**ARYAMAN MISHRA**

**19BCE1027**

**SQL Injection Attacks**

XVWA is a badly coded web application written in PHP/MySQL that helps security enthusiasts to learn application security. It’s not advisable to host this application online as it is designed to be “Xtremely Vulnerable”. We recommend hosting this application in local/controlled environment and sharpening your application security ninja skills with any tools of your own choice. It’s totally legal to break or hack into this. The idea is to evangelize web application security to the community in possibly the easiest and fundamental way. Learn and acquire these skills for good purpose. How you use these skills and knowledge base is not our responsibility.

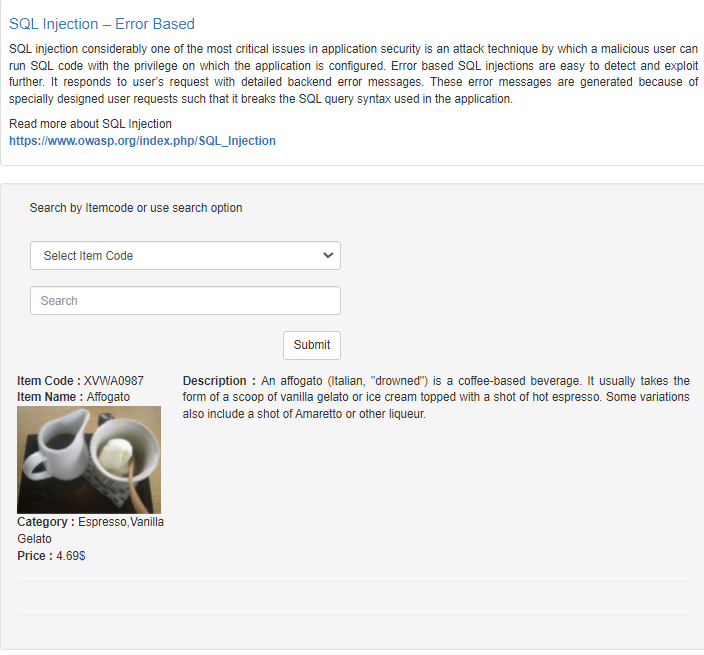
XVWA is designed to understand following security issues.

**SQL Injection – Error Based**

Error-based SQL injection is an In-band injection technique where the error output from the SQL database is used to manipulate the data inside the database. In In-band injection, the attacker uses the same communication channel for both attacks and collect data from the database. This is the easiest and common intrusion technique used by an attacker. You can force data extraction by using a vulnerability in which the code will output a SQL error rather than the required data from the server. This method can be easily automated using grep extract functionality. In many cases, the error generated by the database is enough for the attacker to understand the database entirely.

Enter or search code of item to view details.

If the server responds to this URL with an SQL error, it proves that, the server is connected to the database in an insecure manner. The quote(‘) breaks the SQL syntax. Now, its just a matter of running a few SQL commands to completely collapse/destroy the database.

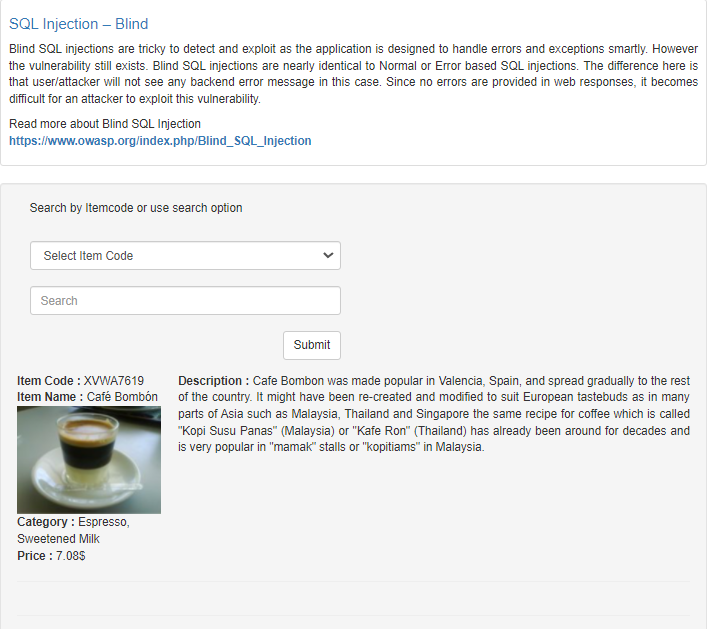


**SQL Injection – Blind**

Blind SQL (Structured Query Language) injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

When an attacker exploits SQL injection, sometimes the web application displays error messages from the database complaining that the SQL Query’s syntax is incorrect. Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the way the data is retrieved from the database. When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible.

Follow same directives as in SQL Injection-Error based.



**OS Command Injection**

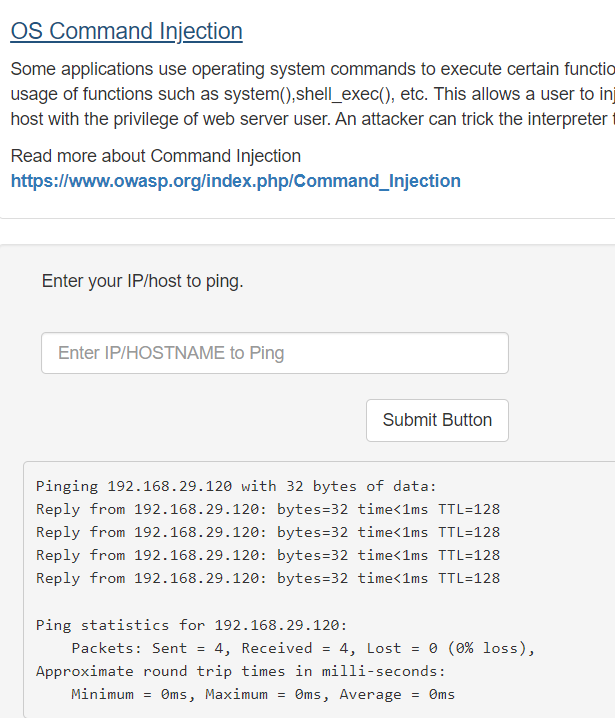
OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data. Very often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.

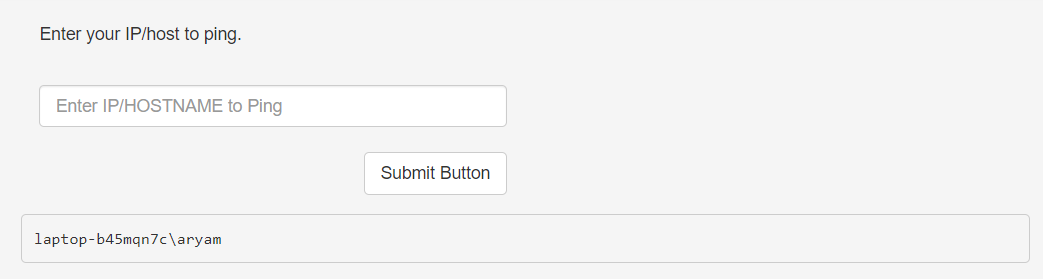
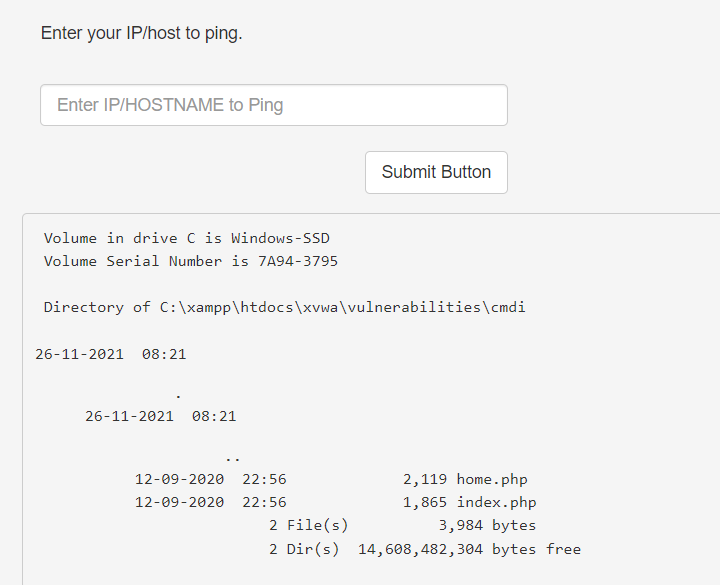
Consider a shopping application that lets the user view whether an item is in stock in a particular store. This information is accessed via a URL like:

https://insecure-website.com/stockStatus?productID=381&storeID=29

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data. Very often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.

Ping home IP to access directories and information on device.





**XPATH Injection**

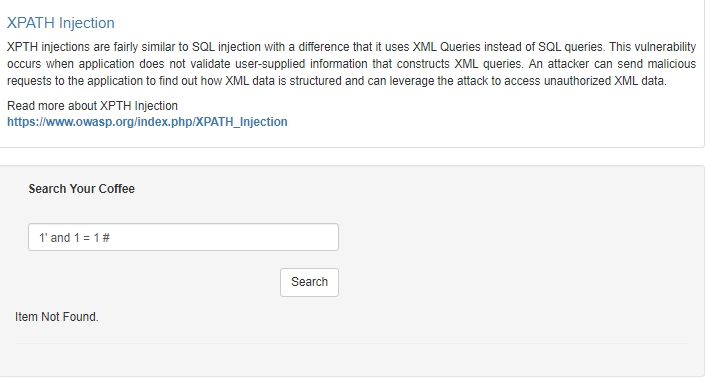
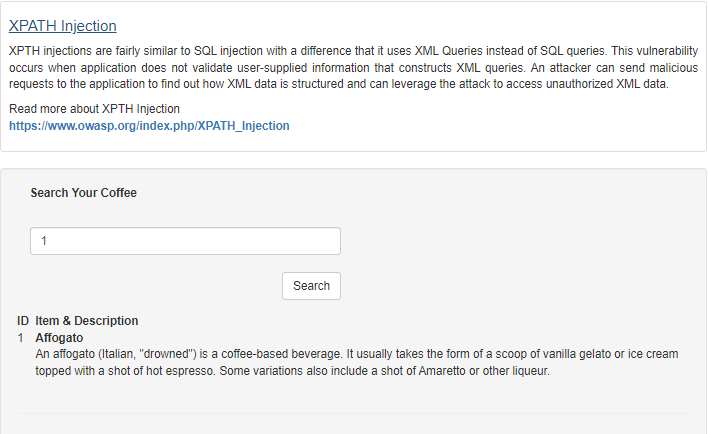
XPath Injection is an attack technique used to exploit applications that construct XPath (XML Path Language) queries from user-supplied input to query or navigate XML documents. It can be used directly by an application to query an XML document, as part of a larger operation such as applying an XSLT transformation to an XML document, or applying an XQuery to an XML document. The syntax of XPath bears some resemblance to an SQL query, and indeed, it is possible to form SQL-like queries on an XML document using XPath. For example, assume an XML document that contains elements by the name user, each of which contains three subelements - name, password and account. The following XPath expression yields the account number of the user whose name is "jsmith" and whose password is "Demo1234" (or an empty string if no such user exists):

