ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Sicurezza delle API REST: analisi sperimentale di vulnerabilità comuni e strategie di difesa

Tesi di laurea in:
Programmazione di Reti

Relatore
Prof. Andrea Piroddi

Candidato
Leonardo Grimaldi

IV Sessione di Laurea
Anno Accademico 2024-2025

Sommario

Le vulnerabilità web sono una delle principali sfide nel campo della sicurezza informatica, in quanto possono causare conseguenze significative come furto di dati sensibili, accesso non autorizzato ai sistemi e la loro compromissione. Questa tesi analizza le vulnerabilità più diffuse nelle API REST attraverso uno studio pratico condotto su OWASP Juice Shop, un'applicazione web appositamente sviluppata per scopi didattici in ambito sicurezza. Il documento illustra inoltre la metodologia sistematica per l'identificazione e la risoluzione di queste vulnerabilità, evidenziando l'impiego di strumenti di testing specializzati quali Zed Attack Proxy (ZAP) e Postman, che hanno permesso sia l'analisi approfondita che la validazione delle soluzioni implementate.

Ai miei genitori e a mio fratello

Indice

Elenco delle figure

Elenco dei codici

Capitolo 1

Introduzione

Le vulnerabilità web sono una delle principali minacce alla sicurezza delle applicazioni odierne. Di recente, una falla di sicurezza su un portale web automobilistico ha portato all'accesso completo su un sistema aziendale statunitense, consentendo anche l'accesso fisico ad automobili e segreti aziendali[1].

In questa tesi sono state esplorate alcune delle vulnerabilità più comuni nelle **API REST** con l'ausilio di un'applicazione web vulnerabile chiamata **OWASP Juice Shop**. Inoltre è stato mostrato codice di esempio per ogni vulnerabilità elencata e sono state proposte delle contromisure per mitigare i rischi associati.

L'obiettivo di questa tesi è di fornire una panoramica sulle insidie di sicurezza più comuni nello sviluppo web e quali minacce informatiche possano derivarne, nonché implementare il codice risolutivo validandone l'efficacia attraverso test specifici.

La metodologia adottata per l'implementazione di contromisure si è basata su una prima analisi dell'applicazione per trovare i punti di attacco. Dopodiché si è esaminato il codice associato alla pagina vulnerabile e consultate le documentazioni in rete per trovare i migliori standard e modi di aggiornamento (*patching*) del codice. Infine, dopo aver applicato le modifiche, sono stati eseguiti dei test per verificare la risoluzione della vulnerabilità e l'efficacia della contromisura adottata.

Struttura della tesi Il primo capitolo di Introduzione presenta il contesto generale e le motivazioni per i successivi capitoli. Il secondo capitolo di Background definisce alcuni concetti chiave per la comprensione della sicurezza web tra cui le

Application Programming Interface (API) e la lista OWASP TOP 10 con figure e riferimenti pratici. Inoltre, illustra i programmi di testing della sicurezza utilizzati nella fase di analisi e implementazione che costituiscono una parte importante della metodologia adottata.

A seguire, l'Analisi del dominio introduce l'applicazione vulnerabile *Juice Shop* che verrà sottoposta ad attacchi informatici mirati che consentiranno di approfondire i concetti teorici imparati e di utilizzare i strumenti come Zed Attack Proxy (ZAP) per trovare vulnerabilità. Questa parte contiene numerose immagini e passi riproducibili di attacchi informatici che forniscono la base solida per la risoluzione degli stessi nel capitolo successivo. Implementazione è il capitolo centrale della tesi che contiene la descrizione delle quattro vulnerabilità comuni, il codice vulnerabile dentro *Juice Shop*, la spiegazione della falla di sicurezza, il payload di attacco e infine la contromisura adottata per risolvere il problema. Infine, il capitolo di Conclusioni riassume il lavoro svolto, i limiti e i proseguimenti futuri.

Capitolo 2

Background

In questo capitolo vengono descritti i concetti fondamentali che riguardano la sicurezza web sia a livello teorico che pratico.

La prima definizione importante e ricorrente è di **vulnerabilità**, ovvero una debolezza nel sistema software che può essere sfruttata per comprometterne la sicurezza[2].

2.1 API REST

Le **API** sono un insieme di regole e protocolli che consentono la comunicazione tra servizi o programmi. Nell'ambito web un software che utilizza le API è sicuramente il browser che continuamente scambia informazioni con i siti web per mostrare contenuti all'utente. Per far fronte alla complessità del Web è stato introdotto lo stile architetturale **Representational State Transfer (REST)** che impone nuove condizioni sulle API per renderne più semplice l'utilizzo. Alcune di queste sono:

- **Client-Server**: l'utente (client) richiede servizi al server tramite richieste HyperText Transfer Protocol (HTTP)

- **Stateless**: le richieste sono indipendenti e contengono tutte le informazioni necessarie per essere elaborate

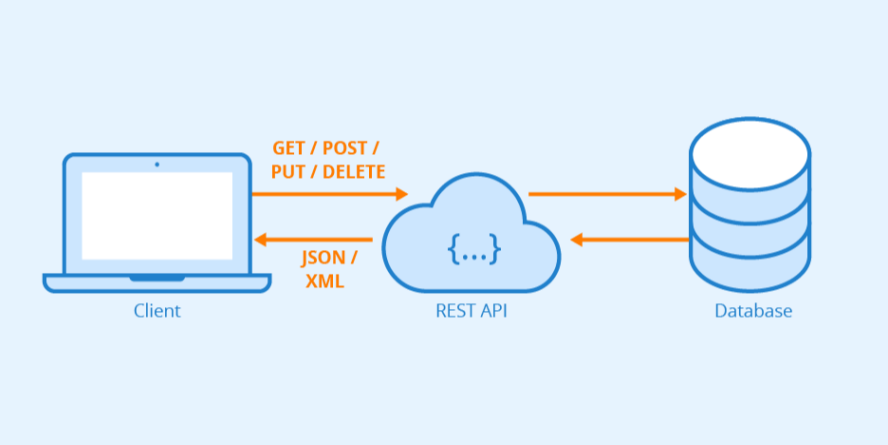

Figura 2.1: Diagramma delle interazioni tra i tre principali attori in un'architettura REST[4]

- **Uniform Interface**: si addotta un formato standard di comunicazione, indipendente dal programma
- **Cacheable**: le risposte del server devono essere contrassegnate come memorizzabili o non memorizzabili nella cache del client

Nell'analisi di sicurezza di un applicazione la esposizione di API è il modo principale per l'individuazione di vulnerabilità. L'architettura client-server del primo punto rende possibile l'interazione tra un utente malintenzionato e il server con i metodi HTTP `GET`, `POST`, `PUT`, `DELETE` che devono essere gestiti in modo appropriato per evitare che l'utente legga o modifichi dati sensibili. La fig. 2.1 mostra le interazioni tra il client (computer) e le REST API del server tramite le richieste elencate prima. Si noti la presenza del database, che non viene raggiunto direttamente dal client e il formato *JSON* o *XML* usato per la risposta del server.

La proprietà stateless del secondo punto mostra un problema di sicurezza molto diffuso: se i servizi devono avere un permesso specifico, tutte le richieste HTTP devono essere accompagnate da una chiave di sicurezza o token per autenticare e autorizzarne l'uso. Poiché il server non può inferire l'identità del client da richieste precedenti, è necessario che ogni richiesta contenga le credenziali di accesso.

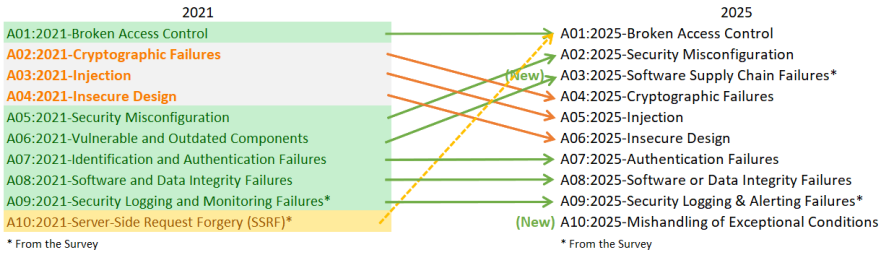

Figura 2.2: Cambiamenti di classifica delle vulnerabilità web più comuni tra gli anni 2021 e 2025[7]

Tuttavia, nello sviluppo delle API è comune dimenticare di implementare questi controlli di accesso, esponendo il server a vulnerabilità di *Broken Access Control*. La sezione 4.3.1 mostra un esempio di questo tipo di vulnerabilità tramite API e descrive alcune conseguenze legate a essa.

2.2 OWASP TOP 10

La lista OWASP TOP 10[5] è il riferimento centrale per le vulnerabilità web più comuni ed è gestita dal team Open Web Application Security Project (OWASP). È stata ampiamente utilizzata per la ricerca di vulnerabilità e la loro risoluzione in questa tesi. Essa contiene dieci delle più comuni categorie di vulnerabilità web, classificate tramite dati raccolti da aziende, organizzazioni e anche sondaggi. La lista viene aggiornata ogni quattro anni e la fig. 2.2 mostra i cambiamenti dalla scorsa edizione 2021, dove si noti il Broken Access Control è rimasto al primo posto. Per questa tesi si è scelto di analizzare le categorie descritte nelle seguenti sottosezioni.

2.2.1 Broken Access Control

Il **controllo degli accessi** è un processo di verifica che consente al server di determinare i permessi di accesso alle risorse. Coinvolge i due più importanti metodi di verifica dell'identità, ovvero l'**autenticazione** e l'**autorizzazione**. Il primo consiste nel verificare l'identità dell'utente tramite credenziali di accesso

Figura 2.3: Richiesta API a `/rest/basket/6` con Authorization header a riga 6

mentre il secondo di assegnare i permessi all'utente associato. Per esempio, per accedere a una pagina riservata come la lista di utenti registrati, è norma chiedere l'accesso tramite un username e password. Similmente, l'accesso a un endpoint API riservato richiederebbe un token di autenticazione, come un *JSON Web Token* (JWT) dentro all'header di autenticazione della richiesta HTTP[8].

