

Website Security Research Project

Instructional Documentation

CS 467 - Summer 2021

Team Members:

Cody Medaris

Ray Franklin

Emanuel Ramirez

## Introduction

The purpose of this project is to demonstrate common web based vulnerabilities and to teach defenses against those vulnerabilities. The website is hosted locally and should be restricted from being hosted on the internet due to the many vulnerabilities.

**To start:** You should navigate to localhost after setting up the environment as instructed in the Installation Documentation. You will be greeted by a listing of the various vulnerabilities as different links. Simply navigate to the page of your choice and read about that vulnerability below.

Example of the index page for navigating:

# Vulnerable Site

## Vulnerability 1: Injection

## **Vulnerability 2: Broken Authentication**

## **Vulnerability 3: Sensitive Data Exposure**

## Vulnerability 4: XML External Entities

## Vulnerability 5: Broken Access Control

## Vulnerability 6: Security Misconfiguration

## Vulnerability 7: Cross-Site Scripting - Vulnerable

## Vulnerability 7: Cross-Site Scripting - Secured

## **Vulnerability 8: Insecure Deserialization**

## **Vulnerability 9: Using Components With Known Vulnerabilities**

## **Vulnerability 10: Insufficient Logging & Monitoring**

*Introduction Image 1: Index home page:*

## **Vulnerability 1: Injection**

### **HTML INJECTION:**

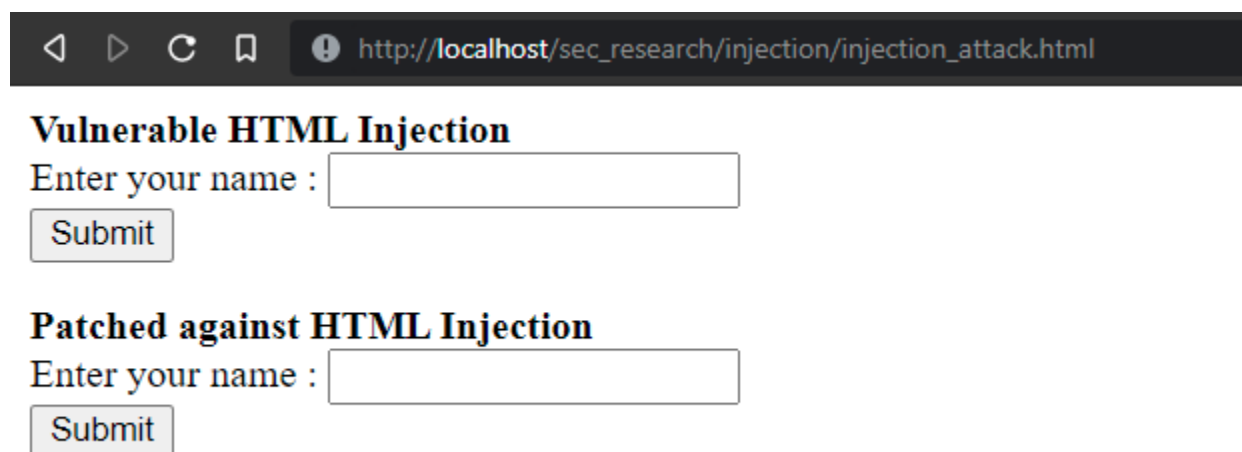
#### **How it works:**

Infection attacks work by inserting external data, often malicious data, to an application which then interprets and executes that data. Applications that do not validate or sanitize user input are vulnerable to injection attacks. In this project we are going to be focusing on SQL and HTML injection attacks. In SQL injection attacks, the attacker will try to manipulate the SQL queries used in the web application in order to gain direct access to the data stored in the database. This can put at risk private information from the user of the web application. On the other hand, HTML attacks are typically used to manipulate the web application interface by inserting unwanted images or messages into the main interface of the web application.

#### **In the site**

In the injection section of the website there is a html page that asks for the user's name. One form (the top-most one in *image1*) is a vulnerable form that does not validate nor sanitizes the user's input. Hence the form is vulnerable to any kind of attack, especially HTML injection attacks. In this case the user can insert the following html line `<p style=%22color:red;margin-left:20px;font-size:80pt;>YOU HAVE BEEN HTML INJECTED</p>`, shown in *image2*, in the vulnerable form. The result of this query can be seen in *image3*. The vulnerable code is shown on *image4*.

*HTML Injection Image 1: Injection attack dashboard*



**Vulnerable HTML Injection**  
Enter your name :

**Patched against HTML Injection**  
Enter your name :

*HTML Injection Image 2: attacker input containing and HTML <p> tag*



**Vulnerable HTML Injection**  
Enter your name :

**Patched against HTML Injection**  
Enter your name :

*HTML Injection Image 3: HTML injection result*



*HTML Injection Image 4: Vulnerable piece of code*

```
injection > submit.php
1  <?php
2  // This is vulnerable to an HTML attack, as it doesn't validates
3  // nor sanitizes user's input.
4  echo "Your name is: " . $_GET['name'];
5  ?>
```

### Defenses:

In order to prevent HTML injection attacks the developer of the web application must validate and sanitize any user input. Having a helper function to remove any possible HTML tags present in the user input will suffice in this case. *Image 5* shows the code behind the patch for

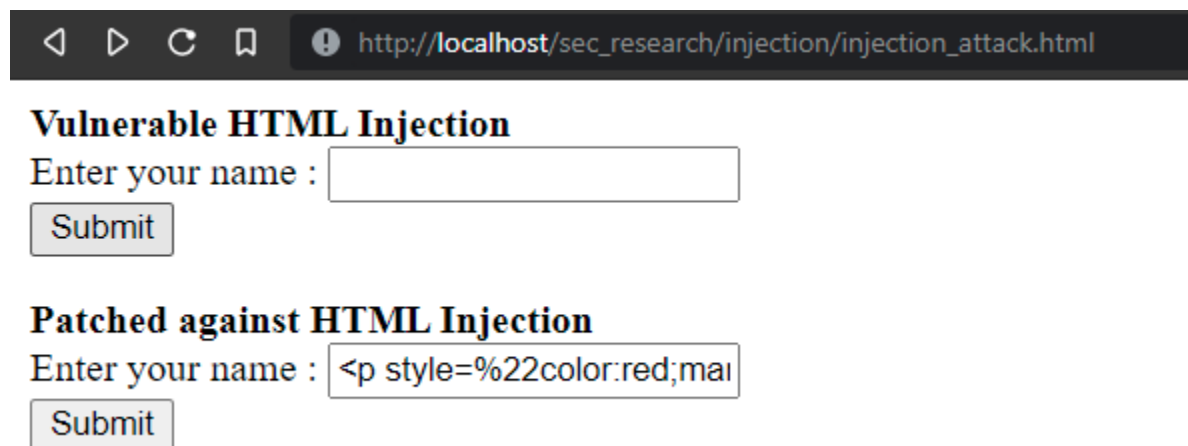
Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

the above presented HTML injection attack and *image 6* and 7, show the patched input validation result.

