Security in Kraken with Lusca

In this lab, you will familiarize yourself with the security features in Kraken, offered by Lusca.

# Objectives

You will learn how to prevent

* cross-site request forgeries (CSRF/XSRF),
* cross-site scripting attacks (XSS),
* a browser from downloading unauthorized content via content security policies (CSP).

# Install Tools

The first thing we have to do is to use Node Package Manager, npm, to install generator-kraken. Do this by opening a command prompt in the lesson directory and issuing the command npm install –g generator-kraken.

# Lab Steps

Now that the tools are installed, we can begin the lab, whose goal is to scaffold a Kraken web application, then use Lusca to add our security features for CSRF & CSP.

## Scaffold a new web application with Kraken

1. Open a terminal in the lab directory, then issue the command yo kraken. Enter lusca-test for the app name, Lusca Test Application for the description, and whatever you want for the author. Take the defaults for the remaining prompts.

This will scaffold a web application that uses Express and Kraken in the directory lusca-test.

## Start & view the web application

1. Change into the lusca-test directory and issue the command npm start; this will start the server process.
2. Open http://localhost:8000 in a web browser. You should see something similar to the following:



Once you see this, move on to the next step.

## Protect the web application from cross-site request forgery

Out of the box, our web application is susceptible to cross-site request forgery (CSRF or XSRF). In order to prevent this, let's configure this protection now.

1. Open lusca-test/config/config.json and copy and paste the following at the end of the middleware object:

"appsec": {  
 "enabled": true,  
 "priority": 100,  
 "module": {  
 "name": "lusca",  
 "arguments": [  
 {  
 "csrf": true,  
 "csp": false  
 }  
 ]

}  
}

At this point, our web application is now no longer susceptible to CSRF attacks. Let's write some code now to prove it.

## Add a controller to test CSRF prevention

1. Kill the web server (Ctrl-C) and issue the command yo kraken:controller csrf. This will create a controller at lusca-test/controllers/csrf/index.js, a model at lusca-test/models/csrf.js, and a Dust.js view template at lusca-test/public/templates/csrf/index.dust.

## Update the view to demonstrate CSRF prevention

1. Open the Dust template at lusca-test/public/templates/csrf/index.dust and change its content to the following:

{>"layouts/master" /}  
  
{<body}  
 <h1>{@pre type="content" key="greetings"/}</h1>  
 <h2>{@pre type="content" key="form"/}</h2>  
 <form method="post">  
 Name: <input type="text" name="name" value="{name}" size="100"/>  
 <br/>  
 CSRF Token: <input type="text" name="\_csrf" value="{\_csrf}"  
 size="100"/>  
 <br/>  
 <input type="submit" value="Submit"/>   
 </form>  
{/body}

Take a moment to look at this form. It renders a simple form that includes an input for a name and an input showing the current CSRF token that is used to prevent CSRF attacks.

Normally, this value is used in hidden form inputs, as it's not intended for human consumption.

## Update the i18n CSRF bundle

1. Notice the <h2> element in the template; it's expecting a message keys called greetings and form. Open lusca-test/locales/US/en/csrf/index.properties and replace its content with the following:

form=Enter name, then click submit. Change the CSRF token to force error.  
greetings=Hello, {name}.

## Update the default CSRF model

1. Open the model file lusca-test/models/csrf.js and change it to return the literal JavaScript object { name: 'Dude' }, which will be the default model that our controller will have the view render.

## Update the CSRF controller

1. Open the controller in lusca-test/controllers/index.js and change its content to export a function that handles HTTP GETs & POSTs:

module.exports = function(app) {  
 var model = new CsrfModel();  
  
 app.get('/', function(req, res) {  
 res.render('csrf', model);  
 });  
  
 app.post('/', function(req, res) {  
 model.name = req.body.name || '(none)';  
 res.render('csrf', model);  
});

As you can see, GETs will cause simply cause the form to be rendered will the default model, and POSTs will update the model with the name given in the name input from the HTML form, or (none) if one wasn't entered. So far, so good. Easy peasy!