Quando la pagina o l'API non effettua un controllo di sicurezza adeguato e consente l'accesso a utenti non autorizzati, si parla di **Broken Access Control**. La fig. 2.3 mostra un esempio di richiesta API con header di autorizzazione contenente un token JWT. Se il server non verifica la validità del token e consente l'accesso a utenti non autorizzati, si configura una vulnerabilità di Broken Access Control.

Le vulnerabilità Broken Access Control si trovano al primo posto nella lista poiché sono le più comuni e relativamente facili da sfruttare e concedono accesso potenziale a infrastrutture amministrative che espongono dati sensibili.

2.2.2 Injection

I web server utilizzano ampiamente database per memorizzare e recuperare dati. Inoltre, elaborano dati inseriti dagli utenti per fornire pagine dinamiche oppure servizi personalizzati. Un attacco di tipo Injection si verifica quando un attaccante riesce a inserire comandi all'interno di caselle input (non sanificate) e farle eseguire al server.

Le Injection si trovano al 5° posto della lista e si dividono in due sottocategorie principali: SQL injection e Cross-Site Scripting (XSS).

SQL Injection

In una vulnerabilità di tipo SQL Injection l'attaccante compone una interrogazione Structured Query Language (SQL) specifica che consente di reperire dati sensibili dal database oppure di manipolare il database in modi non previsti (aggiunta di dati, eliminazione). La sezione 4.2.1 mostra un esempio di questa vulnerabilità importante e come viene risolta.

Cross-Site Scripting (XSS)

Le XSS sono vulnerabilità che coinvolgono i client, gli utenti del sito, oltre che il server stesso. Spesso sfruttano codice *JavaScript* per eseguire script maliziosi sulla macchina dell'utente e ottenere dati sensibili dal sito, come la password o i dati di pagamento.

I loro vettori di attacco sono Uniform Resource Locator (URL) che contengono parametri malevoli come `https://localhost:3000/search?q=<script src="javascript:alert('codice malevolo')"></script>` che vengono inviati alla vittima (client XSS) oppure pagine del web server compromesse con codice malevolo eseguito all'apertura (server XSS). La sezione 4.1 contiene la risoluzione e alcune dimostrazioni di questa vulnerabilità.

La fig. 2.4 mostra il diagramma di sequenza di un attacco XSS di tipo *stored*, dove il codice malevolo viene memorizzato sul server e inviato a ogni utente che visita la pagina compromessa. Questo tipo di attacco può essere anche sfruttato tramite la vulnerabilità Broken Access Control, risolta in sezione 4.3.1.

2.2.3 Security Misconfiguration

Nelle specifiche di OWASP viene definito che: "La configurazione errata avviene quando un sistema, applicazione o servizio cloud è configurato in modo non corretto dal punto di vista della sicurezza, creando vulnerabilità"[11]. Questa definizione comprende la visualizzazione di errori, uso di credenziali predefinite e abilitazione di pagine o servizi non necessari e quindi anche API. Durante lo sviluppo web è comune dimenticarsi di disabilitare alcune funzionalità o endpoint API che non

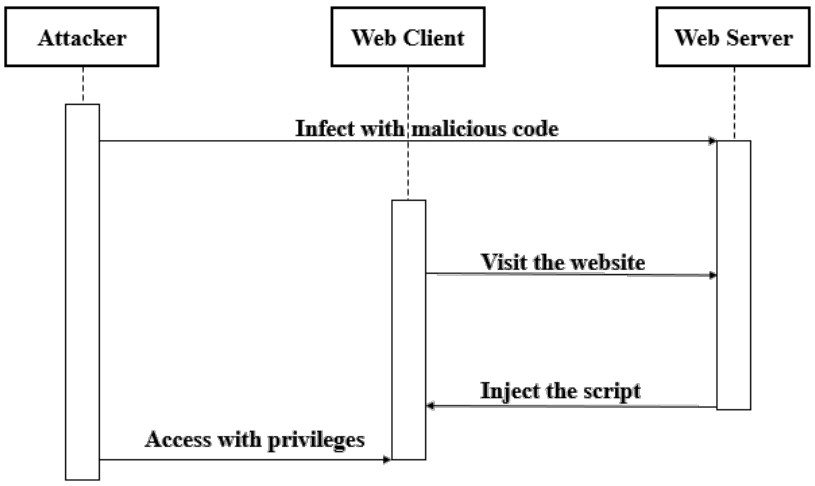

Figura 2.4: Diagramma della sequenza di attacco XSS di tipo *stored*[10]

sono più necessari, esponendo il server a potenziali attacchi informatici. La sezione 4.4.1 mostra un esempio di questa vulnerabilità e come viene risolta.

2.3 Zed Attack Proxy (ZAP)

La ZAP è uno strumento fondamentale per la verifica di sicurezza delle applicazioni web. Funziona come proxy tra il browser e il server web, consentendo di intercettare, modificare e analizzare le richieste e le risposte HTTP. Tra le sue funzionalità principali vi sono:

- Intercettazione delle richieste e risposte HTTP
- Scansione automatica delle vulnerabilità
- Analisi delle sessioni e gestione dei cookie
- Fuzzing delle richieste per individuare vulnerabilità

In questa tesi si è fatto uso principalmente della intercettazione delle richieste e risposte. Nella fig. 2.5 viene mostrata la schermata principale di ZAP, con la

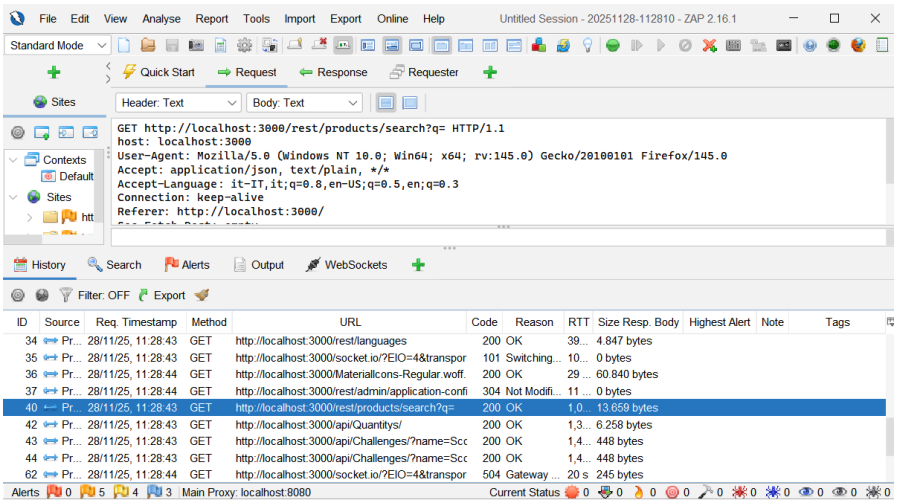

Figura 2.5: Finestra principale di ZAP con la lista delle richieste intercettate e il pannello di analisi della richiesta selezionata.

cronologia delle richieste HTTP effettuate dal browser nella finestra inferiore e al centro il pannello di analisi della richiesta selezionata, da cui è possibile modificare i parametri prima di inviarla nuovamente al server con il click destro. La sezione 4.4.1 mostra un esempio di come ZAP sia stato utilizzato per intercettare una richiesta di caricamento file e modificarne il contenuto per testare la vulnerabilità di *Security Misconfiguration*. La fig. 4.7 mostra come con ZAP sia possibile modificare una richiesta HTTP per inviare un file di tipo non consentito e sfruttare una vulnerabilità API del server.

2.4 Postman

Postman è uno strumento molto diffuso, usato per testare e sviluppare API. Consente di creare, inviare e analizzare richieste HTTP in modo semplice e intuitivo. Può intendersi come un'alternativa a ZAP (soprattutto al *Request Editor*) per l'invio di richieste HTTP, ma con un'interfaccia più user-friendly. Tra le sue funzionalità principali vi sono:

- Creazione di richieste HTTP personalizzate
- Gestione delle collezioni di richieste

- Test automatizzati delle API
- Generazione di documentazione per le API

Nel corso della tesi Postman è stato utilizzato principalmente per inviare richieste `PUT` e `POST` alle API del server per testare le vulnerabilità di Broken Access Control e Security Misconfiguration come in fig. 3.8.

Capitolo 3

Analisi del dominio

3.1 Applicazione vulnerabile

Prima ancora di incominciare la valutazione di sicurezza è necessario scegliere un sito web vulnerabile. In rete sono presenti una vasta gamma di applicazioni per la simulazione di attacchi informatici, come Damn Vulnerable Web Application (DVWA)[12] (basato su PHP) e OWASP Juice Shop[13] (basato su Node.js). In questa tesi si è scelto di utilizzare quest'ultima per modernità di tecnologie e aggiornamenti.

3.1.1 OWASP Juice Shop

Juice Shop è un sito web che simula un portale di vendita per prodotti di vario tipo. Presenta numerose funzionalità tra cui la registrazione, ordinazione, invio di recensioni, commenti e molto altro ancora, il che la rende un'applicazione completa. Essendo ideata per l'apprendimento e la pratica del testing di sicurezza, presenta anche numerose vulnerabilità web che fanno parte della lista OWASP TOP 10.

3.1.2 Architettura

L'applicazione web si divide in due componenti: il *front end* e il *back end*. Il front end gestisce la parte visiva dell'applicazione e fruibile dall'utente finale. È stato sviluppato con Angular, un framework moderno che consente la modulazione di

Figura 3.1: Logo dell'applicazione OWASP Juice Shop.

