



universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Curso 8316 – Licenciatura em Engenharia de Computadores e Informática
Disciplina 42573 – Segurança Informática e nas Organizações
Ano letivo 2021/2022

Projeto 1

Autores:

75943 Daniel Fernandes Capitão
89078 Luís Filipe Correia do Couto
89082 António Jacinto Coelho Ferreira
93245 Lucas Pinho de Matos

Equipa: 6

Regente: João Paulo Silva Barraca

Resumo: Este projeto tem como objetivo a construção de duas aplicações web funcionais, uma com determinadas vulnerabilidades, e outra sem elas. Neste relatório vão ser apresentadas e explicadas as CWE escolhidas, uma demonstração da exploração destas na versão não segura da aplicação e finalmente o que foi adicionado/alterado/removido ao código para remover essas vulnerabilidades.

Aplicação Web Realizada - Breve Introdução

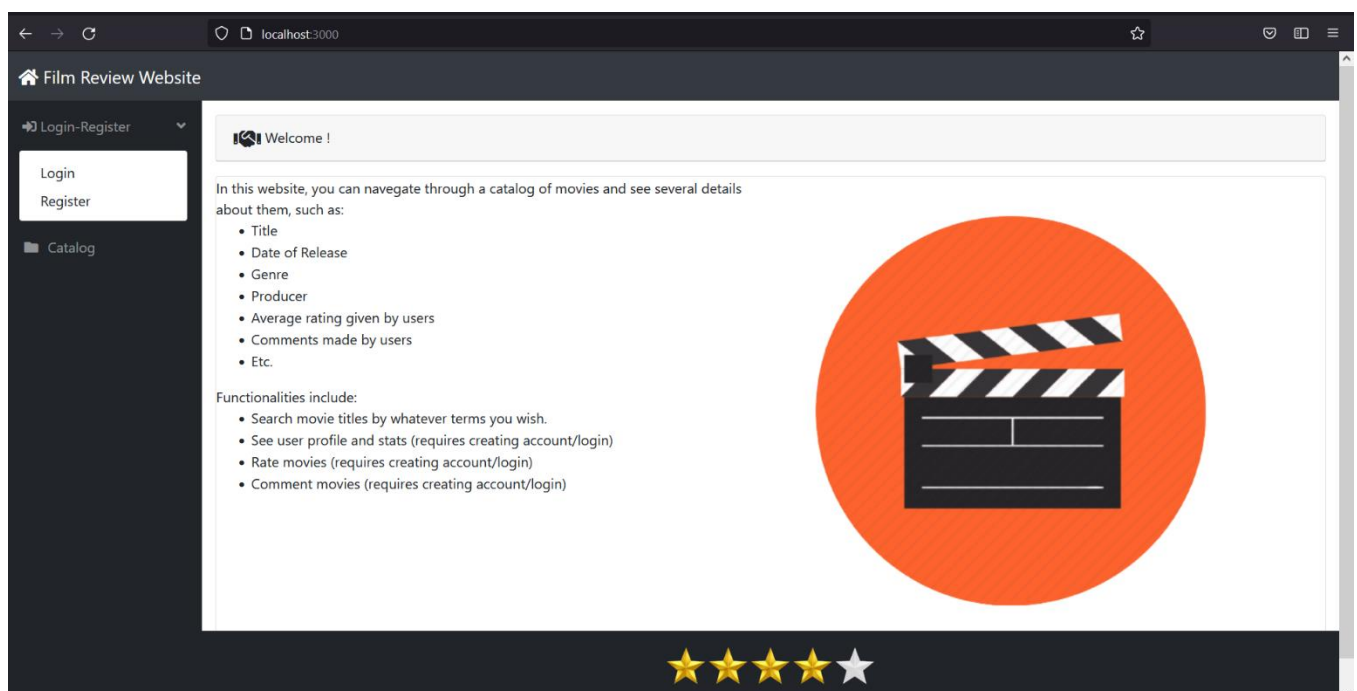
Para este projeto foi feita uma aplicação de análise de filmes.

Um utilizador sem conta tem acesso às seguintes funcionalidades:

- Visualizar catálogo de filmes
- Visualizar informação sobre qualquer filme existente no catálogo
- Visualizar comentários de outros utilizadores em qualquer filme existente no catálogo
- Visualizar rating média e número total de ratings dadas pelos utilizadores em qualquer filme existente no catálogo
- Fazer registo de uma nova conta e subsequente login

Um utilizador com conta e sessão iniciada tem acesso a funcionalidades extra:

- Comentar qualquer filme do catálogo
- Dar rating a qualquer filme do catálogo



Vulnerabilidades Escolhidas - Breve Explicação

CWE-20: Improper Input Validation

Esta vulnerabilidade está associada ao input de dados fornecidos à aplicação. Caso estes dados não sejam verificados/validados pelo servidor antes do seu processamento, podem ocorrer erros ou comportamentos originalmente não previstos e indesejados pela aplicação.

