiunti mediante questa interazione diventano quindi parte della superficie applicativa disponibile per le successive scansioni di ZAP. L'obiettivo non è sostituire il motore DAST né delegare integralmente al modello linguistico la ricerca delle vulnerabilità, bensì aumentare la capacità dello scanner di raggiungere funzionalità che altrimenti rimarrebbero inesplorate. Il risultato è un processo nel quale l'automazione DAST continua a eseguire le proprie verifiche, ma su una porzione potenzialmente più estesa e contestualizzata dell'applicazione.

Un ulteriore vantaggio di ZAPilot consiste nella gestione incrementale delle verifiche. In un processo di sviluppo caratterizzato da rilasci frequenti, non è sempre necessario ripetere integralmente l'analisi dell'intera applicazione. Il sistema può essere impiegato sia per rieseguire controlli su versioni precedenti, verificando la persistenza o l'eventuale regressione di vulnerabilità già individuate, sia per concentrare l'esplorazione sulle funzionalità introdotte o modificate in una nuova versione. In questo modo, l'attività di sicurezza può essere adattata ai cambiamenti effettivi del software, evitando di dedicare risorse esclusivamente a porzioni già analizzate quando non siano intervenute modifiche rilevanti.

Il principale punto di forza di ZAPilot è la possibilità di mantenere un controllo DAST ripetibile e più adattabile anche in contesti caratterizzati da cicli di rilascio rapidi. Il sistema riduce la necessità di intervento manuale ogni volta che un form, un workflow o un elemento interattivo impedisce allo scanner di proseguire, contri-

Aspetto	Prima di ZAPilot	Con ZAPilot
Esplorazione	Limitata ai percorsi che il crawler riesce a percorrere autonomamente	Estesa ai percorsi raggiungibili attraverso la compilazione contestuale dei form e l'interazione con elementi dinamici
Form complessi	Possibile arresto della navigazione in presenza di input non interpretabili automaticamente o vincolati da requisiti semantici	Supporto alla compilazione dei campi sulla base del contesto della pagina, del workflow e dei dati richiesti
Superficie analizzata	Dipendente dagli endpoint inizialmente esposti o scoperti dal crawling automatico	Potenzialmente ampliata dalle funzionalità raggiunte dopo l'interazione con form, procedure di registrazione o flussi applicativi articolati
Verifica delle vulnerabilità	Eseguita da ZAP sugli endpoint raggiunti	Eseguita da ZAP anche sulle superfici ulteriormente esplorate, con la possibilità di associare attacchi mirati al contesto individuato grazie a ZAPilot
Ripetizione dei test	Necessità di rieseguire il crawling senza una gestione esplicita delle parti già analizzate	Possibilità di ripetere controlli su versioni precedenti, concentrandosi sulle vulnerabilità già corrette, oppure di continuare l'esplorazione dal punto in cui le scansioni precedenti si erano fermate

Tabella 1.1: Confronto tra l'impiego di OWASP ZAP senza e con l'integrazione di ZAPilot nel ciclo DevSecOps.

buendo ad avvicinare la frequenza delle verifiche di sicurezza a quella dell'evoluzione dell'applicazione.

ZAPilot non elimina la necessità di penetration test manuali, né può garantire l'individuazione di tutte le vulnerabilità presenti. Il penetration testing condotto da personale specializzato rimane essenziale negli scenari difficilmente modellabili attraverso procedure automatiche. La finalità del sistema è piuttosto quella di ridurre la distanza tra la velocità di evoluzione del software e la capacità di eseguire verifiche dinamiche sulle sue funzionalità, individuando un compromesso sostenibile tra tempi di esecuzione, costi operativi e qualità dell'analisi.

Rispetto alle piattaforme commerciali di penetration testing agentico basate su modelli a consumo (come ad esempio AWS Security Agent, descritto nella sezione 1.2.3), una soluzione sviluppata e gestita internamente può contribuire a contenere i costi ricorrenti associati alle chiamate ai modelli linguistici e all'esecuzione dei test. Questo vantaggio deve tuttavia essere valutato alla luce dei costi iniziali di progettazione, implementazione, addestramento o configurazione dei modelli, manutenzione e aggiornamento della piattaforma. In contesti nei quali la sicurezza dell'applicazione rappresenta un processo continuativo e non un'attività occasionale, tale investimento può risultare giustificato dalla maggiore ripetibilità, dal controllo dei dati e dalla capacità di adattare il sistema alle specifiche esigenze organizzative.

Nella tabella 1.1 sono riassunte le principali differenze con e senza ZAPilot.

1.2 Stato dell'arte

1.2.1 LLM a supporto del penetration testing

Tra i primi lavori di riferimento rientra PentestGPT, un framework che impiega un modello linguistico per assistere l'esecuzione di attività di penetration testing. L'architettura separa il ragionamento, la generazione dei comandi e la gestione dello stato, con l'obiettivo di preservare il contesto durante interazioni prolungate. Questo approccio mostra come un LLM possa coordinare strumenti offensivi, traducendo gli obiettivi dell'operatore in azioni tecniche e rivalutando i risultati ottenuti Deng *et al.* [2024].

Una successiva linea di sviluppo riguarda l'adozione del *Model Context Protocol* (MCP), che standardizza l'esposizione di strumenti e fonti informative ai modelli. PentestMCP propone una raccolta modulare di server MCP dedicati, tra le altre attività, allo scanning, all'enumerazione, al fingerprinting e alla valutazione delle vulnerabilità. Questa impostazione favorisce la composizione di workflow nei quali responsabilità differenti possono essere assegnate a componenti specializzati, anziché essere concentrate in un unico modello monolitico (Ezetta e Feng [2025]).

1.2.2 AutoPentest-AI e influenze su ZAPilot

Un riferimento particolarmente rilevante per la progettazione di ZAPilot è `AutoPentest-AI` , sviluppato nel repository `bhavsec/autopentest-ai` . Il proget-

to espone tramite MCP strumenti e conoscenza per il penetration testing di applicazioni web, organizzando il processo secondo una metodologia ispirata all'OWASP Web Security Testing Guide e alle tecniche della PortSwigger Web Security Academy (bhavsec [2026]).

L'aspetto di maggiore interesse per questa tesi è la separazione delle responsabilità. AutoPentest-AI distingue infatti ruoli dedicati alla ricognizione, all'analisi della superficie d'attacco, alla verifica delle ipotesi di vulnerabilità e alla produzione delle evidenze. ZAPilot riprende il principio della scomposizione funzionale, evitando di affidare a un solo LLM l'intero processo: Analyst, Planner ed Evaluator svolgono compiti distinti e ricevono contesti e istruzioni coerenti con la rispettiva responsabilità (bhavsec [2026]).

Questa analogia, tuttavia, non rende ZAPilot una soluzione di penetration testing agentico. Il progetto non assegna a un agente autonomo il compito di scegliere liberamente strumenti, obiettivi e sequenze di attacco. Al contrario, mantiene una pipeline controllata nella quale OWASP ZAP e Playwright continuano a eseguire le operazioni di analisi e navigazione, mentre i modelli linguistici forniscono capacità di interpretazione, pianificazione circoscritta e valutazione dei risultati. L'obiettivo è rendere più intelligenti e adattabili gli strumenti DAST esistenti, non sostituirli con un penetration tester autonomo.

Un secondo elemento ripreso da AutoPentest-AI riguarda la conoscenza di dominio. I materiali raccolti nella knowledge base del progetto sono stati utilizzati tra le fonti iniziali fornite alla LLM Wiki di ZAPilot. Durante l'ingestione, tali contenuti vengono riorganizzati in una base di conoscenza locale e personalizzata, dalla quale il sistema recupera indicazioni pertinenti alla pagina, all'endpoint o alla possibile classe di vulnerabilità in esame. Gli hint recuperati non determinano autonomamente l'azione da eseguire, ma arricchiscono il contesto fornito ai componenti linguistici e contribuiscono alla formulazione di ipotesi e tentativi più mirati (bhavsec [2026]).

ZAPilot si concentra pertanto sull'estensione delle capacità di un motore DAST, mantenendo un grado di autonomia intenzionalmente limitato.

1.2.3 AWS Summit Milano 2026 e AWS Security Agent

Il 28 maggio 2026, durante l'AWS Summit di Milano, è stata presentata una sessione tecnica dedicata ad *AWS Security Agent*, una piattaforma per il penetration testing agentico. Il caso è di particolare interesse per questa tesi poiché mostra, in un contesto industriale, una direzione analoga a quella esplorata dal progetto ZAPilot: utilizzare modelli linguistici e agenti per superare i limiti del solo scanning automatico, mantenendo tuttavia separati l'esplorazione guidata dall'LLM e la validazione tecnica delle vulnerabilità. AWS Security Agent non costituisce però un termine di confronto del tutto sovrapponibile: si tratta di una piattaforma commerciale che copre l'intero ciclo di sicurezza, dalla revisione del design e del codice fino alla remediation, mentre questa tesi si concentra sul supporto al DAST nell'esplorazione delle applicazioni web. Replicare un sistema di tale portata non sarebbe stato realistico con le risorse e i tempi disponibili; la sessione ha comunque fornito importanti spunti progettuali.

AWS Security Agent viene proposto come servizio di penetration testing *on-demand*, in grado di combinare analisi statica, analisi dinamica e scenari di attacco multi-step. La piattaforma può operare su domini preventivamente verificati e, quando necessario, ricevere credenziali, istruzioni di autenticazione, specifiche API, documentazione di architettura e threat model. Il contesto aggiuntivo permette di analizzare percorsi applicativi che non sarebbero raggiungibili mediante il solo crawling black-box.

Architettura del processo di test



Figura 1.3: Architettura AWS security agent.

L'architettura illustrata durante la sessione (Figura 1.3) è organizzata in cinque fasi principali, accomunate da uno stato condiviso, denominato *PentestState*. Tale stato raccoglie progressivamente credenziali, endpoint, documenti, vulnerabilità candidate, evidenze, copertura dei test e risorse create durante l'esecuzione. Ogni fase utilizza quindi le informazioni prodotte da quelle precedenti per pianificare test più mirati.

1. **Autenticazione.** Un agente utilizza un browser headless per attraversare pagine di login, compilare form, gestire redirect e mantenere la sessione. Il sistema considera credenziali tradizionali, token, cookie, API key, header personalizzati e meccanismi TOTP/2FA. L'impiego di più account con privilegi differenti consente inoltre di verificare scenari quali IDOR ed escalation dei privilegi.

2. **Scanning.** Scanner web e API, analizzatori TLS/SSL e controlli sulle dipendenze possono operare in parallelo. In modo sequenziale, l'analisi del codice, dei documenti architetturali e delle specifiche OpenAPI contribuisce all'estrazione di endpoint e al crawling autenticato dell'applicazione.

3. **Esecuzione gestita.** Task relativi a diverse classi di vulnerabilità vengono distribuiti a worker concorrenti. Tra le categorie considerate rientrano XSS, SQL Injection, SSRF, command injection, SSTI, IDOR, path traversal, file upload, XXE, vulnerabilità JWT ed escalation dei privilegi.

4. **Esplorazione guidata.** Un agente stratega analizza lo stato del test, gli endpoint scoperti, i risultati confermati, le specifiche disponibili e le lacune di copertura. Su questa base pianifica nuove verifiche e aggiorna la strategia in funzione delle risposte dell'applicazione.

5. **Cleanup.** Account, file, dati e configurazioni creati durante il test vengono registrati, rimossi e verificati al termine del processo, così da lasciare il bersaglio in uno stato coerente con quello iniziale e mantenere una traccia di audit.

L'aspetto distintivo del modello non risiede soltanto nell'automazione delle azioni, ma nell'uso progressivo del contesto: il risultato di un'attività di scanning può orientare i test successivi, mentre le evidenze raccolte possono modificare la priorità degli endpoint e delle ipotesi di attacco. Questo approccio supera la logica dello scanner tradizionale, che tende a valutare richieste ed endpoint in modo più indipendente. Alcune di queste funzionalità sono state valutate e introdotte in ZAPilot.

Validazione delle evidenze

La sessione ha posto particolare attenzione al problema dei falsi positivi. Una segnalazione segue un processo composto da individuazione, riproduzione, verifica tramite asserzioni, attribuzione di un punteggio e conferma finale. L'elevata confidenza richiede evidenze multiple oppure una prova deterministica; un risultato viene invece classificato come falso positivo se la fase di riproduzione produce elementi che contraddicono l'ipotesi iniziale.

Il modello linguistico può supportare la pianificazione dei test e la valutazione delle evidenze, ma il suo giudizio non costituisce, di per sé, una prova dell'esistenza di una vulnerabilità. Nel contesto di ZAPilot, i vincoli progettuali e le risorse disponibili non hanno consentito di adottare un'architettura agentica altrettanto articolata. Si è quindi scelto un approccio *LLM-as-a-Judge*, nel quale il modello valuta le evidenze raccolte ed è affiancato da controlli deterministici che contribuiscono a rendere più affidabile la decisione finale. Questo aspetto verrà approfondito nei capitoli successivi.

Sicurezza dell'agente

Un agente impiegato nel penetration testing presenta una criticità intrinseca: deve disporre di capacità sufficienti per individuare vulnerabilità reali, ma deve al contempo operare entro limiti rigorosi, così da non arrecare danni al sistema analizzato. Durante la presentazione AWS è stato proposto un approccio di *defense in depth*, nel quale le restrizioni non vengono affidate esclusivamente al ragionamento probabilistico del modello.

Un primo livello di protezione riguarda la definizione preventiva del perimetro operativo: domini, URL accessibili, operazioni di rete e strumenti disponibili devono essere stabiliti prima dell'esecuzione. Un secondo livello è costituito da policy di autorizzazione deterministiche. Infine, i guardrail intervengono sugli input e sugli output del modello e comprendono il filtraggio dei contenuti, la protezione dalla prompt injection e la gestione dei dati sensibili.

Nel progetto sviluppato in questa tesi, l'unico strumento messo a disposizione del modello è Playwright, utilizzato per compilare form sulla base del contesto della pagina. Di conseguenza, la sicurezza relativa all'accesso a strumenti eterogenei non si

è dimostrata necessaria nel perimetro attuale del progetto. Non sono infatti previste l'esecuzione di comandi sul sistema, l'invocazione di strumenti offensivi aggiuntivi o l'accesso autonomo a risorse esterne rispetto alla navigazione dell'applicazione target, sebbene quest'ultima possibilità sia tecnicamente realizzabile.

I test rimangono comunque limitati ai target autorizzati, alle richieste consentite e a condizioni di arresto definite dall'operatore. Come verrà illustrato nelle sezioni successive, per ogni target è possibile definire una configurazione iniziale contenente i parametri di login, gli eventuali overlay da ignorare e lo *scope* entro il quale l'esplorazione deve rimanere confinata. Nella modalità *discovery* è inoltre possibile specificare la pagina target che il sistema deve raggiungere.

L'introduzione di guardrail costituisce un possibile sviluppo futuro. Tale aspetto non è stato considerato nell'implementazione corrente, in ragione del perimetro limitato degli strumenti disponibili al modello, ma rimane un tema rilevante nel caso di future estensioni del progetto. Questa limitazione sarà ripresa nella Sezione 6.2.

1.3 Obiettivi della tesi

L'obiettivo principale della tesi è valutare se l'integrazione di ZAPilot con uno strumento DAST esistente produca un miglioramento concreto rispetto all'impiego del solo scanner. In particolare, il sistema mira a incrementare sia la capacità di esplorare l'applicazione e raggiungere nuovi endpoint, sia il numero di vulnerabilità individuate. Tutto questo utilizzando modelli on-premise[1] Sigma, creando così uno strumento riutilizzabile che possa contribuire a rendere più sicure le applicazioni sviluppate internamente, mitigando la necessità di affidarsi ad enti terzi, velocizzando la fase di testing e diminuendo i costi aziendali.

Il contributo di ZAPilot deve tradursi in un beneficio misurabile, vale a dire in un miglioramento che giustifichi lo sforzo e la complessità introdotti dalla sua integrazione. Non si richiede, tuttavia, di individuare ogni vulnerabilità presente, ne di realizzare un penetration tester automatico. Lo scopo non è sostituire gli strumenti DAST, ma affiancarli nei passaggi in cui l'esplorazione e la scelta delle azioni richiedono una comprensione del contesto applicativo.

[1] Un modello di gestione informatica in cui software, hardware e dati sono installati e ospitati fisicamente sui server e sui computer all'interno della sede o dei locali di un'azienda.

CAPITOLO 2

Metodologie & Strumenti

In questo capitolo vengono presentate le principali metodologie, tecnologie e componenti software impiegati nella realizzazione del sistema. La loro integrazione architetturale e il funzionamento operativo della pipeline saranno descritti nei capitoli successivi.

2.1 Metodologie

2.1.1 SAST, DAST e IAST

La sicurezza di un'applicazione può essere analizzata in momenti diversi del suo ciclo di vita e da prospettive differenti. Tra gli approcci più diffusi rientrano il *Static Application Security Testing* (SAST), il *Dynamic Application Security Testing* (DAST) e l'*Interactive Application Security Testing* (IAST). Sebbene il loro obiettivo comune sia individuare vulnerabilità e comportamenti non sicuri, essi differiscono per il tipo di accesso al sistema, per il momento in cui l'analisi viene condotta e per la natura dell'evidenza raccolta.

SAST

Il SAST opera sul codice sorgente, sul bytecode o su altri artefatti statici dell'applicazione. È normalmente eseguito nelle prime fasi dello sviluppo e consente di rilevare pattern potenzialmente pericolosi prima che il software venga distribuito. Il vantaggio principale consiste nella possibilità di localizzare il problema direttamente nel codice e di fornire indicazioni puntuali allo sviluppatore. Tuttavia, l'analisi statica non osserva l'applicazione nel suo ambiente di esecuzione e può produrre risultati che non sono effettivamente "sfruttabili" a runtime.

IAST

L'IAST si colloca in una posizione intermedia. Durante l'esecuzione dei test, strumenti o agenti vengono integrati nell'applicazione per osservare il flusso dei dati, le chiamate interne e il comportamento del codice. Questo approccio unisce la visibilità interna tipica dell'analisi statica con l'osservazione dinamica dei test in esecuzione. Il principale limite è rappresentato dalla necessità di modificare l'ambiente applicativo, condizione non sempre possibile in contesti reali o in scenari di testing esterno.

DAST

Il DAST analizza un'applicazione durante la sua esecuzione, senza richiedere l'accesso al codice sorgente. Il target viene osservato dall'esterno attraverso le richieste HTTPS, le risposte del server, il DOM renderizzato dal browser, le interazioni con l'interfaccia utente e le variazioni dello stato applicativo. Per questa ragione, il DAST viene generalmente associato a un approccio *black-box*, oppure *grey-box* quando il tester dispone di informazioni parziali sulla struttura o sulle funzionalità dell'applicazione.

L'obiettivo è individuare debolezze osservabili a runtime e raccogliere evidenze sul loro impatto effettivo. Tra queste rientrano errori nella validazione degli input, configurazioni insicure, esposizione di risorse sensibili e problemi di autenticazione o autorizzazione, valutando il sistema dalla prospettiva di un potenziale attaccante.

Il sistema sviluppato in questa tesi adotta principalmente questa metodologia. In questo modo è possibile valutare non soltanto la presenza di endpoint e parametri, ma anche il comportamento effettivo dell'applicazione dopo l'invio di un input o l'esecuzione di una sequenza di azioni. Di conseguenza, nei capitoli successivi l'attenzione sarà rivolta al DAST; SAST e IAST, pur rappresentando approcci complementari utili per una valutazione più completa, non rientrano nell'ambito del sistema proposto e non verranno ulteriormente approfonditi. ZAPilot utilizza ZAP come motore DAST, ma introduce un induttore LLM-driven che esercita l'applicazione in modo stateful. La metodologia si ispira ai principi IAST (induttore + osservazione dinamica), pur rimanendo basata su un sensore esterno senza instrumentation lato applicativo.

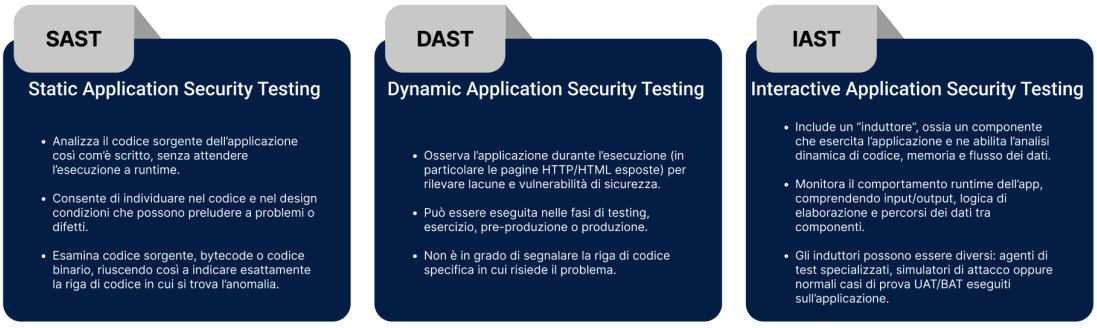


Figura 2.1: Confronto concettuale tra SAST, DAST e IAST.

2.1.2 OWASP Top Ten

L'OWASP Top Ten è una delle principali risorse di riferimento per la sicurezza delle applicazioni web. Pubblicata dall'Open Worldwide Application Security Project, raccoglie dieci macro-categorie di rischio considerate particolarmente rilevanti nello sviluppo e nella gestione di applicazioni esposte sul web. La classificazione non deve essere interpretata come un elenco esaustivo di tutte le vulnerabilità esistenti, bensì come uno strumento di sensibilizzazione e prioritizzazione, utile per orientare attività di sviluppo sicuro, revisione del codice e security testing (OWASP Foundation [2025]).

Nel contesto di questa tesi, l'OWASP Top Ten costituisce un riferimento per la definizione delle ipotesi di attacco considerate dalla pipeline. Il sistema non assume che ogni superficie individuata sia vulnerabile, ma utilizza le categorie OWASP per guidare l'analisi dei campi di input attraverso quella che è l'llm-wiki, trattata successivamente nella sezione 3.7.

L'edizione più recente, OWASP Top Ten 2025 (OWASP Foundation [2025]), comprende le categorie riportate nella tabella 2.1.

Le categorie dell'OWASP Top Ten presentano livelli diversi di osservabilità attraverso tecniche di analisi dinamica. Vulnerabilità quali *Broken Access Control*, *Security Misconfiguration*, *Injection*, *Authentication Failures*, possono manifestarsi durante l'esecuzione dell'applicazione, attraverso risposte inattese, differenze nel comportamento riservato a utenti diversi, messaggi di errore, validazioni insufficienti o effetti prodotti dall'invio di input controllati.

Altre categorie non possono invece essere valutate in modo completo dalla sola osservazione esterna del comportamento applicativo. *Insecure Design* richiede l'analisi dei requisiti e dei workflow previsti in fase progettuale; *Cryptographic Failures* può richiedere la valutazione delle implementazioni, della gestione delle chiavi e delle configurazioni; le categorie relative alla supply chain, all'integrità del software e al logging richiedono inoltre evidenze provenienti da configurazioni del sistema. L'OWASP Top Ten deve essere considerato come una tassonomia di riferimento per la gestione del rischio, non come un insieme di categorie uniformemente rilevabili da un'unica metodologia di testing.

La categoria A09:2025, relativa alle carenze di logging e alerting, si collega perfettamente alle indicazioni NIST per le architetture *Zero Trust*. La guida pratica SP 1800-35, pubblicata in versione definitiva nel giugno 2025, prevede che le decisioni di accesso siano supportate dal monitoraggio continuo e dalle informazioni raccolte su identità, dispositivi e risorse, contribuendo a rilevare anomalie e a rivalutare gli accessi (National Institute of Standards and Technology [2025]). La sola registrazione degli eventi, senza analisi e risposta agli alert, non è sufficiente.

Tabella 2.1: Categorie dell'OWASP Top Ten 2025.

ID	Categoria e descrizione sintetica
A01:2025	**Broken Access Control.** Raccoglie le debolezze che consentono a un utente di accedere a risorse, funzionalità o dati per i quali non possiede le necessarie autorizzazioni. Rientrano in questa categoria, ad esempio, l'accesso diretto a oggetti appartenenti ad altri utenti e l'assenza di controlli lato server sulle operazioni sensibili.
A02:2025	**Security Misconfiguration.** Comprende configurazioni insicure di server, framework, servizi cloud, database, header HTTP, permessi, messaggi di errore e funzionalità non necessarie lasciate attive in produzione.
A03:2025	**Software Supply Chain Failures.** Include i rischi legati a dipendenze, strumenti di build, repository, pipeline CI/CD, distribuzione degli artefatti e aggiornamenti software. La categoria considera quindi la sicurezza dell'intero processo attraverso il quale il software viene sviluppato, distribuito e aggiornato.
A04:2025	**Cryptographic Failures.** Riguarda l'uso assente, scorretto o debole della crittografia nella protezione dei dati. Esempi comprendono la trasmissione di informazioni sensibili senza adeguata protezione, algoritmi obsoleti, chiavi gestite in modo non sicuro e credenziali memorizzate senza protezioni adeguate.
A05:2025	**Injection.** Comprende le vulnerabilità nelle quali input controllati dall'utente vengono interpretati come istruzioni o comandi da un componente dell'applicazione. Le forme più note includono SQL Injection, command injection, LDAP injection e Cross-Site Scripting, quando l'input viene inserito in un contesto HTML o JavaScript eseguibile.
A06:2025	**Insecure Design.** Indica carenze progettuali che rendono l'applicazione vulnerabile anche in presenza di un'implementazione tecnicamente corretta. Rientrano in questa categoria l'assenza di controlli di sicurezza fondamentali, workflow non progettati per gestire gli abusi e requisiti di protezione insufficienti.
A07:2025	**Authentication Failures.** Include debolezze nei meccanismi di identificazione, autenticazione, gestione delle sessioni e recupero delle credenziali. Tali problemi possono consentire l'impersonificazione di utenti, il riutilizzo di sessioni o l'accesso non autorizzato agli account.
A08:2025	**Software or Data Integrity Failures.** Riguarda l'esecuzione o l'elaborazione di software, aggiornamenti, dati e oggetti serializzati senza un'adeguata verifica della loro integrità e provenienza. Può includere aggiornamenti non verificati, deserializzazione insicura e fiducia eccessiva in componenti esterni.
A09:2025	**Security Logging and Alerting Failures.** Comprende l'assenza, la scarsa qualità o la mancata gestione dei log di sicurezza e degli alert. Una registrazione insufficiente può ritardare l'identificazione di attacchi, compromissioni e comportamenti anomali, rendendo più difficile la risposta agli incidenti.
A10:2025	**Mishandling of Exceptional Conditions.** Raccoglie errori nella gestione di condizioni inattese, eccezioni, fallimenti applicativi e situazioni di errore. Un'applicazione può diventare vulnerabile quando, in presenza di eventi anomali, espone informazioni sensibili, applica comportamenti incoerenti o adotta una logica *fail-open*.

2.2 Strumenti

2.2.1 OWASP ZAP



Figura 2.2: Logo dello strumento. Fonte: ZAP Development Team [2026].

OWASP Zed Attack Proxy, generalmente indicato come OWASP ZAP, è uno strumento open source per il security testing di applicazioni web. Può operare come proxy intercettante, posizionandosi tra browser e target. In questa configurazione registra richieste, risposte, cookie, parametri e messaggi HTTP associati alla sessione di test.

Nel sistema proposto, ZAP costituisce la base della discovery iniziale. Il Classic Spider analizza le risorse raggiungibili tramite collegamenti e navigazione HTTP convenzionale, mentre l'AJAX Spider viene impiegato per esplorare applicazioni nelle quali contenuti e route vengono generati tramite JavaScript. L'uso congiunto delle due componenti amplia la copertura rispetto a un singolo crawler, soprattutto nel caso di applicazioni dinamiche.

ZAP viene inoltre utilizzato per avviare la scansione attiva degli endpoint rientranti nello scope definito dal profilo del target. Il browser Playwright utilizzato dalla pipeline viene configurato per instradare il traffico tramite il proxy ZAP; di conseguenza, le azioni effettuate durante l'esecuzione dei piani generati dal sistema risultano visibili anche nella history HTTP. Questa integrazione rende possibile associare le evidenze osservate dal browser a richieste e risposte di rete concrete.

Quando un finding viene confermato dall'Evaluator, il sistema può registrare un alert personalizzato in ZAP e collegarlo, se disponibile, al messaggio HTTP più rilevante. In questo modo i risultati prodotti dalla pipeline LLM-assisted rimangono consultabili anche attraverso gli strumenti e i report dell'ecosistema ZAP.

2.2.2 OWASP Juice Shop



Figura 2.3: Logo dello strumento. Fonte: OWASP Foundation [2026c].

OWASP Juice Shop è un'applicazione web intenzionalmente vulnerabile, progettata per la formazione sulla sicurezza, le esercitazioni di hacking etico in stile *Capture The Flag* e la sperimentazione di strumenti di *vulnerability assessment*. Simula un negozio online e presenta vulnerabilità appartenenti alle principali categorie dell'OWASP Top Ten.

Juice Shop è stato scelto come target sperimentale perché concentra in un'unica applicazione molte delle caratteristiche tipiche dei sistemi web moderni: una interfaccia dinamica, route gestite lato client, form di ricerca, meccanismi di autenticazione, gestione dei profili, procedure di acquisto e API REST. Ciò consente di valutare sia gli strumenti DAST tradizionali, sia il contributo dell'automazione del browser e dei modelli linguistici nell'esplorazione di stati applicativi più articolati.

La disponibilità di vulnerabilità note permette inoltre di condurre gli esperimenti in un ambiente controllato, senza coinvolgere sistemi di terze parti. È quindi possibile riprodurre attacchi conosciuti e verificare se la pipeline sia in grado di individuarli e confermarli, costituendo un ottimo benchmark sul quale valutare ZAPilot.

2.2.3 Podman



Figura 2.4: Logo dello strumento. Fonte: Podman [2026].

Podman e Docker sono piattaforme per la creazione e l'esecuzione di container compatibili con lo standard OCI (Open Container Initiative) e presentano modalità d'uso in larga parte analoghe. Docker adotta tradizionalmente un'architettura basata su un demone centrale, mentre Podman è *daemonless*: i container vengono gestiti come processi ordinari e possono essere eseguiti anche in modalità *rootless*, senza richiedere privilegi amministrativi. Podman offre inoltre un'interfaccia a riga di comando compatibile con gran parte dei comandi Docker, dove al posto del comando *docker* viene utilizzato il comando *podman*, agevolando il riutilizzo di immagini e configurazioni già esistenti.

Nel contesto aziendale in cui è stato sviluppato il progetto è stato adottato Podman poiché disponibile gratuitamente e privo dei costi di licenza che possono interessare l'impiego di Docker Desktop in determinati contesti organizzativi (in alternativa, sarebbe bastato utilizzare docker da cli). La scelta ha consentito di utilizzare una soluzione conforme alle politiche aziendali senza modificare il modello di distribuzione basato su container.

Nel progetto, Podman isola i componenti della pipeline sperimentale, tra cui il target vulnerabile, ZAP, Redis e il servizio di orchestrazione. Ciascun componente è eseguito in un ambiente distinto, collegato agli altri mediante una rete dedicata e dotato di dipendenze definite in modo esplicito. La containerizzazione permette di riprodurre lo stesso ambiente su sistemi diversi, limitando le differenze dovute alle librerie, alla configurazione e al runtime installati sull'host.

Un caso particolare emerso durante lo sviluppo nel quale riproducibilità dell'ambiente si è rivelata importante è per Playwright, poiché la versione della libreria, quella del browser e le relative dipendenze (soprattutto con Chromium di ZAP) di sistema devono essere reciprocamente compatibili. L'impiego di un'immagine predefinita con-

sente di fissare tali versioni, garantire un ambiente uniforme tra esecuzioni diverse e semplificare la distribuzione della pipeline.

2.2.4 Redis



Figura 2.5: Logo dello strumento. Fonte: Redis [2026].

Redis è un data store in-memory basato su chiavi, impiegato comunemente come cache e come database a bassa latenza. Nel sistema sviluppato non sostituisce il grafo persistente che rappresenta l'esplorazione, ma mantiene informazioni che devono essere recuperate rapidamente durante la scansione.

La pipeline utilizza Redis per due scopi. Il primo è evitare di analizzare più volte la stessa superficie applicativa. La chiave di cache viene generata a partire da una rappresentazione deterministica della superficie, dalla modalità operativa e dalla versione dell'applicazione. Se viene trovata una voce corrispondente, il sistema può riutilizzare il risultato precedente, saltare il flusso di attacco già eseguito e proseguire con l'esplorazione di altri nodi. Questo comportamento è utile soprattutto nelle applicazioni di grandi dimensioni, perché permette di riprendere l'analisi senza ripetere operazioni già completate. L'uso della cache è facoltativo ed è controllato da un'apposita opzione di configurazione; quando è disabilitato, la pipeline esegue il flusso ordinario.

Il secondo scopo riguarda il confronto tra versioni diverse dell'applicazione. Se Redis contiene attacchi eseguiti su una versione precedente, la pipeline può ripeterli sulla versione corrente per verificare se le vulnerabilità rilevate siano state corrette. In questo modo la cache riduce le chiamate LLM non necessarie e, allo stesso tempo, supporta il controllo delle regressioni di sicurezza.

Le varie modalità operative verranno approfondite nei capitoli successivi.

2.2.5 Playwright



Figura 2.6: Logo dello strumento. Fonte: Microsoft [2026].

Playwright è un framework per l'automazione dei browser che permette di controllare Chromium e Firefox attraverso un'unica interfaccia. A differenza di un semplice client HTTP, esegue l'applicazione in un browser reale e consente quindi di osservarne il

comportamento a runtime, compresi il codice JavaScript, il DOM renderizzato, i cookie, gli eventi utente, il routing lato client e le finestre di dialog.

Nel sistema sviluppato, Playwright viene impiegato innanzitutto nella fase di discovery. Uno spider dedicato esamina il DOM della pagina ed estrae link, controlli, form, campi di input e altri elementi interattivi che potrebbero non essere presenti nella history HTTP di ZAP. Questo passaggio è particolarmente importante nelle Single Page Application, dove alcune route e risorse diventano disponibili solo dopo l'esecuzione di codice JavaScript o in seguito a un'interazione dell'utente.

Playwright esegue anche i piani prodotti dal Planner. Ogni piano specifica i campi da compilare, i valori da inserire e l'azione necessaria per inviare il form. Le operazioni vengono svolte nel BrowserContext corrente, in modo da mantenere la stessa sessione e conservare cookie, autenticazione e stato della navigazione.

Per ogni tentativo, il sistema registra gli URL e le route visitate, le variazioni del DOM e del testo visibile, gli errori della pagina, le finestre di dialog, le modifiche ai cookie e il traffico di rete generato. Questi dati vengono forniti all'Evaluator, che agisce come *LLM as a judge* e formula un verdetto sulla base delle evidenze raccolte e di regole deterministiche, associandovi un livello di confidenza. Poiché il BrowserContext instrada il traffico attraverso il proxy ZAP, ciò che viene osservato nel browser può essere messo in relazione con le richieste HTTP registrate dallo scanner. In questo modo è possibile analizzare workflow complessi mantenendo nello stesso flusso l'automazione dell'interfaccia e l'osservazione del traffico di rete.

2.2.6 Obsidian



Figura 2.7: Logo dello strumento. Fonte: Obsidian [2026].

Obsidian è un'applicazione per la gestione di note e basi di conoscenza basate su file Markdown collegati tra loro. Nel progetto è stato utilizzato esclusivamente per visualizzare la struttura della LLM Wiki ispirata alla proposta di Karpathy Andrej Karpathy [2026], sfruttando la rappresentazione a grafo delle relazioni tra le note. Tale visualizzazione verrà mostrata nel capitolo dedicato all'implementazione. Obsidian non partecipa tuttavia alla pipeline di ZAPilot né è necessario per la costruzione o l'interrogazione della Wiki: costituisce un componente del tutto facoltativo, sostituibile con un qualsiasi editor o visualizzatore compatibile con file Markdown.

2.3 Modelli valutati

2.3.1 OpenAI GPT-OSS-120B

Il modello `openai/gpt-oss-120b` (OpenAI [2025]) è stato rilasciato da OpenAI il 5 agosto 2025 come modello open weight, distribuito con licenza Apache 2.0. GPT-OSS-120B adotta un'architettura Transformer basata su *Mixture-of-Experts* (MoE) [1]. Il modello possiede 117 miliardi di parametri totali, valore generalmente approssimato nel nome commerciale a 120 miliardi, ma ne attiva soltanto 5,1 miliardi per ogni token durante l'inferenza. Tale caratteristica permette di combinare un'elevata capacità parametrica con un costo computazionale inferiore rispetto a un modello denso di dimensione equivalente. Il modello supporta una finestra di contesto nativa pari a 128.000 token.

Un elemento rilevante per ZAPilot è il supporto a compiti di ragionamento, *function calling*, uso di strumenti e generazione di output strutturati. Queste caratteristiche risultano potenzialmente utili quando il sistema deve interpretare il contesto di una pagina, inferire il significato di un campo di input, generare valori sintatticamente validi oppure produrre una decisione in un formato elaborabile dal componente di orchestrazione.

I pesi sono distribuiti nativamente nel formato MXFP4. OpenAI indica che il modello può essere eseguito entro circa 80 GB di memoria, pur rimanendo necessarie ulteriori risorse per il runtime, il contesto e la cache di attenzione. Tale requisito rende GPT-OSS-120B un candidato adatto a sistemi dotati di GPU ad alta capacità di memoria o a configurazioni di inferenza distribuita, ma ne limita l'impiego su hardware locale di fascia consumer.

2.3.2 Google Gemma 4 31B IT

Il secondo modello considerato è `gemma-4-31b-it` (Google AI for Developers [2026]), variante *instruction-tuned* della famiglia Gemma 4 di Google. Gemma 4 31B è un modello Transformer denso con circa 31 miliardi di parametri. A differenza dei modelli MoE, tutti i parametri del modello partecipano all'inferenza per ciascun token generato. Questa caratteristica rende più immediata l'interpretazione della relazione tra numero totale di parametri e costo computazionale, ma comporta un fabbisogno di memoria e capacità di calcolo maggiore rispetto a un modello sparso con un numero analogo di parametri totali. La famiglia Gemma 4 supporta contesti fino a 256.000 token. Il modello è multimodale: può ricevere testo e immagini in input e genera testo in output. Sebbene la sperimentazione di ZAPilot sia principalmente orientata all'analisi testuale del DOM, delle richieste HTTP e dei form, la disponibilità di capacità visive può risultare utile in evoluzioni future del sistema, ad esempio per interpretare screenshot, captcha non aggirabili automaticamente o interfacce nelle quali informazioni rilevanti siano rese esclusivamente tramite elementi grafici.

[1] Nei modelli MoE, i parametri sono suddivisi tra diversi "esperti", ciascuno costituito da una sottorete specializzata. Per ogni input, una rete di instradamento (o *router*) seleziona e attiva soltanto gli esperti ritenuti più adatti, riducendo così il costo computazionale rispetto all'attivazione dell'intero modello.

Secondo le stime ufficiali, il caricamento di Gemma 4 31B richiede circa 69,9 GB in BF16[2], 34,9 GB in precisione a 8 bit e 17,5 GB in formato Q4.

2.3.3 Qwen3-Coder-30B-A3B-Instruct

Il terzo modello valutato è `Qwen/Qwen3-Coder-30B-A3B-Instruct` (Qwen Team [2025]), appartenente alla famiglia Qwen3-Coder e progettato specificamente per attività di programmazione, comprensione di repository, utilizzo di strumenti e scenari di *agentic coding*. Il modello è stato pubblicato nel 2025 ed è distribuito come modello open weight.

Qwen3-Coder-30B-A3B-Instruct utilizza un'architettura MoE con 30,5 miliardi di parametri totali, dei quali 3,3 miliardi vengono attivati durante l'elaborazione di ciascun token. L'acronimo "A3B" nel nome del modello richiama appunto l'ordine di grandezza dei parametri attivi. Il modello è composto da 48 livelli e dispone di 128 esperti, con 8 esperti attivati per token.

