

Web Application Hacking Lesson XSS Basics

Lesson Objectives

-
- Review lecture on the Cross Site Scripting(below)
- Complete lab on Cross Site Scripting(below)

Cross Site Scripting Lecture:

What is cross-site scripting?

Takeaway: is a class of security exploit that has gotten a fair bit of attention in the last few years. Many users, and even Web developers, aren't entirely clear on what the term means, however. I'll explain cross-site scripting for you, so you will know where the dangers lie.

Defining cross-site scripting

Definition:

XSS vulnerabilities occur whenever an application takes data that originated from a user and sends it to a web browser without first properly validating or encoding it. XSS attacks can be used to hijack user sessions, deface websites, conduct port scans on victims' internal networks, conduct phishing attacks and/or take over users' web browsers.

JavaScript is a powerful tool for developing rich Web applications. Without client-side execution of code embedded in HTML and XHTML pages, the dynamic nature of Web applications like [Google Maps](#), [Try Ruby!](#) and [Zoho Office](#) would not be possible. Unfortunately, any time you add complexity to a system, you increase the potential for security issues — and adding JavaScript to a Web page is no exception. Among the problems introduced by JavaScript are:

1. A malicious website might employ JavaScript to make changes to the local system, such as copying or deleting files.
2. A malicious website might employ JavaScript to monitor activity on the local system, such as with keystroke logging.
3. A malicious website might employ JavaScript to interact with other Websites the user has open in other browser windows or tabs.

The first and second problems in the above list can be mitigated by turning the browser into a sort of “sandbox” that limits the way JavaScript is allowed to behave so that it only works within the browser’s little world. The third can be limited somewhat as well, but it is all too easy to get around that limitation because whether a particular webpage can interact with another webpage in a given manner may not be something that can be controlled by the software employed by the end user. Sometimes, the ability of one website’s JavaScript to steal data meant for another Website can only be limited by the due diligence of the other website’s developers.

The key to defining cross-site scripting is in the fact that vulnerabilities in a given website’s use of dynamic Web design elements may give someone the opportunity to use JavaScript for security compromises. It’s called “cross-site” because it involves interactions between two separate websites to achieve its goals. In many cases, however, even though the exploit involves the use of JavaScript, the website that’s vulnerable to cross-site scripting exploits does not have to employ JavaScript itself at all. Only in the case of local cross-site scripting exploits does the vulnerability have to exist in JavaScript sent to the browser by a legitimate website.

Types of cross-site scripting

There are currently three major categories of cross-site scripting. Others may be discovered in the future, however, so don’t think this sort of misuse of Web page vulnerability is necessarily limited to these three types.

- Reflected: Probably the most common type of cross-site scripting exploit is the reflected exploit. It targets vulnerabilities that occur in some websites when data submitted by the client is immediately processed by the server to generate results that are then sent back to the browser on the client system. An exploit is successful if it can send code to the server that is included in the Web page results sent back to the browser, and when those results are sent the code is not encoded using HTML special character encoding — thus being interpreted by the browser rather than being displayed as inert visible text.

The most common way to make use of this exploit probably involves a link using a malformed URL, such that a variable passed in a URL to be displayed on the page contains malicious code. Something as simple as another URL used by the server-side code to produce links on the page, or even a user’s name to be included in the text page so that the user can be greeted by name, can become a vulnerability employed in a reflected cross-site scripting exploit.

- Stored: Also known as HTML injection attacks, stored cross-site scripting exploits are those where some data sent to the server is stored (typically in a database) to be used in the creation of pages that will be served to other users later. This form of cross-site scripting exploit can affect any visitor to your website, if your site is subject to a stored cross-site scripting vulnerability. The classic example of this sort of vulnerability is content management software such as forums and bulletin boards where users are allowed to use raw HTML and XHTML to format their posts.

As with preventing reflected exploits, the key to securing your site against stored exploits is ensuring that all submitted data is translated to display entities before display so that it will not be interpreted by the browser as code.