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Esta vulnerabilidade está associada à não neutralização/neutralização incorreta de dados introduzidos por utilizadores.

Isto acontece pois a aplicação não prevê que o utilizador insira comandos executáveis pelo browser (JavaScript, HTML, etc.), sendo estes comandos posteriormente gerados pelo servidor como parte de uma página web e servidos a outros que acessem a essa página.

Em particular, este projeto demonstra um Stored XSS – quando a aplicação guarda estes comandos executáveis na base de dados que, aquando da leitura desta pela aplicação, são apresentados aos utilizadores como parte do conteúdo dinâmico.

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Esta vulnerabilidade está associada à não neutralização/neutralização incorreta de elementos especiais em dados introduzidos por utilizadores, que vão depois ser utilizados como parte de um comando SQL.

Estes dados podem ser usados para alterar a lógica do comando SQL, com objetivo de ultrapassar verificação e autenticação, obter acesso a informação originalmente não disponível ou até mesmo alterar/eliminar a base de dados

CWE-328: Use of a Weak Hash

Esta vulnerabilidade é encontrada quando é usada uma função hash criptográfica considerada fraca. Funções hash são geralmente consideradas fracas quando um atacante tem hipótese de sucesso moderada em determinar:

- O input original, tendo apenas o digest (ataque da pré-imagem)
- Outro input que produz o mesmo digest, tendo apenas o input original (segundo ataque da pré-imagem)
- Um conjunto de dois ou mais inputs que se traduzem no mesmo digest (ataque de aniversário), dado que o atacante pode escolher dois inputs arbitrários para ser hashed e o consegue fazer um número razoável de vezes

CWE-759: Use of a One-Way Hash without a Salt

Esta vulnerabilidade é encontrada quando, ao aplicar uma função hash criptográfica não reversível a dados sensíveis como passwords, não é usado um valor de salt.

Isto tem como consequência uma maior facilidade/menor esforço computacional por parte dos atacantes para realizar ataques de dicionário, pois passwords iguais era ter sempre o mesmo valor hashed.

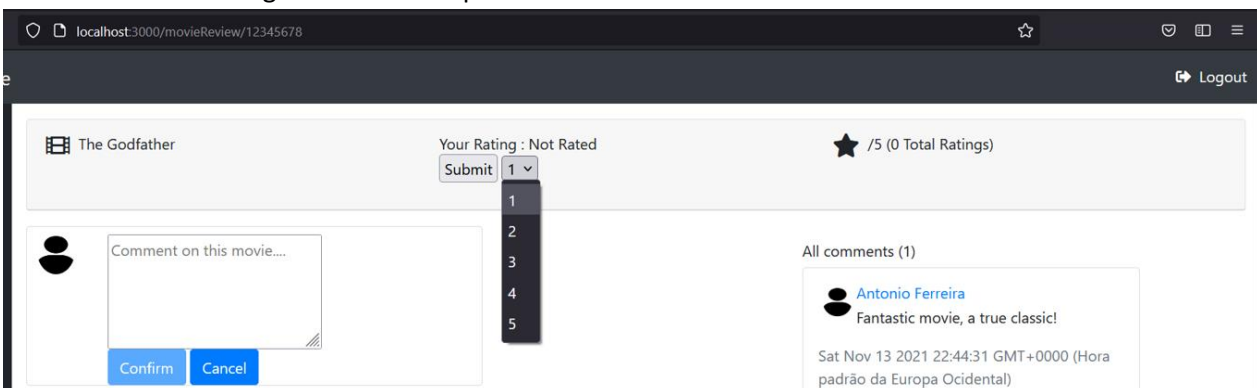
Em especial, deixa a base de dados vulnerável a ataques tabela arco-íris, sendo que com uso adequados de salt, este ataque seria praticamente infazível devido ao esforço computacional necessário.

Implementação --> CWE-20: Improper Input Validation

Versão Não Segura – Linha 531 do ficheiro appNotSafe.js

Esta vulnerabilidade está presente quando um utilizador com sessão iniciada dá rating a um filme, sendo que este pode dar rating pela primeira vez e depois mudar se assim o desejar, sendo o valor atualizado na base de dados.

Estes valores de rating são escolhidos pelo utilizador:



À primeira vista, parece que o utilizador apenas pode escolher um rating de 1-5, visto serem os únicos valores apresentados no dropdown. Sendo assim aparenta ser impossível qualquer outro valor ser enviado para processamento, pelo não foi implementada no server-side uma verificação do valor de rating antes deste ser escrito na base de dados.

