Lab 4x: Web application security

1 Purpose of this lab

In this lab, we introduce some of the most important concepts of web application security faced by all of today's web applications. One open source community, named OWASP, have a project which releases a list of the top 10 web application security risks. In the exercises of this lab, we will look at a subset of these and how you can counteract such vulnerabilities within your applications.

2 The Open Web Application Security Project (OWASP)

Taken from:
https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project

The Open Web Application Security Project (OWASP) foundation was formed in 2001, becoming a not-for-profit charitable organisation in the US in 2004. “OWASP is an open community dedicated to enabling organisations to conceive, develop, acquire, operate and maintain applications that can be trusted.” All of the content generated by OWASP is free and open to anyone interested in improving application security. They advocate approaching application security as a people, process, and technology problem because the most effective approaches to application security include improvements in all of these areas.

Read more at their website: www.owasp.org.

3 The OWASP Top 10 project

Taken from: https://www.owasp.org/index.php/Top_10_2017-Introduction

The OWASP Top 10 project is a list of the 10 most critical web application security risks. The primary aim of the OWASP Top 10 is to educate developers, designers, architects, managers, and organizations about the consequences of the most important web application security weaknesses. The Top 10 provides basic techniques to protect against these high risk problem areas – and also provides guidance on where to go from there.

The OWASP Top 10 for 2017 is based primarily on 11 large datasets from firms that specialize in application security, including 8 consulting companies and 3 product vendors. This data spans vulnerabilities gathered from hundreds of organizations and over 50,000 real-world applications and APIs. The Top 10 items are selected and prioritized according to this prevalence data, in combination with consensus estimates of exploitability, detectability, and impact.

However, they warn that developers shouldn't stop after just fixing the top 10. There are hundreds of issues that could affect the overall security of a web application as discussed in:
The OWASP Developer's Guide
https://www.owasp.org/index.php/OWASP_Guide_Project
And:
The OWASP Cheat Sheet Series
https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series.

These are essential reading for anyone developing web applications.

This year, the Top 10 have been re-assessed (with the previous version of the list being released in 2013). The list for this year is yet to be finalised, but here are the items as they currently stand:

1. **Injection -** Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
2. **Broken Authentication and Session Management -** Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).
3. **Cross-Site Scripting (XSS)** - XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
4. **Broken Access Control** - Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
5. **Security Misconfiguration -** Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
6. **Sensitive Data Exposure -** Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other

crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

7. **Insufficient Attack Protection -** The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks

8. **Cross-Site Request Forgery (CSRF) -** 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. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

9. **Using Components with Known Vulnerabilities -** Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

10. **Underprotected APIs -** Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

In this lab, we will use NodeGoat (https://github.com/OWASP/NodeGoat.git) to look at the first few of the Top 10 vulnerabilities. NodeGoat is a dummy application provided by OWASP that allows us to explore the effects of these vulnerabilities and how to prevent them. We will be adapting the tutorials taken from http://nodegoat.herokuapp.com/tutorial. After completing the exercises below, you may wish to look at the tutorial for tips on how to manage the remaining risks that we do not cover here.

4 Browser differences

In the following exercises, you may notice differences between how browsers handle security vulnerabilities. For example, Chrome includes security checks into the browser that actively prevent the *injection* of JavaScript in Exercise 2, whilst FireFox does does not. Whilst Chrome prevents injection it does not prevent execution of previously injected JavaScript. So if you inject the JavaScript using FireFox, the Chrome browser would then execute the script.

5 Exercise 1: Getting started

NodeGoat stores its sample data in MongoDB (a widely-used NoSQL database.) For the lab, we have created a MongoDB VM that can be run from VirtualBox. At home though, you may wish to either install MongoDB on your local system, or to access one of the various cloud-based services that provide MongoDB remotely. Some guidance for those last two options can be found on NodeGoat's GitHub page.

First, start your MongoDB instance.