*HTML Injection Image 5: Patched HTML injection code*

```
injection > submit_patched.php
1  <?php
2      $name = $_GET['name'];
3      echo "Your name is: ".remove_tags_from_name($name);
4
5      // Helper function that validates and sanitizes the user input
6      function remove_tags_from_name($name) {
7          $fixed_name = strip_tags($name);
8          if (strlen($fixed_name) == 0) {
9              return "ERROR PARSING NAME";
10         } else {
11             return $fixed_name;
12         }
13     }
14  ?>
```

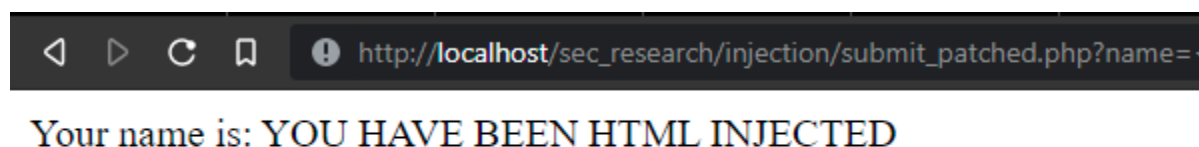
*HTML Injection Image 6: HTML user input in patched HTML vulnerability form*



**Vulnerable HTML Injection**  
Enter your name :

**Patched against HTML Injection**  
Enter your name :

*HTML Injection Image 7: Query result showing the user HTML input as a string which doesn't render the website usability like it does on image 3*



Your name is: YOU HAVE BEEN HTML INJECTED

## Vulnerability 2: Broken Authentication Attack

### How it works:

If an application is vulnerable to broken authentication attacks an attacker can gain access to another user account. Without a proper authentication system, users using public machines to log in to the website will evidently log out of the session once done using the website. But if the authentication system is not properly implemented then another person or attacker can simply go back to the page and the user account will be accessible. Another potential source of attack is allowing the user to have a weak password without any numerical character and not having a minimum amount of characters for password creation. This way an attacker can use a brute



Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

force attack and have the user's credential in a matter of minutes if the password is weak enough.

#### **In the site:**

In the vulnerable page there is a request submit form to register a user. Once registered it will direct the user to the user account page. There, the user can log out of the application. But because the user is using a application that is vulnerable to broken authentication; after logging out the user can simply go back to the previous page and the user, supposedly logged out session, will be running and logged in, with the same session id, as if the user never logged out

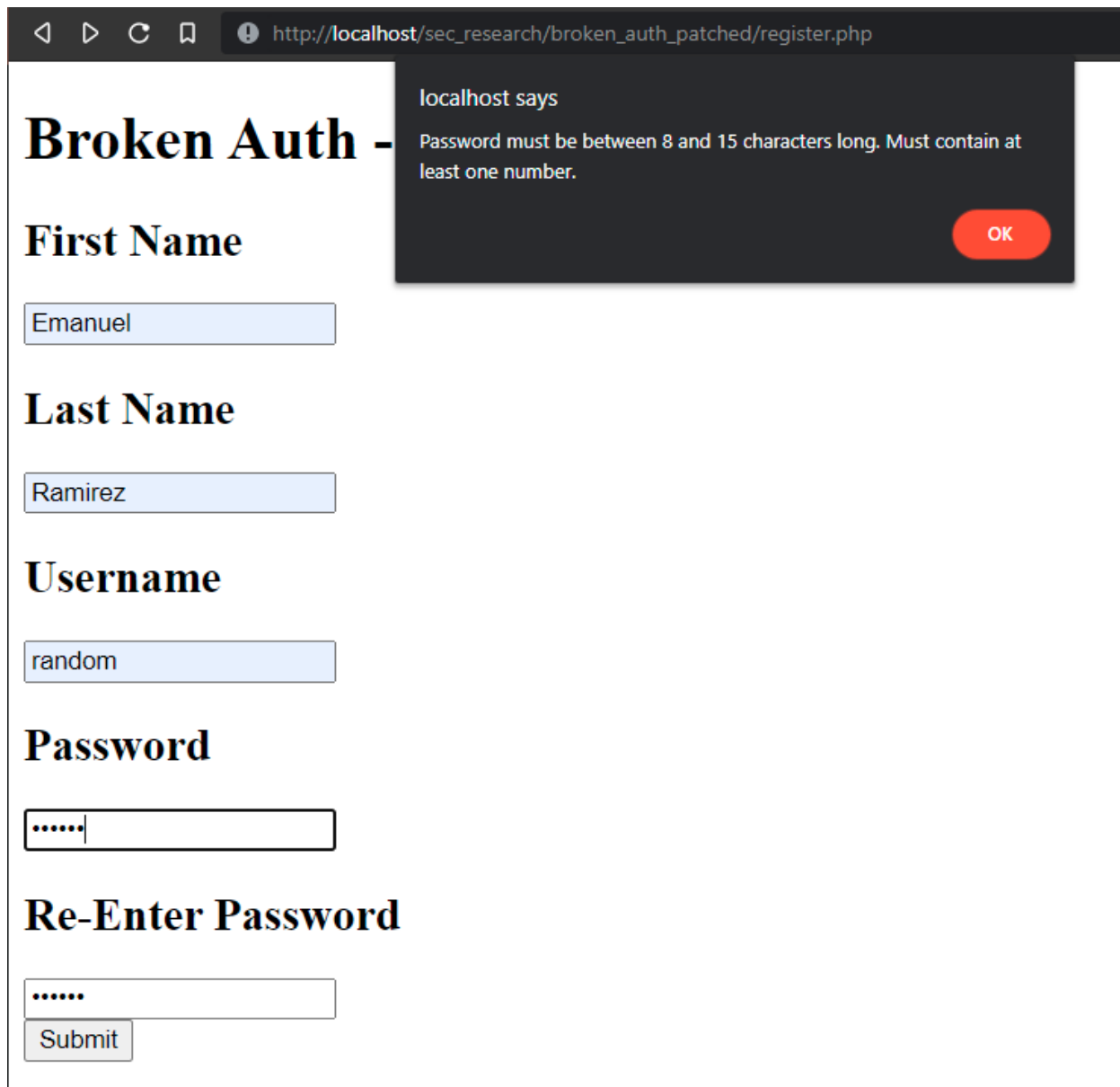
## **Broken Auth - Vulnerable**

**Signed in as: test\_username**

**Log out**

Session ID: bl9754mdn7a48uv8sfikp69okc

*Broken Authentication Image 1: User test\_username logged in with password 'admin'*



The screenshot shows a web browser window with the address bar displaying `http://localhost/sec_research/broken_auth_patched/register.php`. The page title is "Broken Auth -". The form contains the following fields and elements:

- First Name**: Input field containing "Emanuel".
- Last Name**: Input field containing "Ramirez".
- Username**: Input field containing "random".
- Password**: Input field containing ".....".
- Re-Enter Password**: Input field containing ".....".
- Submit**: A button at the bottom of the form.

A JavaScript alert box is displayed over the form, titled "localhost says". The message inside the alert reads: "Password must be between 8 and 15 characters long. Must contain at least one number." There is an "OK" button in the bottom right corner of the alert box.

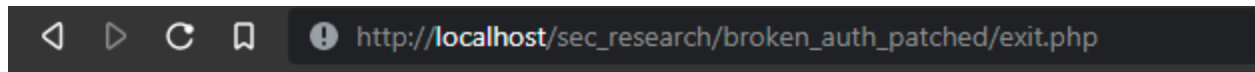
*Broken Authentication Image 2: User tried to register using 'admin' as password on the secured portion of the application*



*Broken Authentication Image 3: User random logged in*



*Broken Authentication Image 4: User random left the page and came back without logging out and the session ID is different from the previous one.*



## Home

### You have been logged out!

*Broken Authentication Image 5: User random logged out and at this point a log-in is required to get back the user account page.*

#### Defenses:

The simplest way to prevent attackers from brute forcing a user's password is to require a password composed of at least 8 characters and at least a digit on registration. Further requirements could be made like requiring at least one capital letter to be present in addition to the previously mentioned requirements. This should at least make it harder for an external attacker to brute force your user's password. Another source of broken authentication is user's session regeneration on page refresh. Having the session id be regenerated every time the user refreshes is a good way to prevent attacks. Last but not least, logging out the user completely out of the application when requested should be a priority. *Image 8* shows a simple script on how to accomplish this.

```
if (!(/[a-z A-Z]*\d[a-z A-Z]*/.test(document.forms['yes']['pwd0'].value)) || !(/^\{8,15}$/.test(document.forms['yes']['pwd0'].value)) {  
    alert("Password must be between 8 and 15 characters long. Must contain at least one number.");  
    return false;  
}
```