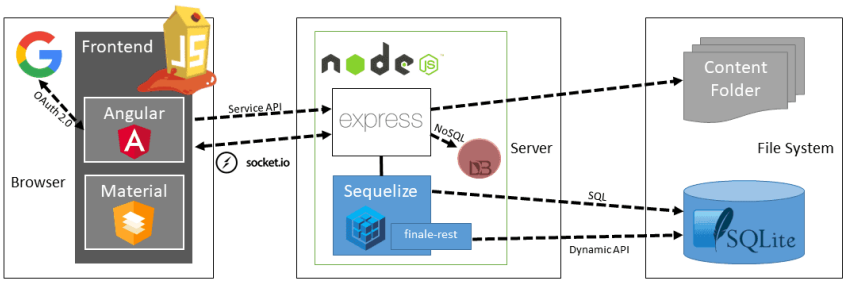

Figura 3.2: Diagramma dell'architettura dell'applicazione OWASP Juice Shop[16]

elementi web scritti con TypeScript. Il back end costituisce la parte più complessa dell'applicazione, che interagisce con i database e fornisce i servizi necessari per molte funzionalità del sito. Il web server è fornito da Node.js che si affida alla libreria Express per le REST API e Sequelize per le interrogazioni al database SQLite.

La fig. 3.2 mostra tutte le componenti di sviluppo dell'applicazione. Le frecce nere indicano le chiamate o interazioni API tra di esse. Una complessità maggiore dell'applicazione porta inevitabilmente a un rischio più elevato di attacchi informatici.

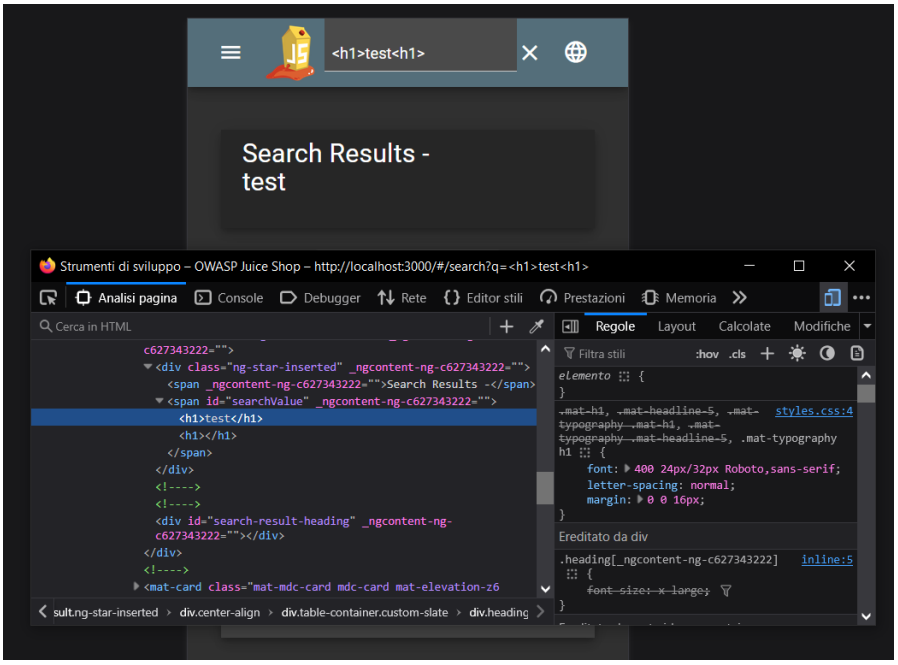

Figura 3.3: Probing della barra di ricerca con tag *HTML* `<h1>`. L'*ispeziona elementi* mostra la presenza del tag nell'albero della pagina suggerendo una possibile vulnerabilità.

3.2 Esplorazione

Una parte essenziale della risoluzione di vulnerabilità è l'analisi delle pagine web e delle richieste effettuate senza avere accesso al codice, il cosiddetto *black box testing*, per individuare i principali punti e vettori di attacco. In questa sezione verranno effettuati alcuni attacchi per identificare i punti più critici dell'applicazione che consentiranno di trovare con più facilità il codice vulnerabile durante la fase di *patching* del capitolo 4.

3.2.1 Barra di ricerca

Nella pagina principale di Juice Shop è presente una barra di ricerca che non esegue la sanificazione dell'input, ovvero consente l'inserimento di codice HTML e JavaScript. È possibile notare questo meccanismo nella fig. 3.3 dove viene inserito nella barra di ricerca il seguente codice `<h1>Test<h1>` e analizzando la pagina modificata si trova l'elemento inserito all'interno del DOM.

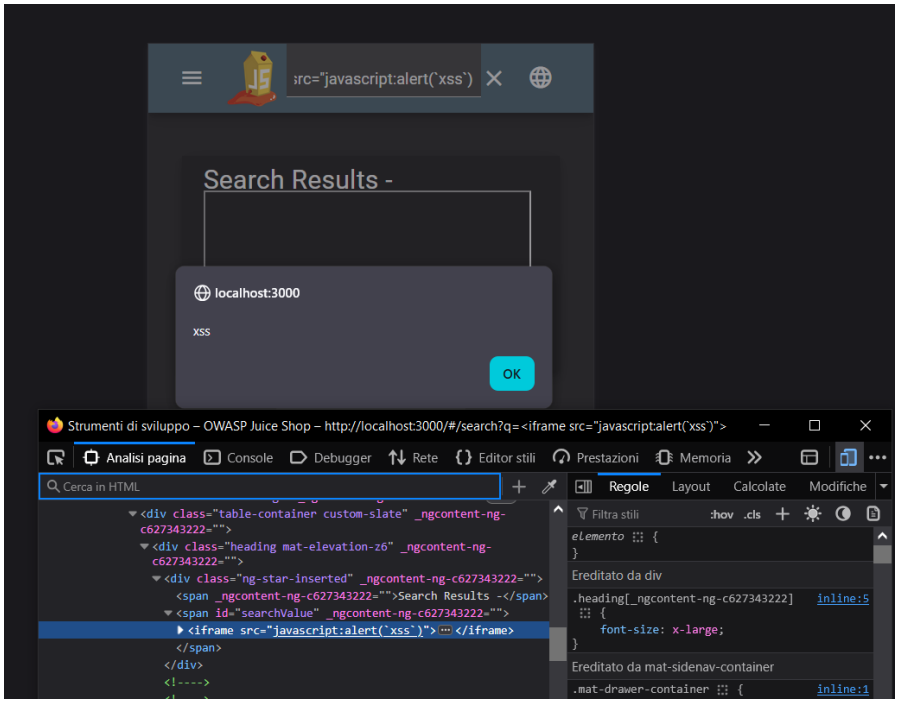

Figura 3.4: Risultato di esecuzione del script malizioso contenuto dentro un tag `<iframe>`. L'alert dimostra che il codice JavaScript è stato eseguito con successo.

Dopo aver individuato questo punto di attacco, si tenta di eseguire un payload di tipo XSS inserendo nella barra di ricerca il seguente codice: `<iframe src="javascript:alert(‘xss‘)">`. La fig. 3.4 mostra il risultato dell'esecuzione del payload con l'elemento *iframe* inserito nel DOM e l'alert che dimostra la piena esecuzione del codice JavaScript.

3.2.2 API per la ricerca

Il componente precedente della barra di ricerca si affida a un API che può essere intercettata utilizzando *ZAP*. La fig. 3.5 mostra che il browser fa richiesta a `/rest/products/search?q=` per ottenere la lista dei prodotti da visualizzare. Navigando sull'URL se ne può avere conferma. In questo caso si può sfruttare il parametro `q` per eseguire un payload di tipo *SQL Injection* contenente un comando di interrogazione. Infatti, si può ottenere l'intero schema del database (e leggerne i dati) eseguendo un attacco di tipo *UNION* descritto nella sezione di 4.2

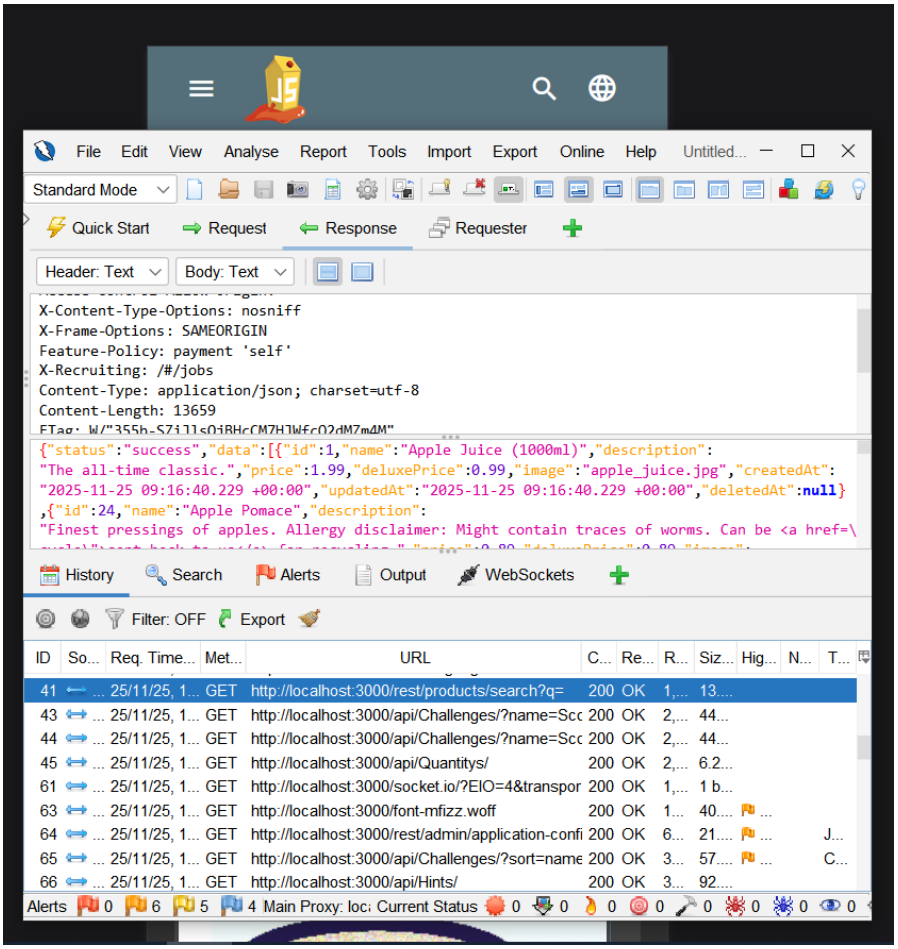

Figura 3.5: Richiesta API per la ricerca dei prodotti evidenziata in blu nella finestra inferiore. In alto la risposta del server contenente i header e il JSON dei prodotti (giallo e rosa)

da un punto di vista risolutivo. Il payload di attacco che consente di ottenere i dati sensibili è il seguente: `test'))UNION SELECT sql,2,3,4,5,6,7,8,9 FROM sqlite_master--` e il risultato è mostrato in fig. 3.6 dove si possono vedere i nomi delle tabelle e delle colonne del database SQLite dentro una risposta in formato JSON.