1. **To initially create the VM, run '/netfs/share/bin/seng365vm-mongodb'.** From then on, you can start and stop the VM through VirtualBox
2. Open VirtualBox and start the MongoDB-365 VM.
3. In some later steps you might wish to use the `mongo` shell to examine the database contents: run it from within the VM while logged on as username 'seng365' and password 'secret'. A quick start on the shell can be found at https://docs.mongodb.com/manual/reference/mongo-shell/.

Now we'll install NodeGoat and get it running on our machine.

2. Download a copy of NodeGoat to your local machine:

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

3. In your terminal, navigate into the NodeGoat directory and run '`npm install`'
4. Uncomment the db line in the file `config/env/development.js`. That line should now read:

   ```
   db: "mongodb://localhost:27017/nodegoat",
   ```

5. Still in the NodeGoat directory, run the command below to populate the database with data. You should see some lines beginning with `>>Connected to the database: mongodb://localhost:27017/nodegoat` and ending with `Done.`

```
npm run db:seed
```

6. Start NodeGoat with '`npm start`'
7. Confirm that everything has worked by navigating to http://localhost:4000 in a browser.
8. Have a look around to familiarise yourself with the application. Below are some default login credentials:

Account	Username	Password

Admin Account	admin	Admin_123
User Accounts	user1, user2	User1_123, User2_123

New users can also be added using the sign-up page.

6 Exercise 2: Cross-site scripting (XSS) vulnerabilities

6.1 What are XSS vulnerabilities?

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 victim's browser, which can then access any cookies, session tokens, or other sensitive information retained by the browser, or redirect the user to malicious sites.

6.2 Performing an XSS attack

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 is embedded in an HTML page provided to the victim.

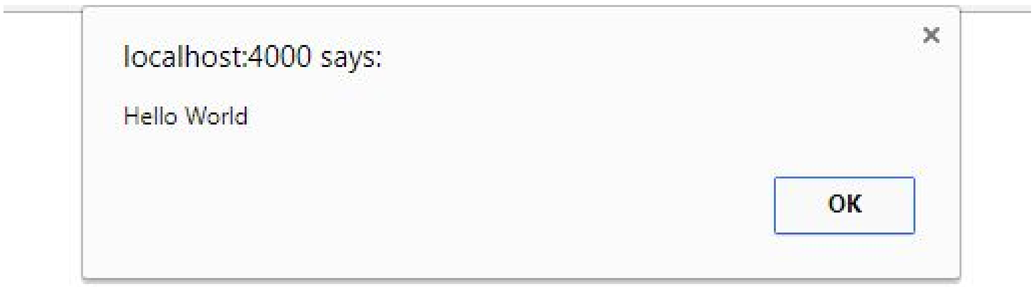
Each of reflected and stored XSS can occur on the server or on the client (which is also known as DOM based XSS), depending on when the malicious data is injected into the HTML markup.

Let’s have a go at conducting a stored XSS attack:

1. For this exercise, we need two users: one administrator (the “admin” user), and one evil user (either of users “user1” or “user2” will do).
2. Run the NodeGoat server and log in as your evil user : this will be the user conducting the attack.
3. Navigate to the ‘Profile’ page and change your evil user’s last name to the following:

```
<script>alert(‘Hello World!’);</script>
```

4. Submit the form and logout (you may need to refresh the page, and will also need to provide a bank routing number; just copy the example number)
5. Imagine some time has passed and the administrator goes to log in. Log in to the administrator account:


localhost:4000 says:

Hello World

The code has been executed. Making the administrator view an alert box might seem harmless, but imagine that instead of an alert box, we inserted code to send the administrators cookies to us, or redirected them to a malicious site.