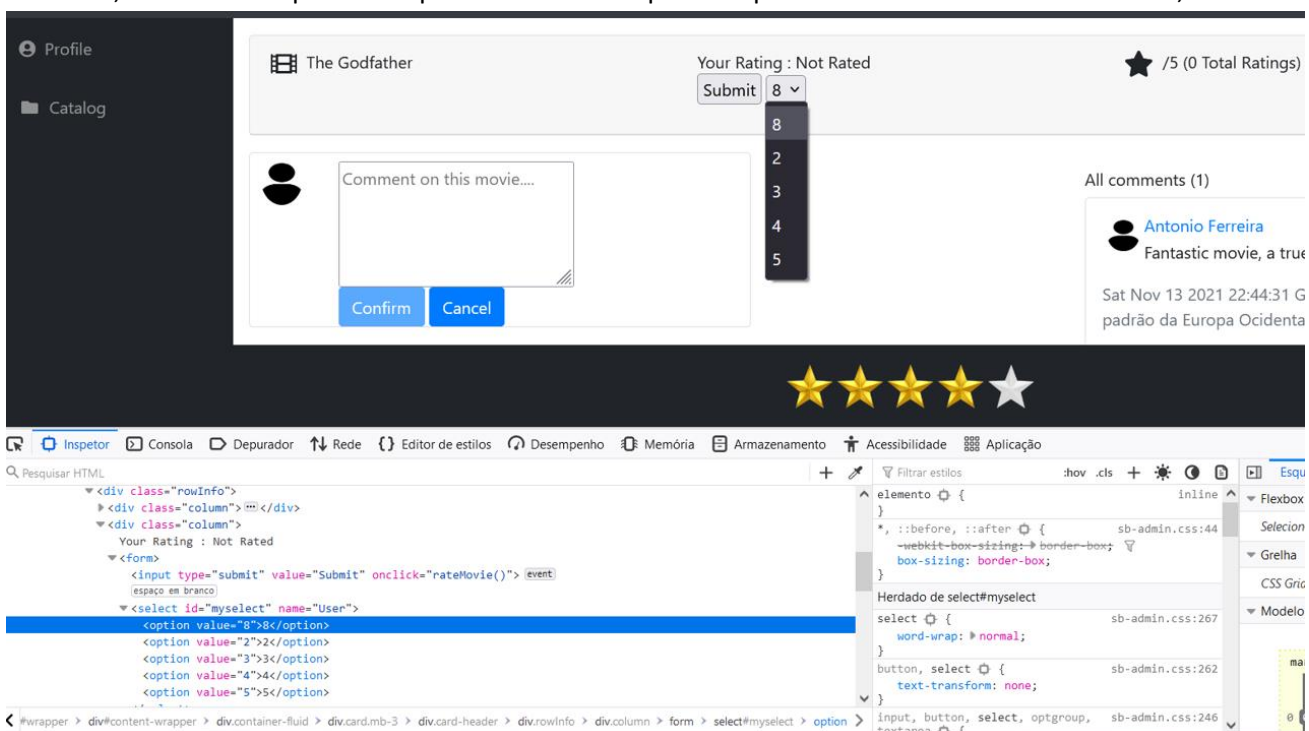
Esta funcionalidade está implementada num método post, que verifica primeiro se já existe um rating deste user no filme em questão. Se não, é inserido o registo na base de dados, caso contrário o registo é atualizado.

```
app.post('/giveRating', (req, res) => {
  var mv = req.body.mvToPost;
  var rt = req.body.rate;
  var usr = req.session.userId;
```

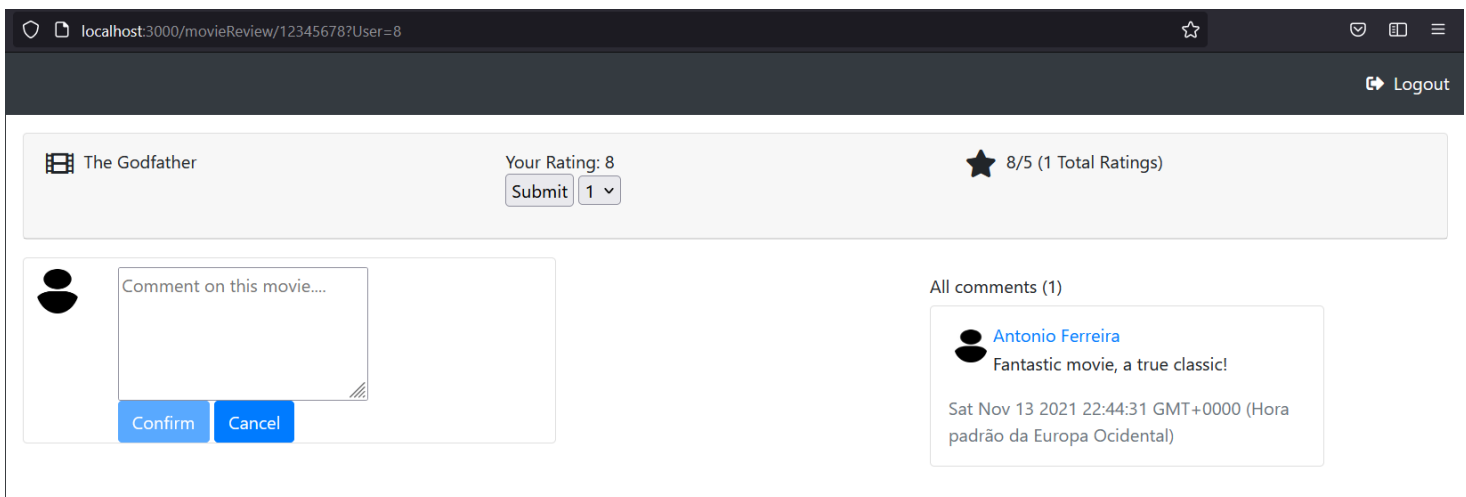
A variável de interesse aqui é **rt**, que tem como valor o input dado pelo utilizador e vai ser o rating dado ao filme.