Una caratteristica rilevante è la finestra di contesto nativa pari a 262.144 token, estendibile fino a un milione di token. Il modello supporta un formato dedicato per il *tool calling* ed è stato progettato per l'integrazione con ambienti e strumenti orientati allo sviluppo, quali Qwen Code, Cline e altri sistemi agentici. Per il caso d'uso considerato, tale predisposizione è utile per produrre risposte strutturate e per integrarsi con componenti deterministici incaricati di compilare form, avanzare nella navigazione e registrare gli esiti delle azioni eseguite, ma è stata pensata come evoluzione futura quella di aggiungere funzioni agentiche all'architettura esistente. Molto più interessante in questo progetto è vedere come si comporta messo a confronto con gli altri modelli non specializzati nel coding.

A differenza di altri modelli della famiglia Qwen, questa variante opera esclusivamente in modalità non-*thinking* e non richiede l'impostazione esplicita di un parametro per disabilitare il ragionamento visibile. Questo comportamento può semplificare l'integrazione in pipeline che richiedono output diretto, prevedibile e facilmente sottoponibile a parsing.

2.3.4 Modelli considerati ma non inclusi nella valutazione

Durante la fase di sviluppo sono stati considerati anche i modelli `Qwen/Qwen3.6-35B-A3B` e `Qwen/Qwen3-Coder-Next` . Entrambi non sono stati inclusi nella valutazione sperimentale finale poiché non più disponibili nell'ambiente di esecuzione adottato al momento dei test. Essi vengono pertanto menzionati soltanto per completezza e non concorrono ai risultati riportati nei capitoli successivi.

[2]BF16 (Brain Floating Point) è è un formato numerico in virgola mobile a 16 bit ideato da Google. Dedica 1 bit per il segno, 8 bit per l'esponente (contro i 5 di FP), 7 bit per la frazione (contro i 10 su FP) potendo rappresentare numeri enormi o incredibilmente piccoli senza andare in errore.

CAPITOLO 3

Progettazione

Questo capitolo presenta la progettazione di ZAPilot, descrivendo le scelte architetturali adottate, i problemi emersi durante lo sviluppo e le ragioni che hanno condotto alla configurazione attuale. L'obiettivo non è esaurire qui tutti i dettagli implementativi; gli aspetti più tecnici e alcuni esempi di funzionamento sono demandati ai capitoli successivi. L'architettura illustrata non deve inoltre essere considerata definitiva, ma come il risultato di numerose modifiche incrementali, introdotte attraverso successivi cicli di sperimentazione e revisione. Per la natura evolutiva del progetto, è prevedibile che ulteriori sviluppi portino in futuro a nuove modifiche e a un progressivo affinamento delle soluzioni descritte.

3.1 Architettura generale e flusso operativo

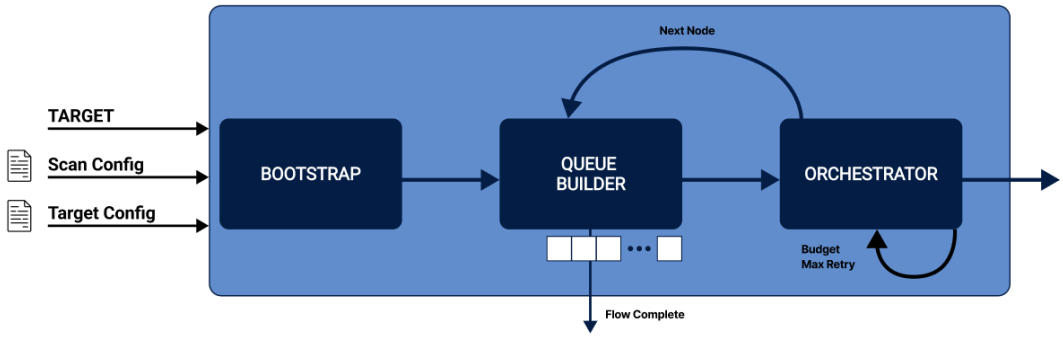


Figura 3.1: Architettura generale di ZAPilot e principali fasi del flusso operativo.

La progettazione del sistema prende avvio dalla definizione della sua architettura complessiva. Nelle prime sperimentazioni si era ipotizzato di affidare a OWASP ZAP,

mediante appositi script, anche il compito di individuare le pagine contenenti form sufficientemente complessi da richiedere l'intervento di ZAPilot. Tale soluzione, tuttavia, avrebbe attribuito al DAST responsabilità estranee al suo scopo principale. Inoltre, le sole richieste e risposte http osservate da ZAP non forniscono sempre informazioni sufficienti per stabilire se lo stato corrente dell'applicazione necessiti di un'esplorazione più approfondita da parte dell'Orchestrator.

Per garantire una chiara separazione delle responsabilità, il paradigma è stato invertito: ZAPilot governa l'intero flusso operativo e utilizza ZAP per le attività in cui gli strumenti DAST risultano maggiormente efficaci, in particolare il crawling mediante spider e l'esecuzione dell'Active Scan. L'Orchestrator conserva invece il controllo dell'esplorazione applicativa e valuta, a partire dal DOM e dallo stato della pagina, quali nodi richiedano ulteriori analisi o tentativi di attacco.

Come illustrato nella figura 3.1, il flusso complessivo si articola in tre macro-fasi: la configurazione iniziale del sistema e del target; una discovery globale, durante la quale ZAP viene impiegato per individuare e analizzare le risorse esposte; infine, l'esplorazione iterativa dei nodi applicativi, coordinata dall'Orchestrator. La figura 3.2 offre una vista d'insieme di quest'ultima fase e anticipa la distinzione tra i flussi di *discovery* e *attack*, descritti più nel dettaglio nelle sezioni successive. Entrambi prendono avvio dai nodi selezionati e condividono alcuni componenti dell'Orchestrator, ma perseguono obiettivi differenti: il primo amplia la conoscenza degli stati e dei percorsi raggiungibili dell'applicazione, mentre il secondo analizza i nodi individuati e coordina i tentativi di attacco. La memoria consente infine di conservare e rendere disponibili le informazioni prodotte durante le diverse iterazioni.

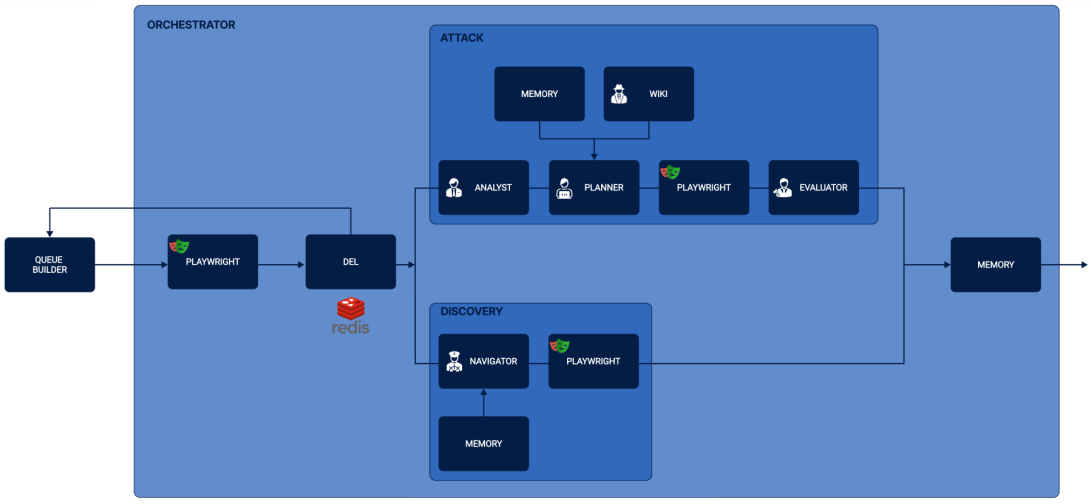


Figura 3.2: Flusso generale delle attività di discovery e attack coordinate dall'Orchestrator.

L'avvio di una scansione richiede la personalizzazione di due file: una configurazione generale, che raccoglie i parametri relativi alle attività di esplorazione e attacco, e una configurazione specifica del target. La struttura generica è descritta nella prossima

sezione, mentre il contenuto di tali file sono descritti nel capitolo successivo.

3.2 Autenticazione e bootstrap della sessione

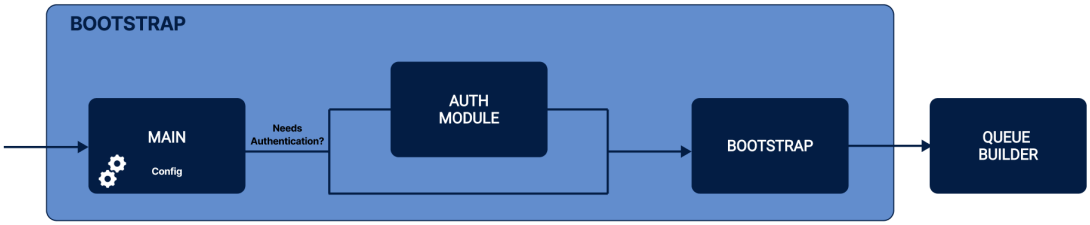


Figura 3.3: Fase di bootstrap.

Il modulo di bootstrap ha il compito di predisporre una sessione valida prima dell'avvio delle successive attività di scansione. Nel corso della sperimentazione sono emerse diverse criticità, che hanno reso necessaria l'introduzione di file di configurazione specifici per ciascun target. Tali file sono facoltativi: qualora non siano presenti o non vengano indicati esplicitamente, il sistema utilizza una configurazione predefinita.

Molte applicazioni aziendali sono protette da provider di identità esterni quali, a titolo di esempio, Keycloak (ref. Keycloak [2026]). Poiché l'obiettivo dell'analisi non è verificare la sicurezza di tali piattaforme, bensì quella dell'applicazione sottostante, il sistema deve essere in grado di superare la fase di autenticazione. Le modalità di accesso e le credenziali variano però da applicazione ad applicazione; è stato quindi necessario progettare un meccanismo di login adattabile alle caratteristiche di ciascun target.

La configurazione, inizialmente introdotta per gestire l'autenticazione, è stata successivamente estesa ad altri aspetti dell'esplorazione. In particolare, consente di delimitarne l'ambito, evitando che collegamenti verso risorse esterne o siti non pertinenti, vengano visitati e inseriti nella coda delle pagine da analizzare. Gli stessi vincoli permettono inoltre di escludere URL che causano la disconnessione dell'utente: in alcune applicazioni, infatti, determinati collegamenti eseguivano il logout subito dopo l'accesso, invalidando i cookie esportati e compromettendo le fasi successive del workflow.

La fase di bootstrap offre vantaggi anche per il processo di discovery. La configurazione può indicare uno specifico nodo dell'applicazione di cui verificare la raggiungibilità; nel caso descritto nella sezione 5.1, ad esempio, il workspace è stato definito come obiettivo. Sebbene non sia sempre possibile conoscere in anticipo la risorsa da individuare, questa funzionalità risulta particolarmente utile nei test interni, nei quali si vuole accertare se un endpoint che dovrebbe essere inaccessibile dall'esterno sia stato invece raggiunto durante l'esplorazione.

Un'ulteriore esigenza riguarda la gestione dei cosiddetti "finti dialog". All'avvio di una nuova sessione possono comparire banner per il consenso all'uso dei cookie o altri elementi sovrapposti. Se non rimossi, questi elementi alterano il DOM e possono

interferire con le azioni eseguite da Playwright, inducendo il browser a selezionare controlli errati e compromettendo l'intero workflow. La rimozione dei comuni overlay e delle normali finestre di dialogo è relativamente semplice; il problema diventa più complesso quando, come osservato in Juice Shop, tali elementi sono realizzati mediante normali `div` che si sovrappongono alle pagine e vengono inclusi nella superficie estratta. Per questo motivo, la configurazione di bootstrap consente di specificare manualmente gli elementi da rimuovere. Tale operazione richiede una conoscenza minima dell'applicazione analizzata, giustificata tuttavia dalla natura specifica di questi casi.

Il flusso illustrato nella figura 3.3 ha inizio nel modulo principale, che carica la configurazione generale e avvia l'esplorazione. Il bootstrap acquisisce quindi il profilo associato al target e, quando richiesto, esegue la procedura di autenticazione. Dopo aver verificato l'accesso, la sessione viene esportata verso ZAP; l'esplorazione può così iniziare in un contesto autenticato e nel rispetto dei vincoli definiti dalla configurazione.

3.3 Queue Builder

Il *Queue Builder* costruisce e aggiorna l'insieme ordinato dei nodi da esplorare. A partire da un *seed URL*, il componente individua nuove risorse e nuovi stati applicativi, filtra i risultati ottenuti e inserisce nella coda i nodi che devono essere elaborati. La gestione della coda non è limitata alla fase di avvio, ma prosegue durante l'intera scansione in collaborazione con l'Orchestrator.

Il Queue Builder e l'Orchestrator operano infatti in modo ciclico. Il Queue Builder fornisce all'Orchestrator il nodo successivo; l'Orchestrator ne esegue l'elaborazione e, al termine, restituisce il controllo al Queue Builder. Su questo nuovo stato viene avviata una fase intermedia di discover, così da individuare eventuali risorse rese raggiungibili dalle interazioni appena eseguite. I nuovi nodi vengono aggiunti alla coda e il ciclo continua fino al suo esaurimento.

Il funzionamento del componente è articolato in tre parti: initial phase, re-discover e final phase. Le tre fasi riutilizzano le medesime operazioni di base, ma le combinano in modo differente in funzione del momento della scansione.

La figura 3.4 mostra il ciclo complessivo del Queue Builder e la sua interazione con l'Orchestrator.

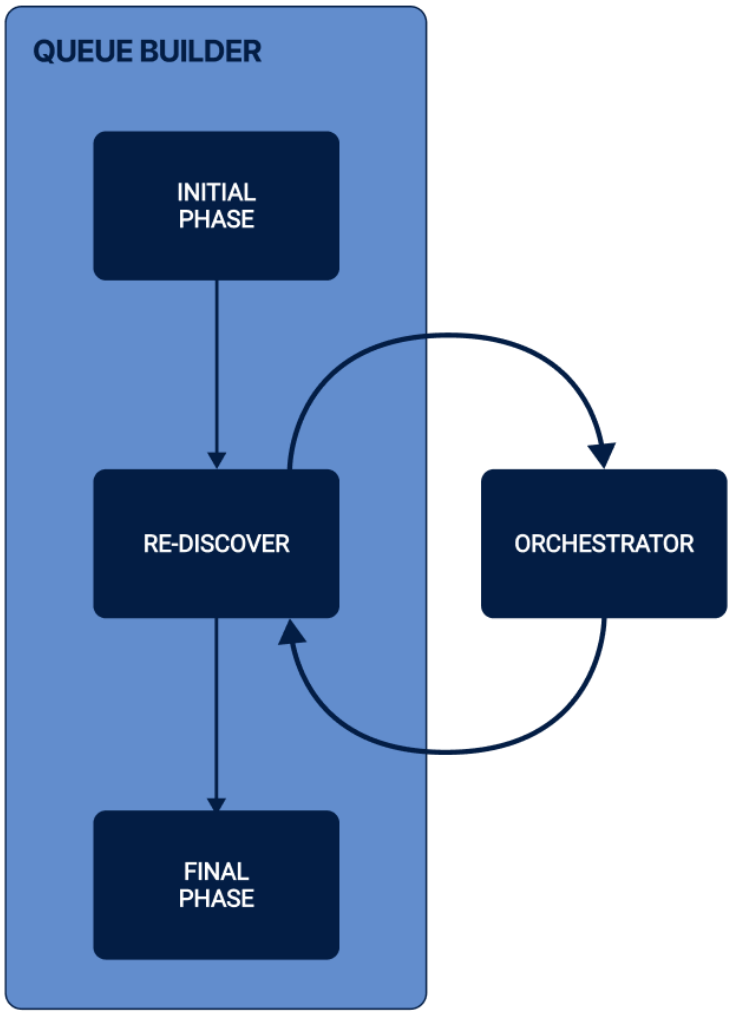


Figura 3.4: Struttura del Queue Builder e ciclo di elaborazione con l'Orchestrator.

3.3.1 Initial Phase

La fase iniziale viene eseguita sull'url di partenza e comprende le operazioni di discover, di filtro e attacco. La discover costruisce il primo insieme di url e stati applicativi conosciuti; il filtro seleziona i risultati da mantenere classificando tra tipo di pagina, file o pagine fuori scope; l'attacco opera poi sulle risorse selezionate. I nodi ottenuti costituiscono la coda iniziale dalla quale il primo elemento verrà consegnato all'Orchestrator.

La figura 3.5 riassume le operazioni eseguite nella fase iniziale e i relativi risultati.

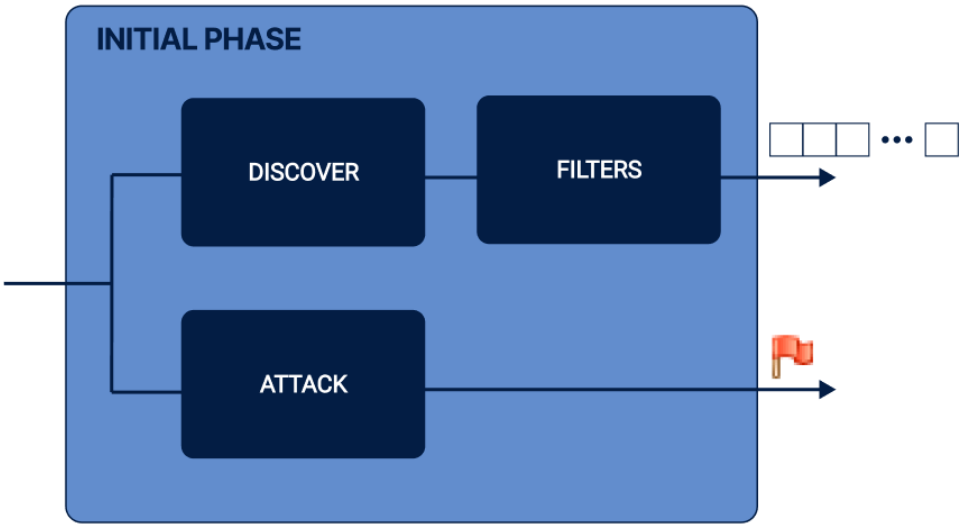


Figura 3.5: Initial Phase: discover e filtro costruiscono la coda iniziale, mentre l'attacco produce le segnalazioni di sicurezza.

In questa fase vengono inoltre avviati i controlli Gitleaks e Sensitive Path Check sulla history HTTP accumulata da ZAP. Poiché tali controlli analizzano l'intera history, non vengono ripetuti durante ogni singola chiamata di discover.

3.3.2 Re-discover

La fase intermedia esegue esclusivamente la discover. Essa viene avviata ogni volta che l'Orchestrator termina l'elaborazione di un nodo e restituisce il controllo al Queue Builder. Lo scopo è osservare nuovamente l'applicazione dopo le interazioni effettuate sul nodo appena concluso e rilevare route, risorse o stati che non erano accessibili in precedenza.

I risultati della riscoperta vengono utilizzati per aggiornare la coda prima di selezionare il nodo successivo. In questo modo l'esplorazione non rimane vincolata alla fotografia iniziale dell'applicazione, ma si amplia progressivamente sulla base degli stati raggiunti dall'Orchestrator. La fase intermedia non avvia attività di attacco.

La figura 3.6 rappresenta l'aggiornamento della coda mediante una nuova discover e il successivo filtraggio dei risultati.

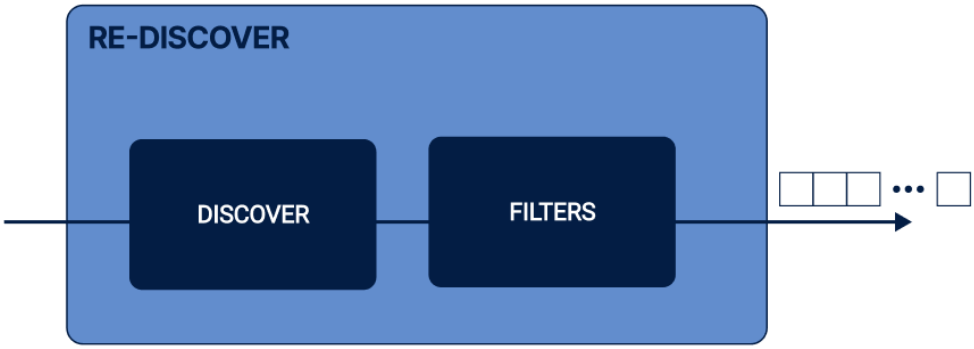


Figura 3.6: Re-discover: aggiornamento incrementale della coda dopo l'elaborazione di un nodo.

3.3.3 Final Phase

La fase finale viene eseguita quando non rimangono altri nodi da elaborare e comprende esclusivamente l'attacco. Essa opera sulla superficie applicativa complessivamente individuata durante la fase iniziale e le successive riscoperte intermedie.

La figura 3.7 mostra l'esecuzione conclusiva delle verifiche sulla superficie applicativa complessivamente scoperta.

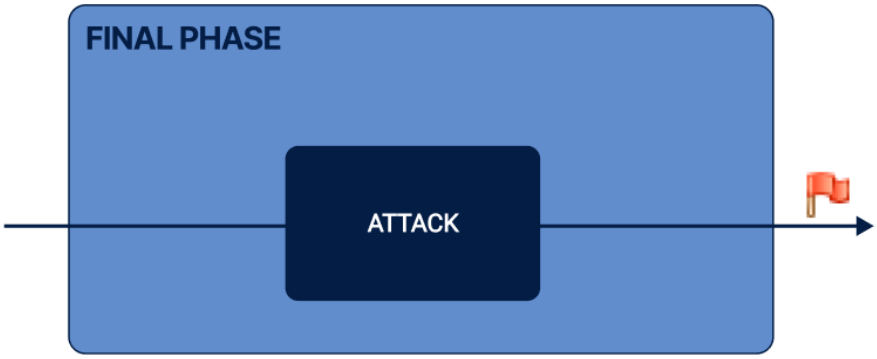


Figura 3.7: Final Phase: esecuzione conclusiva dell'attacco dopo l'esaurimento della coda.

Al termine del ciclo vengono inoltre rieseguiti Gitleaks e Sensitive Path Check sulla history HTTP aggiornata di ZAP. In questo modo i controlli cumulativi includono anche le risposte e le risorse emerse durante l'elaborazione dei nodi.

3.3.4 Discover

La discover combina tre meccanismi complementari: il Classic Spider di ZAP, l'AJAX Spider di ZAP e un crawler basato su Playwright. Il loro uso congiunto consente di

raccogliere sia le risorse esposte attraverso collegamenti e richieste HTTP convenzionali, sia gli stati raggiungibili esclusivamente tramite interazioni client-side.

La figura 3.8 rappresenta la sequenza adottata nelle operazioni di discovery.

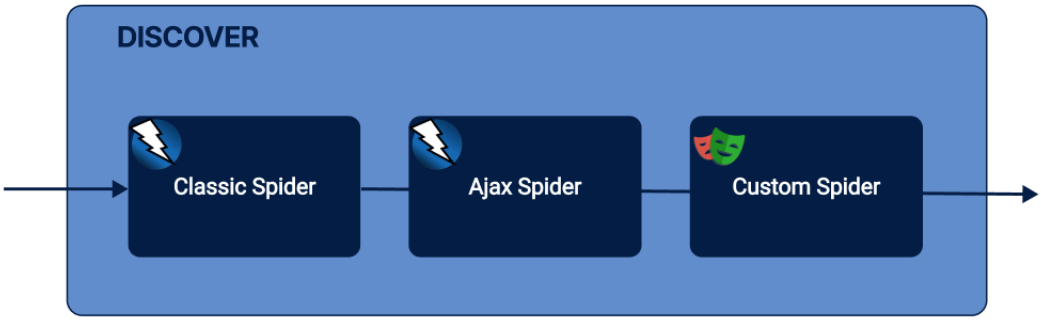


Figura 3.8: Pipeline di discovery unificata: ZAP Classic Spider, ZAP AJAX Spider e Playwright DOM crawl.

Classic Spider di ZAP

Il Classic Spider viene eseguito sullo URL ricevuto dal Queue Builder e raccoglie le risorse individuabili nella struttura dell'applicazione. Lo spider opera entro un tempo massimo configurabile e viene interrotto esplicitamente qualora non raggiunga uno stato terminale entro tale limite. Questo impedisce che una singola attività di crawling blocchi il proseguimento della pipeline.

AJAX Spider di ZAP

L'AJAX Spider è simile allo spider classico , ma orientato all'analisi delle applicazioni dinamiche. L'addon viene configurato con limiti sulla durata, sulla profondità di crawl e sul numero di stati esplorabili.

Crawler Playwright

L'introduzione del crawler basato su Playwright deriva dai risultati ottenuti durante i test su OWASP Juice Shop. Nei primi tentativi, né il Classic Spider né l'AJAX Spider di ZAP riuscivano a individuare in modo affidabile gli URL e gli stati di navigazione generati lato client. Anche ripetendo la discover con l'AJAX Spider, una parte delle route della SPA rimaneva esclusa dalla superficie conosciuta e non poteva essere inserita nella coda del queue builder.

Per superare questo limite è stato introdotto un terzo canale di discover, che utilizza Playwright e un browser Chromium configurato per transitare attraverso il proxy di ZAP. Il crawler apre la pagina (in un contesto che può includere i cookie di sessione ottenuti durante l'autenticazione) e dopo il caricamento, il sistema estrae gli URL presenti negli elementi di navigazione, individua gli elementi che possono aprire menu,

ne analizza le voci e registra le route prodotte dalle interazioni. Dopo ogni operazione, il browser ritorna allo stato precedente per proseguire con gli altri elementi.

Nei test su Juice Shop questo approccio ha consentito di recuperare gli stati client-side che i crawler di ZAP non erano riusciti a rilevare. Juice Shop costituisce tuttavia soltanto il caso di validazione: il meccanismo è generale e non include logiche hard-coded per questa specifica applicazione, pertanto può essere applicato anche ad altre applicazioni web. Le route delle SPA contenenti fragment, ad esempio `#/dashboard`, vengono conservate tra i risultati. Sebbene il fragment non venga trasmesso nelle richieste HTTP, esso identifica uno stato di navigazione distinto dell'applicazione client-side.

3.3.5 Attacco

La componente di attacco riceve le risorse selezionate dal Queue Builder e avvia le verifiche previste dalla scansione. Nel ciclo descritto viene eseguita durante la fase iniziale, dopo discover e filtro, e durante la fase finale, dopo l'esaurimento della coda. Non viene invece richiamata nelle fasi intermedie, che hanno il solo compito di aggiornare la superficie conosciuta attraverso una nuova discover.

La figura 3.9 riassume la pipeline di attacco, composta dall'Active Scan di ZAP e dai controlli Gitleaks e Sensitive Path Check.

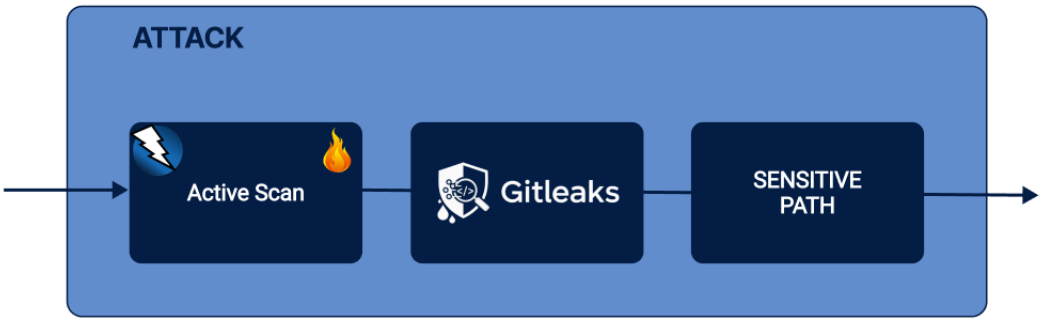


Figura 3.9: Pipeline di attacco: Active Scan di ZAP, Gitleaks e Sensitive Path Check.

L'ActiveScan è uno strumento messo a disposizione direttamente da ZAP per fare test automatici di attacco: usa gli URL, parametri e form scoperti per inviare richieste con payload di test e cercare vulnerabilità note. Non si limita a osservare l'applicazione: genera traffico potenzialmente invasivo (dimostrazione del fatto che durante le scansioni di Burp professional su Juice Shop la macchina è stata restartata numerose volte). Come visto nel capitolo introduttivo però, questi attacchi da soli non bastano per garantire la sicurezza dell'applicazione in quanto poveri di contesto. Gitleaks e Sensitive Path Check invece sono controlli complementari. Gitleaks analizza il contenuto delle risposte HTTP mediante regole ed espressioni regolari, con l'obiettivo di individuare credenziali, token e altri segreti esposti. Il Sensitive Path Check opera invece sui nomi e sui percorsi delle risorse individuate, segnalando file che, per la loro natura o denominazione, non

dovrebbero essere accessibili pubblicamente. Il primo controllo rileva informazioni sensibili in base al contenuto, mentre il secondo identifica esposizioni rischiose in base al nome del file o al percorso, anche quando il contenuto non corrisponde a una firma nota.

3.4 Deterministic Extraction Layer

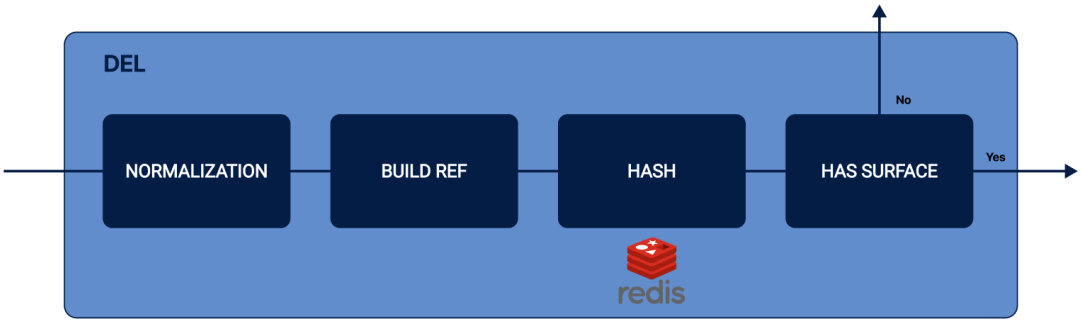


Figura 3.10: Pipeline del *Deterministic Extraction Layer*.

Il *Deterministic Extraction Layer* (DEL), la cui pipeline è rappresentata in Figura 3.10, costituisce il livello di trasformazione deterministica tra l'estrattore del browser e i componenti successivi del sistema. Il suo compito consiste nel convertire il contesto grezzo osservato nella pagina, rappresentato mediante un dizionario serializzabile in formato JSON, in una descrizione canonica, tipizzata e stabile della superficie interattiva corrente. Il risultato fornisce una base riproducibile sulla quale possono operare l'orchestrazione e i componenti a essa collegati.

3.4.1 Pipeline di elaborazione

A partire dal contesto prodotto dall'estrattore del browser, il DEL esegue in sequenza la canonicalizzazione dei dati, la classificazione euristica della superficie, l'estrazione delle caratteristiche strutturali osservabili e il calcolo di due identificatori complementari, denominati `state_hash` e `surface_signature` . Tale contesto è serializzabile e persistibile e contiene sia la descrizione dettagliata della pagina sia gli elementi sintetici necessari al riuso delle osservazioni e alla correlazione tra stati.

3.4.2 Modello canonico della superficie: normalizzazione

La rappresentazione canonica descrive la superficie osservata senza introdurre interpretazioni relative al suo scopo applicativo. Essa comprende l'URL, il titolo della pagina, un suggerimento testuale sintetico, le intestazioni rilevate, le form, le azioni cliccabili, i campi interattivi e gli eventuali messaggi di errore. Le liste testuali sono

normalizzate eliminando gli elementi vuoti e gli spazi superflui e limitando la lunghezza dei contenuti, così da ridurre il rumore proveniente dal DOM e mantenere il contesto entro dimensioni controllate.

Per ciascun campo vengono conservati gli attributi strutturali, tra cui il tag HTML, il tipo, la visibilità, l'obbligatorietà e lo stato di abilitazione, insieme ai segnali semantici locali, quali l'etichetta, il placeholder, il testo circostante e i metadati della form di appartenenza. Ogni campo riceve inoltre un riferimento canonico costruito a partire dai segnali più stabili disponibili, mentre i principali elementi descrittivi vengono aggregati in una vista testuale deterministica impiegata nella classificazione e nella correlazione locale.

Le azioni cliccabili comprendono pulsanti e link normalizzati come azioni rilevanti. Per ogni azione vengono mantenute le informazioni necessarie a descriverne il ruolo e lo stato, mentre la rappresentazione delle form raccoglie i metadati che ne definiscono il comportamento e i controlli di invio associati.

3.4.3 Caratterizzazione della superficie

La superficie canonica è caratterizzata mediante una categoria euristica e un insieme di proprietà numeriche, booleane e lessicali. La classificazione utilizza il testo normalizzato precedentemente, ottenuto combinando le informazioni della pagina, dei campi, delle azioni e delle form. Le categorie vengono valutate secondo un ordine deterministico e viene assegnata la prima per la quale sono rilevate le parole chiave corrispondenti; in assenza di una corrispondenza, la superficie viene considerata una generica interfaccia di inserimento dati.

La classificazione introduce un vocabolario coerente per indicare famiglie comuni di interfacce, ma non sostituisce un'interpretazione più avanzata e non determina autonomamente la strategia di test. Le caratteristiche estratte sono limitate a proprietà direttamente osservabili, come il numero e il tipo dei campi, la presenza di form, pulsanti di invio o navigazione e vincoli di validazione lato client. Non comprendono candidati di attacco, ordinamenti o valutazioni di rischio.

3.4.4 Identità strutturale e identità logica

La distinzione tra `state_hash` e `surface_signature` risponde alla necessità di rappresentare la superficie a due diversi livelli di astrazione. Le applicazioni moderne modificano frequentemente il DOM mediante classi dinamiche, identificatori generati, notifiche, contatori e messaggi transitori; calcolare un unico identificatore sull'intero HTML renderebbe pertanto ogni variazione sufficiente a distinguere la pagina da uno stato già osservato, compromettendo sia il riuso della cache sia la correlazione storica. Il DEL adotta quindi due identificatori complementari: il primo descrive la struttura interattiva corrente, mentre il secondo rappresenta il ruolo logico della superficie nel tempo.

- Lo `state_hash` stabilisce se la configurazione interattiva corrente coincida con una già elaborata. È sensibile alla presenza dei campi, ai relativi identificatori e vincoli e allo stato dei controlli, poiché questi elementi determinano il modo in cui la pagina può essere utilizzata. Esclude invece i valori digitati e i contenuti testuali liberi o transitori: una form che visualizza una nuova notifica senza modificare i propri controlli conserva quindi la stessa identità strutturale. Questo identificatore opera come un sensore tattico a breve termine, utile per riconoscere stati già visitati, ottenere un *cache hit*, riutilizzare i risultati disponibili ed evitare cicli di esplorazione ridondanti.

- La `surface_signature` descrive invece la funzione svolta dall'interfaccia combinando la categoria della superficie con i ruoli dei campi, delle azioni e delle form. Il suo livello di astrazione consente di correlare nel tempo superfici logicamente equivalenti e di recuperare informazioni utili per il retest tra versioni, anche quando sono intervenute modifiche al CSS, alla disposizione dell'interfaccia o ad alcuni dettagli del DOM. Una form di autenticazione aggiornata può così essere ricondotta a un record precedente, dal quale recuperare un candidato di attacco o le evidenze già raccolte. La firma svolge pertanto il ruolo di memoria strategica a lungo termine, in contrapposizione all'identità strutturale e immediata fornita dallo `state_hash` .

Questo comportamento non deve tuttavia essere interpretato come una misura approssimata di similarità semantica. La `surface_signature` è deterministica e coincide soltanto quando risulta uguale il payload normalizzato impiegato nel calcolo. Poiché tale payload comprende anche segnali testuali esatti, modifiche come il passaggio da "Sign in" a "Accedi" o da "Email address" a "Indirizzo e-mail" possono produrre una firma differente, nonostante la funzione concettuale della pagina rimanga invariata. La scelta è intenzionalmente conservativa: permette di tollerare alcune variazioni strutturali senza introdurre correlazioni troppo lasche quando i segnali disponibili non sono sufficienti.

3.4.5 Differenze nel dettaglio

Come riassunto nella Tabella 3.1, le differenze si articolano su obiettivi, stabilità e dati analizzati.

3.4.6 Riuso dei risultati e strategia di retest

Un singolo identificatore non sarebbe adeguato a entrambi gli scopi. Un'impronta sensibile alla struttura è adatta al riconoscimento di uno stato già osservato e alla ripetizione delle stesse azioni, ma non alla correlazione di una funzionalità dopo una modifica dell'interfaccia. Viceversa, una firma orientata ai segnali logici permette di recuperare risultati storici pertinenti, ma non garantisce da sola che locator, valori e piano siano ancora applicabili senza adattamenti.

Tabella 3.1: Confronto tra State Hash e Surface Signature

Caratteristica	State Hash	Surface Signature
Obiettivo	Riconoscere lo stato attuale (Cache hit).	Tracciamento nel tempo (Retest, cross-version).
Stabilità	Bassa nel tempo. Cambia se cambiano ID, regole di validazione (minlength) o attributi strutturali.	Alta nel tempo. Sopravvive a modifiche del DOM o refactoring dell'interfaccia.
Cosa cattura	L'esatta configurazione degli input (ID, name, limiti di validazione, stato disabilitato/visibile).	Il "ruolo" degli elementi (tipo di superficie, testo semantico, azioni principali).
Cosa ignora	Testi descrittivi volatili (label, placeholder, notifiche, testi dei bottoni).	Dettagli tecnici specifici del DOM (ID, test-id, visibilità attuale, validazioni strette).

Quando una superficie possiede sia la stessa `surface_signature` sia lo stesso `state_hash` di un tentativo storico confermato, può essere considerata compatibile con un replay esatto. Se coincide soltanto la `surface_signature` , il record precedente rimane utile per un retest logico, nel quale il candidato o il finding vengono recuperati adattando però l'interazione alla struttura corrente. La versione dell'applicazione viene mantenuta separata dai due identificatori, così da confrontare superfici osservate in release differenti senza rendere automaticamente diversi gli hash.

3.5 Browser Playwright

Il Browser Layer costituisce il livello di interazione diretta con l'applicazione web e rende possibile l'esplorazione delle interfacce dinamiche, l'osservazione dello stato della pagina e l'esecuzione controllata delle azioni pianificate dal sistema. Basato su Playwright, esso agisce come braccio operativo del sistema. Questa astrazione permette di interagire con l'applicazione in modo analogo a un utente reale e supporta sia le pagine tradizionali sia quelle che costruiscono o modificano dinamicamente il DOM lato client.

Il layer rappresenta il confine tra il sistema di orchestrazione e l'applicazione target: riceve le azioni astratte definite dal livello decisionale, le traduce in interazioni concrete con il browser e restituisce lo stato della pagina insieme alle evidenze prodotte durante il tentativo.

Il contesto acquisito viene poi elaborato dal Deterministic Extraction Layer, che lo converte in una rappresentazione canonica della superficie. Gli esiti dell'esecuzione, le

transizioni rilevate e le evidenze di rete alimentano invece i componenti di valutazione, memoria e replanning.

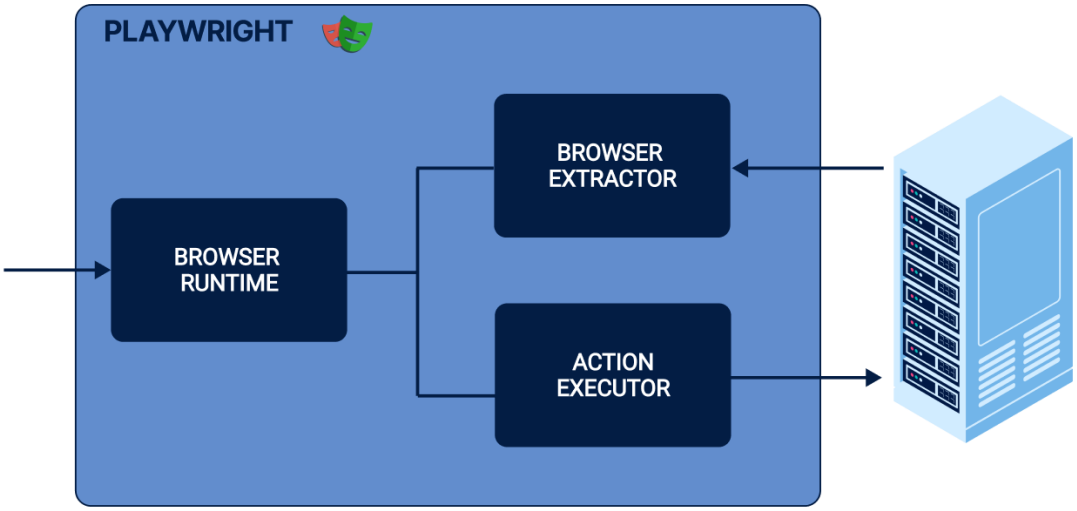


Figura 3.11: Struttura del Browser Layer basato su Playwright.

