WA3358 Security Fundamentals for Software Engineers

Student Labs

Web Age Solutions

Lab 01 - A Simple Application Security Demo Lab

Let us start with a simple server-side JavaScript application to study and learn the security problems (we will call it vulnerabilities) in a web application.

Use your lab machine. Let us use the following working directory for this lab.

C:/appsecuritylabs.

1. Open a command window, create a directory for lab work. Type the following commands, one by one.

   cd C:
   mkdir appsecuritylabs
   cd appsecuritylabs
   title simple-app-security-demo

```
simple-app-security-demo

C:\Users\saravanan>cd C:\

C:\>mkdir appsecuritylabs

C:\>cd appsecuritylabs

C:\appsecuritylabs>title simple-app-security-demo

C:\appsecuritylabs>_
```

2. Our demo application is available at: https://github.com/foxwas/app-security-demo

   In the command window, Type the following command to clone the demo app repository code.

   git clone https://github.com/foxwas/app-security-demo.git

```
Select simple-app-security-demo

C:\appsecuritylabs>git clone https://github.com/foxwas/app-security-demo.git
Cloning into 'app-security-demo'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.

C:\appsecuritylabs>_
```

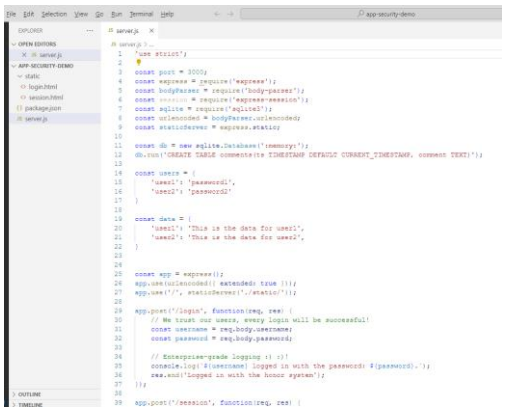
3. In the command window, Type the command to change directory to app-security-demo

   cd app-security-demo

Review the resources.

4. Use Visual Studio code to review the source code files and package.json.

Open a File Explorer, navigate to **C:\appsecuritylabs\app-security-demo**, open the project in Visual Studio code.



5. Let us install application dependencies using **npm install** command. In the command window, type the following command to install dependencies defined in package.json

npm install

```
Select simple-app-security-demo
C:\appsecuritylabs\app-security-demo>npm install
npm WARN deprecated @npmcli/move-file@1.1.2: This functionality has been moved to @npmcli/fs

added 194 packages, and audited 195 packages in 13s

23 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

6. Type the following command to view the list of dependencies.

npm list

```
C:\appsecuritylabs\app-security-demo>npm list
appsecuitydemo@1.0.0 C:\appsecuritylabs\app-security-demo
+-- express-session@1.18.0
+-- express@4.19.2
`-- sqlite3@5.1.7


C:\appsecuritylabs\app-security-demo>
```

7. The package.json contains a script to start the server.

```
"scripts": {
 "start": "node server.js",
 },
```

If we use **npm start** command, it will run the command: "**node server.js**" to launch the application in node environment.

In the command window, type the following command to start the application.

npm start

```
npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
```

The application will start and ready to serve request at

http://localhost:3000

8. Open a browser and test the following pages, one by one.

http://localhost:3000

http://localhost:3000/login.html

http://localhost:3000/xss

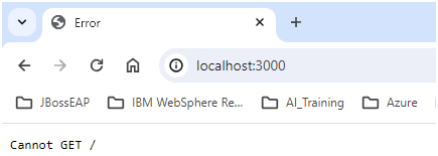
http://localhost:3000/session.html

http://localhost:3000/fox

9. **Can you find any security problems/vulnerabilities ? List them and document them .**

The following steps will walk you through my findings.

10. Open a browser, go to http://localhost:3000



Error!

The application should have a welcome page or landing page.

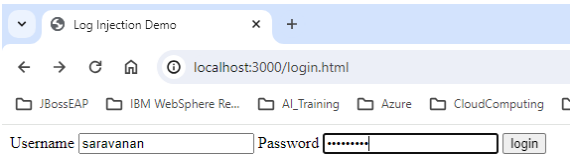
And an error page with meaningful redirection links to main pages.

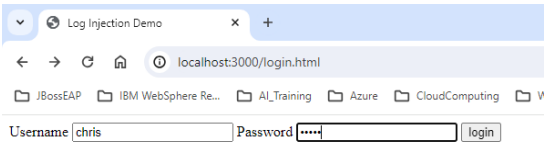
11. Open a browser, go to http://localhost:3000/login.html

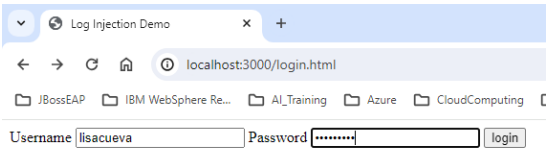
In Log Injection demo page, you can **enter username and password**.

Enter any username and password, click on **login** button. What is the response? See the server console. Do you see any messages?

User Agent View







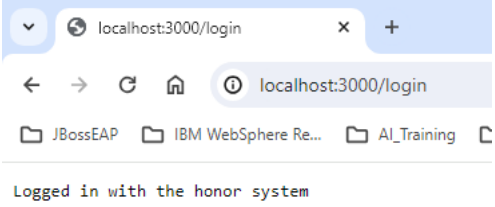
Server Console Log

```
npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
saravanan logged in with the password: saravanan.
```

```
npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
saravanan logged in with the password: saravanan.
chris logged in with the password: chris.
```

```
npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
saravanan logged in with the password: saravanan.
chris logged in with the password: chris.
lisacueva logged in with the password: lisacueva.
```

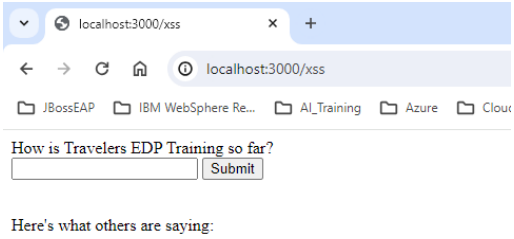
This application **accepts any username and password** and returns the following response: Logged in with the honor system



This application needs to implement strong authentication and authorization mechanisms.

12. Open a browser, go to the url:

http://localhost:3000/xss



You can enter your comments in the text field component and click on the **Submit** button.

The application will store the comments in a database table **comments**. See the source code of server.js for more information.

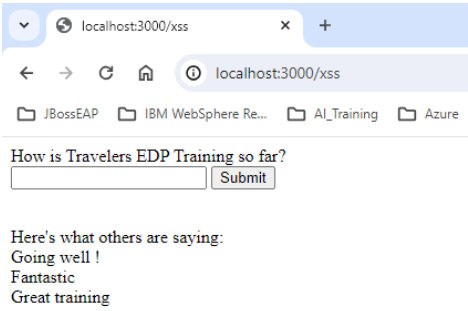
In the browser page, enter the following comments in the text field component one by one and submit.

Great training

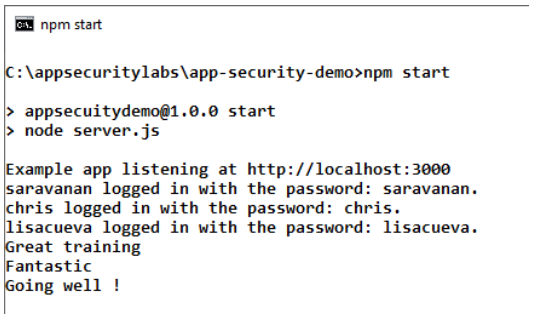
Fantastic

Going well !

User Agent View



Server Console Log

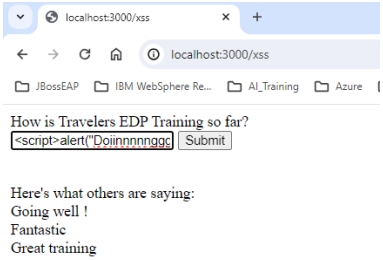


13. In the same page, (http://localhost:3000/xss) enter the following comment in the text field component.

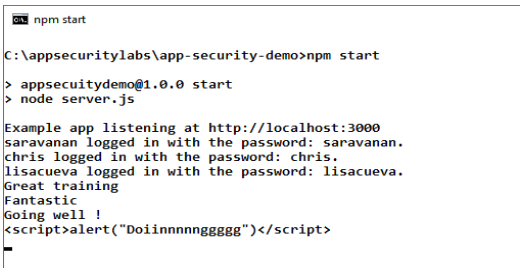
<script>alert("Doiinnnnngggg")</script>

Click on the **Submit** button.

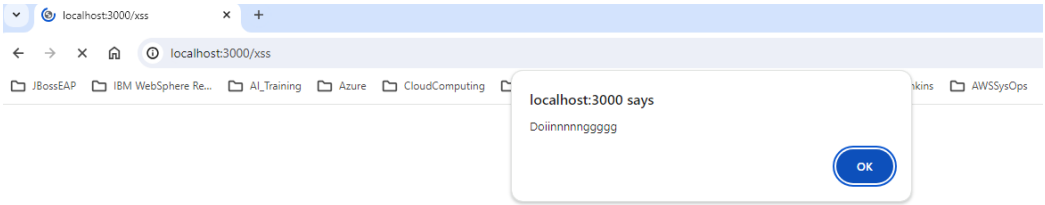
Browser page with input



Server Console Log



Browser response



This is an example of code injection. Click on the **OK** button in the alert box.

Can you see this comment on the page?

Check the server console log message.

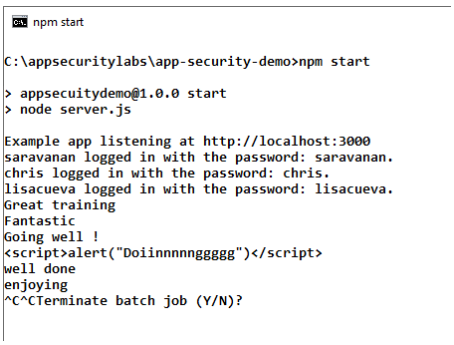
14. **Refresh the page. Every time you refresh the page**, It will display the alert message dialog.

Type some new comments and test again.

How can stop this annoying alert box and its message dialog ?

15. Go to server console window**.** Move the cursor inside the terminal window, and type the command: **CTRL + C**

Type **Y** to terminate batch job.



16. Start the server again using the command: **npm start**

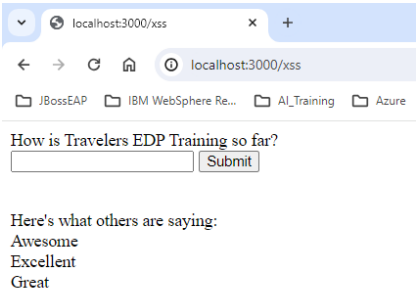
```
npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
```

17. Open a browser, go to the page

http://localhost:3000/xss

Type a few good comments and click on the **Submit** button.

Browser page



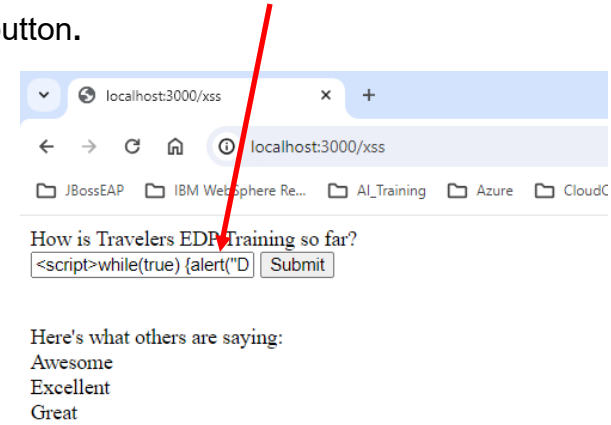
Server Console Log