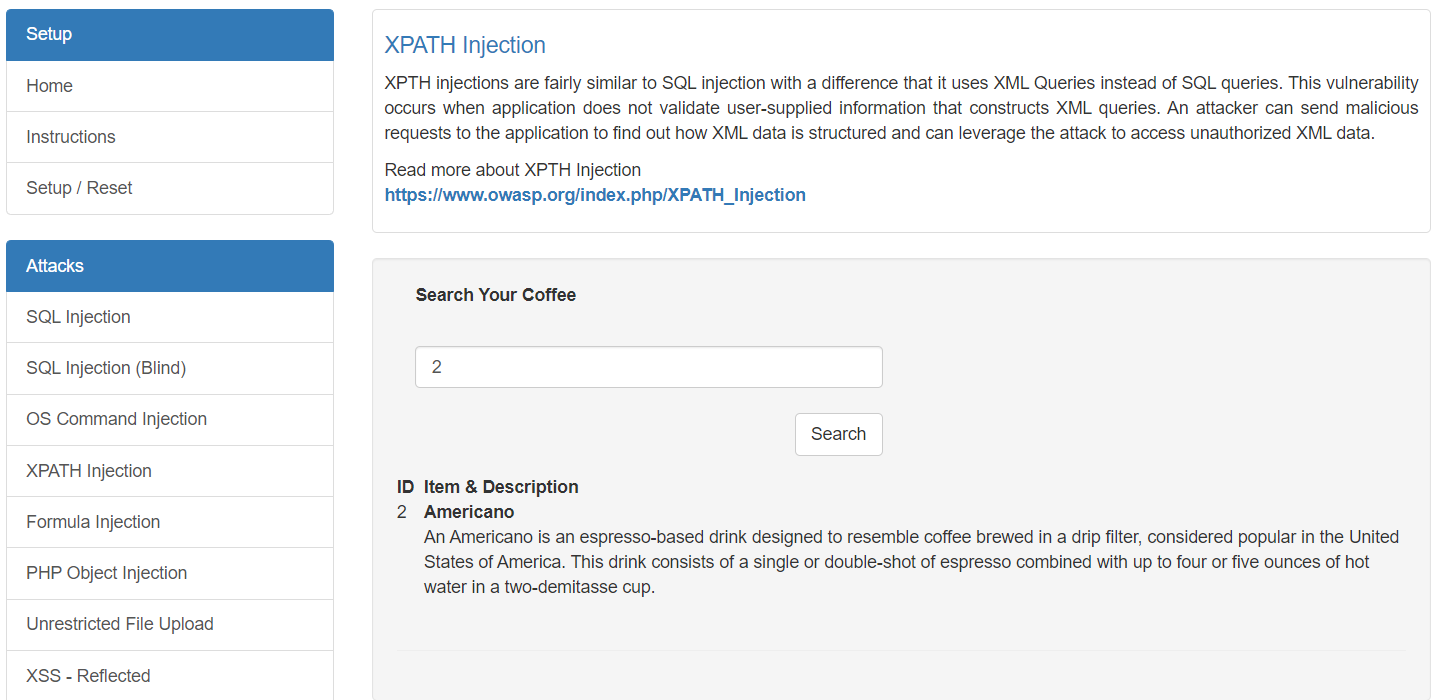
string(//user[name/text()='aryaman' and

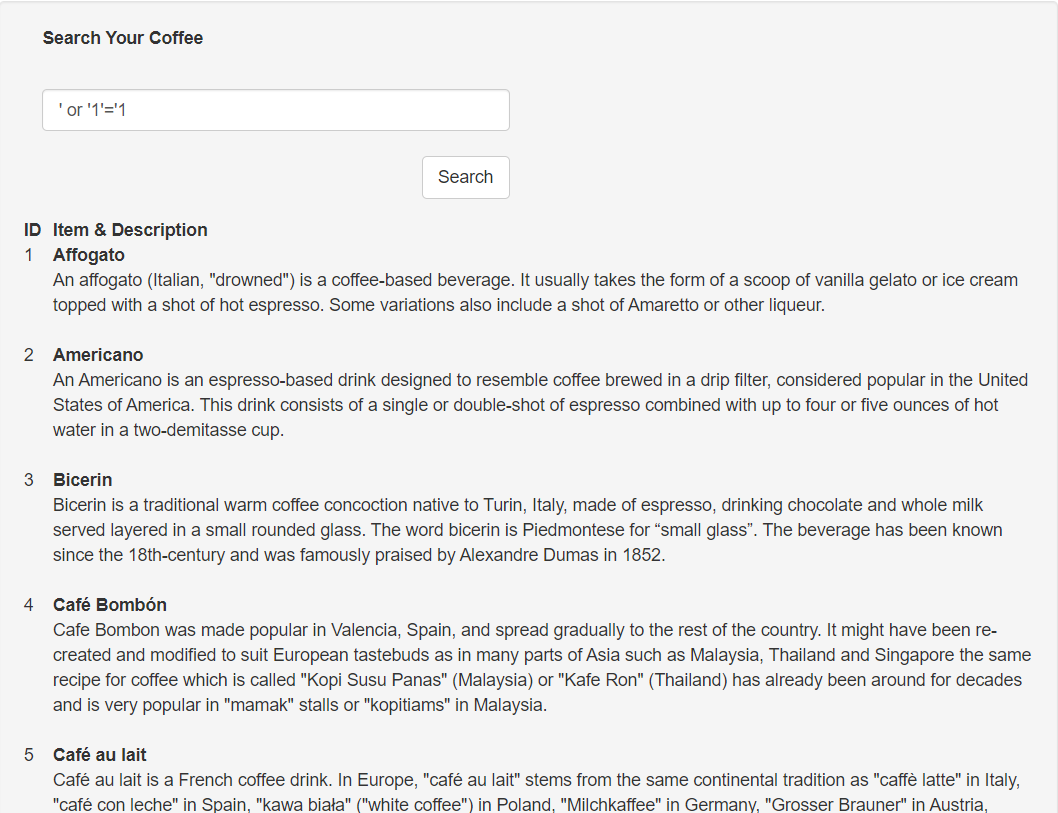
password/text()='admin']/account/text())

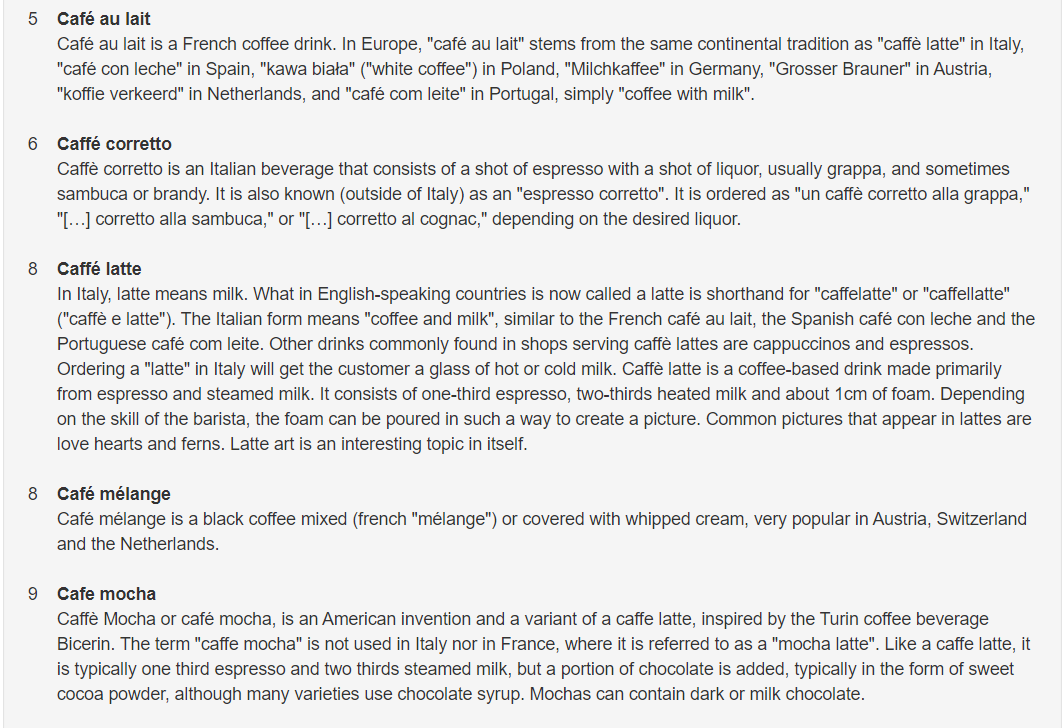
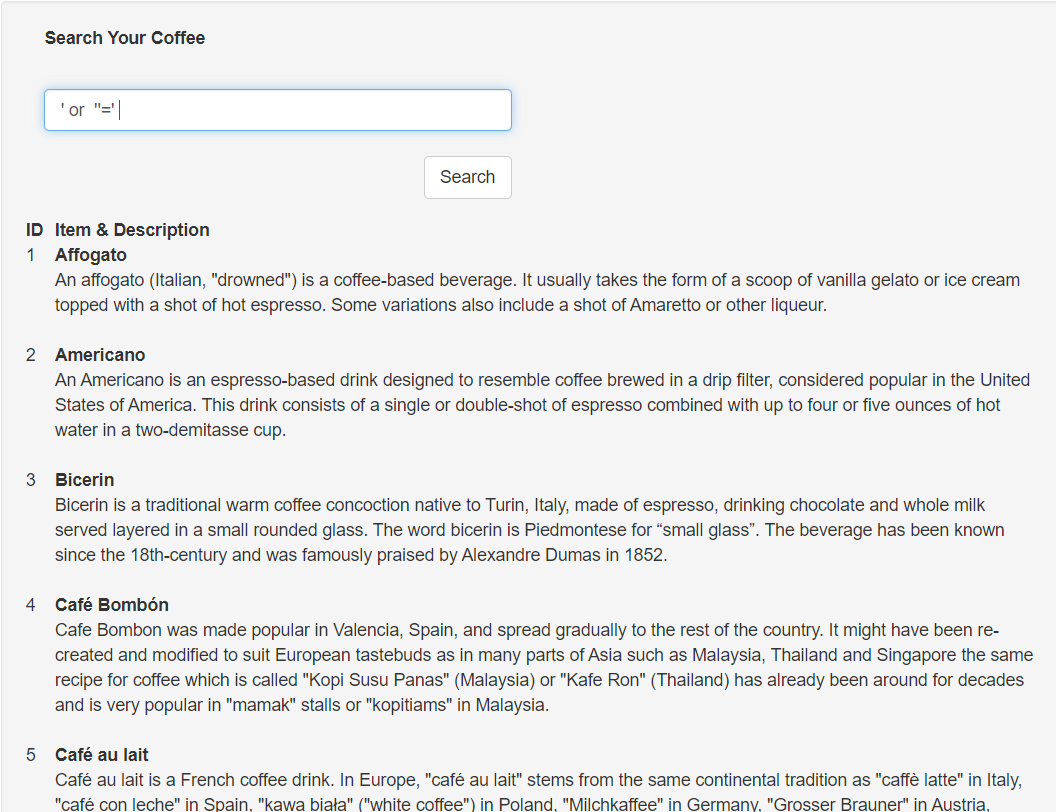
If an application uses run-time XPath query construction, embedding unsafe user input into the query, it may be possible for the attacker to inject data into the query such that the newly formed query will be parsed in a way differing from the programmer's intention.

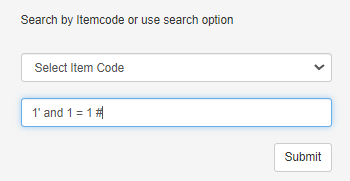
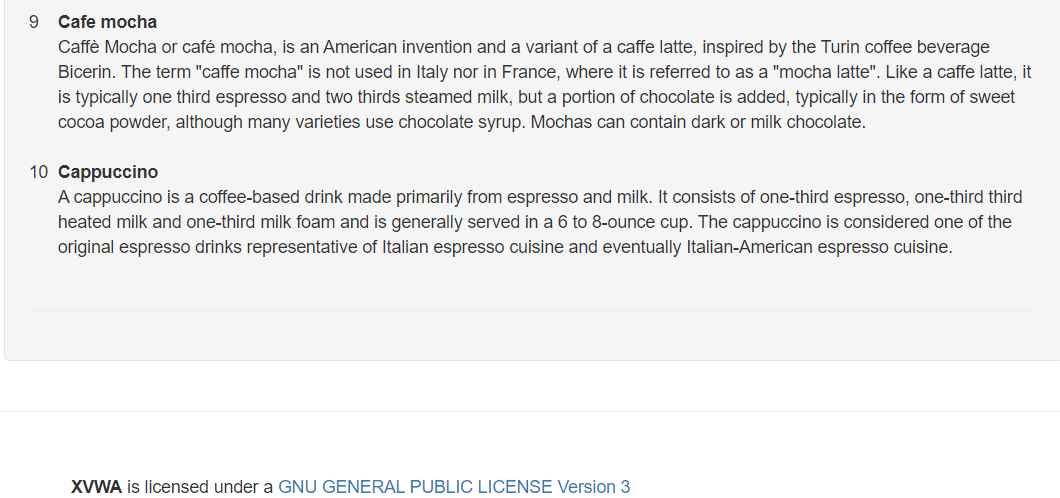
Execute code commands to search for various types of coffees.











