ModSecurity Use Case:

Web 2.0 Defense with Ajax Fingerprinting and Filtering

Ajax is fast becoming an integral part of new generation Web applications known as Web 2.0 applications. This evolution has led to new attack vectors coming into existence around these new technologies.

This article is largely based on Shreeraj Shah's article of the same name that appeared in the December 2006 edition of Insecure Magazine (http://www.net-security.org/dl/insecure/INSECURE-Mag-9.pdf). Ryan C. Barnett, Director of Application Security Training at Breach Security, has updated numerous sections to reflect the advanced ModSecurity 2.0 rules language.

To combat these new threats one needs to look at different strategies as well. In this paper we shall look at different approaches and tools to improve security posture at both, the server as well as browser ends. Listed below are the key learning objectives:

- The need for Ajax fingerprinting and content filtering.
- The concept of Ajax fingerprinting and its implementation in the browser using XHR.
- Processing Ajax fingerprints on the Web server.
- Implementation using `ModSecurity` for Apache
- Strengthening browser security using HTTP response content filtering of untrusted information directed at the browser in the form of RSS feeds or blogs.
- Web application firewall (WAF) for content filtering and defense against Cross-Site Scripting (XSS)

Requirement for Ajax fingerprints and filtering

Ajax is being used very liberally in next generation Web applications, forming an invisible layer in the browser's transport stack and bringing to the fore numerous browser-related attacks, all centered around Ajax. Although Ajax applications hold a lot of promise, there are loopholes being exploited by viruses, worms and malicious attackers in Web 2.0 applications that need to be looked at a little more closely. Ajax hides a lot of server-side critical resources due to its calling mechanism, bringing in sloppiness in coding patterns and fueling vulnerabilities in the server-side application layer as well. Untrusted resource processing from blogs, feeds and mash-ups are making Ajax vulnerabilities relatively easy to exploit. In such situations Ajax request and response fingerprinting and filtering mechanisms can enhance the security posture of Web applications.

Web 2.0 applications have a special set of resources that are accessed by the web browsers over Ajax calls using the `XMLHttpRequest` (XHR) object. Resources can be grouped into two broad spaces – one with "Ajax-only" access and other non-Ajax (traditional) resources. In the application architecture, one can wrap security around Ajax resources by creating a separate virtual sandbox for all incoming and outgoing Ajax calls as shown in Figure 1.0.
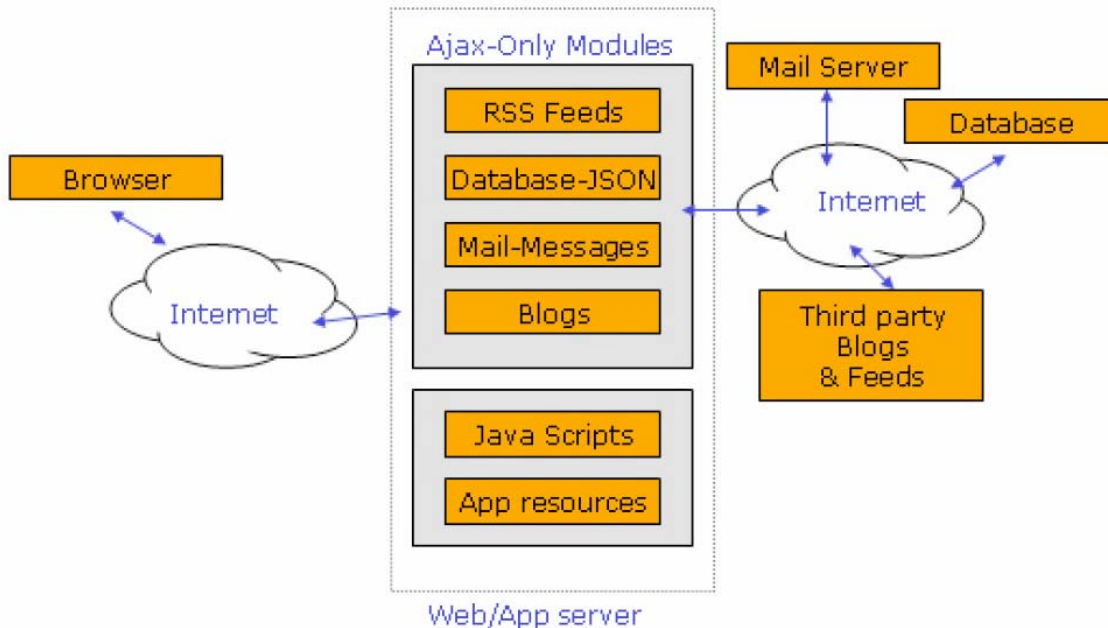
Figure 1.0 – Ajax sandbox on the server-side.