Como se pode visualizar, não existe qualquer verificação de dados.

Contudo, um utilizador pode manipular os valores disponíveis pelo client-side e enviar ao servidor, tal como:



Após submeter o valor manipulado, como não existe verificação de inputs no server-side, qualquer valor inteiro é aceite. Portanto a operação vai correr normalmente e este valor vai ser guardado na base de dados.



Result Grid			
	user_id	numSerie	rating
▶	1	12345678	8
★	NULL	NULL	NULL

Base de dados após o utilizador dar submit ao valor manipulado.

No contexto desta aplicação, esta vulnerabilidade apresenta um problema **grave**, pois abre oportunidades para manipulação de ratings, tirando credibilidade/utilidade à aplicação, dado que os valores apresentados não correspondem á realidade.

Versão Segura – Linha 568 do ficheiro appSafe.js

De maneira a resolver esta vulnerabilidade, foi implementada uma verificação do input no server-side, antes deste ser processado:

```
app.post('/giveRating', (req, res) => {

  var mv = req.body.mvToPost;
  var rt = req.body.rate;
  var usr = req.session.userId;

  if (rt < 1 || rt > 5 || rt % 1 != 0) {
    res.json({ session: req.session });
  }
  else {
```

A variável *rt* que resulta do input do utilizador e é o rating dado ao filme, vai ser verificada antes de ser processada:

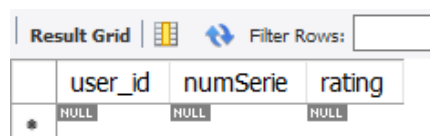
- É primeiro verificado se se encontra no intervalo esperado (1-5).
- Depois é verificado se é um inteiro, pois caso o valor fosse mudado pelo atacante para 2.6 e submetido, o resultado guardado seria 3.

Continua a ser possível para um utilizador mudar os valores no client-side e enviar ao servidor:

The screenshot shows a web application interface for rating a movie. The movie is "The Godfather". The user's rating is "Not Rated". A dropdown menu is open showing options 2, 3, 4, and 5. The user has submitted a comment: "One of the best I've ever saw!". The interface also shows a star rating bar with 4 stars filled and 1 star empty. Below the interface, the browser's developer tools are open, showing the HTML structure of the rating dropdown menu. The dropdown menu is a select element with options 2, 3, 4, and 5.

Contudo, como esses valores são depois verificados pelo servidor, caso não sejam válidos a operação não vai ocorrer, e o rating não vai ser dado:

The screenshot shows the same web application interface, but now the rating dropdown menu shows only the option 1. The user's rating is still "Not Rated". The comment section remains the same. The browser's address bar shows the URL "localhost:3000/movieReview/12345678". The developer tools are not visible in this screenshot.



	user_id	numSerie	rating
*	NULL	NULL	NULL

Base de dados após o utilizador dar submit ao valor manipulado. Verifica-se que a operação não ocorreu.

Com esta validação de dados, é possível agora afirmar que qualquer filme tem ratings genuínas, pois manipulação de valores já não é possível.

Implementação --> CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross Site Scripting')

Versão Não Segura – Linha 484 do ficheiro appNotSafe.js

Esta vulnerabilidade (**STORED XSS**) está presente quando um utilizador com sessão iniciada faz um comentário que contém comandos executáveis (JavaScript, HTML, etc.) sendo estes guardados na base de dados e depois gerados como conteúdo dinâmico da página ao serem recuperados. Um utilizador pode fazer vários comentários. Qualquer utilizador, com sessão iniciada ou não, pode ver estes comentários.

