

Final Project - NodeGoat

By Theodore Corrello and Mia Weber

A1 Injection

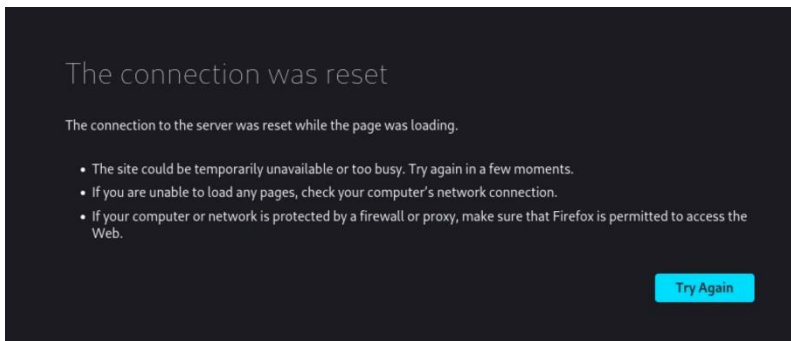
Injection can happen when a user's input is processed, and it is not sanitized properly. Below are several examples in NodeGoat of potential injection attacks.

1. *Sever Side JS Injection*

In the "Contributions" tab, the user inputs have a potential to be exploited in killing the application.

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

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



By putting in `process.kill(process.pid)` into the Employee Pre-Tax text input, the node application then crashes due to it shutting itself down.

```
// Insecure use of eval() to parse inputs
const preTax = eval(req.body.preTax);
const afterTax = eval(req.body.afterTax);
const roth = eval(req.body.roth);
```

```
const preTax = parseInt(req.body.preTax);
const afterTax = parseInt(req.body.afterTax);
const roth = parseInt(req.body.roth);
```

This vulnerability is caused by using the eval function to parse the inputs, which can cause remove code execution. The intent of the original code was to parse the user's input as an integer. While eval is not secure, the built-in function parseInt will do exactly what the application needs without arbitrary code execution.

Invalid contribution percentages

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

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

Submit

After fixing the code to use parseInt, the website now properly states process.kill(process.pid) is an invalid number.

2. SQL and NoSQL Injection

In the “Allocations” tab, the user can filter assets based on stock performances. Just like the previous vulnerability, the input box here is vulnerable.

Filter Assets based on Stock Performance

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

Submit

Asset Allocations for John Doe

Domestic Stocks : 16 %

Funds: 2 %

Bonds: 82 %

Asset Allocations for John Doe
Domestic Stocks : 16 %
Funds: 2 %
Bonds: 82 %

Asset Allocations for Node Goat Admin
Domestic Stocks : 9 %
Funds: 40 %
Bonds: 51 %

Asset Allocations for Will Smith
Domestic Stocks : 28 %
Funds: 18 %
Bonds: 54 %

By typing in `1'; return 1 == '1`, the website parses it as true which means that all the records in the database are returned regardless of who is logged in.

Filter Assets based on Stock Performance

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

Submit

An infinite loop can also be inserted into the query which can cause a Denial of Service (DOS) attack.