"Ajax-Only" modules access third-party resources such as blogs and feeds using their own proxy code. These proxies are essential since direct cross-domain access with Ajax is not possible. However, JavaScript scripts residing in the browser can access database streams directly over JSON or a JavaScript array as shown in Figure 1.0. Ajax resources serve a lot of untrusted and unfiltered information to the browser, in the process leaving an end-users' browser vulnerable to several client side attacks such as XSS and XSRF.

To provide better security framework to both applications and browsers, Ajax resources on the server-side can be defended by applying Ajax fingerprinting methods. The key question, however, that we need to ask is, "is there a way to identify an HTTP Ajax call?" It would be easy to build several security controls for both application and browser security provided an Ajax call can be fingerprinted. This is the topic of discussion in this article.

Applying firewall rules for incoming traffic is always important, but in an Ajax-Only framework, filtering outgoing traffic is of greater importance given the fact that the application serves untrusted information to the browser in the current application DOM context. Put simply, if a DOM-based XSS attack is successful, the client session can be hijacked with ease. This application may be running a banking system, financial transactions, mailing system or blogs. Losing session information can result in financial or non-financial losses.

Implementing Ajax fingerprinting – Adding extra HTTP headers

To implement Ajax fingerprinting, we need to first identify the `HTTP GET` and `POST` request structure for Ajax calls. Figure 2.0 illustrates a simple example of an Ajax call. The browser loads the "news.html" page. Clicking the link "Get today's headline", will make a backend Ajax call to the server requesting for the "/ajax-only/headline" resource. The code snippet in Listing 1.0 gets executed by the browser when a "click" action occurs. i.e. the `getHeadline()` function is invoked.
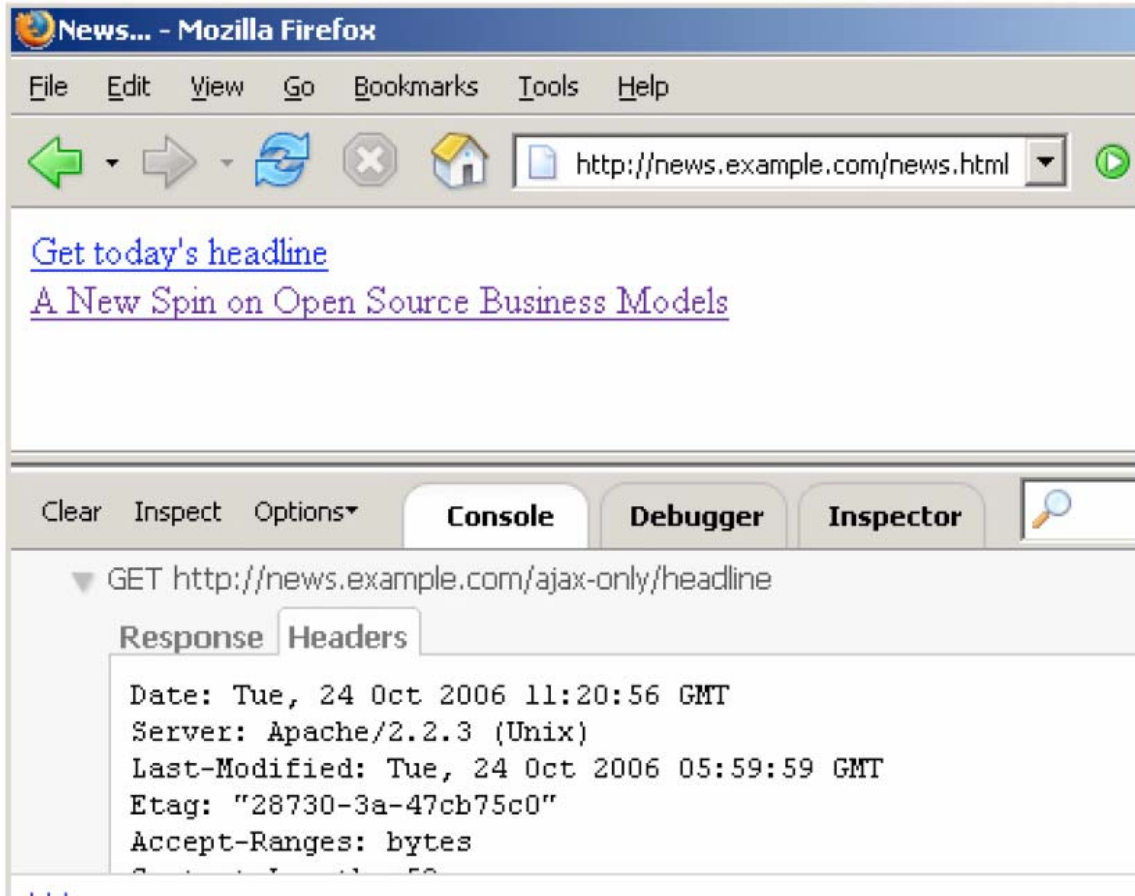
Figure 2.0 – Sample Ajax call.

```
function getHeadline()
{
// Intializing the XHR Object
var http;
if(window.XMLHttpRequest) {
http = new XMLHttpRequest();
} else if (window.ActiveXObject) {
http=new ActiveXObject("Msxml2.XMLHTTP");
if (! http) {
http=new ActiveXObject("Microsoft.XMLHTTP");
}
}
// Building a request
http.open("GET", "/ajax-only/headline", true);
// Getting ready for response processing
http.onreadystatechange = function()
{
if (http.readyState == 4) {
var response = http.responseText;
document.getElementById('result').innerHTML = response;
}
}
//Sending Async request on the wire
http.send(null);
}
```