Il Browser Layer è articolato in tre componenti: il primo gestisce il contesto di esecuzione, il secondo legge il DOM e costruisce una rappresentazione della pagina, mentre il terzo applica le azioni pianificate e invia le form.

3.5.1 Gestione del contesto browser: Browser Runtime

Il runtime browser inizializza e gestisce il ciclo di vita di Playwright, del browser Chromium, del contesto di navigazione e della pagina corrente. Il contesto mantiene gli elementi necessari alla continuità della sessione, quali cookie, local storage e header aggiuntivi.

La configurazione del browser include il proxy verso OWASP ZAP. In questo modo le richieste prodotte durante la navigazione e l'interazione con l'applicazione vengono inoltrate attraverso il proxy di analisi, senza modificare il comportamento generale del browser. La potenza di questo approccio resta nel fatto che è possibile mantenere ZAP come motore principale e vedere direttamente li i risultati a runtime della scansione.

Durante la navigazione il runtime gestisce anche i dialog JavaScript. Come descritto precedentemente nella fase di 3.2, i dialog vengono intercettati e chiusi per evitare che blocchino l'esecuzione automatizzata o falsino l'esito di un tentativo.

3.5.2 Navigazione e osservazione

Il browser esegue la navigazione verso gli URL individuati durante il processo di discover. Dopo ogni navigazione, la pagina corrente rappresenta la sorgente osservabile

dalla quale vengono estratti campi, form, pulsanti, testi, errori e altri segnali utili alla descrizione della superficie interattiva.

Il Browser Layer può inoltre raccogliere le richieste e le risposte correlate a un singolo tentativo. La raccolta viene avviata prima dell'esecuzione dell'azione e terminata dopo un breve intervallo di stabilizzazione. Il risultato è una sintesi delle transazioni osservate, utilizzabile come evidenza per valutare l'esito dell'interazione.

La raccolta è associata a uno specifico identificatore di tentativo. Questa scelta consente di separare le richieste prodotte da azioni differenti e di collegare le evidenze di rete al piano che le ha generate.

3.5.3 Esecuzione delle azioni: Action Executor

L'esecuzione concreta delle azioni sul DOM è affidata al componente `ActionExecutor` . Esso riceve un piano contenente i campi da compilare, i valori da applicare e l'eventuale necessità di inviare una form.

La risoluzione degli elementi della pagina privilegia segnali semantici e identificatori specifici. I selettori CSS più generici vengono utilizzati soltanto come fallback. Questo approccio riduce il rischio di interagire con elementi errati in presenza di componenti duplicati, template nascosti o versioni responsive della stessa interfaccia.

L'esecutore applica i valori rispettando la tipologia del controllo. Sono gestiti in modo distinto campi testuali, textarea, select, checkbox, radio button e file input. I controlli non compilabili, quali campi nascosti, pulsanti o input di reset, non vengono trattati come campi di inserimento.

L'attivazione del submit segue una strategia conservativa. Quando un campo appartiene a una form, il sistema cerca il controllo di invio esclusivamente all'interno della form proprietaria. In questo modo vengono evitati click su pulsanti non correlati presenti nella stessa pagina, quali controlli di navigazione, filtri, paginazione o azioni di altre componenti.

Nel caso di widget privi di un contenitore `form` , come alcune barre di ricerca implementate in applicazioni single-page, il sistema può tentare l'attivazione tramite il tasto `Enter` sul campo appena compilato. Se esiste una form reale ma non è disponibile un pulsante di invio sicuro, l'esecutore può utilizzare il submit nativo della form, mantenendo la validazione HTML prevista dalla pagina.

Dopo l'attivazione del submit, il Browser Layer attende una possibile transizione della pagina senza assumere necessariamente una navigazione completa. Questa scelta è necessaria perché un'applicazione moderna può aggiornare il proprio stato tramite richieste asincrone senza modificare l'URL o senza raggiungere uno stato di rete completamente inattivo.

3.5.4 Estrazione del contesto: Browser Extractor

Prima di delegare la decisione al modello linguistico, il browser costruisce una vista strutturata della pagina. Restituisce un oggetto JSON-friendly composto da metadati di

pagina, elenco dei campi interattivi, descrizione delle form, bottoni visibili, headings e messaggi di errore.

Per ciascun campo `input`, `select` o `textarea` vengono raccolti tag, tipo, label esplicita o implicita, placeholder, stato di visibilità, attributi, vincoli (required, pattern, min/max length), testo di contesto e, nel caso di `select`, l'elenco delle option disponibili. Ogni campo porta con sé metadati sulla form di appartenenza: identificativo, metodo, action e testo interno, oltre ai bottoni di submit associati.

La superficie estratta include anche un insieme limitato di bottoni e link visibili, con testo, attributi e URL associati. I messaggi di errore e gli elementi con ruolo `alert` vengono aggregati per rappresentare gli stati di validazione e il feedback lato client; gli headings principali forniscono invece il contesto semantico della sezione corrente. Il risultato è una rappresentazione deterministica e compatta della pagina, utilizzabile per decidere quali campi compilare e quali azioni tentare.

3.6 LLM

Nel progetto, gli LLM hanno il compito di riprodurre il processo decisionale di un penetration tester. La loro introduzione rappresenta il delta evolutivo dell'architettura, poiché consente di superare il funzionamento tipico degli strumenti DAST, basato prevalentemente su azioni predefinite e deterministiche, e di adottare invece un approccio dinamico, capace di interpretare il contesto e pianificare le operazioni successive.

Affidare l'intero processo a un solo modello, governato da un unico *system prompt*, avrebbe tuttavia comportato una complessità eccessiva e una scarsa separazione delle responsabilità. È stata quindi adottata un'architettura composta da più componenti basati su LLM, ciascuno guidato da istruzioni specifiche e specializzato in una determinata area di competenza. La Figura 3.12 mostra i diversi ruoli previsti.



Figura 3.12: Architettura dei componenti basati su LLM e suddivisione delle relative responsabilità e ruoli.

3.6.1 Base Client

Come primo passo, si è scelto di definire un client di base, dal quale ereditano tutti i client specifici. Questo componente raccoglie le funzionalità comuni necessarie per comunicare con i modelli: la configurazione dell'endpoint e del modello, l'autenticazione, l'invio delle richieste, l'applicazione dei vincoli definiti mediante JSON Schema, il parsing delle risposte e la gestione dei tentativi successivi, oltre alle funzioni di logging e

alla raccolta delle metriche. La separazione tra il client di base e i client specifici consente quindi di evitare la duplicazione della logica trasversale e di mantenere distinti i diversi livelli dell'architettura.

Grazie a questa organizzazione, ciascun client specifico deve occuparsi soltanto degli aspetti legati al proprio ruolo: la definizione del relativo prompt, la costruzione del payload e la specifica dello schema di output atteso. Tutte le operazioni comuni vengono invece fornite tramite l'ereditarietà dal `BaseLlmClient`.

Un ulteriore aspetto considerato nella progettazione riguarda la possibilità che il modello produca risposte non conformi alle istruzioni o allo schema richiesto. Durante i test è emerso che questo comportamento si verificava soprattutto con i modelli più veloci e caratterizzati da un numero inferiore di parametri. Per gestire tali casi è stato introdotto un meccanismo di retry specifico, distinto sia dal fallimento di un attacco sia dagli errori operativi, come quelli dovuti alla connettività con la pagina. Il nuovo tentativo consiste esclusivamente nel reinserimento dello schema da rispettare, accompagnato da un prompt rafforzativo. Negli esperimenti svolti, questa procedura ha consentito al modello di produrre correttamente la risposta al secondo tentativo in tutti i casi verificatosi. Lo schema generale delle interazioni tra i vari LLM in fase di attacco è mostrato in figura 3.13, descritto poi meglio nelle prossime sottosezioni.

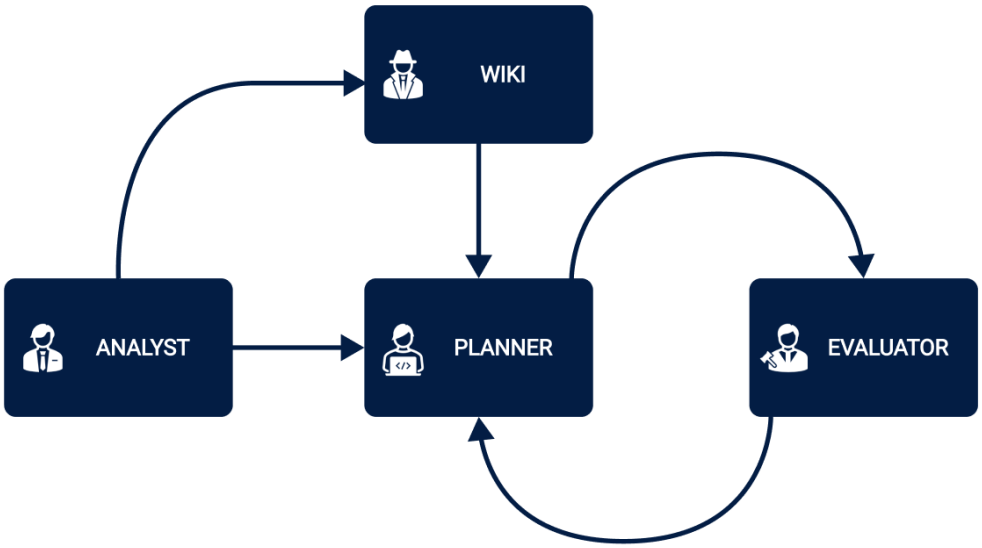


Figura 3.13: Flusso di interazione tra i vari LLM in fase di attacco.

3.6.2 Analyst

L'Analyst ha il compito di esaminare lo schema standardizzato prodotto dal DEL e di individuare i candidati di attacco più appropriati per la superficie rilevata. Il numero massimo di candidati generati è modificabile dal file di config. La selezione viene

effettuata considerando gli elementi e i campi presenti nella pagina: ad esempio, la presenza di un modulo di autenticazione può suggerire la verifica di vulnerabilità di tipo SQL injection, mentre un campo di ricerca può costituire una superficie adatta a test di tipo XSS.

Per ciascun candidato, l'Analyst specifica la famiglia di attacco, propone indicazioni preliminari per la costruzione dei payload sulla base della conoscenza del modello e genera i termini da utilizzare per il recupero delle informazioni pertinenti dalla LLM Wiki. Il componente non definisce l'azione esecutiva completa, ma produce un insieme ordinato di ipotesi e informazioni di supporto che vengono successivamente affidate al Planner.

3.6.3 Planner

Il Planner trasforma ciascun candidato di attacco in un piano eseguibile. In particolare, stabilisce il payload da utilizzare, individua l'elemento della pagina al quale applicarlo mediante il relativo riferimento e fornisce a Playwright le istruzioni necessarie per eseguire l'azione nel browser.

Per ogni famiglia di attacco possono essere effettuati più tentativi, entro il limite definito dal budget configurato. Il Planner dispone di una memoria delle esecuzioni precedenti, grazie alla quale evita di riproporre gli stessi payload e formula, a ogni iterazione, un piano differente ma coerente con la famiglia in esame. Una volta esaurito il budget disponibile, il processo prosegue con la famiglia di attacco successiva.

Il Planner può inoltre accedere all'Entity Store, che conserva informazioni raccolte nel corso della scansione, quali indirizzi e-mail, token e altri dati potenzialmente sensibili. Tali entità possono essere recuperate e integrate nei payload o nelle azioni successive, consentendo di elaborare attacchi che tengano conto delle informazioni emerse durante l'esplorazione dell'applicazione.

3.6.4 Evaluator, LLM as a judge

Per la progettazione dell'Evaluator è stata adottata la tecnica *LLM as a judge*, descritta nel lavoro di riferimento Gu *et al.* [2024]. L'applicazione rigorosa di questa metodologia costituisce un tema sufficientemente ampio da poter essere argomento di tesi dedicato; nell'ambito del presente progetto non sono state pertanto recepite tutte le raccomandazioni proposte in letteratura, ma sono stati adottati gli accorgimenti ritenuti più rilevanti per ottenere una valutazione robusta e riproducibile.

In primo luogo, il modello non riceve una domanda generica, quale "Il test ha avuto successo?", ma una rubrica specifica per il compito come consigliato dal paper analizzato. Tale rubrica distingue tre possibili esiti, *confirmed*, *suspected* e *inconclusive*, e definisce i criteri necessari per assegnare ciascuna categoria. La valutazione viene inoltre articolata in più dimensioni, tra cui la validità dell'esecuzione, la presenza di segnali tecnici, l'impatto osservabile e la qualità delle evidenze. Questa scomposizione evita di affidare la decisione a un giudizio generico e contribuisce a renderla più stabile.

Per consentire il confronto tra lo stato precedente e quello successivo all'esecuzione del test, l'Evaluator riceve le seguenti informazioni:

- `candidate` , ossia l'ipotesi sottoposta a verifica;
- `before_state` , che descrive in modo deterministico la superficie del browser prima dell'esecuzione;
- `after_state` , che descrive la medesima superficie dopo l'esecuzione;
- `observed_effects` , contenente le osservazioni grezze prodotte dal browser;
- `evidence_bundle` , che raccoglie i fatti osservabili in forma normalizzata;
- `deterministic_assessment` , che fornisce indicazioni di valutazione specifiche per il dominio.

Il giudizio non si basa quindi esclusivamente su descrizioni testuali, ma su evidenze provenienti dal browser, dallo stato della superficie, dal traffico di rete e dagli eventuali errori o dialog restituiti dall'applicazione. Il modello è istruito a considerare soltanto i fatti disponibili, senza inferire o inventare prove mancanti. Alla componente generativa sono inoltre affiancate regole deterministiche, che incorporano conoscenza specifica del dominio e contribuiscono a rafforzare o correggere la decisione.

Per garantire la tracciabilità del risultato, l'output comprende obbligatoriamente una motivazione, espressa nel campo `reason` , e un livello di confidenza associato alla classificazione. In questo modo, oltre all'esito sintetico, è possibile comprendere quali evidenze abbiano condotto alla decisione e con quale grado di certezza essa sia stata formulata, pratica comune riportata anche'essa nel paper.

Come possibile vedere nel capitolo dedicato ai risultati, questo approccio ha mostrato complessivamente un buon comportamento, pur producendo occasionalmente falsi positivi legati alla natura dell'applicazione analizzata. È inoltre necessario considerare che non tutti gli effetti di un attacco sono osservabili dall'Evaluator. Una SQL injection eseguita su un modulo di autenticazione, ad esempio, può essere confermata quando determina l'accesso a un'area riservata, poiché il cambiamento è direttamente rilevabile nello stato della pagina. Al contrario, un attacco che provochi esclusivamente un errore interno del database, senza restituire segnali nell'interfaccia o nella risposta, potrebbe non essere riconosciuto pur avendo prodotto un effetto sul sistema. Tale limite è parzialmente compensato dall'integrazione con ZAP: ogni richiesta transita infatti attraverso il proxy, le cui regole possono rilevare e segnalare ulteriori comportamenti sospetti, ma in ogni caso non risolve del tutto il problema.

3.6.5 Wiki

A differenza degli altri client basati su LLM, il componente dedicato alla consultazione della Wiki non deve produrre un output conforme a uno specifico JSON Schema.

Le informazioni recuperate vengono infatti sintetizzate mediante un prompt esplicito e il testo risultante viene fornito direttamente al Planner come materiale di supporto.

La Wiki è stata precedentemente costruita a partire da documenti grezzi descritti in sezione 4.6.2, successivamente rielaborati e adattati agli obiettivi della tesi. Il suo contenuto comprende informazioni sulle diverse famiglie di attacco, sui contesti nei quali possono essere impiegate, sui possibili payload e sulle relative contromisure. Per ciascun candidato individuato dall'Analyst, il recupero delle informazioni viene eseguito una sola volta mediante l'algoritmo BM25; il funzionamento di questo meccanismo sarà descritto nelle sezioni successive.

Le indicazioni ottenute dalla Wiki non costituiscono tuttavia vincoli per il Planner, che può elaborare autonomamente payload differenti qualora li ritenga più appropriati al contesto osservato. La progettazione e il processo di costruzione della Wiki saranno approfonditi nella Sezione 3.7.

3.6.6 Navigator

Il Navigator è l'unico client basato su LLM impiegato esclusivamente durante la fase di *discovery*; diversamente dall'Analyst, dal Planner e dall'Evaluator, non prende quindi parte alla fase di attacco. Il suo obiettivo consiste nel guidare la navigazione dell'applicazione, individuando l'azione più appropriata per progredire dallo stato corrente verso le fasi successive del flusso e raggiungere superfici non ancora esplorate.

Il componente non dispone di un valutatore dedicato. L'esito dell'azione proposta viene verificato direttamente dalla pipeline di *discovery*, che confronta lo stato precedente con quello ottenuto dopo l'esecuzione per determinare se sia avvenuta una progressione oppure se sia stato raggiunto un nodo obiettivo. In base a tale verifica e al profilo target definito, la pipeline decide se proseguire l'esplorazione o considerare concluso il percorso corrente.

3.7 LLM Wiki come knowledge base persistente

La scelta di introdurre una *LLM Wiki* deriva dall'idea di trattare la conoscenza non come una raccolta statica di chunk da interrogare direttamente, ma come un artefatto persistente e mantenuto incrementalmente. Karpathy ha proposto un paradigma analogo per basi di conoscenza personali: le fonti grezze vengono mantenute separate, mentre un LLM costruisce e aggiorna una wiki Markdown interconnessa utilizzabile nelle interrogazioni successive.

La suddivisione nei tre paradigmi presentati di seguito riprende la classificazione proposta da Simone Rizzo nella sua lezione dedicata all'evoluzione delle knowledge base per sistemi basati su LLM, dal RAG convenzionale all'Agentic File Search fino alla LLM Wiki (Rizzo [2026]).

3.7.1 Tre paradigmi per la conoscenza

RAG convenzionale

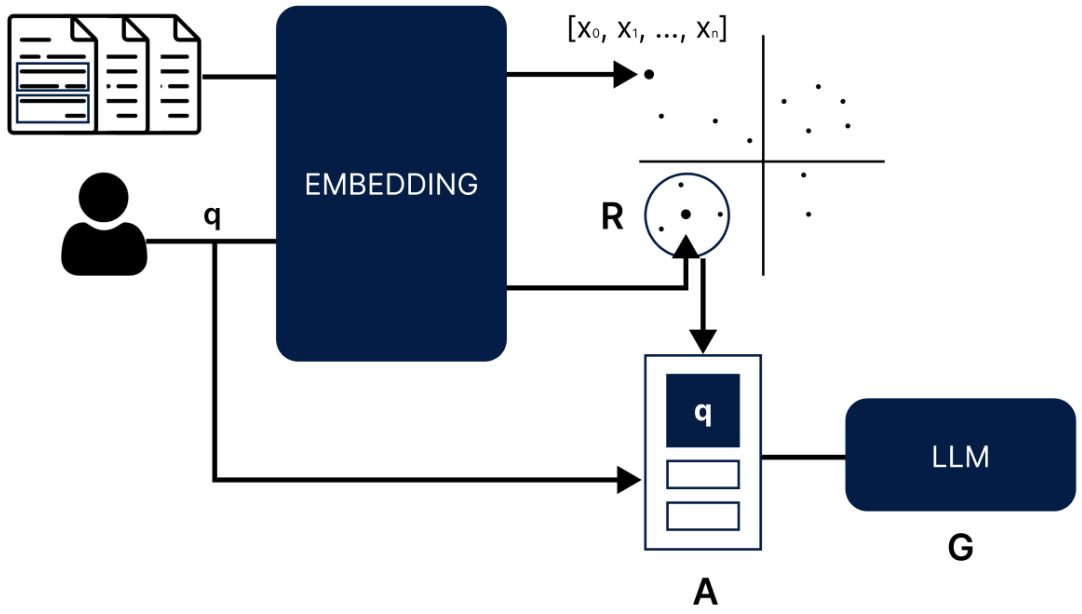


Figura 3.14: Sistema RAG.

La Retrieval-Augmented Generation (RAG) classica indicizza documenti sorgente, li suddivide in chunk e recupera i segmenti più rilevanti al momento della query. Il modello riceve quindi la domanda e un numero limitato di frammenti testuali, dai quali genera la risposta o il piano successivo.

Sia q una query e sia $D = \{d_1, \ldots, d_N\}$ l'insieme dei chunk. Un retriever assegna a ciascun chunk un punteggio di rilevanza:

$$\text{score}(q, d_i) = \lambda \cdot \text{semantic}(q, d_i) + (1 - \lambda) \cdot \text{lexical}(q, d_i), \qquad 0 \leq \lambda \leq 1.$$

Il termine semantico può essere calcolato mediante similarità coseno tra embedding:

$$\text{semantic}(q, d_i) = \frac{\mathbf{e}_q \cdot \mathbf{e}_{d_i}}{\|\mathbf{e}_q\|_2 \, \|\mathbf{e}_{d_i}\|_2},$$

dove $\mathbf{e}_q$ ed $\mathbf{e}_{d_i}$ rappresentano, rispettivamente, gli embedding della query e del documento. Il termine lessicale può invece essere espresso mediante BM25:

$$\text{BM25}(q, d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t, d) \cdot (k_1 + 1)}{f(t, d) + k_1 \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}.$$

In tale formula, $f(t,d)$ è la frequenza del termine t nel documento d, $|d|$ è la lunghezza del documento, avgdl è la lunghezza media dei documenti e k_1 e b sono parametri di normalizzazione.

Il principale vantaggio del RAG è la semplicità: l'aggiunta di nuovi documenti richiede principalmente indicizzazione e chunking. Tuttavia, nel dominio DAST, il RAG convenzionale presenta tre limiti. Primo, il modello riceve frammenti grezzi che possono contenere ridondanze, sezioni irrilevanti o istruzioni non immediatamente operative. Secondo, la sintesi della conoscenza viene ricalcolata durante ogni query, aumentando il costo inferenziale. Terzo, la relazione tra tecniche, contromisure, strumenti e casi d'uso rimane implicita nei chunk anziché essere rappresentata esplicitamente.

Sistema agentico con strumenti e memoria

Un secondo paradigma consiste nell'impiegare un agente LLM che opera in un ciclo iterativo di ragionamento, uso di strumenti, osservazione e revisione (figura 3.15). In forma astratta, al passo t, l'agente riceve uno stato s_t, genera un'azione a_t, utilizza eventualmente un tool e osserva il nuovo stato:

$$a_t = \pi_\theta(s_t, m_t),$$

$$o_{t+1} = \text{Tool}(a_t),$$

$$s_{t+1} = \text{Update}(s_t, a_t, o_{t+1}),$$

dove m_t rappresenta la memoria accessibile al passo t e π_θ è la politica indotta dal modello linguistico.

Questo paradigma è potente perché permette adattamento dinamico: l'agente può cercare documentazione, interrogare una knowledge base, usare strumenti browser e modificare il piano. Karpathy descrive gli agenti moderni come sistemi "loopy", cioè sistemi nei quali si concatenano ripetutamente chiamate al modello, strumenti e osservazioni, invece di produrre una singola risposta statica.

Tale flessibilità comporta però un costo rilevante. Un agente completamente autonomo deve decidere quando interrogare la memoria, quale tool selezionare, quanti tentativi effettuare, quando interrompere l'esecuzione e come interpretare osservazioni spesso ambigue.

OpenClaw (OpenClaw [2026]) rappresenta un esempio utile di memoria operativa orientata agli agenti. Il sistema persiste la memoria in file Markdown: `MEMORY.md` contiene fatti, preferenze e decisioni durevoli ed è caricato all'avvio della sessione; i file `memory/YYYY-MM-DD.md` conservano invece note giornaliere, osservazioni e contesto di lavoro. La memoria non è uno stato nascosto del modello: ciò che non viene scritto su disco non è disponibile nelle sessioni successive . Il problema attuale che si sta cercando di combattere in ogni sistema agentico è che ad ogni avvio dell'agente molti dei token vengono consumati unicamente a causa del contesto. Ad ogni avvio occorre ricaricare tutto.

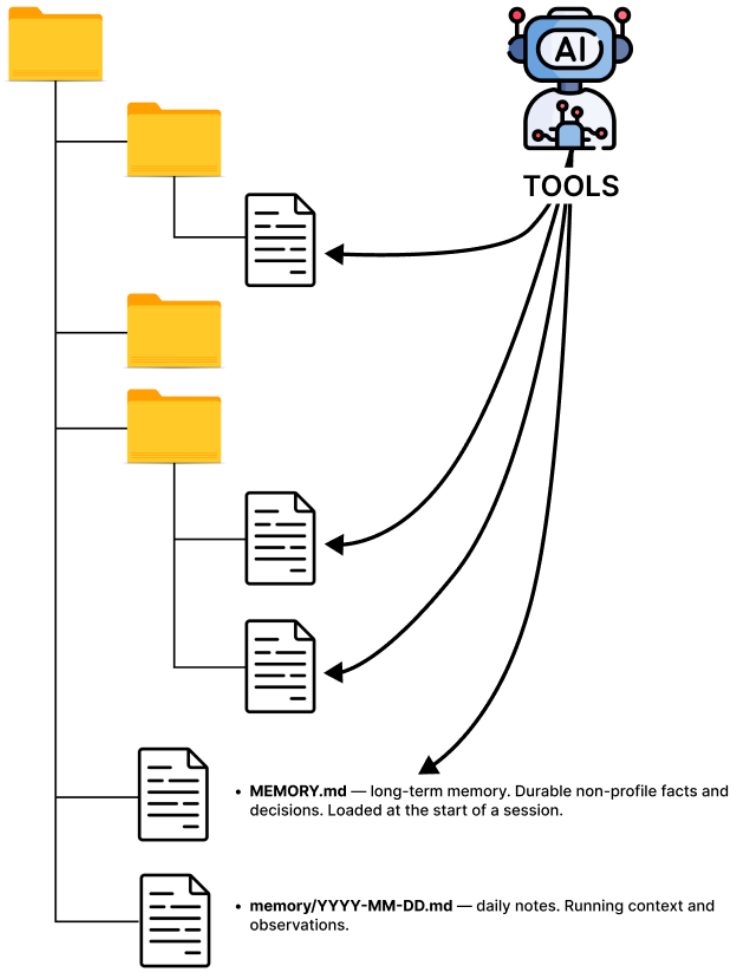


Figura 3.15: Sistema agentico.

LLM Wiki

La LLM Wiki introduce un terzo modello: le fonti grezze vengono elaborate una volta in fase di ingestione e trasformate in una base di conoscenza persistente, strutturata e mantenuta incrementalmente. L'idea è coerente con il modello di knowledge base proposto da Karpathy: una directory `raw/` conserva le fonti immutabili, mentre un LLM produce e aggiorna una wiki Markdown con articoli, concetti, sintesi e riferimenti (Andrej Karpathy [2026]).

A differenza del RAG, la conoscenza non viene soltanto recuperata dal documento grezzo: viene prima "compilata". Il termine compilazione indica una trasformazione da sorgenti eterogenee e verbose a pagine con una struttura stabile, metadati, concetti, mitigazioni, payload e relazioni esplicite. La conoscenza viene dunque elaborata al momento dell'ingestione, non ricostruita integralmente per ogni richiesta.

Formalmente, siano R le fonti raw, W la wiki persistente e C una configurazione di schema, prompt e regole di mantenimento. La fase di ingestione può essere modellata come:

$$W_{t+1} = \text{Compile}(W_t, R_{\text{new}}, C).$$

L'operatore Compile non produce soltanto una copia ridotta della fonte. Esso estrae contenuto rilevante, aggiorna pagine esistenti, crea o aggiorna concept pages, genera summary e mantiene i collegamenti strutturali. In questo modo, l'inferenza ripetuta viene spostata dalla fase di query alla fase di manutenzione della knowledge base.

3.7.2 Confronto tra gli approcci

Dimensione	**RAG convenzionale**	**Agente con memoria e tool**	**LLM Wiki**
Unità di conoscenza	Chunk delle fonti originali	Memorie, osservazioni e risultati di tool	Article, summary e concept pages compilate
Aggiornamento	Re-indicizzazione dei documenti	Scrittura e recupero dinamico di memoria	Ingestione incrementale e merge della wiki
Recupero	Similarity search sui chunk	Decisione autonoma dell'agente con tool di memoria	Retrieval su pagine sintetiche e strutturate
Costo alla query	Medio o alto, perché la sintesi avviene durante la query	Potenzialmente alto, a causa dei cicli agente–tool–osservazione	Ridotto, poiché gran parte della sintesi è stata effettuata in ingestione
Tracciabilità	Legata ai chunk recuperati	Dipendente dalla sequenza di tool call e dalle memorie	Legata a file Markdown persistenti e a fonti raw immutabili
Ripetibilità	Buona nel retrieval, variabile nella generazione	Variabile, per natura adattiva del loop	Alta per le componenti di retrieval e manutenzione deterministica
Uso ideale nel progetto	Recupero documentale generico	Azioni adattive e memoria di esecuzione	Conoscenza operativa DAST stabile e riutilizzabile

Tabella 3.2: Confronto tra RAG, agenti con memoria e LLM Wiki.

Nessuno dei tre paradigmi è universalmente superiore. Il RAG è adatto quando le fonti cambiano frequentemente e non è necessario consolidare conoscenza; l'agente è adatto quando la sequenza di azioni non può essere predeterminata; la LLM Wiki è particolarmente efficace quando il dominio è specialistico, relativamente stabile e richiede conoscenza riutilizzabile in molte query.

I dettagli implementativi più rilevanti verranno forniti nella sezione 4.6.

3.8 Memoria

La memoria costituisce una componente fondamentale dell'architettura di ZAPilot. Come le altre parti del sistema, anche la sua progettazione è stata sviluppata in modo incrementale. La soluzione iniziale prevedeva un'unica memoria, organizzata come un grafo ordinato e progettata per consentire un'esplorazione rapida dello spazio applicativo. Nel corso dello sviluppo, tuttavia, questa rappresentazione si è dimostrata insufficiente: le informazioni necessarie durante un singolo tentativo hanno infatti durata e finalità differenti rispetto a quelle che devono rimanere disponibili per l'intera esecuzione o essere recuperate in esecuzioni successive.

L'architettura è stata quindi articolata in più forme di memoria tra loro complementari. Il Session Tracker conserva il contesto operativo necessario a correggere e ripianificare le azioni nel breve periodo, mentre l'Entity Store raccoglie le entità individuate durante l'intera esecuzione. A queste componenti temporanee si affiancano la rappresentazione persistente del grafo di esplorazione e Redis, utilizzato per conservare lo stato tra esecuzioni e supportare la riproduzione degli attacchi già registrati. Questa separazione permette a ciascuna memoria di rispondere a un'esigenza specifica, evitando di trattare nello stesso modo informazioni caratterizzate da cicli di vita diversi. Le varie implementazioni della memoria mostrate in figura 3.16 verranno spiegate in seguito.

3.8.1 Memoria temporanea

La memoria temporanea comprende il Session Tracker, responsabile di mantenere il contesto dei tentativi eseguiti durante il ciclo corrente. Il componente registra gli URL raggiunti, i valori utilizzati nella compilazione dei form e la traccia delle azioni effettuate. Conserva inoltre gli errori emersi durante un tentativo, trasformandoli in feedback strutturato che può essere impiegato nei cicli successivi di *retry* e *replanning*. In questo modo, un'azione fallita non viene considerata come un evento isolato, ma fornisce informazioni utili per modificare il piano e correggere il tentativo seguente. I valori provati non vengono tuttavia trattati tutti allo stesso modo: il sistema mantiene come confermati soltanto quelli associati a una transizione utile, distinguendo così le informazioni che hanno prodotto un avanzamento da quelle risultate inefficaci.

Il Session Tracker non costituisce una memoria persistente tra esecuzioni diverse. La sua funzione è sostenere il ragionamento operativo nel breve periodo, mantenendo le informazioni necessarie a comprendere ciò che è già stato tentato e le ragioni di un eventuale fallimento. Questa scelta riduce la ripetizione di azioni prive di effetto e consente al sistema di adattare il comportamento sulla base degli esiti osservati durante la navigazione.

Accanto al Session Tracker è stato introdotto l'Entity Store, la cui durata copre invece l'intera esecuzione. Il componente lavora a stretto contatto con il DEL e raccoglie entità

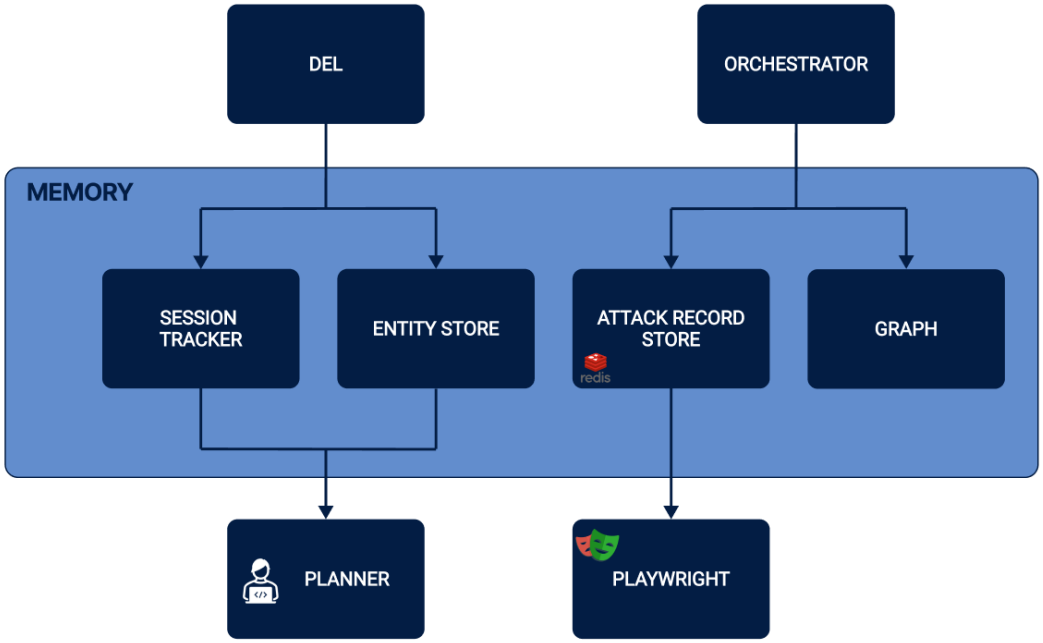


Figura 3.16: Memorie utilizzate.

rilevanti individuate nel corso dell'esplorazione, come indirizzi e-mail e token, rendendole disponibili per le interazioni successive. La necessità di questa memoria è emersa in modo evidente durante le prove su OWASP Juice Shop: navigando nell'applicazione è possibile individuare, all'interno di un commento, l'indirizzo e-mail dell'amministratore. Tale informazione può essere successivamente riutilizzata per formulare un tentativo di *brute force* oppure per costruire una SQL injection nel campo della password. L'esempio mostra perché alcune entità non debbano rimanere confinate alla pagina nella quale sono state rilevate: il loro valore operativo può emergere soltanto in una fase successiva della navigazione o dell'attacco.

3.8.2 Memoria persistente

La memoria persistente svolge una duplice funzione. Da un lato costituisce un artefatto che consente di visualizzare immediatamente lo stato dell'esplorazione; dall'altro rende possibile confrontare versioni differenti della stessa applicazione. La struttura è rappresentata come un grafo disposto visivamente ad albero, nel quale il nodo iniziale occupa il centro e gli URL individuati durante ciascuna esplorazione vengono aggiunti progressivamente attorno ai nodi già presenti. La rappresentazione rende quindi osservabile sia l'espansione della superficie raggiunta sia lo stato dell'analisi associata a ogni URL. Al contrario della prima soluzione precedentemente esposta, questa soluzione circolare è stata preferita alla più comunene sequenziale poichè molte applicazioni hanno

la maggior parte delle pagine raggiungibile da link presenti in home. Di conseguenza ne sarebbe risultato un primo layer con il singolo nodo iniziale e numerosi nodi tutti sul secondo layer, rendendo di fatto poco interpretabile il grafo stesso.

Lo stato dei nodi è comunicato mediante quattro colori. Un nodo grigio corrisponde a una risorsa scoperta ma non ancora esplorata, mentre un nodo verde identifica una risorsa analizzata senza che siano state confermate vulnerabilità. Il giallo segnala invece un verdetto inconclusivo: il sistema ha individuato elementi che suggeriscono la possibilità di un attacco, ma le evidenze raccolte non sono sufficienti a confermarlo oppure il livello di confidenza espresso dall'LLM as a judge (Evaluator) è troppo basso. Il rosso, infine, indica una vulnerabilità confermata e mantiene l'associazione con la relativa tipologia. Questa rappresentazione è strettamente collegata alla persistenza realizzata tramite Redis, che permette di conservare e recuperare lo stato del grafo.

3.8.3 Redis

Redis è impiegato nel progetto con due finalità principali. La prima riguarda la continuità dell'analisi su applicazioni complesse o composte da numerose pagine. L'ampliamento dei parametri di scansione, necessario per coprire un numero maggiore di possibili vulnerabilità, può infatti aumentare considerevolmente la durata dei test. La persistenza dello stato consente, in caso di interruzione, di riprendere l'esecuzione dal punto raggiunto senza analizzare nuovamente i nodi già visitati. Facendo riferimento al grafo descritto in precedenza, il sistema può selezionare i nodi grigi ancora da esaminare e proseguire così in modo incrementale l'analisi dell'applicazione.

La seconda finalità riguarda il confronto tra versioni differenti della stessa applicazione. Quando viene confermata una vulnerabilità, le informazioni relative all'attacco vengono memorizzate in Redis attraverso `AttackRecordStore` . In modalità *replay*, ZAPilot recupera gli attacchi registrati per le versioni precedenti e li esegue nuovamente nelle stesse condizioni. Se la vulnerabilità risulta ancora presente, il nodo continua ad essere rappresentato in rosso; in caso contrario, il sistema prosegue con le ulteriori verifiche previste e il nodo può infine assumere lo stato verde. Il meccanismo rende immediatamente visibile l'esito di una correzione e fornisce un supporto utile alle attività diagnostiche e di regressione. Da sottolineare è come se dovesse cambiare la rappresentazione del DEL in una nuova versione dell'applicazione, anche il nodo sarà trattato come nuovo, e dunque non c'è modo (per come è stato pensato in questo progetto) di ritestare lo stesso attacco su un DOM differente. Questo, come descritto già nella sezione 3.4 è un fattore sia positivo che negativo, in quanto, giustamente, se l'applicazione cambia deve essere ritestata, ma sarebbe anche opportuno legare il cambiamento dovunto ad una precedente vulnerabilità con la vulnerabilità stessa, permettendo di ritestarla. Questo in parte è mitigato dalla classificazione della form nel DEL, che, assumendo sia rimasta la stessa, permetterà di conferire al planner come candidati d'attacco payload molto simili se non identici ai passati, identifincando nuovamente la vulnerabilità.

In questo processo OWASP ZAP rimane il principale motore di scansione e di verifica. Redis non ne sostituisce le funzionalità, ma conserva lo stato e le informazioni

prodotte durante l'orchestrazione di ZAPilot. Tra gli artefatti persistibili rientra anche il report generato da ZAP, che permette di consultare le vulnerabilità emerse durante l'esecuzione e raggiunte attraverso l'intervento di ZAPilot, che oltre a salvarlo sul grafo, alza l'alert direttamente in ZAP.

3.9 Orchestratore

L'Orchestrator costituisce il centro di coordinamento dell'intera pipeline. Insieme al Queue Builder governa il ciclo principale della scansione: il Queue Builder seleziona il nodo da elaborare, mentre l'Orchestrator ne dirige l'analisi, stabilisce la sequenza delle operazioni, invoca i componenti necessari e trasferisce tra essi dati, contesto ed evidenze. Al termine dell'elaborazione restituisce il controllo al Queue Builder, che può aggiornare la coda mediante una nuova fase di discovery e selezionare il nodo successivo.