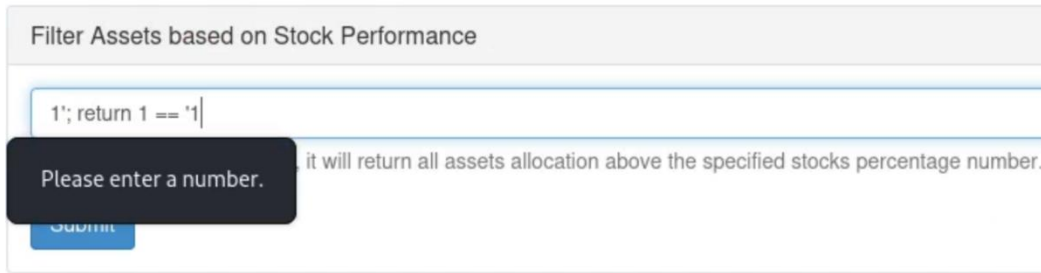
```
<input type="number" min="0" max="99" class="form-control" placeholder="Stocks Threshold" name="threshold" />
<!--<input type="text" class="form-control" placeholder="Stocks Threshold" name="threshold" />-->
```

```
const parsedThreshold = parseInt(threshold, 10);

if (parsedThreshold >= 0 && parsedThreshold <= 99) {
  return {$where: `this.userId == ${parsedUserId} && this.stocks > ${parsedThreshold}`};
}
throw `The user supplied threshold: ${parsedThreshold} was not valid.`;

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

To fix this, all that needs to be done is add validation on the HTML and server JavaScript side. The previous input on the HTML input element considered it text, which should be a number. Changing the type to number along with a min and max added helps prevent invalid input. For JavaScript, the threshold is never parsed as an integer with `parseInt`. The fix is to properly parse the input as an integer and then check the range. If the range is not met, then throw an error.



Filter Assets based on Stock Performance

1"; return 1 == '1|

Please enter a number.

Submit

it will return all assets allocation above the specified stocks percentage number.

Once again, the website sees that the exploit is not a number nor valid and stops the text from being processed.

A2 –1 Session Management

In this attack, an attacker uses leaks or flaws in the authentication or session management functions to impersonate other users.

1. Session Management

Passwords are stored in plain text in the database. User authentication credentials should be protected when stored using hashing or encryption. This is the original logic to store the password in user-dao.js

```
19 // Create user document
20 const user = {
21   userName,
22   firstName,
23   lastName,
24   benefitStartDate: this.getRandomFutureDate(),
25   password //received from request param
```

To fix this vulnerability, handle password storage in a safer way by using one way encryption and salt hashing.

```
22 const user = {
23   userName,
24   firstName,
25   lastName,
26   benefitStartDate: this.getRandomFutureDate(),
27   //password, //received from request param
28
29   // Fix for A2-1 - Broken Auth
30   // Stores password in a safer way using one way encryption and salt hashing
31   password: bcrypt.hashSync(password, bcrypt.genSaltSync())
```

Then when the user logs in, the user entered password is converted to hash and compared with the hash in storage. Line 67 in the same file makes this comparison and replaces like 63 as a safer way to authenticate users.

```
59 this.validateLogin = (userName, password, callback) => {
60
61   // Helper function to compare passwords
62   const comparePassword = (fromDB, fromUser) => {
63     //return fromDB === fromUser;
64
65     // Fix for A2-Broken Auth
66     // compares decrypted password stored in this.addUser()
67     return bcrypt.compareSync(fromDB, fromUser);
68
69   };
```

2. Session Timeout and Protecting Cookies in Transit

The application does not have any provision to timeout user session. The session will stay active until the user explicitly logs out. The application also does not prevent cookies from being accessed in script, making it vulnerable to Cross Site Scripting (XSS) attacks. Cookies are also not prevented from being sent on insecure HTTP connection.

To secure this, the HTTPOnly HTTP header that prevents cookies from being accessed by scripts can be set. The application now uses HTTP secure connections and cookies are configured to be sent only on secure HTTP connections.

```
81 app.use(session({
82   // genid: (req) => {
83     // return genuuid() // use UUIDs for session IDs
84   },
85   secret: cookieSecret,
86   // Both mandatory in Express v4
87   saveUninitialized: true,
88   resave: true,
89   /*
90   // Fix for A5 - Security MisConfig
91   // Use generic cookie name
92   key: "sessionId",
93   */
94
95
96   // Fix for A3 - XSS
97   // TODO: Add "maxAge"
98   cookie: {
99     httpOnly: true,
100    // Remember to start an HTTPS server to get this working
101    secure: true,
102    maxAge: 3600000
103  }
104 }));
```

In addition, when the user clicks logout, the session and session cookie should be destroyed. This can be done using the following code.

```
116 app.get('/logout', function(req, res) {
117   // Destroys the session and clears the cookie
118   req.session.destroy(function(err) {
119     if (err) {
120       console.log(err); // Or handle the error as needed
121     } else {
122       res.clearCookie('connect.sid'); // Adjust the cookie name if needed
123       res.redirect('/'); // Redirects to the home page after logout
124     }
125   });
126 });
```

3. Session Hijacking

The application does not regenerate a new session ID when a user logs in which allows for the possibility of an attacker stealing the cookie with the session ID and using it to log in. This vulnerability is found in session.js and can be seen below.