Esta funcionalidade está implementada num método post, que não faz qualquer neutralização/sanitização aos dados introduzidos pelo utilizador antes de os processar e os inserir na base de dados:

```
app.post('/postComment', (req, res) => {

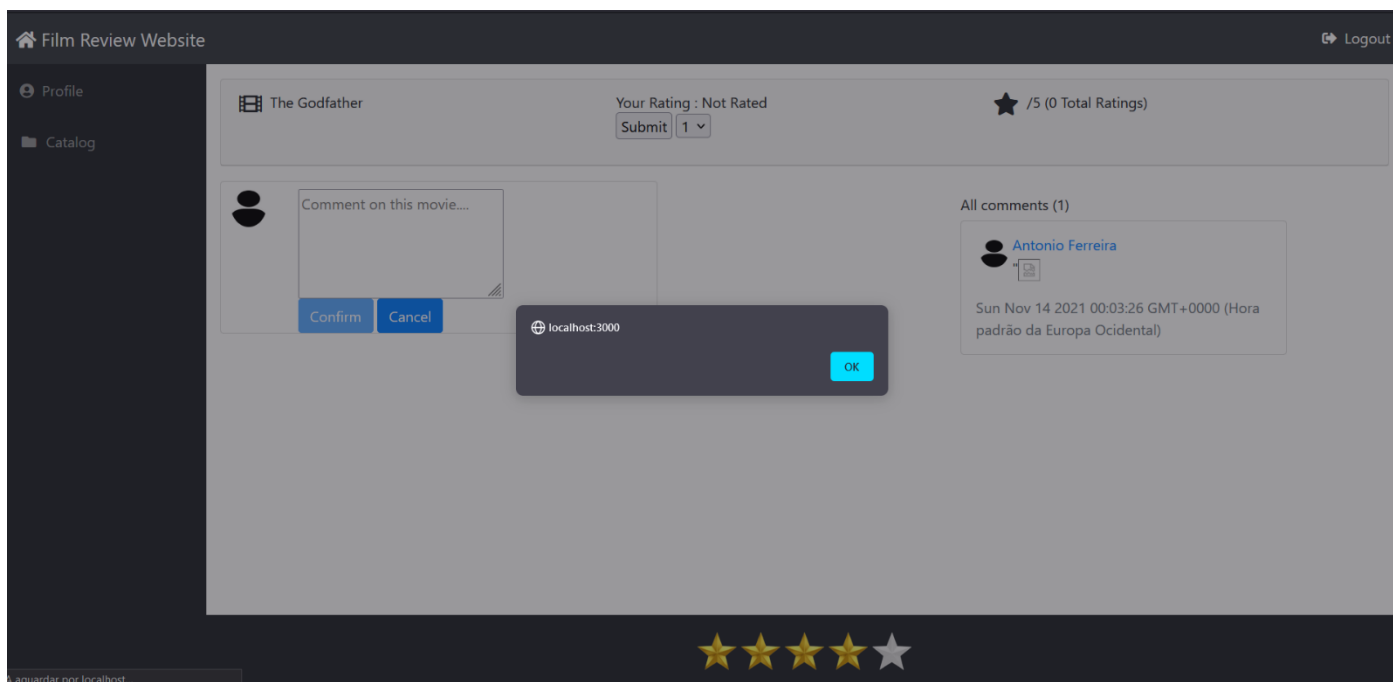
    var sql = 'INSERT INTO review(user_id, numSerie, review) VALUES (' + req.session.userId +
    "',' + req.body.mvToPost + "',' + req.body.cmtToPost + "',' + ')';
    dbconnection.query(sql, (err, data) => {
```

Caso um utilizador introduza um comentário com um comando executável, este vai ser guardado na base de dados.

Result Grid					
Filter Rows:					
	review_id	user_id	numSerie	review	postDate
▶	1	1	12345678	"<img src="fakeimage.jpg" onerror=...	2021-11-14 00:03:26
*	NULL	NULL	NULL	NULL	NULL

Base de dados após o utilizador dar submit ao comentário não neutralizado/sanitizado.

Posteriormente este comentário vai ser carregado na página como parte do conteúdo dinâmico, sendo executado para qualquer utilizador que a visite, estando este com sessão iniciado ou não:



No contexto desta aplicação, esta vulnerabilidade apresenta um problema **extremamente grave**, pois um atacante pode introduzir scripts maliciosos que irão correr para qualquer utilizador que abra a página, tendo portanto um alcance de ataque muito vasto.

Versão Segura – Linha 521 do ficheiro appSafe.js

Para neutralizar esta vulnerabilidade foi utilizada a package **DOMPurify** que, de acordo com a documentação, utiliza tecnologia que o browser disponibiliza e transforma-a num "filtro" contra ataques XSS. Esta documentação foi analisada e o conteúdo parece estar de acordo com o funcionamento.

```
const createDOMPurify = require('dompurify');
const { JSDOM } = require('jsdom');

const window = new JSDOM('').window;
const DOMPurify = createDOMPurify(window);
```

Utilizando esta package foi criada uma função **sanitizeValue** que aceita como input uma string que contém código “sujo”, usa a função **.sanitize** do package **DOMPurify**, e retorna uma string limpa, o que previne ataques XSS.

```
const sanitizeValue = function (value) {
  return DOMPurify.sanitize(value);
}
```

```
var sql = 'INSERT INTO review(user_id, numSerie, review) VALUES (?, ?, ?)';
dbconnection.query(sql, [req.session.userId, req.body.mvToPost, sanitizeValue(req.body.cmtToPost)], (err, data) => {
```

Um utilizador ainda pode inserir um comentário com código/comandos executáveis e maliciosos, sendo estes guardados na base de dados. Contudo devido a ação da função que “limpa” o código, este não vai ser interpretado como código executável da próxima vez que seja carregado na página web.