**Formula Injection**

CSV Injection, also known as Formula Injection, occurs when websites embed untrusted input inside CSV files.

When a spreadsheet program such as Microsoft Excel or LibreOffice Calc is used to open a CSV, any cells starting with = will be interpreted by the software as a formula. Maliciously crafted formulas can be used for three key attacks:

* Hijacking the user’s computer by exploiting vulnerabilities in the spreadsheet software, such as CVE-2014-3524.
* Hijacking the user’s computer by exploiting the user’s tendency to ignore security warnings in spreadsheets that they downloaded from their own website.
* Exfiltrating contents from the spreadsheet, or other open spreadsheets.

This attack is difficult to mitigate, and explicitly disallowed from quite a few bug bounty programs. To remediate it, ensure that no cells begin with any of the following characters:

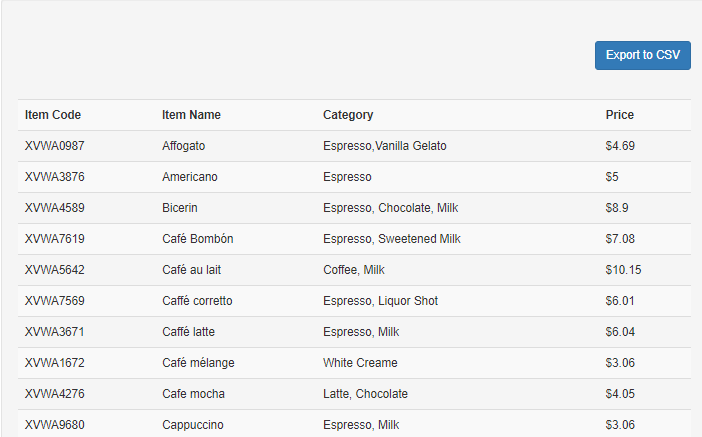
* Equals to (=)
* Plus (+)
* Minus (-)
* At (@)
* Tab (0x09)
* Carriage return (0x0D)

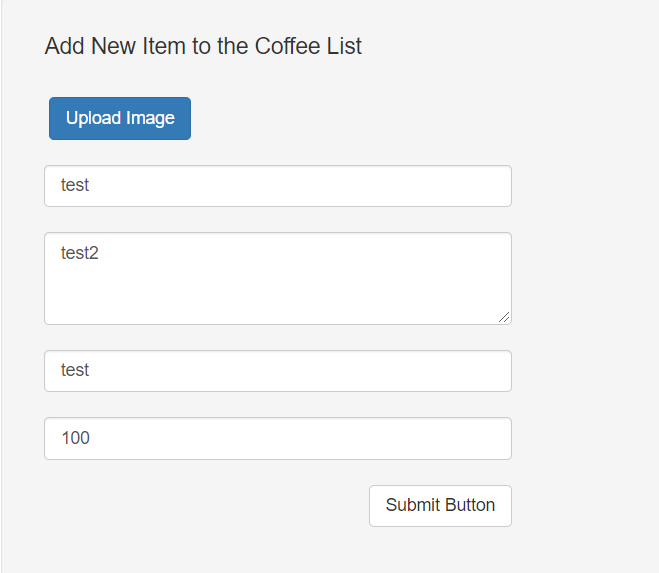
Keep in mind that it is not sufficient to make sure that the untrusted user input does not start with these characters. You also need to take care of the field separator (e.g., ‘,’, or ‘;’) and quotes (e.g., ', or "), as attackers could use this to start a new cell and then have the dangerous character in the middle of the user input, but at the beginning of a cell.

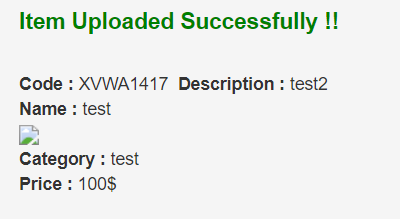
Alternatively, apply the following sanitization to each field of the CSV, so that their content will be read as text by the spreadsheet editor:

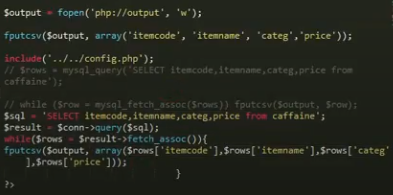
* Wrap each cell field in double quotes
* Prepend each cell field with a single quote
* Escape every double quote using an additional double quote

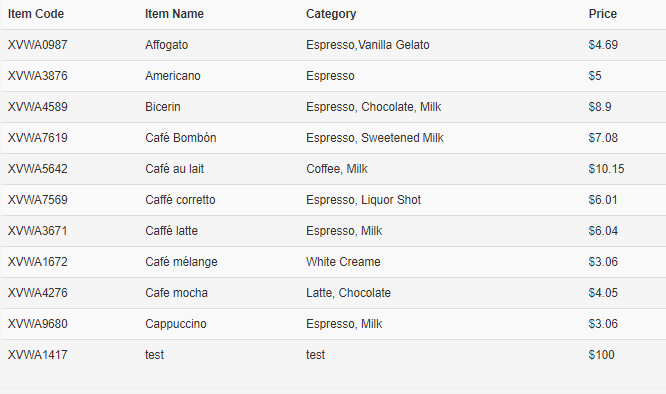
Add another object to database to view in .csv file.

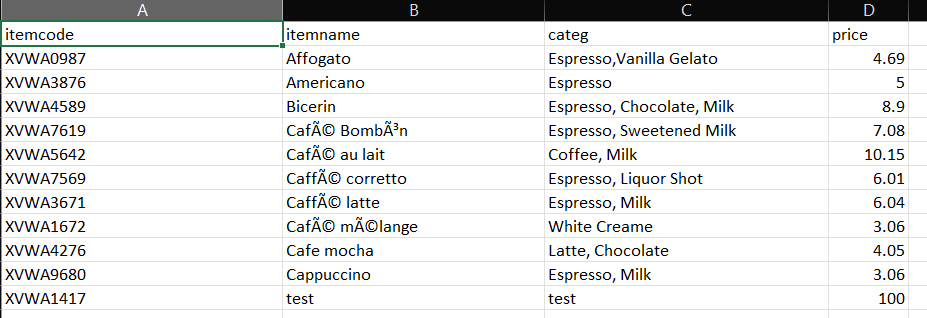












**PHP Object Injection**

PHP Object Injection is an application level vulnerability that could allow an attacker to perform different kinds of malicious attacks, such as Code Injection, SQL Injection, Path Traversal and Application Denial of Service, depending on the context. The vulnerability occurs when user-supplied input is not properly sanitized before being passed to the unserialize() PHP function. Since PHP allows object serialization, attackers could pass ad-hoc serialized strings to a vulnerable unserialize() call, resulting in an arbitrary PHP object(s) injection into the application scope.

In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met:

* The application must have a class which implements a PHP magic method (such as \_\_wakeup or \_\_destruct) that can be used to carry out malicious attacks, or to start a “POP chain”.
* All of the classes used during the attack must be declared when the vulnerable unserialize() is being called, otherwise object autoloading must be supported for such classes.

## Examples

**Example 1**

The example below shows a PHP class with an exploitable \_\_destruct method:

class Example1

{

public $cache\_file;

function \_\_construct()

{

// some PHP code...

}

function \_\_destruct()

{

$file = "/var/www/cache/tmp/{$this->cache\_file}";

if (file\_exists($file)) @unlink($file);

}

}

// some PHP code...

$user\_data = unserialize($\_GET['data']);

// some PHP code...

In this example an attacker might be able to delete an arbitrary file via a Path Traversal attack, for e.g. requesting the following URL:

http://testsite.com/vuln.php?data=O:8:"Example1":1:{s:10:"cache\_file";s:15:"../../index.php";}

**Example 2**

The example below shows a PHP class with an exploitable \_\_wakeup method:

class Example2

{

private $hook;

function \_\_construct()

{

// some PHP code...

}

function \_\_wakeup()

{

if (isset($this->hook)) eval($this->hook);

}

}

// some PHP code...

$user\_data = unserialize($\_COOKIE['data']);

// some PHP code...

In this example an attacker might be able to perform a Code Injection attack by sending an HTTP request like this:

GET /vuln.php HTTP/1.0

Host: testsite.com

Cookie: data=O%3A8%3A%22Example2%22%3A1%3A%7Bs%3A14%3A%22%00Example2%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D

Connection: close

Where the cookie parameter “data” has been generated by the following script:

class Example2

{

private $hook = "phpinfo();";

}

print urlencode(serialize(new Example2));

**Example 3**

This last example shows how it is possible to perform a SQL Injection attack using a “POP chain”, for instance by leveraging a \_\_toString method like this:

class Example3

{

protected $obj;

function \_\_construct()

{

// some PHP code...

}

function \_\_toString()

{

if (isset($this->obj)) return $this->obj->getValue();

}

}

// some PHP code...

$user\_data = unserialize($\_POST['data']);

// some PHP code...

If the $user\_data variable is an “Example3” object and it will be treated like a string somewhere in the code, then its \_\_toString method will be called. Since this method will call the getValue method of the object stored into the “obj” property, it’s possible to set that property to an arbitrary object, and execute its getValue method, if it is accessible, otherwise its \_\_call method will be called, if it is defined. For the sake of ease, just think that when unserialize() is called, the class below is available within the application scope (or supported by autoloading):

class SQL\_Row\_Value

{

private $\_table;

// some PHP code...

function getValue($id)

{

$sql = "SELECT \* FROM {$this->\_table} WHERE id = " . (int)$id;

$result = mysql\_query($sql, $DBFactory::getConnection());

$row = mysql\_fetch\_assoc($result);

return $row['value'];

}

}

In this example an attacker might be able to perform a SQL Injection attack by sending a POST request containing a “data” parameter generated by a script like this:

class SQL\_Row\_Value

{

private $\_table = "SQL Injection";

}

class Example3

{

protected $obj;

function \_\_construct()

{

$this->obj = new SQL\_Row\_Value;

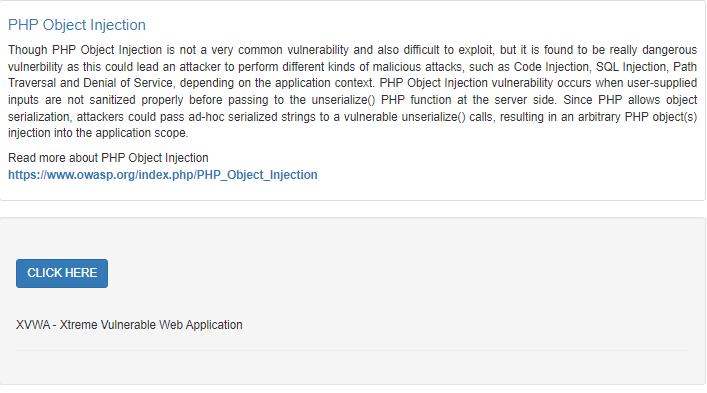
}

}

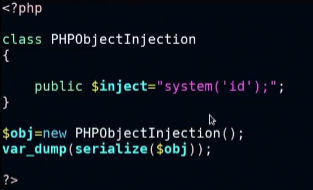
print urlencode(serialize(new Example3));

[http://localhost/xvwa/vulnerabilities/php\_object\_injection/?r=a:2:{i:0;s:4:%22XVWA%22;i:1;s:33:%22Xtreme%20Vulnerable%20Web%20Application%22;}](http://localhost/xvwa/vulnerabilities/php_object_injection/?r=a:2:%7bi:0;s:4:%22XVWA%22;i:1;s:33:%22Xtreme%20Vulnerable%20Web%20Application%22;%7d)

Make code changes to study code output in XVWA Window.