Listing 1.0 - The `getHeadline()` function.

The following GET request will be sent to the Web server:

```
GET /ajax-only/headline HTTP/1.1
Host: news.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.8.0.6)
Gecko/20060728
Firefox/1.5.0.6
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/pla
in;q=0.8,image/png,*/*;q=0.5
Accept-Language: en,en-us;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

A cursory glance at the request gives no indication that the request is made by the XHR object from within the browser. It is possible to add an extra header to the HTTP request as per XHR's methods that would aid in identifying and fingerprinting the Ajax call.

```
// Building request
http.open("GET", "/ajax-only/headline", true);
http.setRequestHeader("Ajax-Timestamp",Date())
```

Modify the code snippet in Listing 1.0 to attach an "Ajax-Timestamp" header to the outgoing HTTP requests. By using the output of the **Date()** function and a browser fingerprinting technique, we can identify browsers as well.

Now, click the same link again. This is the GET request that will be generated on the wire:

```
GET /ajax-only/headline HTTP/1.1
Host: news.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.8.0.6)
Gecko/20060728
Firefox/1.5.0.6
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/pla
in;q=0.8,image/png,*/*;q=0.5
Accept-Language: en,en-us;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```
Ajax-Timestamp: Tue Oct 24 2006 17:37:46 GMT+0530 (India Standard Time)

Look closely at the GET request. From this GET request we can determine the fingerprint of the Ajax call. On the server we receive the following timestamp header:

Ajax-Timestamp: Tue Oct 24 2006 17:37:46 GMT+0530 (India Standard Time)

This fingerprinting technique helps in determining the type of client code that has sent this request. It is possible to lockdown resources for just the right client on the server side as well. This type of header is harder to add by automated crawlers and bots since the logic and calls

need to be understood first. Consequently, automated attacks on your Ajax resources can be avoided.

Fingerprinting is just a starting point for securing Ajax resources. It is possible to build a security control around this extra header mechanism. You can add JavaScript libraries in your client-side code and use MD5 hashing and other encryption methods. The XHR object controls the POST method along with buffer that the client sends to the server. A secure tunnel can be built over HTTP using Ajax calls by encrypting data as well along with the extra header – another option that needs to be explored.

Detecting Ajax fingerprints on the Web server

We have Ajax fingerprints on an outgoing request from the browser. The Web application passes JavaScript to the browser in such a way that each legitimate request made by the browser has correct fingerprints. All that remains to be done is to process the request on the Web server prior to serving the resource to the browser. This will be our first line of defense for Ajax-locked resources. We can build a defense bundled into the Web application firewall. We shall utilize ModSecurity for the Apache platform. Let us see this approach in a little detail.