3.2.3 Modifica prodotti

Scorrendo nel codice *JavaScript* `main.js` del sito web è possibile trovare con grande facilità le API utilizzate per gestire i prodotti. Una di queste è `/api/Products`

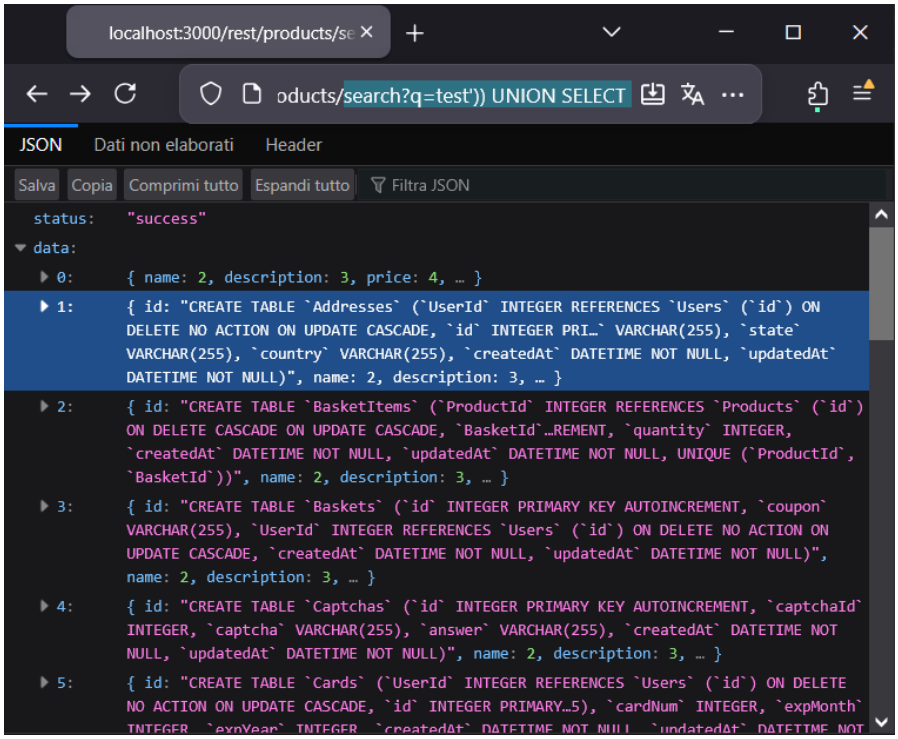

Figura 3.6: Risultato di invio di un payload UNION sull'API di ricerca dei prodotti. La risposta del server contiene i dati sensibili del database SQLite, come il nome delle tabelle e delle colonne.

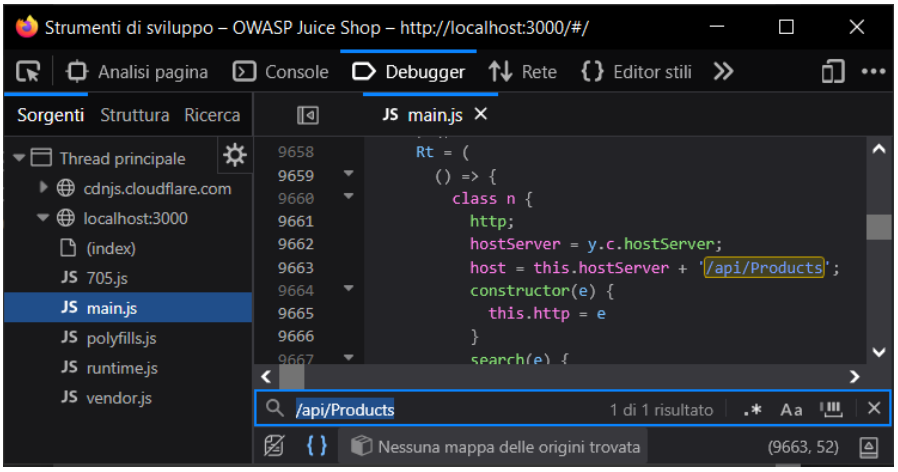

Figura 3.7: Codice *JavaScript* che mostra l'uso dell'endpoint `/api/Products` visualizzato nei strumenti di sviluppo del browser.

(fig. 3.7). Si presuppone che l'endpoint venga utilizzato dagli amministratori per creare o modificare i prodotti, ma presto si vedrà che è possibile farne uso senza autenticazione.

Visitando l'endpoint API in questione vengono mostrati in formato JSON tutti i prodotti. Aggiungendo l'id di un prodotto: `/api/Products/9` è possibile interagire con il singolo prodotto. In HTTP si hanno diversi metodi di richiesta tra cui `GET`, `POST`, `PUT` e `DELETE`. Utilizzando *Postman* si può inviare una richiesta di tipo `PUT` per modificare il prodotto con id 9 (fig. 3.8). Nella richiesta si specifica la nuova descrizione del prodotto e si invia la richiesta senza nessun header di autenticazione. La risposta del server è positiva (codice 200) e il prodotto viene modificato con successo.

3.2.4 Pagina reclami

Dopo aver eseguito il login con un account personale è possibile vedere il link alla pagina *Complaint* nella barra laterale (`/complain`). Visitandola si è presentati con un form (fig. 3.9) contenente i campi *Customer* (non modificabile), *Message* e *Invoice*, il quale consente di allegare un file opzionale. Il vettore di attacco è esattamente quest'ultimo, su cui verranno controllate le restrizioni di upload.

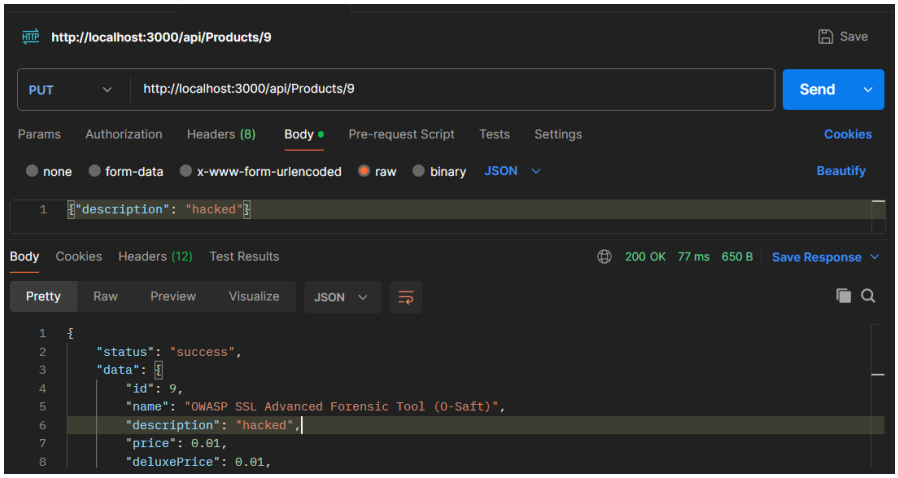

Figura 3.8: Esecuzione con *Postman* di una richiesta `PUT` sull'endpoint `/api/Products/9` per modificare il prodotto con id 9 senza autenticazione. La risposta del server (in basso) indica che la modifica è avvenuta con successo.

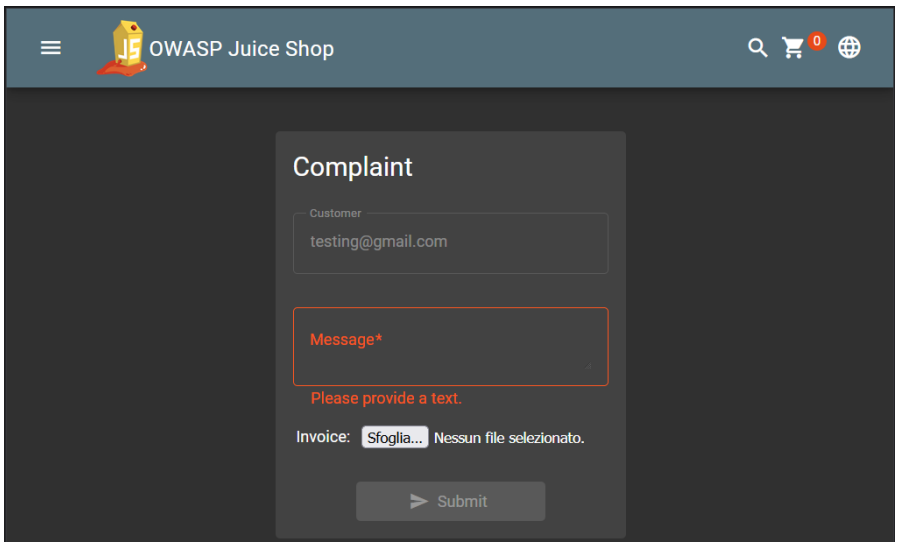

Figura 3.9: Form di invio reclami con campo di caricamento file

Figura 3.10: Violazione degli allegati consentiti dopo aver inserito un immagine `.jpg` nel form di invio reclami.