*Broken Authentication Image 6: Script to request at least 8 characters and one digit on password*

```
broken_auth > submit.php
1  <?php
2      session_start();
3      $_SESSION['name']=$_GET['uname'];
4  ?>
5
```

*Broken Authentication Image 7: In the vulnerable section of the application every time the user account home is refreshed this script is run, thus the user session is initialized with the same session id. This even persists when user's log out. This is bad.*

```
broken_auth_patched > submit.php
1  <?php
2      session_start();
3      session_regenerate_id();
4  if (isset($_POST['uname'])) {
5      $_SESSION['name'] = $_POST['uname'];
6  }
7  else {
8      if (!isset($_SESSION['name'])) {
9          header('location:exit.php');
10     }
11 }
12 ?>
13
```

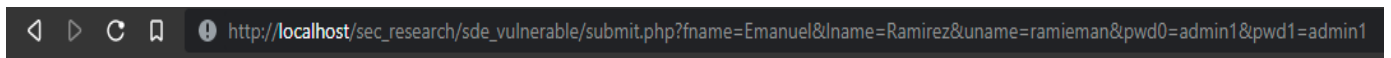
*Broken Authentication Image 8: In the secured portion of the application when the user account page is refreshed the session id is regenerated and if the user has logged out then redirect the page to a logged out page (exit.php). This is better.*

### Vulnerability 3: Sensitive Data Exposure

#### How it works:

Sensitive data exposures in a website put the user's data at risk and developers should ensure proper handling of the user's data. Exposing the user's data should be avoided at any cost.

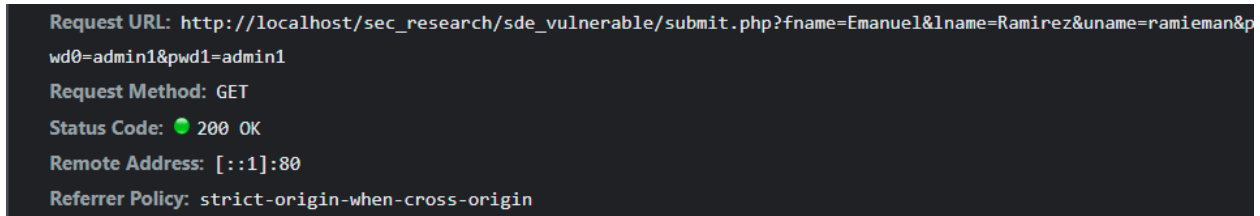
#### In the site:



## Sensitive Data Exposure - Vulnerable

**Signed in as: ramieman**

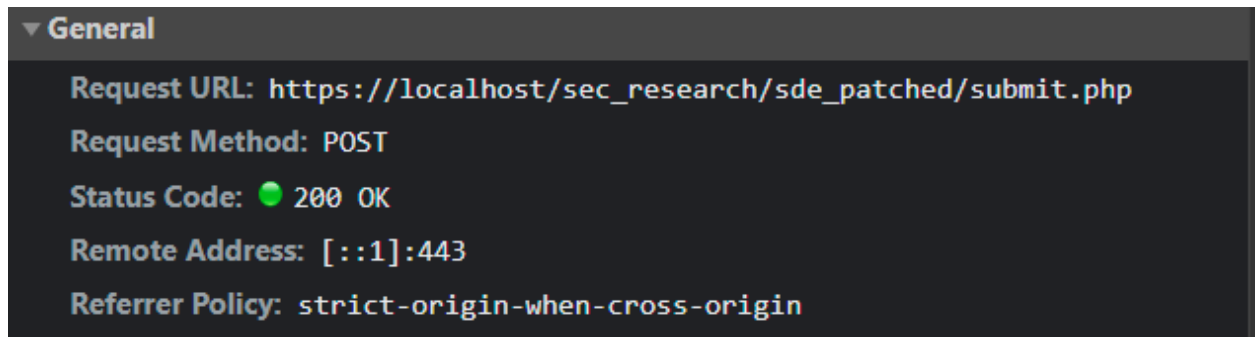
*Sensitive Data Exposure Image 1: the user registration information is displayed as part of the url exposing user data. This is bad.*



*Sensitive Data Exposure Image 2: The network data showing the Request Method as GET and the request url containing all the user registration details. This is bad.*



*Sensitive Data Exposure Image 3: The user registration details are not displayed as part of the request url. Also, https is selected as the preferred protocol instead of http. If http is not being used it will show Not secure to the user. This is better.*



*Sensitive Data Exposure Image 4: The network data showing the Request Method as POST and the request url does not contain any registration details. This is better.*

### Defenses:

The best way to submit form data is to use the POST method instead of the GET method.

```
<form action='submit.php' method='get' class='formplace' name='yes' onsubmit='return validate()>
```

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

*Sensitive Data Exposure Image 5: User submit form using GET method. This is bad.*

```
<form action='submit.php' method='POST' class='formplace' name='yes' onsubmit='return validate()'
```

*Sensitive Data Exposure Image 6: User submit form using POST method. This is better.*

## Vulnerability 4: XML External Entities (XXE) Attack

### How it works:

XML allows entities to be defined within a DOM structure, and be referenced later on in the structure. Using the SYSTEM command, the content of a file somewhere else on the machine can be used as the content of the XML entity. An XXE Vulnerability arises whenever an adversary can define and use their own XML entities to access resources stored on the server in normally inaccessible areas.

### In the site:

In the site, there is a page called 'faux-definition.php' that has a mock functionality of obtaining a definition of a word by submitting an XML request that looks similar to the following:

```
<? xml version="1.0" encoding="UTF-8"?>
```

```
<define>
```

```
<word> myword </word>
```

```
</define>
```

The correct usage would be through a script like:



```
<script>
function subdata() {
    var xml = "<?xml version='1.0' encoding='UTF-8'?><define><word>word</word></define>"
    var req = new XMLHttpRequest();
    req.open("POST", "/tests/dumbpage.php", true);
    req.setRequestHeader("Content-Type", "text/xml");
    req.addEventListener("load", function() {
        console.log(req.responseText)
    });
    req.send(xml);
    event.preventDefault();
}
</script>
```

*XML External Entities (XXE) Attack Image 1: XML request.*

To get a result like:

Sending server: <?xml version='1.0'	<a href="#">randpage.html:10</a>
encoding='UTF-8'?><root><thing>word</thing></root>	
Server sent back: definition of word	<a href="#">randpage.html:14</a>

*XML External Entities (XXE) Attack Image 2: XML result.*

But if an attacker wanted, they could use an external entity to attempt to access the server's htpasswd file by using the following script, found in the 'xxe-attack.html' file:

```
<script>
function subdata() {
    var xxe = "<!DOCTYPE foo [<!ENTITY xxe SYSTEM '../../.htpasswd\" >]]>"
    var xml = "<?xml version='1.0' encoding='UTF-8'?>"+xxe+"<root><thing>&xxe;</thing></root>"
    var req = new XMLHttpRequest();
    console.log("Sending server: "+xml);
    req.open("POST", "/tests/dumbpage.php", true);
    req.setRequestHeader("Content-Type", "text/xml");
    req.addEventListener("load", function() {
        console.log("Server sent back: "+req.responseText);
    });
    req.send(xml);
    event.preventDefault();
}
</script>
```

*XML External Entities (XXE) Attack Image 3: XML attack script.*

And obtain the following result in their browser after executing:

Server sent back: user1:password1 user2:password2 user3:password3 is not a word

#### *XML External Entities (XXE) Attack Image 4: Browser result.*

By defining the entity xxe using SYSTEM ../../.htpasswd and submitting it to the server, when the server parses the XML, it creates the entity xxe and places the contents of the .htpasswd file within the entity, even though the file is located outside of the public directory of the site. The site then displays an error message that reveals the content of the file to the attacker.