- Local: A local cross-site scripting exploit targets vulnerabilities within the code of a webpage itself. These vulnerabilities are the result of incautious use of the Document Object Model in JavaScript so that opening another Web page with malicious JavaScript code in it at the same time might actually alter the code in the first page on the local system. In older versions of Internet Explorer (before IE 6 on MS Windows XP Service Pack 2), in fact, this could even be used on local Web pages (stored on the local computer rather than retrieved from the World Wide Web), and through those pages break out of the browser “sandbox” to affect the local system with the user privileges used to run the browser. Because most MS Windows users have tended to run everything as the Administrator account, this effectively meant that local cross-site scripting exploits on MS Windows before XP Service Pack 2 could do just about anything.

In a local cross-site scripting exploit, unlike reflected and stored exploits, no malicious code is sent to the server at all. The behavior of the exploit takes place entirely on the local client system, but it alters the pages provided by the otherwise benign Website before they are interpreted by the browser so that they behave as though they carried the malicious payload to the client from the server. This means that server-side protections that filter out or block malicious cross-site scripting will not work with this sort of exploit. For more about local cross-site scripting, see the explanation at [DOM Based Cross Site Scripting](#).

Protection Against Cross-Site Scripting

The most comprehensive way to protect your Web design from being exploited by cross-site scripting is to translate any and all special characters in user-provided input — even in URLs — into display entities, such as [HTML entities](#). This applies not only to server-side code like PHP, Perl, and ASP.NET code, but also JavaScript that works with any user-provided input as well. This may interfere with the operation of Websites where users expect to be able to use HTML and XHTML in their input, such as for Website design helper applications — in which case more complex code may be needed to protect against malicious code. Such fine-grained filtering is just one side of an arms race against malicious security crackers, however, and cannot reasonably be 100% effective.

Another way to protect your Website from cross-site scripting exploits is to never directly use any user-provided input in your pages.

Accepting a limited number of values in user-provided input that are each used as “keys,” for lack of a better term, to choose from among certain predefined options is an example of how user-provided input can be used to define output, but obviously greatly limits the dynamic

nature of Web applications. If your website does not need greater dynamism than this provides, however, this may be your safest option for generating output based on user input.

Similarly, input validation that simply strips out all characters unauthorized for specific, limited input types (such as removing everything but dashes, parentheses, periods, and digits from input expected to contain telephone numbers), or that rejects input containing unauthorized characters entirely, can be used. This is a useful technique for many forms of input, but not all. Such validation techniques should be used whenever possible, because they not only provide some protection against cross-site scripting, but also against direct attempts to compromise the server itself through buffer overflows, SQL injection, and other attempts to exceed the bounds of the system.

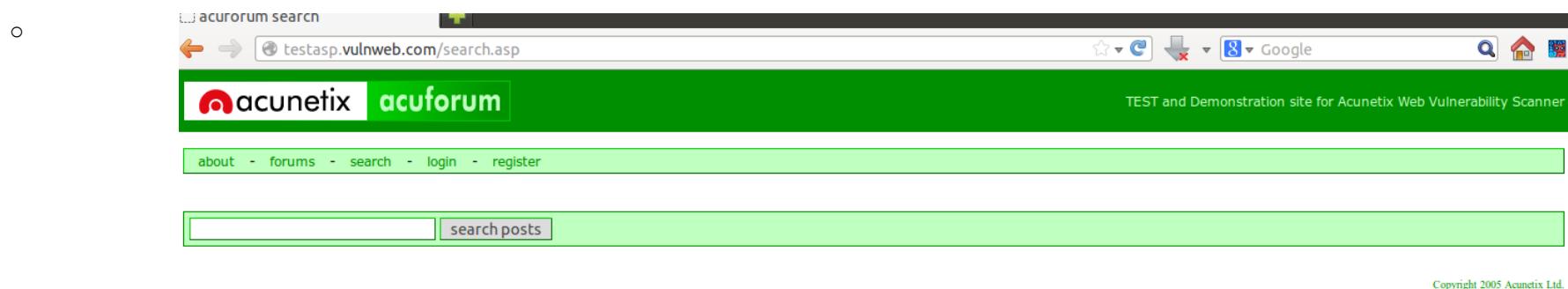
Cookies are often used to provide some form of security against cross-site scripting. Many cross-site scripting exploits are designed to “steal” session cookies, but a cookie can be “tied” to a particular IP address so that hijacked cookies fail validation when employed by cross-site scripting exploits. There are potential work-arounds for this sort of security, such as when the legitimate user of a given cookie and a cross-site scripting exploit both originate from behind the same proxy server or NAT device, of course. Internet Explorer implements an `HTTPOnly` flag that prevents local scripts from affecting a cookie to try to guard against this sort of cookie abuse, but it is ineffective against cross-site request forgery attacks, where unintended requests may be sent via cross-site scripting exploits alongside a cookie used to authorize the requests at the server.