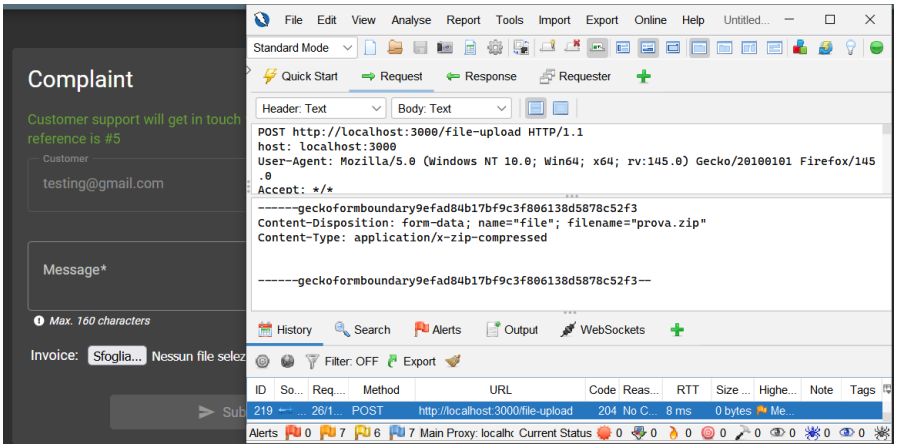

Figura 3.11: Intercettazione con *ZAP* di una richiesta pulita con allegato `prova.zip` visibile nel attributo `filename=""` del corpo della richiesta.

Tentando di allegare un file di tipo arbitrario come `.jpg` si ottiene a schermo l'errore di fig. 3.10 che comunica le estensioni consentite, ovvero solo `.zip` e `.pdf`. Per intercettare la richiesta (e scovare la vulnerabilità) si procede allora inserendo come allegato un file `.zip` e si analizza la richiesta dentro ZAP. La fig. 3.11 mostra nella finestra bianca di ZAP la richiesta POST all'endpoint `/file-upload` che carica effettivamente il file.

Aprendo il *Request Editor* di ZAP viene modificato il file allegato con uno di tipo non consentito, ad esempio `.txt` nel corpo della richiesta, come mostrato in fig. 3.12 e cliccando su *Send* la richiesta modificata viene inviata al server. La risposta del server è nuovamente un `HTTP 204 code` e il file viene caricato con successo nonostante la violazione delle estensioni consentite. Ciò dimostra che il controllo delle estensioni è effettuato solo lato client e non sul server, permettendo così a un attaccante di caricare file arbitrari.

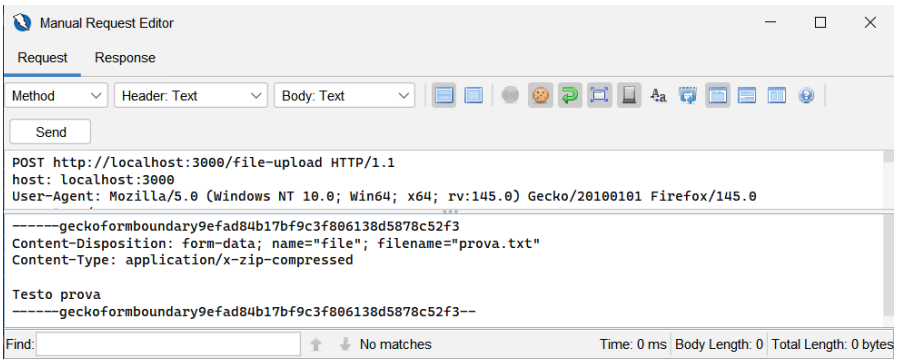

Figura 3.12: Richiesta HTTP modificata in *ZAP* con *Manual Request Editor* per allegare un file di tipo non consentito `.txt`.

Capitolo 4

Implementazione

In questo capitolo vengono implementate le contromisure per quattro tipi di vulnerabilità presenti su Juice Shop. Tutto il codice riportato in questa sezione è rilasciato sotto licenza MIT[14]. È possibile trovare il codice completo di tutte le modifiche effettuate sulla piattaforma Zenodo[17].

4.1 XSS (Cross-Site Scripting)

La componente front end della barra di ricerca è spesso una fonte di vulnerabilità per il computer dell'utente se non gestita in modo sicuro. Molte volte i valori di ricerca non vengono sanificati (ripuliti da codice speciale come tag HTML con codice JavaScript) e ciò può portare all'esecuzione di codice non previsto sulla macchina utente. Se il valore di ricerca viene immesso nell'URL, questo può essere usato per un attacco di hacking dove si inietta del codice malizioso all'interno di un link che la vittima apre, compromettendo i suoi dati sul sito. È proprio ciò che accadde in questa sezione, dove verrà risolta una vulnerabilità XSS nella barra di ricerca del sito.

4.1.1 Barra di ricerca

Il framework Angular assiste i sviluppatori diffidando gli input provenienti da utenti e quindi implementando una rigida sanificazione dei valori da inserire nel DOM[18]. Tuttavia, rende anche possibile la disattivazione[19] del meccanismo

di sanificazione automatica per scopi precisi (come inserimento di `<iframe>`) con `bypassSecurityTrust...`, come visibile nel listing 4.1 che concerne la funzione di ricerca di prodotti[1].

Listing 4.1: Filtro dei prodotti in base alla stringa di ricerca. La riga 6 mostra un uso scorretto di `bypassSecurityTrustHtml`

```
1  filterTable() {
2      let queryParam: string = this.route.snapshot.queryParams.q
3      if (queryParam) {
4          queryParam = queryParam.trim()
5          this.dataSource.filter = queryParam.toLowerCase()
6          this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)
7          this.gridDataSource.subscribe((result: any) => {
8              if (result.length === 0) {
9                  this.emptyState = true
10             } else {
11                 this.emptyState = false
12             }
13         })
14     } else {
15         this.dataSource.filter = ''
16         this.searchValue = undefined
17         this.emptyState = false
18     }
19 }
```

In questo caso però gli sviluppatori del sito hanno commesso un errore: la barra di ricerca non necessità di particolari funzionalità aggiuntive; essa deve solamente elaborare la stringa di ricerca e restituire i prodotti che soddisfano la stringa. Nel codice quindi viene eliminato `this.sanitizer.bypassSecurityTrustHtml(queryParam)` e sostituito con `queryParam`.

Ora avendo apportato la modifica si tenta il payload `<iframe src="javascript:alert(‘xss‘)">` nella barra di ricerca. Visualmente non si ha nessun riscontro e neppure nell'ispeziona elementi si vede alcuna traccia dell'elemento `<iframe>`. La fig. 4.1 mostra l'albero degli elementi dopo l'inserimento del payload, dove si nota che l'elemento non viene inserito nel DOM, poiché contiene un tag non consentito.

[1]Il codice di *filterTable()* si trova all'interno di `frontend/src/app/search-result/search-result.component.ts`

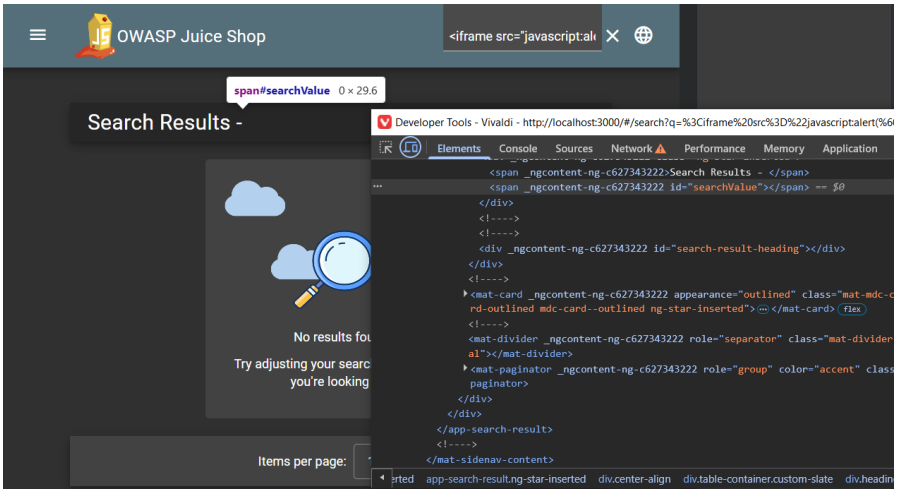

Figura 4.1: Payload iframe XSS nella barra di ricerca. L'albero degli elementi mostra che l'elemento non viene inserito

Figura 4.2: Errore in console con payload di tipo `<img src=javascript:alert('XSS')>`

Ci si può domandare cosa avviene utilizzando un payload diverso, come `<img src=javascript:alert('XSS')>`. La risposta è che Angular immette l'elemento nel DOM con la peculiarità che viene aggiunto l'attributo *unsafe* (fig. 4.3) dinanzi al codice JavaScript, rendendolo del tutto inagibile e stampando in console l'errore di fig. 4.2.

La vulnerabilità si ritiene così risolta poiché il codice JavaScript non viene eseguito e l'inserimento di elementi pericolosi come `<iframe>` o `<script>` non produce più nessun risultato. Ci si affida in questo caso ad Angular che esegue automaticamente la sanificazione. Per tecnologie diverse da Angular si sarebbe dovuta implementare una funzione di sanificazione personalizzata.