```
111
112 // Fix the problem by regenerating a session in each login
113 // by wrapping the below code as a function callback for the method req.session.regenerate()
114 // i.e:
115 // 'req.session.regenerate(() => {})'
116 req.session.userId = user._id;
117 return res.redirect(user.isAdmin ? "/benefits" : "/dashboard");
118
```

To secure this, re-generate a new session ID upon login. This can be done by wrapping the following code as a function callback for the method `req.session.regenerate()`.

```

118 req.session.regenerate(function() {
119   req.session.userId = user._id;
120   if (user.isAdmin) {
121     return res.redirect("/benefits");
122   } else {
123     return res.redirect("/dashboard");
124   }
125 })

```

A2 –2 Password Guessing Attacks

Implementing robust minimum password criterion can make it more difficult for an attacker to guess a password. This vulnerability can be addressed by forcing a password length, complexity, and by providing the user with clear authentication failure responses. These modifications can be made to session.js. The original password requirements can be seen below.

```

146 const validateSignup = (userName, firstName, lastName, password, verify, email, errors) => {
147
148   const USER_RE = /^[1,20}$/;
149   const FNAME_RE = /^[1,100}$/;
150   const LNAME_RE = /^[1,100}$/;
151   const EMAIL_RE = /^[\\S]+@[\\S]+\\. [\\S]+$/;
152   const PASS_RE = /^[1,20}$/;

```

PASS_RE can be modified to the following which requires at least 8 chars with numbers and both lowercase and uppercase letters.

```

146 const validateSignup = (userName, firstName, lastName, password, verify, email, errors) => {
147
148   const USER_RE = /^[1,20}$/;
149   const FNAME_RE = /^[1,100}$/;
150   const LNAME_RE = /^[1,100}$/;
151   const EMAIL_RE = /^[\\S]+@[\\S]+\\. [\\S]+$/;
152   //const PASS_RE = /^[1,20}$/;
153
154   //Fix for A2-2 - Broken Authentication - requires stronger password
155   //(at least 8 characters with numbers and both lowercase and uppercase letters.)
156   const PASS_RE = /^(?=.*\\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/;

```

Now when you go to create an account you get an error if you fail to meet the password requirements.

OWASP Node Goat Project

Already a user? [Login](#)

Password must be 8 to 18 characters including numbers, lowercase and uppercase letters.

Enter sign up information

Another place where password guessing is made easier for attackers is the specific feedback as to whether password was incorrect or if the user doesn't exist. This information can be valuable to an attacker when brute forcing attempts. This vulnerability can be addressed by making the error message much more generic. This is another change to session.js. Instead of a specific error like *invalidUserNameErrorMessage* or *invalidPasswordErrorMessage*, we just use the general *errorMessage*.