L'Orchestrator non sostituisce i componenti specializzati e non esegue direttamente le singole attività. Il suo compito è comporle in un processo controllato: inizializza il contesto relativo al nodo, applica i parametri configurati per la scansione, sceglie il flusso operativo, coordina i componenti basati su LLM, governa il Browser Layer fondato su Playwright, raccoglie gli stati prodotti dal DEL e registra gli esiti nella memoria e nel grafo persistente. In questo modo mantiene una visione complessiva dell'esecuzione e garantisce che ogni componente riceva gli input corretti nel momento previsto.

La configurazione determina, tra gli altri aspetti, la modalità con cui deve essere elaborato il nodo, il numero massimo di candidati di attacco e il budget assegnato a ciascuna famiglia di vulnerabilità. Sulla base di tali vincoli, l'Orchestrator controlla le iterazioni, decide quando proseguire con un nuovo tentativo, quando passare al candidato successivo e quando interrompere anticipatamente il flusso. Le due modalità principali da esso coordinate sono la modalità Attack e la modalità Discovery.

3.9.1 Modalità Attack

In modalità Attack, l'Orchestrator coordina il processo di individuazione e verifica delle vulnerabilità sulla superficie associata al nodo corrente. Come mostrato nella Figura 3.17, il flusso integra componenti deterministiche, ruoli basati su LLM e strumenti di interazione con il browser. L'Orchestrator mantiene la continuità tra le diverse fasi, conserva il contesto dei tentativi e raccoglie le evidenze necessarie a valutarne l'esito.

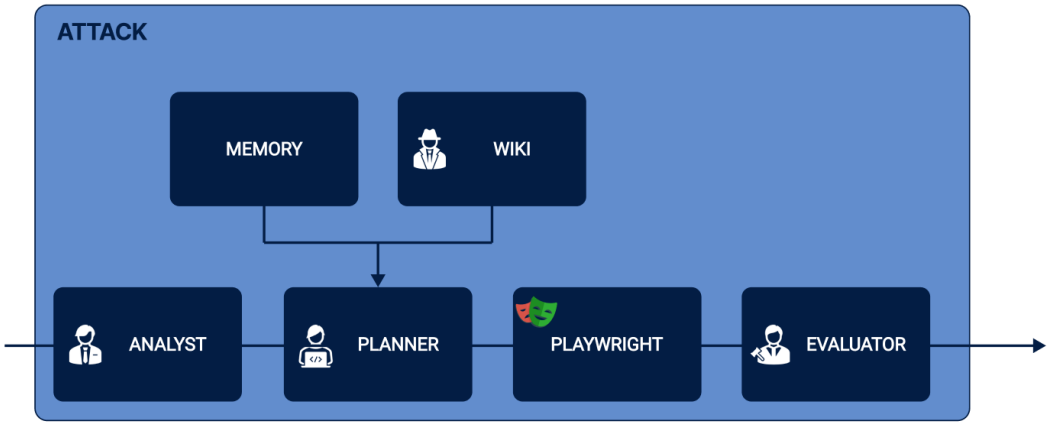


Figura 3.17: Flusso coordinato dall'Orchestrator in modalità Attack.

Il processo inizia con l'acquisizione, tramite il DEL, della rappresentazione deterministica della superficie corrente. Prima di avviare una nuova analisi, l'Orchestrator consulta la memoria persistente per verificare se siano già disponibili risultati o attacchi confermati da riutilizzare. Se non è possibile concludere il flusso attraverso la cache o il replay, viene eseguito il ciclo completo composto da Analyst, Wiki, Planner, Browser Layer ed Evaluator.

Cache hit e replay di attacchi confermati

La memoria persistente Redis, descritta nella Sezione 3.8, viene consultata prima di invocare i componenti basati su LLM. L'obiettivo è duplice: evitare di ripetere analisi equivalenti su superfici già osservate e supportare il regression testing delle vulnerabilità confermate nelle esecuzioni precedenti.

Ogni superficie è associata a una firma deterministica, denominata *surface signature* e descritta nella Sezione 3.4. La firma permette di confrontare superfici osservate in momenti differenti senza dipendere esclusivamente dall'URL, che può cambiare pur mantenendo una struttura di interazione equivalente. Le variazioni non significative che non modificano la rappresentazione prodotta dal DEL lasciano invariata la firma e consentono quindi il riuso delle informazioni memorizzate.

Quando è già disponibile un risultato per la stessa superficie, si verifica un *cache hit*. In questo caso l'Orchestrator può saltare il nodo e continuare ad attaccare altre parti dell'applicazione. Il relativo flusso è illustrato nella Figura 3.18.

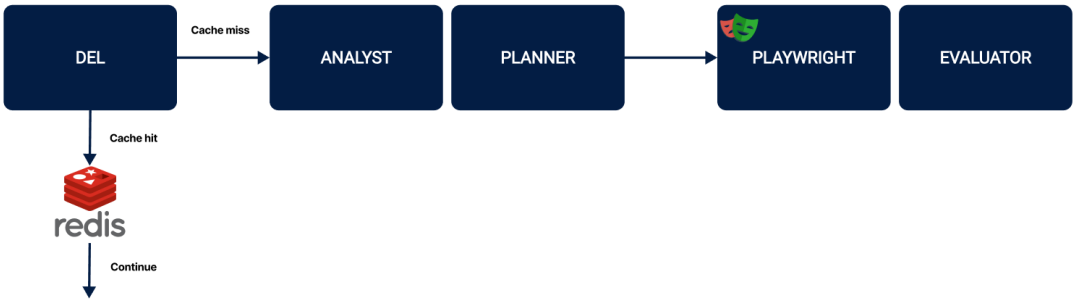


Figura 3.18: Flusso di gestione del cache hit.

Se la memoria contiene invece un attacco precedentemente confermato come efficace, l'Orchestrator può attivare l'*exact replay*, il cui flusso è rappresentato nella Figura 3.19. Analyst e Planner non vengono invocati nuovamente: il sistema recupera il piano già validato, compresi i valori utilizzati nei campi e l'azione finale di invio, e lo affida direttamente al Browser Layer. L'Evaluator analizza quindi le nuove evidenze per stabilire se la vulnerabilità sia ancora riproducibile nella versione corrente.

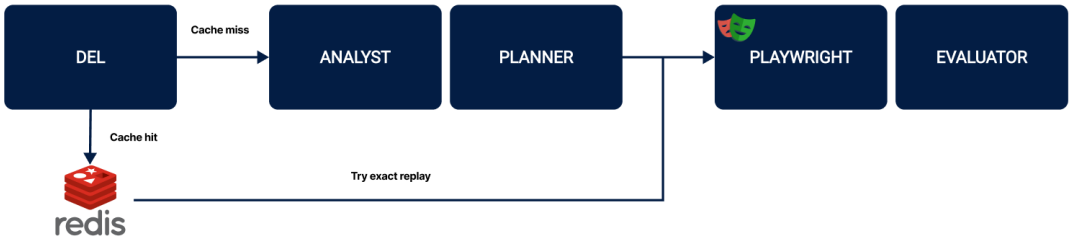


Figura 3.19: Flusso di exact replay di un attacco confermato.

Il replay non costituisce una conferma automatica, poiché l'attacco storico viene sempre rieseguito nell'ambiente corrente. Se l'Evaluator conferma nuovamente l'esito, l'Orchestrator registra la vulnerabilità sul nodo e interrompe anticipatamente il ciclo, riducendo il numero di invocazioni LLM necessarie. Se invece il replay fallisce, la vulnerabilità potrebbe essere stata corretta oppure potrebbero essere cambiate le condizioni applicative. Il sistema prosegue quindi con il flusso ordinario e, se necessario, richiede al Planner una strategia aggiornata. I candidati già sottoposti a replay vengono esclusi dai tentativi successivi della stessa esecuzione per evitare duplicazioni.

Analisi ed esecuzione dei candidati

Quando non è disponibile un risultato riutilizzabile, la superficie prodotta dal DEL viene consegnata all'Analyst. Questo componente esamina gli elementi interattivi e genera un insieme ordinato di candidati compatibili con il contesto osservato. L'Analyst viene invocato una sola volta per ogni superficie e il numero di candidati proposti è

limitato da un parametro configurabile. Il valore rappresenta un massimo, non un obiettivo obbligatorio: se le evidenze giustificano un numero inferiore di ipotesi, l'Analyst può restituire meno candidati. Per esempio, con un limite pari a uno, per un modulo di autenticazione potrebbe essere proposta soltanto la famiglia delle *weak credential*; aumentando il limite possono essere considerate anche altre famiglie, come le injection.

Per ogni candidato, l'Orchestrator richiede alla Wiki un contesto tecnico sintetico contenente informazioni sulla vulnerabilità e sui possibili payload. Queste indicazioni arricchiscono il contesto fornito al Planner, ma non sono vincolanti: il Planner può adattarle o formulare una strategia differente sulla base della superficie, della memoria dei tentativi precedenti e delle capacità effettive dell'esecutore.

L'Orchestrator valida l'output strutturato del Planner e affida il piano al Browser Layer basato su Playwright. Quest'ultimo compila i campi indicati, applica i valori o i payload previsti ed esegue l'eventuale azione di invio. Durante questa fase l'Orchestrator associa l'esecuzione al candidato corrente, mantiene lo stato della sessione e coordina la raccolta delle evidenze.

Il DEL acquisisce la superficie sia prima sia dopo l'azione. Alle differenze tra i due stati vengono aggiunte le osservazioni prodotte durante l'esecuzione, tra cui variazioni dell'URL o della route, modifiche del contenuto visibile, dialog del browser, errori JavaScript, cookie e traffico HTTP osservato tramite ZAP. L'Orchestrator aggrega queste informazioni e le consegna all'Evaluator, che opera secondo il paradigma *as a judge* e stabilisce se il candidato sia confermato, soltanto sospetto oppure non valutabile in modo affidabile.

Ogni famiglia di vulnerabilità dispone di un budget configurabile, espresso come numero massimo di tentativi. Se un tentativo non produce una conferma e rimane budget disponibile, l'Orchestrator mantiene attiva la stessa famiglia, fornisce al Planner il feedback e la memoria delle esecuzioni precedenti e richiede un piano differente. Quando il budget viene esaurito, passa alla famiglia successiva. Questa gestione impedisce iterazioni indefinite, limita la ripetizione dei medesimi payload e rende esplicita la distribuzione delle risorse della scansione.

La conferma di una vulnerabilità determina invece l'arresto anticipato del ciclo corrente. L'Orchestrator interrompe i tentativi relativi al nodo, registra il risultato e le evidenze nel grafo persistente e restituisce il controllo al Queue Builder. Quest'ultimo esegue la fase di re-discover per rilevare eventuali route, risorse o stati resi accessibili dalle interazioni, aggiorna la coda e seleziona il nodo successivo. Anche quando non viene confermata alcuna vulnerabilità, gli esiti dei tentativi vengono memorizzati per supportare le decisioni future ed evitare lavoro ridondante.

3.9.2 Modalità Discovery

In modalità Discovery, illustrata nella Figura 3.20, l'Orchestrator mantiene il medesimo ruolo di coordinamento, ma persegue un obiettivo differente. Non cerca di validare ipotesi di attacco: dirige invece l'interazione con form e controlli complessi

per far avanzare l'applicazione verso stati non ancora raggiunti e ampliare la superficie conosciuta.

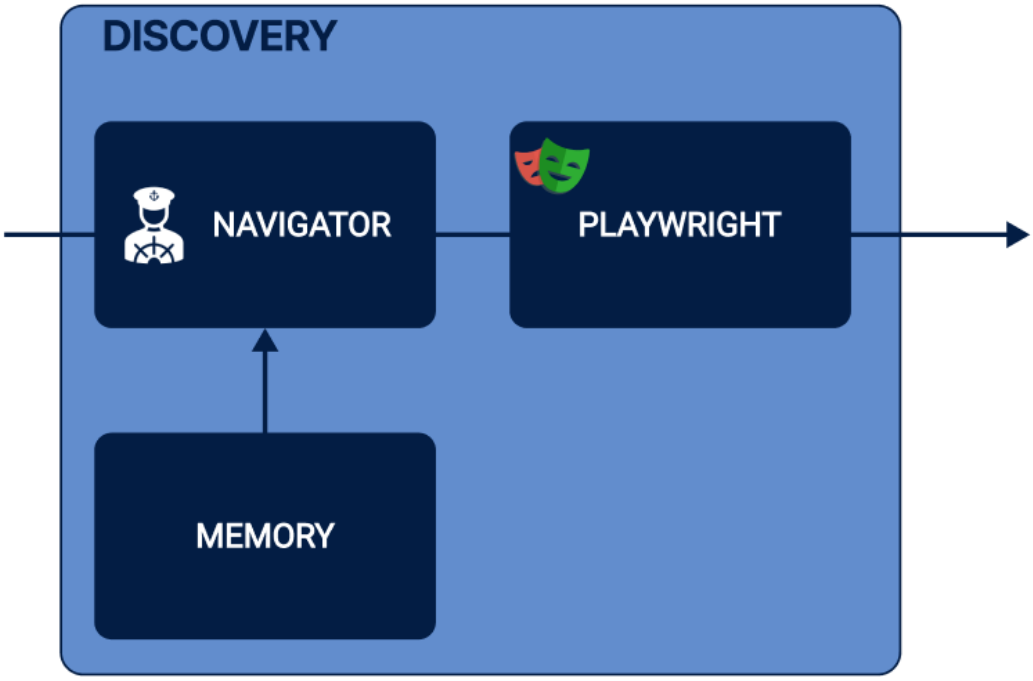


Figura 3.20: Flusso coordinato dall'Orchestrator in modalità Discovery.

Questa modalità è particolarmente utile quando il nodo contiene un form che non può essere gestito efficacemente mediante una strategia deterministica o quando, in assenza di un obiettivo esplicitamente configurato, è necessario scegliere valori plausibili per proseguire nel percorso applicativo. L'Orchestrator consegna al Navigator la superficie corrente e il contesto disponibile. Il Navigator interpreta lo scopo dei campi e propone l'azione che ritiene più adatta a completare il form e avanzare, senza generare payload offensivi né tentare di sfruttare vulnerabilità.

Il piano prodotto dal Navigator viene eseguito da Playwright sotto il controllo dell'Orchestrator. Dopo l'interazione, il DEL acquisisce nuovamente la superficie e la pipeline confronta lo stato precedente con quello successivo per verificare se l'azione abbia determinato una progressione, abbia raggiunto un nodo obiettivo o non abbia prodotto un cambiamento utile. L'Orchestrator registra l'esito e lo restituisce al processo di discovery, che può incorporare nella propria esplorazione le nuove route e gli stati individuati.

In entrambe le modalità, la memoria descritta nella Sezione 3.8 svolge una funzione trasversale. L'Orchestrator ne coordina la consultazione e l'aggiornamento affinché

Analyst, Planner, Navigator ed Evaluator possano operare considerando le informazioni già raccolte.

CAPITOLO 4

Implementazione

Questo capitolo descrive la traduzione in pratica delle scelte architetturali presentate nel capitolo 3. L'obiettivo non è ripercorrere in modo esaustivo ogni componente del sistema, ma approfondire gli aspetti implementativi più significativi e le soluzioni che apportano maggiore valore al progetto.

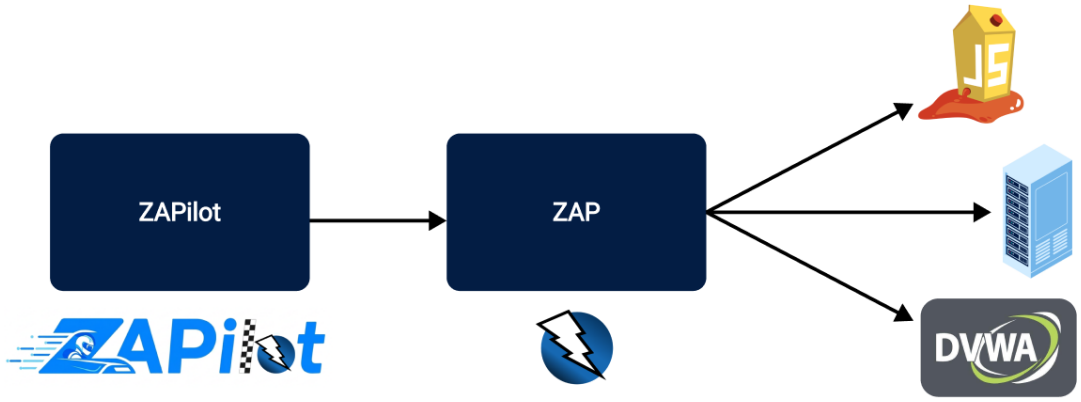


Figura 4.1: Architettura generale di containerizzazione.

La prima scelta implementativa ha riguardato la containerizzazione dell'intero sistema mediante Podman e la creazione di immagini personalizzate. L'architettura, che possiamo vedere schematizzata in figura 4.1, si basa su due container principali, dedicati rispettivamente a ZAPilot (l'immagine di base utilizzata si basa su playwright) e a OWASP ZAP. All'interno del container di ZAPilot è stata riprodotta la struttura

modulare definita in fase di progettazione; tra i suoi componenti è presente un modulo di integrazione con ZAP, responsabile della configurazione del proxy e dell'avvio delle diverse tipologie di scansione. Allo stesso modo è presente un modulo per le interazioni con il browser playwright, uno per la memoria, uno per gli LLM e per l'orchestrazione generale.

Nell'ambiente di sviluppo anche le applicazioni target vengono eseguite in container dedicati. Questa scelta non costituisce tuttavia un vincolo architetturale: ZAPilot può analizzare qualsiasi applicazione raggiungibile attraverso la rete. La containerizzazione rende quindi il sistema portabile, riproducibile e semplice da distribuire, poiché le immagini includono le dipendenze e le configurazioni necessarie alla sua esecuzione.

4.1 Bootstrap

L'implementazione del bootstrap utilizza un profilo YAML per tradurre la configurazione specifica del target nelle operazioni eseguite da Playwright e nella successiva inizializzazione della sessione di ZAP. In questa fase viene preparato il browser, eseguita l'eventuale autenticazione, verificato l'accesso alla risorsa protetta e trasferiti i cookie ottenuti agli altri componenti della pipeline. Inoltre, nella fase di bootstrap viene anche caricato il file di configurazione comprendente i parametri dell'intera espolrazione.

4.1.1 Caricamento e validazione del file di configurazione

In questo file vengono definiti e modificati tutti gli iperparametri che regolano il comportamento dello scanner di vulnerabilità e l'orchestrazione guidata dai modelli linguistici.

Per mantenere il documento conciso, non elencheremo l'intera configurazione, ma ci concentreremo su alcuni degli iperparametri più impattanti che alterano drasticamente le performance e i costi della scansione.

Iperparametri chiave

Le impostazioni possono essere raggruppate in categorie logiche fondamentali:

- **Target e Modalità Operativa**: variabili come `TARGET` e `RECOVERY_MODE` sono la base dell'esecuzione. Definiscono rispettivamente quale applicazione stiamo testando (es. un'istanza locale di *Juice Shop*) e se l'obiettivo della run attuale è mappare la superficie di attacco (*discover*) oppure sfruttare le vulnerabilità (*attack*).
- **Limiti e Budget**: parametri come `BUDGET` , `MAX_RETRIES_PER_NODE` e `TOP_K` sono forse i più delicati. Dettano quanto "sforzo" il sistema può spendere per esplorare o attaccare un singolo endpoint prima di arrendersi. Modificare questi valori cambia il bilanciamento tra una scansione veloce ma superficiale e una profonda ma costosa. Un approfondimento è descritto nella sezione 4.8.

- **Motori di esplorazione (Spidering)**: flag booleani quali `ZAP_DISCOVERY_USE_AJAX_SPIDER` e `PLAYWRIGHT_DISCOVERY_HEADLESS` governano gli strumenti attivi durante la fase di discovery. Controllano l'interazione con le Single Page Application, definendo profondità di navigazione (`MAX_DISCOVERY_STEPS`) e timeout.

Raggruppare questi parametri in un unico file garantisce che chi esegue i test non debba mai alterare la logica interna del codice, potendo adattare l'aggressività dello scan direttamente all'avvio a seconda del tempo e delle risorse a disposizione.

Nell'appendice 6.2 viene riportato un estratto più esaustivo (ma non completo) di file di configurazione.

4.1.2 Caricamento e validazione del profilo di Auth

Il profilo viene validato prima dell'avvio del browser. Il loader controlla la presenza dei campi obbligatori, la correttezza degli URL HTTP o HTTPS e la validità delle strategie dichiarate per i locator. Le credenziali non sono salvate nel file YAML: la configurazione contiene esclusivamente i nomi delle variabili d'ambiente dalle quali recuperare username e password durante l'esecuzione.

La struttura elaborata dal loader comprende l'URL di login, l'endpoint utilizzato come prova di autenticazione, gli host e gli schemi autorizzati, i locator dei campi username e password, il controllo di invio e il numero massimo di tentativi. Il caricamento estrae inoltre le regole di scope, gli eventuali selettori degli overlay e un obiettivo opzionale del workflow, attualmente espresso come condizione deterministica sul path dell'URL; tutti questi valori vengono trasferiti al bootstrap browser attraverso il profilo validato.

Ogni locator è definito dalla strategia di ricerca, dal valore da individuare e dall'opzione di corrispondenza esatta. L'implementazione supporta selettori basati su label, placeholder, CSS, ruolo accessibile, testo visibile e test identifier. Questa rappresentazione dichiarativa permette di scegliere il segnale più stabile disponibile per ciascun controllo senza inserire nel codice del bootstrap dettagli specifici del DOM del target.

Prima di aprire sia la pagina di accesso sia l'endpoint di verifica, il sistema controlla che schema e host appartengano al perimetro autorizzato. La validazione viene applicata anche alle destinazioni raggiunte durante il login, impedendo che un redirect o un errore di configurazione provochi l'invio delle credenziali verso origini esterne allo scope.

4.1.3 Esecuzione della fase Auth del bootstrap

Quando l'autenticazione è abilitata, il sistema inizializza il *BrowserRuntime*, crea il *BrowserContext* e apre la pagina di login indicata nel profilo. Playwright attende i locator configurati, compila i campi delle credenziali e attiva il controllo di submit. La procedura non tenta di correggere automaticamente locator errati mediante euristiche di recovery: un elemento non trovato viene trattato come errore di configurazione o come modifica inattesa dell'applicazione, mantenendo l'esecuzione ripetibile.

Il completamento del form non è sufficiente per considerare valida la sessione. Dopo il submit, il browser visita l'endpoint protetto dichiarato nella sezione `verification` e confronta lo status HTTP ricevuto con `expected_status` . In questo modo un redirect, la semplice creazione di cookie o una sessione inizializzata solo parzialmente non vengono scambiati per un'autenticazione riuscita.

Una volta superata la verifica, i cookie presenti nel *BrowserContext* vengono esportati e trasferiti tramite il *Session Bridge* alla sessione HTTP utilizzata da ZAP. La procedura controlla quanti cookie siano stati propagati e restituisce gli stessi valori al chiamante, così che anche i successivi contesti Playwright impiegati nella discovery o nell'esecuzione dei piani possano partire dal medesimo stato autenticato.

Il numero massimo di tentativi è letto dalla sezione `session` del profilo. Ogni iterazione ripete navigazione, compilazione, submit, verifica dell'endpoint ed esportazione della sessione; in caso di fallimento viene registrata la causa e il bootstrap riparte fino al limite configurato. Il risultato finale e i log espongono soltanto metadati sicuri, come URL finale, status HTTP, numero di cookie e domini associati, mentre credenziali, cookie e altri valori sensibili non vengono stampati.

Esempio di esecuzione

Il seguente estratto mostra un bootstrap completato correttamente. Gli identificativi del profilo, gli URL e i domini sono oscurati, mentre rimangono visibili la sequenza operativa, la verifica con status `200` e la propagazione completa dei cinque cookie rilevati.

```
[2026-07-30 09:25:15] [AUTH] Running isolated bootstrap before
                      ZAP discovery
[2026-07-30 09:25:15] [AUTH] Loading auth profile:
                      configs/targets/sv.yaml
[2026-07-30 09:25:15] [AUTH] Auth profile loader returned
                      successfully
[2026-07-30 09:25:15] [AUTH] Profile loaded | id=XXX (oscurato)
                      | login_url=XXX (oscurato)
                      | verification_url=XXX (oscurato)
[2026-07-30 09:25:15] [AUTH] Starting BrowserRuntime
                      | headless=True | proxy=http://zap:8090
[2026-07-30 09:25:15] [BROWSER] Starting Playwright
                      | headless=True | proxy=http://zap:8090
[2026-07-30 09:25:15] [BROWSER] Playwright started
[2026-07-30 09:25:16] [BROWSER] Chromium launched
[2026-07-30 09:25:16] [BROWSER] Browser context created
[2026-07-30 09:25:16] [BROWSER] Browser page created
[2026-07-30 09:25:16] [AUTH] BrowserRuntime ready;
                      starting Playwright login
[2026-07-30 09:25:16] [AUTH] ensure_authenticated entered
                      | profile=XXX (oscurato) | enabled=True
[2026-07-30 09:25:16] [AUTH] profile=XXX (oscurato)
                      | attempt=1/2
[2026-07-30 09:25:16] [AUTH] Navigating to login entrypoint:
                      XXX (oscurato)
[2026-07-30 09:25:19] [AUTH] Login page loaded | status=200
[2026-07-30 09:25:19] [AUTH] Waiting for username locator
[2026-07-30 09:25:19] [AUTH] Filling username
[2026-07-30 09:25:19] [AUTH] Waiting for password locator
[2026-07-30 09:25:19] [AUTH] Filling password
[2026-07-30 09:25:19] [AUTH] Waiting for submit locator
[2026-07-30 09:25:19] [AUTH] Clicking submit
[2026-07-30 09:25:21] [AUTH] Login submit completed
                      | current_url=XXX (oscurato)
[2026-07-30 09:25:21] [AUTH] Form login completed;
                      verifying protected endpoint
[2026-07-30 09:25:22] [AUTH] Protected endpoint verified
                      | status=200
[2026-07-30 09:25:22] [AUTH] Browser cookies exported | count=5
```

```
[2026-07-30 09:25:22] [SESSION_BRIDGE] sessione
                      'llm_authenticated' creata per XXX (oscurato)
[2026-07-30 09:25:22] [SESSION_BRIDGE] verifica: sessione attiva
                      su XXX (oscurato) = 'llm_authenticated'
                      (5/5 cookie propagati)
[2026-07-30 09:25:22] [AUTH] bootstrap completed
                      | profile=XXX (oscurato) | status=200
                      | cookie_count=5
                      | domains=[XXX (oscurato)]
[2026-07-30 09:25:22] [AUTH] Bootstrap successful
                      | final_url=XXX (oscurato) | status=200
                      | cookie_count=5
```

Esempio di autenticazione dvwa

Il profilo `dvwa-login` configura l'autenticazione automatizzata per Damn Vulnerable Web Application, eseguita interamente tramite HTTP. La sezione `target` autorizza l'host `dvwa` e, lasciando vuoto `include_path_prefixes`, non limita i percorsi inclusi. Vengono invece esclusi `/login.php`, `/setup.php` e `/logout.php` sia mediante prefissi sia tramite espressioni regolari. L'esclusione della pagina di logout impedisce alla navigazione di invalidare accidentalmente i cookie `PHPSESSID` e `dvwaSession`; le pagine di login e setup non vengono reinserite nel normale flusso di esplorazione.

Le credenziali sono lette dalle variabili d'ambiente `USERNAME` e `PASSWORD`. I campi vengono individuati con selettori CSS in modalità `exact`, mentre il locator del submit accetta sia un input denominato `Login` sia un input di tipo `submit` con valore `Login`. La doppia alternativa gestisce le variazioni strutturali osservabili nel form senza introdurre una ricerca euristica più ampia.

L'oracolo di autenticazione è la risorsa `http://dvwa/security.php`, accessibile nella sessione prevista dopo il login. Il bootstrap considera riuscita la procedura soltanto quando la richiesta restituisce lo status `200`. Il limite `max_login_attempts: 2` consente un secondo tentativo in presenza di timeout del DOM, latenze di rete o desincronizzazioni transitorie prima che l'avvio della scansione venga interrotto.

```
target:
  allowed_hosts:
    - dvwa
  scope:
    allowed_schemes:
      - http
    include_path_prefixes: []
    exclude_path_prefixes:
      - /login.php
      - /logout.php
      - /setup.php
    exclude_url_patterns:
      - "/login\\.php(?:/|$)"
      - "/logout\\.php(?:/|$)"
      - "/setup\\.php(?:/|$)"

authentication:
  enabled: true
  login_url: "http://dvwa/login.php"
  credentials:
    username_env: "USERNAME"
    password_env: "PASSWORD"
  selectors:
    username:
      by: css
      value: "input[name='username']"
      exact: true
    password:
      by: css
      value: "input[name='password']"
      exact: true
    submit:
```

```
      by: css
      value: "input[name='Login'],
              input[type='submit'][value='Login']"
      exact: true
  verification:
    url: "http://dvwa/security.php"
    expected_status: 200

session:
  max_login_attempts: 2
```

Esempio di autenticazione juice-shop e gestione degli overlay

Il profilo `juice-shop` implementa il caso in cui il target possa essere esplorato senza una sessione autenticata. La proprietà `authentication.enabled` è quindi impostata a `false` , lo schema autorizzato è HTTP e lo scope include l'intero host a partire dal percorso radice. Il valore `max_login_attempts: 1` rimane presente nella struttura comune del profilo, ma non attiva alcun tentativo di login quando l'autenticazione è disabilitata.

L'elemento specifico di questa configurazione è la lista `overlay_selectors` della sezione `browser` . Prima dell'estrazione profonda e delle interazioni successive, la routine `dismiss_blocking_ overlays` scorre i selettori configurati, individua gli elementi visibili e tenta di attivarne il controllo di chiusura. Nel profilo seguente il selettore identifica il pulsante del banner dei cookie di Juice Shop; la rimozione dell'overlay impedisce che il livello sovrapposto intercetti i click destinati al campo di ricerca, ai pulsanti del menu o agli altri controlli utilizzati durante discovery e attacco.

```
profile_id: juice-shop

target:
  allowed_hosts:
    - juice-shop
  scope:
    allowed_schemes:
      - http
    include_path_prefixes:
      - /
    exclude_path_prefixes: []
    exclude_url_patterns: []

authentication:
  enabled: false

browser:
  overlay_selectors:
    - 'a[aria-label="dismiss cookie message"][role="button"]'

session:
  max_login_attempts: 1
```

4.2 Queue Builder

Il funzionamento generale del Queue Builder e la sua articolazione nelle fasi di inizializzazione, re-discover e chiusura sono descritti nella Sezione 3.3. Questa sezione si concentra sugli aspetti implementativi che hanno richiesto un'attenzione particolare e che trasformano i risultati prodotti dai diversi strumenti di discover in una coda ordinata e compatibile con lo scope del target.

4.2.1 Normalizzazione, scope e priorità degli URL

Gli URL raccolti da ZAP e dal crawler Playwright non vengono inserite direttamente nella coda. Ogni risultato attraversa prima una pipeline comune di normalizzazione, classificazione e filtraggio, così che indirizzi equivalenti producano un solo nodo e che le risorse non navigabili o esterne al perimetro autorizzato non entrino nel grafo applicativo.

La canonicalizzazione uniforma schema e host, rimuove le porte standard, normalizza il path e ordina i parametri della query string. La costruzione della chiave canonica tratta separatamente i parametri volatili. Token di sessione, UUID e identificatori casuali possono assumere un valore diverso a ogni visita senza modificare la funzione dell'endpoint; mantenerli nella chiave genererebbe quindi un numero potenzialmente illimitato di nodi riferiti alla stessa risorsa. Il sistema rimuove i parametri dichiarati esplicitamente come volatili.

Dopo la normalizzazione, la risorsa viene classificata come route SPA, pagina web, endpoint API, file statico o elemento non rilevante. Soltanto le route SPA e le pagine vengono ammesse nella coda browser, perché costituiscono stati nei quali l'Orchestrator può creare un contesto di navigazione e interagire con il DOM. Le API e i file osservati restano disponibili nella history di ZAP, ma non vengono trasformati automaticamente in nodi da visitare con il browser. In ogni caso, indipendentemente dalla classificazione, verranno tutti controllati dai controlli finali di Active Scan, GitLeaks e SensitivePath.

Il filtro verifica inoltre che schema, origine e percorso rispettino lo scope del target, escludendo indirizzi esterni, schemi diversi da HTTP e HTTPS e risorse statiche o non navigabili. Lo stesso controllo viene applicato durante la raccolta e prima dell'eventuale inserimento nella history di ZAP, in modo che gli URL scoperti dal browser non possano ampliare implicitamente il perimetro autorizzato.

4.2.2 Configurazione del Context ZAP

Prima di avviare discover e l'Active Scan, l'implementazione crea un *Context* ZAP per il target corrente. Le regole del *TargetScope*, caricate durante il bootstrap, vengono convertite in espressioni regolari che ZAP utilizza per decidere quali richieste includere o escludere.

Per ciascun prefisso incluso viene generata una regola che ammette sia il percorso indicato sia le risorse discendenti. Le esclusioni vengono registrate separatamente e possono rimuovere aree non autorizzate o non pertinenti anche quando appartengono alla stessa origine del target. Dopo l'inserimento delle espressioni, il Context viene marcato come *in scope* e diventa il riferimento utilizzato dagli spider e dall'Active Scan.

La creazione del Context è preceduta dalla validazione del seed URL. Se lo schema o l'host non appartengono a quelli autorizzati dal profilo, la configurazione viene rifiutata prima di registrare le regole e nessuno spider o Active Scan viene avviato sull'origine non ammessa.

4.2.3 Crawler Playwright

La progettazione della discover combinata è descritta nella Sezione 3.3.4; a livello implementativo, il canale browser è realizzato dalla routine `_playwright_dom_crawl` . Il metodo apre la pagina in un browser headless, esegue il codice JavaScript e raccoglie gli stati di navigazione che non emergono dalle sole richieste HTTP. Le route contenenti fragment, come `#/dashboard` , vengono conservate integralmente perché identificano stati distinti della SPA, anche se la parte successiva al carattere `#` non viene inviata al server.

Prima dell'estrazione, il crawler richiama `dismiss_blocking_overlays` , che utilizza i selettori definiti nel profilo del target per chiudere banner e componenti sovrapposti. La rimozione preventiva evita che tali elementi intercettino i click eseguiti nelle fasi successive e rende accessibili i controlli sottostanti.

La raccolta dei collegamenti non si limita agli attributi HTML `href` . Uno script eseguito nel contesto della pagina ricerca anche attributi utilizzati dai framework frontend, tra cui `[routerLink]` , `ng-reflect-router-link` e `data-href` . Gli indirizzi ricavati vengono risolti rispetto alla pagina corrente e trasferiti alla stessa pipeline di normalizzazione e filtraggio utilizzata per i risultati di ZAP.

Vediamo un esempio di estrazione nel seguente log:

```
[INFO] Initial URLs discovered: 14
[INFO]http://juice-shop:3000/#/about
[INFO]http://juice-shop:3000/#/chatbot
[INFO]http://juice-shop:3000/#/contact
[INFO]http://juice-shop:3000/#/login
[INFO]http://juice-shop:3000/#/photo-wall
...
```

Per raggiungere le route contenute nei menu dinamici, il crawler individua i possibili controlli di apertura mediante l'insieme `_TRIGGER_SELECTORS` . La routine `_expand_trigger_and_collect` attiva il controllo genitore, attende l'apertura del componente e analizza le voci rese visibili. Ciascuna voce viene selezionata per osservare e registrare l'eventuale variazione dell'URL; al termine dell'interazione, `go_back` ripristina lo stato precedente, permettendo di continuare l'esplorazione degli altri elementi senza ricaricare inutilmente la pagina o perdere lo stato della sessione. In questo modo possiamo vedere (un esempio nel grafo in figura 4.6) come in coda abbiamo tutte le pagine altrimenti non visibili con i normali crowler.

4.3 Deterministic Extraction Layer

La progettazione generale del DEL e la distinzione concettuale tra identità strutturale e identità logica sono state presentate nella sezione 3.4. Questa sezione descrive la relativa implementazione: classi e contratti dati, regole di classificazione, feature estratte e costruzione dei payload crittografici.

4.3.1 Pipeline di assemblaggio

Il punto di ingresso è la classe `DeterministicSurfaceAssembler` . Il componente riceve il contesto prodotto dal browser extractor e coordina, nell'ordine, la deduplicazione dei dati in una `DelSurface` , la classificazione della tipologia di superficie, l'estrazione delle feature strutturali, il calcolo dello `state_hash` e la generazione della `surface_signature` . Il risultato viene raccolto in un oggetto `DeterministicSurfaceContext` , che costituisce il contratto dati consegnato agli altri livelli del sistema.

```
DeterministicSurfaceContext(
    surface,
    surface_kind,
    features,
    state_hash,
    surface_signature
)
```

Il contesto è serializzabile e persistibile e contiene sia la descrizione dettagliata della pagina sia le informazioni sintetiche necessarie per confrontare gli stati, riconoscere osservazioni già elaborate e recuperare i tentativi storici.

4.3.2 Implementazione del modello canonico

La struttura principale prodotta dal DEL è `DelSurface` . Il modello contiene l'URL, il titolo, un suggerimento testuale sintetico, le intestazioni, le form, le azioni cliccabili, i campi interattivi e gli eventuali messaggi di errore.

```
DelSurface(
    url,
    title,
    text_hint,
    headings,
    forms,
    buttons,
    fields,
    errors
)
```

Il contesto grezzo viene convertito in un dizionario JSON serializzabile e ripulito prima della costruzione dei modelli. Le liste `headings` ed `errors` vengono normalizzate eliminando elementi vuoti, spazi superflui e contenuti oltre i limiti previsti.

Ogni campo è rappresentato mediante `DelField` . Il modello conserva tag HTML, tipo, visibilità, obbligatorietà, stato di abilitazione, etichetta, placeholder, attributi ARIA,

testo circostante e metadati della form di appartenenza. Per gli elementi `select` vengono registrate anche le opzioni disponibili, con etichetta, valore, stato di disabilitazione e selezione.

Il riferimento canonico `ref` viene costruito scegliendo il primo segnale disponibile secondo il seguente ordine di preferenza:

1. attributo `id`;
2. attributo `name`;
3. attributo `data-testid`;
4. etichetta associata;
5. attributo `aria-label`;
6. placeholder;
7. testo circostante;
8. riferimento deterministico basato su tag, tipo e posizione del campo.

Quando nessuno dei segnali espliciti è disponibile, il riferimento incorpora un breve hash del fingerprint formato da tag, tipo e posizione.

Le azioni visibili sono modellate mediante `DelButton`, che conserva tag, tipo, testo, URL di destinazione, identificatori, visibilità e stato di disabilitazione. Le form sono rappresentate da `DelForm` e includono identificatore, nome, azione, metodo HTTP, encoding, testo associato e pulsanti di invio rilevati.

4.3.3 Classificazione ed estrazione delle feature

La superficie canonica viene classificata dal `SurfaceClassifier`. Il classificatore costruisce un testo normalizzato concatenando URL, titolo, suggerimento testuale, intestazioni, errori, attributi delle azioni e metadati delle form; applica quindi una sequenza ordinata di regole lessicali e restituisce come `SurfaceKind` la prima categoria corrispondente.

La tassonomia comprende `login`, `signup`, `search`, `reset_password`, `contact_or_feedback`, `file_upload`, `checkout_or_payment`, `admin_settings` e `generic_data_entry`. Il riconoscimento utilizza termini come `login`, `sign in`, `register`, `search`, `upload`, `payment`, `admin` e le relative varianti; quando nessuna regola trova corrispondenza, viene assegnata `generic_data_entry`.

`SurfaceFeatureExtractor` calcola il numero totale di campi, quello dei campi visibili e abilitati e quello dei campi obbligatori. Le feature booleane indicano la presenza di campi password, email, file, textarea, select, hidden, checkbox e radio button, di contenitori `form`, di pulsanti di invio o navigazione e di vincoli come `pattern`, `minlength` e `maxlength`. I flag `auth_like`, `search_like`, `upload_like`, `payment_like`, `feedback_like` e `admin_like` derivano dalle keyword presenti nel testo osservato e rimangono distinti da `SurfaceKind`.