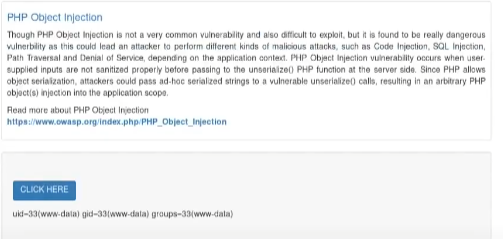




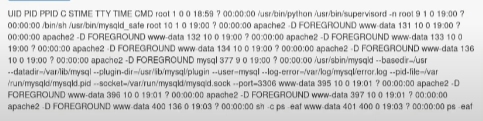












**Unrestricted File Upload**

Uploaded files represent a significant risk to applications. The first step in many attacks is to get some code to the system to be attacked. Then the attack only needs to find a way to get the code executed. Using a file upload helps the attacker accomplish the first step.

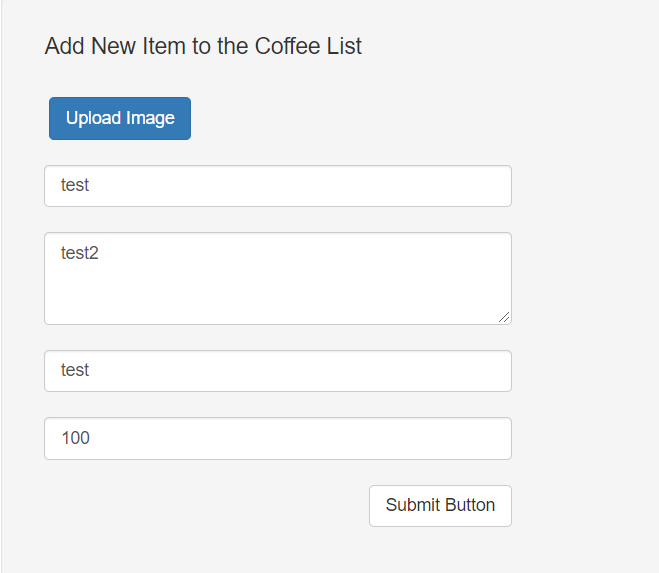
The consequences of unrestricted file upload can vary, including complete system takeover, an overloaded file system or database, forwarding attacks to back-end systems, client-side attacks, or simple defacement. It depends on what the application does with the uploaded file and especially where it is stored.

There are really two classes of problems here. The first is with the file metadata, like the path and file name. These are generally provided by the transport, such as HTTP multi-part encoding. This data may trick the application into overwriting a critical file or storing the file in a bad location. You must validate the metadata extremely carefully before using it.

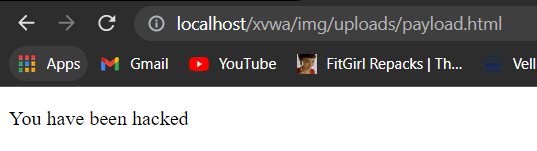
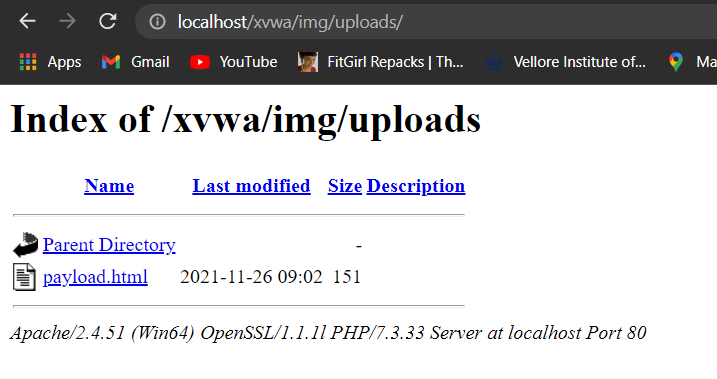
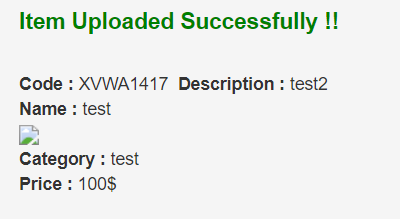
The other class of problem is with the file size or content. The range of problems here depends entirely on what the file is used for. See the examples below for some ideas about how files might be misused. To protect against this type of attack, you should analyse everything your application does with files and think carefully about what processing and interpreters are involved.

## Risk Factors

* The impact of this vulnerability is high, supposed code can be executed in the server context or on the client side. The likelihood of detection for the attacker is high. The prevalence is common. As a result the severity of this type of vulnerability is high.
* It is important to check a file upload module’s access controls to examine the risks properly.
* Server-side attacks: The web server can be compromised by uploading and executing a web-shell which can run commands, browse system files, browse local resources, attack other servers, or exploit the local vulnerabilities, and so forth.
* Client-side attacks: Uploading malicious files can make the website vulnerable to client-side attacks such as XSS or Cross-site Content Hijacking.
* Uploaded files can be abused to exploit other vulnerable sections of an application when a file on the same or a trusted server is needed (can again lead to client-side or server-side attacks)
* Uploaded files might trigger vulnerabilities in broken libraries/applications on the client side (e.g. iPhone MobileSafari LibTIFF Buffer Overflow).
* Uploaded files might trigger vulnerabilities in broken libraries/applications on the server side (e.g. ImageMagick flaw that called ImageTragick!).
* Uploaded files might trigger vulnerabilities in broken real-time monitoring tools (e.g. Symantec antivirus exploit by unpacking a RAR file)
* A malicious file such as a Unix shell script, a windows virus, an Excel file with a dangerous formula, or a reverse shell can be uploaded on the server in order to execute code by an administrator or webmaster later – on the victim’s machine.
* An attacker might be able to put a phishing page into the website or deface the website.
* The file storage server might be abused to host troublesome files including malwares, illegal software, or adult contents. Uploaded files might also contain malwares’ command and control data, violence and harassment messages, or steganographic data that can be used by criminal organisations.
* Uploaded sensitive files might be accessible by unauthorised people.
* File uploaders may disclose internal information such as server internal paths in their error messages.



Upload .html or image file to access in parent directory.



**Reflected Cross Site Scripting**

Reflected cross-site scripting (or XSS) arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Suppose a website has a search function which receives the user-supplied search term in a URL parameter:

https://insecure-website.com/search?term=gift

The application echoes the supplied search term in the response to this URL:

<p>You searched for: gift</p>

Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

https://insecure-website.com/search?term=<script>/\*+Bad+stuff+here...+\*/</script>

This URL results in the following response:

<p>You searched for: <script>/\* Bad stuff here... \*/</script></p>

If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.

 an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can:

* Perform any action within the application that the user can perform.
* View any information that the user is able to view.
* Modify any information that the user is able to modify.
* Initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.

There are various means by which an attacker might induce a victim user to make a request that they control, to deliver a reflected XSS attack. These include placing links on a website controlled by the attacker, or on another website that allows content to be generated, or by sending a link in an email, tweet or other message. The attack could be targeted directly against a known user, or could an indiscriminate attack against any users of the application:

The need for an external delivery mechanism for the attack means that the impact of reflected XSS is generally less severe than stored XSS, where a self-contained attack can be delivered within the vulnerable application itself.

There are many different varieties of reflected cross-site scripting. The location of the reflected data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

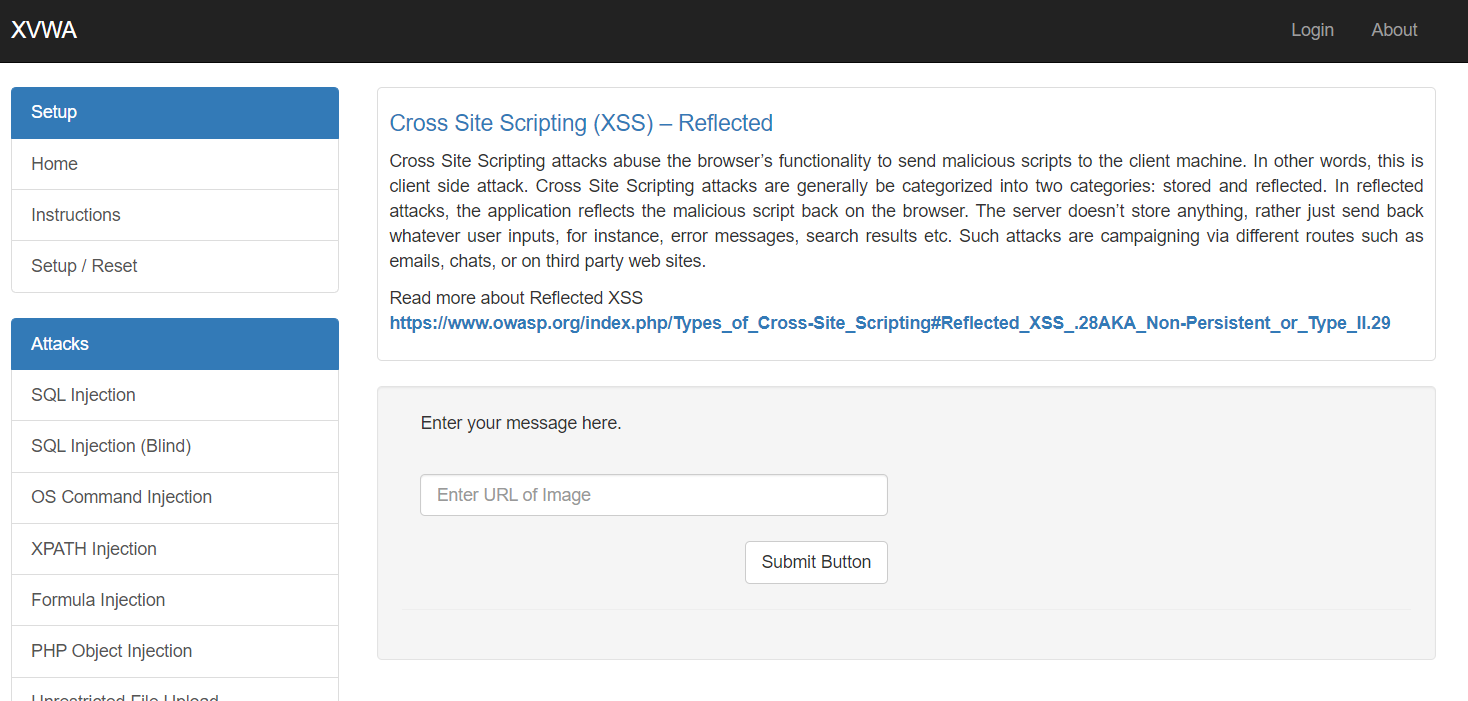
In addition, if the application performs any validation or other processing on the submitted data before it is reflected, this will generally affect what kind of XSS payload is needed.