## Run the web application

1. Start the web application again with npm start, return to the browser and view the URL http://localhost:8000/csrf. You should see the form looking similar to this:



Notice the CSRF token. This changes on each request to ensure that only authorized requests are coming into this web application. Go ahead and refresh the form and notice that the CSRF token changes each time.

When you see the above form and the new tokens upon each GET request, move on to the next step.

## Change the name and view the result

1. Enter a new name in the name field and view the result. You should see something similar to the following:



The site successfully handled the POST and rendered a new model. When the site is behaving properly, move on to the next step.

## Elicit a CSRF attack prevention

Now, we're going to simulate a CSRF attack. An attacker cannot know the CSRF token when attempting a request from a third party machine; the CSRF can only come from the origin web server, and is cryptographically secure.

1. Before the next form submittal, change the value of the CSRF token (or delete it), simulating an attack. The resultant page should look something like the following:



Our CSRF middleware component, configured in lusca-test/config/config.json, has detected the unauthorized request and denied it! Issue a new GET request to http://localhost:8000/csrf and resubmit the form without changing the CSRF token; you should see a successful result.

When you see the site behaving correctly, move on to the next step, knowing that the browser can now only respond to requests that originated from this server.

## Introduce a cross-site scripting (XSS) vulnerability

Next, we're going to see how a site can open a hole for an attacker to issue a cross-site scripting (XSS) attack.

1. Stop the web server and issue the command yo kraken:controller echo, which will generate a model, controller view, and i18n bundle in their respective places. As you might guess by the name, we're going to simply echo a string that the user submits via a web form.

## Implement the echo functionality

Now, let's implement the echo functionality.

1. First, let's modify the generated echo model, by replacing the content of lusca-test/models/echo.js with the following:

module.exports = function EchoModel() {  
 return {  
 message: 'Enter a message'  
 };  
};

1. Next, update the controller, controllers/echo/index.js to handle HTTP GET & POST requests by simply populating the model with the value of the message variable:

module.exports = function (router) {  
 var model = new EchoModel();  
 function handle(req, res) {  
 model.message = req.param('message') || model.message;  
 res.render('echo', model);  
 }  
  
 router.get('/', handle);  
 router.post('/', handle);  
};

1. Now, update the Dust template lusca-test/public/templates/echo/index.dust that is the HTML form page to have the following content:

{>"layouts/master" /}  
  
{<body}  
 <h1>{@pre type="content" key="greeting"/}</h1>  
 <form method="post">  
 Message: <input type="text" name="message" value="{message}"  
 size="100"/>  
 <input type="hidden" name="\_csrf" value="{\_csrf}"/>  
 <input type="submit" value="Submit"/>   
 </form>  
{/body}

As you can see, it's a simple form with an input that will be echoed in the <h1> element.

1. Update the i18n properties to introduce the XSS vulnerability by changing its content to the following:

greeting=You entered: {message|s}

Notice the trailing |s in the curly braces; this instructs Dust.js not to do any escaping of the content, instead rendering the content as given. *This, by the way, is the key to the XSS vulnerability.*

## Start the vulnerable site

1. Restart the website with npm start and test that the form is behaving properly; that is, that it is echoing the contents of the message form variable.
2. Now, it's time to put on our attacker hat. Try entering a message value that includes some HTML, like Boo <em>Hoo!</em> and notice what happens when its rendered. You should see something like the following:



The browser is being given literal HTML from the Dust template, so it interprets the HTML verbatim! What do you suppose would happen if, instead of <em>, an attacker used <script> tags? Yes, you guessed it. They could get the browser to execute arbitrary JavaScript! Let's try it.

## Creating an HTML attack page

All our attacker needs to do now is to get a potential victim to click on a link that looks something like http://localhost:8000/echo?message=Hi&lt;script src=&#39;http://localhost:8000/js/evil.js&#39;&gt;&lt;/script&gt;. As you might expect, once the victim clicks on the link, the browser will see the <script> element, then load and execute the script.