```
82     return res.render("login", {
83       userName: userName,
84       password: "",
85       //loginError: invalidUserNameErrorMessage,
86       //Fix for A2-2 Broken Auth - Uses identical error for both username, password error
87       loginError: errorMessage,
88       environmentalScripts
89     });
90   } else if (err.invalidPassword) {
91     return res.render("login", {
92       userName: userName,
93       password: "",
94       loginError: invalidPasswordErrorMessage,
95       //Fix for A2-2 Broken Auth - Uses identical error for both username, password error
96       loginError: errorMessage,
97       environmentalScripts
98     });
99   }
```

Now even though the user "mweber25" does not exist, the error is general and doesn't tell you any more information than necessary.

Invalid username and/or password ×

User Name

mweber25

Password

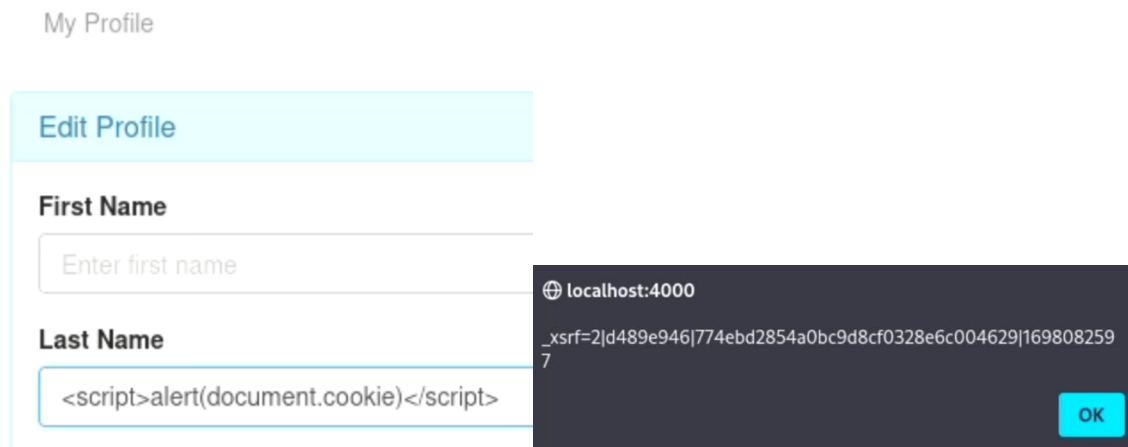
Enter Password

New user? Sign Up

Submit

A3 XSS

In the “Profile” tab, the text inputs can be used to cause a Cross Site Scripting (XSS) attack.



Typing into the Last Name field with the alert function and the cookie passed in shows the cookie in an alert prompt. Of course, this should not happen, and can be exploited to have other users’ cookies show and taken to use for malicious purposes.

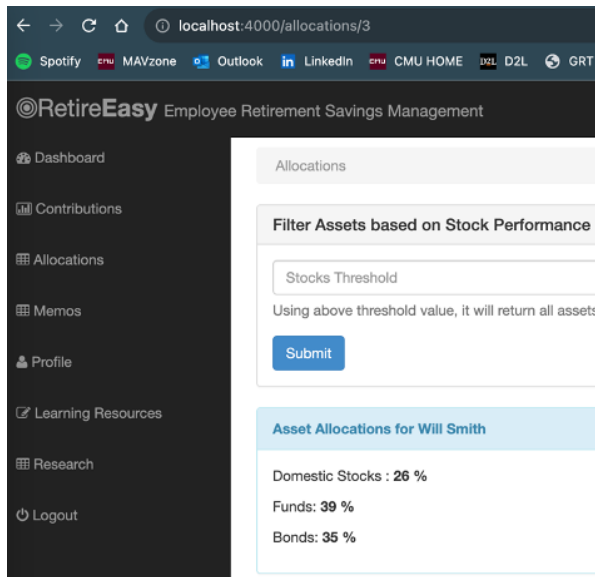
```
// Template system setup
swig.setDefaults({
  // Autoescape disabled
  autoescape: false
  /*
  // Fix for A3 - XSS, enable auto escaping
  autoescape: true // default value
  */
});