```
npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
Great
Excellent
Awesome
```

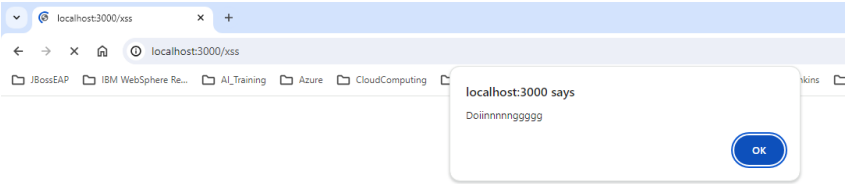
18. Let us inject an infinite loop code (I call it devil's loop, sorry) as a user comment and see what happens. Type the following comment in the text field component.

<script>while(true) {alert("Doiinnnnngggg")}</script>

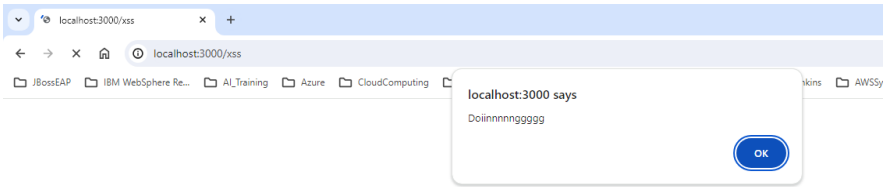
Click on the **Submit** button.



What is the response?

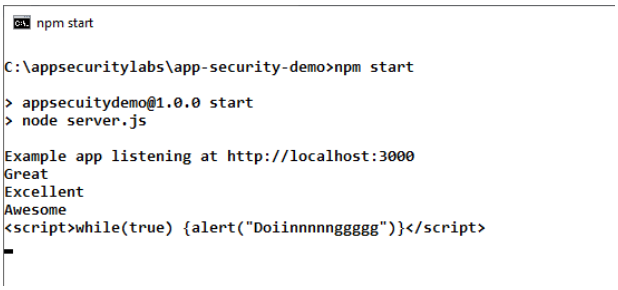


Click on the OK button in the alert box. What is the response?

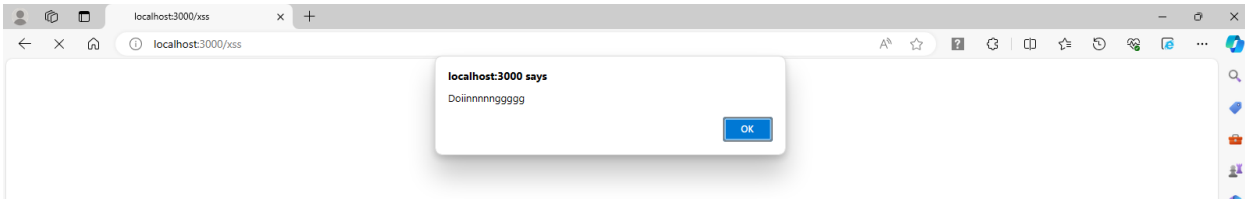


The alert box will prompt you to respond again. The browser will continue to prompt the alert box forever !

See the Server Console Log



19. Open another browser application, Firefox or Microsoft edge and try to load the page http://localhost:3000/xss



20. Let us terminate the server and then restart the server. **Go to the server terminal window, and type CTRL + C to terminate the server.**

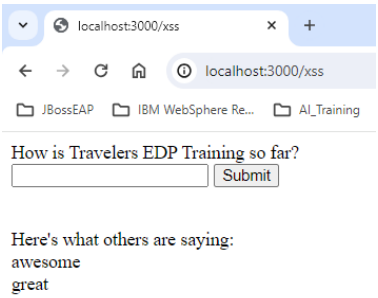
21. In the command window, start the server using the command **npm start**

22. Open a browser, go to the page

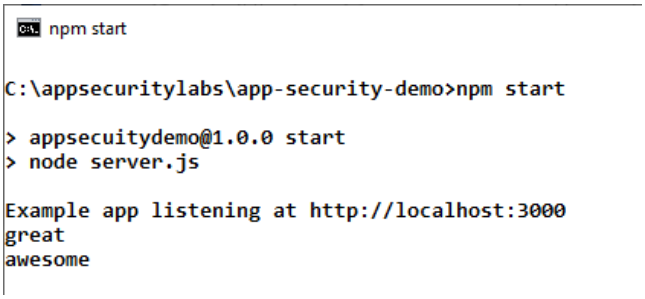
http://localhost:3000/xss

Enter a few clean positive comments and click on the Submit button.

Browser page

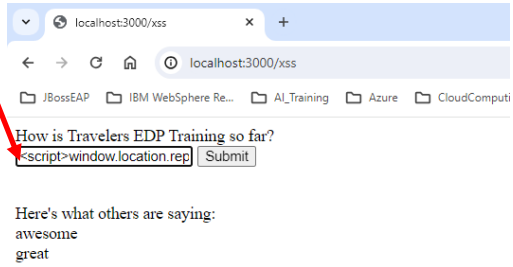


Server Console Log



23. Let us inject code to redirect to different web site. Type the following comment in the text field component.

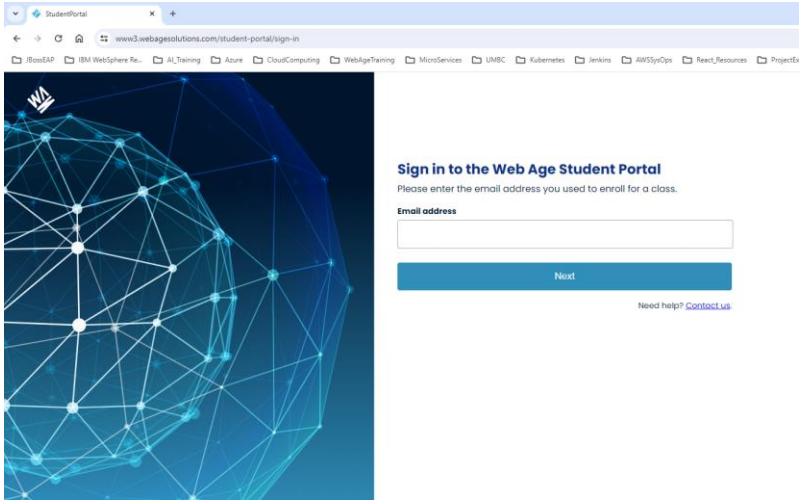
<script>window.location.replace("https://www3.webagesolutions.com/student-portal/sign-in")</script>



Click on the **Submit** button.

What is the response ?

Are you getting the following page ?



See the server console log messages

```
Select npm start

C:\appsecuritylabs\app-security-demo>npm start

> appsecuitydemo@1.0.0 start
> node server.js

Example app listening at http://localhost:3000
great
awesome
<script>window.location.replace("https://www3.webagesolutions.com/student-portal/sign-in")</script>
```

Open another browser tab or window and type the URL

http://localhost:3000/xss

Open another browser application (Firefox of Microsoft Edge) and try to load the page

http://localhost:3000/xss

Every time you load the page, it will redirect to WebAge Solutions Student portal page.

24. Let us terminate the server and then restart the server. **Go to the server terminal window, and type CTRL + C to terminate the server.**

In the command window, start the server using the command **npm start**

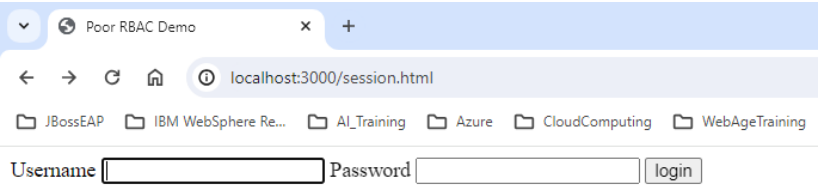
25. Open a browser, go to the page

http://localhost:3000/xss

Enter a few clean positive comments and click on the Submit button.

26. Open another browser window, go to the page

http://localhost:3000/session.html

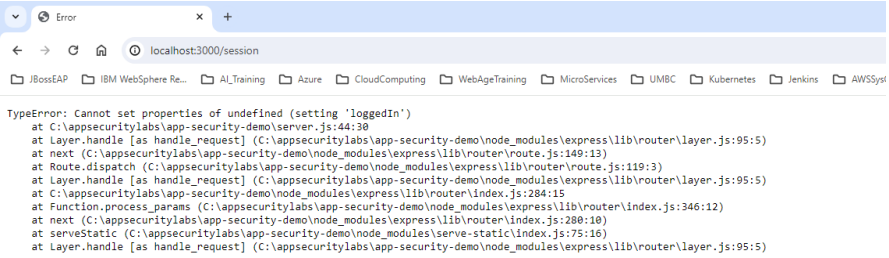


27. Enter the following set of username and password and test the application responses.

Username	**Password**
user1	password1
user2	password2
sara	sara

What are the responses?

Are you getting error like the one in the following screen shot?



28. **What can you do to fix all the problems, issues, security vulnerabilities ?**

Lab 02 - React Application SSL Lab

In this lab, let us configure a react app running on a development server to serve requests using the protocol **https!**

We will need to

- Create and develop a new react project OR Use an existing react application
- Generate and Configure server key and ssl certificate
- Install the certificate to a local Certificate Authority
- Configure the **react-scripts start** to use the generated certificate and key

Use an Existing React Application

1. Open a command window, create a directory for this lab.

   mkdir react-labs

   cd react-labs

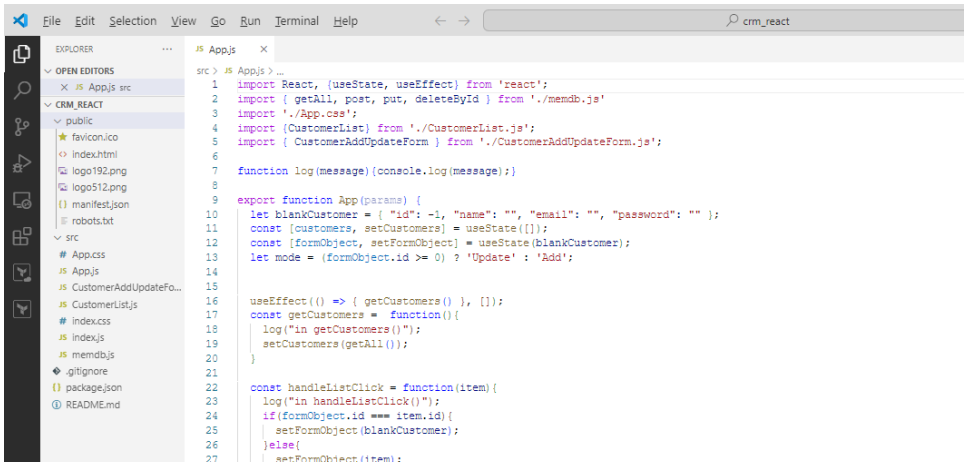
2. Clone the sample react application using the following git command.

   git clone https://github.com/foxwas/was-react-apps.git

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4170]
(c) Microsoft Corporation. All rights reserved.

C:\react-labs>git clone https://github.com/foxwas/was-react-apps.git
Cloning into 'was-react-apps'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (25/25), done.
Receiving objects:  56% (14/25)sed 25 (delta 0), pack-reused 0
Receiving objects: 100% (25/25), 27.37 KiB | 4.56 MiB/s, done.

C:\react-labs>dir
 Volume in drive C is Windows
 Volume Serial Number is 1255-3BEE

 Directory of C:\react-labs

04/04/2024  12:27 PM    <DIR>          .
04/04/2024  12:27 PM    <DIR>          ..
04/04/2024  12:27 PM    <DIR>          was-react-apps
               0 File(s)              0 bytes
               3 Dir(s)  30,356,709,376 bytes free
```

3. In this project was-react-apps, let us use the react app: **crm_react**

   Review this react app code.



4. In the command window type the following commands to change the directory to crm_react and then install application dependencies.

   cd crm_react

   npm install

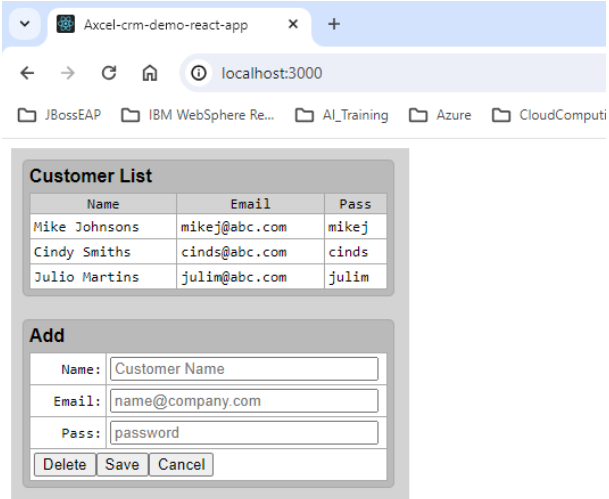
```
C:\react-labs\was-react-apps>cd crm_react

C:\react-labs\was-react-apps\crm_react>npm install
```

5. Launch the react application on a development server using the following command

   npm start

```
Windows PowerShell
Compiled successfully!

You can now view Axcel-crm-demo-react-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.76:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

6. Open a browser, go to http://localhost:3000



7. You can view existing customers, update the customers data, and add a new customer. Test the application functional CRUD operations.

Generate and Configure server key and ssl certificate

8. Open another command window, change directory to C:\react-labs\was-react-apps\crm_react

   cd C:\react-labs\was-react-apps\crm_react

   Type the following command to verify the openssl version.

   openssl -version

9. Use openssl tool to Generate self-signed ssl certificate using the following command

   Enter your data to complete the interactive shell prompts as shown in the following screenshot.

```
C:\Windows\System32\cmd.exe

C:\react-labs\was-react-apps\crm_react>openssl req -x509 -newkey rsa:2048 -nodes -sha256  -keyout server.key -out server.crt
..+.........+.......+...+........+.........+.....+....+++++++++++++++++++++++++++++++++++++++*..+....+........++++++++++++++
...+..+...+....++++++
......+...+..+........+.+..+.++++++++++++++++++++++++++++++++++++++++*.+...+...+.+......+.............++++++++++++++++++++++
......+......+......+.........................+......+......+..+.+..+...+......+.+..+.............+...+............+..+......+
...+........+................+....+...+.....+.........+..+........++++++
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MI
Locality Name (eg, city) []:Farmington Hills
Organization Name (eg, company) [Internet Widgits Pty Ltd]:WebAge Solutions
Organizational Unit Name (eg, section) []:Training
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:

C:\react-labs\was-react-apps\crm_react>
```

10. This tool will generate **server.key** and **server.crt**. Can you see the files in the current directory?