The single most effective means of avoiding cross-site scripting in Web development, however, is to design your website so that it does not require client-side code at all. That way, if your users want to turn off the JavaScript interpreters in their browsers, they can do so without losing the ability to make use of your Website. This does not protect against all forms of potential malicious input to your server, of course, and it does not actually limit the vulnerability of your website all by itself — but it does give visitors to your website the option of protecting *themselves*.

Lab for Cross Site Scripting Introduction

Lab setup

- **Instructions:**
 1. Our target site is here <http://testasp.vulnweb.com/search.asp>



Lab – Cross Site Scripting

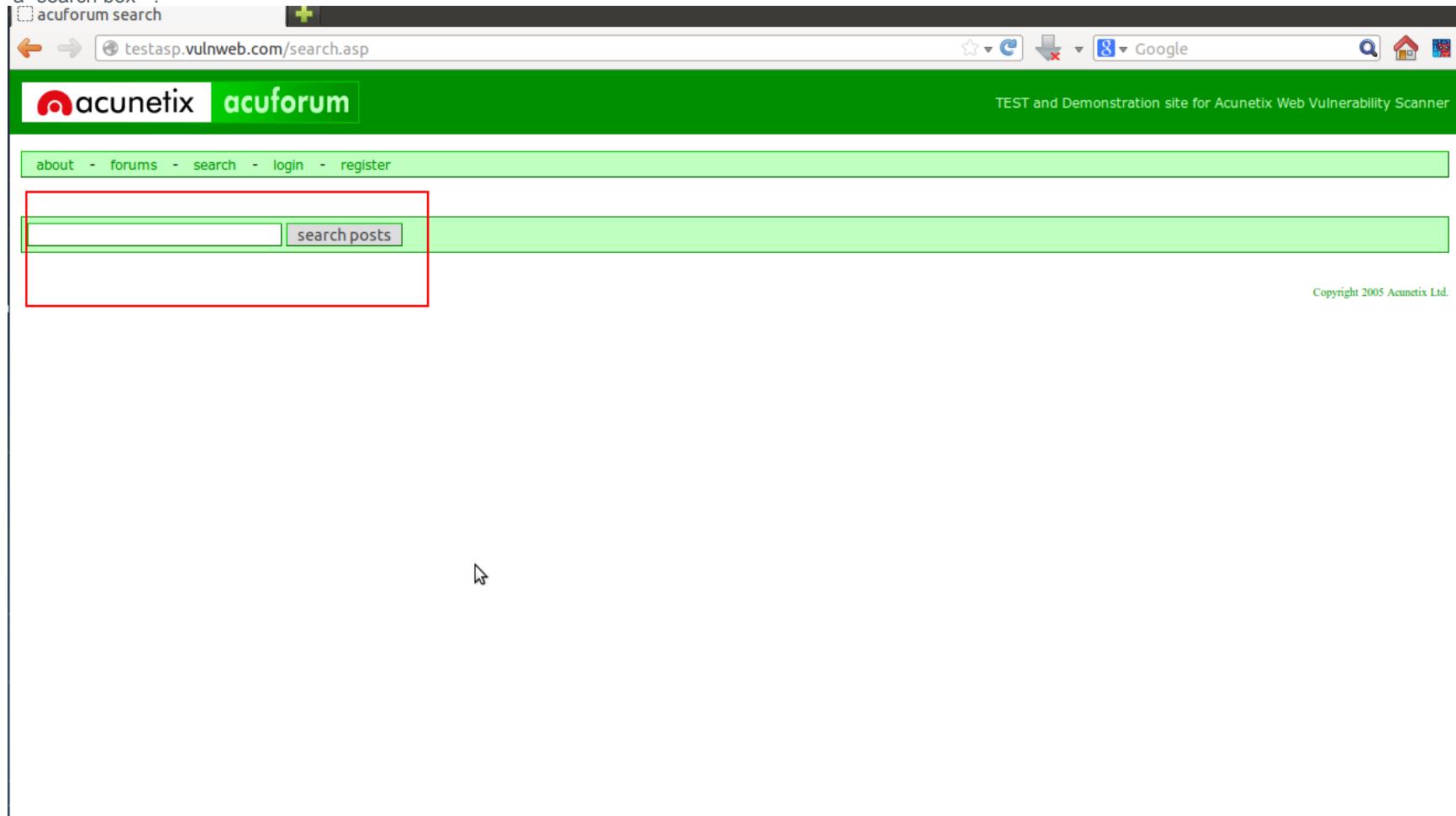
Cross Site Scripting (XSS) attacks occur when a website fails to properly prevent attackers from inserting malicious code into an area that takes user provided data, such as form fields on a webpage, HTTP headers, URLs etc. This code is then used to attack other users, rather than the actual website or the server it's hosted on. This leads many to underestimate the dangers of XSS, despite the fact that an XSS flaw could allow attackers to record what users are typing, redirect users to malicious sites to capture their credentials and change the content of pages to defraud users, to name but a few.