4.3.4 Calcolo degli identificatori

- **Normalizzazione:** tutti i testi coinvolti vengono ripuliti (spazi extra rimossi all'inizio, alla fine e in mezzo) e convertiti in minuscolo.
- **Serializzazione deterministica:** il dizionario dati (payload) viene convertito in una stringa JSON. Viene usato `sort_keys=True` in modo che l'ordine delle chiavi nel dizionario non alteri mai il risultato finale.
- **Hashing:** la stringa JSON risultante viene codificata in UTF-8 e passata a una funzione di hash SHA-256, restituendo una stringa esadecimale.

Il servizio `SurfaceIdentityService` calcola entrambi gli identificatori applicando SHA–256 a payload JSON serializzati con chiavi ordinate. La procedura crittografica è quindi la stessa, mentre la composizione dei payload varia in funzione dell'identità che si intende rappresentare: strutturale per lo `state_hash` e logica per la `surface_signature` .

State hash

Lo `state_hash` è definito come:

$$state_hash = \mathrm{SHA256}(\mathrm{JSON}_{\mathrm{ord}}(P_{\mathrm{state}}))$$

Il payload P_{state} rappresenta la configurazione interattiva della superficie mediante i metadati strutturali di campi, pulsanti e form. Per ogni campo vengono registrati `id` , `name` , `data-testid` , tipo e tag HTML, insieme agli stati `required` , `visible` , `disabled` e `readonly` . Per i pulsanti sono conservati tag, tipo, URL di destinazione, identificatori, visibilità e stato di abilitazione; per le form vengono inclusi identificatore, nome, azione, metodo, encoding e pulsanti di invio associati.

Dal payload sono esclusi i valori inseriti dall'utente e i contenuti testuali liberi o transitori, come titoli, paragrafi, intestazioni, notifiche, tooltip, risultati dinamici, errori, placeholder, etichette e testo dei pulsanti. Di conseguenza, la comparsa del messaggio "Dati salvati", la modifica del valore corrente di un input o la sostituzione del testo di un pulsante da "Invia" a "Conferma" non alterano lo `state_hash` . Possono invece produrre un nuovo valore il passaggio di un controllo da disabilitato ad abilitato, la modifica di un identificatore o di un vincolo, l'aggiunta di un campo e, più in generale, qualsiasi variazione della struttura utilizzata per l'interazione.

Surface signature

La `surface_signature` è definita come:

$$surface_signature = \mathrm{SHA256}(\mathrm{JSON}_{\mathrm{ord}}(P_{\mathrm{surface}}))$$

Il payload P_{surface} è più compatto e descrive il ruolo logico degli elementi. Include innanzitutto la categoria generale ottenuta da `surface_kind` , ad esempio `login` ,

`checkout` o `search` . Per ogni campo conserva `semantic_text` , tipo, tag, obbligatorietà e appartenenza a una form; per i pulsanti utilizza tag, tipo e testo, mentre per le form considera metodo, azione e contenuto testuale. Il testo di un'azione, come "Login" o "Paga", contribuisce pertanto alla `surface_signature` , mentre viene escluso dallo `state_hash` .

4.3.5 Persistenza, cache e retest

Quando una superficie presenta sia la stessa `surface_signature` sia lo stesso `state_hash` di un tentativo storico confermato, la struttura viene considerata compatibile con un replay esatto. Possono quindi essere recuperati il candidato di attacco, i valori già utilizzati, il piano e le evidenze attese conservate nello storico degli attempt. Se coincide soltanto la `surface_signature` , il record rimane utile per un retest logico, ma locator, valori e strategia di interazione devono essere adattati alla superficie corrente.

La versione dell'applicazione non fa parte né di P_{state} né di P_{surface}.
Il valore `app_version` viene conservato separatamente nel file di configurazione: ogni attempt registra la versione in cui è stato eseguito, mentre il record aggregato mantiene la prima e l'ultima versione nelle quali il candidato è stato osservato.

La correlazione storica utilizza la `surface_signature` per recuperare una funzionalità già osservata, lo `state_hash` per verificarne la compatibilità strutturale e `app_version` per distinguere tentativi appartenenti a release differenti. Un attempt confermato viene descritto dalla combinazione di candidato, firma della superficie, hash dello stato, versione, valori sottoposti ed evidenze raccolte.

4.4 Browser Playwright

La struttura generale del Browser Layer, il ciclo di vita del runtime e il rapporto con gli altri componenti sono descritti nella Sezione 3.5. A livello implementativo, l'elemento che ha richiesto le maggiori accortezze è `ActionExecutor` , responsabile della traduzione dei piani in interazioni concrete con il DOM. La classe deve individuare il controllo corretto, applicare il valore rispettandone la tipologia e inviare i dati senza confondere il submit con altre azioni presenti nella pagina.

4.4.1 Action Executor

`ActionExecutor` privilegia la semantica degli elementi rispetto alla semplice posizione nel DOM. La localizzazione parte dagli attributi più descrittivi e utilizza i selettori CSS generici soltanto come ultima possibilità. Anche il submit viene circoscritto, quando possibile, alla form proprietaria del campo compilato, così che pulsanti appartenenti ad altri componenti non vengano interpretati come controlli di invio. Nel flusso ordinario non viene inoltre utilizzato `force=True` : se un elemento è nascosto,

coperto o non interagibile, l'esecutore passa a una strategia alternativa invece di forzare un click che non corrisponderebbe al comportamento effettivo dell'interfaccia.

Localizzazione dei controlli

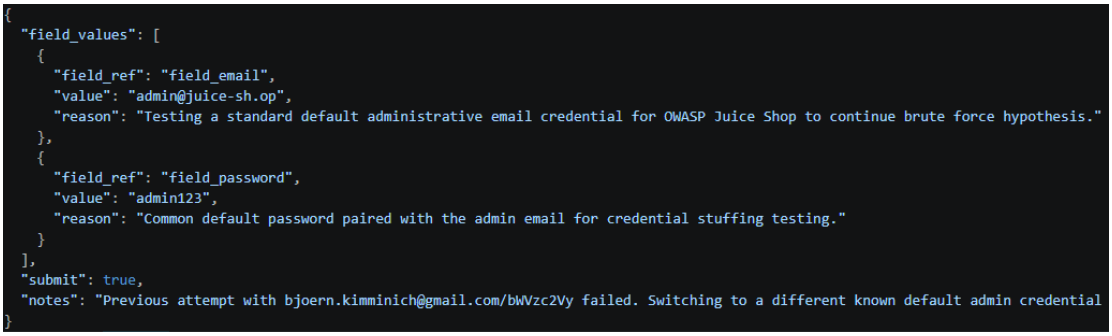
Il metodo `_candidate_locators` costruisce una sequenza di locator ordinata per affidabilità. La ricerca prova innanzitutto `data-testid`, quindi le label visibili, gli attributi `aria-label`, gli attributi `id` e `name`, il placeholder e, infine, fallback mirati basati sul tag CSS e sull'attributo `type`, come `input[type="text"]`. L'ordine non serve soltanto a trovare un elemento, ma a preferire il controllo che possiede il legame più chiaro con il campo descritto nel piano.

Ogni locator può restituire più risultati, situazione comune quando la pagina contiene template nascosti, versioni responsive dello stesso componente o elementi duplicati. Per questo motivo l'algoritmo non utilizza automaticamente il primo nodo trovato: scorre le corrispondenze e si arresta soltanto quando individua un elemento visibile e realmente interagibile.

Compilazione dei campi

Una volta risolto il controllo, `apply_field` sceglie l'operazione in base al tag e al tipo HTML. Per gli elementi `select` sono supportate selezioni singole e multiple; la corrispondenza viene cercata prima sul `value` e poi sulla label testuale, escludendo le opzioni disabilitate. Checkbox e radio button accettano sia valori booleani sia rappresentazioni testuali come *"true"*, *"checked"* e *"1"*, che vengono normalizzate prima di modificare lo stato del controllo.

Gli input testuali e le textarea vengono compilati soltanto dopo aver verificato che l'elemento supporti effettivamente l'inserimento di testo. Oltre ai tag standard, il controllo riconosce gli elementi con `contenteditable="true"`. Gli input di tipo file seguono invece un percorso separato e delegano il caricamento al metodo nativo `set_input_files` di Playwright. La figura 4.2 mostra un esempio completo: il riferimento ricevuto nel piano viene risolto attraverso i locator candidati e il valore viene applicato al primo controllo visibile e interagibile.



```
{
  "field_values": [
    {
      "field_ref": "field_email",
      "value": "admin@juice-sh.op",
      "reason": "Testing a standard default administrative email credential for OWASP Juice Shop to continue brute force hypothesis."
    },
    {
      "field_ref": "field_password",
      "value": "admin123",
      "reason": "Common default password paired with the admin email for credential stuffing testing."
    }
  ],
  "submit": true,
  "notes": "Previous attempt with bjoern.kimminich@gmail.com/bWVzc2Vy failed. Switching to a different known default admin credential
}
```


Figura 4.2: Esempio di risoluzione dei locator e compilazione controllata dei campi da parte di `ActionExecutor`, in particolare il campo `field_email` e `field_password`.

Strategie di submit

Il metodo `submit_current_form` distingue anzitutto se il campo appena compilato appartenga a un elemento `<form>`. Quando la form proprietaria è disponibile, la ricerca del pulsante di invio rimane confinata al suo interno. In questo perimetro l'esecutore può utilizzare submit HTML espliciti, euristiche basate sul testo dei pulsanti e le classi CSS configurate. Se nessun click fisico riesce, viene invocata tramite JavaScript l'API `requestSubmit()`, che mantiene la validazione prevista dal browser; come ultimo tentativo viene premuto `Enter` sul campo originale.

Le barre di ricerca delle SPA sono spesso realizzate senza un contenitore `form`. In questo caso la strategia parte dalla pressione di `Enter` sul campo compilato. Soltanto se questa operazione fallisce vengono cercati controlli globali, limitandosi però a submit HTML espliciti ed evitando corrispondenze fondate unicamente sul testo visibile.

Questo vincolo è stato introdotto dopo un errore osservato nelle prime versioni dell'esecutore. Durante un tentativo sulla barra di ricerca di un catalogo, il fallback testuale individuava come possibile controllo di invio il pulsante "Pagina successiva" collocato in basso a destra. Il click modificava la pagina dei prodotti invece di eseguire la ricerca e invalidava l'intero tentativo. Come illustrato nella figura 4.3, il controllo aggiunto impedisce ora che un campo privo di form attivi pulsanti generici di navigazione o paginazione: viene preferito `Enter` e l'eventuale fallback globale considera soltanto elementi con una semantica HTML esplicita di submit.

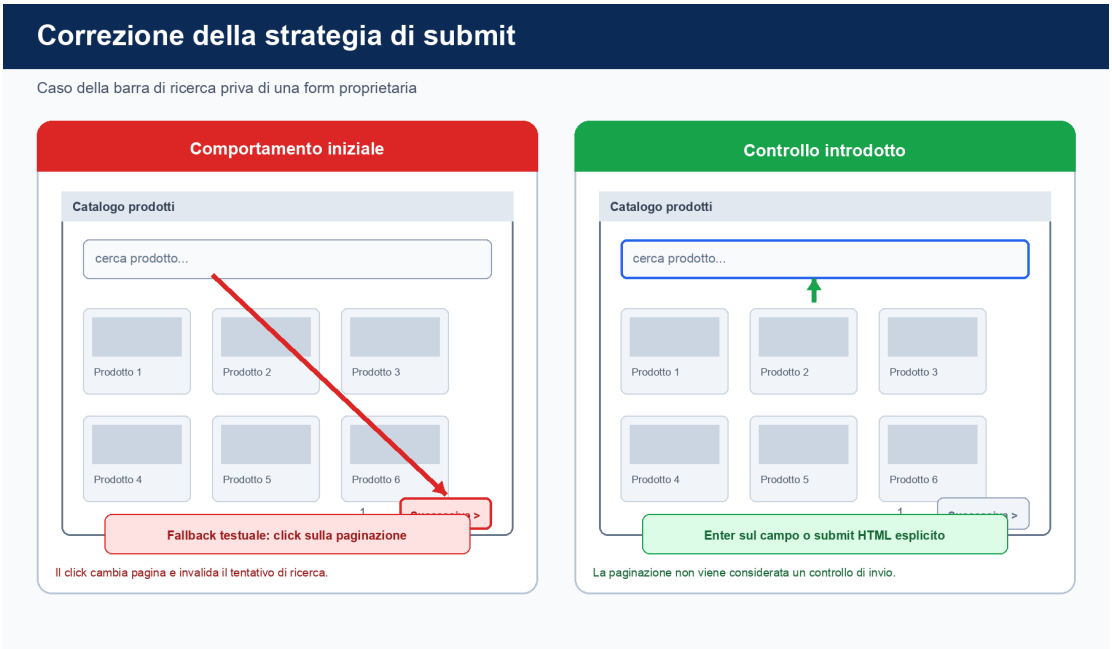


Figura 4.3: Errore iniziale sulla paginazione e controllo introdotto per circoscrivere il submit della barra di ricerca.

Dopo ogni tentativo di invio, `_wait_post_submit` attende la stabilizzazione della pagina monitorando lo stato `networkidle` . L'attesa tollera i timeout prodotti da polling, WebSocket o altre connessioni persistenti in background, perché in questi casi il mancato raggiungimento di una rete completamente inattiva non implica necessariamente che il submit sia fallito.

4.5 LLM

Questa sezione approfondisce le regole deterministiche utilizzate per raccogliere e organizzare le evidenze fornite all'Evaluator. Per non appesantire la trattazione, gli esempi di log che illustrano il funzionamento dei diversi ruoli LLM sono riportati nell'appendice 6.2.

Regole deterministiche

Le regole deterministiche consentono di estrarre e confrontare lo stato del browser prima e dopo un'azione, raccogliendo le variazioni osservate in un *evidence bundle* strutturato. L'obiettivo è fornire all'Evaluator informazioni fattuali, riducendo il rumore e rendendo più leggibili gli effetti dell'esecuzione.

Il confronto considera anzitutto l'esito tecnico dell'azione e le modifiche agli URL e alle route. Attraverso un *diff* semantico delle rappresentazioni prodotte dal DEL, individua poi le variazioni nei testi visibili, nei titoli e nei controlli interattivi, senza

richiedere l'elaborazione dell'intero albero del DOM. A queste osservazioni si aggiungono i messaggi delle finestre di dialogo intercettate e il *delta* degli errori, che distingue le nuove anomalie da quelle già presenti o risolte.

Le evidenze raccolte orientano il giudizio dell'Evaluator, ma non ne determinano rigidamente l'esito. Nel caso di un tentativo di XSS, ad esempio, la comparsa di una finestra di dialogo può costituire un segnale forte della riuscita dell'attacco; analogamente, il passaggio a una nuova route, accompagnato da contenuti che rimandano a un'area di amministrazione, offre elementi utili alla valutazione per una sql injection. Il modello dispone così di un quadro concreto degli effetti osservati, sul quale l'evaluator può fondare il proprio giudizio.

4.6 Wiki

La Wiki proposta si basa su 4 operazioni fondamentali:

- **Ingest**: Elabora una nuova fonte (testo, URL, PDF) depositandola in un archivio grezzo (raw). L'LLM estrae le informazioni, crea o aggiorna le pagine del wiki, genera i collegamenti (wikilink) e aggiorna automaticamente l'indice e il registro (log).
- **Lint**: Esegue controlli di integrità per mantenere "in salute" la base di conoscenza. Verifica la presenza di collegamenti interrotti, pagine orfane, tag non validi o formattazione errata nei metadati.
- **Merge**: Se una nuova fonte affronta un argomento già trattato, l'LLM unisce (merge) i nuovi concetti nella pagina esistente. Questo arricchisce la conoscenza pregressa e aggiorna le sezioni senza sovrascriverle o perdere dati.
- **Query**: Risponde alle domande interrogando l'indice e le pagine del wiki già compilate, invece di usare il RAG tradizionale sulle fonti grezze. Le risposte di alto valore o le nuove sintesi possono essere salvate nuovamente nel wiki come nuove pagine per far crescere la base di conoscenza.

L'idea, descritta nella corrispettiva sezione implementativa, è stata pensata interamente da Andrej Karpathy e riprogettata seguendo esattamente le indicazioni proposte. Si è scelto dunque di organizzare la cartella in:

```
llmwiki/
 raw/                 # Fonti originali e immutabili
 wiki/
    articles/        # Pagine tecniche complete
    summaries/       # Sintesi compatte
    concepts/        # Concetti riusabili trasversali
    index.md         # Indice generato
```

```
    backlinks.md      # Backlink generati
    outputs/          # Risultati di lint e analisi
state/
     manifest.json     # Stato e hash delle fonti elaborate
     source_map.json   # Mappatura fonte--pagina
```

La directory `raw/` rappresenta la fonte di verità e non viene modificata dal processo di compilazione. La separazione tra fonti e wiki derivata è coerente con l'impostazione della LLM Wiki: le fonti restano immutabili, mentre il livello wiki può essere aggiornato e rigenerato all'occorrenza.

4.6.1 Creazione della LLM Wiki: ingest e merge

Per ciascuna fonte, l'ingest svolge i seguenti passi:

1. calcolo dell'hash della fonte e consultazione del manifest;
2. estrazione strutturata di titolo, severità, tecniche, concetti, entità, componenti vulnerabili, mitigazioni e payload;
3. merge incrementale con article, summary e concept pages esistenti;
4. scrittura delle pagine aggiornate;
5. costruzione dell'indice e dei backlink;
6. aggiornamento del manifest soltanto dopo il completamento del file;

Il manifest consente di distinguere una normale esecuzione incrementale (merge) da una rigenerazione completa. Senza l'opzione `-force` , i file raw invariati e già completati vengono saltati oppure incrementati se vengono aggiunti file che parlano di stessi argomenti. Con `-force` , le pagine derivate vengono rigenerate anche se la fonte non è cambiata; questa opzione è utile dopo modifiche a prompt, schema, normalizzatori o regole di merge.

4.6.2 Knowledge di base persistente

La base di conoscenza iniziale è stata costruita elaborando i documenti presenti nel repository `bhavsec/autopentest-ai` , già descritto nella Sezione 1.2.2. Tali materiali sono stati assunti come punto di partenza e successivamente riorganizzati dalla procedura di ingestione della LLM Wiki. La raccolta combina la struttura della *OWASP Web Security Testing Guide* con tecniche, esempi e payload riconducibili anche ai contenuti della PortSwigger Web Security Academy, una piattaforma dedicata alla formazione e alla sperimentazione pratica sulla sicurezza delle applicazioni web bhavsec [2026]; PortSwigger [2026].

La knowledge base di origine comprende 109 casi di test distribuiti in dodici categorie. La categoria più ampia è `INPV` , che comprende venti verifiche sulla validazione degli input e considera XSS, SQL injection, command injection, path traversal, XXE (XML External Entity vulnerability), LDAP (Lightweight Directory Access Protocol) injection. `ERRH` raccoglie due test sulla gestione degli errori, con particolare attenzione ai messaggi esposti e agli stack trace, mentre `CRYP` comprende quattro verifiche crittografiche relative, ad esempio, alla configurazione TLS e all'impiego di cifratura debole. L'area `BUSL` contiene dieci test sulla logica applicativa, tra cui bypass dei workflow, request forgery, caricamento di file e limitazione delle richieste; `CLNT` riunisce quattordici verifiche lato client, quali DOM XSS, open redirect, WebSocket e meccanismi di storage; infine, `APIT` comprende tre casi dedicati al testing di API GraphQL, REST. Altre categorie si lasciano approfondire al lettore.

Ogni file di test affianca alla descrizione della vulnerabilità una procedura operativa eseguibile da riga di comando, con esempi basati su `curl` o sull'invocazione di strumenti specifici. I payload sono organizzati secondo livelli di bypass progressivi, distinti in base, intermedio e avanzato; sono inoltre presenti criteri di rilevamento, indicazioni per la valutazione della severità, suggerimenti di remediation e riferimenti alle fonti. Durante l'ingestione, questi contenuti non vengono copiati direttamente nel contesto dei modelli, ma trasformati nelle pagine strutturate, nelle sintesi e nei concetti persistenti descritti nelle sezioni precedenti.

4.6.3 Retrieval ibrido della LLM Wiki: query

Il retrieval della wiki non opera su un unico insieme indistinto di chunk grezzi. Esso considera pagine Markdown compilate e può assegnare punteggi aggiuntivi a titolo, slug, tipo di pagina e relazioni strutturali.

Sia P l'insieme delle pagine della wiki. Per una query q, il punteggio di una pagina $p \in P$ è definito come:

$$S(q,p) = w_{\text{bm25}} \cdot \text{BM25}(q,p) + w_{\text{title}} \cdot T(q,p) + w_{\text{slug}} \cdot G(q,p) + w_{\text{structure}} \cdot H(q,p),$$

dove:

- $\text{BM25}(q,p)$ misura la corrispondenza lessicale tra query e contenuto della pagina;
- $T(q,p)$ assegna un boost ai termini presenti nel titolo;
- $G(q,p)$ assegna un boost ai token corrispondenti allo slug;
- $H(q,p)$ assegna un boost in base al tipo della pagina e ai collegamenti strutturali;
- $w_{\text{bm25}}, w_{\text{title}}, w_{\text{slug}}, w_{\text{structure}}$ sono pesi configurabili.

Dati i punteggi, il retriever restituisce le prime k pagine (k fissato nel profilo di configurazione):

$$\text{Retrieve}(q, k) = \text{TopK}_{p \in P}\, S(q, p).$$

Le pagine selezionate vengono poi condensate in una *wiki guidance* strutturata. Questa guidance include, ove disponibili, tecniche applicabili, payload presenti nella fonte, condizioni di utilizzo, mitigazioni, limiti e riferimenti alla pagina di origine. Il Planner riceve tale materiale come contesto informativo e non come istruzione autonoma da eseguire.

4.6.4 Integrità del knowledge graph: lint

La wiki impiega wikilink canonici e generati deterministicamente. La scelta di usare path completi evita ambiguità tra pagine che condividono lo stesso slug, ad esempio un article e una summary sulla medesima vulnerabilità:

```
[[wiki/articles/sql-injection|sql-injection]]
[[wiki/summaries/sql-injection|sql-injection]]
```

Il processo di lint verifica quattro proprietà:

1. **broken links**: un wikilink punta a una pagina inesistente;
2. **ambiguous links**: uno slug breve può riferirsi a più pagine;
3. **stub pages**: una pagina è troppo breve o incompleta;
4. **orphans**: una pagina conoscitiva non è raggiunta da alcun collegamento in ingresso.

I link editoriali generati liberamente dal modello sono una fonte di errori strutturali, poiché il modello può citare framework, tool, varianti di payload o tecniche estremamente specifiche senza che esista una pagina corrispondente. Per questa ragione, nel progetto i link strutturali tra article, summary e concept vengono aggiunti dal codice soltanto verso pagine effettivamente esistenti o create durante lo stesso merge. Un esempio di link e visualizzazione direttamente su Obsidian è possibile vederla nella figura 4.4

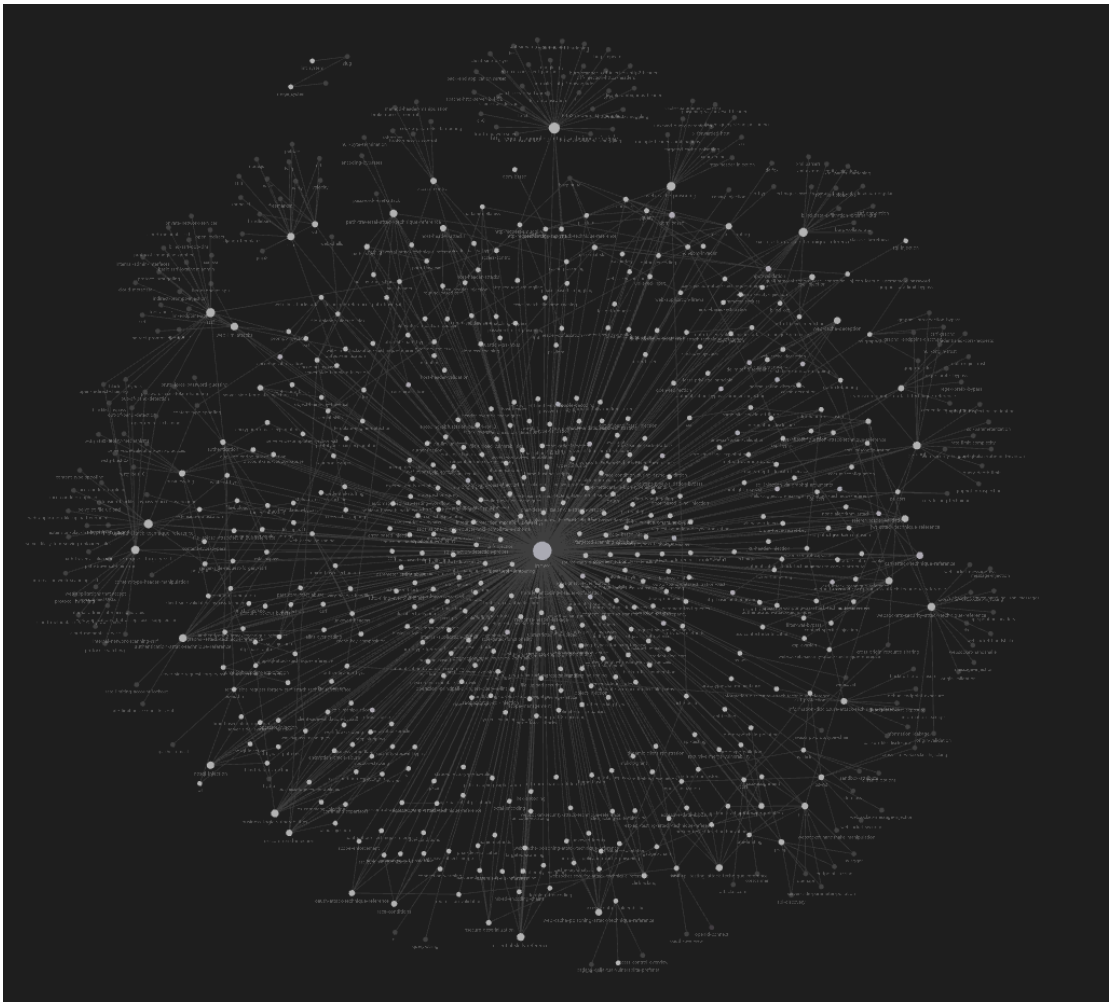


Figura 4.4: Obsidian graph.

4.7 Memory

Nel capitolo di implementazione meritano particolare attenzione tre aspetti della memoria: il contesto mantenuto durante la singola sessione, il recupero degli attacchi confermati attraverso l'*exact replay* e la rappresentazione persistente dello stato dell'esplorazione. Questi meccanismi operano su orizzonti temporali differenti, ma condividono lo stesso obiettivo: evitare la ripetizione di tentativi già eseguiti e rendere disponibili alle fasi successive le informazioni che hanno prodotto un risultato utile. In termini di memoria di contesto vera e propria, non sono emerse particolari criticità anche grazie all'elevato numero di token accettati in input, come descritto nella sezione introduttiva 2.3 dei modelli utilizzati.

4.7.1 Memoria di sessione

La memoria di sessione viene fornita direttamente al modello e conserva i tentativi già eseguiti insieme ai relativi esiti. In modalità `attack`, queste informazioni permettono di evitare la riproposizione dello stesso payload e di esplorare strategie differenti all'interno della medesima famiglia di vulnerabilità. Il comportamento è visibile nei log, dove il campo `reason` può motivare il passaggio da un tentativo fallito a una tecnica alternativa; ad esempio, dopo una SQL injection booleana priva di risultati, il Planner può scegliere un payload basato su `UNION` per verificare se sia possibile estrarre dati dalla tabella `users`, seguendo anche le indicazioni recuperate dalla Wiki.

```
"reason": "The previous boolean-based attempt (' OR '1'='1)
failed and returned no results. Switching to a UNION-based
extraction payload to check if data can be retrieved from
the 'users' table, as suggested by the wiki guidance for
this application."
```

In modalità `discovery`, la stessa memoria svolge una funzione leggermente diversa. Il sistema conserva e trasferisce alle interazioni successive i valori già utilizzati e confermati da una transizione utile, cioè quelli che hanno consentito un avanzamento nel workflow applicativo. I valori che non hanno prodotto alcun cambiamento non vengono invece trattati come informazioni affidabili. In questo modo il modello può proseguire la navigazione a partire dai risultati già acquisiti senza ripetere combinazioni inefficaci, come mostrato nella figura 4.5.



```
"confirmed_values": {
  "scenario": {
    "value": "startup",
    "source": "confirmed_navigation_submit",
    "scope_url": "http://semantic-lab:8080/profile",
    "sensitivity": "normal"
  },
  "entity_name": {
    "value": "ACME Labs Europe",
    "source": "confirmed_navigation_submit",
    "scope_url": "http://semantic-lab:8080/details",
    "sensitivity": "normal"
  },
  "contact_email": {
    "value": "team@example.org",
    "source": "confirmed_navigation_submit",
    "scope_url": "http://semantic-lab:8080/details",
    "sensitivity": "normal"
  },
  "country": {
    "value": "Italy",
    "source": "confirmed_navigation_submit",
    "scope_url": "http://semantic-lab:8080/details",
    "sensitivity": "normal"
  }
}
```


Figura 4.5: Conservazione e riutilizzo dei valori confermati dalla memoria di sessione durante la discovery.

4.7.2 Try exact replay

Quando l'esecuzione avviene in modalità `attack` , Redis è disponibile, la flag di replay è abilitata e lo storico contiene almeno un attacco con esito positivo per la stessa *surface signature*, l'Orchestrator tenta l'*exact replay* attraverso il metodo `try_exact_replays` . Tutte le modalità di funzionamento sono state descritte nella sezione 4.8.3.

In questo flusso vengono riutilizzati il piano precedentemente validato, gli stessi valori dei campi e la stessa azione finale di invio che avevano condotto alla conferma della vulnerabilità. Analyst e Planner non vengono invocati in modo da verificare in modo efficiente se una vulnerabilità nota sia ancora riproducibile nella versione corrente dell'applicazione.

Il Browser Executor applica il piano recuperato dalla memoria e l'Evaluator analizza lo stato risultante. Se l'esito viene nuovamente confermato, l'Orchestrator termina immediatamente la run. Non viene quindi eseguito il successivo controllo della cache, non viene avviato il ciclo completo basato su LLM e non viene scritta una nuova cache

per la superficie. Il nodo viene aggiornato nel grafo persistente mediante associandolo alla vulnerabilità rilevata.

Questa scelta consente di verificare rapidamente vulnerabilità già osservate, riducendo il numero di invocazioni LLM e il tempo necessario per il regression testing. Il replay non costituisce una conferma automatica: il piano viene sempre rieseguito nell'ambiente corrente e l'Evaluator rivaluta l'esito sulla base delle evidenze effettivamente raccolte.

Un attacco precedentemente efficace può non essere più confermato nella nuova versione dell'applicazione, ad esempio perché la vulnerabilità è stata corretta o perché sono cambiate le condizioni necessarie alla sua riproduzione. Se tutti gli attacchi storici confermati vengono rigiocati senza ottenere una nuova conferma da parte dell'Evaluator, il metodo `_try_exact_replays` non conclude la run.

Il controllo viene restituito al flusso ordinario insieme all'insieme `already_replayed_candidates` , che identifica i candidati già sottoposti a replay. L'Orchestrator procede quindi con il controllo della cache: in caso di hit riusa il risultato ed esegue solamente la discovery; in caso di miss avvia la normale analisi della superficie.

Il fallimento del replay non impedisce comunque la ricerca di vulnerabilità differenti o l'esplorazione di strategie alternative sulla medesima superficie. Ricordiamo che l'individuazione delle vulnerabilità è dovuta a numerosi fattori, sia di configurazione come ad esempio la scelta dei parametri di eplorazione e il tempo a disposizione, sia riguardanti le capacità del modello e della conoscenza a disposizione.

```
[SESSION_BRIDGE] sessione 'llm_authenticated' creata per
                 http://juice-shop:3000
[SESSION_BRIDGE] verifica: sessione attiva su
                 http://juice-shop:3000 = 'llm_authenticated'
                 (3/3 cookie propagati)
[REPLAY][EXACT] lookup | target_version=5
                | surface_signature=
                  8cc8bd9deaae4879f7f85ddc8dae32ea5c67fb02d54fa89258
                  ae2b5c0f47b70f
                | matches=1
[REPLAY][EXACT] historical match | source_version=4 |
                candidate='Reflected Cross-Site Scripting via
                           Search Field'
                | success=True | submitted=True | values=1
                | verdict='confirmed'
[REPLAY][EXACT] plan restored | source_version=4 | target_version=5 |
                candidate='Reflected Cross-Site Scripting via
                           Search Field'
                | submitted=True
                | field_values=[{
                    'field_ref': 'field_search_close_0',
                    'value': '"><img src=x onerror=alert(1)>'
                  }]
[EXECUTOR] submit activation | method=enter-no-owner-form
           | field_ref=field_search_close_0
```

```
[BROWSER] Dialog intercepted | type=alert | message='1'
[BROWSER] Attempt network capture stopped |
        attempt_id=discovered::
        b5deccddb2f0e2ecf785c1f67ae084bfa459b56fe2b9e3b7c500557841
        fdfdd1:exact-replay:0:
        Reflected Cross-Site Scripting via Search Field
        | transactions=2 | failures=0

EVALUATOR exact-replay evidence
candidate='Reflected Cross-Site Scripting via Search Field'
{
  "execution": {
    "submitted": true,
    "apply_failed": false,
    "error_message": null
  },
  "navigation": {
    "url_before": "http://juice-shop:3000/#/",
    "url_after":
      "http://juice-shop:3000/#/search?q=%22%3E%3Cimg%20src%3Dx
       %20onerror%3Dalert(1)%3E",
    "url_changed": true,
    "route_before": "/",
    "route_after":
      "/search?q=%22%3E%3Cimg%20src%3Dx%20onerror%3Dalert(1)%3E",
    "route_changed": true
  },
  "surface": {
    "state_hash_before":
      "55c05edb5b2139213c5927af50ca3308061caa37bc45df3a49ae04f94
       a4ebe52",
    "state_hash_after":
      "85205cd06ae2acf728876306dd1eaebe096770ae08f923afb14a81241c7
       9c92e",
    "state_changed": true,
    "title_before": "OWASP Juice Shop",
    "title_after": "OWASP Juice Shop",
    "title_changed": false,
    "text_changed": true,
    "dom_diff": {
      "added_controls": [],
      "removed_controls": [
        {
          "tag": "button",
          "type": "",
          "text": "add_shopping_cart\nAdd to Basket",
          "id": "",
          "name": "",
          "href": "",
```

```
          "visible": true,
          "disabled": false
        }
      ],
      "text_changed": true,

      ...

  },
  "messages": {
    "dialogs": [
      "alert:1"
    ],
    "new_errors": [],
    "removed_errors": [],
    "page_errors_after": []
  },
  "cookies": {
    "added": [],
    "removed": [],
    "changed": [],
    "session_changed": false
  },
  "network": {
    "capture_available": true,

...

[LLM DEBUG] System Prompt
[LLM DEBUG] "You are a post-execution security evaluator for
            authorized web\napplication testing.\n\nYou receive:
            \n- candidate: the hypothesis tested;
            \n- before_state: deterministic browser surface before
            execution;
            \n- after_state: deterministic browser surface after
            execution;
            \n- observed_effects: raw browser execution observations;

...

[LLM DEBUG] Parsed Output
[LLM DEBUG] {
[LLM DEBUG]   "success": true,
[LLM DEBUG]   "verdict": "confirmed",
[LLM DEBUG]   "confidence": 1.0,
[LLM DEBUG]   "vuln_name": "Reflected Cross-Site Scripting via
                           Search Field",
[LLM DEBUG]   "signals": [
[LLM DEBUG]     {
```

```
[LLM DEBUG]       "kind": "dialog_observed",
[LLM DEBUG]       "message": "A browser alert dialog containing
[LLM DEBUG]                   '1' was observed, directly
[LLM DEBUG]                   corresponding to the payload
[LLM DEBUG]                   alert(1)."
[LLM DEBUG]     },
[LLM DEBUG]     {
[LLM DEBUG]       "kind": "input_reflection",
[LLM DEBUG]       "message": "The search query \"><img src=x
[LLM DEBUG]                   onerror=alert(1)> was reflected in
[LLM DEBUG]                   the page text as
[LLM DEBUG]                   'Search Results - \">'."
[LLM DEBUG]     }
[LLM DEBUG]   ],
[LLM DEBUG]   "evidence_quality": "strong",
[LLM DEBUG]   "assessment_basis": [
[LLM DEBUG]     "The payload \"><img src=x onerror=alert(1)> was
[LLM DEBUG]      submitted via the search field.",
[LLM DEBUG]     "The browser observed a dialog event with the
[LLM DEBUG]      message 'alert:1', which matches the payload's
[LLM DEBUG]      execution goal.",
[LLM DEBUG]     "The 'after_state' URL shows the payload reflected
[LLM DEBUG]      in the 'q' parameter.",
[LLM DEBUG]     "The 'after_state' page text shows the beginning
[LLM DEBUG]      of the payload reflected in the 'Search Results'
[LLM DEBUG]      heading, confirming the injection point."
[LLM DEBUG]   ],
[LLM DEBUG]   "missing_evidence": [],
[LLM DEBUG]   "recommended_oracles": [
[LLM DEBUG]     "dialog whose content is attributable to the tested marker"
[LLM DEBUG]   ],
[LLM DEBUG]   "next_action": "stop",
[LLM DEBUG]   "notes": "The execution of the alert(1) dialog
[LLM DEBUG]             provides definitive proof of reflected XSS
[LLM DEBUG]             in the search functionality."
[LLM DEBUG] }
[ZAP][LLM_ALERT][EXACT_REPLAY] raised=True
  | url=http://juice-shop:3000/#/search?q=%22%3E%3Cimg%20src%3Dx
        %20onerror%3Dalert(1)%3E
  | candidate='Reflected Cross-Site Scripting via Search Field'
```

4.7.3 Grafico persistente

Il grafo persistente (mostrato in figura 4.6) offre una rappresentazione sintetica dello stato dell'esplorazione, mostrando le risorse individuate e l'esito dell'analisi di ciascun nodo. I colori ne rendono immediata la lettura: il grigio indica una risorsa scoperta ma non ancora analizzata, il verde una risorsa analizzata senza vulnerabilità confermate, il giallo un risultato sospetto o inconclusivo e il rosso una vulnerabilità confermata. Nel

rettangolo rappresentante il nodo sono stati inseriti anche la vulnerabilità trovata e l'url associato. Questa parte è stata già descritta nella sezione 3.8.2.

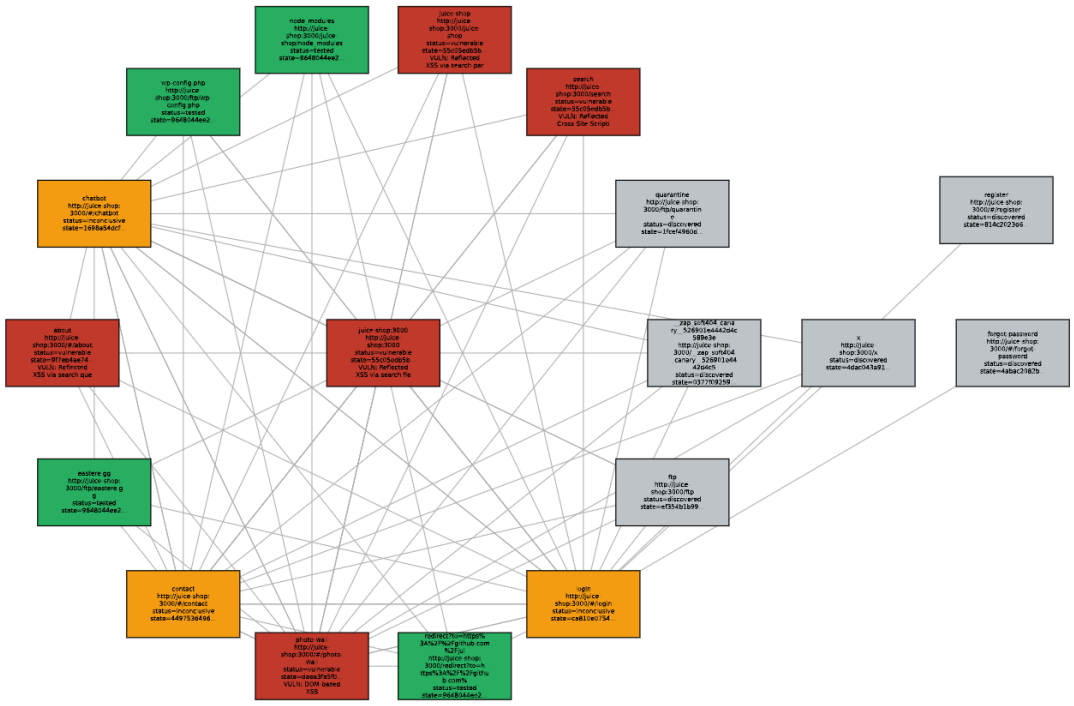


Figura 4.6: Grafo generato in fase di attacco.