#### **Defenses:**

The best method of defense against XXE attacks is to make sure the XML parser you use is configured to not expand external entities. The XML parser used in the site, simplexml, actually has external entities turned off by default, and had to be enabled using the LIBXML\_NOENT argument in order to demonstrate the attack. So we can just change this line of code:

```
$xml = simplexml_load_string($postData, 'SimpleXMLElement', LIBXML_NOENT);
```

#### *XML External Entities (XXE) Attack Image 5: XXE parsing enabled*

To:

```
$xml = simplexml_load_string($postData, 'SimpleXMLElement');
```

#### *XML External Entities (XXE) Attack Image 6: XXE parsing disabled*

We will see that our attack no longer works as the simplexml\_load\_string function returns False if it tries to parse XML with an external entity in it:

Server sent back: is not a word

#### *XML External Entities (XXE) Attack Image 7: Server response with XXE parsing disabled*

Another thing that doesn't really fix the vulnerability, but at least wouldn't easily expose the sensitive info, would be to make sure the error sent back from the server doesn't go too in depth. So I'll change the error message so that it doesn't say "[this word you sent] isn't a word" to something more generic like "no definition known for that word":

```
if (!ctype_alpha($thing1)) {  
    echo "$thing1 is not a word";  
}
```

*XML External Entities (XXE) Attack Image 8: Too descriptive error message*

```
if (!ctype_alpha($thing1)) {  
    echo "Sorry, looks like what you sent isn't a known word";  
}
```

*XML External Entities (XXE) Attack Image 9: More vague error message*

## **Vulnerability 5: Broken Access Controls**

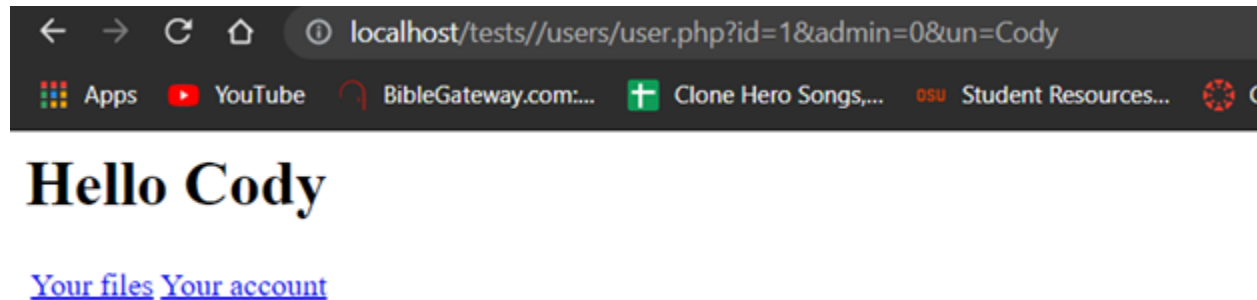
### **How it Works:**

Every business needs to ensure that only trusted individuals can access its sensitive information, and to ensure that personal information remains private. This is done through access control. Sometimes, though, users who shouldn't have access to certain information or privileges may be able to obtain those privileges or imitate a different user who does have those privileges.

### **In the site:**

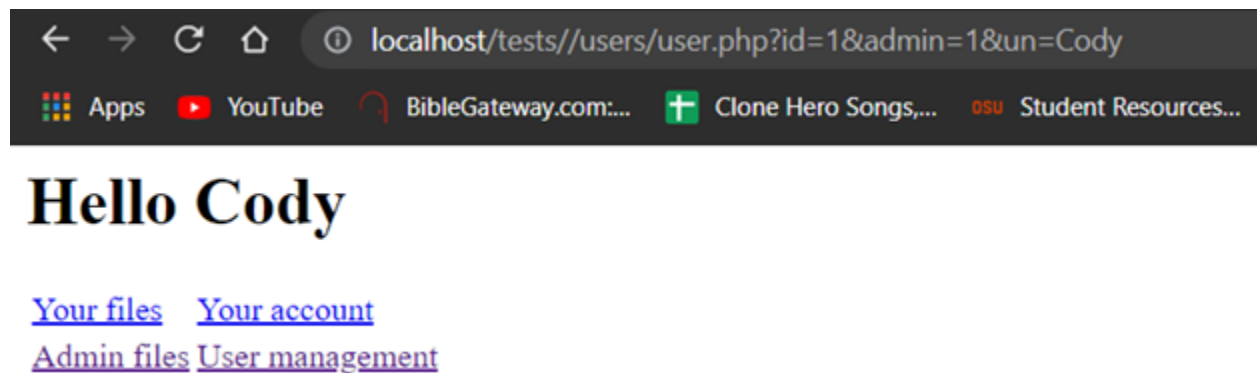
The site contains two examples of broken access control. The first and most easily exploitable is that a user's credentials are submitted to the /user.php page through a query in the URL. So an attacker can easily change the query from something like

/user.php?id=123&admin=0 to /users?id=123&admin=1 to gain access to administrative privileges, or to /user.php?id=456&admin=0 to gain access to another person's information. An example of what a regular user's page should look like:



*Broken Access Controls Image 1: Regular user page.*

Notice that the query parameter 'admin' in the URL is 0. If I go to the address bar and change it to 1, I get access to admin pages:

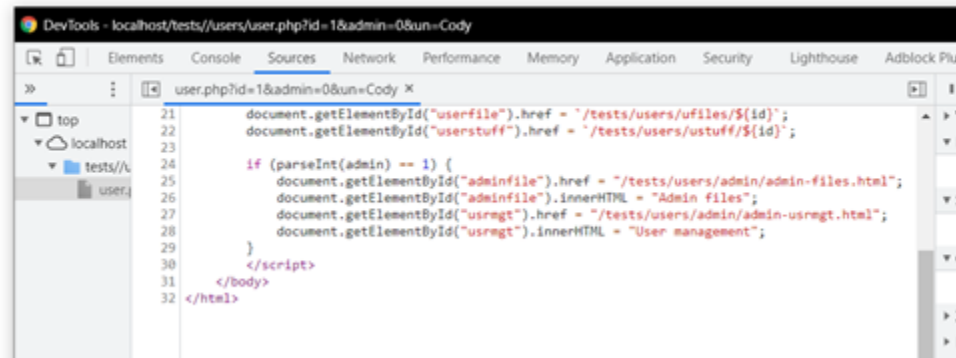


*Broken Access Controls Image 1: Admin page.*

The other issue is that the links for pages containing functionality meant for admins are given away by the javascript code used to add links to those pages to the admin's home page. Anyone visiting /user.php could open their browser's dev tools and see those links:

## Hello Cody

[Your files](#) [Your account](#)



*Broken Access Controls Image 3: Admin links.*

Here I see that /users/admin/admin-files.html and /admin-usrmt.html would be linked on the admin page, so I simply type the URL localhost/bac/users/admin/admin-usrmt.html into the address bar and mess around with user info.