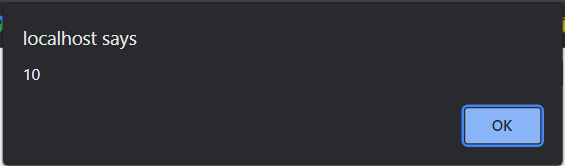
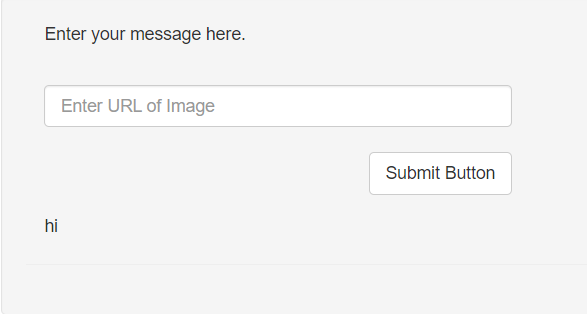
The vast majority of reflected cross-site scripting vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

Testing for reflected XSS vulnerabilities manually involves the following steps:

* **Test every entry point.** Test separately every entry point for data within the application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path. It also includes HTTP headers, although XSS-like behavior that can only be triggered via certain HTTP headers may not be exploitable in practice.
* **Submit random alphanumeric values.** For each entry point, submit a unique random value and determine whether the value is reflected in the response. The value should be designed to survive most input validation, so needs to be fairly short and contain only alphanumeric characters. But it needs to be long enough to make accidental matches within the response highly unlikely. A random alphanumeric value of around 8 characters is normally ideal. You can use Burp Intruder's number payloads [https://portswigger.net/burp/documentation/desktop/tools/intruder/payloads/types#numbers] with randomly generated hex values to generate suitable random values. And you can use Burp Intruder's grep payloads option to automatically flag responses that contain the submitted value.
* **Determine the reflection context.** For each location within the response where the random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.
* **Test a candidate payload.** Based on the context of the reflection, test an initial candidate XSS payload that will trigger JavaScript execution if it is reflected unmodified within the response. The easiest way to test payloads is to send the request to Burp Repeater, modify the request to insert the candidate payload, issue the request, and then review the response to see if the payload worked. An efficient way to work is to leave the original random value in the request and place the candidate XSS payload before or after it. Then set the random value as the search term in Burp Repeater's response view. Burp will highlight each location where the search term appears, letting you quickly locate the reflection.
* **Test alternative payloads.** If the candidate XSS payload was modified by the application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS attack based on the context of the reflection and the type of input validation that is being performed. For more details, see cross-site scripting contexts
* **Test the attack in a browser.** Finally, if you succeed in finding a payload that appears to work within Burp Repeater, transfer the attack to a real browser (by pasting the URL into the address bar, or by modifying the request in Burp Proxy's intercept view, and see if the injected JavaScript is indeed executed. Often, it is best to execute some simple JavaScript like alert(document.domain) which will trigger a visible popup within the browser if the attack succeeds.

Try HTML scripts ot Image URL to reflect in localhost.





**Stored Cross Site Scripting**

Stored cross-site scripting (also known as second-order or persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Suppose a website allows users to submit comments on blog posts, which are displayed to other users. Users submit comments using an HTTP request like the following:

POST /post/comment HTTP/1.1  
Host: vulnerable-website.com  
Content-Length: 100  
  
postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net

After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

<p>This post was extremely helpful.</p>

Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

<script>/\* Bad stuff here... \*/</script>

Within the attacker's request, this comment would be URL-encoded as:

comment=%3Cscript%3E%2F\*%2BBad%2Bstuff%2Bhere...%2B\*%2F%3C%2Fscript%3E

Any user who visits the blog post will now receive the following within the application's response:

<p><script>/\* Bad stuff here... \*/</script></p>

The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application.

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. The attacker can carry out any of the actions that are applicable to the impact of reflected XSS vulnerabilities.

In terms of exploitability, the key difference between reflected and stored XSS is that a stored XSS vulnerability enables attacks that are self-contained within the application itself. The attacker does not need to find an external way of inducing other users to make a particular request containing their exploit. Rather, the attacker places their exploit into the application itself and simply waits for users to encounter it.

The self-contained nature of stored cross-site scripting exploits is particularly relevant in situations where an XSS vulnerability only affects users who are currently logged in to the application. If the XSS is reflected, then the attack must be fortuitously timed: a user who is induced to make the attacker's request at a time when they are not logged in will not be compromised. In contrast, if the XSS is stored, then the user is guaranteed to be logged in at the time they encounter the exploit.

Many stored XSS vulnerabilities can be found using Burp Suite's web vulnerability scanner.

Testing for stored XSS vulnerabilities manually can be challenging. You need to test all relevant "entry points" via which attacker-controllable data can enter the application's processing, and all "exit points" at which that data might appear in the application's responses.

Entry points into the application's processing include:

* Parameters or other data within the URL query string and message body.
* The URL file path.
* HTTP request headers that might not be exploitable in relation to reflected XSS.
* Any out-of-band routes via which an attacker can deliver data into the application. The routes that exist depend entirely on the functionality implemented by the application: a webmail application will process data received in emails; an application displaying a Twitter feed might process data contained in third-party tweets; and a news aggregator will include data originating on other web sites.

The exit points for stored XSS attacks are all possible HTTP responses that are returned to any kind of application user in any situation.

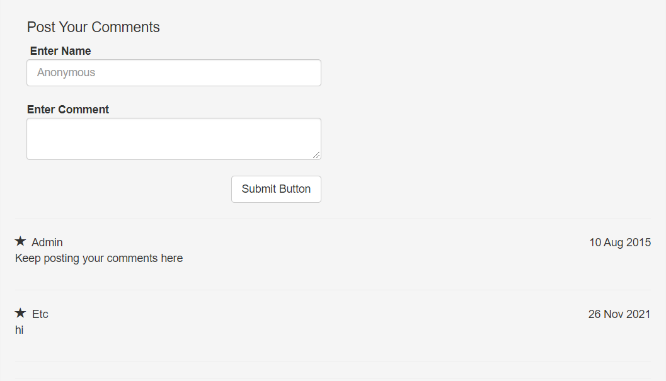
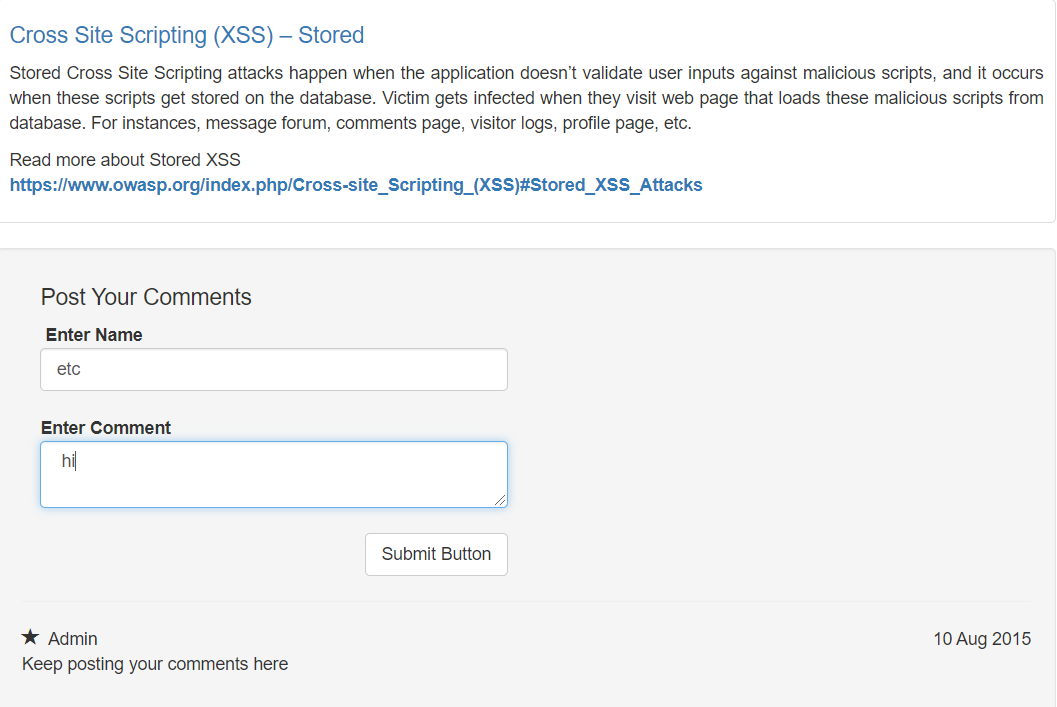
The first step in testing for stored XSS vulnerabilities is to locate the links between entry and exit points, whereby data submitted to an entry point is emitted from an exit point. The reasons why this can be challenging are that:

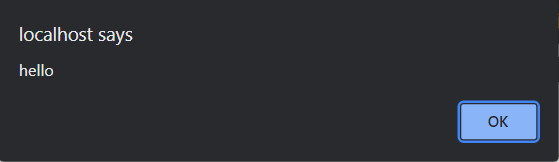
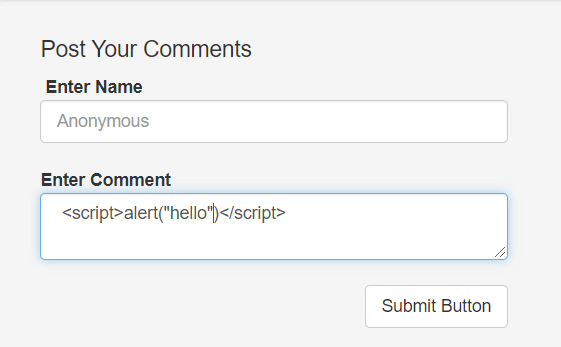
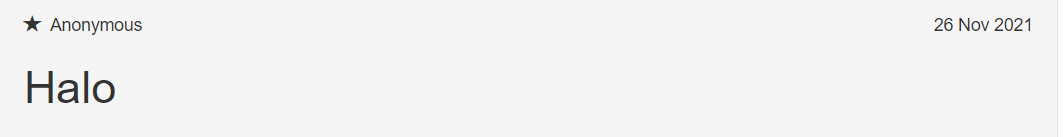
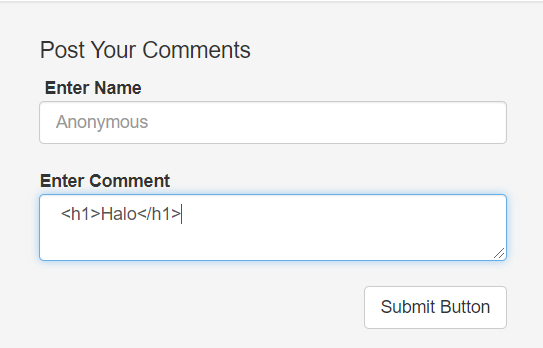
* Data submitted to any entry point could in principle be emitted from any exit point. For example, user-supplied display names could appear within an obscure audit log that is only visible to some application users.
* Data that is currently stored by the application is often vulnerable to being overwritten due to other actions performed within the application. For example, a search function might display a list of recent searches, which are quickly replaced as users perform other searches.

To comprehensively identify links between entry and exit points would involve testing each permutation separately, submitting a specific value into the entry point, navigating directly to the exit point, and determining whether the value appears there. However, this approach is not practical in an application with more than a few pages.

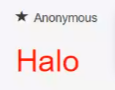
Instead, a more realistic approach is to work systematically through the data entry points, submitting a specific value into each one, and monitoring the application's responses to detect cases where the submitted value appears. Particular attention can be paid to relevant application functions, such as comments on blog posts. When the submitted value is observed in a response, you need to determine whether the data is indeed being stored across different requests, as opposed to being simply reflected in the immediate response.

When you have identified links between entry and exit points in the application's processing, each link needs to be specifically tested to detect if a stored XSS vulnerability is present. This involves determining the context within the response where the stored data appears and testing suitable candidate XSS payloads that are applicable to that context. At this point, the testing methodology is broadly the same as for finding reflected XSS vulnerabilities.