4.2 SQL Injection

Le interrogazioni SQL stanno alla base di ogni sito dinamico. Mentre nella sezione precedente si è vista una vulnerabilità che coinvolge il client, qui verrà mostrato

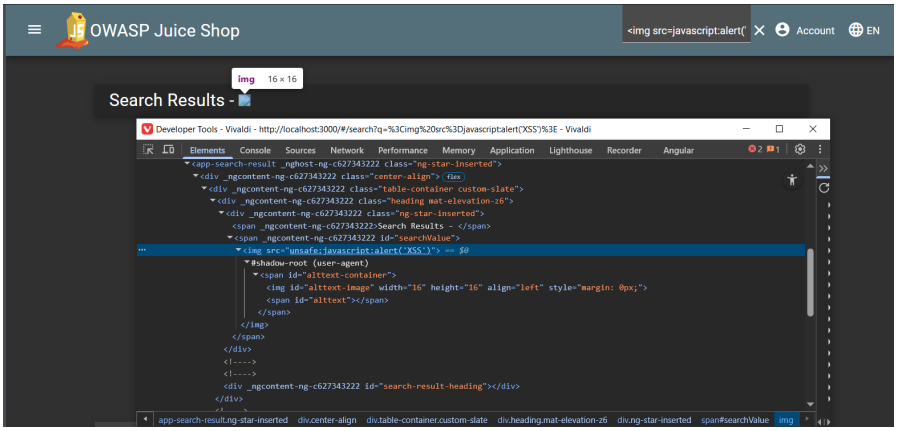

Figura 4.3: Payload `<img src=javascript:alert('XSS')>` nella barra di ricerca. L'elemento immagine viene inserito, ma il codice JavaScript viene etichettato non sicuro

come un utente malizioso potrebbe interagire con il sito web (server) per ottenere informazioni riservate del database.

Il punto di partenza per il seguente attacco è nuovamente la barra di ricerca che esegue una interrogazione vulnerabile sul server, accettando ogni tipo di parametro.

4.2.1 Fuga di dati tramite API

Listing 4.2: Ricerca dei prodotti con query SQL vulnerabile

```
export function searchProducts () {
  return (req: Request, res: Response, next: NextFunction) => {
    let criteria: any = req.query.q === 'undefined' ? '' : req.query.q ?? ''
    criteria = (criteria.length <= 200) ? criteria : criteria.substring(0, 200)
    models.sequelize.query('SELECT * FROM Products WHERE ((name LIKE '%${criteria
        }%' OR description LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER BY
        name')
      .then(([products]: any) => {
        const dataString = JSON.stringify(products)
        for (let i = 0; i < products.length; i++) {
          products[i].name = req.__(products[i].name)
          products[i].description = req.__(products[i].description)
        }
        res.json(utils.queryResultToJson(products))
      }).catch((error: ErrorWithParent) => {
        next(error.parent)
      })
  }
```

```
17 }
```

Nella funzione `searchProducts`[2] (listing 4.2) che espone l'API di ricerca dei prodotti `/rest/products/search` è presente codice SQL vulnerabile, legato a un uso non corretto della libreria Sequelize[20]. Sequelize è un ORM (Object-Relational Mapping) per Node.js che consente di interagire con database come MySQL e, in questo caso, SQLite.

La funzione `query()`[21] consente di passare un parametro stringa contenente la query SQL e restituisce una Promise[22] contenente il risultato. La vulnerabilità back-end del sito si trova precisamente nell'utilizzo di questo metodo: gli sviluppatori hanno costruito la stringa SQL in modo dinamico facendo un passaggio di parametro.

Listing 4.3: Interrogazione SQL di ricerca dei prodotti con criterio vulnerabile

```
1 models.sequelize.query(`SELECT * FROM Products WHERE ((name LIKE '%${criteria}%'
    OR description LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER BY name`)
```

Nel codice sopra riportato viene mostrata la query di selezione contenente il parametro `criteria`, ovvero la chiave di ricerca del prodotto. Questo viene fornito usando la formattazione `` `${chiave}` `` che corrisponde al Template Literal[23] di TypeScript. In questo modo la query SQL può essere costruita dinamicamente e in un modo facile, ma rende possibile l'Injection poiché il parametro non viene sanificato. L'utente malizioso, infatti, potrà eseguire un attacco di tipo `UNION` per ottenere l'intero schema del database!

Per procedere alla risoluzione di questo problema è necessario leggere la documentazione sulle *Raw Queries*[24] di Sequelize e utilizzare correttamente le funzionalità *Replacements* oppure *Bind Parameter* per eseguire la sanificazione della stringa proveniente dal client e interagire nel modo più sicuro con il database. Usando i *replacements* si crea un formato di rimpiazzamento riscrivendo il blocco `'%${criteria}%'` con `:chiave`, formando la query finale in questo modo:

Listing 4.4: Query SQL sicura con rimpiazzamento del parametro

```
1 SELECT * FROM Products WHERE ((name LIKE :chiave OR description LIKE :chiave) AND
    deletedAt IS NULL) ORDER BY name
```

[2]La funzione *searchProducts()* si può trovare dentro `routes/search.ts`

Successivamente, viene richiamato il parametro nominato `:chiave` dentro l'oggetto di opzioni passato come secondo parametro a `query()`. Il risultato di questa modifica è visibile nel listing 4.5:

Listing 4.5: Uso di *replacements* per sanificare il parametro di ricerca nel metodo `query()` di Sequelize

```
models.sequelize.query('SELECT * FROM Products WHERE ((name LIKE :chiave OR
    description LIKE :chiave) AND deletedAt IS NULL) ORDER BY name', {
    replacements: { chiave: '%${criteria}%' }
})
```

Anche la wildcard `%` (che nel `LIKE` di SQL corrisponde alla ricerca di tutti i pattern con la parola chiave) si trova ora attorno alla variabile `criteria`. Proseguendo, viene eseguito il testing di Injection per assicurarsi che la vulnerabilità sia stata corretta. La fig. 4.4 mostra la risposta JSON del server dopo che si è inviata la richiesta di ricerca con il payload di Injection. Il payload `test'))UNION SELECT sql,2,3,4,5,6,7,8,9 FROM sqlite_master--` che in precedenza poteva restituire l'intero schema SQLite ora restituisce un array vuoto.

La vulnerabilità è stata in questo modo risolta, poiché la funzione back end non accetta più comandi SQL, che ora vengono trattati come stringhe di ricerca e sanificati, restituendo così solo i prodotti che soddisfano la chiave di ricerca.

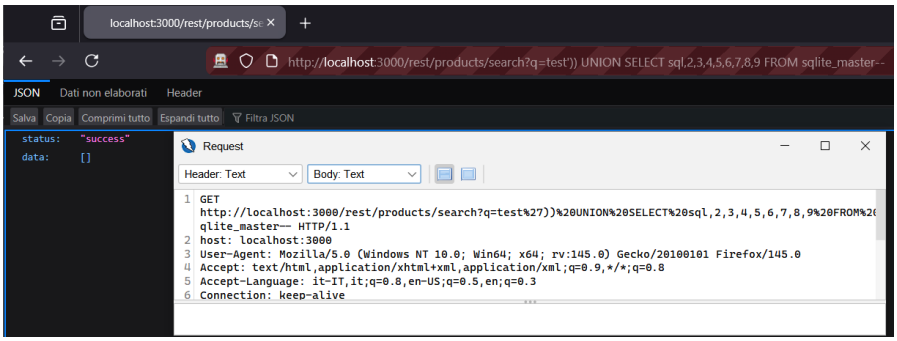

Figura 4.4: Tentativo di Injection con UNION a `rest/search?q=` da browser e visualizzazione richiesta in ZAP

4.3 Broken Access Control

Durante lo sviluppo di un web server è importante definire i permessi di accesso alle risorse ai soli utenti abilitati a utilizzarle. Per un sistema complesso è comune dimenticarsi di disabilitare gli accessi ad alcune interfacce, esponendole ad attacchi dove un hacker può modificare dati sensibili o eseguire operazioni non autorizzate. In questa sezione viene mostrato un esempio di questo tipo di vulnerabilità e le conseguenze.

4.3.1 Manomissione dei prodotti

Il file `server.ts` è il punto di entrata principale invocato all'avvio dell'applicazione Node.js. Contiene la procedura `start()` che inizializza i servizi di WebSocket riguardanti le notifiche, le metriche di Prometheus e assegna la porta di ascolto del server. Inoltre, fa uso di ExpressJS per impostare gli endpoint API personalizzati. Riunisce tutte le componenti del back end per offrire indirizzi URL all'utente finale che potrà interagire e usufruire dei servizi esposti.

Listing 4.6: Assegnazione degli endpoint API e relative funzioni *middleware* di sicurezza

```
1 /* Baskets: Unauthorized users are not allowed to access baskets */
2 app.use('/rest/basket', security.isAuthorized(), security.appendUserId())
3 /* BasketItems: API only accessible for authenticated users */
4 app.use('/api/BasketItems', security.isAuthorized())
5 app.use('/api/BasketItems/:id', security.isAuthorized())
6 // [...]
7 /* Products: Only GET is allowed in order to view products */
8 app.post('/api/Products', security.isAuthorized())
9 app.delete('/api/Products/:id', security.denyAll())
```

Il codice sopraesposto mostra come l'oggetto `app` (Express) viene invocato per creare gli endpoint API e abilitare le richieste con metodi HTTP come `GET`, `POST`, `PUT`, `DELETE` a indirizzi personalizzati. Il secondo parametro delle funzioni in `app` riceve generalmente la funzione di *callback*. Questa viene utilizzata per elaborare le richieste e, nel caso più specifico, confermare l'identità dell'utente per determinarne i suoi permessi. Essa viene denominata *middleware function*, poiché agisce da tramite tra la richiesta e la risposta, gestendo l'autenticazione, il logging o gli errori.

In questo semplice caso, gli sviluppatori si sono dimenticati di assegnare i controlli di sicurezza sulla richiesta HTTP PUT di `/api/Products/:id`, consentendo a **tutti** gli utenti di modificare i prodotti a proprio piacimento. In questo modo, oltre a interrompere un servizio e procurare danno finanziario, l'utente malizioso potrebbe iniettare un payload XSS, inserendo script arbitrari nelle descrizioni dei prodotti.