There are three main types of XSS attacks, reflected, stored and DOM based. Reflected are the most common, whilst stored are the most deadly.

These occur when the attack is reflected off a webserver, via a search result, error message or any response which echoes the user provided data back to the user. Reflected attacks are normally delivered via a link and therefore require some amount of coercion, such as tricking the victim into clicking a link sent to them in an email.

So say if a search function on a website was vulnerable to XSS, i.e.

a "search box" :



Then we could enter as a search term:

```
<script>alert("boom!")</script>
```

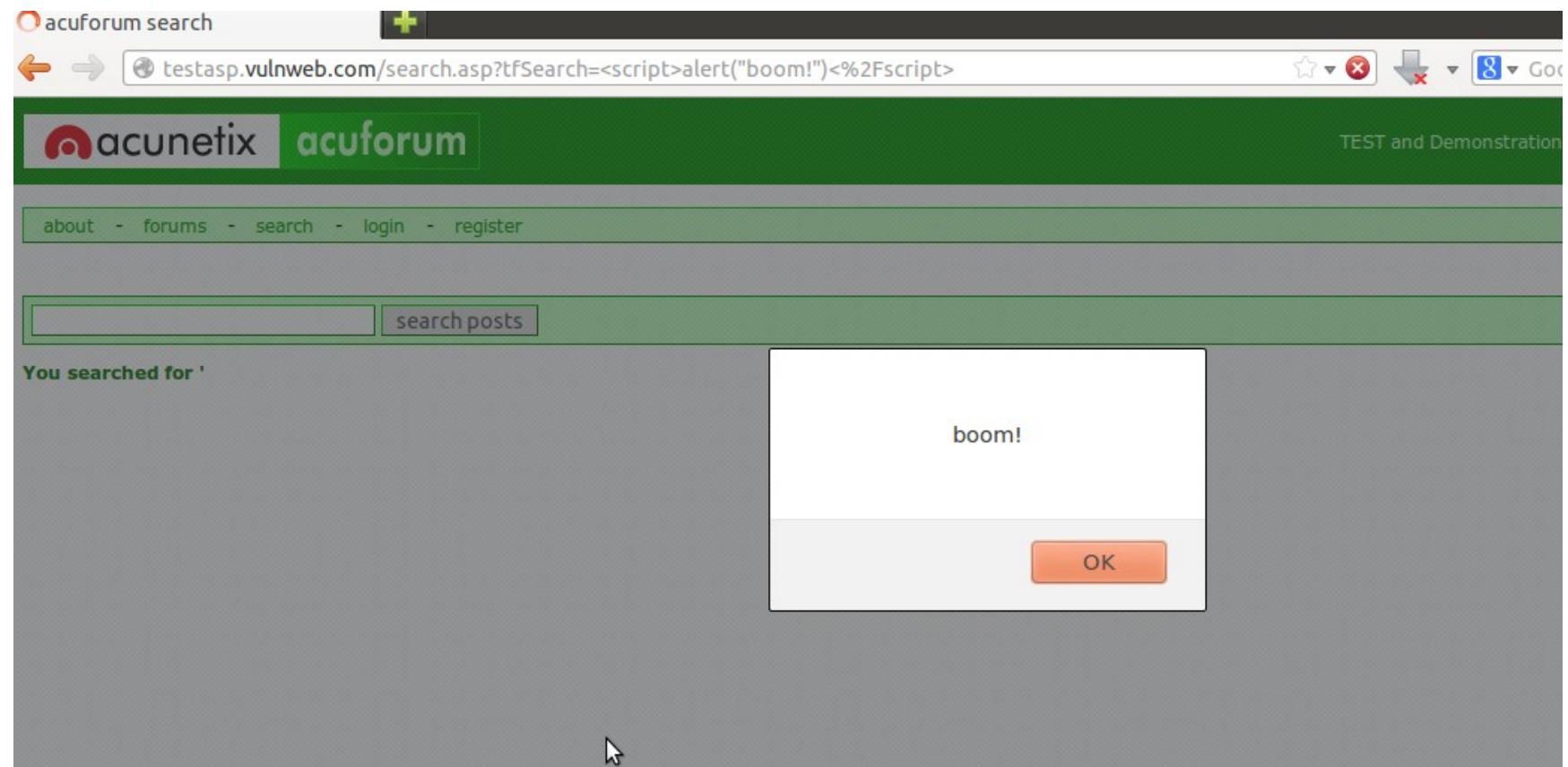
t>