<script style color:”red”>Halo</script>



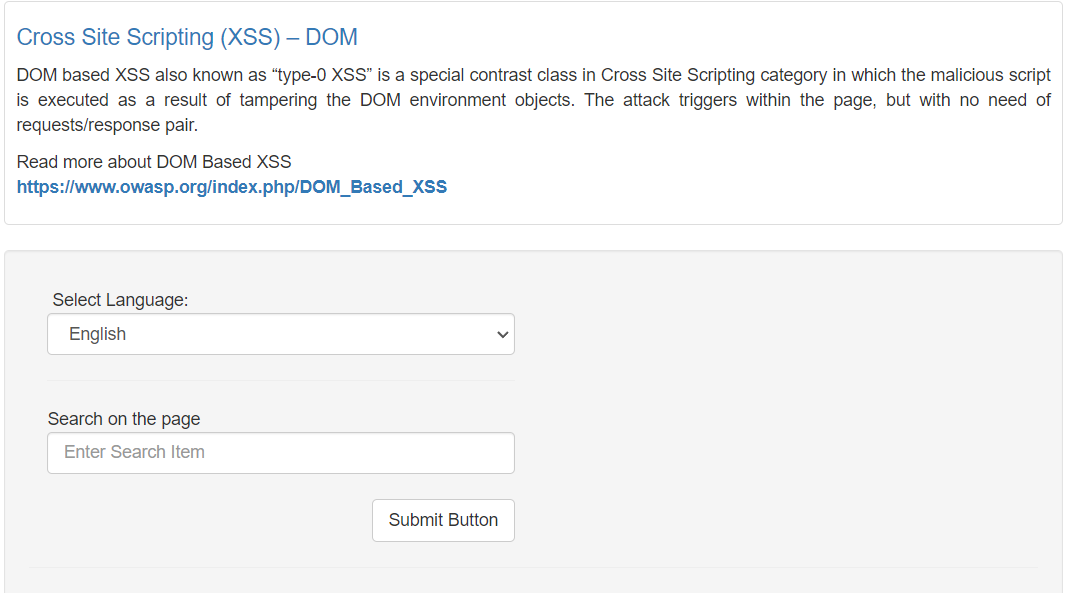
**DOM Based Cross Site Scripting**

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.

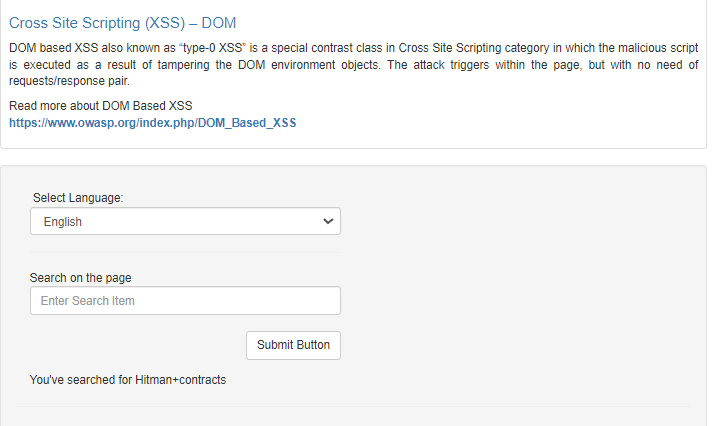
To deliver a DOM-based XSS attack, you need to place data into a source so that it is propagated to a sink and causes execution of arbitrary JavaScript.

The most common source for DOM XSS is the URL, which is typically accessed with the window.location object. An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL. In certain circumstances, such as when targeting a 404 page or a website running PHP, the payload can also be placed in the path.

Search for anything to reflect in DOM terminal.



Try DOM based file usage to reflect in Window.



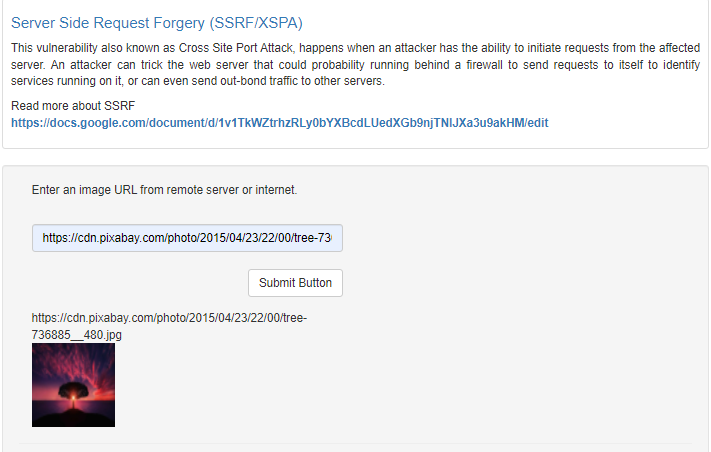
**Server Side Request Forgery (Cross Site Port Attacks)**

Server-side request forgery (also known as SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems, potentially leaking sensitive data such as authorization credentials.

Get URL of image from google:





**File Inclusion**

File Inclusion vulnerabilities often affect web applications that rely on a scripting run time, and occur when a web application allows users to submit input into files or upload files to the server. They are often found in poorly-written applications.

File Inclusion vulnerabilities allow an attacker to read and sometimes execute files on the victim server or, as is the case with Remote File Inclusion, to execute code hosted on the attacker’s machine.

An attacker may use remote code execution to create a web shell on the server, and use that web shell for website defacement.

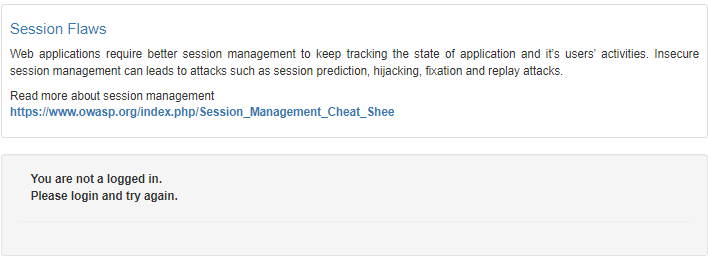


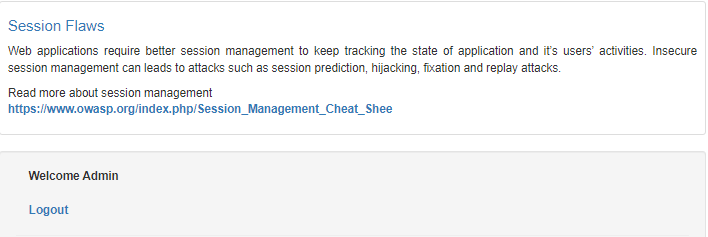
**Session Issues**

Session security plays a key factor in building secure web applications. A web application is not secure unless it is protected from external attacks like XSS. These malicious scripts are designed to gain access to sensitive data in web applications, including cookies, as they act as a key to store session tokens.

Attackers can exploit and gain unauthorized access to the web application because of the improper implementation of authorisation or authentication. According to OWASP (Open Web Application Security Project) Top 10, broken authentication is the second biggest risk to web application security.

Login as admin in window to create and destroy XVWA session.



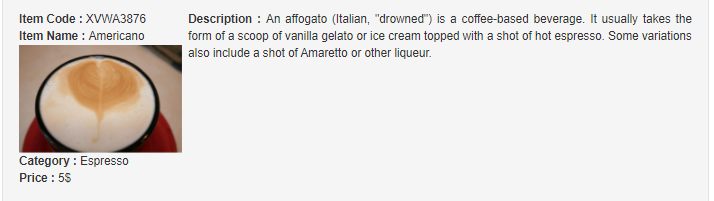
 

**Insecure Direct Object Reference**

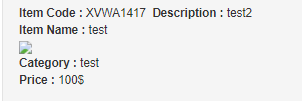
Insecure direct object references (IDOR) are a type of access control vulnerability that arises when an application uses user-supplied input to access objects directly. The term IDOR was popularized by its appearance in the OWASP 2007 Top Ten. However, it is just one example of many access control implementation mistakes that can lead to access controls being circumvented. IDOR vulnerabilities are most commonly associated with horizontal privilege escalation, but they can also arise in relation to vertical privilege escalation.

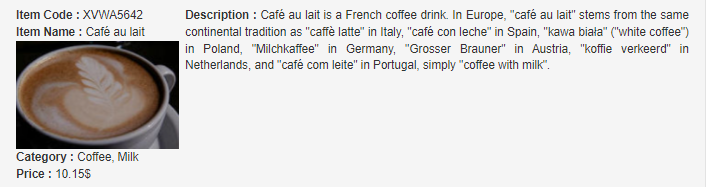
Change Localhost URL to reflect item change in window.

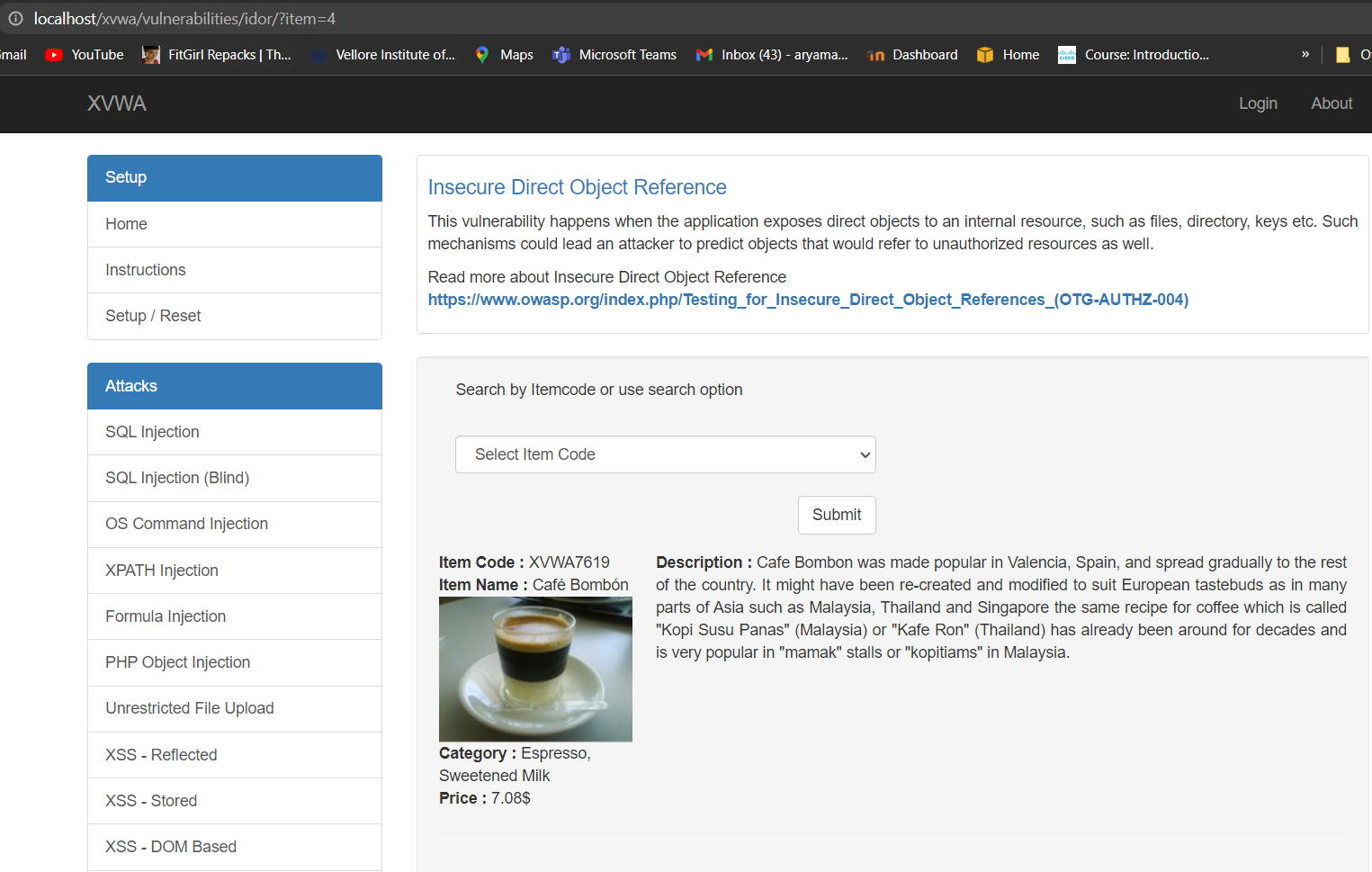
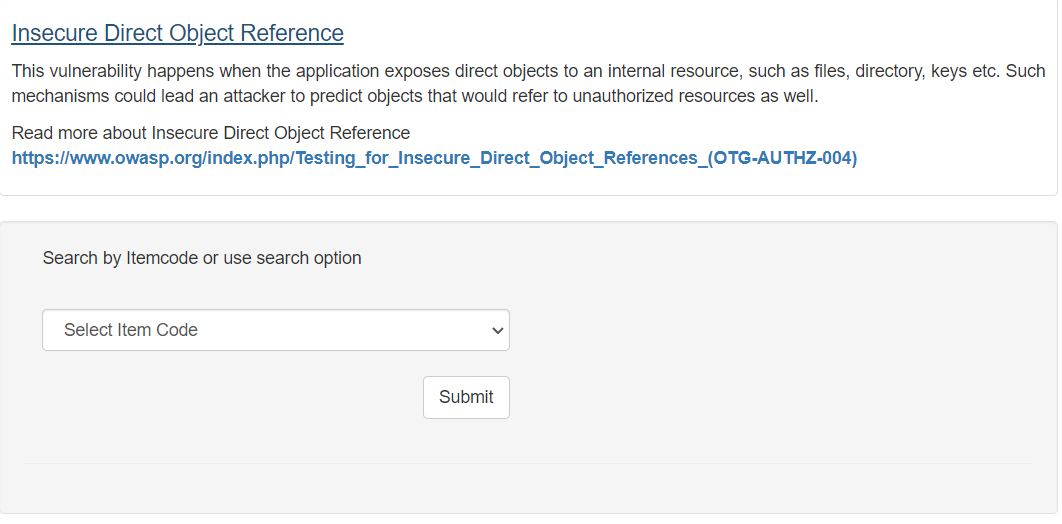