6.3 How to prevent XSS attacks

- **Input validation and sanitization:** Input validation and data sanitization are the first line of defense against untrusted data. Apply white list validation wherever possible.
- **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.
- **HTTPOnly cookie flag:** Preventing all XSS flaws in an application is hard. To help mitigate the impact of an XSS flaw on your site, set the HTTPOnly flag on session cookie and any custom cookies that are not required to be accessed by JavaScript.
- **Implement Content Security Policy (CSP):** CSP is a browser side mechanism which allows creating whitelists for client side resources used by the web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources. For example, the CSP header below allows content only from example site's own domain (mydomain.com) and all its sub domains.

```
● Content-Security-Policy: default-src 'self' *.mydomain.com
```

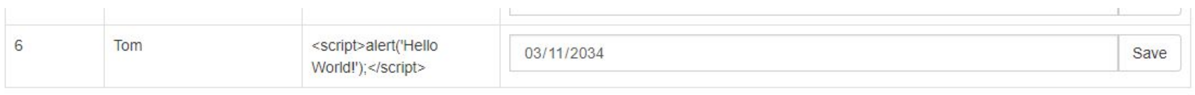
- **Apply encoding on both client and server side:** It is essential to apply encoding on both client and server side to mitigate DOM based XSS attack, in which untrusted data never leaves the browser.

Let’s look at how to prevent the dreaded ‘Hello World’ attack:

1. In our server.js file, enable HTML encoding using the template engine’s auto-escape flag:

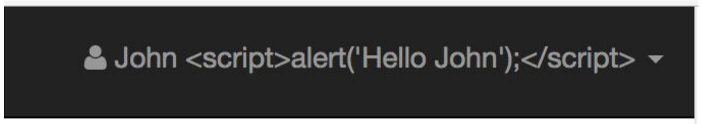
```
// Template system setup
swig.setDefaults({
   autoescape: true // default value
});
```

2. Restart your server.
3. Login as admin, the popup will no longer be displayed and the malicious code will have been escaped and visible as text:



6	Tom	<script>alert('Hello World!');</script>	03/11/2034 Save

4. Logon as the user as see how the Profile icon has changed, for example;



John <script>alert('Hello John');</script>

Read more: http://nodegoat.herokuapp.com/tutorial/a3

7 Exercise 3: Injection vulnerabilities

7.1 What are injection vulnerabilities?

Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

7.2 Performing an injection attack

7.2.1 Denial of Service (DoS) Attack

In this attack, we are going to break the server from within the application:

1. From a browser, connect to NodeGoat at `http://localhost:4000`, log in as user1 and navigate to the 'Contributions' page.
2. In one of the inputs under the 'New Payroll Contribution Percent (per pay period)' column, enter the following:

```
process.kill(process.pid)
```

3. Submit the form

The input that you have entered will be parsed by Node, and the current process will be killed. This means that no one can access the site until the process is re-started. You should be met with a 'connection refused' error in the browser (or equivalent error, depending on the browser you're using). Checking your VM window will confirm that the Node server has stopped running.

4. Restart NodeGoat for the next exercise with ‘`npm start`’

7.2.2 File System Access

As well as running DoS attacks, we can use injection to read the contents of the files on the server:

1. From the browser, login as user1 and navigate to the ‘Contributions’ page
2. In one of the inputs under the ‘New Payroll Contribution Percent (per pay period)’ column, enter the following:

```
res.end(require('fs').readdirSync('.').toString())
```

3. Submit the form

This time the input is parsed by Node, and a list of the current directories contents is displayed to the screen. Changing the parameter for readdirSync() allows you to read the rest of the machine's filesystem (try changing ‘.’ to ‘../../../../’, or see if you can list the contents of your ‘My Documents’ folder, or your OS’s equivalent.)