Which is a really small JavaScript script that popups an alert box, saying boom!

Thus if we then send the entire link to someone else, i.e.:

<http://testasp.vulnweb.com/search.asp?tfSearch=%3Cscript%3Ealert%28%22boom!%22%29%3C%2Fscript%3E>

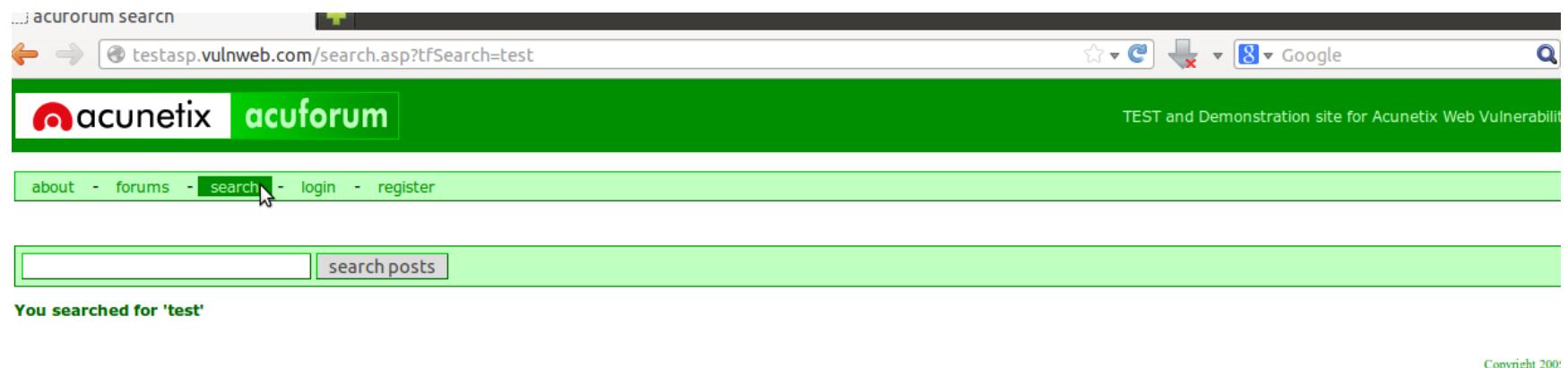
They'll see that the host is testasp.vulnweb.com think it's a legitimate site and click it, and then fall victim to the popup box of doom!



Ok maybe not of doom, but this is most common way of testing for XSS flaws, via these alert boxes. Once a flaw is confirmed, more malicious attacks can be demonstrated. You can see where the name 'reflected' comes from too, as you need a vulnerable site to 'reflect' the attack off.