Inoltre, se il *middleware* conteneva un controllo dei limiti (rate-limiting, ovvero la limitazione del numero di richieste in un intervallo di tempo), con una sola riga mancante si espone il server a un attacco DoS (Denial-of-Service).

Per risolvere questa vulnerabilità occorre aggiungere il seguente blocco dentro `server.ts`:

Listing 4.7: Controllo di accesso per l'endpoint `/api/Products`

```
app.route('/api/Products')
  .get()
  .post(security.denyAll())
  .put(security.denyAll())
  .delete(security.denyAll())
app.use('/api/Products/:id', security.isAccounting())
```

I servizi di `PUT`, `DELETE` e `POST` vengono così rifiutati per l'API *Products*; `GET` resta abilitato normalmente per tutti. Inoltre, per utilizzare l'API `/api/Products/:id` è necessario essere amministratori. La fig. 4.5 mostra l'errore di autorizzazione restituito dal server dopo aver tentato di eseguire una richiesta `PUT` su `/api/Products/9` senza autenticazione.

L'applicazione così modificata richiede i permessi per la modifica dei prodotti e la vulnerabilità di Broken Access Control si ritiene risolta poiché ora solo gli utenti autorizzati possono eseguire operazioni di modifica sui prodotti, mentre tutti gli altri ricevono un errore di autorizzazione.

4.4 Security Misconfiguration

Un'impostazione errata dei servizi web può portare a gravi falle di sicurezza, come l'esecuzione di codice remoto. In questa sezione viene mostrato un esempio di vulnerabilità legata al caricamento di file non sicuri che deriva da una cattiva configurazione del server.

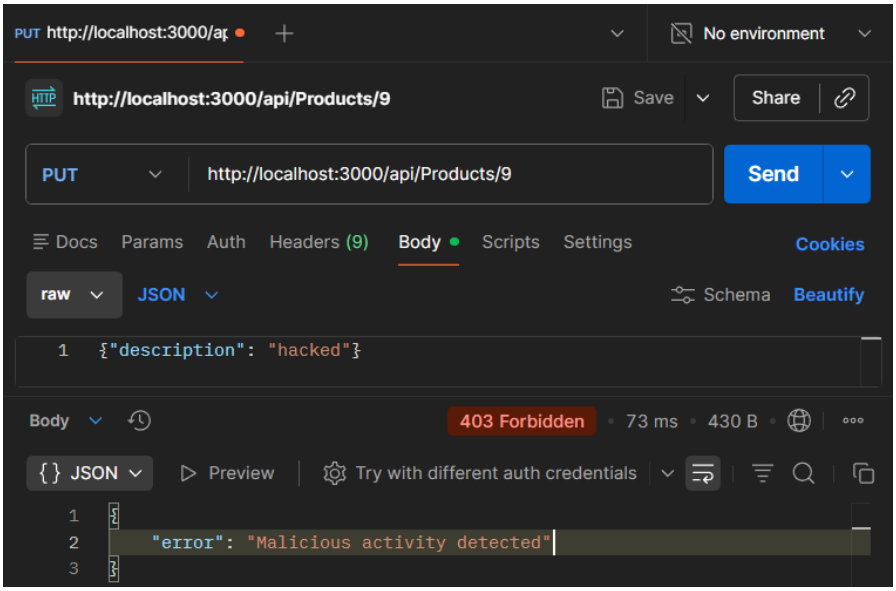

Figura 4.5: Errore di autorizzazione dopo aver tentato una richiesta `PUT` su `/api/Products/9`

4.4.1 Interfaccia deprecata

La pagina di invio reclami `/complaint` (fig. 4.6) consente agli utenti di allegare file di tipo `.pdf`, `.zip` (da front end). Tuttavia, intercettando la richiesta è possibile modificare il file allegato con uno di tipo diverso, come `.txt.zip`, aggirando il controllo sul front end. Questo perché il server non esegue un controllo sul tipo di estensione del file caricato, consentendo così l'upload di file non sicuri. All'interno di `server.ts` si può notare la presenza dell'interfaccia di caricamento dei file su indirizzo `/file-upload`:

```
app.post('/file-upload', uploadToMemory.single('file'), ensureFileIsPassed,
    metrics.observeFileUploadMetricsMiddleware(), handleZipFileUpload,
    handleXmlUpload, handleYamlUpload)
```

Su questa API vengono eseguite diverse funzioni *middleware*. Innanzitutto viene chiamata `uploadToMemory.single('file')` la quale esegue il caricamento del file nel buffer di memoria volatile, tramite la libreria `multer` di Node.JS, e aggiunge un limite di dimensione.

```
const uploadToMemory = multer({ storage: multer.memoryStorage(), limits: {
    fileSize: 200000 } })
```

La seconda funzione `ensureFileIsPassed` controlla la presenza del file, assicurando che non sia nullo.

La terza `metrics.observeFileUploadMetricsMiddleware()` riguarda le metriche di Prometheus ed esegue il logging del tipo di file oppure il conteggio di upload erronei. Nel listing 4.8 è possibile vedere il codice di questa funzione, che incrementa i contatori di Prometheus in base al tipo di file caricato e alla risposta del server.

Listing 4.8: Funzione di conteggio dei file caricati e degli errori di upload

```
export function observeFileUploadMetricsMiddleware () {
  return ({ file }: Request, res: Response, next: NextFunction) => {
    onFinished(res, () => {
      if (file != null) {
        res.statusCode < 400 ? fileUploadsCountMetric.labels(file.mimetype).inc()
            : fileUploadErrorsMetric.labels(file.mimetype).inc()
      }
    })
    next()
  }
}
```

Le successive funzioni riguardano la elaborazione di file specifici con estensione `.zip`, `.xml` e `yaml`. In tutto questo, si può notare la **mancanza** di un'importante funzione protettiva: il controllo sul tipo di estensioni (Nel codice è già presente la funzione `checkFileType`, ma questa serve semplicemente per segnare il completamento della challenge. In un caso reale si può presupporre che i developer si siano dimenticati di eseguire il controllo oppure l'hanno realizzato male, poiché non comporta un semplice controllo della stringa di estensione). Nel front end di Angular è presente un controllo sul tipo di estensione[3], ma occorre prestare attenzione poiché è codice lato client che può essere modificato o immediatamente aggirato intercettando la richiesta API con OWASP ZAP.

Si sposta l'attenzione quindi su `routes/fileUpload.ts` dove si modificherà la funzione `checkFileType`. Si potrebbe molto facilmente pensare di usare il parametro `.mimetype` di un oggetto `Multer.File`, ma questo **non** è sicuro, poiché fa riferimento al header che viene inviato nella richiesta, che può essere intercettato dall'utente.

[3]Si veda `frontend/src/app/complaint/complaint.component.ts`

Per eseguire un check **sicuro** occorre leggere il file per trovare il *magic number*, ovvero un codice specifico che identifica il contenuto del file. Su Node.JS questo viene eseguito con la funzione `fromBuffer` o `fileTypeFromBuffer` importata dal package `file-type`. Successivamente, si tratta semplicemente di controllare che il tipo ritornato sia incluso in quelli consentiti e in caso contrario ritornare un messaggio di errore. Il listing 4.9 mostra la funzione di controllo del tipo di file caricato, che ora esegue un controllo sicuro basato sul contenuto del file e non sull'estensione o sul mimetype dichiarato.

Listing 4.9: Funzione di controllo del tipo di file caricato

```
import { fromBuffer } from 'file-type'
async function checkFileType ({ file }: Request, res: Response, next: NextFunction
    ) {
  try {
    const type = (file?.buffer != null) ? await fromBuffer(file.buffer) :
        undefined
    const allowedTypes = ['application/pdf', 'application/xml', 'application/zip',
        'application/x-yaml', 'text/yaml']
    if (!type || !allowedTypes.includes(type.mime)) {
      res.status(415)
      next(new Error('Invalid file type'))
    }
  } catch (err) {
    res.status(503)
    next(new Error('Internal Server Error'))
  }
  next()
}
```

Infine, si aggiorna `server.ts` per includere la funzione appena creata:

Listing 4.10: Endpoint di caricamento file con controllo del tipo di estensione `checkFileType`

```
app.post('/file-upload', uploadToMemory.single('file'), ensureFileIsPassed,
    checkFileType, metrics.observeFileUploadMetricsMiddleware(),
    handleZipFileUpload, handleXmlUpload, handleYamlUpload)
```

Ora, quando si tenta di caricare un file malevolo come `test.txt.zip` già subito si ottiene il riscontro HTTP 415, `Unsupported File Type` e la pagina HTML comunica l'errore correttamente. È stata tentata anche la modifica dell'attributo `filename=test.txt.zip` in `filename=test.txt` e la risposta è la *stessa*: il patch della vulnerabilità è stato eseguito correttamente.