```
 Directory of C:\react-labs\was-react-apps\crm_react

04/04/2024  01:02 PM    <DIR>          .
04/04/2024  01:02 PM    <DIR>          ..
04/04/2024  12:27 PM               333 .gitignore
04/04/2024  12:42 PM    <DIR>          node_modules
04/04/2024  12:40 PM           602,665 package-lock.json
04/04/2024  12:27 PM               865 package.json
04/04/2024  12:27 PM    <DIR>          public
04/04/2024  12:27 PM             3,429 README.md
04/04/2024  01:02 PM             1,348 server.crt
04/04/2024  01:02 PM             1,732 server.key
04/04/2024  12:27 PM    <DIR>          src
               6 File(s)        610,372 bytes
               5 Dir(s)  30,011,727,872 bytes free

C:\react-labs\was-react-apps\crm_react>_
```

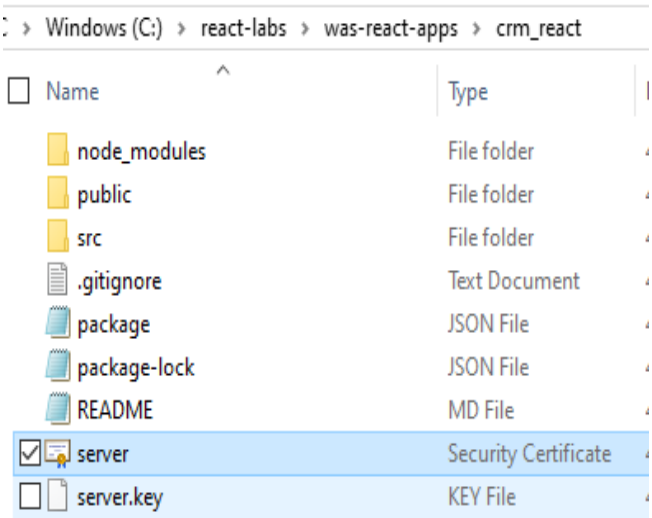
11. Use the following openssl command to view the contents in **server.crt**

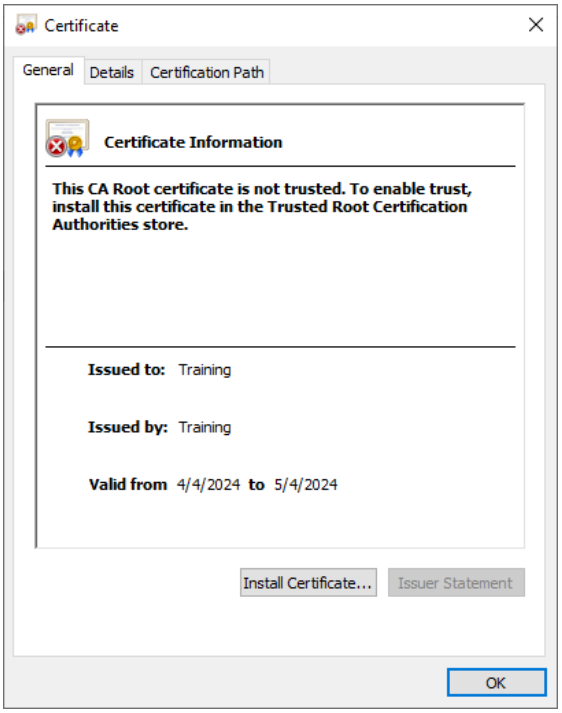
    openssl x509 -in server.crt -noout -text

```
C:\Windows\System32\cmd.exe

C:\react-labs\was-react-apps\crm_react>openssl x509 -in server.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            5e:80:1b:c6:e5:ce:89:da:3e:0f:5c:34:d8:9b:a7:2d:20:67:64:67
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=MI, L=Farmington Hills, O=WebAge Solutions, OU=Training
        Validity
            Not Before: Apr  4 17:02:50 2024 GMT
            Not After : May  4 17:02:50 2024 GMT
        Subject: C=US, ST=MI, L=Farmington Hills, O=WebAge Solutions, OU=Training
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ac:02:95:74:e9:fe:23:61:28:28:6d:95:41:ee:
                    0a:92:3e:c2:84:fe:23:94:19:8d:86:e7:91:6f:9d:
                    5b:2e:c8:b4:33:e4:b9:46:70:eb:5b:a3:e2:a2:ed:
                    d1:b9:73:88:ae:f1:0d:76:0e:01:00:a0:77:4f:bc:
                    7e:47:4e:10:42:d6:5c:ab:4e:59:ab:29:99:2c:c5:
                    eb:24:4e:9d:52:db:4c:56:81:fe:5e:dc:91:63:f6:
                    21:57:d5:93:ef:fe:b1:89:40:cc:66:af:2a:00:a6:
                    c4:8b:f9:72:14:dc:cf:73:49:97:7f:71:ea:a5:6b:
                    41:e4:b2:cf:9d:e4:f2:61:03:f9:ac:44:eb:49:70:
                    20:2c:fe:1c:31:25:ae:6d:ee:39:3f:1e:d6:43:3f:
                    b7:6c:bb:9c:16:41:dc:d6:31:f1:94:b8:72:6f:da:
                    59:ed:85:9c:3d:2e:f2:c0:54:47:ff:b7:1d:58:d9:
                    d3:e0:c1:a0:cf:ea:a3:22:dc:a1:7c:d3:86:27:4c:
                    45:61:73:5d:b5:07:f6:b4:73:10:f8:db:09:e8:11:
                    55:14:88:0c:09:2a:9a:68:05:31:2c:ea:5a:a2:0a:
                    d6:e2:32:ac:2d:ce:4b:8a:31:d7:5b:85:a4:d2:13:
                    77:c3:46:f1:53:7f:a7:eb:d3:9b:01:a0:88:73:b0:
                    c4:c7
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                8B:D8:4E:04:5F:29:73:FE:FC:6E:F2:A6:43:0A:CA:4B:7D:1F:51:10
            X509v3 Authority Key Identifier:
                8B:D8:4E:04:5F:29:73:FE:FC:6E:F2:A6:43:0A:CA:4B:7D:1F:51:10
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        aa:9f:d3:9a:d8:4f:f3:a8:96:24:b1:b1:15:92:9c:b6:62:f8:
        42:ac:fd:4c:6b:7d:54:ec:36:64:44:cb:9e:11:2a:90:c8:91:
        68:ed:94:d1:b6:3e:b7:26:4f:fa:ec:d6:d7:3a:11:60:5a:5d:
        d3:fe:92:60:0c:90:43:09:88:7e:26:d9:83:f7:eb:02:d7:c4:
        18:76:7f:8f:66:d7:19:52:6a:80:8e:70:78:69:ab:bb:1c:f5:
        04:bc:20:3d:fa:58:9c:d2:9b:1a:b7:fd:c0:44:bb:af:c9:a4:
```

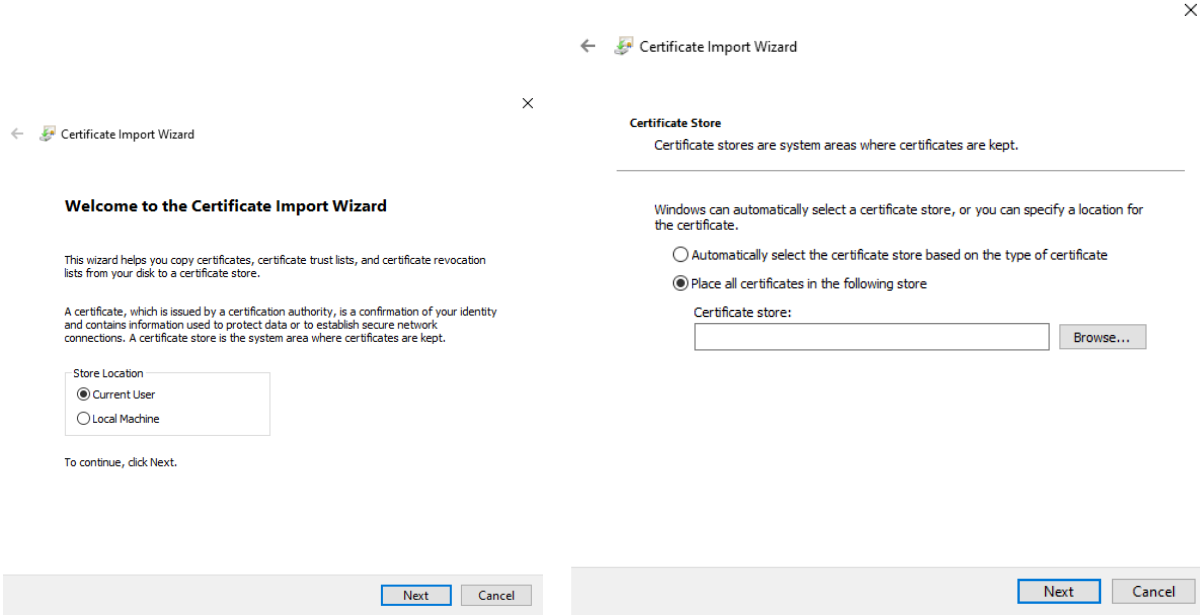
Install the certificate to a local Certificate Authority

12. To install the certificate, open a File Explorer, navigate to the certificate directory, and double click on the certificate.

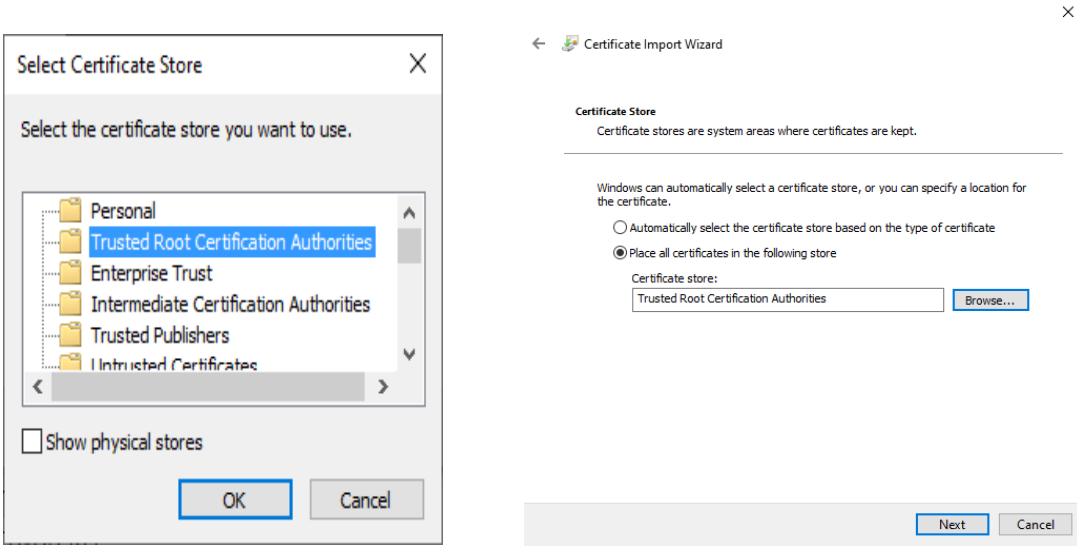




13. Click on **Install Certificate.** In the certification import wizard, select the **current user,** click on **Next.**
    Select **Place all certificates in the following store**



Browse → **select Trusted Root Certification Authorities**



Click on **Finish**.

Configure the react-scripts start to use the generated certificate and key

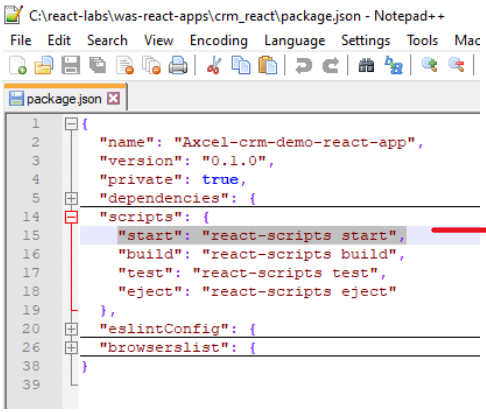
14. In your react project, modify the **package,json** scripts to start using HTTPS

    Replace the existing script command: **"start": "react-scripts start",**

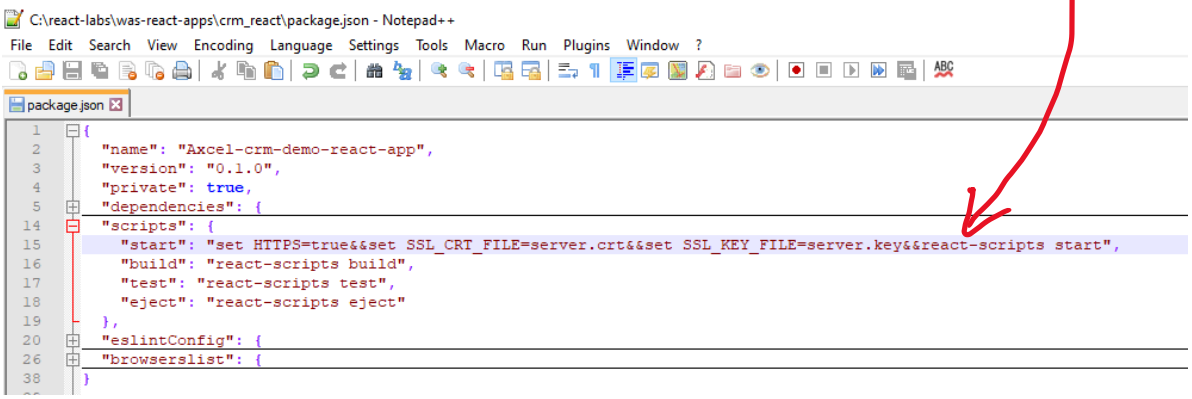
    With the **following script command**:

"start": "set HTTPS=true&&set SSL_CRT_FILE=server.crt&&set SSL_KEY_FILE=server.key&&react-scripts start",

package.json



Modified package.json using HTTPS



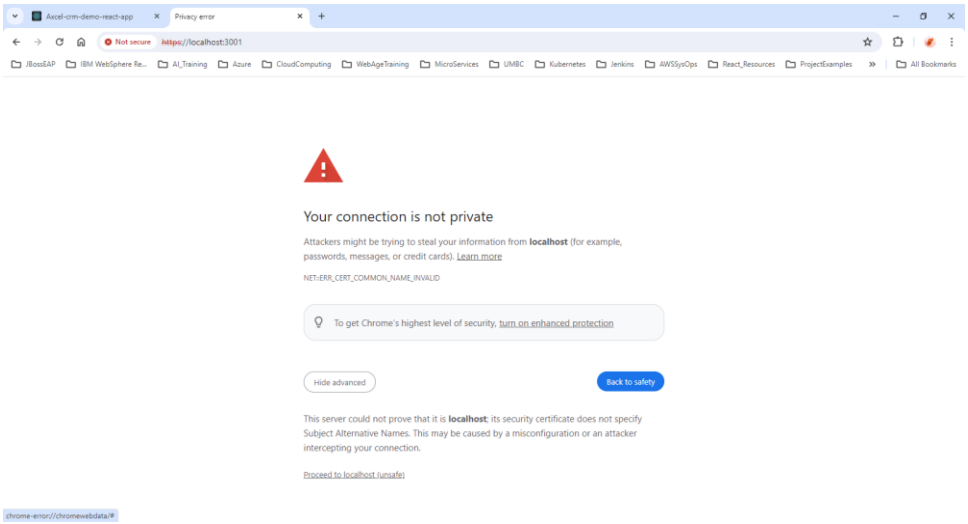
and **save** the file.

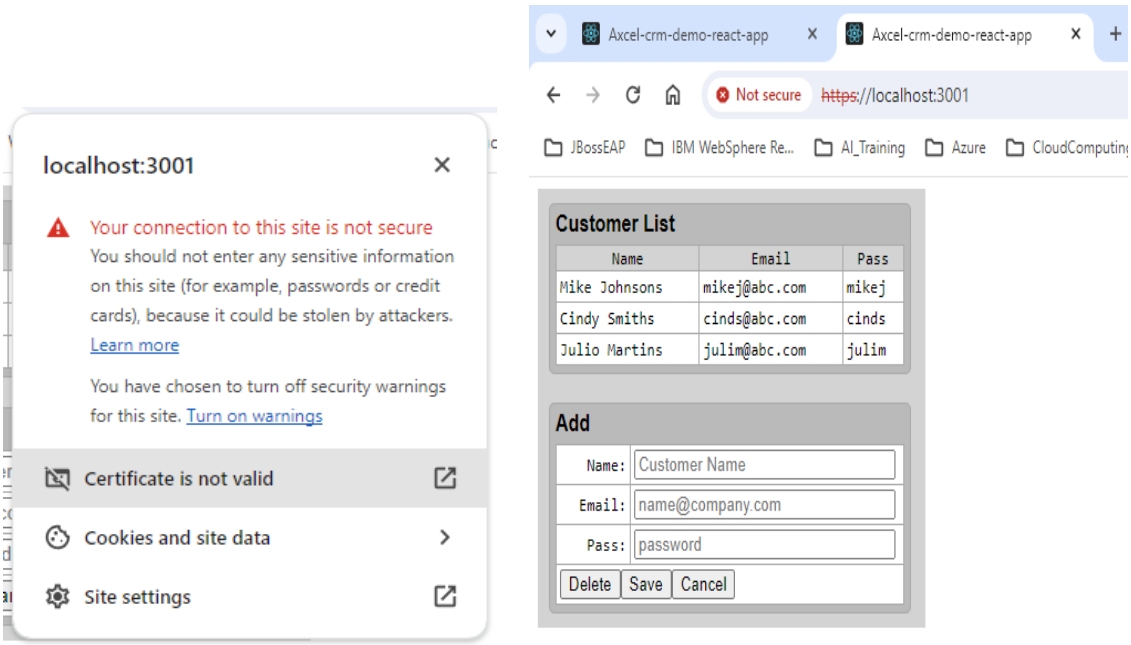
15. Open a command window, cd to C:\react-labs\was-react-apps\crm_react

    Type the npm command to run the application

    npm start

16. Open a browser, go to **https://localhost:3001**





What can you do to make your ssl certificate valid?

Lab 03 - Node.js Application SSL Lab

In this lab, let us develop a node.js server-side application running to serve requests using the protocol **https!**

1. Open a command window, create a directory for this lab and cd to the directory.

   mkdir nodejs-labs

   cd nodejs-labs