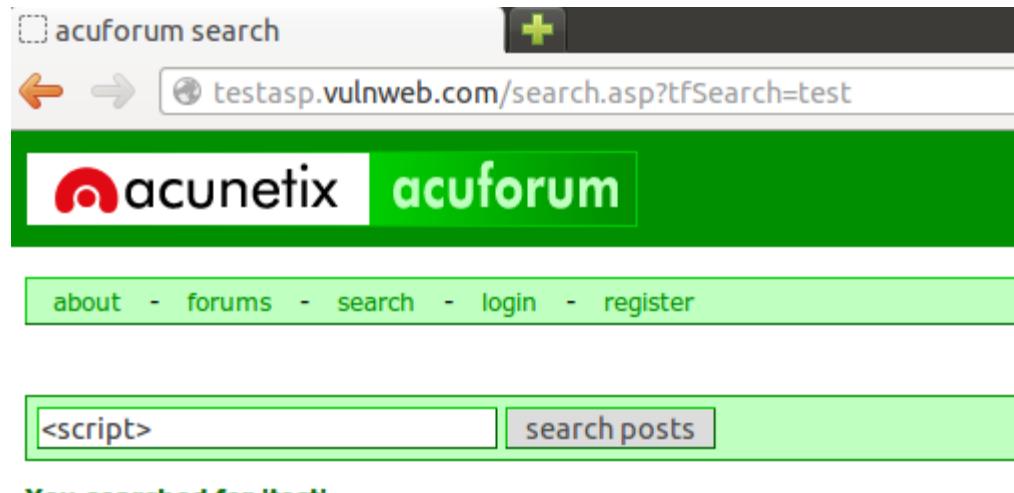
Note, if you would rather practice these skills online, i.e. without installing DVWA, you can try reflected XSS attacks [here](#) and stored attacks [here](#) (be sure to reset the database for the stored example prior to use [here](#)). All my screenshots are of DVWA, but the same XSS strings and principles **will work in the online practice environments too.**

So head over to the <http://testasp.vulnweb.com/search.asp> and enter some text into the search box, like "Test". The first thing we notice is that what we entered is being echoed exactly back to us on the page:



Our next step is to confirm whether the page is performing any sort of filtering of the JavaScript syntax required for our malicious script to work. From the above XSS string, you can see that we use the angle bracket symbols, < >. These are essential to JavaScript as they indicate to the browser that what it's reading is a line of code, as opposed to just text. Without the angle brackets our code won't be read as JavaScript and so our attack will fail. Because of this, angle brackets