### Defenses:

Rather than placing the user's id & admin information in the URL to pass the info from one page to another, we're going to store the information in a session instead. This means we will need to include a `session_start()`; line at the start of each page:

```
session_start();
```

*Broken Access Controls Image 4: Declaring a session*

Now the session variables can't be altered by whoever wants to, and if someone tries to visit the user.php page when they aren't logged in, they'll be redirected to the login page:

```
session_start();  
// If URL parameters set, store values. Otherwise, redirect to login page  
if (isset($_SESSION["id"]) && isset($_SESSION["username"]) && isset($_SESSION["admin"])) {  
    $user = $_SESSION["username"];  
    $id = $_SESSION["id"];  
    $admin = $_SESSION["admin"];  
} else {  
    header("Location: http://localhost/bac/better-bac-login.php");  
    exit();  
}
```

*Broken Access Controls Image 5: Redirection code for users who aren't logged in*

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

I also made the admin page its own page rather than dynamically populating the user page with admin links, and if someone tries to visit the page while not logged in as an admin, they get redirected to the login page as well (this is also the case if someone tries to visit the admin-files or usr-mgt pages):

```
session_start();  
// If someone comes to the page who isn't logged in as admin, redirect to login page  
if (isset($_SESSION["admin"])) {  
    if ($_SESSION["admin"] != 1) {  
        header("Location: http://localhost/bac/better-bac-login.php");  
        exit();  
    }  
} else {  
    header("Location: http://localhost/bac/better-bac-login.php");  
    exit();  
}
```

*Broken Access Controls Image 6: Redirection on admin pages*

Now there is an important consequence of using sessions. PHP by default stores the session ID in a cookie on your machine, which is sent by your browser to the server every time you make an HTTP request. If you aren't using HTTPS, that cookie info is unencrypted, meaning anyone with a packet sniffing application (in this case, WireShark) can see your session id:

```
✓ Cookie: sesh=jj9f43qg22g3ohthm6f78b39r1\r\n  
Cookie pair: sesh=jj9f43qg22g3ohthm6f78b39r1
```

*Broken Access Controls Image 7: Session ID from HTTP request captured by packet sniffer*

The main defense for this lies in obtaining an SSL/TLS certificate for your site in order to use HTTPS, which encrypts the data sent back and forth between browser and server. This can be done in XAMPP, but it requires several changes to config files and installing a certificate on your OS, and I felt it best to leave it out of the final site.

## Vulnerability 6: Security Misconfiguration

**How it works:**

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

Security misconfigurations can happen in a variety of ways. Maybe you password protected a directory but you used the easily guessed username admin and a password of, well, password. Or maybe you didn't disable directory indexing and so the whole world can see all the files on your server. Not to mention the improper handling of untrusted input that can lead to Injection attacks, XXE attacks, and XSS attacks.

### In the Site:

When you think about it, all the vulnerabilities in the site can be considered the results of security misconfigurations. Rather than repeat what has already been mentioned in those vulnerabilities' write-ups, I wanted to focus on the ways that directories and files can be left exposed to anyone who wants to see them. If we go to the security misconfiguration page of the site, we see something like:

---

## Index of /tests/sec-mis

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 <a href="#">Parent Directory</a>		-	
 <a href="#">home-page.html</a>	2021-07-15 21:25	76	
 <a href="#">scripts/</a>	2021-07-15 21:21	-	
 <a href="#">secretstuff/</a>	2021-07-15 21:23	-	

*Security Misconfiguration Image 1: Exposed directories.*

Instead of displaying the home-page.html file like intended, the user gets to see the files and directories in the sec-mis folder, including ones that are meant to be secret. This gives an attacker a glimpse into the structure of the site, and if the "secretstuff" directory is unprotected (and it is), they could get access to sensitive information:

---

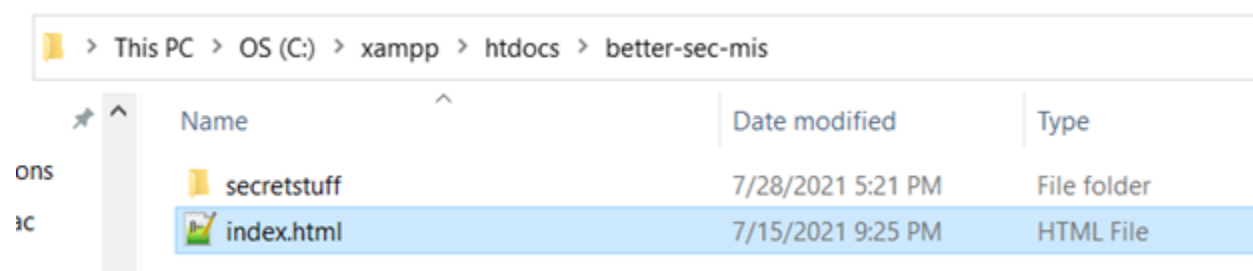
## Index of /tests/sec-mis/secretstuff

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 <a href="#">Parent Directory</a>		-	
 <a href="#">secret.txt</a>	2021-07-08 01:47	18	
 <a href="#">userlog.txt</a>	2021-07-15 21:23	36	

*Security Misconfiguration Image 2: Unprotected files.*

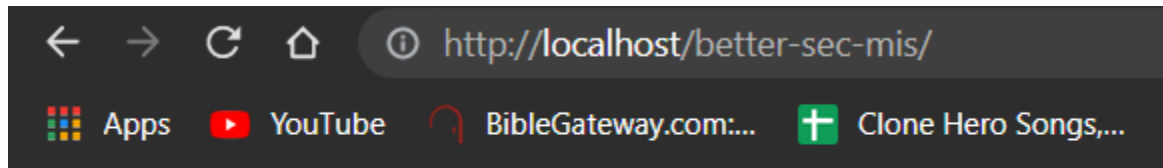
### Defenses:

To disable indexing for our sec-mis directory, we need to put an index file in the directory. There are several different names that can be used depending on the server's config file, but the most common name for the index file is simply index.html. Once we place an index.html file in the directory, when we navigate to the directory, the index file is served instead of an index listing:



*Security Misconfiguration Image 3: Creation of index.html*





## No secrets to be found here

*Security Misconfiguration Image 4: Index.html is displayed when visiting the directory*

Now, while it's great that the files in the site are no longer exposed to prying eyes, if someone were to guess the path to the files we want to protect, they could still view those files. We need to take additional measures to prevent that from happening.

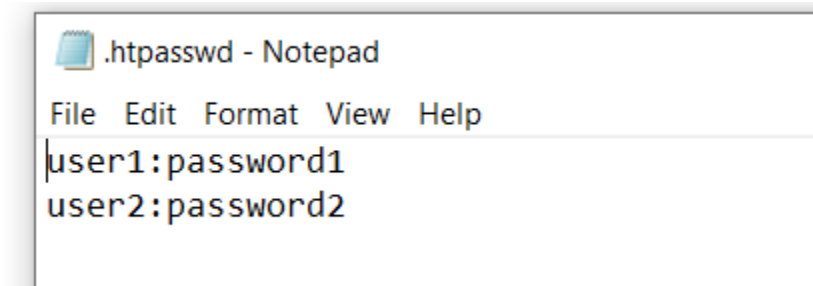
To prevent access to a certain directory (in our case the secretstuff directory), we have to add some content to our server's configuration file. To deny access to everything in a directory, we can use the following block:

```
<Directory "C:/xampp/htdocs/better-sec-mis/secretstuff">  
    Require all denied  
</Directory>
```

*Security Misconfiguration Image 5: Denying access to a directory in apache config file*

Now anyone attempting to access any files in the secretstuff folder will be met with a 403 Forbidden page.

