Intranet Invasion
Through Anti-DNS Pinning

David Byrne, CISSP, MCSE
Security Architect
EchoStar Satellite L.L.C. / DISH Network
DavidRiByrne@yahoo.com

Introduction

DNS pinning was introduced by web browsers to avoid DNS-spoofing attacks facilitated by client-side code execution. A number of factors including incomplete implementation, browser plug-in vulnerability, plug-in integration and proxy servers have allowed for successful anti-DNS pinning attacks. Using client-side code, such as JavaScript, an internet-based attacker can turn a browser into a proxy server, directing arbitrary attacks at internal servers.

DNS Spoofing

DNS-spoofing attacks against web browsers are primarily intended to trick a browser into violating the same-origin policy[1]. Since same-origin applies to hosts, but not IP addresses, an attacker could use a DNS server he controls to erase the distinction between two different servers. This is the basic attack sequence:

1. Get the victim browser to visit a site (probably using XSS), on an attacker-controlled domain. The hostname is typically randomly generated.
2. The victim browser queries the DNS server for the attack domain and receives the attacker-controlled IP address
3. The victim browser requests content from the attack server and becomes infected with the attack code
4. The attack code pauses long enough for the DNS record's TTL to expire
5. The attack code initiates a new request to the attack server, and requeries DNS
6. The attack DNS server responds with the IP address of a victim server
7. The attack code connects to the victim server, and does something useful
8. The results are returned to the attacker

DNS Pinning

To prevent DNS Spoofing attacks, browser makers introduced DNS pinning. This forces the browser into using a single IP address for any given host. Once the DNS response has been received, the browser will "pin" it in the cache as long as the browser is running.

Interestingly, this could be interpreted as a violation of the HTTP 1.1 standard. It states that “if HTTP clients cache the results of host name lookups in order to achieve a performance improvement, they MUST observe the TTL information reported by DNS.” [2] The authors were concerned that while a DNS response was cached too long, the IP address could have been assigned to a new party. Any requests sent to the IP address would be subject to abuse by the new owner.

Fundamentals of Anti-DNS Pinning Attacks

Most techniques for defeating DNS pinning exploit the necessity to eventually expire the DNS record. One method has this sequence[3]:

1. The victim browser loads attack code
2. The victim browser closes, either by user action or by attack
3. When the browser is opened, the attack code is loaded from disk cache
4. The attack code initiates a request to the attack web server
5. The attack DNS-server responds with the IP address of the victim server

This technique is difficult to defeat by browser design because the browser must dump its DNS cache eventually, and because a disk-based content cache is considered critical for modern browsers. However, it is very difficult to properly execute: simply clearing the content cache is enough to stop the attack, and it is very difficult to get the cached content reloaded. The biggest obstacle is speed; every time the attacker wants to add a victim server, the process must start over.

Considering that major web browsers do not fully implement DNS pinning, there is a much simpler attack[4]. To support DNS-based fault-tolerance, browsers will dump their DNS cache if the web server becomes unavailable. The attack sequence becomes much simpler to execute:

1. The victim browser loads the attack code
2. The attacker firewalls the attack web server
3. The attack code initiates a request to the attack web server
4. The request times out due to the firewall rule, and the victim browser dumps its DNS cache
5. The browser requeries the attack DNS server and receives the IP address of the victim server

A successful attack does not rely on the victim website hosting critical data. Secondary attacks against the web server are possible by exploiting vulnerabilities such as URL or header-based buffer overflows. A more likely scenario would be to find a website vulnerable to SQL injection, then use a tertiary vulnerability, such as xp_cmdshell[5], to execute arbitrary code on the database server. Once arbitrary code can be executed, more traditional and less limited techniques to tunnel traffic become available.

The primary limitation of this method is the lack of control over host headers. Since the browser will use the host name initially associated with the attack web server, only the default website on the victim web server will be accessible. However, there are plenty of servers on the Internet, and many more on intranets with default websites. Critical, high-profile sites are likely to have a dedicated web server, which means they are probably the default website.

Regardless, it is important to note that **HOST HEADERS ARE NOT A RELIABLE SECURITY CONTROL**. Multiple vulnerabilities[6,7,8] in the past have allowed headers, including "host", to be arbitrarily set in a code-generated request. There is no reason to believe that everyone has installed these patches, or that similar vulnerabilities will not be found in the future.

Practical Anti-DNS Attacks Using JavaScript

Attack Components

Anti-DNS attacks can be coupled with other JavaScript-based attack techniques to turn a victim web browser into a proxy server. This attack scenario has several components:

Victim browser: Tricked into visiting a malicious site, probably via XSS, the victim browser loads code that periodically polls the attack server for new commands. In the BlackHat demonstration, every 1.5 seconds, JavaScript appends a new script tag onto the document body. The source of the tag is a request to the controller script, which returns either a blank document, or new JavaScript commands.