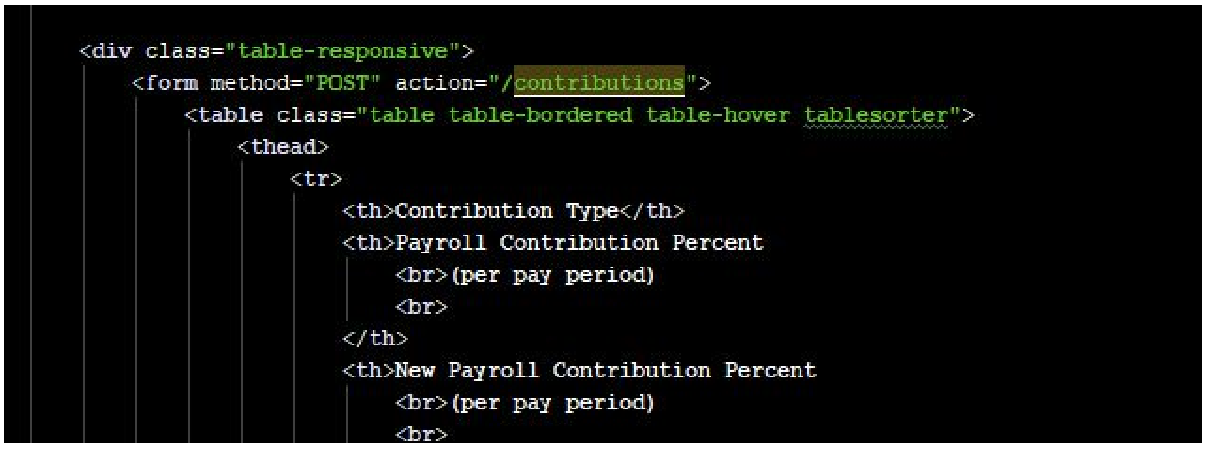
7.3 How to prevent injection attacks

To prevent server-side js injection attacks:

- Validate user inputs on server side before processing
- Do not use the `eval()` function to parse user inputs. Avoid using other commands with similar effect, such as `setTimeOut()`, `setInterval()`, and `Function()`.
- For parsing JSON input, instead of using eval(), use a safer alternative such as `JSON.parse()`. For type conversions use type `relatedparseXXX()` methods.
- Include `"use strict";` at the top of each source file.

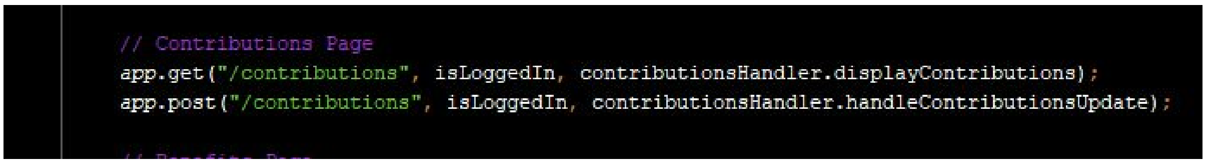
With these points in mind, let’s fix the form on the Contributions page that we have just ran our injection attacks against:

1. First we need to find the code. In WebStorm, open up the /app/views/ directory and open `contributions.html`. In this file, we can see the form, and the action that is taken when the form is submitted.



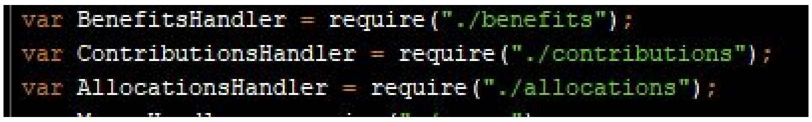
```
<div class="table-responsive">
    <form method="POST" action="/contributions">
        <table class="table table-bordered table-hover tablesorter">
            <thead>
                <tr>
                    <th>Contribution Type</th>
                    <th>Payroll Contribution Percent
                        <br>(per pay period)
                        <br>
                    </th>
                    <th>New Payroll Contribution Percent
                        <br>(per pay period)
                        <br>
```

2. Now open the /app/routes/ directory. The `index.js` file tells us that the /contributions route is handled by the `handleContributionsUpdate()` function in the contibutionsHandler:



```
// Contributions Page
app.get("/contributions", isLoggedIn, contributionsHandler.displayContributions);
app.post("/contributions", isLoggedIn, contributionsHandler.handleContributionsUpdate);
```

...which is imported at the top of `index.js` file from the `contributions.js` file:



```
var BenefitsHandler = require("./benefits");
var ContributionsHandler = require("./contributions");
var AllocationsHandler = require("./allocations");
```

3. Open up the `contributions.js` file and locate the `handleContributionsUpdate` function. This is the function that we need to fix. (You will see that NodeGoat has provided some useful comments that tell us what we need to do).
4. Replace the `eval()` calls with calls to `parseInt()`.
5. Now restart the server and retry the attacks. The input is now parsed safely and an error message will be displayed.