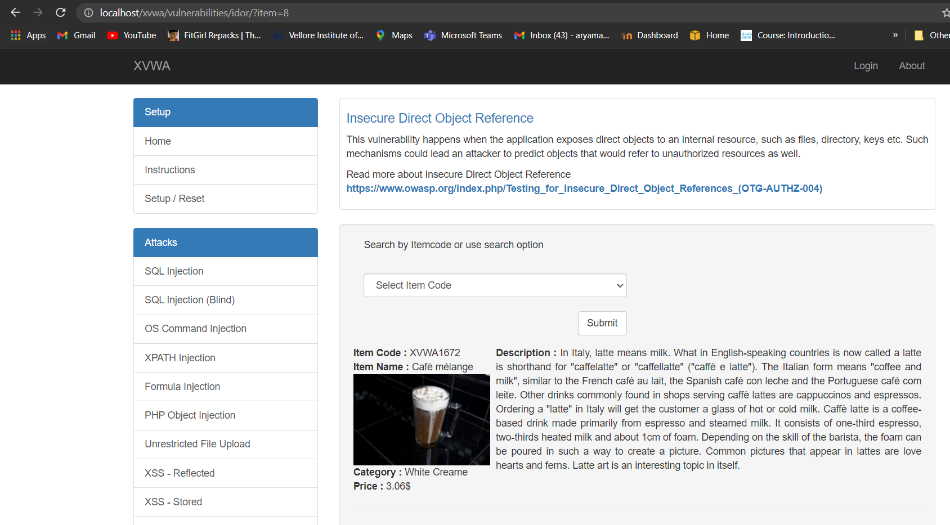






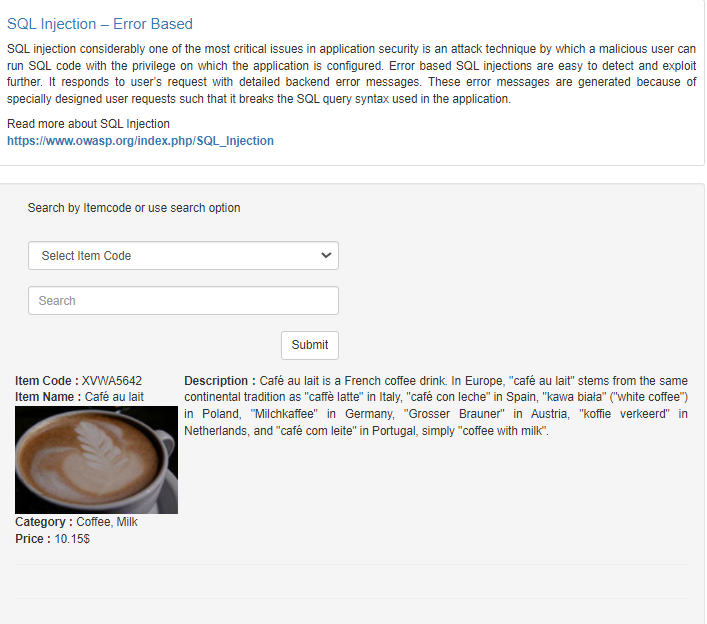


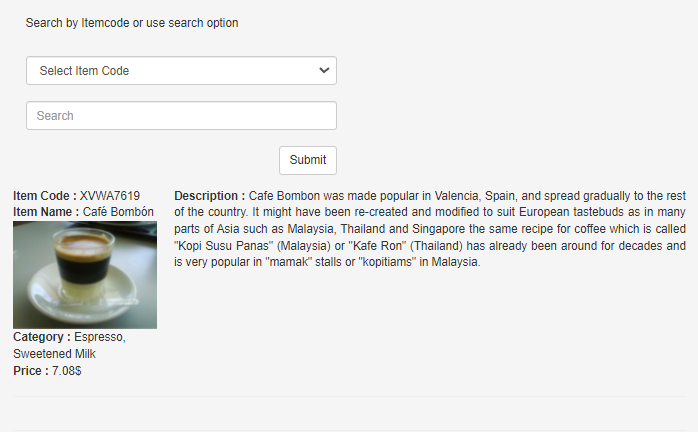


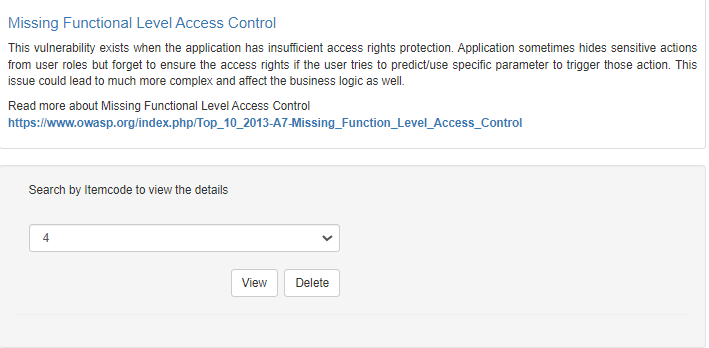
**Missing Functional Level Access Control**

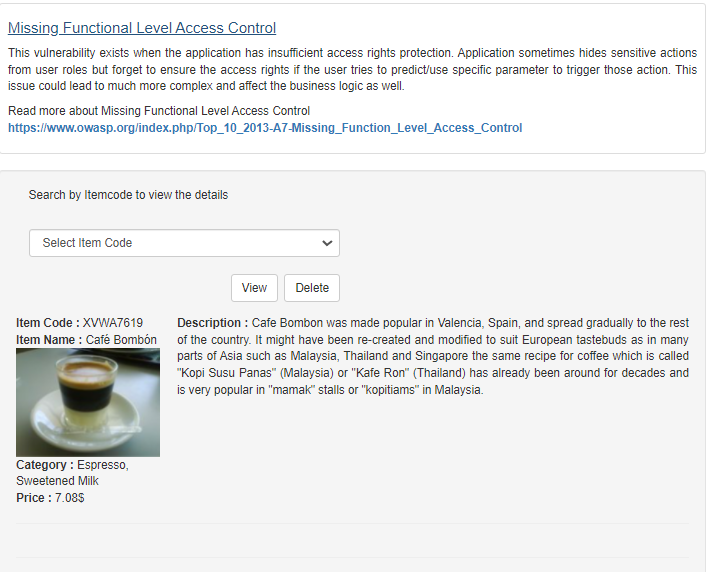
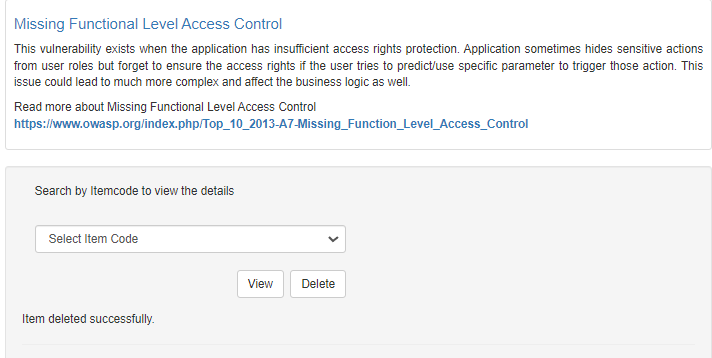
The **missing function level access control** vulnerability allows users to perform functions that should be restricted, or lets them access resources that should be protected. Normally, functions and resources are directly protected in the code or by configuration settings, but it's not always easy to do correctly. Implementing proper checks can be difficult because modern applications often contain many types of roles and groups, plus a complex user hierarchy.

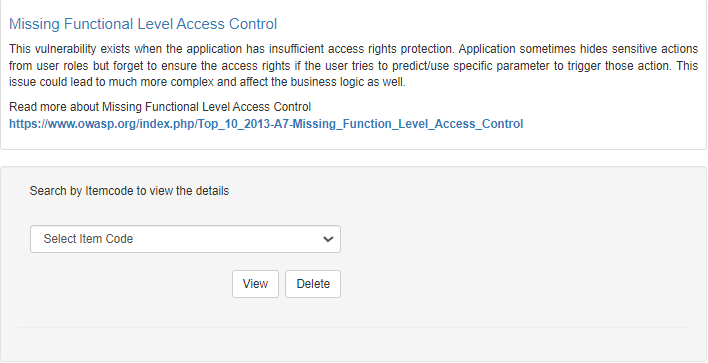
Search for different type of coffee in window.







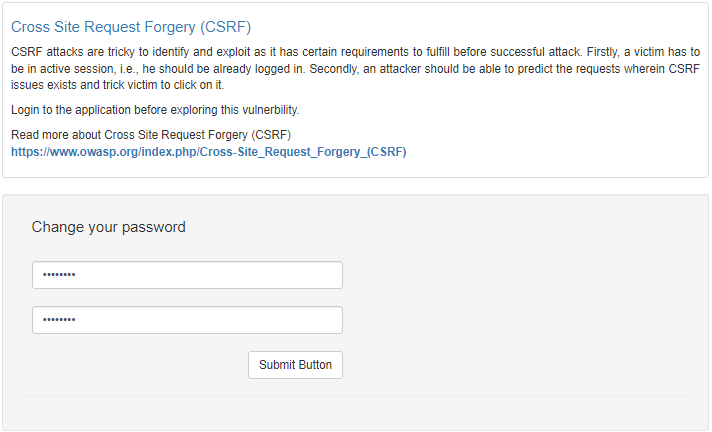
 

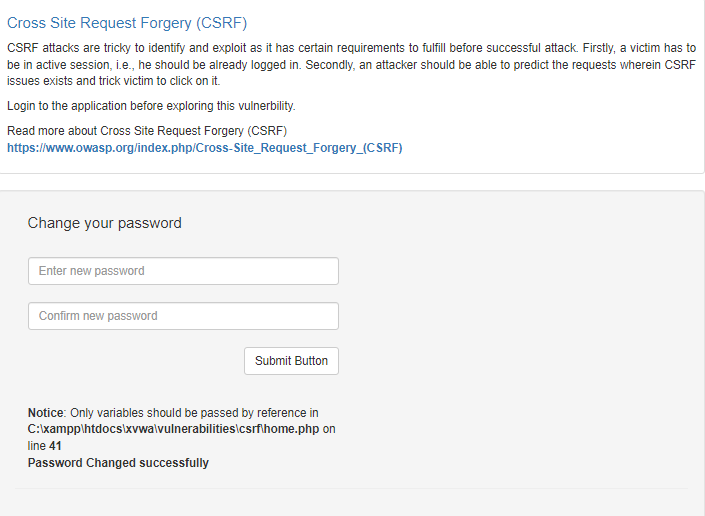


**Cross Site Request Forgery (CSRF)**

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they’re currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker’s choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

Change password in XVWA window.