```
C:\WINDOWS\system32\cmd.exe

C:\>mkdir nodejs-labs

C:\>cd nodejs-labs

C:\nodejs-labs>
```

2. Run the following command to initialize a new npm project

   npm init --y

```
C:\WINDOWS\system32\cmd.exe

C:\nodejs-labs>npm init --y
Wrote to C:\nodejs-labs\package.json:

{
  "name": "nodejs-labs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


C:\nodejs-labs>
```

3. Install express dependency. Type the following command

   npm install --save express

```
C:\WINDOWS\system32\cmd.exe

C:\nodejs-labs>npm install --save express
npm [redacted] created a lockfile as package-lock.json. You should commit this file.
npm WARN nodejs-labs@1.0.0 No description
npm WARN nodejs-labs@1.0.0 No repository field.

+ express@4.19.2
added 64 packages from 41 contributors and audited 64 packages in 4.29s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities


C:\nodejs-labs>
```

3. Review the package.json document.

```
package.json
1  {
2    "name": "nodejs-labs",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "express": "^4.19.2"
14   }
15 }
16
```

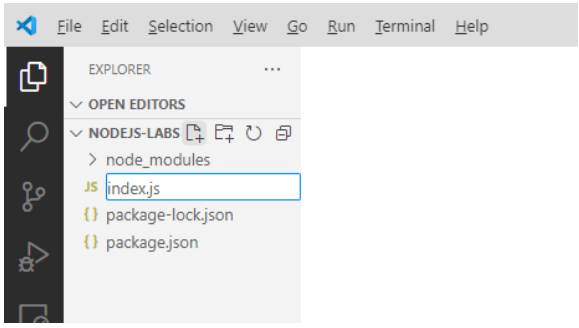
4. Add the following line to **scripts** block in package.json.

   "start": "node index.js",

and **save** it.

```
package.json
1  {
2    "name": "nodejs-labs",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "start": "node index.js",
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "dependencies": {
14     "express": "^4.19.2"
15   }
16 }
17
```

5. We must create this file **index.js** in our project environment.
   Add a file called **index.js** to your application.



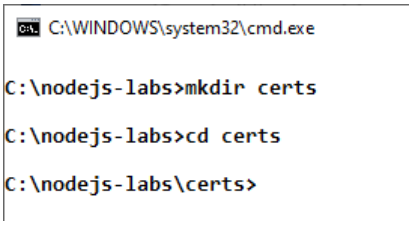
6. Add the following code in **index.js** and save it.

```
const express= require('express')
const https=require('https')
const fs=require('fs')
const path=require('path')
const app=express();
app.use('/',(req,res,next)=>{
res.send('hello, This response is from SSL  node Server !')
})
const options={
key: '',
cert: ''
}
const sslServer=https.createServer(options,app);
sslServer.listen(8443,()=>{
console.log('Secure server is listening on port 8443')
})
```

9. We must create ssl key and ssl certificate and configure the code to use them. Create a directory to store ssl certificates Type the following commands in the command window

mkdir certs

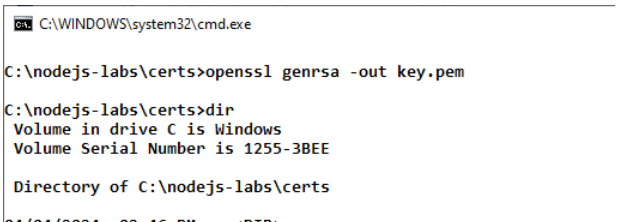
cd certs

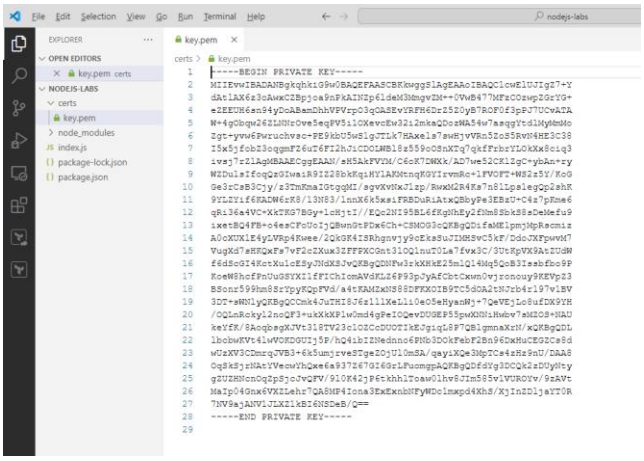


10. Let us generate a private key using openssl. Use the following command

openssl genrsa -out key.pem

This command will generate the private key and save it in **key.pem** file inside certs directory.





11. Create a CSR (Certificate Signing Request). Use the following command

openssl req -new -key key.pem -out csr.pem

This command will prompt a dialog to get input and run. Provide meaningful input.

```
C:\WINDOWS\system32\cmd.exe

C:\nodejs-labs\certs>openssl req -new -key key.pem -out csr.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Michigan
Locality Name (eg, city) []:Farmington Hills
Organization Name (eg, company) [Internet Widgits Pty Ltd]:WebAge Solutions
Organizational Unit Name (eg, section) []:Training
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:

C:\nodejs-labs\certs>_
```

Check the directory. You will see the **csr.pem** file.

```
C:\WINDOWS\system32\cmd.exe

C:\nodejs-labs\certs>dir
 Volume in drive C is Windows
 Volume Serial Number is 1255-3BEE

 Directory of C:\nodejs-labs\certs

04/04/2024  02:50 PM    <DIR>          .
04/04/2024  02:50 PM    <DIR>          ..
04/04/2024  02:50 PM             1,022 csr.pem
04/04/2024  02:46 PM             1,736 key.pem
               2 File(s)          2,758 bytes
               2 Dir(s)  27,419,635,712 bytes free

C:\nodejs-labs\certs>_
```

12. Let us use the **key.pem** and **csr.pem** files to generate your ssl certificate.

Type the following command using openssl

openssl x509 -req -days 365 -in csr.pem -signkey key.pem -out cert.pem

```
C:\WINDOWS\system32\cmd.exe

C:\nodejs-labs\certs>openssl x509 -req -days 365 -in csr.pem -signkey key.pem -out cert.pem
Certificate request self-signature ok
subject=C=US, ST=Michigan, L=Farmington Hills, O=WebAge Solutions, OU=Training
```

13. Check your **certs** directory. Can you see the following files?

cert.pem

csr.pem

key.pem

14. Run the following command in the command window, to view the details of your certificate.

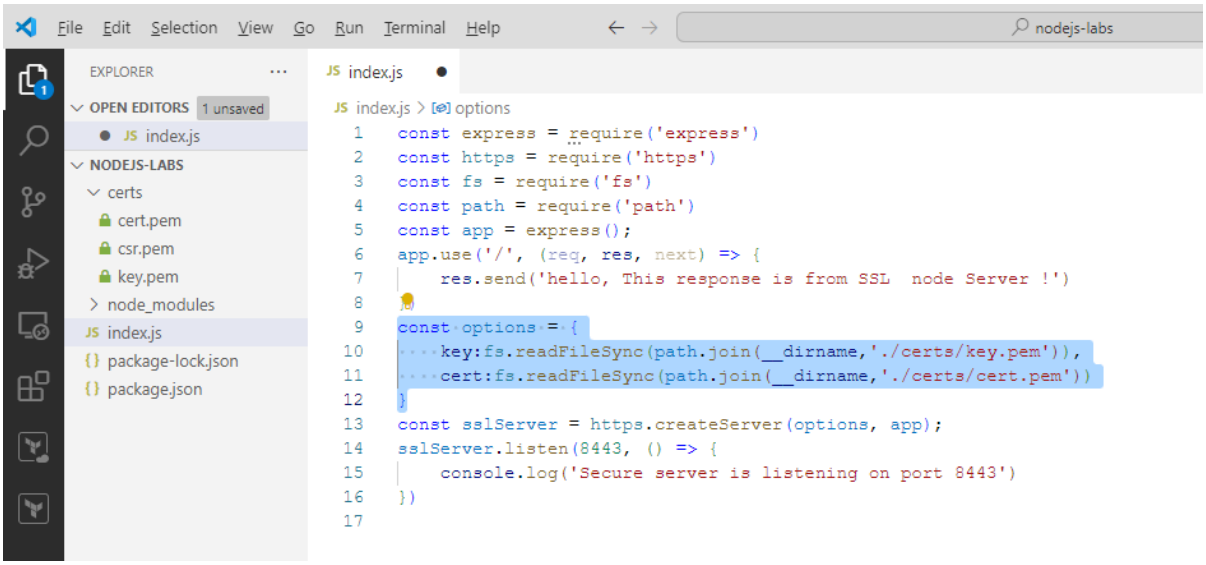
openssl x509 -in cert.pem -noout -text

```
C:\WINDOWS\system32\cmd.exe
C:\nodejs-labs\certs>openssl x509 -in cert.pem -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            67:f3:91:6f:34:a6:94:46:83:9b:96:b0:f3:7b:4d:5b:9c:ad:a8:ee
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=Michigan, L=Farmington Hills, O=WebAge Solutions, OU=Training
        Validity
            Not Before: Apr  4 18:52:49 2024 GMT
            Not After : Apr  4 18:52:49 2025 GMT
        Subject: C=US, ST=Michigan, L=Farmington Hills, O=WebAge Solutions, OU=Training
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b5:73:01:25:50:92:20:67:bf:98:74:0b:65:01:
                    7e:b3:de:80:30:c4:26:41:a6:3a:1a:f6:73:e4:00:
                    83:59:a7:a9:5d:78:cd:cc:9a:0b:d9:33:ef:b4:57:
                    00:78:ef:b3:05:cc:23:b3:c2:96:46:ad:81:be:79:
                    91:04:50:7e:ac:9f:de:32:0e:80:01:6a:60:e1:85:
                    53:d5:ae:93:b7:a8:e0:12:12:f6:11:14:7e:83:ad:
                    9e:59:d3:20:7b:44:e1:74:7f:7a:4f:27:b5:02:bc:
                    04:c0:5b:ee:20:39:ba:b0:db:a6:4b:34:da:ce:bd:
                    ee:5e:a8:f5:79:8b:53:97:7a:f7:04:c3:7d:a2:da:
                    69:1a:40:3a:33:58:0e:78:c3:b6:ac:aa:06:2d:76:
                    53:32:32:63:28:66:0b:7e:ca:fc:3a:3f:0a:ee:72:
                    1b:ec:73:e3:c4:f6:46:d4:e7:04:b5:80:94:cb:93:
                    b1:c0:c5:e9:6c:ee:cc:07:8e:f5:51:9f:96:68:4b:
                    94:6f:37:81:c4:dc:2d:fc:23:9c:79:8d:fa:1b:67:
                    7a:2a:82:61:59:ea:e4:fa:14:8d:a1:26:20:83:38:
                    b5:81:97:cc:f9:e7:da:0e:4a:75:d3:ab:ba:a4:7c:
                    5a:db:ad:82:ce:91:7c:7c:72:2a:b7:8a:fb:23:ee:
                    b6:65
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                9D:FF:65:68:4C:7D:4D:57:FD:43:4E:0C:4C:30:9A:A8:CC:2D:1C:2B
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        28:2f:8c:2d:2b:f1:a5:9b:b8:85:b3:6f:c3:e8:96:4d:df:c8:
        25:4d:16:49:5e:e7:a7:51:9d:27:cb:3a:f0:5d:be:91:14:ea:
        69:2e:e3:fe:30:47:4f:1c:de:50:1c:8c:50:3c:cd:27:fc:76:
        df:3d:f4:f8:9a:47:21:8a:0e:f3:f0:7a:00:e1:43:84:50:1b:
        ae:d2:39:c2:80:25:40:77:6f:60:fd:15:4c:ce:cf:2f:59:a9:
        73:12:9f:61:cc:c9:bc:7b:9f:92:77:80:d5:96:20:47:ab:00:
        8d:29:f8:03:57:6b:bc:2e:f7:e5:54:48:b5:c8:7f:15:11:7b:
        4a:4f:6d:0f:32:b3:08:a3:c5:0a:d5:02:e2:45:c2:68:8a:00:
        be:d2:a6:68:99:a2:e6:ce:8a:ca:75:ef:ca:93:3b:2d:4a:18:
        75:70:d9:cf:56:b3:88:b5:76:3a:2a:c1:cd:51:10:25:8b:b2:
        34:d2:cf:6a:4c:43:a6:96:42:15:f0:53:26:3b:72:3f:72:85:
        2d:b5:8d:47:6d:74:ae:f0:60:7e:38:e1:07:7f:de:f5:17:f5:
```

15. **How to integrate your certificate and key to express code**?

Add the following code in index.js and save it.