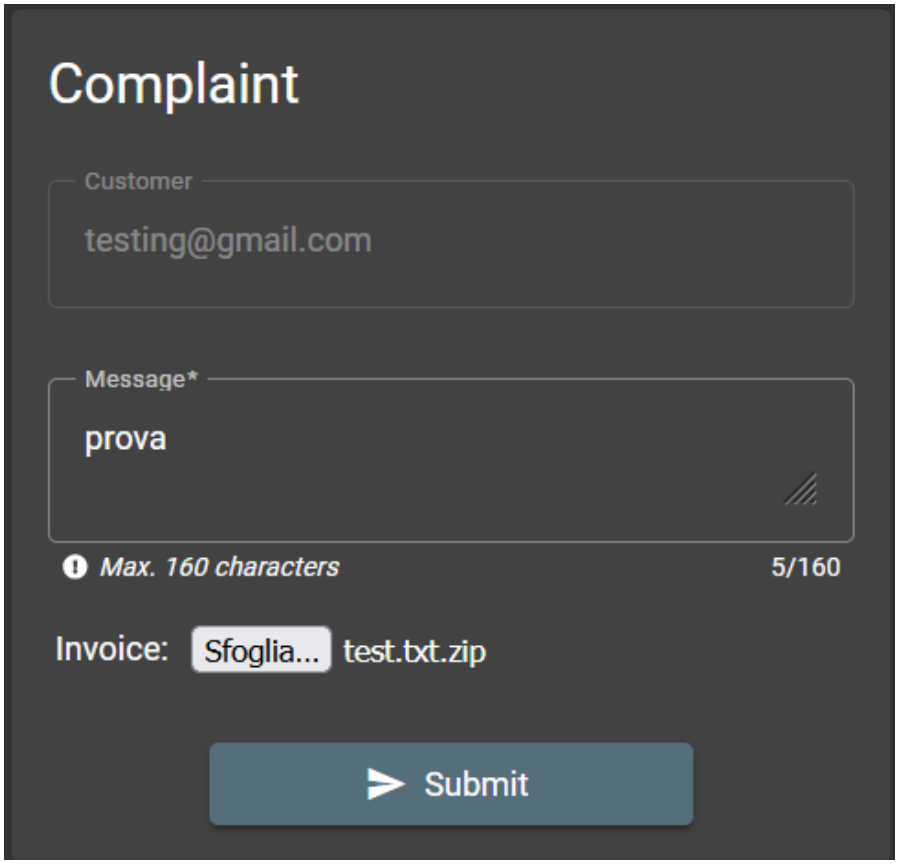

Figura 4.6: Complaint form `/complaint` con allegato un file `text.txt.zip`

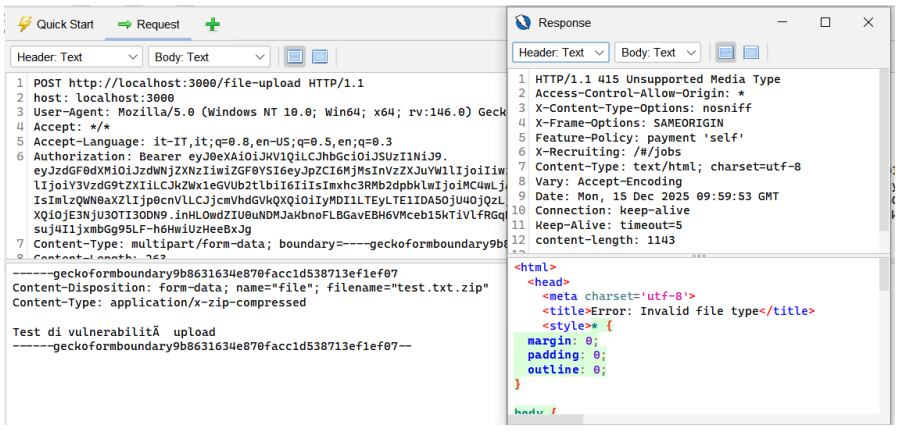

Figura 4.7: Risposta di invio del file `test.txt.zip` a destra, con messaggio di errore `Invalid file type` e risposta HTTP 415

L'ultimo procedimento da fare per completare la contromisura è quello di disabilitare effettivamente l'interfaccia deprecata. Nel listing 4.9 si può osservare la presenza di `handleXmlUpload` e `handleYamlUpload` che però internamente risultano deprecate per motivi di sicurezza. Questo perché entrambi sono esposti ad attacchi XML External Entity (XXE) che consentono di eseguire `Server-Side Request Forgery` e letture di file, tramite payload specifici dentro file XML.

Per disabilitarle, si possono effettivamente eliminare le funzioni di gestione di queste due estensioni da `app.post()` formando così la istruzione finale:

Listing 4.11: Endpoint di caricamento file dopo la rimozione delle interfacce deprecate

```
app.post('/file-upload', uploadToMemory.single('file'), ensureFileIsPassed,
    metrics.observeFileUploadMetricsMiddleware(), checkUploadSize, checkFileType,
    handleZipFileUpload)
```

Il server ora non effettua più il parsing di codice `XML` o `YAML`. In più, dentro la funzione `checkFileType` precedentemente scritta si tolgono le due estensioni non più consentite:

```
const allowedTypes = ['application/pdf', 'application/zip']
```

Per confermare la risoluzione della falla di sicurezza, si tenta di caricare un file `test.xml.zip` e la risposta del server è un codice `HTTP 415`, come mostrato in fig. 4.7.

Finalmente, il lavoro di patching è terminato a buon fine e l'API `/file-upload` non è più vulnerabile a *XXE* e *Unrestricted File Upload*. L'interfaccia deprecata è stata disabilitata, il controllo del tipo di estensione è stato implementato in modo sicuro e la vulnerabilità di Security Misconfiguration si ritiene risolta.

Capitolo 5

Conclusioni

In questa tesi si è mostrato il lavoro svolto per la risoluzione di vulnerabilità di sicurezza di un'applicazione vulnerabile. La base di partenza è stata la teoria dietro alle API, proseguendo con la lista OWASP delle vulnerabilità più comuni che è stata un fondamento solido per l'analisi pratica dell'applicazione. L'uso di strumenti come ZAP ha permesso di visualizzare le componenti software in interazione e di comprendere il meccanismo delle API studiato precedentemente, contribuendo a una crescita graduale nelle abilità di caccia proattiva di minacce informatiche. In seguito alla scoperta delle stesse, la risoluzione delle vulnerabilità si è rivelata un processo sistematico che ha coinvolto una prima analisi del codice sorgente e successivamente la lettura di documentazione in rete per trovare l'approccio implementativo più sicuro e adeguato al sistema. Con un ultima fase di testing si è potuto confermare la risoluzione dei problemi di sicurezza e concludere il processo, verificando che gli attacchi precedentemente attuabili ora non sono più eseguibili.

Ovviamente, la protezione da attacchi informatici non è stata terminata (e non lo sarà mai!): il continuo sviluppo di un applicazione o la sola messa in rete di un sito web espone a nuove vulnerabilità che devono essere risolte regolarmente. Un proseguimento naturale del lavoro svolto in questa tesi potrebbe essere sicuramente l'automazione del processo di scansione delle vulnerabilità e l'integrazione di test di sicurezza automatici all'interno della pipeline di sviluppo software.

Nonostante lo stato di sicurezza futuro dell'applicazione, in questa tesi sono state acquisite competenze fondamentali per l'analisi e la risoluzione di vulnerabi-

lità comuni, che potranno essere applicate in contesti reali di sviluppo software e sicurezza informatica.

Bibliografia

[1] E. Zveare, "How i hacked over 1,000 car dealerships across the us." https://eaton-works.com/2025/10/13/def-con-33/, 2025. Accessed: 2026-01-19.

[2] R. Shirey, "Internet security glossary, version 2," RFC 4949, RFC Editor, August 2007. http://www.rfc-editor.org/rfc/rfc4949.txt, Copyright (C) The IETF Trust (2007).

[3] Seobility, "Creative commons license by-sa 4.0." https://www.seobility.net/en/wiki/creative-commons-license-by-sa-4-0.

[4] Seobility, "Rest api." https://www.seobility.net/en/wiki/REST_API, 2026.

[5] O. T. . Team, "Owasp top ten web application security risks," tech. rep., OWASP, nov 2025.

[6] C. Commons, "Creative commons attribution 3.0 unported license." https://creativecommons.org/licenses/by/3.0/deed.en.

[7] OWASP, "The ten most critical web application security risks - introduction." https://owasp.org/Top10/2025/0x00_2025-Introduction/, nov 2025.

[8] M. Corporation, "Authorization header." https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Authorization, 2026.

[9] C. Commons, "Attribution-sharealike 4.0 international." https://creativecommons.org/licenses/by-sa/4.0/deed.en, 2026.

[10] M. Bakni, "Cross-site scripting attack sequence diagram." https://commons.wikimedia.org/wiki/File:Cross-site_scripting_attack_sequence_diagram_-_en.png, 2017.

[11] OWASP, "Security misconfiguration." https://owasp.org/Top10/2025/A02_2025-Security_Misconfiguration/, 2025. Licenza Creative Commons Attribution 3.0 Unported https://creativecommons.org/licenses/by/3.0/deed.en.

[12] D. Contributors, "Damn vulnerable web application." https://github.com/digininja/DVWA, 2025. Accessed: 2026-01-16.

[13] B. Kimminich and O. J. S. Contributors, "Owasp juice shop." https://owasp.org/www-project-juice-shop/, 2025. Accessed: 2026-01-16.

[14] B. Kimminich, "Juice shop mit license." https://github.com/juice-shop/juice-shop/blob/master/LICENSE, 2026.

[15] C. Commons, "Attribution-noncommercial-noderivatives 4.0 international." https://creativecommons.org/licenses/by-nc-nd/4.0/.

[16] B. Kimminich, "Architecture overview of owasp juice shop." https://pwning.owasp-juice.shop/companion-guide/latest/introduction/architecture.html.

[17] B. Kimminich, O. J. S. Contributors, and L. Grimaldi, "leonardogrimaldi/juice-shop-patching: v1.0.1," Feb. 2026.

[18] G. Angular, "Preventing cross-site scripting (xss)." https://angular.dev/best-practices/security#preventing-cross-site-scripting-xss. Accessed: 2026-01-15.

[19] G. Angular, "Trusting safe values." https://angular.dev/best-practices/security#trusting-safe-values. Accessed: 2026-01-15.

[20] S. Contributors, "Sequelize, a modern typescript and node.js orm." https://sequelize.org. Accessed: 2026-01-15.

[21] S. Contributors, "Instance method query api v6." https://sequelize.org/api/v6/class/src/sequelize.js~sequelize#instance-method-query. Accessed: 2026-01-15.

[22] M. O. Contributors, "Promise object." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed: 2026-01-15.

[23] Microsoft, "Template literal types." https://www.typescriptlang.org/docs/handbook/2/template-literal-types.html. Accessed: 2026-01-15.

[24] S. Contributors, "Raw queries." https://sequelize.org/docs/v6/core-concepts/raw-queries/. Accessed: 2026-01-15.