If we want to allow some users to access the directory and forbid everyone else, we first need to create a file that stores the usernames and passwords of the people we want to allow access to. This file should be named `.htpasswd` and should be saved somewhere outside of the site's public folders so it can't be accessed by the visitors to the site. The file must have a format like the following:



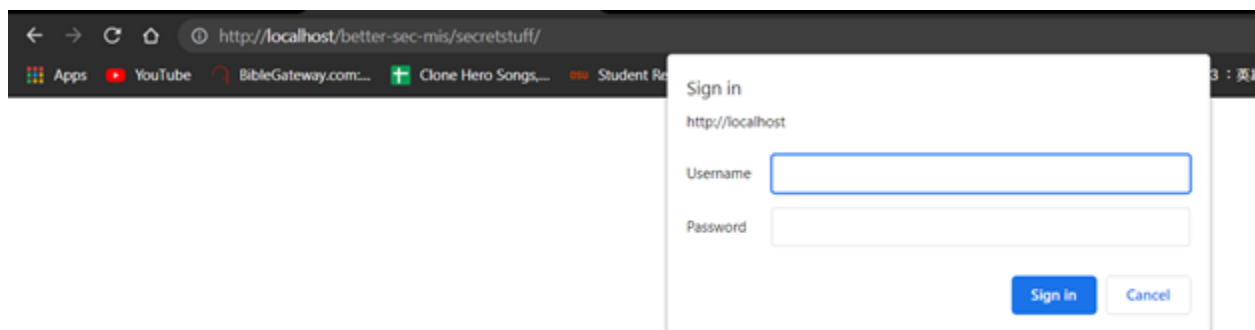
*Security Misconfiguration Image 6: Example .htpasswd file contents*

Then we can use the following block in the config file:

```
<Directory "C:/xampp/htdocs/better-sec-mis/secretstuff">  
    AuthType Basic  
    AuthName "Secrets"  
    AuthUserFile "C:/xampp/.htpasswd"  
    Require user user1 user2  
</Directory>
```

*Security Misconfiguration Image 7: Permitting access only to user1 and user2*

With this block in the config file, anyone attempting to access a file in the secretstuff directory will be prompted to supply their credentials, and only user1 and user2 (as defined in the .htpasswd file) can access the directory:



*Security Misconfiguration Image 8: Server requests credentials when trying to access secretstuff directory*

## **Vulnerability 7: Cross Site Scripting (XSS )**

### **How it Works:**

Cross site scripting is a common vulnerability where the attacker injects script based code into a website. The browser interprets the code and executes the attacker's script. There are two basic styles of XSS and one lesser common version.

- **Stored XSS Attacks:** This occurs when the script is stored on the server and the script is run each time the data is retrieved from that location..
- **Blind Cross-Site Scripting:** Similar to a stored attack, but the stored information is only triggered from the backend application, usually when accessed by an administrator.
- **Reflected XSS Attacks:** This is an attack that is non-persistent. It is not stored in the server, but rather it is sent to the server then executed once the code returns to the client. Often the attacker will use a malicious link via email, the goal is to trick the user into clicking this link, which will send the script to the server and execute when reflected back to the user.

### **In the Site:**

There are two pages for cross-site scripting examples, one vulnerable and one secured. The vulnerable page contains a search field and the user can enter any text they like into it. Here we see the site and it's displayed hint.

## Please search below.

You searched for:

*Hint, try searching for: `<script>alert('XSS attack successful!')</script>`*

**This website was made for educational purposes.**

See below for further reading-

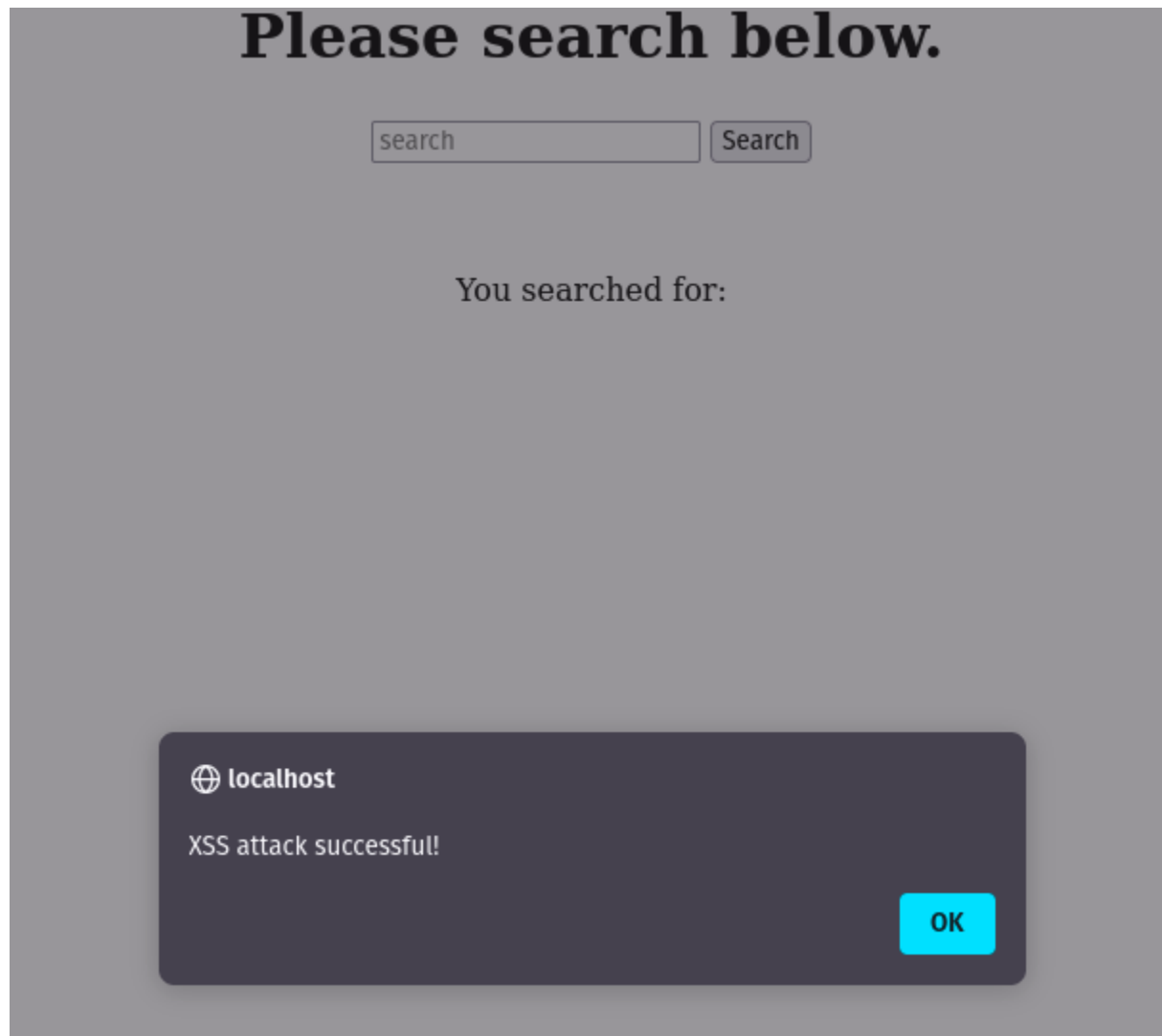
- OWASP: <https://owasp.org/www-community/attacks/xss/>
- Wikipedia: [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)

*XSS Image 1: Vulnerable search field.*

When the user enters a request in the field, the browser reads the request and interprets it accordingly. Usually this would be simple text, but in our case we're going to use the browsers built-in ability to interpret script commands. Here's the input we will use.

```
<script>alert('XSS attack successful!')</script>
```

The browser sees the script tag and executes the code within. Here's the exploited attack succeeding:



*XSS Image 2: Successful attack.*

Instead of displaying the text that we searched for, when we pressed the search button, the code was executed by the browser. Now that we know the browser will run the script we enter, it could be rewritten to contain malicious code that steals user data like session cookies and would allow the attacker to login as that user. There are a lot of other creative and damaging ways an attacker could successfully craft an attack to capitalize on this script vulnerability.

## **Defenses:**

*Detection:* Any user input from a web application will be subject to XSS attacks unless proper actions are taken to secure the input. This means the vast majority of inputs will be vulnerable by default, and preventative measures will need to be taken. You can perform some simple tests by sending one of the many filter evasion methods found on OWASP's website. It will provide a large list of possible scripts to try and see if your site is vulnerable. Effectively, if the script successfully runs, your site is not secure.