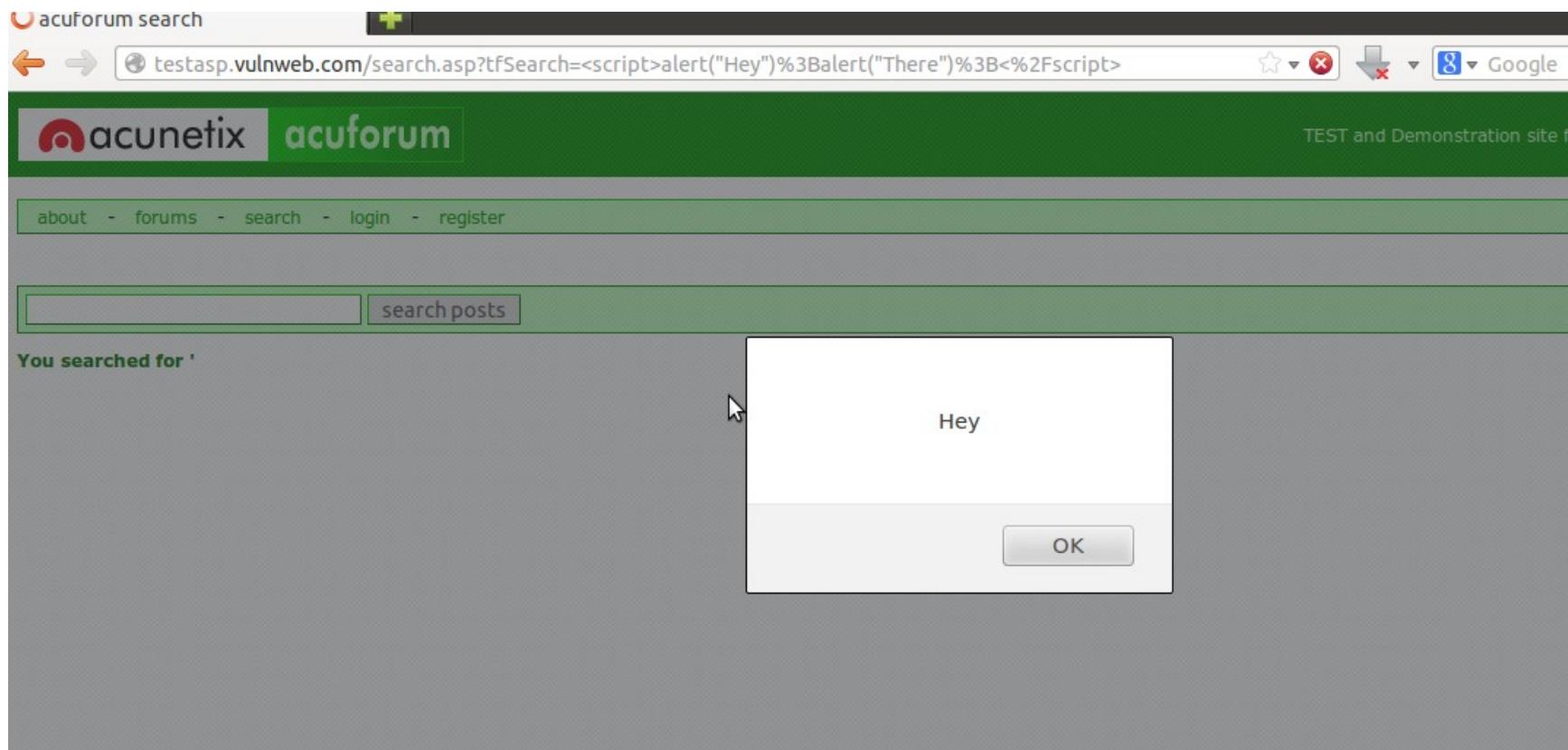
are commonly removed from user supplied input, or are encoded in such a way that the browser won't read them as code, such as [HTML entity encoding](#). Similarly, the word 'script' is often treated in the same way, as some (but not all!) XSS attacks rely on it. This process is known as sanitization or filtering. So let's try entering the word <script>:



As you can see, <script> is not displayed, because it's being read as part of the source of the webpage. If we check the source we can see that it's there and that there's no encoding taking place either. Cue rush of adrenaline because we've found an XSS flaw!

```
<head>
<!-- InstanceBeginEditable name="doctitle" -->
<title>acuforum search</title>
<!-- InstanceEndEditable -->
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<!-- InstanceBeginEditable name="head" --><!-- InstanceEndEditable -->
<link href="styles.css" rel="stylesheet" type="text/css">
</head>
<body>
<table width="100%" border="0" cellpadding="10" cellspacing="0">
<tr bgcolor="#008F00">
  <td width="306px"><a href="http://www.acunetix.com/"></a></td>
  <td align="right" valign="middle" bgcolor="#008F00" class="disclaimer">TEST and Demonstration site for Acunetix Web Vulnerability
</tr>
<tr>
  <td colspan="2"><div class="menubar"><a href="Templatize.asp?item=html/about.html" class="menu">about</a> - <a href="Default.asp"
    - <a href=".../Login.asp?RetURL=%2Fsearch%2Easp%3FtfSearch%3D%253Cscript%253E" class="menu">login</a> - <a href=".../Register.asp?Re
  </div></td>
</tr>
<tr>
  <td colspan="2"><!-- InstanceBeginEditable name="MainContentLeft" -->
    <form name="frmSearch" method="get" action="">
      <div class="FramedForm">
        <input name="tfSearch" type="text" class="search">
        <input class="search" type="submit" value="search posts">
      </div>
    </form>
    <div class='path'>You searched for '<script>'</div><table width="100%" cellspacing="1" cellpadding="5" border="1" bordercolor="#E5E5E5">
<!-- InstanceEndEditable --></td>
</tr>
<tr align="right" bgcolor="#FFFFFF">
  <td colspan="2" class="footer">Copyright 2005 Acunetix Ltd.</td>
</tr>
</table>
</body>
<!-- InstanceEnd --></html>
```

So now all we have to do is pop an alert box to confirm... Type `<script>alert("Hey");alert("There");</script>` in the search box.