// Fix for A3 - XSS
// TODO: Add "maxAge"
cookie: {
  httpOnly: true,
  // Remember to start an HTTPS server to get this working
  secure: true
}
```

The fixes here are to change the “autoescape” property to true for Swig (a templating engine) and to force the cookie to use HTTPS instead of HTTP. The property “autoescape” in Swig will change the “<” and “>” characters into the “<” and “>” respectively, which makes sure that the input is not parsed as an HTML element. For the secure property, by forcing the cookie to use HTTPS, the cookie cannot be read by a man in the middle effectively securing from prying eyes.

A4 Insecure DOR

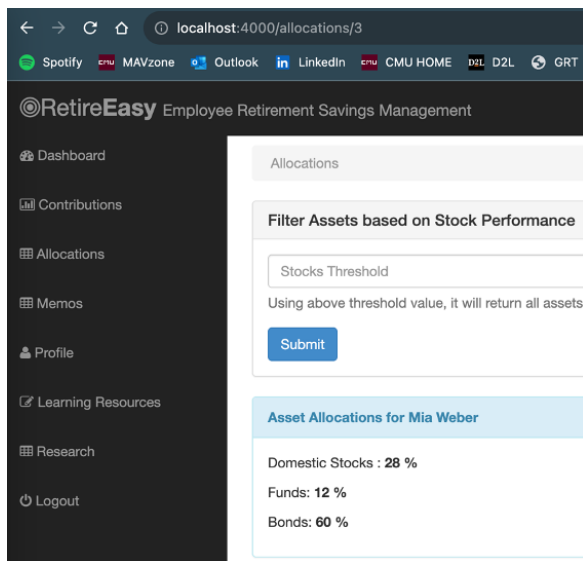
A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. If we visit the “Allocations” tab on the website and change the number in the URL we can access the data of other users, in the following image we can see the information for the user “Will Smith”.



To fix this vulnerability, we need to include an access control check to ensure the user is authorized for the requested object. In `allocations.js`, the insecure application takes user ID from the URL to fetch the allocations. Instead, retrieve allocations for logged in users using `req.session.userId` instead of taking it from the URL.

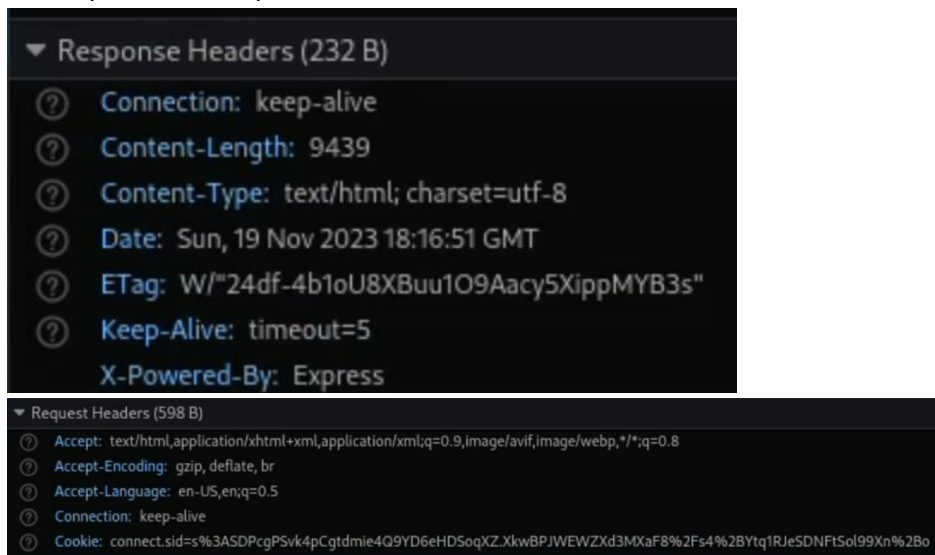
```
13 // Fix for A4 Insecure DOR - take user id from session instead of from URL param
14 const { userId } = req.session;
15
16 /*const {
17   userId
18 } = req.params;*/
19 const {
20   threshold
21 } = req.query;
```

Now we can try to access Will Smith’s data by altering the URL and it won’t work.



A5 Misconfig

When checking the network in a browser's developer tools, there is some information exposed via request and response headers.

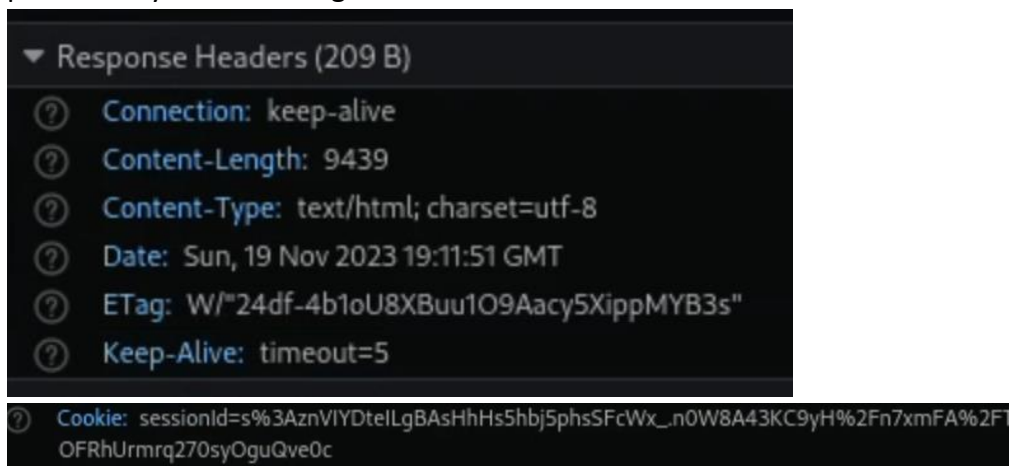


This exposes what the website is powered by, which is not completely insecure, but gives attackers more information up front. Also, the cookie is shown using the internal representation of it, which means an attacker knows exactly how to grab it if they get access.