*Solution:* The best method to prevent XSS is by sanitizing inputs. This is performed by utilizing various functions built into the coding languages themselves. Using these security encoding libraries will help filter out any special characters that allow the attacker to execute scripts. If you must allow some level of special characters in your inputs, it can be done such that you disallow all of them by default and add in only the ones you need to allow. This does open your site to a potential attack, but the likelihood decreases significantly by using this method.

Here's the secured version where the Javascript does not get executed:

## Please search below.

You searched for: `<script>alert('XSS attack successful!')</script>`

*Attack didn't work? Check out [OWASP](#) it has a list of all common filter evasion methods.*

### **This website was made for educational purposes.**

See below for further reading-

- OWASP: <https://owasp.org/www-community/attacks/xss/>
- Wikipedia: [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)

*XSS Image 3: Secure site.*

The script was entered as previously done, but this time the browser intercepted and changed the request to turn it into benign text. The Javascript alert box was not triggered. This was accomplished by sanitizing the user input before the browser could interpret it.

*Further prevention:* To further prevent threats, the developer should have the mindset that all user data is not to be trusted. If you assume every input from a user is malicious, you may seem a bit paranoid, but the reality is that you will be one of the few who are building secure sites. This does mean every bit of information input into the application will need to be carefully reviewed and controlled. Through completely managing the user inputs before performing any actions with the information you can build a secure application.

## Vulnerability 8: Insecure Deserialization

### How it works:

When transferring data between different architectures in a network, we may need to transform the data to a byte stream so it can be easily transferred and stored. This process known as serialization helps standardize the communication between applications, architectures, and provides an effective means to store data in memory, a database, or a file.

This serialization process is not without its drawbacks. It must be reversed in order to be effective. This is simply called deserialization. During the process, if the data is exposed it could be edited to change the outcome. This occurs when a would-be attacker sends object data through a network to an end point that deserializes the data. As with many attacks, the vulnerability occurs when the user inputs data that the developer would not want or shouldn't trust.

### In the site:

We have two pages again, one insecure and one secure. On the insecure page, there are two text entry boxes with corresponding buttons.

---

## Insecure Deserialization:

Enter a user name in the first box to create a user object.  
Then enter the serialized data into the second box. Change the 0 to 1 in b:0; to see the admin status change.

*Insecure Deserialization Image 1: Insecure site.*



Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

Entering a user name in the first field will create an object with the entered username. It will also print out the user object in a serialized format so the user can see what would be transferred to another network destination.

## Insecure Deserialization:

Enter a user name in the first box to create a user object.  
Then enter the serialized data into the second box. Change the 0 to 1 in b:0; to see the admin status change.

User name: Bob, is a regular user  
Serialized data: O:4:"User":2:{s:9:"user\_name";s:3:"Bob";s:8:"is\_admin";b:0;}

### *Insecure Deserialization Image 2: User Bob.*

The serialized data shows us the format that will be used to transfer the data. The format takes the following shape: [*one\_letter\_code\_of\_variable:value:*]. For example in our page, we have “O:4” which is an object with 4 as the value. User is the type of object, with value 2. Then ‘s’ for a string of length 9 for the user\_name, and value with a string of length 3 for “Bob”. There’s a string of length 8 for the function is\_admin, and finally a boolean value of 1 or 0.

This serialized object data was created by PHP calling serialize() on the object. When that data is accessed again, PHP will need to call unserialize(). This is exactly what the second entry field performs.

## Insecure Deserialization:

Enter a user name in the first box to create a user object.  
Then enter the serialized data into the second box. Change the 0 to 1 in b:0; to see the admin status change.

User name: Bob, is a regular user  
Serialized data: O:4:"User":2:{s:9:"user\_name";s:3:"Bob";s:8:"is\_admin";b:0;}

*Insecure Deserialization Image 3: Insecure accept with no change.*

Here we have copied the serialized data into the second field and printed the results of the is\_admin method, which checks whether the boolean value is true or false. If you were instead to change some data in the serialized object that is passed through to the unserialize() function, PHP would pass along the changed data as if it were the same. This is the crux of the issue, anytime you unserialize, if the user or anyone unauthorized can access the data, it could be compromised.

## Insecure Deserialization:

Enter a user name in the first box to create a user object.  
Then enter the serialized data into the second box. Change the 0 to 1 in b:0; to see the admin status change.

User name: Bob, is an admin  
Serialized data: O:4:"User":2:{s:9:"user\_name";s:3:"Bob";s:8:"is\_admin";b:1;}

*Insecure Deserialization Image 4: Unauthorized changes.*

Now, we have made the seemingly small change to the boolean value and passed it back through the insecure accept text field. In the background PHP is calling unserialize on the data and see

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

the boolean value as true now, thus we have Bob who was previously not an admin, being escalated to admin status.

Anytime an attacker can edit or intercept the data to be unserialized, this could be the result.

Now that we know what not to do, let's see what is the recommended best practice to prevent this attack.

### **Defenses:**

Preventing the attack is something that must be considered before beginning the development process. The only way to truly stop this type of attack is to not accept serialized objects from untrusted sources. This means the developer will need to consider their architecture and how to design the application so it will not need to expose the serialized data to the attacker.

## **Insecure Deserialization:**

The best defense is said best by OWASP:

"The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types."

Here we've removed the option from the user to enter the serialized data.  
See OWASP for further reading: [10 steps to avoid insecure deserialization](#)

*Insecure Deserialization Image 5: Secured.*

Here we've taken the most straightforward approach. We've removed the option for the user to interact with the application and the unserialize functionality.

If simply removing the interaction from the user is not an option, then there are some methods that can be used to take advantage of serialized data without exposing the application to too much risk.

OWASP lists the following recommendations:

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
- Isolating and running code that deserializes in low privilege environments when possible.
- Log deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitoring deserialization, alerting if a user deserializes constantly.

We see there are some additional methods to secure your application against the insecure deserialization attacks, but this type of attack is becoming more prevalent and our ability to identify the vulnerability has not been fully automated so as we learn better methods of identifying the vulnerable applications we may see this become a much larger issue.

## Vulnerability 9: Using Components with Known Vulnerabilities

Every page of this site could be considered a known vulnerability. We purposefully chose to demonstrate these vulnerabilities for educational purposes. As such, this web application should not be hosted on a publicly facing address as it has many components with known vulnerabilities.

### How it works:

People build the component architecture developers use to create new applications. The complexity and scale of a large application can require many hundreds of people to work on various sections of the application together. Unfortunately, humans make mistakes and bugs are created. These lead to exploits and vulnerabilities that attackers will utilize to their advantage. Since these vulnerabilities are being found continuously, developers work tirelessly to patch the problems out of their programs.

## Using Components with Known Vulnerabilities

This site was purpose built to display known vulnerabilities.  
It is **not** secure and should **not** be hosted publically.

We built this site with XAMPP, Apache Web Server, and MySQL.  
Current number of publicly disclosed cybersecurity vulnerabilities for XAMPP: 17  
Current number of publicly disclosed cybersecurity vulnerabilities for mySQL: 1432  
Current number of publicly disclosed cybersecurity vulnerabilities for Apache: 1874

We used PHP, Javascript, and HTML.  
Current number of publicly disclosed cybersecurity vulnerabilities for Javascript: 3465  
Current number of publicly disclosed cybersecurity vulnerabilities for PHP: 6960  
Current number of publicly disclosed cybersecurity vulnerabilities for HTML: 13,307  
Source: [CVE List Home](#)

*Using Components with Known Vulnerabilities Image 1: Site example.*

We see in the above example that the currently known vulnerabilities we could be invoking from this project alone numbers in the tens of thousands.