Film Review Website

Profile

Catalog

The Godfather

Your Rating : Not Rated

Result Grid					Filter Rows:	Edit:	Export/Import:	Wrap Cell Co
	review_id	user_id	numSerie	review	postDate			
▶	1	1	12345678	"	2021-11-14 01:13:35			
✱	NULL	NULL	NULL	NULL	NULL			

Base de dados após o utilizador dar submit ao comentário e este ser “limpo” pela função.

De notar que apesar de este registo parecer igual ao da versão não segura, assim que for carregado na página, vai ser apresentado como uma string e não irá executar qualquer função

Como se pode ver, após um utilizador publicar um comentário deste tipo e outro utilizador aceder à página, nada acontece e apenas vemos o comentário.

Film Review Website
Logout

Profile
Catalog

The Godfather
Your Rating : Not Rated
Submit 1
★ /5 (0 Total Ratings)

Comment on this movie....
Confirm Cancel

All comments (1)

Antonio Ferreira
Sun Nov 14 2021 01:13:35 GMT+0000 (Hora padrão da Europa Ocidental)

★★★★★

Com esta validação de dados, é possível agora assegurar que mesmo que um utilizador poste um comentário com comandos ao princípio executáveis, estes ao serem gerados na página não irão executar.

Implementação --> CWE-89: Improper Neutralization of Special Elements used in na SQL Command ('SQL Injection')

Versão Não Segura – Linha 467 do ficheiro appNotSafe.js

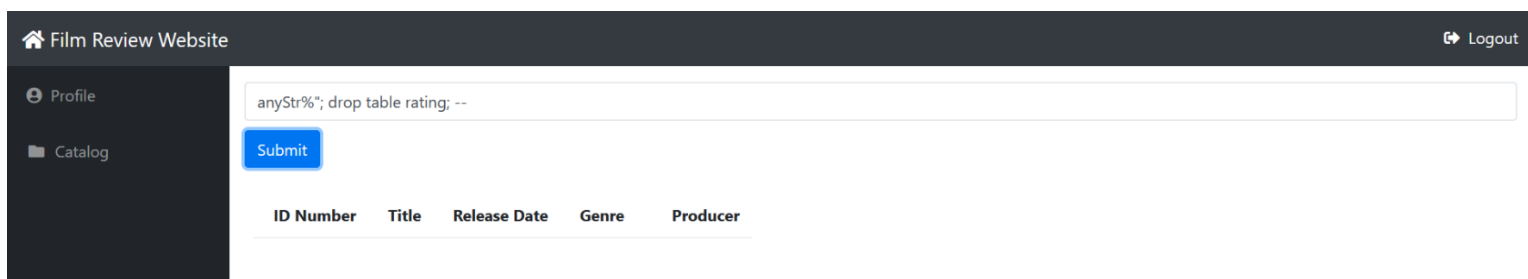
Esta vulnerabilidade está presente quando um utilizador procura por um título com um input que contém comandos SQL, sendo os elementos especiais deste input não neutralizados pelo servidor antes do seu processamento.

Esta funcionalidade está implementada num método post:

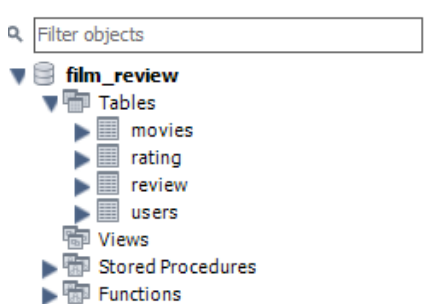
```
app.post('/searchTitle', (req, res) => {

  const sql = 'SELECT * FROM movies WHERE title LIKE "%" + req.body.first + "%"'
  dbconnection.query(sql, (err, data) => {
```

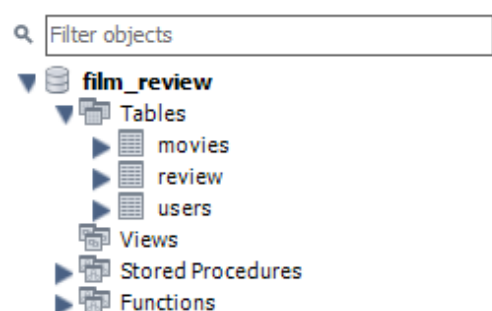
Caso um utilizador introduza um input com elementos especiais, estes podem alterar o comando SQL no qual iriam ser corridos, podendo até mesmo correr vários comandos.