```
const options = {
    key:fs.readFileSync(path.join(__dirname,'./certs/key.pem')),
    cert:fs.readFileSync(path.join(__dirname,'./certs/cert.pem'))
}
```



16. In the command window, cd to C:\nodejs-labs

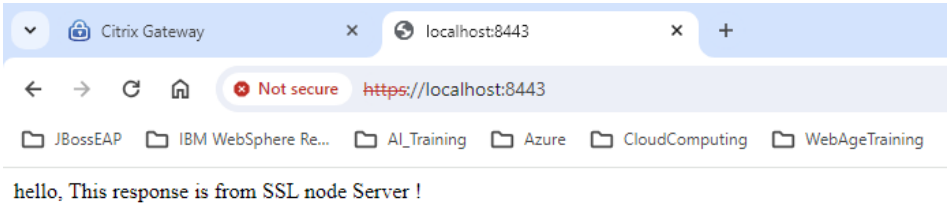
cd C:\nodejs-labs

Run the node server using the command in the command window.

npm start

```
npm

C:\nodejs-labs>npm start

> nodejs-labs@1.0.0 start C:\nodejs-labs
> node index.js

Secure server is listening on port 8443
```

17. Open a browser, go to https://localhost:8443



Notice the Not Secure https **warning ! How can you fix this?**

Lab 04 - OWASP Node Goat Labs

The OWASP NodeGoat project provides an environment to learn how OWASP Top 10 security risks apply to web applications developed using Node.js and how to effectively address them.

https://wiki.owasp.org/index.php/Projects/OWASP_Node_js_Goat_Project

Project source code: https://github.com/OWASP/NodeGoat

Use your lab machine. Let us use the following working directory for this lab.

C:/appsecuritylabs.

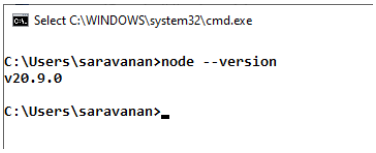
1. Install Node.js software. NodeGoat requires Node v8 or above. Skip this step if your machine is ready with pre-installed node.js software.

   https://nodejs.org/en

   Download and Install node.js software LTS version.

2. Open a command window and test the version of node on your system. Type the following command in the command window.

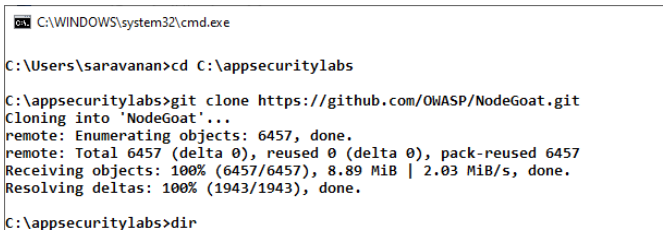
   node --version



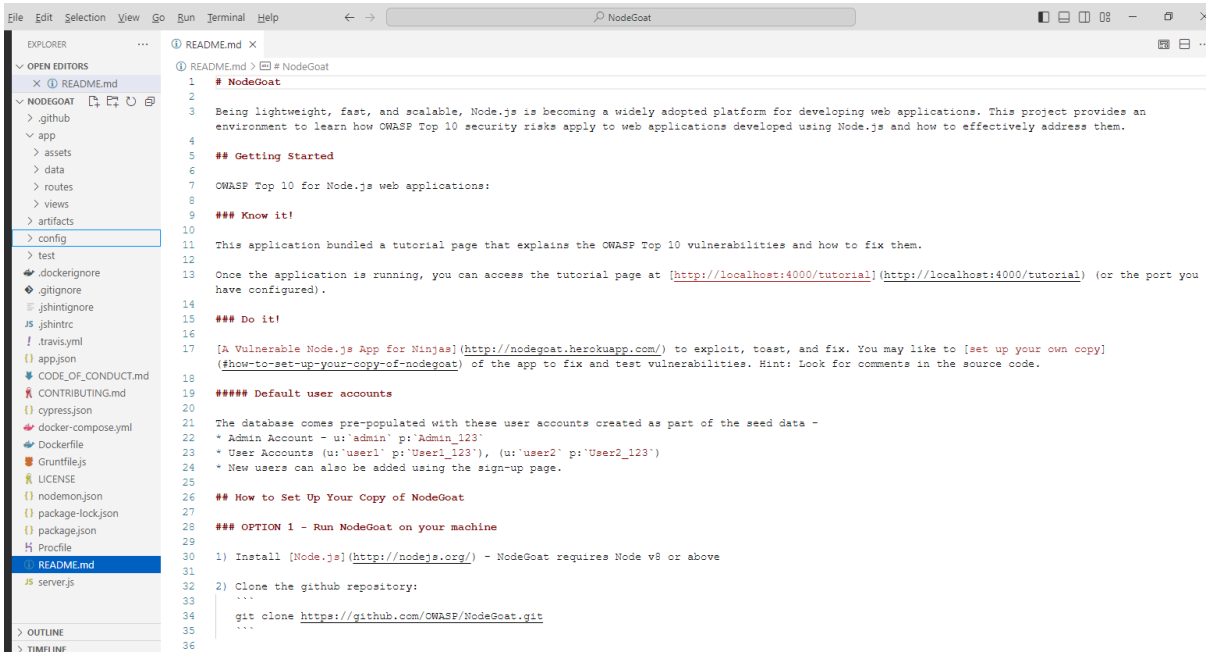
3. Clone the OWASP NodeGoat GitHub repository. In the command window, type the following commands.

   cd C:\appsecuritylabs

   git clone https://github.com/OWASP/NodeGoat.git



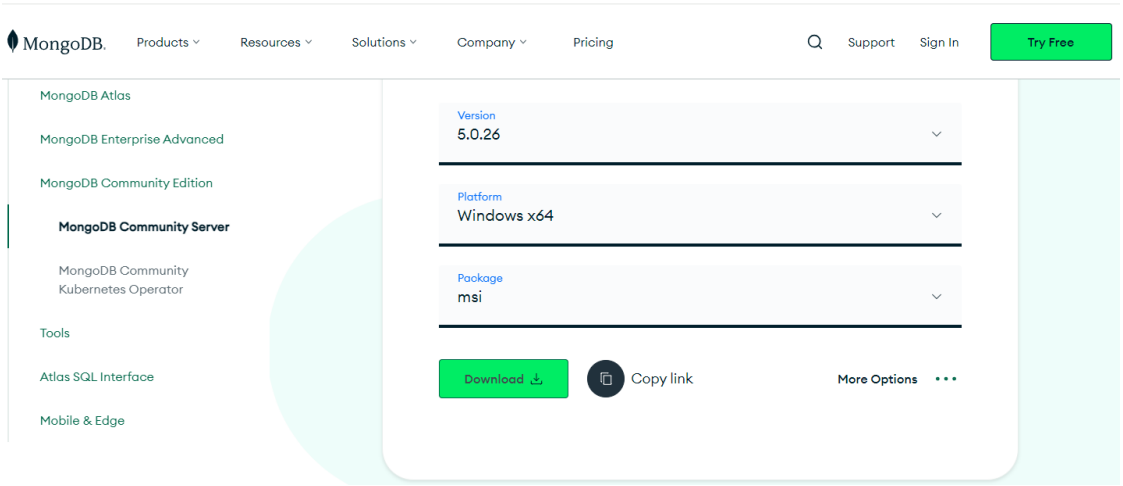
4. Review the application configuration and code. Use Visual Studio Code to open the project and review this project resources.



5. Let us use a **local MongoDB instance** for this application. Download and Install MongoDB version 5.

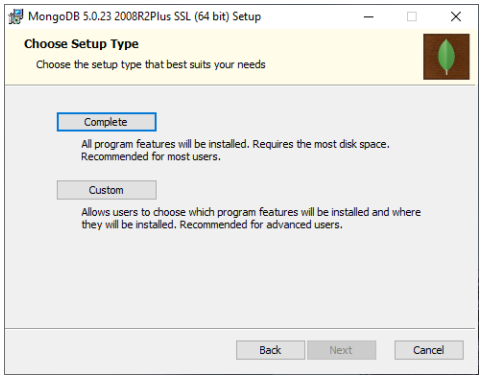
   Go to https://www.mongodb.com/try/download/community

   Select a package **version**: **5.0.26, Platform: Windows x64** and download the **msi** package distribution.

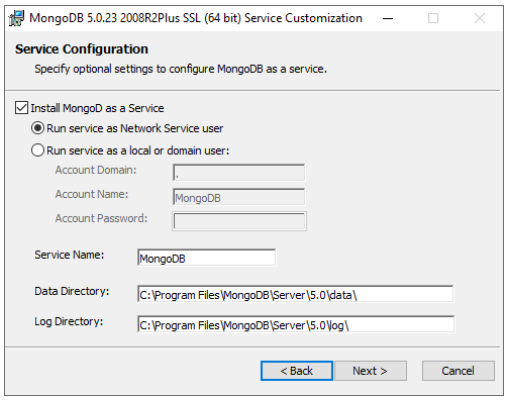


6. Install MongoDB 5.

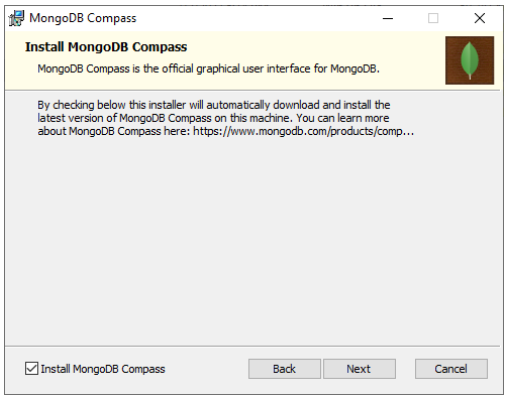
   Choose the setup type: **Complete**



Install MongoD as a service

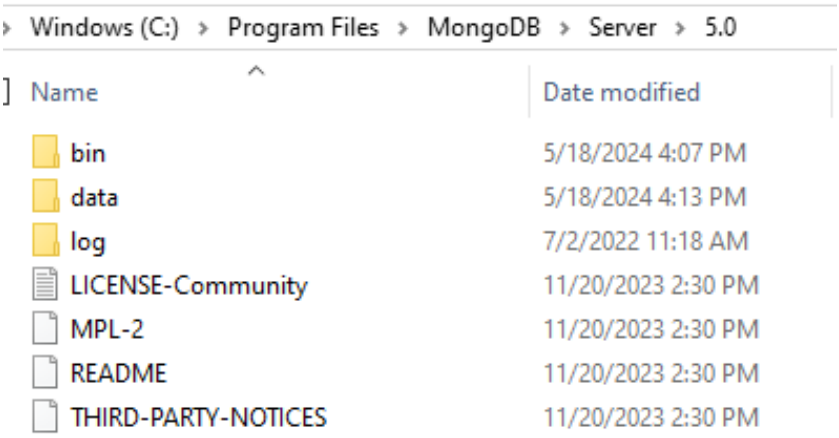


Install MongoDB Compass



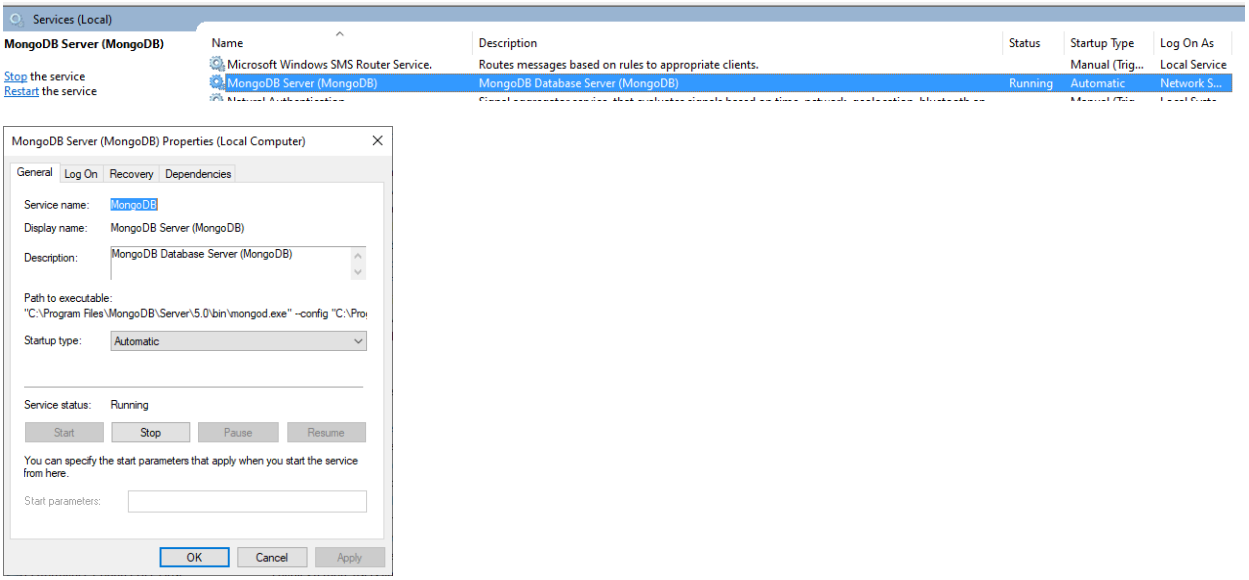
Install the software with all the above settings.

The Mongo DB server software location:

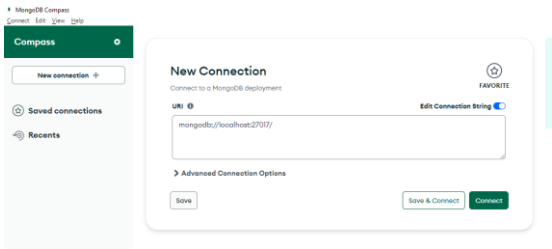


Verify that the Mongodb server is running as a window service.

Open window services and verify that the service is running.



7. Click on the MongoDB Compass shortcut on the desktop to Open MongoDB Compass. Connect to MongoDB server running on localhost:27017



8. Open a command window, change directory to NodeGoat project directory.

   cd C:\appsecuritylabs\NodeGoat