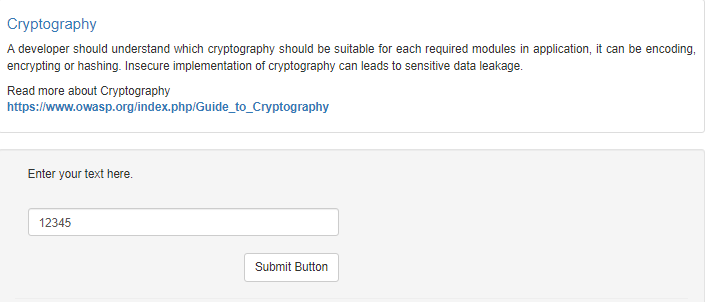


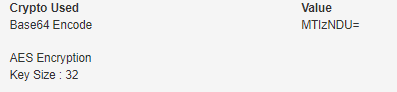
**Cryptography**

Cryptography is the study of secure communications techniques that allow only the sender and intended recipient of a message to view its contents. The term is derived from the Greek word kryptos, which means hidden. It is closely associated to encryption, which is the act of scrambling ordinary text into what's known as ciphertext and then back again upon arrival. In addition, cryptography also covers the obfuscation of information in images using techniques such as microdots or merging. Ancient Egyptians were known to use these methods in complex hieroglyphics, and Roman Emperor Julius Caesar is credited with using one of the first modern ciphers.

When transmitting electronic data, the most common use of cryptography is to encrypt and decrypt email and other plain-text messages. The simplest method uses the symmetric or "secret key" system. Here, data is encrypted using a secret key, and then both the encoded message and secret key are sent to the recipient for decryption.

Encrypt 12345 using this option In XVWA.



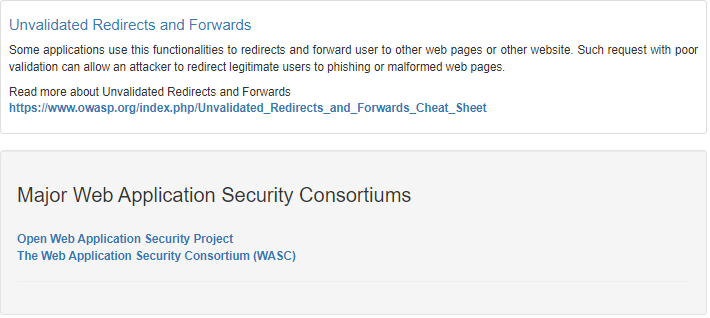


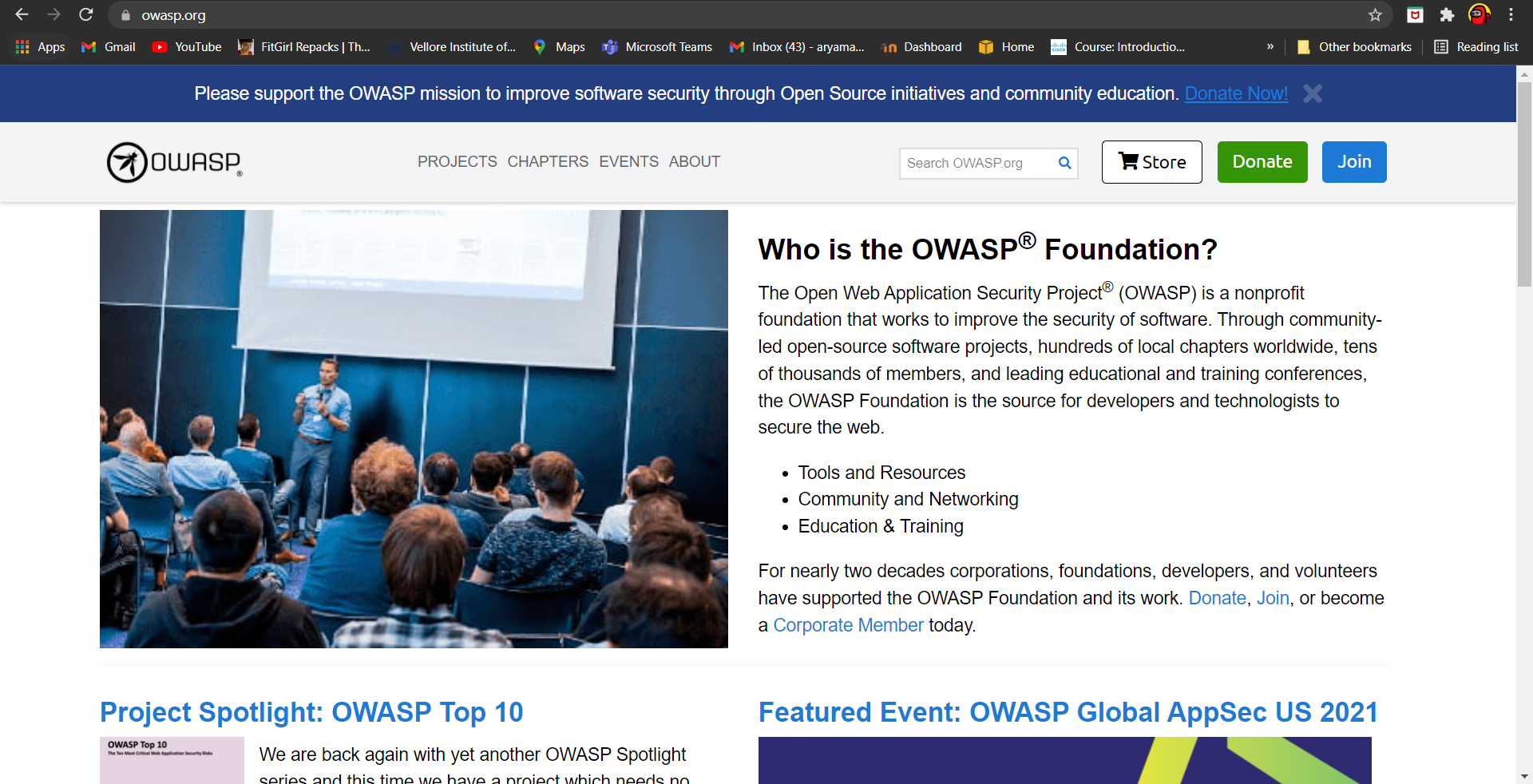
**Unvalidated Redirect & Forwards**

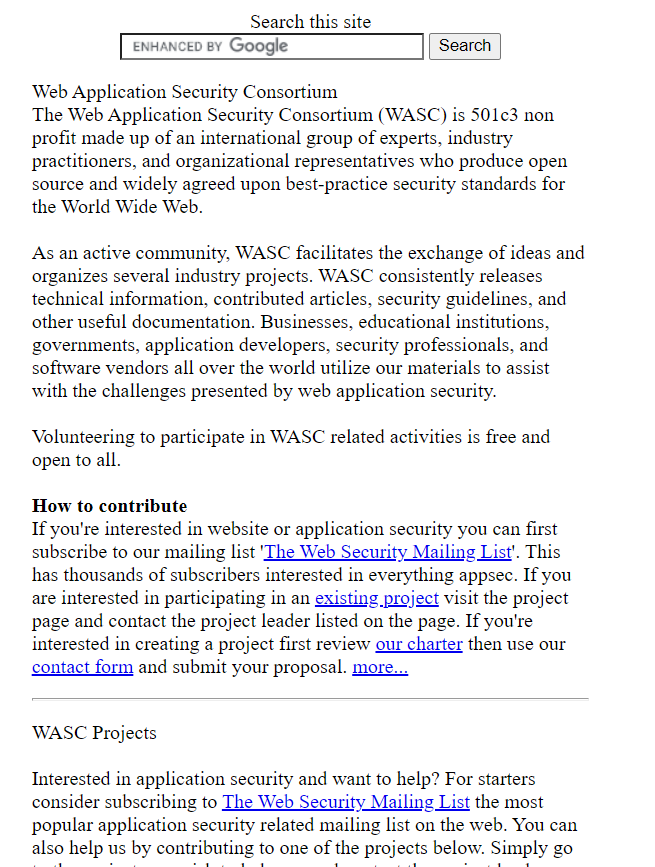
Unvalidated redirects and forwards are possible when a web application accepts untrusted input that could cause the web application to redirect the request to a URL contained within untrusted input. By modifying untrusted URL input to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials.

Because the server name in the modified link is identical to the original site, phishing attempts may have a more trustworthy appearance. Unvalidated redirect and forward attacks can also be used to maliciously craft a URL that would pass the application's access control check and then forward the attacker to privileged functions that they would normally not be able to access.

Click on links to study manipulated projects.







**Server Side Template Injection**

Server-side template injection is when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

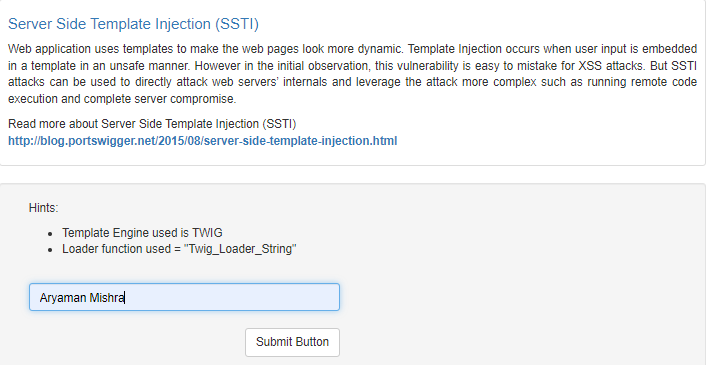
Template engines are designed to generate web pages by combining fixed templates with volatile data. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server. As the name suggests, server-side template injection payloads are delivered and evaluated server-side, potentially making them much more dangerous than a typical client-side template injection.

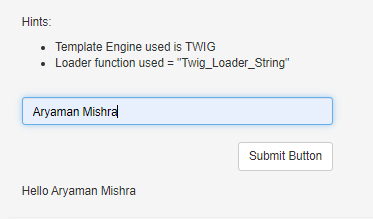
Server-side template injection vulnerabilities can expose websites to a variety of attacks depending on the template engine in question and how exactly the application uses it. In certain rare circumstances, these vulnerabilities pose no real security risk. However, most of the time, the impact of server-side template injection can be catastrophic.

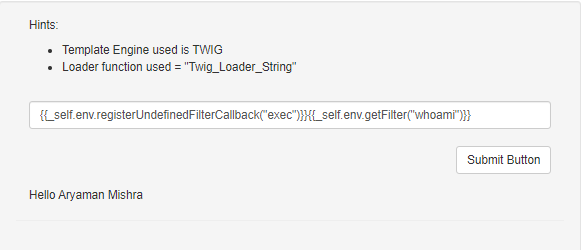
At the severe end of the scale, an attacker can potentially achieve remote code execution, taking full control of the back-end server and using it to perform other attacks on internal infrastructure.

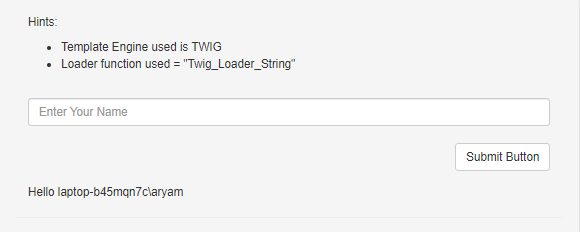
Even in cases where full remote code execution is not possible, an attacker can often still use server-side template injection as the basis for numerous other attacks, potentially gaining read access to sensitive data and arbitrary files on the server.

Enter name in box to print output or any other search related term.









**CONCLUSION:**

**ALL SQL INJECTION Attacks have been successfully implemented and recorded.**