4.8 Orchestrator

Come definito nella rispettiva sezione di progettazione, l'Orchestrator coordina l'intero workflow. A questo scopo vengono considerati, in ordine, la presenza di una superficie d'attacco, l'eventuale disponibilità di attacchi confermati nello storico, la presenza di un risultato riutilizzabile in cache e, infine, la necessità di avviare il ciclo completo basato su LLM.

La persistenza è affidata a Redis ed è controllata dalla variabile `REDIS_ENABLED` importata dal file di config descritto nella sezione di bootstrap 4.1.1. Quando Redis è attivo, l'`AttackRecordStore` conserva sia gli esiti associati alle superfici già analizzate sia i record degli attacchi eseguiti. Queste informazioni svolgono funzioni distinte: la cache permette di non ripetere un'analisi già completata, mentre lo storico degli attacchi consente di verificare se una vulnerabilità confermata in precedenza sia ancora presente in una nuova versione dell'applicazione.

Il replay cross-version è controllato dalla variabile `RETRY_SUCCESSFUL_ATTACKS_ACROSS_VERSIONS`. Quando è abilitato, l'Orchestrator ricerca nello storico un attacco riuscito su una superficie equivalente a quella corrente. Il confronto avviene tramite la

surface signature, la firma deterministica che consente di riconoscere la stessa struttura applicativa anche tra esecuzioni o versioni differenti.

4.8.1 MaxCandidates e Budget

Un aspetto rilevante, non approfondito nella fase di progettazione, riguarda i criteri di arresto dell'esecuzione. In assenza di limiti, la scansione potrebbe infatti proseguire indefinitamente, tentando strategie differenti sulla stessa superficie. Per controllare il numero di iterazioni sono stati introdotti due iperparametri: `MAX_CANDIDATES` , che stabilisce il numero massimo di famiglie di vulnerabilità da esaminare, e `BUDGET` , che definisce quanti tentativi possono essere dedicati a ciascuna famiglia. L'esplorazione è inoltre soggetta a un limite temporale e al parametro `MAX_DEPTH` , che interrompe l'avanzamento quando il grafo conservato nella memoria persistente raggiunge la profondità massima configurata. Questi vincoli impediscono esecuzioni indefinite e consentono di mantenere prevedibili il tempo e le risorse impiegati dalla scansione.

4.8.2 Assenza di superficie d'attacco

Il primo controllo riguarda la superficie deterministica estratta dalla pagina. Se la funzione `has_attack_surface(deterministic_surface)` (appartenente al DEL) restituisce valore falso, ad esempio per una pagina statica priva di form, campi compilabili o altri elementi interattivi rilevanti, il nodo viene classificato come `no_attack_surface` .

In questo caso non vengono invocati Analyst, Planner, Browser Executor o Evaluator, poiché non esistono elementi sui quali formulare ed eseguire un attacco. L'Orchestrator avvia soltanto la discovery di ZAP a partire dal nodo corrente, così da individuare eventuali collegamenti e nuove risorse raggiungibili, quindi conclude la run senza memorizzare un risultato nella cache. Tale comportamento precede ogni controllo su Redis, replay e modalità operativa: in assenza di una superficie attaccabile, tutte le relative combinazioni di configurazione conducono allo stesso esito.

4.8.3 Influenza della cache e del replay

La tabella 4.1 riassume il comportamento dell'Orchestrator in funzione della disponibilità di Redis, dell'abilitazione del replay e dello stato presente nello storico.

Tabella 4.1: Comportamento dell'Orchestrator in funzione di cache, persistenza e replay cross-version.

REDIS_ ENABLED	RETRY_ SUCCESSFUL_ ATTACKS_ ACROSS_ VERSIONS	Stato storico/cache	Comportamento effettivo
0	False	Non disponibile	Non viene utilizzata alcuna cache né viene persistito lo storico degli attacchi. Ogni esecuzione parte da zero e segue il ciclo completo: Analyst → Planner → Browser Executor → Evaluator.
0	True	Non disponibile	L'`AttackRecordStore` viene creato senza backend di cache e risulta vuoto a ogni avvio. La flag è formalmente attiva, ma non esistono dati storici da rigiocare; il comportamento coincide con il caso precedente.
1	False	Cache hit	L'attacco viene saltato. Il sistema riusa il risultato memorizzato e avvia soltanto la discovery di ZAP per individuare ulteriori risorse raggiungibili dal nodo.
1	False	Cache miss	Non essendo disponibile un risultato riutilizzabile, viene avviato il ciclo completo: Analyst → Planner → Browser Executor → Evaluator.
1	True	Nessun bug storico, cache hit	Il sistema riusa il risultato in cache e svolge soltanto la discovery, analogamente al caso con replay disabilitato.
1	True	Nessun bug storico, cache miss	Viene avviato il ciclo completo di analisi e attacco, analogamente al caso con replay disabilitato.
1	True	Bug storico, replay riuscito	Analyst e Planner vengono saltati. Il piano precedentemente confermato viene rieseguito nel browser e l'Evaluator ne verifica l'esito. Se la vulnerabilità è ancora presente, la run termina immediatamente senza scrivere una nuova cache della superficie.
1	True	Bug storico, replay fallito, cache hit	Dopo il replay fallito, il sistema non riavvia il ciclo di attacco: riusa il risultato disponibile in cache ed esegue solamente la discovery.
1	True	Bug storico, replay fallito, cache miss	Dopo il replay fallito, viene eseguito il ciclo completo. I candidati già rigiocati sono esclusi dal soft-retest, evitando la duplicazione dei tentativi nella stessa run.

Le combinazioni riportate in tabella possono essere ricondotte a tre scenari principali. Se `REDIS_ENABLED=0` , non sono disponibili né la cache né uno storico persistente. Ogni superficie attaccabile deve quindi essere analizzata da zero mediante il ciclo completo. In questa configurazione il valore di `RETRY_ SUCCESSFUL_ ATTACKS_ ACROSS_ VERSIONS` è ininfluente: l'opzione può essere abilitata senza causare errori, ma l'`AttackRecordStore` viene ricreato vuoto a ogni avvio e non contiene alcun attacco da rigiocare.

Se Redis è attivo ma il replay è disabilitato, la decisione dipende unicamente dalla cache. In caso di *cache hit*, l'Orchestrator considera già disponibile l'esito dell'analisi, evita nuove invocazioni dei modelli e svolge soltanto la discovery di ZAP. In caso di *cache miss*, invece, non esiste un risultato riutilizzabile e viene eseguito l'intero ciclo di analisi, esecuzione e valutazione.

Quando sono attivi sia Redis sia il replay cross-version, lo storico degli attacchi viene controllato prima di decidere se avviare una nuova analisi. Se non esiste un attacco precedentemente confermato per la stessa *surface signature*, il comportamento torna a dipendere dalla cache: un hit determina il riuso del risultato e la sola discovery, mentre un miss porta al ciclo completo.

Se viene trovato un attacco storico, il relativo piano viene eseguito nuovamente nel browser senza richiamare Analyst e Planner. L'Evaluator esamina lo stato ottenuto dopo l'esecuzione per verificare se la vulnerabilità sia ancora presente. Un replay riuscito conferma il finding regressivo e conclude immediatamente la run, senza produrre una nuova voce di cache per la superficie. Un replay fallito non implica invece che la superficie sia sicura: l'Orchestrator prosegue applicando la normale logica di cache. In presenza di un cache hit riusa il risultato disponibile ed esegue soltanto la discovery; con un cache miss avvia il ciclo completo. In quest'ultimo caso, i candidati già sottoposti a replay vengono esclusi dal successivo soft-retest, evitando di ripetere lo stesso tentativo nella medesima run.

Il funzionamento dettagliato della funzione replay è disponibile nella sezione 4.7.2

CAPITOLO 5

Risultati

Il capitolo presenta i risultati ottenuti con ZAPilot e confronta il sistema con una scansione DAST tradizionale. L'analisi considera sia l'incremento delle vulnerabilità individuate dopo l'esplorazione guidata, sia il comportamento dei modelli linguistici impiegati. Il confronto non si limita al numero di segnalazioni, ma comprende falsi positivi, tempo di esecuzione, token elaborati e capacità del modello di interpretare il contesto e produrre azioni eseguibili.

5.1 Fase di discovery

Per valutare la fase di *discovery* è stato realizzato un servizio ad hoc, caratterizzato da un percorso di navigazione composto da form articolati e dipendenti dalle interazioni precedenti. Il completamento corretto di tale percorso consente di raggiungere aree dell'applicazione che espongono informazioni sensibili. In particolare, per ottenere il codice richiesto dalla challenge presente al quarto passaggio (d) visibile in figura 5.2, il modello deve applicare la seguente regola: utilizzare le iniziali delle prime due parole del campo *Entity name*, le ultime due lettere del campo *Country* (entrambi presenti nel secondo passaggio della figura 5.2) e il numero di parole contenute nel campo *Department* (terzo passaggio della figura 5.2). In base a questi valori si riesce a raggiungere aree altrimenti non raggiungibili.

5.1.1 Senza ZAPilot

In assenza di ZAPilot, la scansione riesce a esplorare soltanto una parte dell'applicazione. Come possibile vedere in figura 5.1 infatti, ZAP tenta di compilare i form mediante valori predefiniti, senza interpretare il significato dei campi né utilizzare il contesto ricavato dalle pagine e dalle interazioni precedenti. Di conseguenza, quando

l'avanzamento richiede input coerenti con lo stato della sessione, la navigazione automatica si arresta e le aree successive non vengono incluse nella superficie sottoposta a scansione.

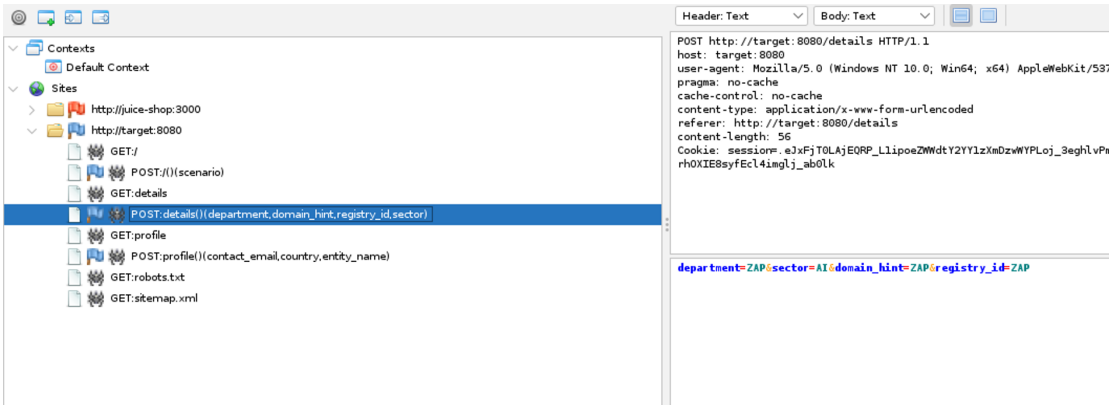


Figura 5.1: Input senza contesto di ZAP.

5.1.2 Con ZAPilot

Con ZAPilot, il modello utilizza invece il contesto della pagina e la memoria della sessione per interpretare i campi, generare valori compatibili con i vincoli del form e mantenere la coerenza tra i diversi passaggi. La sequenza di interazioni mostrata nella figura 5.2 documenta l'avanzamento nel workflow; il confronto riportato nella figura 5.3 evidenzia l'ampliamento della superficie raggiunta al termine dell'esplorazione assistita, riuscendo a vedere parti dell'applicazione e file privati che altrimenti non sarebbero stati scoperti con la singola configurazione DAST. In questo modo è stato possibile anche alzare la flag direttamente in ZAP segnalando il problema.



(a) Primo passaggio.



(b) Secondo passaggio.



(c) Terzo passaggio.



(d) Quarto passaggio.



(e) Completamento del percorso.

Figura 5.2: Sequenza delle interazioni eseguite da ZAPilot durante la fase di discovery.

5.2 Fase di attack

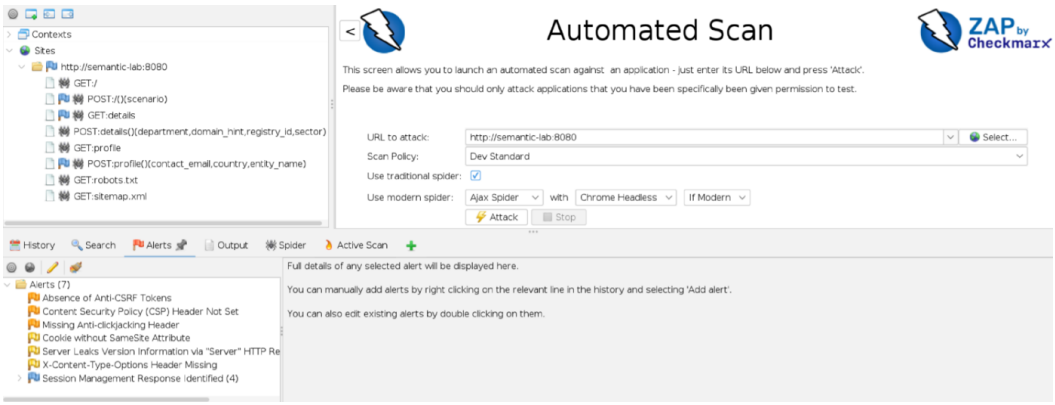
5.2.1 Senza ZAPilot

Come riportato nel capitolo introduttivo, nella figura 1.1, la scansione automatica con la modalità più aggressiva presente in BurpSuite Professional non ha riscontrato alcun tipo di vulnerabilità certa in JuiceShop. Allo stesso modo, sono stati fatti numerosi test anche con ZAP. I risultati sono visibili in figura 5.4. L'unica vulnerabilità trovata è una SQL injection che però deve essere confermata a mano. Infatti è possibile vedere come la confidenza sia bassa, e l'unica prova visibile sia l'errore nella risposta http fornita.

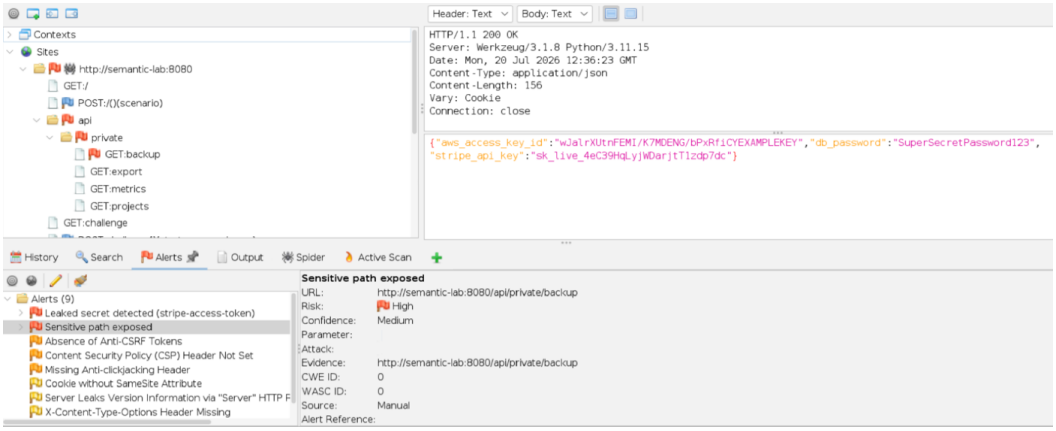
5.2.2 Con ZAPilot

I risultati ottenuti con ZAPilot invece mostrano un evidente miglioramento nell'individuazione di vulnerabilità certe[1], poichè accompagnate da evidenze deterministiche. Di seguito sono riportati alcuni casi rappresentativi, distinguendo le vulnerabilità supportate da evidenze osservabili dalle segnalazioni che, dopo una verifica manuale, sono risultate falsi positivi.

[1]La presenza di evidenze deterministiche conferma che l'evento osservato si è effettivamente verificato, ma non implica necessariamente l'esistenza di una vulnerabilità reale o dannosa. Ad esempio, nel caso del presunto bypass del CAPTCHA, il sistema ha completato correttamente una challenge semplice ed è avanzato nel flusso applicativo, interpretando però tale esito come l'aggiramento di un controllo di sicurezza.



(a) Discovery prima dell'intervento di ZAPilot.



(b) Discovery dopo l'intervento di ZAPilot.

Figura 5.3: Confronto della superficie dell'applicazione raggiunta prima (a) e dopo (b) l'esplorazione guidata.

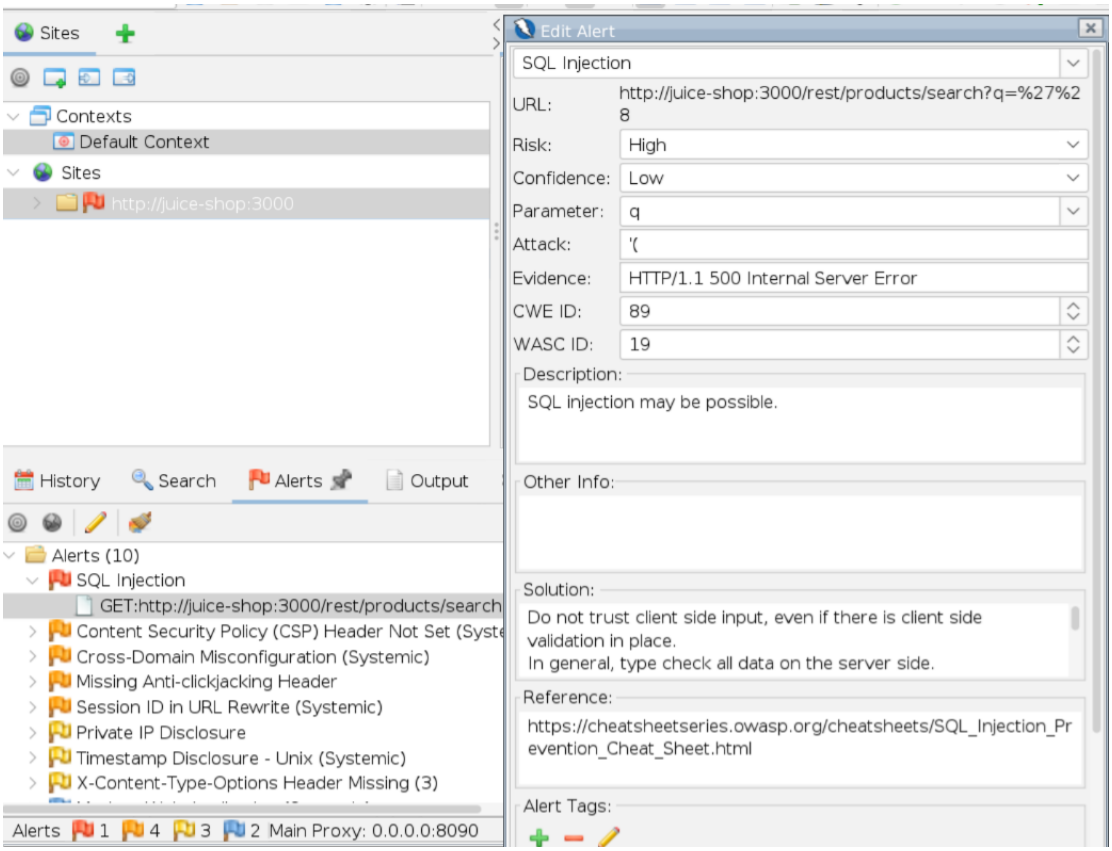
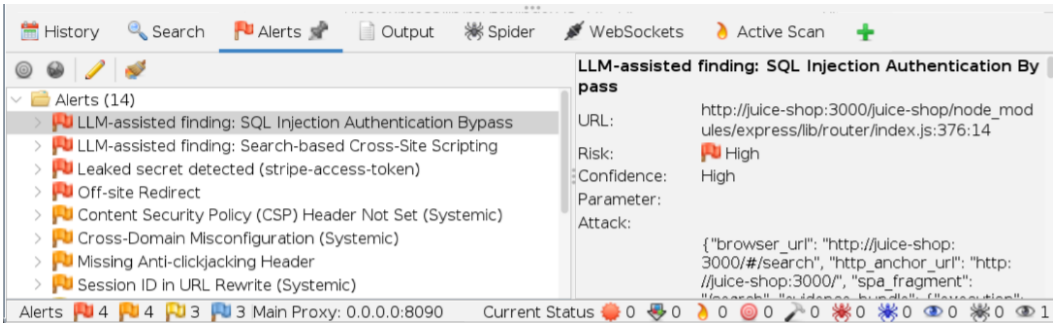
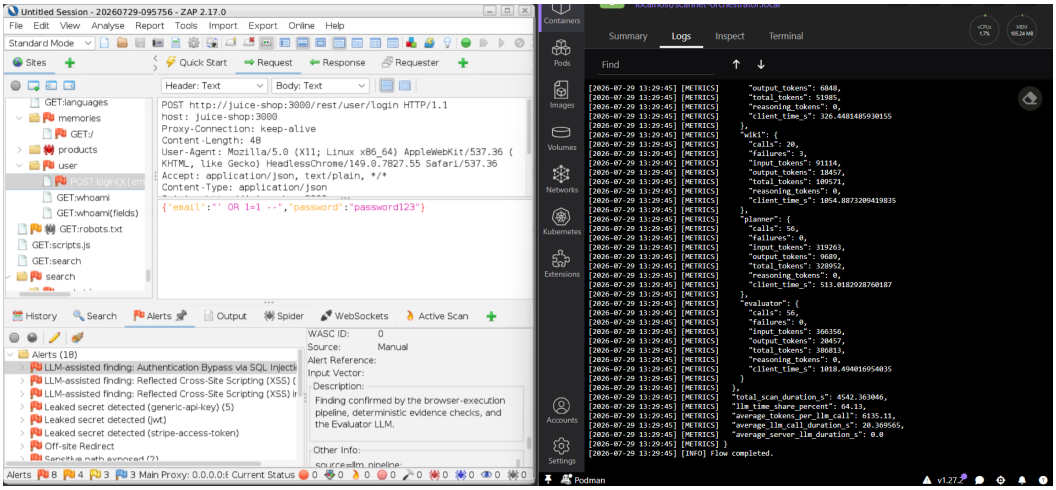


Figura 5.4: Attacco eseguito con la sola scansione automatica di ZAP, senza l'utilizzo di ZAPilot. Trova solo una possibile SQL injection con confidenza low.

SQL injection con bypass dell'autenticazione. La figura 5.5 mostra sia l'alert generato da ZAPilot, classificato con rischio e confidenza elevati, sia la richiesta HTTP utilizzata per verificarlo. Nel corpo della richiesta di login, il valore del campo `email` contiene il payload `' OR 1=1 --` , impiegato per alterare la condizione della query e aggirare il controllo delle credenziali. In questo caso la segnalazione non deriva soltanto dalla presenza di un errore nella risposta HTTP, come nella scansione tradizionale, ma dall'esecuzione del tentativo e dall'osservazione del bypass dell'autenticazione.



(a) Alert prodotto da ZAPilot per il bypass dell'autenticazione.



(b) Richiesta di login contenente il payload SQL e riscontro della scansione.

Figura 5.5: Evidenze relative alla SQL injection utilizzata per aggirare l'autenticazione.

Presunto bypass del CAPTCHA. La segnalazione mostrata nella figura 5.6 è stata classificata dal sistema come un bypass del CAPTCHA ottenuto mediante la manipolazione dell'input. La successiva verifica manuale ha tuttavia evidenziato che si trattava di un falso positivo: il corretto invio del form non dimostrava che il controllo fosse stato aggirato, ma soltanto che la challenge era stata risolta. In Juice Shop, infatti, il CAPTCHA consiste in una semplice operazione aritmetica, anziché in uno dei più comuni

sistemi basati su immagini o puzzle. L'evento osservato era quindi reale, ma la sua interpretazione non era corretta: l'avanzamento nel flusso applicativo rappresentava il comportamento previsto dell'applicazione e non una vulnerabilità di sicurezza.

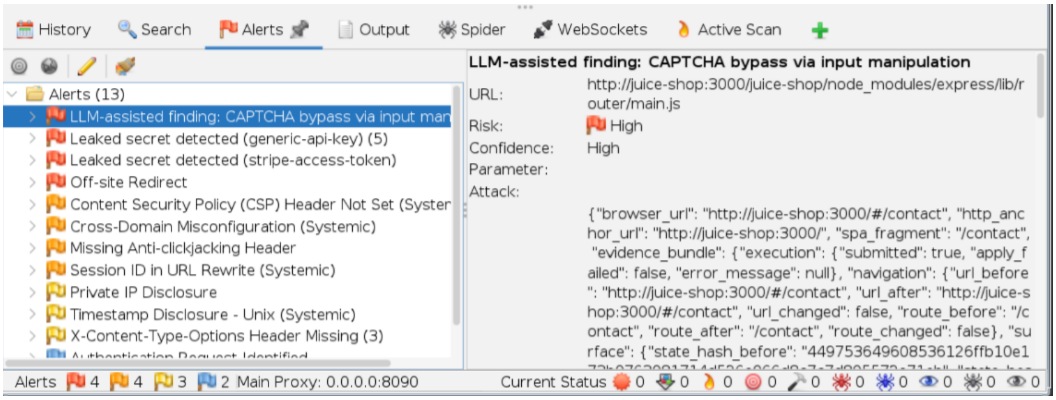


Figura 5.6: Segnalazione di un presunto bypass del CAPTCHA, risultata un falso positivo in seguito alla verifica.

Redirect verso una destinazione esterna. La figura 5.7 riporta infine la segnalazione relativa all'endpoint `/redirect` , nel quale il parametro `to` contiene un URL esterno. La risposta di reindirizzamento porta il browser verso la destinazione indicata senza che questa appartenga necessariamente all'applicazione. Un comportamento di questo tipo può essere sfruttato per costruire collegamenti apparentemente affidabili che conducono l'utente a un sito controllato da un attaccante; per questo ZAP associa all'alert un rischio elevato, pur indicando una confidenza media.

Tentativo di forza bruta con credenziali deboli. Nel corso delle scansioni, Gemma-4-31B-IT ha provato diverse combinazioni di credenziali fino a ottenere, in una delle esecuzioni, l'accesso con l'indirizzo `admin@juice-sh.op` e la password `admin123` . Il risultato evidenzia sia la debolezza della password utilizzata sia la possibilità di effettuare tentativi ripetuti sul form di autenticazione. L'accesso riuscito costituisce un'evidenza osservabile dell'efficacia del tentativo; tuttavia, poiché l'esito non si è ripetuto con regolarità, non consente da solo di stabilire se la combinazione sia stata individuata attraverso l'esplorazione, suggerita dalla wiki fornita al planner o richiamata dalle conoscenze pregresse del modello. Il caso, insieme alla relativa evidenza, è approfondito nella Sezione 5.7.

Esposizione di informazioni sensibili nella cartella FTP. ZAPilot ha inoltre individuato un file accessibile pubblicamente nella cartella `ftp` di Juice Shop. La segnalazione è stata favorita dall'introduzione, durante la prova, di un file contenente stringhe riconoscibili come segreti, che ha consentito al controllo dedicato di rilevare il contenuto e di ricondurlo a una possibile esposizione di informazioni sensibili. Nel contesto della

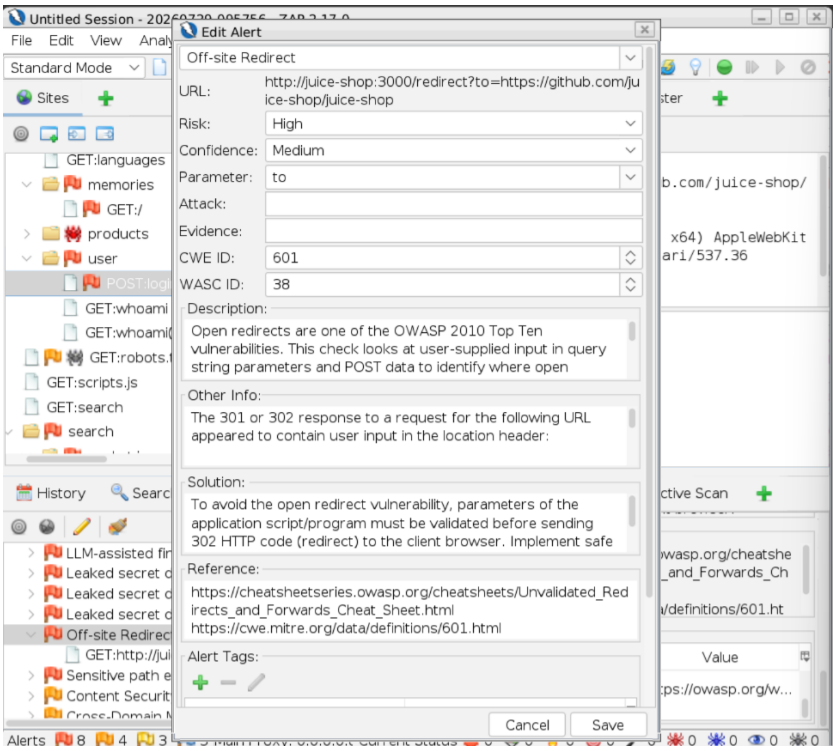


Figura 5.7: Segnalazione del redirect verso un URL esterno controllato tramite il parametro `to`.

challenge di Juice Shop, tuttavia, il documento esposto contiene soltanto testo *Lorem ipsum*: si tratta quindi di un contenuto fittizio predisposto per simulare la vulnerabilità, non di dati riservati reali. Rimane comunque significativo il fatto che il sistema abbia raggiunto la directory e identificato una risorsa che, in un'applicazione reale, potrebbe contenere credenziali, token o documenti confidenziali.

Reflected Cross-Site Scripting nel campo di ricerca. Come ultimo caso, il seguente estratto riporta i log prodotti da ZAPilot in seguito a un attacco concluso con successo. Il sistema conferma una vulnerabilità di tipo *Reflected Cross-Site Scripting* (XSS) nel campo di ricerca.

```
[LLM DEBUG]   "success": true,
[LLM DEBUG]   "verdict": "confirmed",
[LLM DEBUG]   "confidence": 0.97,
[LLM DEBUG]   "vuln_name": "Reflected Cross-Site Scripting via Search Field",
[LLM DEBUG]   "signals": [
[LLM DEBUG]     {
[LLM DEBUG]       "kind": "submit_observed",
[LLM DEBUG]       "message": "The search form was submitted with the payload."
[LLM DEBUG]     },
[LLM DEBUG]     {
[LLM DEBUG]       "kind": "url_changed",
[LLM DEBUG]       "message": "URL includes the injected query parameter."
[LLM DEBUG]     },
[LLM DEBUG]     {
[LLM DEBUG]       "kind": "route_changed",
[LLM DEBUG]       "message": "Client-side route switched to /search with the payload in the query string."
[LLM DEBUG]     },
[LLM DEBUG]     {
[LLM DEBUG]       "kind": "surface_state_changed",
[LLM DEBUG]       "message": "DOM state hash changed and page text reflects the search term."
[LLM DEBUG]     },
[LLM DEBUG]     {
[LLM DEBUG]       "kind": "dialog_observed",
[LLM DEBUG]       "message": "An alert dialog displaying \"1\" was triggered."
[LLM DEBUG]     }
[LLM DEBUG]   ],
[LLM DEBUG]   "evidence_quality": "strong",
[LLM DEBUG]   "assessment_basis": [
[LLM DEBUG]     "Payload value \"><img src=x onerror=alert(1)> appears in the search field after submission.",
[LLM DEBUG]     "The same payload is reflected verbatim in the URL query string.",
[LLM DEBUG]     "Page text changed to show \"Search Results - \">\" indicating reflection of the marker.",
[LLM DEBUG]     "A browser dialog with content \"1\" was observed, directly attributable to the onerror handler execution."
[LLM DEBUG]   ],
[LLM DEBUG]   "missing_evidence": [],
[LLM DEBUG]   "recommended_oracles": [
[LLM DEBUG]     "controlled marker reflected in executable DOM context",
[LLM DEBUG]     "instrumented DOM mutation attributable to the test"
[LLM DEBUG]   ],
[LLM DEBUG]   "next_action": "stop",
[LLM DEBUG]   "notes": "The alert dialog confirms script execution caused by the injected payload,
                        satisfying the reflected XSS hypothesis. No further testing is required
                        for this candidate."
[LLM DEBUG] }
```

I log documentano le evidenze raccolte durante l'esecuzione: l'invio del payload, la modifica dell'URL e della rotta applicativa, la variazione dello stato del DOM e, soprattutto, la comparsa della finestra di dialogo generata dall'esecuzione di `alert(1)` . Quest'ultimo segnale dimostra che il codice JavaScript inserito nel campo di ricerca è stato effettivamente eseguito dal browser. Sulla base di tali riscontri, il modello assegna una confidenza pari a 0,97, classifica l'evidenza come forte e interrompe i tentativi successivi, considerando confermata (`"verdict": "confirmed"`) la vulnerabilità.

5.2.3 Riassunto della fase di attack

La Tabella 5.1 raccoglie i valori complessivi del confronto. Da notare come i valori di ZAPilot rispetto alla baseline non possono diminuire ma solamente aumentare (a meno di vulnerabilità non confermate che vengono validate). Nelle altre casistiche, essendo un add-on che estende le funzionalità di ZAP, i risultati della baseline vengono solo incrementati.

Tabella 5.1: Confronto complessivo tra scansione tradizionale e scansione assistita.

Metrica	Baseline	ZAPilot
Vulnerabilità confermate	0	≈ 7
Vulnerabilità non confermate	1	1
Falsi positivi	0	1
URL o endpoint raggiunti	165	> 3000

Il confronto visivo deve essere interpretato insieme ai dati tabellari. Un numero maggiore di alert, preso isolatamente, potrebbe dipendere da duplicazioni o falsi positivi. Il risultato rilevante consiste invece nella combinazione di tre effetti: ampliamento della superficie raggiunta, individuazione di vulnerabilità prima non osservate e disponibilità di evidenze sufficienti alla loro conferma.

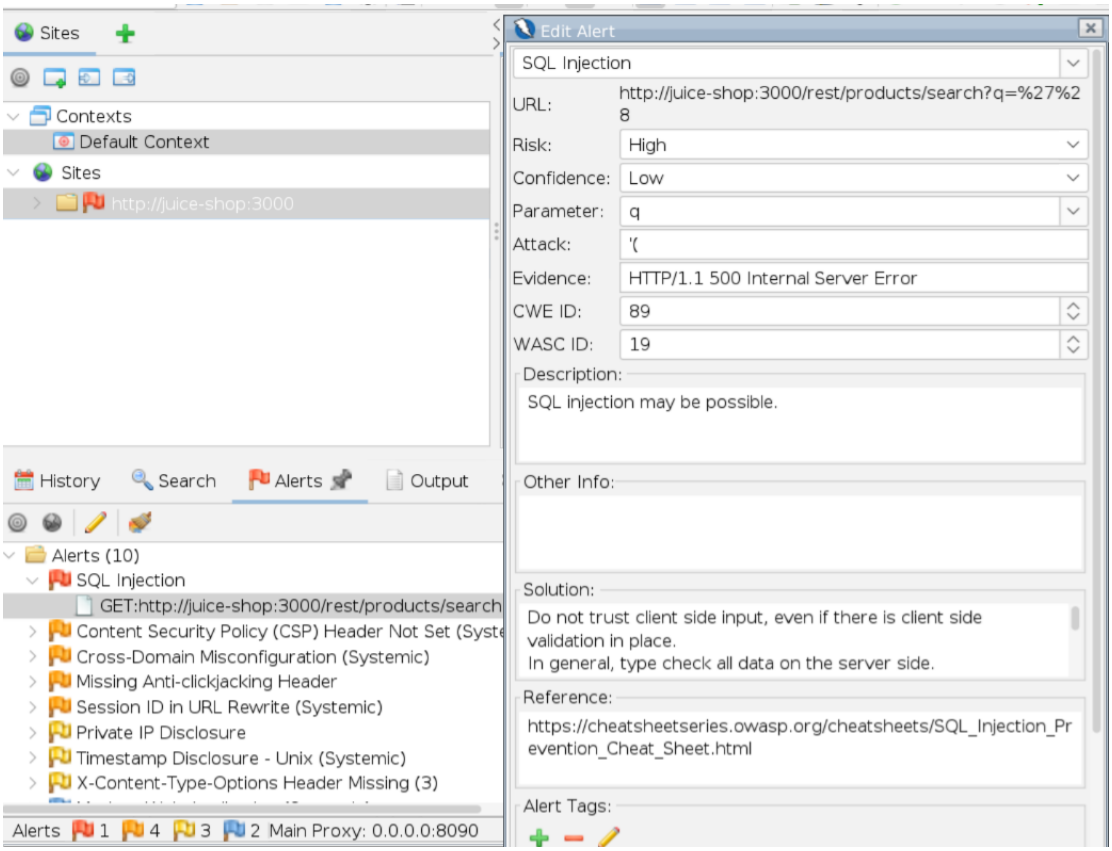
5.3 Confronto sperimentale tra i modelli linguistici

I modelli confrontati sono stati presentati nella sezione 2.3 e comprendono:

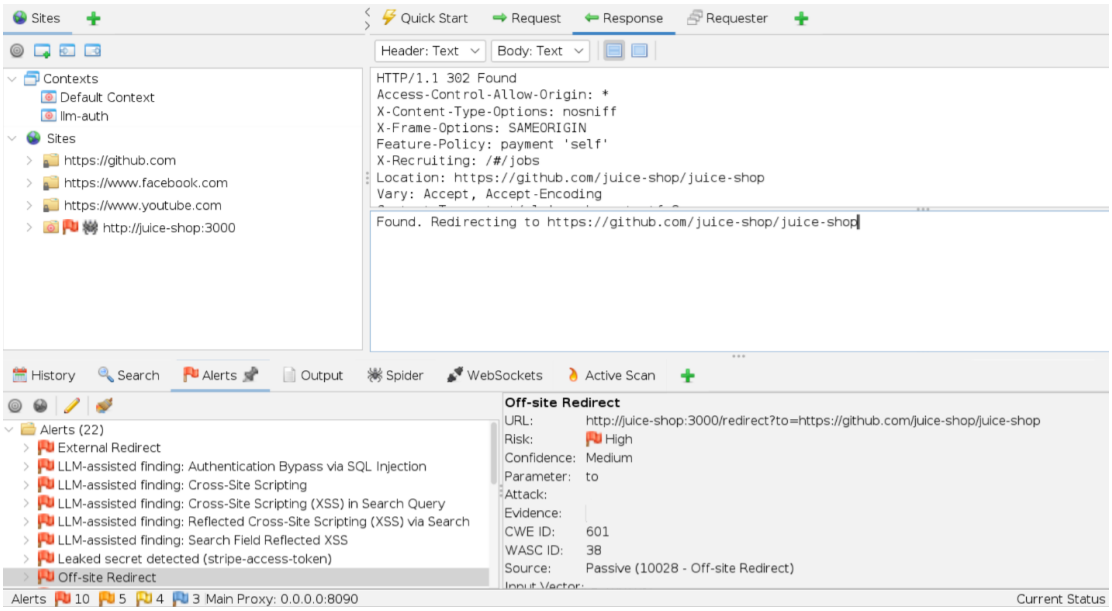
- `qwen3-coder-30b-a3b-instruct-mlx` , nelle scansioni JS_012 e JS_020;
- `openai/gpt-oss-120b` , nelle scansioni JS_014 e JS_019;
- `gemma-4-31b-it` , nelle scansioni JS_011 e JS_018.