```
C:\Users\saravanan>cd C:\appsecuritylabs\NodeGoat

C:\appsecuritylabs\NodeGoat>_
```

9. Run the npm command to install dependencies. Type the command

   npm install

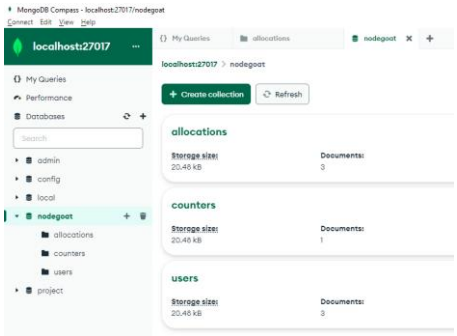
```
npm install

C:\appsecuritylabs\NodeGoat>npm install
[                    ] \ reify:fsevents: sill reify mark deleted [

C:\WINDOWS\system32\cmd.exe

added 962 packages, and audited 1412 packages in 2m

32 packages are looking for funding
  run `npm fund` for details

122 vulnerabilities (4 low, 35 moderate, 62 high, 21 critical)

To address issues that do not require attention, run:
  npm audit fix

To address all issues possible (including breaking changes), run:
  npm audit fix --force

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.

C:\appsecuritylabs\NodeGoat>
```

10. Populate MongoDB with seed data required for the app using the following npm script command. Type the following command.

    npm run db:seed

```
C:\WINDOWS\system32\cmd.exe

C:\appsecuritylabs\NodeGoat>npm run db:seed

> owasp-nodejs-goat@1.3.0 db:seed
> cross-env NODE_ENV=test grunt db-reset

>> Local Npm module "grunt-nodemon" not found. Is it installed?

Running "db-reset" task
>> Current Config:
>> {
>>   port: 4000,
>>   db: 'mongodb://localhost:27017/nodegoat',
>>   cookieSecret: 'session_cookie_secret_key_here',
>>   cryptoKey: 'a_secure_key_for_crypto_here',
>>   cryptoAlgo: 'aes256',
>>   hostName: 'localhost',
>>   environmentalScripts: [],
>>   zapHostName: '192.168.56.20',
>>   zapPort: '8080',
>>   zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
>>   zapApiFeedbackSpeed: 5000
>> }
>> Connected to the database
>> Dropping existing collections
>> Users to insert:
>> {"_id":1,"userName":"admin","firstName":"Node Goat","lastName":"Admin","password":"Admin_123","isAdmin":true}
>> {"_id":2,"userName":"user1","firstName":"John","lastName":"Doe","benefitStartDate":"2030-01-10","password":"User1_123"}
>> {"_id":3,"userName":"user2","firstName":"Will","lastName":"Smith","benefitStartDate":"2025-11-30","password":"User2_123"}
>> countersCol.insert
>> {"result":{"ok":1,"n":1},"ops":[{"_id":"userId","seq":3}],"insertedCount":1,"insertedIds":["userId"]}
>> users.insertMany
>> {"result":{"ok":1,"n":3},"ops":[{"_id":1,"userName":"admin","firstName":"Node Goat","lastName":"Admin","password":"Admin_123","isAdmin":true},{"_id":2,"userName":"user1","firstName":"John","last
Name":"Doe","benefitStartDate":"2030-01-10","password":"User1_123"},{"_id":3,"userName":"user2","firstName":"Will","lastName":"Smith","benefitStartDate":"2025-11-30","password":"User2_123"}],"inser
tedCount":3,"insertedIds":[1,2,3]}
>> Allocations to insert:
>> {"userId":1,"stocks":31,"funds":37,"bonds":32}
>> {"userId":2,"stocks":23,"funds":16,"bonds":61}
>> {"userId":3,"stocks":3,"funds":33,"bonds":64}
>> allocations.insertMany
>> {"result":{"ok":1,"n":3},"ops":[{"userId":1,"stocks":31,"funds":37,"bonds":32,"_id":"66490eaf3ee85948b0582d6a"},{"userId":2,"stocks":23,"funds":16,"bonds":61,"_id":"66490eaf3ee85948b0582d6b"},{"
userId":3,"stocks":3,"funds":33,"bonds":64,"_id":"66490eaf3ee85948b0582d6c"}],"insertedCount":3,"insertedIds":["66490eaf3ee85948b0582d6a","66490eaf3ee85948b0582d6b","66490eaf3ee85948b0582d6c"]}
>> Database reset performed successfully

Done.

C:\appsecuritylabs\NodeGoat>
```

11. Use the MongoDB Compass to verify the database **nodegoat** and view its collections



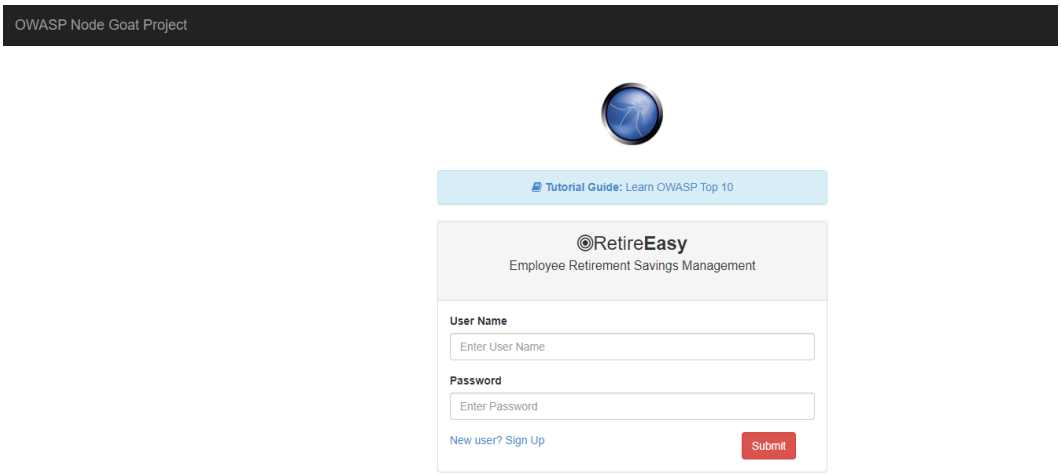
12. In the command window, Start the node goat application using the command: npm start. Type the following command.

npm start

```
npm start

C:\appsecuritylabs\NodeGoat>npm start

> owasp-nodejs-goat@1.3.0 start
> node server.js

Current Config:
{
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostName: 'localhost',
  environmentalScripts: [
    `<script>document.write("<script src='http://" + (location.host || "localhost").split(":")[0] + ":35729/livereload.js'></" + "script>");</script>`
  ],
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
Connected to the database
Express http server listening on port 4000
```

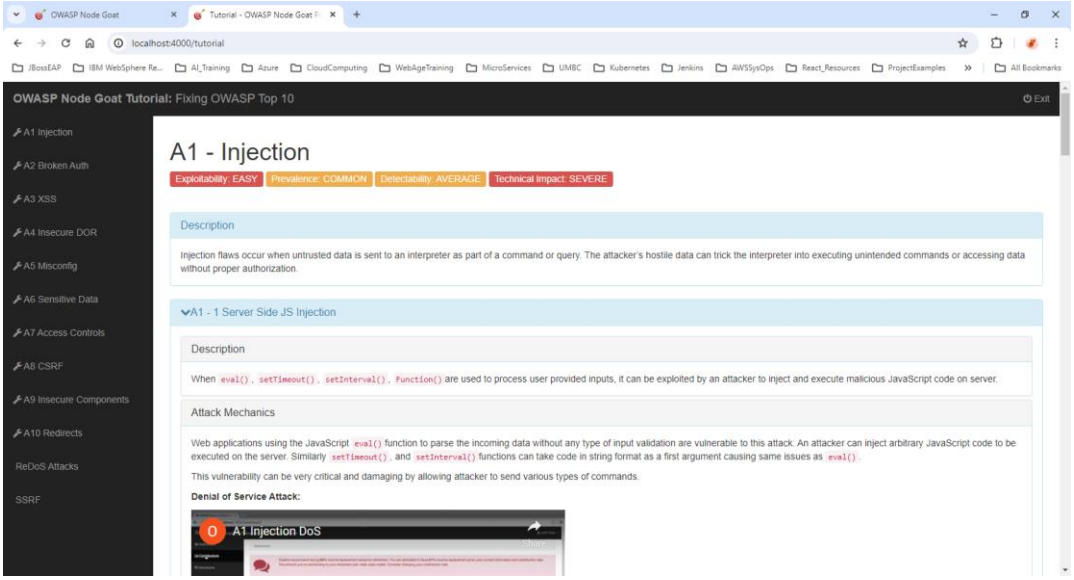
13. Open a browser, go to application page

http://localhost:4000



14. Open another tab/browser window, go to the tutorial page

http://localhost:4000/tutorial



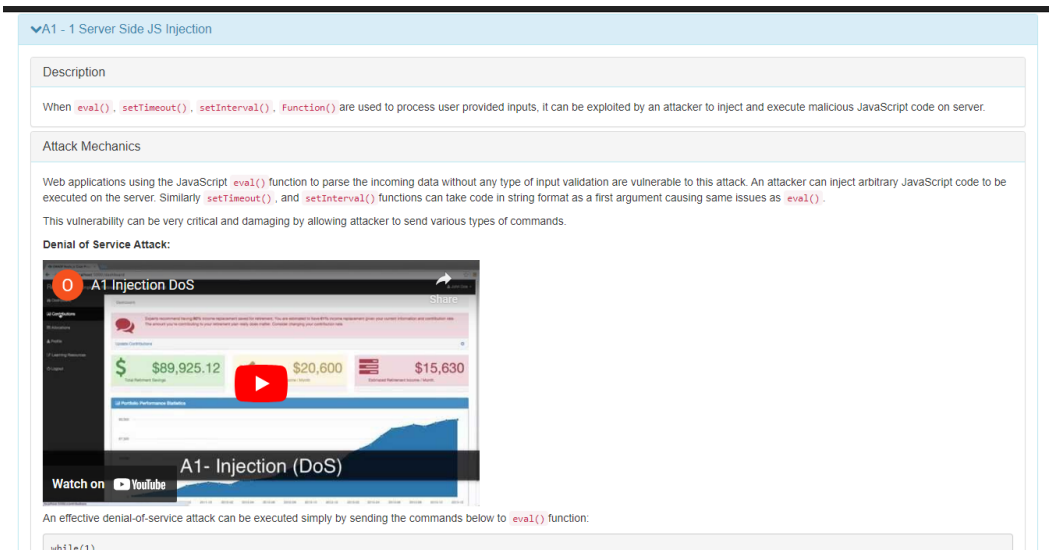
Let us go through all the tutorials and examples code in this node goat application.

A1 – Injection

http://localhost:4000/tutorial/a1

Review the tutorial: A1- Injection and complete the following lab activities.

A1 - 1 Server Side JS Injection



A1 - 2 SQL and NoSQL Injection

A1 - 2 SQL and NoSQL Injection

Description

SQL and NoSQL injections enable an attacker to inject code into the query that would be executed by the database. These flaws are introduced when software developers create dynamic database queries that include user supplied input.

Attack Mechanics

Both SQL and NoSQL databases are vulnerable to injection attack. Here is an example of equivalent attack in both cases, where attacker manages to retrieve admin user's record without knowing password:

1. SQL Injection

Lets consider an example SQL statement used to authenticate the user with username and password

```
SELECT * FROM accounts WHERE username = '$username' AND password = '$password'
```

If this statement is not prepared or properly handled when constructed, an attacker may be able to supply `admin' --` in the username field to access the admin user's account bypassing the condition that checks for the password. The resultant SQL query would looks like:

```
SELECT * FROM accounts WHERE username = 'admin' -- AND password = ''
```

2. NoSQL Injection

The equivalent of above query for NoSQL MongoDB database is:

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

While here we are no longer dealing with query language, an attacker can still achieve the same results as SQL injection by supplying JSON input object as below:

```
{
```

A1 - 3 Log Injection

A1 - 3 Log Injection

Description

Log injection vulnerabilities enable an attacker to forge and tamper with an application's logs.

Attack Mechanics

An attacker may craft a malicious request that may deliberately fail, which the application will log, and when attacker's user input is unsanitized, the payload is sent as-is to the logging facility. Vulnerabilities may vary depending on the logging facility:

1. Log Forging (CRLF)

Lets consider an example where an application logs a failed attempt to login to the system. A very common example for this is as follows:

```
var userName = req.body.userName;
console.log('Error: attempt to login with invalid user: ', userName);
```

When user input is unsanitized and the output mechanism is an ordinary terminal stdout facility then the application will be vulnerable to CRLF injection, where an attacker can create a malicious payload as follows:

```
curl http://localhost:4000/login -X POST --data 'userName=vyva%0aError: alex moldovan failed $1,000,000 transaction&password=Admin_123&_csrf='
```

Where the `userName` parameter is encoding in the request the LF symbol which will result in a new line to begin. Resulting log output will look as follows:

```
Error: attempt to login with invalid user:  vyva
Error: alex moldovan failed $1,000,000 transaction
```

2. Log Injection Escalation

A2-Broken Authentication and Session Management

http://localhost:4000/tutorial/a2

Review the tutorial: A2-Broken Authentication and Session Management and complete the following lab activities.

A2 - 1 Session Management

A2 - 1 Session Management

Description

Session management is a critical piece of application security. It is broader risk, and requires developers take care of protecting session id, user credential secure storage, session duration, and protecting critical session data in transit.

Attack Mechanics

Scenario #1: Application timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #2: Attacker acts as a man-in-middle and acquires user's session id from network traffic. Then uses this authenticated session id to connect to application without needing to enter user name and password.

Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not properly hashed, exposing every users' password to the attacker.

How Do I Prevent It?

Session management related security issues can be prevented by taking these measures:

- User authentication credentials should be protected when stored using hashing or encryption.
- Session IDs should not be exposed in the URL (e.g., URL rewriting).
- Session IDs should timeout. User sessions or authentication tokens should get properly invalidated during logout.
- Session IDs should be recreated after successful login.
- Passwords, session IDs, and other credentials should not be sent over unencrypted connections.

Source Code Examples

In the insecure demo app, following issues exists:

A2 - 2 Password Guessing Attacks

A2 - 2 Password Guessing Attacks

Description

Implementing a robust minimum password criteria (minimum length and complexity) can make it difficult for attacker to guess password.

Attack Mechanics

The attacker can exploit this vulnerability by brute force password guessing, more likely using tools that generate random passwords.

How Do I Prevent It?

Password length

Minimum passwords length should be at least eight (8) characters long. Combining this length with complexity makes a password difficult to guess and/or brute force.

Password complexity

Password characters should be a combination of alphanumeric characters. Alphanumeric characters consist of letters, numbers, punctuation marks, mathematical and other conventional symbols.

Username/Password Enumeration

Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code

Additional Measures

- For additional protection against brute forcing, enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed.
- Only send non-temporary passwords over an encrypted connection or as encrypted data, such as in an encrypted email. Temporary passwords associated with email resets may be an exception. Enforce the changing of temporary passwords on the next use. Temporary passwords and links should have a short expiration time.

Source Code Example

A3-Cross-Site Scripting (XSS)

http://localhost:4000/tutorial/a3

Review the tutorial: A3-Cross-Site Scripting (XSS) and complete the XSS lab activities.

A3-Cross-Site Scripting (XSS)

Exploitability: AVERAGE | Prevalence: VERY WIDESPREAD | Detectability: EASY | Technical Impact: MODERATE

Description

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victims' browser, which can access any cookies, session tokens, or other sensitive information retained by the browser, or redirect user to malicious sites.

Attack Mechanics

There are two types of XSS flaws:

1. Reflected XSS: The malicious data is echoed back by the server in an immediate response to an HTTP request from the victim.
2. Stored XSS: The malicious data is stored on the server or on browser (using HTML5 local storage, for example), and later gets embedded in HTML page provided to the victim.

Each of reflected and stored XSS can occur on the server or on the client (which is also known as DOM based XSS), depending on when the malicious data gets injected in HTML markup.

How Do I Prevent It?

1. **Input validation and sanitization:** Input validation and data sanitization are the first line of defense against untrusted data. Apply white list validation wherever possible.
2. **Output encoding for correct context:** When a browser is rendering HTML and any other associated content like CSS, javascript etc., it follows different rendering rules for each context. Hence *Context-sensitive output encoding* is absolutely critical for mitigating risk of XSS.

Here are the details about applying correct encoding in each context:

Context	Code Sample	Encoding Type
HTML Entity	UNTRUSTED DATA	Convert & to &

Source Code Example

The demo web application is vulnerable to stored XSS attack on profiles form. On form submit, the first and last name field values are submitted to the server, and without any validation get saved in database. The values are then sent back to the browser without proper escaping to be shown at the top right menu.

Two measures can be taken to mitigate XSS risk:

1. In `server.js`, enable the HTML Encoding using template engine's auto escape flag.

```
swig.init({
    root: __dirname + "/app/views",
    autoescape: true //default value
});
```

2. Set HTTPOnly flag for session cookie while configuring the express session

```
// Enable session management using express middleware
app.use(express.session({
    secret: "s3Cur3",
```

A4-Insecure Direct Object References

http://localhost:4000/tutorial/a4

Review the tutorial: A4-Insecure Direct Object References and complete the Insecure DOR lab activity

A4-Insecure Direct Object References

Exploitability: EASY | Prevalence: COMMON | Detectability: EASY | Technical Impact: MODERATE

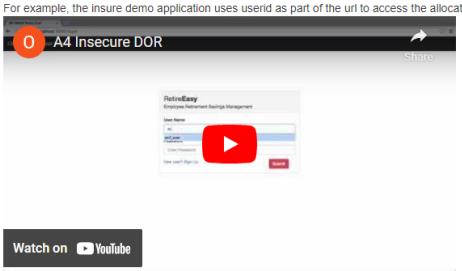
Description

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

Attack Mechanics

If an applications uses the actual name or key of an object when generating web pages, and doesn't verify if the user is authorized for the target object, this can result in an insecure direct object reference flaw. An attacker can exploit such flaws by manipulating parameter values. Unless object references are unpredictable, it is easy for an attacker to access all available data of that type.

For example, the insure demo application uses userid as part of the url to access the allocations (/allocations/{id}). An attacker can manipulate id value and access other user's allocation information.



How Do I Prevent It?

1. **Check access:** Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.
2. **Use per user or session indirect object references:** Instead of exposing actual database keys as part of the access links, use temporary per-user indirect reference. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 or unique random numbers to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server.
3. **Testing and code analysis:** Testers can easily manipulate parameter values to detect such flaws. In addition, code analysis can quickly show whether authorization is properly verified.

Source Code Example

In `routes/allocations.js`, the insecure application takes user id from url to fetch the allocations.

```
var userId = req.params.userId;
allocationsDAO.getByUserId(userId, function(error, allocations) {

    if (error) return next(error);

    return res.render("allocations", allocations);
});
```

A safer alternative is to always retrieve allocations for logged in user (using `req.session.userId`)instead of taking it from url.

A5-Security Misconfiguration

http://localhost:4000/tutorial/a5

Review the tutorial: A5-Security Misconfiguration and complete the Misconfiguration lab activity.

A5-Security Misconfiguration

Exploitability: EASY | Prevalence: COMMON | Detectability: EASY | Technical Impact: MODERATE

Description

This vulnerability allows an attacker to accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.

Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code.

Developers and system administrators need to work together to ensure that the entire stack is configured properly.

Attack Mechanics

This vulnerability encompasses a broad category of attacks, but here are some ways attacker can exploit it:

1. If application server is configured to run as root, an attacker can run malicious scripts (by exploiting eval family functions) or start new child processes on server
2. Read, write, delete files on file system. Create and run binary files
3. If the server is misconfigured to leak internal implementation details via cookie names or HTTP response headers, then attacker can use this information towards building site's risk profile and finding vulnerabilities
4. If request body size is not limited, an attacker can upload large size of input payload, causing server to run out of memory, or make processor and event loop busy.

How Do I Prevent It?

Here are some node.js and express specific configuration measures:

- Use latest stable version of node.js and express (or other web framework you are using). Keep a watch on published vulnerabilities of these. The vulnerabilities for node.js and express.js can be found here and here, respectively.
- Do not run application with root privileges. It may seem necessary to run as root user to access privileged ports such as 80. However, this can achieved either by starting server as root and then downgrading the non-privileged user after listening on port 80 is established, or using a separate proxy, or using port mapping.

Source Code Example

The default HTTP header x-powered-by can reveal implementation details to an attacker. It can be taken out by including this code in `server.js`

```
app.disable("x-powered-by");
```

The default session cookie name for express sessions can be changed by setting key attribute while creating express session.

```
app.use(express.session({
    secret: config.cookieSecret,
    key: "sessionId",
    cookie: {
        httpOnly: true,
        secure: true
    }
}));
```

The default session cookie name for express sessions can be changed by setting key attribute while creating express session.

```
app.use(express.session({
    secret: config.cookieSecret,
    key: "sessionId",
    cookie: {
        httpOnly: true,
        secure: true
    }
}));
```

The security related HTTP Headers can be added using helmet middleware as below

```
// Prevent opening page in frame or iframe to protect from clickjacking
app.disable("x-powered-by");

// Prevent opening page in frame or iframe to protect from clickjacking
app.use(helmet.xframe());

// Prevents browser from caching and storing page
app.use(helmet.noCache());

// Allow loading resources only from white-listed domains
app.use(helmet.csp());

// Allow communication only on HTTPS
app.use(helmet.hsts());

// Forces browser to only use the Content-Type set in the response header instead of sniffing or guessing it
app.use(nosniff());
```

A6-Sensitive Data Exposure

http://localhost:4000/tutorial/a6

Review the tutorial: A6-Sensitive Data Exposure and complete the Sensitive Data lab activity.

A6-Sensitive Data Exposure

Exploitability: DIFFICULT | Prevalence: COMMON | Detectability: AVERAGE | Technical Impact: SEVERE

Description

This vulnerability allows an attacker to access sensitive data such as credit cards, tax IDs, authentication credentials, etc to conduct credit card fraud, identity theft, or other crimes. Losing such data can cause severe business impact and damage to the reputation. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

Attack Mechanics

If a site doesn't use SSL/TLS for all authenticated pages, an attacker can monitor network traffic (such as on open wireless network), and steals user's session cookie. Attacker can then replay this cookie and hijacks the user's session, accessing the user's private data.

If an attacker gets access the application database, he or she can steal the sensitive information not encrypted, or encrypted with weak encryption algorithm

How Do I Prevent It?

- Use Secure HTTPS network protocol
- Encrypt all sensitive data at rest and in transit
- Don't store sensitive data unnecessarily. Discard it as soon as possible.
- Ensure strong standard algorithms and strong keys are used, and proper key management is in place.
- Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data.

Source Code Example

1.The insecure demo application uses HTTP connection to communicate with server. A secure HTTPS sever can be set using https module. This would need a private key and certificate. Here are source code examples from `/server.js`

```
// Load keys for establishing secure HTTPS connection
var fs = require("fs");
var https = require("https");
var path = require("path");
var httpsOptions = {
    key: fs.readFileSync(path.resolve(__dirname, "./app/cert/key.pem")),
    cert: fs.readFileSync(path.resolve(__dirname, "./app/cert/cert.pem"))
};
```

2. Start secure HTTPS sever

```
// Start secure HTTPS server
https.createServer(httpsOptions, app).listen(config.port, function() {
    console.log("Express https server listening on port " + config.port);
});
```

3. The insecure demo application stores users personal sensitive information in plain text. To fix it, The `data/profile-dao.js` can be modified to use crypto module to encrypt and decrypt sensitive information as below:

```
// Include crypto module
var crypto = require("crypto");

//Set keys config object
var config = {
    cryptoKey: "a_secure_key_for_crypto_here",
    cryptoAlgo: "aes256", // or other secure encryption algo here
    iv: ""
};
```

```
//Set keys config object
var config = {
    cryptoKey: "a_secure_key_for_crypto_here",
    cryptoAlgo: "aes256", // or other secure encryption algo here
    iv: ""
};

// Helper method create initialization vector
// By default the initialization vector is not secure enough, so we create our own
var createIV = function() {
    // create a random salt for the PBKDF2 function - 16 bytes is the minimum length according to NIST
    var salt = crypto.randomBytes(16);
    return crypto.pbkdf2Sync(config.cryptoKey, salt, 100000, 512, "sha512");
};

// Helper methods to encryt / decrypt
var encrypt = function(toEncrypt) {
    config.iv = createIV();
    var cipher = crypto.createCipheriv(config.cryptoAlgo, config.cryptoKey, config.iv);
    return cipher.update(toEncrypt, "utf8", "hex") + cipher.final("hex");
};

var decrypt = function(toDecrypt) {
    var decipher = crypto.createDecipheriv(config.cryptoAlgo, config.cryptoKey, config.iv);
    return decipher.update(toDecrypt, "hex", "utf8") + decipher.final("utf8");
};

// Encrypt values before saving in database
user.ssn = encrypt(ssn);
user.dob = encrypt(dob);

// Decrypt values to show on view
user.ssn = decrypt(user.ssn);
user.dob = decrypt(user.dob);
```

A7-Missing Function Level Access Control

http://localhost:4000/tutorial/a7

Review the tutorial: A7-Missing Function Level Access Control and complete the Access Controls lab activity.

A7-Missing Function Level Access Control

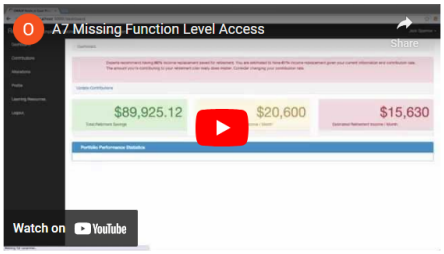
Exploitability: EASY | Prevalence: COMMON | Detectability: AVERAGE | Technical Impact: MODERATE

Description

Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed.

Attack Mechanics

If requests are not verified for access rights on server, attackers can forge requests in order to access functionality without proper authorization.



In the insecure demo application, this vulnerability exists in benefits module, which allows changing benefit start date for employees. The link to the benefits module is visible only to the admin user (user: admin, password: Admin_123). However, an attacker can access this module simply by logging in as any non-admin user and accessing benefits url directly.

How Do I Prevent It?

Most web applications don't display links and buttons to unauthorized functions, but this "presentation layer access control" doesn't actually provide protection. You must also implement checks in the controller or business logic.

Source Code Examples

In vulnerable application, there is no authorization check for benefits related routes in `routes/index.js`

```
// Benefits Page
app.get("/benefits", isLoggedIn, benefitsHandler.displayBenefits);
app.post("/benefits", isLoggedIn, benefitsHandler.updateBenefits);
```

This can be fixed by adding a middleware to verify user's role:

```
// Benefits Page
app.get("/benefits", isLoggedIn, isAdmin, benefitsHandler.displayBenefits);
app.post("/benefits", isLoggedIn, isAdmin, benefitsHandler.updateBenefits);
```

To implement `isAdmin` middleware, check if isAdmin flag is set for the logged in user in database.
For example, here is middleware function that can be added to `routes\session.js`:

```
this.isAdminUserMiddleware = function(req, res, next) {
    if (req.session.userId) {
        userDAO.getUserById(req.session.userId, function(err, user) {
            if(user && user.isAdmin) {
                next();
            } else {
                return res.redirect("/login");
            }
        });
```

A8-Cross-Site Request Forgery (CSRF)

http://localhost:4000/tutorial/a8

Review the tutorial: A8-Cross-Site Request Forgery (CSRF) and complete the CSRF lab activity.

A8-Cross-Site Request Forgery (CSRF)

Exploitability: AVERAGE | Prevalence: COMMON | Detectability: EASY | Technical Impact: MODERATE

Description

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests that the vulnerable application processes as legitimate requests from the victim.

Attack Mechanics

As browsers automatically send credentials like session cookies with HTTP requests to the server where cookies were received from, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones.

For example, CSRF vulnerability can be exploited on profile form on the insecure demo application.

To exploit it:

1. An attacker would need to host a forged form like below on a malicious sever.