Apesar de à partida parecer que o resultado da pesquisa é apenas um ecrã vazio (pois não existe nenhum filme com aquele título), o que realmente aconteceu foi a eliminação da tabela rating da base de dados.



Base de dados antes do utilizador fazer a pesquisa



Base de dados após o utilizador fazer a pesquisa

No contexto desta aplicação, esta vulnerabilidade apresenta um problema **extremamente grave**, pois um atacante pode alterar o comando SQL de maneira a ter acesso a informação privilegiada, ultrapassar verificações de segurança ou até mesmo eliminar informação/tabelas/base de dados.

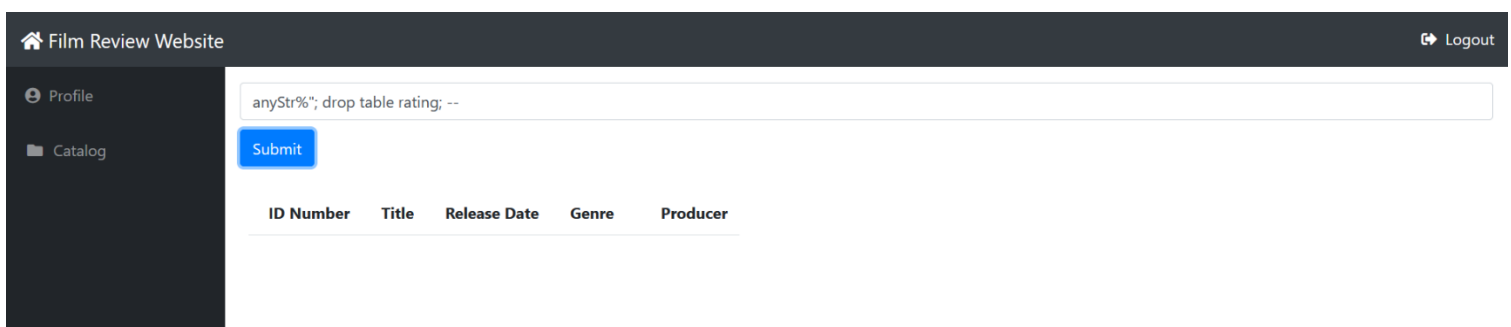
Versão Segura – Linha 503 do ficheiro appSafe.js

Para neutralizar esta vulnerabilidade foi utilizada uma funcionalidade do mysql em node.js chamada *placeholders*, para filtrar e fazer "escape" ao input do utilizador antes deste ser processado no comando SQL.

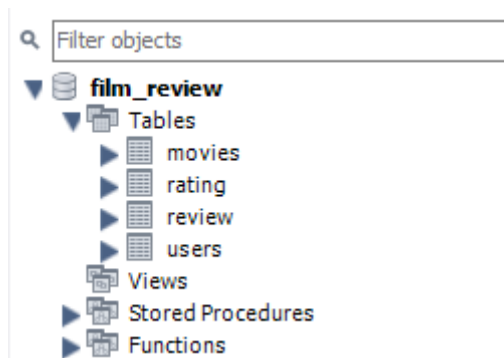
```
app.post('/searchTitle', (req, res) => {

  const sql = 'SELECT * FROM movies WHERE title LIKE ?'
  var aux = '%' + req.body.first + '%'
  dbconnection.query(sql, [aux], (err, data) => {
```

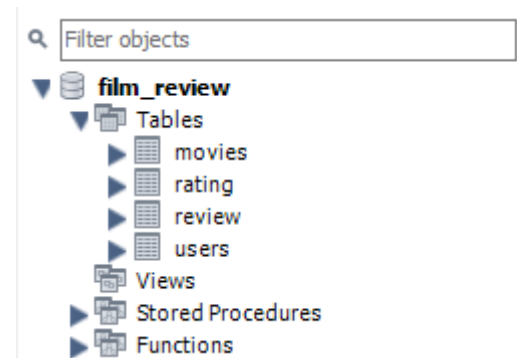
Apesar do utilizador inserir um input com elementos especiais, estes já não vão alterar o comando SQL, pois já foram “filtrados” anteriormente.



Nesta situação o resultado da pesquisa é apenas um ecrã (pois não existe nenhum filme com aquele título), sendo que nada aconteceu à base de dados, mantendo-se esta inalterada.



Base de dados antes do utilizador
fazer a pesquisa



Base de dados após o utilizador
fazer a pesquisa

Com esta validação de dados, é possível agora assegurar que mesmo que um utilizador faça uma pesquisa por comandos SQL, estes não alteram de qualquer maneira o comando SQL a executar pelo servidor sobre esse input.