There are many tools that can be used to scan your site to determine if it has vulnerabilities. Metasploit is perhaps the most famous of these tools. While it can be used for good in an audit or by a security researcher testing their own site, many attackers have taken advantage of it's powerful database of known exploits. It allows the user to not only scan a site for exploitable vulnerabilities, it can in some cases automate the process so someone with minimal knowledge can successfully exploit a site.

It is seldom that a single vulnerability is used to fully exploit an application. It is often the case that attackers will chain together multiple exploits in as many vulnerabilities to completely reach their goals. Whether it be ransomware, stealing confidential or private information, or simply taking control of a host, it becomes incredibly important to not only correct and prevent some of the vulnerabilities, but to try and remedy all of them for your application.

Xanjero estimated that over 25 million people are still running Windows XP which has not been supported by Microsoft since April 18, 2014. This means that any new vulnerabilities found will be fully exploitable and no longer patched out of the operating system. To prevent this particular problem, you must upgrade and/or migrate your data to secured platforms before products reach

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

end of life. This takes careful planning and sometimes years before you can find a secure solution, this is partially why we still see so many legacy applications using vulnerable software.

### **Defenses:**

The best defense to counteract using components with known vulnerabilities is to use a systematic and organized approach to updating your application components. Patching is an ongoing process and a never ending requirement for a supported application. You will need to continually review and update your application with the latest and greatest patches to ensure you remain secure.

Another step to mitigate potential known vulnerabilities is to ensure your components are coming from a trusted source. This is accomplished by verifying packages are signed which will help ensure the components have not changed from the original source package, and through obtaining components from official sources on secure connections.

One of the more beneficial methods to secure your application is by pruning obsolete programs and libraries from your applications. If the project you are using for your application has not been updated in several years, it should be thoroughly reviewed for potential replacement. Some companies go so far as to write their own redundant software with the patches they want built in, so as to further secure their applications. This isn't feasible for many smaller companies but the cost of a breach or ransomware could completely ruin a company financially, so this risk needs to be addressed and weighed early in the design process.

## Vulnerability 10: Insufficient Logging and Monitoring

**Note:** Since monitoring error and access logs is something that goes on behind the scenes of a website, and is something that must be manually done by a server administrator, it isn't really something that can be demonstrated on a web page. Instead, we offer a description of the vulnerability and some examples of how monitoring logs can identify security flaws in the site.

### How it Works:

Apache provides very detailed error and access logs to record and detail each error the server encounters and each time a page on the server is accessed, respectively. These logs provide crucial information, not only to diagnose server issues, but also to detect suspicious activity or broken security.

As an example, imagine you check your server's access log and see this at the bottom:

```
...1 - - [03/Aug/2021:00:52:44 -0400] "GET /icons/back.gif HTTP/1.1" 200 210 "http://localhost/better-sec-mis/"
::1 - - [03/Aug/2021:00:52:44 -0400] "GET /icons/text.gif HTTP/1.1" 200 229 "http://localhost/better-sec-mis/"
123.122.121.120 - user1 [03/Aug/2021:00:52:47 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 200 1247 "-"
123.122.121.120 - user1 [03/Aug/2021:00:52:50 -0400] "GET /better-sec-mis/secretstuff/secret.txt HTTP/1.1" 200 1247 "-"
::1 - - [03/Aug/2021:00:53:11 -0400] "GET /favicon.ico HTTP/1.1" 404 295 "http://localhost/secure.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0"
::1 - - [03/Aug/2021:00:53:18 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 401 480 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0"
127.126.125.124 - user [03/Aug/2021:00:53:24 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 401 480 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0"
127.126.125.124 - userr [03/Aug/2021:00:53:28 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 401 480 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0"
127.126.125.124 - user_1 [03/Aug/2021:00:53:33 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 401 480 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0"
127.126.125.124 - us3r1 [03/Aug/2021:00:53:46 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 401 480 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0"
127.126.125.124 - user1 [03/Aug/2021:00:55:16 -0400] "GET /better-sec-mis/secretstuff/ HTTP/1.1" 200 1247 "-"
```

### *Insufficient Logging & Monitoring Image 1: Repeated attempts to access a page*

Here you see that the same IP address tried to access a password protected directory 4 times with incorrect credentials (as noted by the 401 status code right after the "HTTP/1.1") before



Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

they finally accessed the directory with the username user1 (noted by the 200 status code). Since you can also see that user1 accessed that directory earlier from a different IP address, this should raise concerns that a malicious actor has accessed user1's sensitive information. While paying attention to the logs doesn't provide a way to prevent something like this from happening, it will alert you to attacks that have happened so that you know that you have existing vulnerabilities.

For another example, imagine you have a script file that is used by pages on the site but contains information that you don't want public, so you take steps to deny access to that file. To test that you've set up your denial correctly, you try to navigate to the file on a web browser and get a 403 Forbidden, so you assume all is well. Later on, you check your access log and see:

```
127.126.125.124 - someuser [01/Aug/2021:22:32:02 -0400] "GET /secretscript.php HTTP/1.1" 200 30 "-" "Mozilla/5.0"
```

*Insufficient Logging & Monitoring Image 2: Successful GET on a sensitive file*

Now you know that you have a security flaw somewhere and the file is still somehow accessible. Without paying attention to these logs, you might have just assumed you had protected the file and left the security flaw unnoticed.

### References for sources used:

1. "OWASP Top Ten," OWASP. [Online]. Available:  
<https://owasp.org/www-project-top-ten/>. [Accessed: 19-Jul-2021].

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez

2. *Cross Site Scripting Prevention Cheat Sheet*. Cross Site Scripting Prevention - OWASP Cheat Sheet Series. (n.d.).  
[https://cheatsheetseries.owasp.org/cheatsheets/Cross\\_Site\\_Scripting\\_Prevention\\_Cheat\\_Sheet.html#Why\\_Can\\_I\\_Just\\_HTML\\_Entity\\_Encode\\_Untrusted\\_Data.3F](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#Why_Can_I_Just_HTML_Entity_Encode_Untrusted_Data.3F).
3. DrapsTV. (2015, January 22). *XSS Tutorial #1 - What is Cross Site Scripting?* YouTube.  
[https://www.youtube.com/watch?v=M\\_nIIcKTxGk](https://www.youtube.com/watch?v=M_nIIcKTxGk).
4. Wikimedia Foundation. (2021, July 10). *Cross-site scripting*. Wikipedia.  
[https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting).
5. *XSS Filter Evasion Cheat Sheet*. OWASP. (n.d.).  
<https://owasp.org/www-community/xss-filter-evasion-cheatsheet>.
6. "PHP serialization format," *Wikipedia*, 05-Jul-2020. [Online]. Available:  
[https://en.wikipedia.org/wiki/PHP\\_serialization\\_format](https://en.wikipedia.org/wiki/PHP_serialization_format). [Accessed: 05-Aug-2021].
7. "A8:2017-insecure deserialization," *OWASP*. [Online]. Available:  
[https://owasp.org/www-project-top-ten/2017/A8\\_2017-Insecure\\_Deserialization](https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization).  
[Accessed: 05-Aug-2021].
8. "Penetration testing Software, pen testing security," *Metasploit*. [Online]. Available:  
<https://www.metasploit.com/>. [Accessed: 05-Aug-2021].
9. O. E. R. IV, "Unbelievable: Millions of people are still running Windows XP, even though it's severely outdated and unsecured," *Xanjero*, 08-Sep-2020. [Online]. Available:  
<https://www.xanjero.com/news/approximately-25-million-pcs-are-still-running-the-unsecured-windows-xp-os/>. [Accessed: 05-Aug-2021].

Website Security Research Project  
Instructional Documentation  
CS 467 - Summer 2021

Team Members:  
Cody Medaris  
Ray Franklin  
Emanuel Ramirez