```
<html lang="en">
<head></head>
    <body>
            <form method="POST" action="http://TARGET_APP_URL_HERE/profile">
                    <h1> You are about to win a brand new iPhone!</h1>
                    <h2> Click on the win button to claim it...</h2>
                    <input type="hidden" name="bankAcc" value="9999999"/>
                    <input type="hidden" name="bankRouting" value="88888888"/>
                            <input type="submit" value="Win !!!"/>
            </Form>
    </body>
</html>
```

   Note: A sample app containing form for CSRF attack on NodeGoat app is available here.
2. Next, attacker would need to manage opening the form on logged in victim's browser and attract user to submit it. When user submits this form, it results in victim user's browser sending a malicious request to vulnerable server, causing CSRF attack.

How Do I Prevent It?

Express csrf middleware provides a very effective way to deal with csrf attack. By default this middleware generates a token named "_csrf" which should be added to requests which mutate state (PUT, POST, DELETE), within a hidden form field, or query-string, or header fields.

If using method-override middleware, it is very important that it is used before any middleware that needs to know the method of the request, including CSRF middleware. Otherwise an attacker can use non-state mutating methods (such as GET) to bypass the CSRF middleware checks, and use method override header to convert request to desired method.

When form is submitted, the middleware checks for existence of token and validates it by matching to the generated token for the response-request pair. If tokens do not match, it rejects the request. Thus making it really hard for an attacker to exploit CSRF.

Source Code Example

The `server.js` includes the express CSRF middleware after session is initialized. Then creates a custom middleware to generate new token using `req.csrfToken();` and exposes it to view by setting it in `res.locals`

Source Code Example

The `server.js` includes the express CSRF middleware after session is initialized. Then creates a custom middleware to generate new token using `req.csrfToken();` and exposes it to view by setting it in `res.locals`

```
//Enable Express csrf protection
app.use(express.csrf());

app.use(function(req, res, next) {
    res.locals.csrftoken = req.csrfToken();
    next();
});
```

Next, this token can be included in a hidden form field in `views/profile.html` as below.

```
<input type="hidden" name="_csrf" value="{{ csrftoken } }">
```

A9-Using Components with Known Vulnerabilities

http://localhost:4000/tutorial/a9

Review the tutorial: A9-Using Components with Known Vulnerabilities and complete the Insecure Components lab activity.

A9-Using Components with Known Vulnerabilities

Exploitability: AVERAGE | Prevalence: WIDESPREAD | Detectability: DIFFICULT | Technical Impact: MODERATE

Description

Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

Using insecure npm packages can lead to this vulnerability. Some projects today help test and alert on insecure dependencies:

1. npm audit is a vulnerability scanner built into the npm CLI (version 6 or later)
2. Dependabot security updates can automatically make GitHub pull requests to update vulnerable dependencies
3. Snyk.io is a Node.js CLI tool and Platform to scan and detect vulnerable packages

The tools above make use of vulnerability lists, which can also be viewed directly or searched here:

1. NPM Security Advisories
2. GitHub Advisory Database
3. Snyk Vulnerability DB

There are some other tools that can detect and update outdated packages:

1. npm outdated and yarn outdated are both command line ways to show possibly out of date dependencies
2. Dependabot version updates can automatically make GitHub pull requests to update outdated dependencies
3. David DM gets you an overview of your project dependencies, the version you use and the latest available, so you can quickly see what's drifting
4. npm-check Check for outdated, incorrect, and unused dependencies

Attack Mechanics

The npm packages are essential part of our node application. These packages could either accidentally or maliciously contain insecure code. Through insecure packages an attacker can:

- Create and run scripts at different stages during installation or usage of the package.
- Read, write, update, delete files on system
- Write and execute binary files
- Collect sensitive data send it remotely

How Do I Prevent It?

These are few measures we can take to protect against malicious npm packages

- Do not run application with root privileges
- Prefer packages that include static code analysis. Check JSHint/JSLint the configuration to know what rules code abide by
- Prefer packages that contain comprehensive unit tests and review tests for the functions our application uses
- Review code for any unexpected file or database access
- Research about how popular the package is, what other packages use it, if any other packages are written by the author, etc
- Lock version of packages used
- Watch Github repositories for notifications. This will inform us if any vulnerabilities are discovered in the package in future

Insecure Dependencies Example

Description

The demo web application is using a popular library called Marked which is a Markdown parser in JavaScript and provides an easy way to integrate markdown syntax for rich text to a website, replacing the need to build WYSIWYG editors.

This library has reached almost millions of downloads a month, making it quite popular with also **11,000** stars on GitHub at one point.

Attack Mechanics

In this demo project we are using an insecure version of the Marked library that is vulnerable to XSS exploits.

Scenario: A form on a page allows free text user input which is later parsed using the Marked library to markdown format and compiled in a dedicated view to show the rich text version. An attacker can exploit this form to insert malicious XSS strings which the Markdown library isn't filtering very well, resulting in an XSS attack.

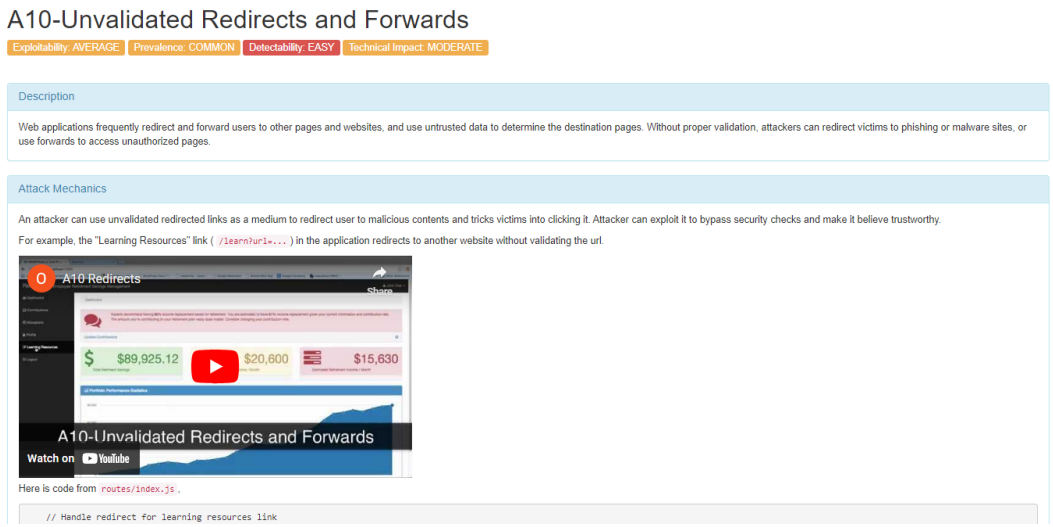
Try sending one of the following markdown syntax strings in the Memos section to exploit it and see which one succeeds:

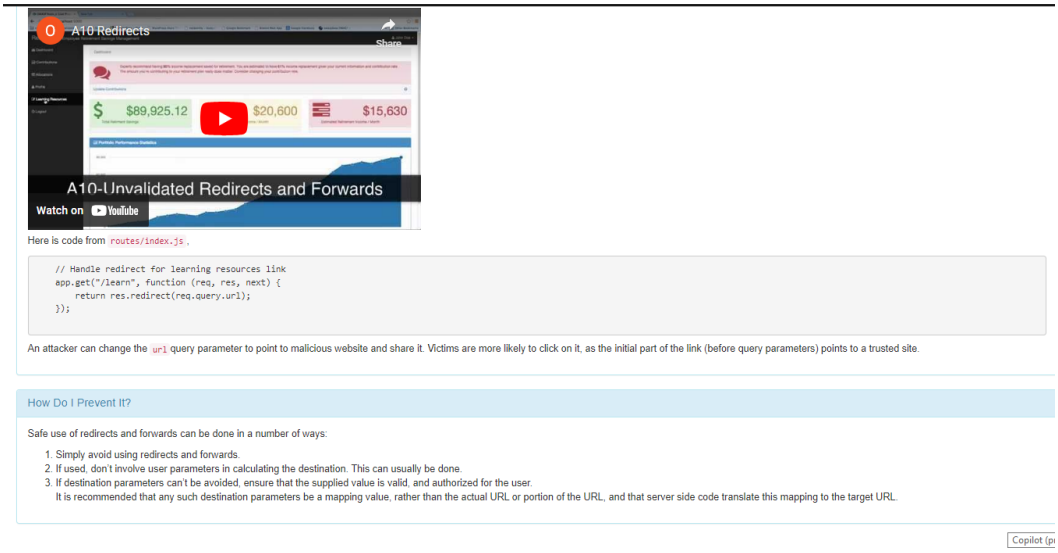
1. `[Nice try](javascript:alert(1))`

A10-Unvalidated Redirects and Forwards

http://localhost:4000/tutorial/a10

Review the tutorial: A10-Unvalidated Redirects and Forwards and complete the Redirects lab activity.





ReDoS Regular Expressions DoS

http://localhost:4000/tutorial/redos

ReDoS Regular Expressions DoS

Description

The Regular expression Denial of Service (ReDoS) is a Denial of Service attack, that exploits the fact that most Regular Expression implementations may reach extreme situations that cause them to work very slowly (exponentially related to input size). An attacker can then cause a program using a Regular Expression to enter these extreme situations and then hang for a very long time.

Attack Mechanics

When untrusted data input is executed on a regex pattern, it may exploit vulnerable patterns into running long calculations to match for a given string. For Node.js this is extremely important due to the single-threaded event-loop architecture which means that the main Node.js process is blocked from serving any other requests.

How Do I Prevent It?

1. Avoid writing your own regular expressions
2. Use Node.js's validator.js package to validate expected data format instead of writing your own regular expressions
3. As a last resort of writing your own regex patterns you can utilize Node.js's safe-regex package which allows detecting if a regex is prone to catastrophic backtracking, and also allows to configure threshold for maximum repetitions.

Source Code Example

Even simple regex patterns are vulnerable to ReDoS. The NodeGoat project uses the following source code to validate text format from the user based on a regex pattern:

```
// Allow only numbers with a suffix of #, for example: 'XXXXXX#'
var regexPattern = /([0-9]+)+\#/;
var testComplyWithRequirements = regexPattern.test(bankRouting)
```

If a long enough input is provided it will stall the Node.js process and render it useless (in the background the Node.js process will take 100% cpu until stopped or the regex yields a result (true or false)). Try to input the following string in the Bank Routing number in the Profile form:

```
91762612117612121123123123123121
```

Server-Side Request Forgery (SSRF)

http://localhost:4000/tutorial/ssrf

Server-Side Request Forgery (SSRF)

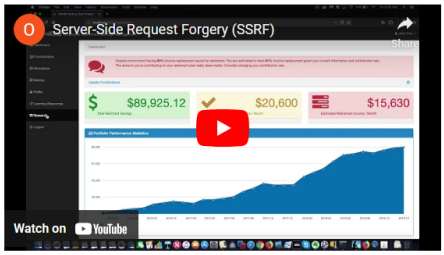
Description

In an SSRF attack, the attacker can abuse functionality on the server to read or update internal resources. The attacker can supply or modify a URL that the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like HTTP-enabled databases or perform HTTP POST requests towards internal services which are not intended to be exposed.

Attack Mechanics

An attacker can use an SSRF vulnerability as a way to gather information about the server and the local network.

For example, on the "Research" page (`/research`) in the application, a user submits a stock symbol. The stock symbol is concatenated to a Yahoo URL and the server fetches the response and displays the page.



Here is a code snippet from `routes/research.js`.

```
// If a stock symbol has been submitted, concatenate the symbol to the URL and return the HTTP Response
if (req.query.symbol) {
    var url = req.query.url+req.query.symbol;
    needle.get(url, function(error, newResponse) { ... }
```

An attacker can change the `url` and `symbol` parameters to point to an attacker-controlled website to interact with the server.

How Do I Prevent It?

To prevent SSRF vulnerabilities in web applications, it is recommended to adhere to the following guidelines:

1. Use a whitelist of allowed domains, resources and protocols from where the web server can fetch resources.
2. Any input accepted from the user should be validated and rejected if it does not match the positive specification expected.
3. If possible, do not accept user input in functions that control where the web server can fetch resources.