Leveraging ModSecurity web application firewall

ModSecurity - (http://www.modsecurity.org) is an application-level firewall that fits into the Apache Web server as a module. After the firewall is loaded into Apache (by modifying **httpd.conf**), start adding filtering rules. We shall add a specific ruleset for Ajax fingerprinting. Here is a sample rule using the new ModSecurity 2.0 rules language.

```
<IfModule mod_security2.c>
SecRuleEngine On
SecRequestBodyAccess On
SecDebugLog logs/modsec_debug_log
SecDebugLogLevel 3
SecAuditLogType Serial
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4\d[^4])"
SecAuditLog logs/mod_audit_log
SecDefaultAction "phase:2,deny,log,status:500
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase"

<LocationMatch ^/ajax-only/>
# Filtering incoming content
SecRuleInheritance On
# Deny the request if the custom Ajax header is not present
SecRule &REQUEST_HEADERS:Ajax-Timestamp "@eq 0"
# Deny the request if the custom Ajax header does not contain expected
# data
SecRule REQUEST_HEADERS:Ajax-Timestamp
"!^\w{3}\s\w{3}\s\d{2}\s\d{4}\s\d{1,2}\:\d{2}\:\d{2}\sGMT(\+|-
)\d{4}\s\(\w+\s\w+\s\w+\)$"
</LocationMatch>
</IfModule>
```

In above code snippet, the first few lines will set up the engine with logging enabled. The most critical ruleset that we want to set up is for the "Ajax-Only" section. All Ajax-serving resources reside in the /ajax-only/ folder. Hence, we define our Ajax sandbox on the server by adding the "LocationMatch" tag with the correct folder. All incoming requests to "Ajax-Only" must have a

proper Ajax-Timestamp. Apache will not serve any request that does not include this timestamp. These are the key filter rules at the application firewall.

```
SecRule &REQUEST_HEADERS:Ajax-Timestamp "@eq 0"
SecRule REQUEST_HEADERS:Ajax-Timestamp
"!^\w{3}\s\w{3}\s\d{2}\s\d{4}\s\d{1,2}\:\d{2}\:\d{2}\sGMT(\+|-
)\d{4}\s\(\w+\s\w+\s\w+\)$"
```

We chop off the "Ajax-Timestamp" header; if it is empty, not present or does not contain data in the specified Date format, a "500" error is thrown back, as shown in the following screenshot.

```
root@host:/home/root# nc news.example.com 80
GET /ajax-only/header HTTP/1.0

HTTP/1.1 500 Internal Server Error
Date: Wed, 25 Oct 2006 15:17:21 GMT
Server: Apache/2.2.3 (Unix)
Content-Length: 607
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
```