Boom! As expected, our script is being entered directly into the source and is thus being executed by the browser, so popping up an alert box! Also, you may notice I have added a ';' at the end of our statement within the script tags. JavaScript normally requires a ';' after every statement, to indicate that the line of code has ended. For example, if we wished to have two consecutive popups display 'Hey' 'There', our code would be:

```
<script>alert("Hey");alert("There");</script>
```

So we have two JavaScript statements in one script, separated by ';'. Fortunately, JavaScript repairs little syntax errors like forgetting the closing ';' for us, so we don't have to worry about it. (This often proves useful when filter evading too, which will be covered in a later post.)

That's reflected covered, now onto its less common but far more deadly friend, stored XSS!

Stored XSS relies on exactly the same principle as reflected XSS; you're trying to insert your own malicious code into the source, so when a victim visits that page, their browser will execute your code. The only difference, and the reason why they are so much more dangerous, is that your code will be permanently stored on the page. As a result, anyone who subsequently visits that page will trigger the attack; no coercion is required by you at all. They have traditionally often been found on message boards, public guestbooks etc.

Open up this link <http://playground.nebulassolutions.com/index.php?page=add-to-your-blog.php>, and immediately we see that big juicy text box, crying out for some JavaScript to be entered into it! We put in our script `<script>alert("boom!")</script>` and...

Connecting... +

playground.nebulassolutions.com/index.php?page=add-to-your-blog.php

Google

Mutillidae: Hack, Learn, Secure, Have Fun!!!

Powered by nLab

Version: 2.0.12 Not Logged In

Home Login/Register Toggle Hints Toggle Security Setup/Reset the DB

Core Controls

OWASP Top 10 2010

Others

Documentation

Resources

Site hacked...err...quality-tested with Samurai WTF, Backtrack, Firefox, Paros, Netcat

boom!

Back Add New Blog Entry OK

Add blog for anonymous

Note: and are now allowed in blog entries

Save Blog Entry

Transferring data from playground.nebulassolutions.com

There it is, being entered directly into the source, with no encoding or filtering. Thus from now on, whenever we revisit the page we will be XSSed, as the browser re-executes the code in the source every time you visit.

```
437 <div>&nbsp;</div><table border="1px" width="100%" class="main-table-frame"><tr class="report-header"><td colspan="3">Current Blog Entries</td></tr>
438 <tr class="report-header">
439     <td><b>Name</b></td>
440     <td><b>Date</b></td>
441     <td><b>Comment</b></td>
442 </tr><tr>
443     <td>anonymous</td>
444     <td>2013-02-08 22:27:28</td>
445     <td><script>alert("boom!")</script></td>
446 </tr>
447 <tr>
448     <td>anonymous</td>
449     <td>2013-02-08 22:26:59</td>
450     <td>Stuck in traffic . <a href="http://lph.ece.utexas.edu/?suid=249">cd key Microsoft Office Visio Professional 2003</a> . <a href="http://lph.ece.utexas.edu">
451 </tr>
452 <tr>
453     <td>anonymous</td>
454     <td>2013-02-08 22:24:19</td>
455     <td>Hola! <a href="http://gravatar.com/eikoolfesa#enttm">buy tramadol online mastercard</a> <a href="http://labs.divx.com/node/126559#anuuu">buy tramadol onl:
456 </tr>
457 <tr>
458     <td>anonymous</td>
459     <td>2013-02-08 22:24:19</td>
460     <td>Hola! <a href="http://gravatar.com/eikoolfesa#zlyrb">buy tramadol online</a> <a href="http://labs.divx.com/node/126559#ahqiv">where can i buy tramadol on:
461 </tr>
462 <tr>
463     <td>anonymous</td>
464     <td>2013-02-08 22:24:18</td>
465     <td>Hola! <a href="http://gravatar.com/eikoolfesa#ydmn">buy tramadol onlin<generic</a> <a href="http://labs.divx.com/node/126559#pzzn">how to buy tramadol
466 </tr>
467 <tr>
468     <td>anonymous</td>
```

Section 15. Proof of Lab

- **Proof of Lab Instructions:**

1. Do a <PrtScn> of all input, and results
2. Paste into a word document
3. Email to me
-

Questions:

1. What are three problems that a malicious website could introduce with javascript?
2. Why is Cross-site Scripting called “Cross Site”?
3. What are the three types of XSS?
- 4.