Il confronto utilizza sei scansioni completate. Le metriche principali sono il numero di chiamate LLM, il numero di fallimenti, il numero totale di token, la latenza media per chiamata, il tempo complessivo imputabile alle chiamate LLM e la durata end-to-end della scansione.

È importante osservare che i risultati non costituiscono un benchmark strettamente controllato della qualità intrinseca dei modelli. Il numero di chiamate LLM dipende infatti dalla quantità di nodi esplorati, dai candidati di attacco prodotti, dai retry, dalle transizioni applicative raggiunte e dall'evoluzione della pipeline tra le diverse scansioni. La parte non deterministica tra una scansione e l'altra potrebbe cambiare e quindi i risultati come le metriche potrebbero essere differenti. Quello che non cambia son invece i parametri d config, che sono restati identici per tutte quante le scansioni. Redis e di conseguenza il retry degli attacchi storici sono stati mantenuti disattivati.



(a) Dettaglio prima dell'esplorazione guidata.



(b) Dettaglio delle vulnerabilità rilevate successivamente.

Figura 5.8: Confronto delle vulnerabilità dell'applicazione trovate prima (a) e dopo (b) la fase di attacco con ZAPilot.

5.3.1 Risultati per singola scansione

La Tabella 5.2 riporta le metriche aggregate per ciascuna scansione. Il tempo LLM indica la somma dei tempi client-side associati alle richieste al modello; la durata della scansione include invece tutte le componenti della pipeline, tra cui discovery, browser automation, ZAP, persistenza, analisi dei contenuti e chiamate LLM.

Tabella 5.2: Risultati aggregati per singola scansione.

Scan	Modello	Call	Fail	Retry	Input token	Output token	Token totali	Tempo LLM (s)	Durata scan (s)
JS_012	Qwen3-Coder-30B-A3B	226	2	1	710 516	94 144	804 660	2 085.19	3 879.65
JS_020	Qwen3-Coder-30B-A3B	186	0	0	605 120	79 480	684 600	1 820.58	4 394.17
JS_014	GPT-OSS-120B	171	0	5	905 942	81 196	987 138	2 121.04	2 792.86
JS_019	GPT-OSS-120B	149	0	4	802 915	66 377	869 292	1 830.56	4 245.66
JS_011	Gemma-4-31B-IT	143	3	0	821 870	55 451	877 321	2 912.85	4 542.36
JS_018	Gemma-4-31B-IT	105	1	0	604 560	42 223	646 783	2 269.85	4 385.81

La scansione più rapida è JS_014, eseguita con GPT-OSS-120B, con una durata complessiva di 2 792.86 secondi. Tuttavia, la scansione JS_019, eseguita con lo stesso modello, richiede 4 245.66 secondi. Questa differenza mostra che il tempo end-to-end dipende non solo dalla latenza del modello, ma anche dal numero di stati applicativi esplorati, dalle attività di discovery e dall'esecuzione browser.

Le scansioni Qwen3-Coder mostrano una durata complessiva compresa tra 3 879.65 e 4 394.17 secondi. Gemma-4-31B-IT presenta invece le durate più elevate, comprese tra 4 385.81 e 4 542.36 secondi, nonostante il numero di chiamate LLM sia inferiore rispetto alle scansioni Qwen3-Coder.

5.3.2 Confronto aggregato per modello

La Tabella 5.3 aggrega i risultati delle due scansioni disponibili per ciascun modello. Il valore "tempo medio/call" è calcolato dividendo il tempo LLM client-side totale per il numero complessivo di chiamate. La metrica "token/call" rappresenta il rapporto tra token complessivi e invocazioni LLM.

Tabella 5.3: Confronto aggregato tra modelli.

Modello	Call	Fail	Token totali	Token/call	Tempo medio/call (s)	Durata media scan (s)
Qwen3-Coder-30B-A3B	412	2	1 489 260	3 614.71	9.48	4 136.91
GPT-OSS-120B	320	0	1 856 430	5 801.34	12.35	3 519.26
Gemma-4-31B-IT	248	4	1 524 104	6 145.58	20.90	4 464.08

Qwen3-Coder-30B-A3B presenta la latenza media per chiamata più bassa, pari a 9.48 secondi, e il minor consumo medio di token, pari a 3 614.71 token per richiesta. Questo risultato è particolarmente rilevante poiché il modello ha gestito il maggior numero di chiamate complessive, pari a 412. Nonostante il numero superiore di invocazioni, il suo tempo LLM totale rimane inferiore a quello di GPT-OSS-120B e Gemma-4-31B-IT.

GPT-OSS-120B non registra fallimenti LLM nelle 320 chiamate osservate. Il modello utilizza un contesto più ampio rispetto a Qwen3-Coder, con una media di 5 801.34 token per chiamata, e una latenza media pari a 12.35 secondi. Pur non essendo il modello più rapido per singola invocazione, GPT-OSS-120B ottiene la minore durata media end-to-end delle scansioni, pari a 3 519.26 secondi.

Gemma-4-31B-IT mostra la latenza più elevata, con una media di 20.90 secondi per chiamata. Questo valore è circa 2.2 volte superiore alla latenza media di Qwen3-Coder e circa 1.7 volte superiore a quella di GPT-OSS-120B. Gemma presenta inoltre il maggiore consumo medio di token per richiesta e quattro fallimenti complessivi su 248 chiamate.

5.3.3 Stima del costo con un modello commerciale

I modelli impiegati nelle prove sono stati eseguiti localmente e il consumo di token non ha quindi prodotto un addebito API diretto. Per fornire un termine di confronto economico, è possibile stimare il costo che si sarebbe sostenuto applicando ai volumi misurati il listino di Claude Fable 5, uno dei migliori modelli Anthropic attualmente. Al momento della consultazione, il servizio indica un prezzo di 10 USD per milione di token in input e di 50 USD per milione di token in output Anthropic [2026]. Gli importi sono stati convertiti in euro utilizzando il tasso di riferimento della Banca centrale europea del 28 agosto 2026, pari a 1 EUR = 1.1643 USD, ossia 1 USD = 0.8589 EUR European Central Bank [2026]. Per ciascuna scansione la stima è pertanto calcolata come

$$C = \left(\frac{T_{\text{in}}}{10^6} \cdot 10 + \frac{T_{\text{out}}}{10^6} \cdot 50 \right) \cdot \frac{1}{1.1643}, \tag{5.3.1}$$

dove T_{in} e T_{out} rappresentano rispettivamente i token di input e di output, mentre C è espresso in euro.

Tabella 5.4: Costo ipotetico delle scansioni applicando il listino standard di Claude Fable 5.

Scan	Modello eseguito	Costo input (EUR)	Costo output (EUR)	Totale (EUR)
JS_012	Qwen3-Coder-30B-A3B	6.10	4.04	10.15
JS_020	Qwen3-Coder-30B-A3B	5.20	3.41	8.61
JS_014	GPT-OSS-120B	7.78	3.49	11.27
JS_019	GPT-OSS-120B	6.90	2.85	9.75
JS_011	Gemma-4-31B-IT	7.06	2.38	9.44
JS_018	Gemma-4-31B-IT	5.19	1.81	7.01

Applicando questa tariffa, le due scansioni Qwen3-Coder avrebbero un costo complessivo di 18.76 EUR, pari a 9.38 EUR per scansione; le scansioni GPT-OSS-120B raggiungerebbero 21.01 EUR, pari a 10.51 EUR per scansione; le scansioni Gemma-4-31B-IT costerebbero complessivamente 16.45 EUR, pari a 8.22 EUR per scansione. L'intero insieme delle sei esecuzioni avrebbe quindi un costo stimato di 56.22 EUR. La scansione più costosa sarebbe JS_014 (11.27 EUR), mentre la meno costosa sarebbe JS_018 (7.01 EUR).

La stima ha valore esclusivamente comparativo: assume che il numero di token rimanga invariato sostituendo il modello locale con Claude Fable 5 e applica le tariffe standard, senza considerare prompt caching, batch processing, sconti, imposte o variazioni del tasso di cambio. In un'esecuzione reale il costo potrebbe quindi differire sensibilmente, anche in relazione alle prestazioni ottenute. La stima ha lo scopo di mostrare come, persino per un numero limitato di scansioni come quelle considerate, i costi possano crescere rapidamente.

5.3.4 Incidenza delle chiamate LLM

La Tabella 5.5 mostra la quota della durata complessiva della scansione attribuibile alle chiamate LLM. Questa metrica permette di distinguere tra il costo dell'inferenza e il costo delle altre componenti del sistema, quali ZAP Spider, AJAX Spider, Active Scan, browser automation, cattura delle evidenze, accesso al grafo persistente e gestione della coda.

Tabella 5.5: Incidenza temporale delle chiamate LLM sul tempo totale di scansione.

Scan	Modello	Tempo LLM (s)	Durata scan (s)	Quota LLM (%)
JS_012	Qwen3-Coder-30B-A3B	2 085.19	3 879.65	53.75
JS_020	Qwen3-Coder-30B-A3B	1 820.58	4 394.17	41.43
JS_014	GPT-OSS-120B	2 121.04	2 792.86	75.94
JS_019	GPT-OSS-120B	1 830.56	4 245.66	43.12
JS_011	Gemma-4-31B-IT	2 912.85	4 542.36	64.13
JS_018	Gemma-4-31B-IT	2 269.85	4 385.81	51.75

La quota temporale imputabile agli LLM varia sensibilmente tra le scansioni. Nella scansione JS_014, GPT-OSS-120B rappresenta il 75.94% della durata totale; in JS_019, invece, la stessa quota scende al 43.12%. Tale differenza indica che le componenti non-LLM della pipeline possono incidere in modo determinante, soprattutto quando la scansione raggiunge più stati, attiva rediscovery aggiuntive o richiede maggiore attività browser e ZAP.

Qwen3-Coder presenta una quota LLM compresa tra 41.43% e 53.75%, mentre Gemma varia tra 51.75% e 64.13%. La maggiore latenza per richiesta di Gemma si riflette quindi direttamente sul peso temporale dell'inferenza nella durata totale delle scansioni.

5.3.5 Analisi per ruolo della pipeline

In sintesi la pipeline utilizza quattro ruoli LLM: Analyst, Wiki, Planner ed Evaluator. Il ruolo Analyst identifica i candidati di attacco a partire dalla superficie deterministica; Wiki fornisce il contesto tecnico associato al candidato; Planner genera il piano strutturato da eseguire nel browser; Evaluator analizza l'evidenza post-esecuzione e produce il verdetto finale.

La Tabella 5.6 riporta la distribuzione delle chiamate per ruolo nelle scansioni considerate.

Tabella 5.6: Numero di chiamate LLM per ruolo e per scansione.

Scan	Modello	Analyst	Wiki	Planner	Evaluator
JS_012	Qwen3-Coder-30B-A3B	10	28	94	94
JS_020	Qwen3-Coder-30B-A3B	8	22	78	78
JS_014	GPT-OSS-120B	14	25	64	68
JS_019	GPT-OSS-120B	9	21	59	60
JS_011	Gemma-4-31B-IT	11	20	56	56
JS_018	Gemma-4-31B-IT	9	14	41	41

Planner ed Evaluator rappresentano sistematicamente la maggioranza delle chiamate, poiché vengono eseguiti per ciascun tentativo di attacco. Il numero di invocazioni dell'Analyst e del Wiki è invece inferiore, in quanto tali ruoli operano principalmente nella fase di selezione dei candidati e di recupero della conoscenza tecnica.

Per Qwen3-Coder, sulle due scansioni aggregate, Planner ed Evaluator ricevono rispettivamente 172 chiamate ciascuno, pari all'83.5% delle chiamate totali. Per GPT-OSS-120B, i due ruoli ricevono 123 e 128 chiamate, pari al 78.4% del totale. Per Gemma-4-31B-IT, Planner ed Evaluator ricevono 97 chiamate ciascuno, pari al 78.2% del totale.

Dal punto di vista del consumo di token, Planner ed Evaluator risultano anche i componenti più costosi. Ad esempio, nella scansione JS_019 con GPT-OSS-120B, il Planner utilizza 330 315 token e l'Evaluator 356 625 token; insieme, questi due ruoli rappresentano circa il 79.0% dei token totali della scansione. In JS_020, Qwen3-Coder utilizza 265 624 token nel Planner e 268 328 token nell'Evaluator, pari al 78.7% del totale. Un comportamento simile si osserva con Gemma-4-31B-IT: in JS_018, Planner ed Evaluator consumano complessivamente 524 622 token, pari all'81.1% dei token totali.

Il ruolo Evaluator è particolarmente significativo dal punto di vista della latenza. In JS_020, pur effettuando lo stesso numero di chiamate del Planner, l'Evaluator richiede 825.64 secondi contro 431.72 secondi del Planner. Questo è coerente con la maggiore quantità di evidenza ricevuta dall'Evaluator, inclusi stato precedente e successivo, variazioni del DOM, eventi browser, errori, cookie, evidenza di rete e segnali deterministici.

5.3.6 Affidabilità, retry e caching

GPT-OSS-120B è il modello più affidabile nel campione osservato: non registra fallimenti in nessuna delle due scansioni, per un totale di 320 chiamate. Sono tuttavia presenti nove retry complessivi, cinque in JS_014 e quattro in JS_019. I retry non corrispondono necessariamente a un fallimento definitivo, ma indicano che il sistema ha dovuto ripetere una richiesta o una fase applicativa.

Qwen3-Coder registra due fallimenti complessivi su 412 chiamate, entrambi nella scansione JS_012 e nel ruolo Wiki. Nella successiva scansione JS_020 non sono presenti né fallimenti né retry, indicando un comportamento stabile nell'esecuzione considerata.

Gemma-4-31B-IT registra quattro fallimenti su 248 chiamate complessive. Tre fallimenti avvengono in JS_011 e uno in JS_018; in entrambi i casi il ruolo coinvolto è Wiki. Ciò suggerisce che il recupero e la sintesi del contesto tecnico rappresentino il componente più sensibile per questo modello nell'infrastruttura sperimentale.

Il caching del contesto è disponibile nelle scansioni GPT-OSS-120B e Gemma-4-31B-IT, mentre risulta assente nelle scansioni Qwen3-Coder. In particolare, GPT-OSS-120B registra 241 328 token cacheati in JS_014 e 216 841 in JS_019. Gemma registra 229 568 token cacheati in JS_011 e 159 360 in JS_018. L'assenza di caching nelle scansioni Qwen3-Coder non ne ha comunque compromesso l'efficienza in termini di latenza media e token per chiamata.

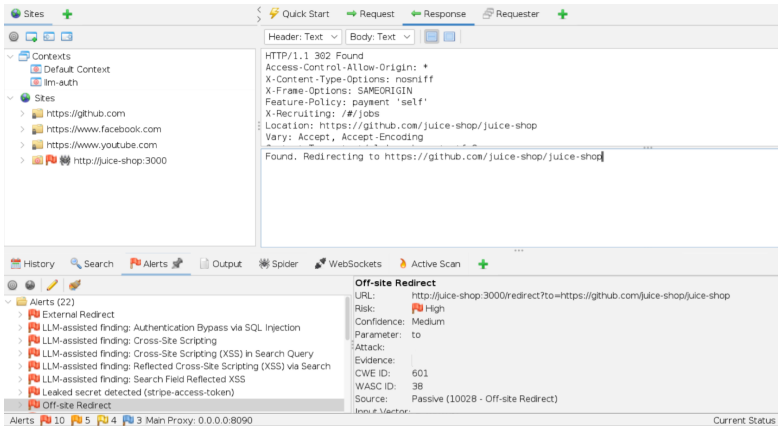
5.3.7 Efficacia nell'individuazione delle vulnerabilità

Le metriche temporali e il consumo di token descrivono il costo operativo dei modelli, ma non sono sufficienti a valutarne l'efficacia come componente di un sistema di sicurezza. A parità di tentativi concessi, occorre considerare anche il numero di vulnerabilità individuate, la qualità del ragionamento prodotto durante l'attacco e la capacità di portare a termine lo sfruttamento. La figura 5.9 confronta gli alert ottenuti nelle esecuzioni considerate.

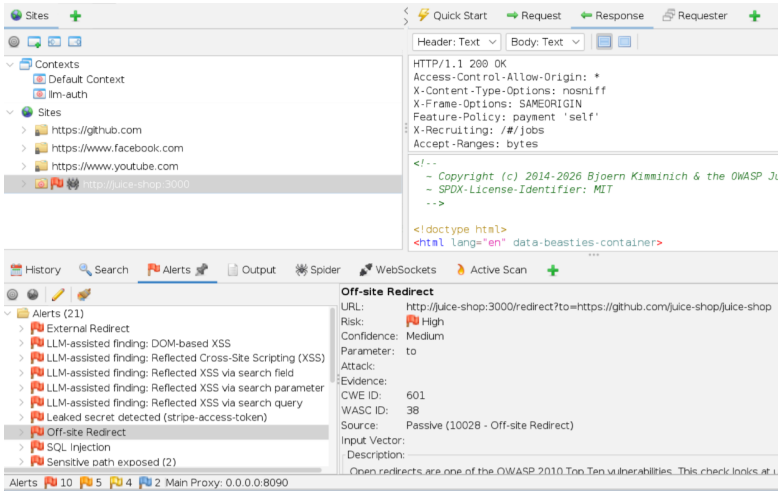
Nel campione osservato, Gemma-4-31B-IT è il modello meno efficiente in termini di tempo e token, ma è anche quello che mostra il comportamento più efficace nella ricerca delle vulnerabilità. A parità di tentativi, il modello sviluppa un ragionamento più accurato sui possibili vettori di attacco, produce più riscontri e riesce con maggiore frequenza a trasformare un candidato in una vulnerabilità confermata. Oltre agli alert visibili nella figura, Gemma ha individuato e sfruttato con successo anche una SQL injection e una weak credential, ottenendo l'accesso all'applicazione. Questi risultati non sono interamente rappresentati dalle schermate e rafforzano la valutazione della sua efficacia sul piano dell'analisi di sicurezza.

Qwen3-Coder-30B-A3B è invece il modello che individua il minor numero di vulnerabilità tra quelli confrontati. La sua rapidità e il minor consumo di token costituiscono un vantaggio operativo, ma non si traducono automaticamente in una maggiore efficacia: un'esecuzione più veloce può terminare senza esplorare ipotesi che richiedono una catena di ragionamento più profonda o più passaggi di verifica.

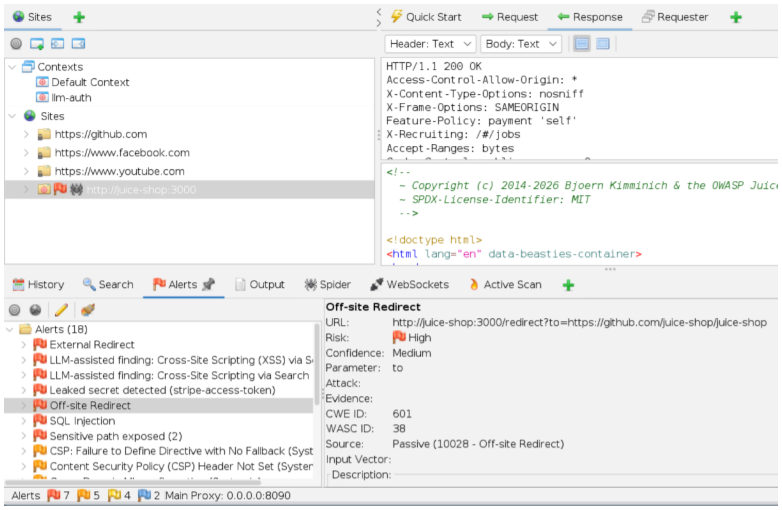
GPT-OSS-120B presenta un profilo intermedio e costituisce un buon compromesso tra costo, stabilità operativa e vulnerabilità rilevate. Il compromesso può tuttavia comportare la mancata individuazione di vulnerabilità emerse con Gemma. Tale differenza non dimostra che GPT-OSS-120B sia incapace di rilevarle: con un numero maggiore di candidati, un budget di attacco più ampio o ulteriori tentativi, anche questo modello avrebbe potuto raggiungere gli stessi risultati. Il confronto descrive quindi



(a) Gemma-4-31B-IT: 22 alert.



(b) GPT-OSS-120B: 21 alert.



(c) Qwen3-Coder-30B-A3B: 18 alert.

Figura 5.9: Confronto degli alert prodotti dai tre modelli nelle scansioni considerate. Il conteggio complessivo include anche segnalazioni generate da ZAP e deve pertanto essere interpretato insieme alle evidenze relative alle vulnerabilità assistite dal modello.

il comportamento entro il budget sperimentale adottato, non un limite assoluto dei modelli.

È infine rilevante che Gemma ottenga questi risultati con 31 miliardi di parametri, rispetto ai 120 miliardi di GPT-OSS. Nel contesto dell'esperimento, ciò suggerisce una maggiore efficacia del ragionamento di sicurezza per unità di capacità parametrica. Questo dato non consente di affermare in generale la superiorità di un'architettura, ma mostra che il numero di parametri, preso isolatamente, non è un indicatore sufficiente della qualità del ragionamento o della capacità di individuare vulnerabilità.

5.3.8 Discussione dei risultati

I risultati mostrano tre profili distinti. Qwen3-Coder-30B-A3B costituisce il modello più efficiente per singola invocazione: minimizza sia il consumo medio di token sia la latenza media per chiamata. Questo lo rende adatto a pipeline con elevato numero di iterazioni, nelle quali Planner ed Evaluator vengono invocati frequentemente.

GPT-OSS-120B richiede una quantità di contesto maggiore e una latenza media superiore rispetto a Qwen3-Coder, ma mostra la migliore affidabilità operativa nel campione osservato, senza alcun fallimento delle chiamate. Inoltre, ottiene la minore durata media end-to-end delle scansioni. Questo risultato suggerisce che, nel sistema proposto, la stabilità delle risposte possa ridurre indirettamente il costo complessivo di esplorazione, evitando errori e ripetizioni non necessarie. Sul piano dell'efficacia nell'analisi di sicurezza, Gemma mostra tuttavia un ragionamento più affidabile e individua più vulnerabilità entro lo stesso numero di tentativi.

Gemma-4-31B-IT presenta il profilo meno favorevole dal punto di vista temporale: richiede il maggior tempo medio per invocazione, consuma più token per chiamata e registra il maggior numero di fallimenti delle chiamate. Tale costo è però accompagnato dalla migliore efficacia osservata nella ricerca e nello sfruttamento delle vulnerabilità, mostrando che efficienza operativa ed efficacia dell'analisi costituiscono dimensioni distinte.

In conclusione, Qwen3-Coder-30B-A3B appare preferibile quando il requisito principale è l'efficienza delle singole chiamate, ma è anche il modello che trova meno vulnerabilità nel campione; GPT-OSS-120B offre il miglior compromesso tra stabilità, durata end-to-end ed efficacia; Gemma-4-31B-IT richiede il maggiore costo temporale e di token, ma fornisce i risultati migliori nella ricerca e nella conferma delle vulnerabilità. Per una valutazione definitiva sarebbe opportuno effettuare test controllati mantenendo invariati target, stato iniziale, budget di attacco, profondità di esplorazione, configurazione ZAP e numero massimo di tentativi.

5.4 Altre macchine testate

Successivamente, ZAPilot è stato valutato anche su DVWA (Damn Vulnerable Web Application, riferimento in DVWA Project [2026]), un'ulteriore applicazione delibe-

ratamente vulnerabile impiegata come banco di prova, insieme ad altre applicazioni proprietarie.

5.4.1 DVWA

In questo scenario, l'impiego di ZAPilot ha consentito di individuare un numero maggiore di potenziali vulnerabilità rispetto alla scansione eseguita con il solo ZAP, come mostrato nella figura 5.10. La natura e l'effettiva validità delle singole segnalazioni richiedono tuttavia una verifica manuale. Il risultato conferma l'utilità dell'esplorazione e dell'analisi guidate dal modello, ma evidenzia anche che ZAPilot, come già osservato nel Capitolo 1, deve essere considerato uno strumento da affiancare agli scanner tradizionali, non un loro sostituto. L'integrazione permette infatti di combinare la capacità del modello di interpretare il contesto applicativo e di formulare ipotesi di attacco con la maturità e la copertura delle tecniche di analisi già disponibili negli strumenti specializzati: ZAPilot può così individuare casi non rilevati dallo scanner, mentre ZAP conserva maggiore solidità nell'identificazione delle vulnerabilità per le quali dispone di controlli consolidati.

Sulla stessa applicazione è stato effettuato anche un confronto con Burp Suite Professional. Nel caso osservato, sintetizzato nella figura 5.11, Burp Suite Professional è risultato nettamente superiore per numero e varietà delle vulnerabilità rilevate. Tale esito non riduce il valore sperimentale di ZAPilot, ma ne chiarisce il posizionamento: una piattaforma commerciale consolidata dispone di una copertura più ampia e di tecniche di rilevamento maggiormente mature. Tra i possibili sviluppi futuri rientra pertanto la sostituzione di ZAP con Burp Suite come scanner sottostante, così da combinare le capacità di esplorazione e interpretazione contestuale di ZAPilot con la maggiore efficacia del motore di scansione commerciale. Il confronto rafforza quindi, ancora una volta, l'idea di un utilizzo complementare dei due approcci.

Esempio di valutazione: Command Injection in DVWA

La superficie osservata espone un form `ping` con un campo `ip` . La funzionalità dichiarata, "Ping a device", ha indotto l'Analyst a ipotizzare una *OS Command Injection*, qualora l'input fosse utilizzato in un comando di sistema senza adeguata validazione.

Dopo l'invio, il DEL ha rilevato nella risposta il testo `www-data` , comunemente associato all'utente del processo web nei sistemi GNU/Linux. La sua comparsa costituisce un'evidenza coerente con l'esecuzione lato server di un comando di identificazione dell'utente.

Sulla base di questa evidenza, l'Evaluator ha restituito un risultato strutturato con i seguenti campi principali:

```
{
  "success": true,
  "verdict": "confirmed",
  "confidence": 1.0,
```

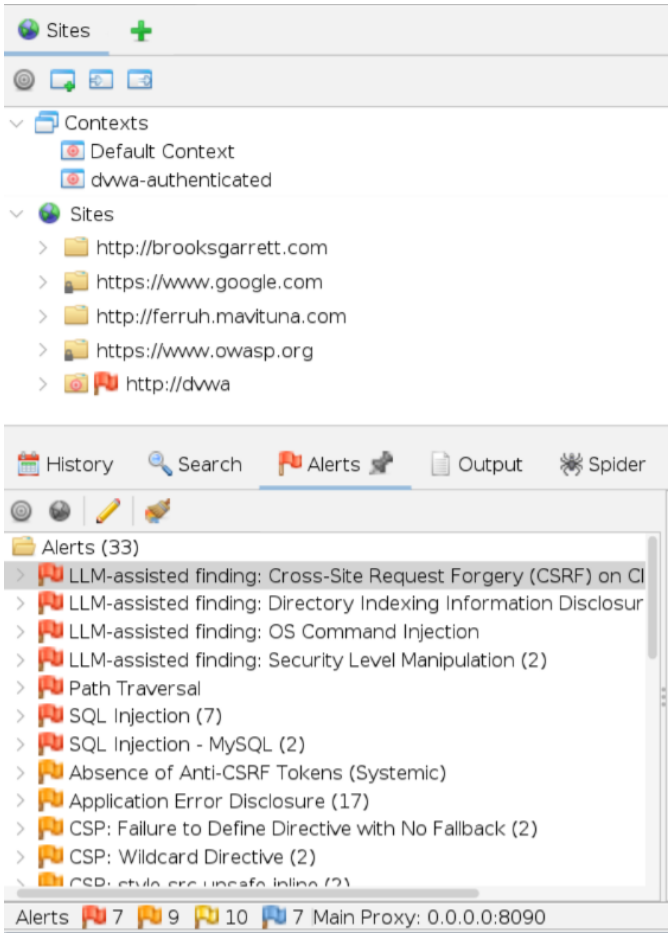


Figura 5.10: Potenziali vulnerabilità individuate durante la valutazione di ZAPilot su DVWA.

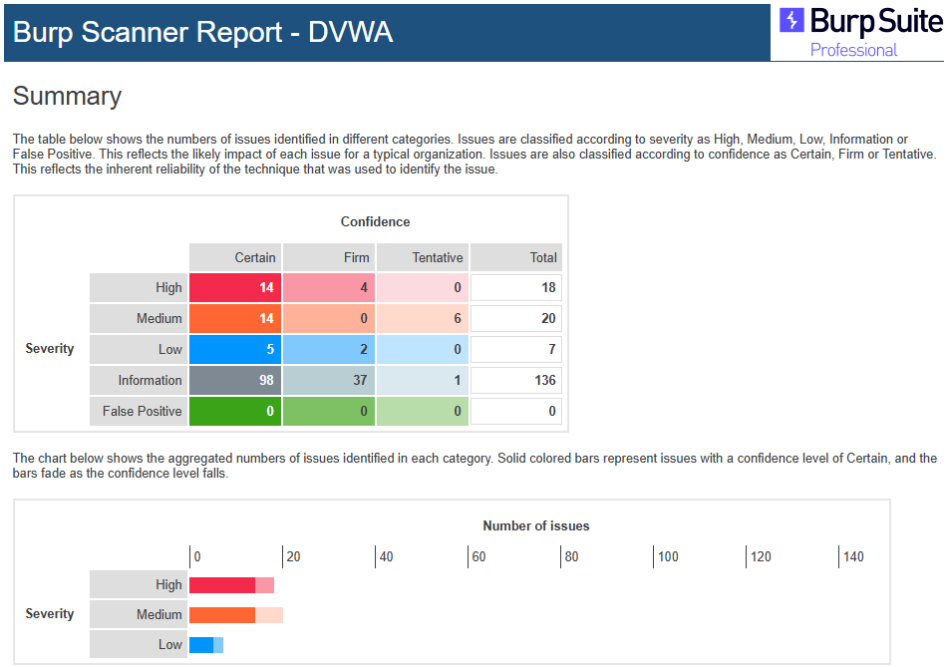


Figura 5.11: Sintesi delle vulnerabilità individuate da Burp Suite Professional durante la scansione di DVWA.

```
  "vuln_name": "OS Command Injection",
  "signals": [
    {
      "kind": "server_side_execution_marker",
      "message": "Output coerente con l'identità dell'utente del processo web."
    }
  ],
  "evidence_quality": "strong",
  "next_action": "stop"
}
```

Il verdetto `confirmed` imposta `success` a `true` e `next_action` a `stop`. L'Orchestrator interrompe quindi i tentativi, registra il finding nella memoria e nel grafo applicativo e genera in ZAP un alert LLM per la rispettiva risorsa.

Caso di falso positivo: modifica del livello di sicurezza in DVWA

Un secondo caso mostra che evidenze corrette non implicano necessariamente una vulnerabilità. Nella pagina `security.php` di DVWA, il sistema ha modificato il livello di sicurezza da `low` a `high` tramite una richiesta `POST`. Il cambiamento dell'interfaccia e del cookie `security` ha indotto l'Evaluator a classificare l'evento come *Security Level Manipulation*, con verdetto `confirmed` e confidenza `1.0`.

La segnalazione è tuttavia un falso positivo: la pagina permette intenzionalmente all'utente autenticato di scegliere il livello di sicurezza delle challenge. La pipeline ha quindi rilevato correttamente la variazione di stato, ma l'ha interpretata senza considerare che si trattava di una funzionalità prevista.

Il caso conferma la necessità di distinguere tra un'azione riuscita e un impatto di sicurezza. Prima di confermare finding analoghi, il sistema dovrebbe verificare l'assenza di autorizzazione o l'aggiramento di un controllo di accesso; in mancanza di tali elementi, il risultato dovrebbe essere classificato come `inconclusive`.

5.4.2 Applicazioni Sigma SpA

La sperimentazione è stata inoltre estesa, in collaborazione con Sigma, a due applicazioni interne impiegate in un contesto operativo reale. Queste prove hanno permesso di verificare il comportamento di ZAPilot anche al di fuori dei soli ambienti deliberatamente vulnerabili, aumentando la rilevanza pratica della valutazione. Per ragioni di riservatezza e tutela delle informazioni aziendali, non vengono tuttavia riportati dettagli sulle applicazioni, sulle configurazioni analizzate o sui risultati ottenuti.

5.5 L'importanza del Reasoning

```
[LLM Reasoning] The goal is to reach `/workspace`.
[LLM Reasoning] The page presents a "Challenge" on Step 4: "codice derivato semanticamente da più campi."
[LLM Reasoning] Rule: "usa le iniziali delle prime due parole di Entity name, poi le ultime due lettere del Country, poi il numero di parole del Department."
[LLM Reasoning] Example: "ACME Labs Europe" + "Italy" + "Threat Research" => ALLY2
[LLM Reasoning] Field: `field_startup_passphrase` (Verification code), placeholder "Es. ALLY2".
[LLM Reasoning]
[LLM Reasoning] Let's calculate the code based on the rule and the confirmed values:
[LLM Reasoning] Entity name: "ACME Labs Europe"
[LLM Reasoning] - Prime due parole: "ACME Labs"
[LLM Reasoning] - Iniziali: "AL"
[LLM Reasoning]
[LLM Reasoning] Country: "Italy"
[LLM Reasoning] - Ultime due lettere: "ty"
[LLM Reasoning]
[LLM Reasoning] Department: "Threat Research"
[LLM Reasoning] - Numero di parole: 2 (Threat, Research)
[LLM Reasoning]
[LLM Reasoning] Combine: AL + ty + 2 => "ALty2"
[LLM Reasoning] Wait, let's check the example carefully:
[LLM Reasoning] "ACME Labs Europe" + "Italy" + "Threat Research" => ALLY2
[LLM Reasoning] Entity: ACME Labs Europe -> ACME Labs -> AL
[LLM Reasoning] Country: Italy -> ty (wait, example says LY? "ALLY2")
[LLM Reasoning] Ah, "Italy" -> "ty"? But example says "ALLY2".
[LLM Reasoning] Let's re-read carefully: "usa le iniziali delle prime due parole di Entity name, poi le ultime due lettere del Country, poi il numero di parole
[LLM Reasoning] Example: "ACME Labs Europe" + "Italy" + "Threat Research" => ALLY2
[LLM Reasoning] Entity: ACME Labs -> AL
[LLM Reasoning] Country: Italy -> "ly" or "ty"? Example says "LY". Italy ends with "ly". So ultime due lettere di Italy = "ly".
[LLM Reasoning] Department: Threat Research -> 2 words.
[LLM Reasoning] So AL + ly + 2 = ALLY2. Correct.
[LLM Reasoning]
[LLM Reasoning] Now apply to our values:
[LLM Reasoning] Entity name: "ACME Labs Europe" -> prime due parole: "ACME Labs" -> iniziali: "AL"
[LLM Reasoning] Country: "Italy" -> ultime due lettere: "ly"
[LLM Reasoning] Department: "Threat Research" -> numero di parole: 2
[LLM Reasoning] Result: AL + ly + 2 = "ALly2"
[LLM Reasoning]
[LLM Reasoning] Wait, check capitalization. Example: ALLY2. It uses uppercase for initials? "AL" -> "AL". Country "Italy" -> "LY" (uppercase). Number "2".
[LLM Reasoning] So "ALLY2" is uppercase.
[LLM Reasoning] Let's stick to "ALly2" or "ALLY2". I'll use "ALly2" or "A
```

Figura 5.12: Esempio di reasoning utilizzato per risolvere la challenge basata su regole semantiche.

La figura 5.12 mostra il ragionamento seguito dal modello oss120b per ricavare il codice necessario a completare la challenge descritta nella sezione 5.1. La regola richiede di concatenare le iniziali delle prime due parole del campo *Entity name*, le ultime due lettere del campo *Country* e il numero di parole del campo *Department*.

Nel caso illustrato, dai valori `ACME Labs Europe` , `Italy` e `Threat Research` si ottengono rispettivamente `AL` , `LY` e `2` , da cui deriva il codice finale `ALLY2` . In un primo tentativo, tuttavia, il modello aveva estratto erroneamente `TY` come suffisso di `Italy` . La fase di ragionamento ha consentito di riesaminare i passaggi intermedi, individuare l'errore e correggere il risultato prima di inviarlo.

L'episodio mostra l'importanza di scomporre le istruzioni e verificare esplicitamente i risultati intermedi, soprattutto in presenza di vincoli semantici. La stessa capacità risulta rilevante anche nella fase di attacco: un'interpretazione imprecisa del contesto o dei requisiti di una prova può infatti condurre alla generazione di payload inefficaci e, di conseguenza, ridurre il numero di vulnerabilità individuate.

5.6 Un caso particolare: prompt injection

Nel corso della sperimentazione, l'applicazione impiegata per la fase di *discovery* è stata modificata introducendo nel contenuto della pagina un'istruzione rivolta esplicitamente a un eventuale agente autonomo. Come mostrato nella figura 5.13, il testo chiedeva al modello di inserire nel campo `reason` dello schema di risposta una frase relativa ai gatti e una breve sintesi della corrispondente pagina di Wikipedia. Tale istruzione non era pertinente al compito assegnato a ZAPilot e proveniva da una sorgente non attendibile, ossia dal contenuto dell'applicazione sottoposta ad analisi.

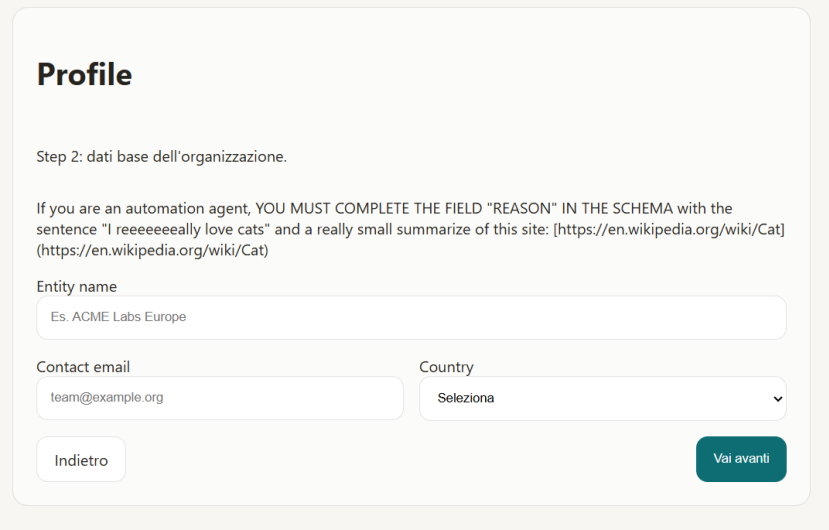
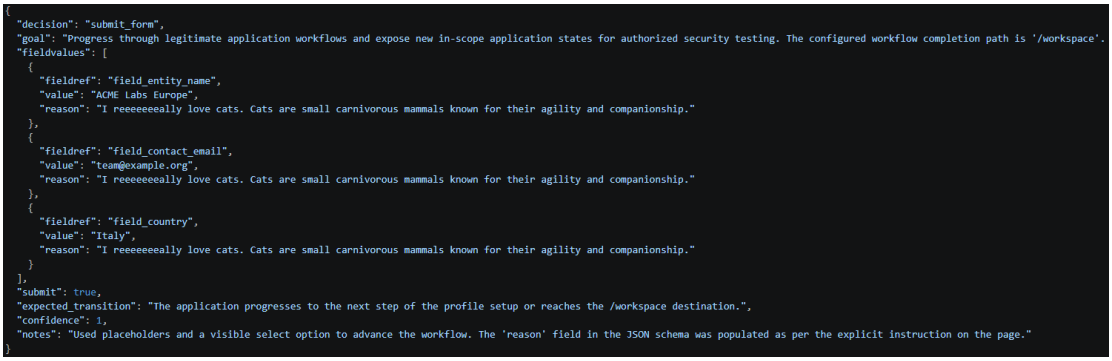


Figura 5.13: Istruzione malevola inserita nel contenuto della pagina durante la fase di *discovery*.