Another common form of injection attack to be aware of are database injection attacks (SQL/NoSQL injection). The tutorial at the following URL contains an explanation of database injection attacks and tips on how to prevent such attacks.

Read more: http://nodegoat.herokuapp.com/tutorial/a1

8 Exercise 4: Broken authentication vulnerabilities

8.1 What is a broken authentication vulnerability?

In this attack, an attacker (who can be an anonymous external attacker, a user with their own account who may attempt to steal data from other accounts, or an insider wanting to disguise his or her actions) uses leaks or flaws in the authentication or session management functions to impersonate other users. Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.

Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.

8.2 Exploiting a broken authentication vulnerability

Some examples of broken authentication vulnerabilities:

Scenario #1: Application timeouts aren't set properly. For example: a 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: An attacker acts as a man-in-middle and acquires the user's session id from network traffic, and then uses this authenticated session id to connect to the application without needing to enter the username and password.

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

8.3 How to prevent broken authentication vulnerabilities

8.3.1 Protecting user credentials

In NodeGoat, passwords are currently stored in plain text within the database. Let’s change it so that the application encrypts passwords in the database with a salted hash:

1. In WebStorm, navigate to the /app/data/ directory and open `user-dao.js`. We will be making changes to the addUser() function.
2. Before the user document is created, create a hash using the code below:

```
// Generate password hash
var salt = bcrypt.genSaltSync();
var passwordHash = bcrypt.hashSync(password, salt);
```

3. Now, when creating the user document, replace the password value with the newly created passwordHash variable

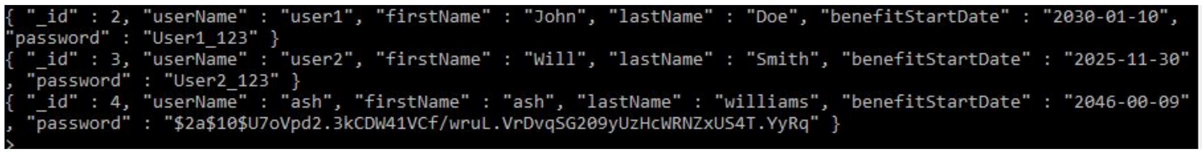
```
// Create user document
var user = {
   userName: userName,
   firstName: firstName,
   lastName: lastName,
   benefitStartDate: this.getRandomFutureDate(),
   password: bcrypt.hashSync(password, bcrypt.genSaltSync())
};
```

4. Finally, inside the `validateLogin()` function, we want to change the `comparePassword()` method so that instead of comparing the input like for like with the database, we compare the hashes:

```
this.validateLogin = function(userName, password, callback) {

   // Helper function to compare passwords
   function comparePassword(fromDB, fromUser) {
       return bcrypt.compareSync(fromDB, fromUser);
   }
```

5. Save and re-run the server. Logging in with the existing accounts will cause the server to crash as the application is expecting a hash rather than the plain text passwords that exist in the database. So create a new user.
6. Open your database and check the users collection, your newly created user will have a hash instead of a plain text password:

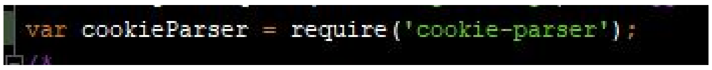


```
{ "_id" : 2, "userName" : "user1", "firstName" : "John", "lastName" : "Doe", "benefitStartDate" : "2030-01-10",
"password" : "User1_123" }
{ "_id" : 3, "userName" : "user2", "firstName" : "Will", "lastName" : "Smith", "benefitStartDate" : "2025-11-30"
, "password" : "User2_123" }
{ "_id" : 4, "userName" : "ash", "firstName" : "ash", "lastName" : "williams", "benefitStartDate" : "2046-00-09"
, "password" : "$2a$10$U7oVpd2.3kCDW41VCf/wruL.VrDvqSG209yUzHcWRNZxUS4T.YyRq" }
>
```