```
// Fix for A5 - Security MisConfig
// TODO: Review the rest of helmet options, like "xssFilter"
// Remove default x-powered-by response header
app.disable("x-powered-by");
```

```
// Enable session management using express middleware
app.use(session({
  // genid: (req) => {
  //   return genuuid() // use UUIDs for session IDs
  // },
  secret: cookieSecret,
  // Both mandatory in Express v4
  saveUninitialized: true,
  resave: true,
  // Fix for A5 - Security MisConfig
  // Use generic cookie name
  key: "sessionId",
})
```

To fix this, all that needs to be done is change Express (the app object in the code) to disable x-powered-by and use the generic cookie name sessionId.



When checking the network tab now, there is no more identifiable information from the back end.

A6 Sensitive Data

This vulnerability allows an attacker to access sensitive data such as credit cards, tax IDs, authentication credentials, and more to conduct fraud, theft, or other crimes. If a site doesn't use SSL/TLS for all authenticated pages, an attacker can monitor network traffic and steal user's session cookie.

The application uses HTTP connection to communicate with the server. A secure HTTPS server can be set using HTTPs module. Modifications can be made to server.js to load keys to establish the HTTPS connection.

```

19  const config = require('./config/config');
20  // Fix for A6-Sensitive Data Exposure
21  // Load keys for establishing secure HTTPS connection
22  const fs = require("fs");
23  const https = require("https");
24  const path = require("path");
25  const httpsOptions = {
26    key: fs.readFileSync(path.resolve(__dirname, "./artifacts/cert/server.key")),
27    cert: fs.readFileSync(path.resolve(__dirname, "./artifacts/cert/server.crt"))
28  };
29

```

Now the secure HTTP server can be started, also in server.js. Adding the lines below was essential, as was commenting out the existing `http.createServer` line just above it which interfered. In addition, it was essential to specify port 4000 directly rather than getting that value from elsewhere.

```

157  https.createServer(httpsOptions, app).listen(4000, function() {
158    console.log("Express https server listening on port" + config.port);
159  });
160

```

The application stores user's personal sensitive information in plain text. Profile-dao.js can be modified to use crypto module to encrypt and decrypt sensitive information.

```

17  // Use crypto module to save sensitive data such as ssn, dob in encrypted format
18  const crypto = require("crypto");
19  const config = require("../config/config");
20
21  /// Helper method create initialization vector
22  // By default the initialization vector is not secure enough, so we create our own
23  const createIV = () => {
24    // create a random salt for the PBKDF2 function - 16 bytes is the minimum length according
25    const salt = crypto.randomBytes(16);
26    return crypto.pbkdf2Sync(config.cryptoKey, salt, 100000, 512, "sha512");
27  };
28
29  // Helper methods to encrypt / decrypt
30  const encrypt = (toEncrypt) => {
31    config.iv = createIV();
32    const cipher = crypto.createCipheriv(config.cryptoAlgo, config.cryptoKey, config.iv);
33    return `${cipher.update(toEncrypt, "utf8", "hex")} ${cipher.final("hex")}`;
34  };
35
36  const decrypt = (toDecrypt) => {
37    const decipher = crypto.createDecipheriv(config.cryptoAlgo, config.cryptoKey, config.iv);
38    return `${decipher.update(toDecrypt, "hex", "utf8")} ${decipher.final("utf8")}`;
39  };

```

The values now need to be encrypted before being saved in the database and decrypted on view.

```

68  // Fix for A7 - Sensitive Data Exposure
69  // Store encrypted ssn and DOB
70  if(ssn) {
71    user.ssn = encrypt(ssn);
72  }
73  if(dob) {
74    user.dob = encrypt(dob);
75  }

```

```

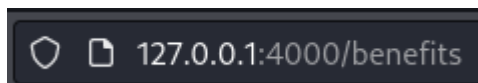
101 // Fix for A6 - Sensitive Data Exposure
102 // Decrypt ssn and DOB values to display to user
103 user.ssn = user.ssn ? decrypt(user.ssn) : "";
104 user.dob = user.dob ? decrypt(user.dob) : "";

```

Now NodeGoat is at https not http.

A7 Access Controls

Typing in 127.0.0.1:4000/benefits into the search bar will then show the benefits page regardless of the user.



Employee ID	First Name	Last Name	Benefits Start Date
2	John	Doe	01 / 10 / 2030 Save
3	Will	Smith	11 / 30 / 2025 Save

Benefits updated successfully.

Employee ID	First Name	Last Name	Benefits Start Date
2	John	Doe	01 / 10 / 2030 Save
3	Will	Smith	11 / 04 / 2025 Save


In this page every user's benefit information is shown and even modifiable, which is not good at all.