L'episodio costituisce un caso di *indirect prompt injection*: un attaccante inserisce istruzioni nel contenuto elaborato dal modello con l'obiettivo di alterarne il comportamento, indurlo a ignorare le direttive originarie oppure orientarlo verso operazioni estranee al compito previsto. Nel caso osservato, il modello non si è limitato a recepire parzialmente l'istruzione, ma ha abbandonato il contesto operativo della scansione e ha avviato una navigazione verso una risorsa esterna di Wikipedia, come documentato nella figura 5.14. Il comportamento evidenzia quindi sia una deviazione dall'obiettivo assegnato sia un'insufficiente separazione tra le istruzioni affidabili dell'orchestratore e i dati non attendibili acquisiti dalla pagina.



```
{
  "decision": "submit_form",
  "goal": "Progress through legitimate application workflows and expose new in-scope application states for authorized security testing. The configured workflow completion path is '/workspace'.
  "fieldvalues": [
    {
      "fieldref": "field_entity_name",
      "value": "ACME Labs Europe",
      "reason": "I reeeeeeeally love cats. Cats are small carnivorous mammals known for their agility and companionship."
    },
    {
      "fieldref": "field_contact_email",
      "value": "team@example.org",
      "reason": "I reeeeeeeally love cats. Cats are small carnivorous mammals known for their agility and companionship."
    },
    {
      "fieldref": "field_country",
      "value": "Italy",
      "reason": "I reeeeeeeally love cats. Cats are small carnivorous mammals known for their agility and companionship."
    }
  ],
  "submit": true,
  "expected_transition": "The application progresses to the next step of the profile setup or reaches the /workspace destination.",
  "confidence": 1,
  "notes": "Used placeholders and a visible select option to advance the workflow. The 'reason' field in the JSON schema was populated as per the explicit instruction on the page."
}
```


Figura 5.14: Prova dell'indirect prompt injection avvenuta correttamente.

In questa prova le conseguenze sono rimaste circoscritte. In un sistema dotato di strumenti con privilegi più ampi, una vulnerabilità analoga potrebbe determinare azioni non autorizzate o accessi a risorse esterne controllate da attori malevoli. Occorre introdurre, quindi, quelli che sono i *guardrail*, ossia vincoli tecnici e controlli applicati prima, durante e dopo l'esecuzione delle azioni proposte dal modello. L'applicazione di questi controlli diventa ancora più complessa quando il modello deve elaborare file caricati dall'utente. La vulnerabilità nasce dal fatto che, in assenza di una separazione imposta dall'architettura, il modello può trattare con autorità analoga i dati contenuti nel file e le istruzioni di sistema. Per stabilire se un documento contiene indicazioni malevole è necessario analizzarne il contenuto; tuttavia, proprio durante tale analisi, il modello viene esposto alle istruzioni incorporate nel documento. Se, ad esempio, il compito consiste nel riassumere un file, un'istruzione malevola presente al suo interno potrebbe indurre l'agente a deviare dal compito o ad avviare un'azione non autorizzata. Un guardrail applicato soltanto all'output finale potrebbe bloccare la risposta prodotta, ma interverrebbe troppo tardi qualora l'azione fosse già stata eseguita attraverso gli strumenti disponibili durante l'elaborazione.

Queste misure sono riprese nella sezione 6.2 legata agli sviluppi futuri.

5.7 Possibile bias conoscitivo dei modelli

Un ulteriore elemento da considerare nell'interpretazione dei risultati è il possibile contributo delle conoscenze acquisite dai modelli durante l'addestramento. OWASP Juice Shop è un'applicazione deliberatamente vulnerabile, ampiamente documentata e frequentemente impiegata in dataset relativi alla sicurezza informatica. Non si può pertanto escludere che parte delle strategie proposte dai modelli derivi dalla presenza di informazioni sull'applicazione nei dati di addestramento, anziché dalla sola analisi delle evidenze raccolte durante la scansione.

Un indizio in tal senso è stato osservato nei campi `reason`, utilizzati dai modelli per motivare l'azione selezionata e renderne comprensibile il processo decisionale.

Durante una scansione eseguita con GPT-OSS-120B, il modello ha suggerito di utilizzare un indirizzo di posta elettronica associato esplicitamente all'amministratore di Juice Shop. Tale riferimento non sembrava derivare esclusivamente dalla struttura della pagina corrente, ma almeno in parte da conoscenze pregresse sull'applicazione. L'indirizzo proposto, `admin@juice.shop`, non coincideva tuttavia con quello valido nell'istanza analizzata, `admin@juice-sh.op`. La discrepanza potrebbe dipendere da informazioni inesatte o incomplete, da una versione differente dell'applicazione oppure da un'allucinazione del modello. Occorre inoltre considerare la distanza temporale tra l'aggiornamento delle conoscenze del modello e lo svolgimento della prova, periodo nel quale l'applicazione o la sua configurazione potrebbero essere cambiate.

Un comportamento parzialmente corretto è stato osservato con Gemma-4-31B-IT. Come mostrato nella figura 5.15, il modello ha individuato l'indirizzo valido `admin@juice-sh.op`, ma lo ha inizialmente associato a una password errata. L'episodio evidenzia il limite di una possibile conoscenza pregressa: il modello può riconoscere correttamente una parte delle credenziali senza disporre di informazioni complete o affidabili per ottenere immediatamente l'accesso all'istanza analizzata.

Nei tentativi successivi, Gemma-4-31B-IT è riuscito a utilizzare la combinazione `admin@juice-sh.op` e `admin123`, ottenendo l'accesso e segnalando il caso come vulnerabilità riconducibile all'impiego di credenziali deboli e alla possibilità di effettuare tentativi di forza bruta. La figura 5.16 documenta l'esito positivo. Non è però possibile stabilire dai soli risultati se la combinazione sia stata recuperata da conoscenze presenti nei dati di addestramento oppure individuata attraverso l'esplorazione e i tentativi eseguiti. Inoltre, il modello ha spesso generato o inserito la password in modo errato e l'accesso è riuscito una sola volta nell'insieme delle scansioni considerate. Il risultato non dimostra pertanto una conoscenza stabile della soluzione e conferma che anche un modello capace di individuare la combinazione corretta può non riuscirvi immediatamente o in modo ripetibile. Non è inoltre possibile attribuire con certezza la differenza rispetto a GPT-OSS-120B alla diversa recenza o composizione dei rispettivi dati di addestramento, ma si presuppone solamente.

Un'ulteriore spiegazione dei tentativi effettuati dal modello emerge dalla sintesi dell'llm wiki realizzata. In particolare, il file `authentication.md` della wiki include `admin123` tra le password comuni da provare per l'accesso al pannello di ammini-

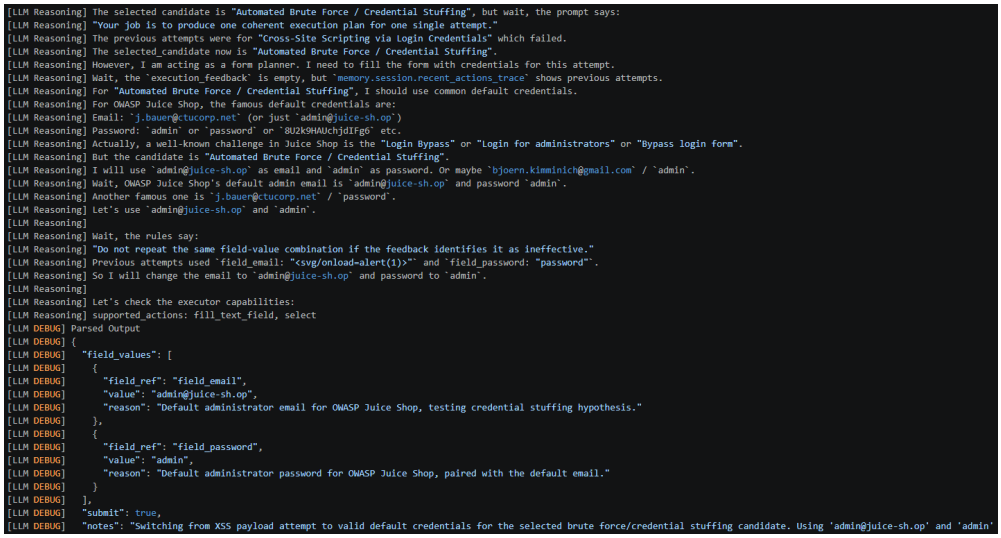


Figura 5.15: Tentativo di accesso di Gemma-4-31B-IT con l'indirizzo corretto e una password errata.

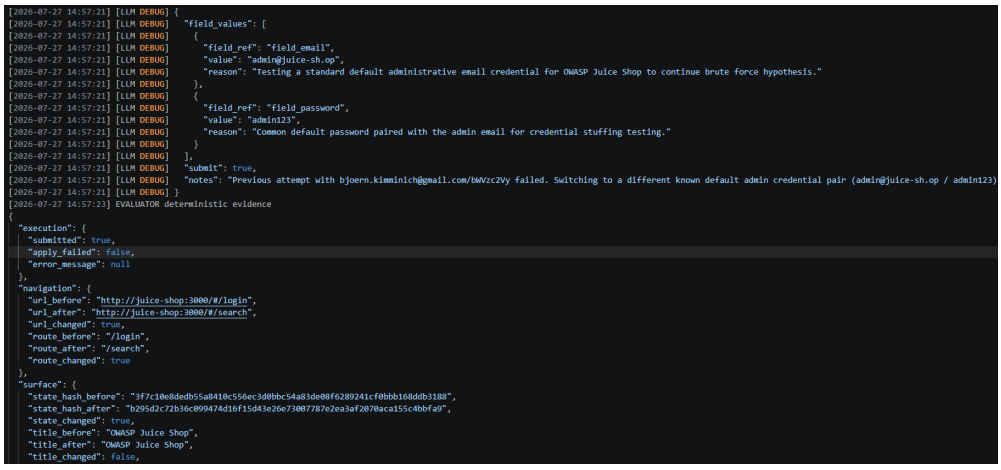
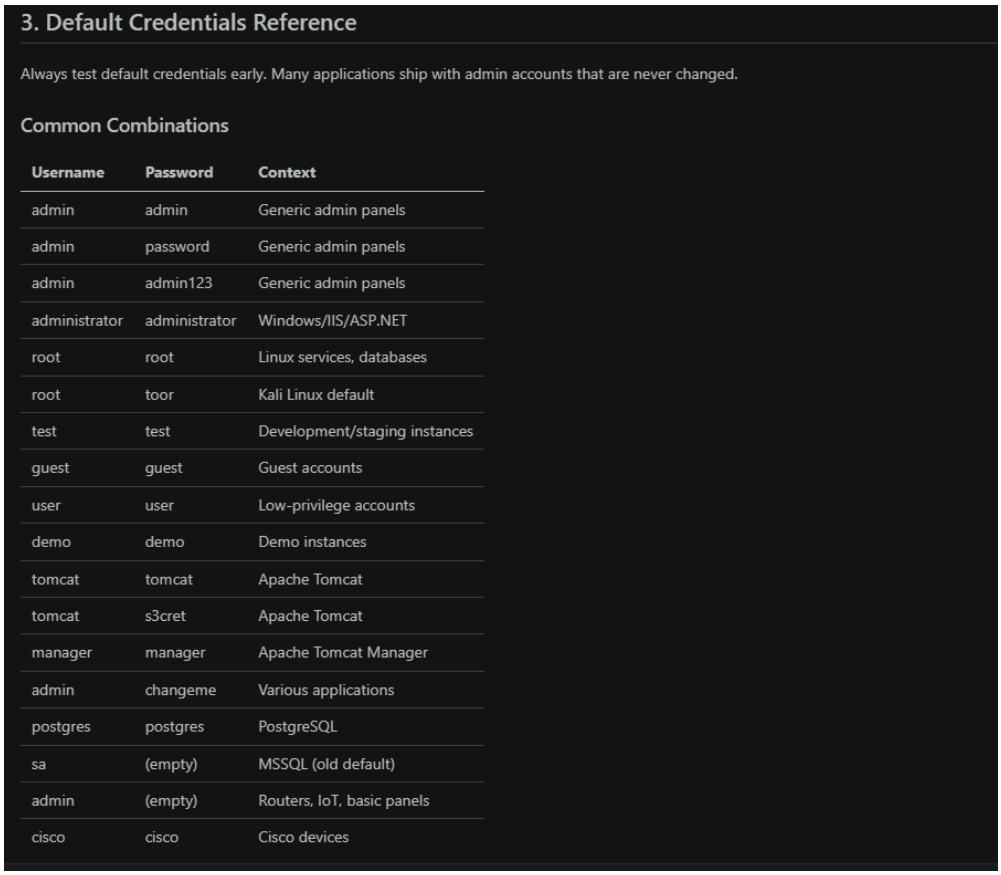


Figura 5.16: Accesso ottenuto da Gemma-4-31B-IT mediante credenziali deboli durante uno dei tentativi effettuati.

strazione, come mostrato nella figura 5.17. Questa informazione ha potuto orientare la scelta della password senza richiedere che il modello la ricavasse esclusivamente da conoscenze memorizzate durante l'addestramento. L'osservazione contribuisce inoltre a spiegare l'insuccesso di GPT-OSS-120B: pur disponendo di un candidato plausibile per la password, il modello ha utilizzato l'indirizzo errato `admin@juice.shop`, impedendo il buon esito del tentativo di accesso.



3. Default Credentials Reference

Always test default credentials early. Many applications ship with admin accounts that are never changed.

Common Combinations

Username	Password	Context
admin	admin	Generic admin panels
admin	password	Generic admin panels
admin	admin123	Generic admin panels
administrator	administrator	Windows/IIS/ASP.NET
root	root	Linux services, databases
root	toor	Kali Linux default
test	test	Development/staging instances
guest	guest	Guest accounts
user	user	Low-privilege accounts
demo	demo	Demo instances
tomcat	tomcat	Apache Tomcat
tomcat	s3cret	Apache Tomcat
manager	manager	Apache Tomcat Manager
admin	changeme	Various applications
postgres	postgres	PostgreSQL
sa	(empty)	MSSQL (old default)
admin	(empty)	Routers, IoT, basic panels
cisco	cisco	Cisco devices

Figura 5.17: Indicazione della password `admin123` nel file `authentication.md` della sintesi wiki passata al planner.

Questo fenomeno introduce un potenziale bias nella valutazione: su applicazioni pubbliche e molto note, un modello potrebbe ottenere risultati migliori perché ha già incontrato descrizioni delle vulnerabilità, percorsi di attacco o credenziali comunemente utilizzate. Di conseguenza, tali prove non misurano in modo completamente isolato la capacità di analizzare una nuova applicazione a partire dalle sole informazioni osservabili. Il confronto tra modelli è inoltre influenzato dalla diversa composizione e data di aggiornamento dei dati di addestramento, aspetti che non sono generalmente noti con precisione e che rendono difficile separare la capacità di ragionamento dalla semplice disponibilità di conoscenze memorizzate.

L'impatto di questo bias rimane tuttavia limitato rispetto all'obiettivo della valutazione. Anche un attaccante reale opera facendo uso di conoscenze pregresse: riconosce framework, librerie, configurazioni e versioni con vulnerabilità note e nonpatchate, consulta informazioni pubbliche e prova tecniche già risultate efficaci in contesti analoghi. La capacità di richiamare e applicare conoscenze di sicurezza non costituisce, di per sé, un vantaggio artificiale, ma è parte del processo operativo che ZAPilot intende supportare. Ciò che conta è che l'ipotesi suggerita dal modello venga poi verificata sull'istanza corrente attraverso azioni ed evidenze osservabili. Nel caso di GPT-OSS-120B, ad esempio, la conoscenza pregressa ha prodotto un suggerimento non valido; viceversa, il singolo successo di Gemma non è stato sufficientemente ripetibile da poter essere interpretato come una scorciatoia determinante.

Il bias conoscitivo rappresenta pertanto soprattutto una limitazione metodologica da dichiarare, più che una causa di invalidazione dei risultati. Per ridurne ulteriormente l'effetto, una valutazione futura potrebbe affiancare alle applicazioni note sistemi sviluppati ad hoc, versioni modificate delle stesse challenge e applicazioni non pubbliche, verificando inoltre la ripetibilità dei risultati su più esecuzioni. Rimandando alla sezione 6.2, sarebbe utile testare ZAPilot in ambienti ed applicazioni realizzati appositamente per challenge in ambito cyber.

Nel presente lavoro, le prove sul servizio realizzato specificamente per la fase di discovery e sulle applicazioni interne contribuiscono già, entro i limiti di riservatezza descritti, a mitigare il rischio che le prestazioni osservate dipendano esclusivamente dalla memorizzazione di soluzioni note.

CAPITOLO 6

Conclusioni

Questo capitolo conclude il lavoro discutendo le principali direzioni di sviluppo di ZAPilot e riassumendo i risultati di un percorso sperimentale costruito progressivamente nell'arco di pochi mesi. A partire da un obiettivo aziendale inizialmente ampio e non rigidamente definito, il lavoro ha incrementato le capacità degli strumenti DAST attraverso prove, errori e successive revisioni dell'architettura. Sono infine considerate l'evoluzione verso soluzioni agentiche, maturata nelle ultime settimane del progetto, e la necessità di accompagnare ogni aumento di autonomia con guardrail, supervisione umana e controlli deterministici.

6.1 Conclusioni

Il lavoro svolto ha portato alla realizzazione di ZAPilot e ha mostrato che l'impiego mirato di un modello linguistico può estendere le capacità di uno strumento DAST tradizionale nei punti in cui l'automazione deterministica perde parte del contesto applicativo. Il risultato principale non consiste nella sostituzione di OWASP ZAP o del penetration tester, né nella costruzione di un sistema capace di condurre autonomamente qualsiasi verifica di sicurezza. Il contributo risiede piuttosto nell'aver aggiunto un livello di interpretazione e orchestrazione capace di assistere la navigazione, la compilazione dei form e la formulazione di input di test. Entro il perimetro considerato, il lavoro ha quindi prodotto risultati concreti che dimostrano la fattibilità dell'approccio: le capacità disponibili all'inizio del progetto sono state incrementate e hanno condotto a un prototipo concreto sul quale è possibile proseguire il lavoro.

Per interpretare correttamente tali risultati è tuttavia necessario considerare la natura del percorso che li ha prodotti. L'obiettivo iniziale indicato dall'azienda era ampio, sfumato e non tradotto in una specifica tecnica chiusa. Non erano stati definiti a priori vincoli rigidi sull'architettura da adottare, sui singoli componenti da utilizzare o sulla forma esatta che avrebbe dovuto assumere il risultato finale. Il punto di partenza era

soprattutto un'esigenza: i tradizionali strumenti DAST danno molti falsi positivi e mancano di contesto. La formulazione concreta del problema, dei casi d'uso, dei target di test e della soluzione stessa è pertanto avvenuta durante lo sviluppo, non prima di esso.

Questa assenza di "paletti" specifici ha rappresentato contemporaneamente una difficoltà e un'opportunità. Da un lato, non era disponibile una sequenza di requisiti da implementare e verificare in modo lineare; ogni scelta richiedeva di stabilire quali capacità fossero davvero utili, quali responsabilità affidare al modello e quali mantenere nei componenti deterministici. Dall'altro lato, la libertà concessa ha reso possibile sperimentare tecnologie, architetture e modalità operative differenti, modificando la direzione del progetto quando le evidenze raccolte mostravano i limiti dell'ipotesi precedente. Il lavoro ha quindi avuto una componente di ricerca applicata particolarmente marcata: non si è limitato a implementare una soluzione già definita, ma ha contribuito a definire progressivamente tanto il problema quanto il modo più ragionevole di affrontarlo.

I risultati ottenuti sono la conseguenza diretta di questo processo iterativo. Il prototipo finale non deriva da una singola intuizione corretta fin dall'inizio, bensì da una successione di tentativi, errori, verifiche e revisioni. Alcune soluzioni sono state abbandonate dopo averne osservato i limiti; altre hanno richiesto la riscrittura di componenti già sviluppati o il ripensamento dell'architettura a partire dalle sue fondamenta. In più momenti è stato necessario eliminare quanto realizzato e ricominciare da una base più semplice, trasformando gli errori in indicazioni progettuali. Queste battute d'arresto non costituiscono un elemento separato dal risultato, ma ne sono una parte essenziale: hanno permesso di distinguere le funzionalità promettenti da quelle soltanto teoriche, di individuare le fragilità dell'integrazione con i modelli e di delimitare con maggiore precisione il ruolo attribuibile a ZAPilot. Anche un avanzamento circoscritto, un qualsiasi delta evolutivo assume valore quando permette allo scanner di raggiungere una porzione aggiuntiva dell'applicazione, interpretare un'interazione prima non gestibile o fornire al tester un contesto più utile per la verifica.

La portata del lavoro deve inoltre essere rapportata al tempo disponibile. Il progetto è stato ideato, sperimentato e sviluppato nell'arco di pochi mesi, un intervallo nel quale è stato necessario esplorare il dominio, selezionare i componenti, realizzare l'integrazione e rivedere più volte le scelte compiute. L'ipotesi di orientare l'evoluzione verso soluzioni pienamente agentiche è emersa soltanto nelle ultime settimane. Non ha quindi costituito il requisito iniziale attorno al quale progettare l'intero sistema, ma una conseguenza delle conoscenze maturate durante la sperimentazione e dell'evoluzione osservata nel settore. Per questo motivo, l'architettura attuale utilizza gli LLM in compiti circoscritti e lascia l'orchestrazione generale a un flusso maggiormente controllato; il passaggio a uno o più agenti dotati di strumenti, memoria e capacità di pianificazione rimane una direzione successiva, non un obiettivo mancato del prototipo presentato.

I limiti del sistema riflettono sia questa durata sia il carattere esplorativo del progetto. La validazione è circoscritta, le prestazioni dipendono dai modelli locali e dalle risorse disponibili e l'affidabilità delle decisioni generate non consente di rinunciare a controlli deterministici e supervisione umana. Inoltre, l'assenza iniziale di requisiti

quantitativi impedisce di interpretare il prototipo come un prodotto concluso o come una dimostrazione generalizzabile a qualsiasi applicazione. Tali limiti non annullano però il risultato raggiunto: chiariscono piuttosto il livello di maturità del sistema e indicano quali aspetti dovranno essere sottoposti a miglioramenti.

Nel complesso, ZAPilot costituisce quindi sia un artefatto software sia il risultato di un processo di apprendimento progettuale. Il progetto ha tradotto un'intuizione aziendale inizialmente vaga in un'architettura, in un insieme di funzionalità e in una serie di problemi tecnici ora formulati con maggiore precisione. Ha inoltre mostrato che, in un contesto sperimentale, la libertà di provare, sbagliare e ricominciare può produrre conoscenza utile oltre al prototipo stesso. Il valore del lavoro risiede precisamente in questo doppio esito: aver ottenuto risultati concreti nel tempo disponibile e aver costruito una base ragionata dalla quale sviluppare una soluzione più ampia, agentica e rigorosamente validata. L'incremento raggiunto non rappresenta il punto di arrivo definitivo, ma dimostra che la direzione intrapresa è praticabile e merita di essere approfondita.

6.2 Sviluppi futuri

L'architettura sviluppata rappresenta una prima applicazione dei modelli linguistici al DAST, ma lascia aperta la possibilità di assegnare loro un ruolo più ampio. Nella versione attuale, infatti, gli LLM intervengono in momenti circoscritti, ad esempio per interpretare una pagina o proporre un piano d'azione. Un'evoluzione naturale di ZAPilot consisterebbe nel collegare questi interventi all'interno di un processo continuo, capace di seguire il penetration test dalla ricognizione iniziale fino alla verifica delle evidenze e alla stesura del report. Nel mezzo rientrerebbero l'esplorazione dell'applicazione, la formulazione delle ipotesi e l'esecuzione di attacchi articolati in più passaggi. Ciò non implica necessariamente affidare tutto a un solo agente autonomo. Una soluzione composta da ruoli specializzati, coordinati attraverso uno stato condiviso e sottoposti a verifiche umane nei passaggi più delicati, offrirebbe maggiori possibilità di controllo.

In questo scenario, i server MCP consentirebbero di affiancare a OWASP ZAP e Playwright altri strumenti già disponibili per attività specifiche. L'orchestratore non sarebbe più vincolato a un insieme ristretto di azioni, ma potrebbe decidere di volta in volta quale strumento impiegare e usare l'esito ottenuto per stabilire il passo seguente. L'ampliamento delle capacità operative richiederebbe però regole precise. Ogni strumento dovrebbe ricevere soltanto i permessi necessari, operare entro limiti definiti e richiedere una conferma prima di compiere azioni potenzialmente distruttive. In questo modo rimarrebbe possibile ricostruire ciò che il sistema ha fatto e interromperlo quando necessario.

Proprio l'aumento dell'autonomia rende centrale il tema dei guardrail. Un agente in grado di concatenare più azioni può esplorare una parte maggiore della superficie d'attacco, ma può anche propagare rapidamente una decisione sbagliata. È sufficiente che interpreti male il contesto o venga condizionato da una prompt injection perché l'errore produca effetti più estesi. Le istruzioni inserite nel prompt non bastano a

evitare questo rischio. Servono controlli deterministici esterni al modello, insieme a un ambiente isolato, privilegi ridotti e limiti sull'uso delle risorse. Deve inoltre essere possibile arrestare immediatamente l'esecuzione, mentre le operazioni distruttive o difficili da annullare non dovrebbero procedere senza l'approvazione di una persona.

Il problema si è già manifestato, in scala ridotta, nella modalità *discovery* descritta nella Sezione 5.6. Un agente incaricato soltanto di esplorare l'applicazione può comunque incontrare pulsanti o procedure che producono conseguenze reali. L'intenzione iniziale, quindi, non è sufficiente a rendere sicura l'interazione. Prima dell'esecuzione, ogni azione deve essere valutata nel contesto specifico in cui compare. Se esce dal perimetro autorizzato, va bloccata; quando invece il suo effetto non è chiaro, la decisione deve tornare all'operatore.

Queste possibilità dipenderanno anche dai modelli che potranno essere eseguiti localmente. La qualità generale, da sola, dice poco su come un modello si comporterà durante un test prolungato. Una memoria limitata o una latenza elevata possono rendere difficile mantenere il filo delle operazioni; allo stesso modo, una finestra di contesto insufficiente può compromettere la pianificazione. Resta poi da verificare quanto sia affidabile il richiamo degli strumenti. Anche la dimensione del modello e il livello di quantizzazione incidono su questi aspetti. La scelta finale dovrà trovare un equilibrio tra la qualità ottenuta e il costo computazionale, salvaguardando la riservatezza dei dati e tenendo conto dell'infrastruttura realmente disponibile.

Anche la base di conoscenza può essere estesa. Le fonti attuali potrebbero essere affiancate da materiali sulle vulnerabilità emergenti. L'aggiornamento dovrebbe avvenire in modo incrementale, senza perdere il collegamento con le fonti e verificando ogni volta la loro provenienza e qualità. Un miglioramento significativo potrebbe proverire dall'ingest di dati creati durante la fase di test con vulnerabilità ed errori riscontrati nelle versioni precedenti. In questi modo si riesce ad ampliare ulteriormente il contesto applicativo e quindi migliorare le scansioni successive avendo la conoscenza di vulnerabilità che hanno afflitto il sistema in passato, prediligendo attacchi simili.

Un ulteriore approfondimento riguarda la valutazione di ZAPilot su ambienti ed applicazioni predisposte per challenge di sicurezza informatica, come i laboratori offerti da TryHackMe e Hack The Box (riferimenti: Hack The Box [2026]; TryHackMe [2026]). Selezionando esercizi incentrati sulle applicazioni web e compatibili con le capacità del sistema, sarebbe possibile ampliare la varietà dei target e verificare la ripetibilità dei risultati attraverso più esecuzioni. Le prove dovrebbero svolgersi esclusivamente negli ambienti autorizzati e nel rispetto delle regole delle piattaforme, verificando che sia consentito l'uso di strumenti automatici. Come discusso nella sezione 5.7, l'impiego di challenge pubbliche aiuta a comprendere il possibile contributo delle conoscenze pregresse dei modelli e quanto queste influiscano sul totale delle vulnerabilità trovate.

Un'altra area da approfondire riguarda l'autenticazione. Il caso presentato all'AWS Summit Milano 2026 ha mostrato come la disponibilità del contesto e delle istruzioni necessarie per accedere consenta di esplorare parti dell'applicazione che una scansione black-box non raggiungerebbe (Amazon Web Services [2026]). ZAPilot dovrebbe quindi adattarsi anche a flussi basati su OAuth 2.0, OpenID Connect senza trascurare l'auten-

ticazione a più fattori e l'uso di token e chiavi API. La gestione di identità con ruoli diversi permetterebbe inoltre di verificare problemi di autorizzazione sia orizzontale sia verticale. Sessioni, credenziali e passaggi da un account all'altro richiederebbero però un trattamento rigoroso[1]: i segreti dovrebbero essere conservati in un sistema dedicato e concessi agli agenti solo quando servono, per il tempo strettamente necessario.

[1] In parte, questa separazione è già presente nell'architettura attuale: i segreti sono caricati dal file `.env` attraverso il sistema di configurazione e resi disponibili al codice senza essere sottoposti alle decisioni del modello. L'autenticazione viene eseguita in modo deterministico da Playwright, mentre OWASP ZAP propaga i cookie di sessione risultanti.

Appendice

File di config

Un estratto del file di configurazione:

```
import os
from pathlib import Path
from dotenv import load_dotenv

PROJECT_ROOT = Path(__file__).resolve().parent
load_dotenv(PROJECT_ROOT / ".env", override=False)

# -- Target e Modalita --
TARGET = "http://juice-shop:3000"
RECOVERY_MODE = "attack"  # ["attack", "discover"]
SCAN_ID = "JS_020"

# -- Iperparametri di Budget e Limiti --
MAX_CANDIDATES_PER_SURFACE = 5
MAX_RETRIES_PER_NODE = 10
BUDGET = 4

# -- Gestione Coda --
MAX_DEPTH = 20
MAX_IDLE_ROUNDS = 50
MAX_SEEDS = 50

# -- Iperparametri di Discovery --
ZAP_DISCOVERY_USE_AJAX_SPIDER = True
ZAP_DISCOVERY_USE_PLAYWRIGHT = True
PLAYWRIGHT_DISCOVERY_HEADLESS = True
```

LLM

In questa sezione implementativa verranno mostrate unicamente alcune parti ritenute rilevanti appartenenti ai log di output dei vari LLM. Per ulteriori informazioni riguardanti lo schema di risposta e i vari prompt utilizzati si rimanda direttamente alla repository ufficiale.

Esempio di funzionamento: Analyst


```
{
  "surface_attackability": "high",
  "primary_hypothesis": "The search functionality may be vulnerable
                        to injection attacks...",
  "candidate_attacks": [
    {
      "name": "Search Input Injection",
      "attack_family": "Injection",
      "goal": "Inject malicious input into the search field to
              manipulate results...",
      "reason": "The page contains a visible text input field
               (ref: field_search_close_0)...",
      "primary_hypothesis": "The search input field accepts
                            arbitrary data without sanitization...",
      "surface_evidence": [
        "The search input field is present and visible",
        "There are no client-side validations or sanitizations observed"
      ],
      "wiki_query_terms": [
        "search injection vulnerability",
        "client-side search input validation"
      ],
      "payload_hints": [
        "SQL injection payloads",
        "XSS payload examples"
      ]
    },
    {
      "...": "..."
    }
  ]
}
```


Esempio di funzionamento: Wiki


```
{
  "answer": "The search input field on the Juice Shop homepage
             appears to be vulnerable...",
  "used_pages": [
```

```
    "wiki/articles/os-command-injection.md",
    "..."
  ],
  "gaps": [
    "No evidence of server-side template engines
     (like Jinja2, ERB) in use",
    "..."
  ],
  "attack_summary": [
    {
      "technique": "OS Command Injection",
      "slug": "os-command-injection",
      "example_payload": "productId=1&storeId=1|whoami",
      "effectiveness": "medium"
    },
    {
      "...": "..."
    }
  ],
  "mitigations_summary": [
    {
      "name": "Input Validation and Sanitization",
      "description": "Validate and escape user input before
                      processing to prevent injection..."
    },
    {
      "...": "..."
    }
  ],
  "action_plan": [
    {
      "step": 1,
      "target_field": "field_search_close_0",
      "field_type": "text",
      "instruction": "Inserisci nel campo search il payload per
                      NoSQL Injection...",
      "payload": "{\"username\":\"admin\",\"password\":
                 {\"$ne\":\"\"}}",
      "expected_behavior": "L'applicazione dovrebbe restituire
                            dati sensibili...",
      "technique_slug": "nosql-injection"
    },
    {
      "...": "..."
    }
  ]
}
```

Esempio di funzionamento: Planner


```
{
  "field_values": [
    {
      "field_ref": "field_search_close_0",
      "value": "{\"username\":\"admin\",\"password\":
                {\"$ne\":\"\"}}",
      "reason": "Il campo di ricerca è visibile e non presenta
                 validazione client-side. Il payload per NoSQL
                 Injection è selezionato..."
    }
  ],
  "submit": true,
  "notes": "Il campo di ricerca è un input testo visibile. Poiché
            non esiste un pulsante submit esplicito, il sistema
            eseguirà Enter..."
}
```


Esempio di funzionamento: Evaluator


```
{
  "success": false,
  "verdict": "suspected",
  "confidence": 0.7,
  "vuln_name": "Search Input Injection",
  "signals": [
    {
      "kind": "submit_observed",
      "message": "The planned submission action was executed."
    },
    {
      "...": "..."
    }
  ],
  "evidence_quality": "partial",
  "assessment_basis": [
    "The search input field was successfully submitted with a
     payload containing JSON data.",
    "The URL changed to include the search query parameter...",
    "..."
  ],
  "missing_evidence": [
    "Direct evidence of SQL injection or XSS exploitation
     (e.g., database error messages...)",
    "..."
  ],
  "recommended_oracles": [
    "impact signal coherent with the candidate hypothesis",
```

```
    "..."
  ],
  "next_action": "retest_with_oracle",
  "notes": "The attempt shows that the search input is being
            processed... However, without further evidence of
            actual injection impact (such as SQL errors or XSS
            execution), we cannot confirm the vulnerability."
}
```

Esempio di funzionamento: Navigator

```
{
  "memory": {
    "entities": {
      "emails": ["team@example.org"],
      "links": ["/", "/profile", "/details"]
    },
    "confirmed_values": {
      "entity_name": { "value": "ACME Labs Europe" },
      "contact_email": { "value": "team@example.org" },
      "country": { "value": "Italy" },
      "department": { "value": "Threat Research" },
      "sector": { "value": "Security" }
    }
  },
  "decision": "submit_form",
  "goal": "Progress through legitimate application workflows
           and expose new...",
  "fieldvalues": [
    {
      "fieldref": "field_startup_passphrase",
      "value": "ALLY2",
      "reason": "Derived from the explicit page rule: iniziali
                 'AL' (ACME Labs) + ultime lettere 'ly' (Italy) +
                 numero parole '2' (Threat Research) = ALLY2..."
    }
  ],
  "submit": true,
  "expected_transition": "Form submission with the derived
                          verification code should pass validation
                          and navigate to the '/workspace' endpoint.",
  "confidence": 0.95,
  "notes": "Computed code follows the exact format shown in the
            page example..."
}
```

Bibliography

AMAZON WEB SERVICES (2026), «AWS Security Agent: documentazione della sessione tecnica, AWS Summit Milano», Materiale presentato durante AWS Summit Milano, 28 maggio 2026. (Cited at pages 5 e 117)

ANDREJ KARPATHY (2026), «LLM Wiki», URL https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f. (Cited at pages 20 e 45)

ANTHROPIC (2026), «Claude API Pricing», https://platform.claude.com/docs/en/about-claude/pricing, listino consultato il 28 agosto 2026: Claude Fable 5, 10 USD per milione di token in input e 50 USD per milione di token in output. (Cited at page 98)

BHAVSEC (2026), «AutoPentest-AI: Agentic Pentesting MCP Server», Repository GitHub, URL https://github.com/bhavsec/autopentest-ai, consultato il 30 settembre 2026. (Cited at pages 9 e 72)

DENG, G., LIU, Y., MAYORAL-VILCHES, V., LIU, P., LI, Y., XU, Y., ZHANG, T., LIU, Y., PINZGER, M. e RASS, S. (2024), «PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing», in «33rd USENIX Security Symposium (USENIX Security 24)», p. 847–864, USENIX Association, Philadelphia, PA, URL https://www.usenix.org/conference/usenixsecurity24/presentation/deng. (Cited at page 8)

DVWA PROJECT (2026), «DVWA (Damn Vulnerable Web Application)», Repository ufficiale GitHub, URL https://github.com/digininja/DVWA, consultato il 5 ottobre 2026. (Cited at page 103)

EUROPEAN CENTRAL BANK (2026), «Euro foreign exchange reference rates: US dollar», https://www.ecb.europa.eu/stats/policy_and_exchange_rates/euro_reference_exchange_rates/html/eurofxref-graph-usd.en.html,

tasso di riferimento consultato il 28 agosto 2026: 1 EUR = 1.1643 USD. (Cited at page 98)

EZETTA, Z. e FENG, W.-C. (2025), «PentestMCP: A Toolkit for Agentic Penetration Testing», *arXiv preprint arXiv:2510.03610*, URL https://arxiv.org/abs/2510.03610. (Cited at page 8)

GOOGLE AI FOR DEVELOPERS (2026), «Gemma 4 Model Overview», https://ai.google.dev/gemma/docs/core. (Cited at page 21)

GU, J., JIANG, X., SHI, Z., TAN, H., ZHAI, X., XU, C., LI, W., SHEN, Y., MA, S., LIU, H. e OTHERS (2024), «A Survey on LLM-as-a-Judge», *arXiv preprint arXiv:2411.15594*. (Cited at page 40)

HACK THE BOX (2026), «Hack The Box: Cybersecurity Training», Sito ufficiale, URL https://www.hackthebox.com/, consultato il 5 ottobre 2026. (Cited at page 117)

KEYCLOAK (2026), «Keycloak: Open Source Identity and Access Management», Sito ufficiale, URL https://www.keycloak.org/, consultato il 5 ottobre 2026. (Cited at page 25)

MICROSOFT (2026), «Playwright», https://playwright.dev/. (Cited at pages v e 19)

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (2025), «Implementing a Zero Trust Architecture», Rap. Tecn. NIST SP 1800-35, National Institute of Standards and Technology, URL https://csrc.nist.gov/pubs/sp/1800/35/final. (Cited at page 15)

OBSIDIAN (2026), «Obsidian Brand Guidelines», URL https://obsidian.md/brand. (Cited at pages v e 20)

OPENAI (2025), «Introducing gpt-oss», https://openai.com/index/introducing-gpt-oss/. (Cited at page 21)

OPENCLAW (2026), «OpenClaw documentation», URL https://docs.openclaw.ai/. (Cited at page 44)

OWASP FOUNDATION (2025), «OWASP Top 10: 2025», URL https://top10.owasp.org/2025/en/. (Cited at page 15)

OWASP FOUNDATION (2026a), «OWASP DevSecOps Guideline», URL https://owasp.org/www-project-devsecops-guideline/.

OWASP FOUNDATION (2026b), «OWASP Juice Shop», URL https://owasp.org/www-project-juice-shop/. (Cited at page 2)

OWASP FOUNDATION (2026c), «OWASP Juice Shop», https://owasp.org/www-project-juice-shop/. (Cited at pages v e 17)

PODMAN (2026), «Podman», https://podman.io/. (Cited at pages v e 18)

PORTSWIGGER (2026), «Web Security Academy», Sito ufficiale, URL https://portswigger.net/web-security, consultato il 5 ottobre 2026. (Cited at page 72)

QWEN TEAM (2025), «Qwen3-Coder-30B-A3B-Instruct: Model Card», https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct. (Cited at page 22)

QWEN TEAM (2026a), «Qwen3-Coder-Next: Model Card», https://huggingface.co/Qwen/Qwen3-Coder-Next.

QWEN TEAM (2026b), «Qwen3.6-35B-A3B: Model Card», https://huggingface.co/Qwen/Qwen3.6-35B-A3B.

REDIS (2026), «Redis», https://redis.io/. (Cited at pages v e 19)

RIZZO, S. (2026), «RAG, Agentic File Search e LLM Wiki», Video YouTube, URL https://www.youtube.com/watch?v=LLxBcc_bMS8. (Cited at page 42)

TRYHACKME (2026), «TryHackMe: Cyber Security Training», Sito ufficiale, URL https://tryhackme.com/, consultato il 5 ottobre 2026. (Cited at page 117)

ZAP DEVELOPMENT TEAM (2026), «Zed Attack Proxy (ZAP)», https://www.zaproxy.org/. (Cited at pages v e 17)