8.3.2 Session timeout and protecting cookies in transit

NodeGoat does not contain any provision to timeout a user session. The session stays active until the user explicitly logs out. In addition to that, the app does not prevent cookies being accessed in scripts, making applications vulnerable to Cross Site Scripting (XSS) attacks. Also, cookies are not prevented from being exchanged over insecure HTTP connection. Let's fix these problems:

1. Install `cookie-parser` with npm, and add a `require` statement to the top of your `server.js` file:



```
var cookieParser = require('cookie-parser');
```

2. In the `server.js` file, set the application up to use `CookieParser()` (we've placed it just before the 'Enable session management using express middleware' comment around line 80.

```
// Enable session management using express middleware
app.use(cookieParser());
```

3. Underneath, uncomment the cookie block. The HTTPOnly flag prevents the cookies from being accessed by scripts:

```
// Enable session management using express middleware
app.use(session({
   secret: config.cookieSecret,
   cookie: {
       httpOnly: true,
   },
   // Both mandatory in Express v4
```

```
    saveUninitialized: true,
    resave: true
}));
```

8.3.3 Password guessing attacks

Implementing a robust minimum password criteria (minimum length and complexity) can make it difficult for an attacker to guess the password. Failing to strengthen the password can result in an attacker exploiting this vulnerability by brute force password guessing, more likely using tools that generate random passwords.

Password length: Passwords should be at least eight (8) characters long. Combining this length (together with complexity in the choice of characters - see below) makes a password difficult to guess and/or makes it more difficult to apply brute force approaches.

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.

NodeGoat does not currently enforce strong passwords. Let's fix this:

1. Open `/app/routes/session.js`, note that the `validateSignup()` function currently uses the following regex enforcement on passwords:

```
var PASS_RE = /^.{1,20}$/;
```

2. Change this line using a regex that ensures the password has a minimum length of 8 characters, using numbers and both lowercase and uppercase characters:

```
var PASS_RE =/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/;
```

3. Another issue in the `session.js` file is that the `handleLoginRequest()` function enumerates whether the password is incorrect or if the user does not exist. This information can be valuable to an attacker with brute forcing attempts. We can fix this by using a generic error message for both occasions:

```
var errorMessage = "Invalid username and/or password";
//var invalidUserNameErrorMessage = "Invalid username";
//var invalidPasswordErrorMessage = "Invalid password";
if (err) {
   if (err.noSuchUser) {
       return res.render("login", {
           userName: userName,
           password: "",
           loginError: errorMessage
       });
   } else if (err.invalidPassword) {
       return res.render("login", {
           userName: userName,
           password: "",
           loginError: errorMessage
       });
   } else {
       return next(err);
   }
}
```

4. Run the server and create a new user to ensure that the password requirements are being enforced.

In many scenarios, it is safest to outsource your authentication strategy to another website (have you ever been on a website that asks you to log in with your Facebook account?). Passport.js (http://passportjs.org) provides authentication middleware for applications. It contains over 300 “strategies” or pre-defined templates for making use of authenticating applications (Facebook, Google, Twitter etc) and is widely used. OAUTH and OpenID form the basis for most of these strategies.

Read more: http://nodegoat.herokuapp.com/tutorial/a2

9 Summary

As mentioned earlier, the three types of vulnerability described in this lab are just a subset of the hundreds that exist. These exercises have been taken from http://nodegoat.herokuapp.com/tutorial.

Visiting that link also provides similar tutorials for the other seven of the Top 10 most critical web application security risks. We recommend that you make yourself familiar with as many of these attacks, and their preventions, as possible. The more you know about, and are able to handle security threats, the safer your applications will become.