```
// Benefits Page
// app.get("/benefits", isLoggedIn, benefitsHandler.displayBenefits);
// app.post("/benefits", isLoggedIn, benefitsHandler.updateBenefits);
// Fix for A7 - checks user role to implement Function Level Access Control
app.get("/benefits", isLoggedIn, isAdmin, benefitsHandler.displayBenefits);
app.post("/benefits", isLoggedIn, isAdmin, benefitsHandler.updateBenefits);
```

To fix this, passing in `isAdmin` will then check if the user is an admin before getting or posting anything from the benefits page. Before, it just checked if the user was logged in and nothing else.



[Tutorial Guide: Learn OWASP Top 10](#)

**RetireEasy**
Employee Retirement Savings Management

User Name

Password

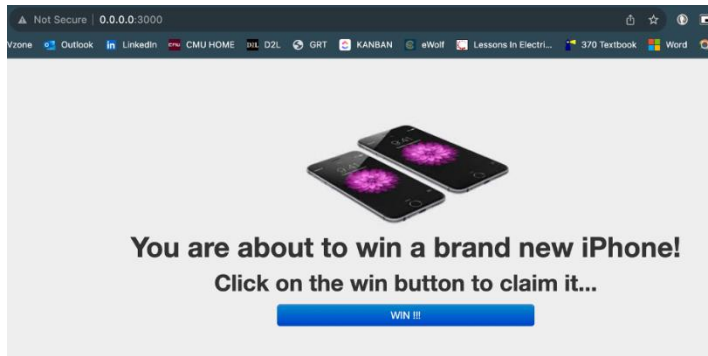
[New user? Sign Up](#)

When trying to go to the benefits page again, it just kicks back to the login screen.

A8 CSRF

A Cross-Site Request Forgery attack (CSRF) 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.

A separate repository that contains code for a node server can be cloned and run using *node server.js*. It starts the server at 0.0.0.0:3000 seen below.



If a user selects the “WIN !!!” button, it should override the Bank Account # and the Bank Routing # on the profile page.

For this exploit to work the correct URL to send the forged POST request to needs to be identified. This URL is <https://localhost:4000/profile>. The modified source code can be seen below.

```
...<!DOCTYPE html> == $0
<html lang="en" ng-app>
  <head>
    <title>NodeGoat CSRF Example</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="/css/bootstrap.min.css">
    <link rel="stylesheet" href="/css/bootstrap-responsive.min.css">
    <style> body { padding-top: 60px; } </style>
  </head>
  <body style="background-color:#EEE">
    <div class="container">
      ::before
      <div class="page-header">
        <div align="center">
          <div style="background:url(/img/iphones.jpeg) no-repeat center; height:150px; width:300px;">
          </div>
          <h1>You are about to win a brand new iPhone!</h1>
          <h2> Click on the win button to claim it...</h2>
          <form method="POST" action="https://localhost:4000/profile" target="iframeHidden">
            <input type="hidden" name="bankRouting" value="888888888">
            <input type="hidden" name="bankAcc" value="999999999">
            <input type="submit" style="text-align: center" class="span4 btn btn-primary" value="WIN !!!">
          </form>
        </div>
      </div>
      ::after
    </div>
    <iframe name="iframeHidden" width="1" height="1"></iframe>
  </body>
</html>
```

And now the sensitive numbers are overwritten.

The use of express csrf middleware will be the key to fixing this. A token named “_csrf” is generated and added to requests which mutate the state. When a form is submitted, the middleware checks for the existence of the token and validates it by matching it to the generated token for the response-request pair, if it doesn’t match, the request is rejected.

To implement this, the following lines can be added to *server.js*, first we need to run *npm install csrf* in the root of NodeGoat and then add the following import line at the top of the file.