Now, if we send a "proper" header to the server, we receive this response:
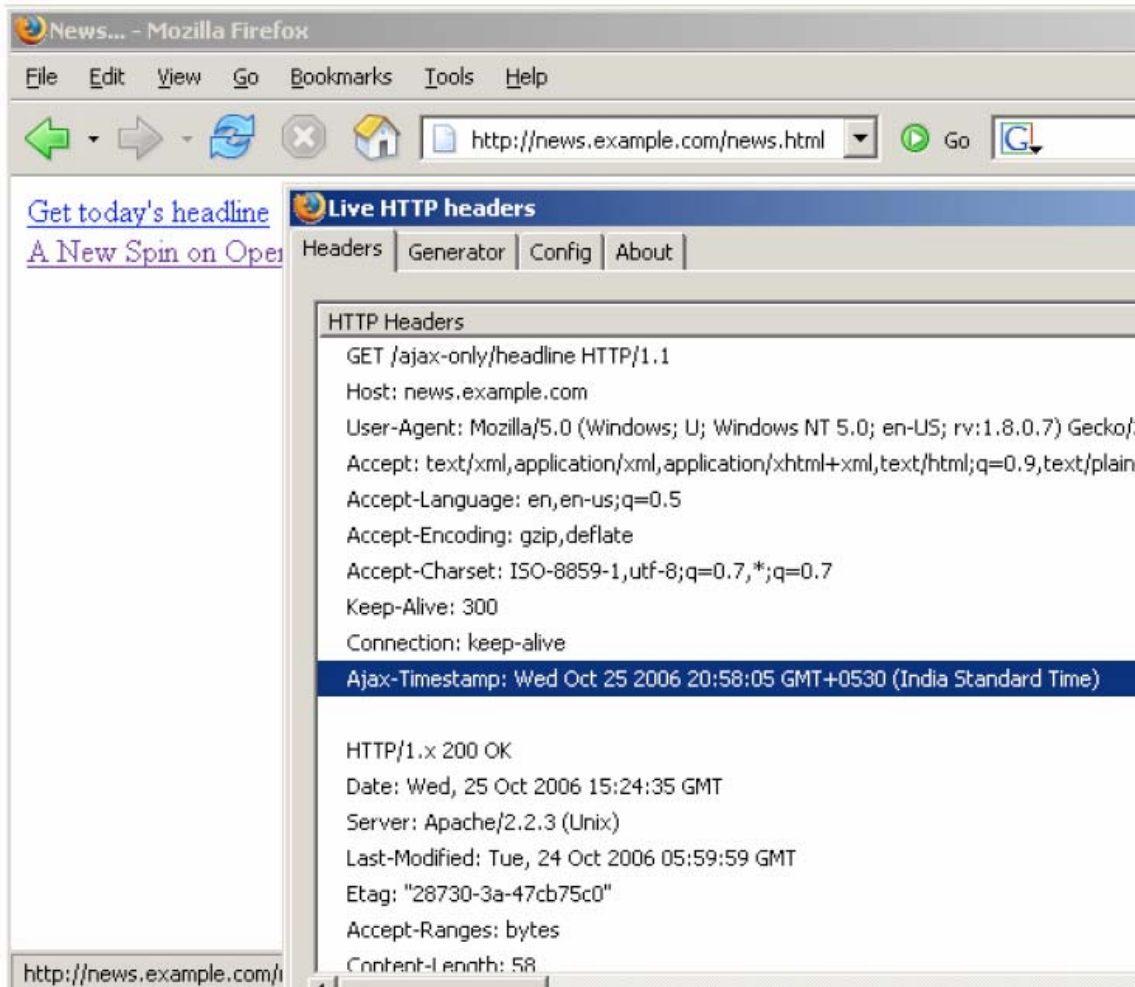
Figure 3.0 – Ajax request with the correct Timestamp.

The correct Ajax fingerprint in the HTTP request provides an entry to resources. This example demonstrates that a web application firewall can be utilized in the right context for Ajax resources.

Implementing content filtering to defend against XSS 2.0

XSS attacks are steadily mounting in Ajax frameworks. Ajax makes a backend call to various third-party resources such as RSS feeds or blogs. Since Ajax can not directly make these calls to the target site, calls are routed through server-side proxy code. It is important to filter out bad content originating from third-party untrusted sources and directed to the end user's browser. One of the approaches that can be adopted is by adding rulesets into the Web application firewall (WAF) for all third-party information modules. Here is an example that demonstrates this approach. Once again we can use ModSecurity to enable response filtering on HTTP/HTTPS content. We add certain rules.

```
<IfModule mod_security2.c>
SecRuleEngine On
SecRequestBodyAccess On
SecResponseBodyAccess On
SecResponseBodyMimeType text/xml text/plain text/html
SecDebugLog logs/modsec_debug_log
```

```
SecDebugLogLevel 3
SecAuditLogType Serial
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4\d[^4])"
SecAuditLog logs/mod_audit_log
SecDefaultAction "phase:2,deny,log,status:500
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase"

<LocationMatch ^/ajax-only/>
# Filtering incoming content
SecRuleInheritance On
# Deny the request if the custom Ajax header is not present
SecRule &REQUEST_HEADERS:Ajax-Timestamp "@eq 0"
# Deny the request if the custom Ajax header does not contain expected
# data
SecRule REQUEST_HEADERS:Ajax-Timestamp
"!^\w{3}\s\w{3}\s\d{2}\s\d{4}\s\d{1,2}\:\d{2}\:\d{2}\sGMT(\+|-
)\d{4}\s\(\w+\s\w+\s\w+\)$"
```
 # XSS Rule for Ajax
**SecRule RESPONSE_BODY "(javascript:|<\s*script.*?\s*>)" **
"phase:4,log,deny,msg:'Cross-site Scripting (XSS) Attack found in Ajax
Response.',id:'2',severity:'2'"
```
</LocationMatch>
</IfModule>
```

The following line enables scanning for outgoing content: **SecResponseBodyAccess On**
The following rule ensures that HREFs are not injected with "**javascript**". Any attempt to inject the **<script>** tag in the HTTP response will also be blocked.