1. Create a standalone HTML page at lusca-test/puppies.html, demonstrating a way an attacker might convince a victim to click on the link. Make its content the following:

<html>  
<body>  
 <h2>Click <a href="http://localhost:8000/echo?message=Hi&lt;script src=&#39;http://localhost:8000/js/evil.js&#39;&gt;&lt;/script&gt;">here</a> if you LOVE PUPPIES!</h2>  
</body>  
</html>

While this example uses localhost:8000 for both the target of the attack and the attacker's JavaScript file, a real example would tell the browser to load the script from some other location, like http://evil.com/js/evil.js.

Any naïve victims who love puppies will certainly click the link.

Let's create an evil JavaScript page that will simply put up an alert indicating that the target site has been hacked.

1. Create the file lusca-test/public/js/evil.js and make its content the following:

alert("You've been duped!");

## View the attack result

1. Restart the web server, then open the file lusca-test/puppies.html in a browser. It should look like the following:



Go ahead & click the link, since you are clearly naïve and love puppies so much. You should see the results of our evil JavaScript file's execution:



As you can see, the browser received literal HTML and simply executed it!

## Close the XSS hole

Now, let's fix our site to no longer be susceptible to the XSS vulnerability.

1. Change the file lusca-test/locales/US/en/echo/index.properties to force HTML escaping:

greeting=You entered: {message|s|h}

1. Restart the server, reopen lusca-test/puppies.html in a browser, and click on the link again. This time, we should see something similar to this:



Now, the browser is no longer executing the HTML, because our Dust template is ensuring that it is HTML encoding any characters in the message value that would be interpreted literally by the browser!

What's the moral of this story? *Make sure that you're escaping content that you don't want the browser to execute!*

## Add a controller to test content security policies (CSP)

Next, we're going to control from which sources our web site can download JavaScript files, fonts, styles, etc.

1. Stop the web server and issue the command yo kraken:controller csp, which will generate a model, controller, view, and i18n bundle in their respective places.

## Update the i18n CSP bundle

1. Open the i18n bundle lusca-test/locales/US/en/csp/index.properties and change its content to the following:

greeting=Hello, CSP!

## Update the CSP view

1. Open the Dust template lusca-test/public/templates/csp/index.dust to contain the following:

{>"layouts/master" /}  
  
{<body}  
 <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>  
 <script type="text/javascript" src="js/csp-detect.js"></script>  
{/body}

Notice that this template is including an external jQuery script and another script from the originating website, <http://localhost:8000/js/csp-detect.js>.

## Create the CSP detection script

1. Create a new file called lusca-test/public/js/csp-detect.js and set its contents to the following:

'use strict';  
  
function detect() {  
 var not;  
 if (window.$) {  
 not = '';  
 } else {  
 not = 'NOT ';  
 }  
 document.write('<h1>External script ' + not + 'loaded!</h1>');  
}  
detect();

As you can see, this script will detect whether or not the jQuery script was loaded and render a message saying so.

## View default CSP behavior

If the appsec property's csp property evaluates to false, then there is no restriction on the sources that the browser is allowed to download from.

1. Restart the web application (npm start) and open the URL http://localhost:8000/csp, which will render the view that includes jQuery and our own csp-detect.js. You should see something similar to the following:



As you can see, the browser successfully loaded the external jQuery script as well as our own csp-detect.js script, which rendered the message above. When you see this message, move on to the next step.

## Restrict loading to the origin server

Now, we're going to instruct the browser that we only want it to load scripts from the originating website.

1. Update the csp entry of the appsec property to contain the following content:

{ "policy": { "default-src": "'self'" } }

This causes the Lusca middleware to include the following HTTP header in all responses:

Content-Security-Policy: default-src 'self';

This HTTP header instructs the browser to only allow scripts, styles, fonts, etc. to be loaded from the originating website. Let's see this in action.

## Elicit script downloading protection

1. Restart the web server now that you've made the change to CSP configuration and open http://localhost:8000/csp in the browser. You should see something similar to the following:



The inclusion of the HTTP Content-Security-Policy header caused the browser to skip loading jQuery from the external site!

When you see this response, you've completed this lab!