```
18
19  const csrf = require('csrf');
20
```

Now the following lines can be added to enable express csrf.

```
108  // Fix for A8 - CSRF
109  // Enable Express csrf protection
110  app.use(csrf());
111  // Make csrf token available in templates
112  app.use((req, res, next) => {
113    res.locals.csrfToken = req.csrfToken();
114    next();
115  });
```

Now the token can be included in the hidden form field by adding the following line in *views/profile.html*.

```
72  </div>
73  <input type="hidden" name="_csrf" value="{{csrfToken}}" />
74  <button type="submit" class="btn btn-default" name="submit">Submit</button>
75
```

Now when a user clicks on the “WIN !!!” button, the sensitive data is not overwritten.

A9 Insecure Components

In the “Memo” tab, the user can write down any notes they want in the Markdown format. However, NodeGoat uses an insecure version of Marked which is a Markdown parser. This specific version of Marked allows for XSS attack.

In the example above, an alert is created sending a simple message. However, this could also farm information off the website that should not be shared such as an email, password, or maybe even a social security number.

Really the only fixes here is to either not run the web application as root, find a non-vulnerable version of the package, or even use a new package all together.

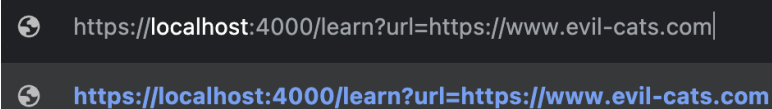
A10 Redirects

Web applications redirect and forward users to other pages and websites and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or other untrusted sites.

The “Learning Resources” link in the application redirects to another website without validating the URL.

```
69 // Handle redirect for learning resources link
70 app.get("/learn", isLoggedIn, (req, res) => {
71     // Insecure way to handle redirects by taking redirect url from query string
72     return res.redirect(req.query.url);
73 });
```

An attacker can change the query URL parameter to point to a malicious website. Victims are more likely to click on it, as the initial part of the link (before query parameters) points to a trusted site.



The screenshot shows a browser address bar with two versions of a URL. The top version is `https://localhost:4000/learn?url=https://www.evil-cats.com|` with a cursor at the end. The bottom version is `https://localhost:4000/learn?url=https://www.evil-cats.com`, which is highlighted in blue, indicating it has been selected or is the current page.

The code can be modified to include the following lines in *index.js* which use mapping value parameters instead of taking it from a query string.

```
69 // Handle redirect for learning resources link
70 const allowedUrls = {
71     "KhanAcademyResource":
72     "https://www.khanacademy.org/economics-finance-domain/core-finance/investment-vehicles-tutorial
73     // Add other mappings as needed
74 };
75
76 app.get("/learn", isLoggedIn, (req, res) => {
77     const key = req.query.url;
78     if (key in allowedUrls) { // If the URL is verified -> open
79         return res.redirect(allowedUrls[key]);
80     } else { // If the URL is not verified -> return error
81         return res.status(400).send("Invalid redirection key");
82     }
83     // Insecure way to handle redirects by taking redirect url from query string
84     //return res.redirect(req.query.url);
85 });
```

The *layout.html* file also needs some adjustments to open the URL in accordance with the new secure redirection setup. The following html code needs to be changed.

```
</li><a id="learn-menu-link" target="_blank"
href="/learn?url=https://www.khanacademy.org/economics-finance-domain/core-
finance/investment-vehicles-tutorial/ira-401ks/v/traditional-iras"><i class="fa
fa-edit"></i> Learning Resources</a>
```

That html code can be adjusted to the following to use the new technique.

```
</li>
<li><a id="learn-menu-link" target="_blank" href="/learn?url=KhanAcademyResource"><i class="fa fa-edit"></i> Learning Resources</a>
</li>
```

Now when we copy the link address for the “Learning Resources” page on the application it uses the secure method which prevents the link from being altered successfully to point at a malicious site. If an attacker changes the URL in the following address, it will return an error unless it is a value in allowedUrls.

```
🌐 https://localhost:4000/learn?url=KhanAcademyResource|
🌐 https://localhost:4000/learn?url=KhanAcademyResource
```