**SecRule RESPONSE_BODY "(javascript:|<\s*script.*?\s*>)" **
"phase:4,log,deny,msg:'Cross-site Scripting (XSS) Attack found in Ajax
Response.',id:'2',severity:'2'"

Any malicious content present in third-party information will cause a "500" error to be thrown. The user's browser stays secure. We have the following resource that fetches RSS feeds' XML file from the target server.

/ajax-only/rss?feed=http://sample.org/daily.xml

/rss is proxy code that will fetch the RSS feed from http://sample.org/daily.xml and send it back to the browser. daily.xml has the pattern "javascript" in one of the links. If the link is clicked, malicious code will get executed and the browser session may get compromised.

With response filtering on HTTP/HTTPS content enabled, the same request responds with a "500" error.

```
root@host:/home/root#  nc news.example.com 80
GET /ajax-only/rss?feed=http://sample.org/daily.xml HTTP/1.0
Ajax-Timestamp: Tue Oct 24 2006 17:37:46 GMT+0530 (India Standard Time)
```

HTTP/1.1 500 Internal Server Error
```
Date: Sun, 29 Oct 2006 06:45:56 GMT
Server: Apache/2.2.3 (Unix)
Connection: close
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
```

```
<html><head>
<title>500 Internal Server Error</title>
```

Similarly, the `<script>` tag will be filtered out too. This filtering approach will help in securing a Web client.

Conclusion

Ajax security is a major issue for next generation Web applications. The techniques discussed in this article can give a head start to security professionals to improve the security posture of Web applications. Web 2.0 applications try to integrate various sources, including untrusted information sources, at one place. This trait of Web 2.0 applications adds new attack vectors to the landscape. The advantage of Ajax fingerprinting with XHR is twofold: one, it gives a clear idea about the origin of a request and, two, it makes it harder for automated attacks and crawler modules to launch discovery techniques. With Web application firewalls becoming an important part of Web application defense, one can leverage this mechanism to defend the web browser as well. Tools such as ModSecurity can help in building better and secure deployment.

About Ryan C. Barnett
Ryan C. Barnett is the Director of Application Security Training at Breach Security. He is also a Faculty Member for the SANS Institute, where his duties include Instructor/Courseware Developer for Apache Security/Building a Web Application Firewall Workshop, Top 20 Vulnerabilities Team Member and Local Mentor for the SANS Track 4, "Hacker Techniques, Exploits and Incident Handling" course. He holds six SANS Global Information Assurance Certifications (GIAC): Intrusion Analyst (GCIA), Systems and Network Auditor (GSNA), Forensic Analyst (GCFA), Incident Handler (GCIH), Unix Security Administrator (GCUX) and Security Essentials (GSEC). In addition to the SANS Institute, he is also the Team Lead for the Center for Internet Security Apache Benchmark Project and a Member of the Web Application Security Consortium. Mr. Barnett has also authored a web security book for Addison/Wesley Publishing entitled "Preventing Web Attacks with Apache."

About Breach Security, Inc.
Breach Security, Inc. is a leading provider of next-generation web application security that protects corporate-critical information. Breach effectively protects web applications of commercial enterprises and government agencies alike against Internet hacking attacks and provides an effective solution for expanding security challenges such as identity theft, information leakage, and insecurely coded applications. Breach's solutions are ideal for any organization's regulatory compliance requirements for security. Breach was founded in 2004 and is headquartered in Carlsbad, Calif. For more information visit: www.breach.com.

About Shreeraj Shah
Shreeraj Shah, BE, MSCS, MBA, is the founder of Net-Square and leads Net-Square's consulting, training and R&D activities. He previously worked with Foundstone, Chase Manhattan Bank and IBM. He is also the author of Hacking Web Services (Thomson) and co-author of Web Hacking: Attacks and Defense (Addison-Wesley). In addition, he has published several advisories, tools, and whitepapers, and has presented at numerous conferences including RSA, AusCERT, InfosecWorld (Misti), HackInTheBox, Blackhat, OSCON, Bellua, Syscan, etc. His articles are published on Securityfocus, O'Reilly, InformIT and HNS. You can read his blog at shreeraj.blogspot.com.