Implementação --> CWE-328: Use of a Weak Hash

Implementação --> CWE-759: Use of One Way Hash without a Salt

Versão Não Segura – ficheiro appNotSafe.js

Estas vulnerabilidades estão presentes quando um utilizador se regista/muda password e o servidor guarda na base de dados a password através dum processo que:

- Usa uma função criptográfica de hash fraca – neste caso *MD5*
- Não utiliza, em conjunto com essa função de hash unilateral, um salt

Registo de utilizador está implementado num método post (linha 314) :

```
var hashPassword = crypto.createHash('md5').update(password).digest('hex');
var sql = 'INSERT INTO film_review.users (email, username, passwordHash) VALUES (?, ?, ?)';

dbconnection.query(sql, [email, username, hashPassword], (err, data) => {
```

Mudança de password está implementada num método post (linha 392):

```
var hashNewPassword = crypto.createHash('md5').update(newPassword).digest('hex');
var sql = 'UPDATE film_review.users SET passwordHash=? WHERE id=?';

dbconnection.query(sql, [hashNewPassword, req.session.userId], (err, data) => {
```

Como consequência de não usar salt, dois utilizadores que se registem com a mesma password/alterem ambos para a mesma password, terão também na base de dados guardada a mesma passwordHash.

Acrescido ao fato da hash ser fraca, isto torna o ataque facilitado, pois uma hash fraca permite uma execução mais rápida/eficiente de ataques pré imagem, onde o atacante determina o input original através da hash, e o não uso de salt permite ataques de tabela arco-íris.

Result Grid					
Filter Rows: <input type="text"/>					
Edit:					
Export/Import:					
Wrap Cell Content:					
	id	username	email	isAdmin	passwordHash
▶	1	Antonio Ferreira	antonio@gmail.com	0	562e02cb6ca7f7805c7809728f35a2a8
	2	Luis Couto	luis@gmail.com	0	562e02cb6ca7f7805c7809728f35a2a8
✱	NULL	NULL	NULL	NULL	NULL

Base de dados após 2 utilizadores se registarem com a mesma password: **k3=p~'zF~X~wH/QU**

De notar que devido a não usar salt, as passwordHash são iguais para passwords iguais.

No contexto desta aplicação, esta vulnerabilidade apresenta um problema **extremamente grave**, pois um atacante pode obter as passwords dos utilizadores, tendo assim acesso às suas contas pessoais.

Versão Segura – ficheiro appSafe.js

Para neutralizar estas vulnerabilidades foi utilizada a popular package **bcrypt**, que contém funções baseadas na cifra **BlowFish**, para além de incorporar um salt que serve como input adicional no processo de hash de dados.

Mais concretamente foi usada a função **bcrypt.hash**, que ao receber a string password (input do utilizador) e um valor das salt rounds desejadas (10 que é o recomendado no website pelos developers), retorna a password hashed, que seguidamente vai ser armazenada na base de dados.

Registo de utilizador está implementado num método post (linha 325) :

```
var sql = 'INSERT INTO film_review.users (email, username, passwordHash) VALUES (?, ?, ?)';

bcrypt.hash(password, saltRounds, (err, hash) => {
  dbconnection.query(sql, [email, username, hash], (err, data) => {
```

Mudança de password está implementada num método post (linha 413):

```
var sql = 'UPDATE film_review.users SET passwordHash=? WHERE id=?';
bcrypt.compare(password, data[0].passwordHash, (err, result) => {

  if (result) {

    bcrypt.hash(newPassword, saltRounds, (err, hash) => {
      dbconnection.query(sql, [hash, req.session.userId], (err, data) => {
```

Com o uso deste package, é possível armazenar as passwordHash de uma maneira a que base de dados seja resistente a ataques pré imagem e tabela arco-íris, pois estes tornar-se-iam muito dispendiosos em termos computacionais.

O **bcrypt** também se adapta, sendo resistente a ataques "brute-force search", onde é utilizado mais poder computacional.

Result Grid

Filter Rows:

Edit:

Export/Import:

Wrap Cell Content:

	id	username	email	isAdmin	passwordHash
▶	1	Antonio Ferreira	antonio@gmail.com	0	\$2b\$10\$Ahu8WAw5GLUDi6jE7BW9FePkB1E5EMOCcf3WfFB654t0EO4ek3iLa
	2	Luis Couto	luis@gmail.com	0	\$2b\$10\$ZGwdeGytNH8T..stHubBvOLQhwU6vCVjTk4uFlvKywv0sk7EYvrku
*	NULL	NULL	NULL	NULL	NULL

Base de dados após 2 utilizadores se registarem com a mesma password: **k3=p~'zF~X~wH/QU**

De notar que, ao contrário de anteriormente, mesmo com a mesma password, são geradas passwordHash diferentes.

Com a utilização deste package, é possível agora ter uma segurança mais robusta, com uma base de dados mais segura face a vários ataques que visam roubar passwords.