Victim web server: The targeted server that the attacker wants access to; most likely an intranet web server protected by perimeter firewalls.

Browser-based JavaScript proxy: The primary purpose of this code is to relay requests and responses between the attack server and the victim web server. In the demonstration, it is loaded into an iframe[9] from the attack server by the victim browser's polling process.

Attacker's browser: Interacts with the attack console to run commands on the victim browser; sends requests to the HTTP proxy running on the attack server.

Attack server: Relays requests and commands from the attacker to the victim browser. There are several ways this could be implemented; in the BlackHat demonstration, there are five sub-components:

Attack DNS server: The attack DNS server is the authoritative name server for a domain that the attacker controls.

Attack web server: The attack web server hosts the browser-based attack code and the controller script.

Controller script: A CGI script with many functions identified by a "command" parameter. The script is hosted on two IP addresses; one is used for performing the anti-DNS pinning attack with the randomly generated hostname; the other is used for communicating commands between other components. Key functions include:

- An attack console listing all active victim browsers, and commands that can be sent
- Periodically polled by the victim browser for new commands
- Changes DNS records and firewall rules as needed to facilitate the actual anti-DNS pinning attack
- Periodically polled by the JavaScript proxy for new requests to process
- Receives the HTTP responses from the JavaScript proxy

Database: Used to store commands and requests sent by the attacker, until they are retrieved by the victim browser. Also stores responses sent by the victim browser until they are retrieved by the attacker.

HTTP proxy: Receives requests from the attacker's browser and inserts them into the database. Polls the database for the response and sends it back to the browser.

Data Exchange

JavaScript proxy and victim web server

The XMLHTTPRequest (XHR) object[10,11,12] is used to initiate requests to the victim web server. Normally, XHR can only handle text data and will effectively strip off the high ASCII bit. By setting the character set to "x-user-defined", the browser will retain all 8-bits of data, allowing for full binary data support[13].

JavaScript proxy to controller script

Because of the same-origin rule, XHR is not suitable for returning data to the attack server. There are two methods used in the demonstration. If it is a small amount of text data, an image object is created with the source pointing at the controller script. The data is included as a parameter value in the URL's query string. When the image is appended to the document, the browser will automatically generate the request. No image is actually returned by the controller script.

For binary data, or large amounts of data that can't fit into a URL, HTML forms are used. Data is stored in a text input box, the form's action attribute is set to the controller script, the method is set to POST, the form's target is set to an unused iframe (to keep the window from loading the action URL), and the form's encoding is set to "multipart/form-data".

Controller script to JavaScript proxy

Since a browser cannot accept inbound network connections, the JavaScript proxy must initiate all communication. When the JavaScript proxy polls the controller script for data (such as the next HTTP request to process), the response is a JavaScript file with the data set in variables that can be retrieved by the JavaScript proxy. Similar techniques are used by the Backframe toolkit.[14]

In essence, this is intentional XSS: the document is loaded from the randomly generated hostname, but the script is loaded from the attack server's secondary IP address (or secondary hostname). As a result, some anti-XSS filters might block this request. However, no XSS is required for a successful attack.

There are several other methods to transfer data from the controller script besides XSS. While the same-origin policy prevents most explicit data exchange, JavaScript can still infer data about content from different origins. For example, the dimensions of an image are accessible in JavaScript, regardless of which server provided the file. This allows for a series of images to be requested by JavaScript with one byte encoded in the width and one byte in the height. Firefox (and perhaps other browsers) will load a bitmap with headers, but no graphic content, allowing the files to be stripped down to 66-bytes. While this technique is slow, it is effective. Considering that cross-domain image loading is very common on the Internet, it would be extremely difficult to detect and block.

A similar technique tunnels data through dynamically loaded Cascading Style Sheets (CSS)[15,16]. Again, most data in a different origin CSS cannot be directly accessed by JavaScript. However, some data in a style class can be inferred once it is applied to a document component. Margin sizes are one example. Firefox allows margins to be set to millions of pixels, allowing at least two bytes of data to be encoded in each margin setting. Bulk data can be transferred by creating series of sequentially named classes. Once the style sheet is loaded, it is trivial for JavaScript to apply each class to a DIV tag, measure the actual margin sizes, and then decode the data. Since an unlimited number of classes can be defined in a single style sheet, performance is much better than the image dimension method, and approaches the XSS method.

Other Anti-DNS Pinning Attacks

Java LiveConnect

The Sun Java Virtual Machine (JVM) supports full network connections, but only a trusted applet[17] can connect to arbitrary hosts. If the code isn't trusted, it can only connect out to the origin server, but on any port. The JVM has its own DNS resolver and DNS pinning logic and is believed to be resistant to standard anti-DNS pinning attacks against applets[18]. LiveConnect[19,20] is a Firefox and Opera feature that allows Java applets to interact with the HTML DOM and allows

JavaScript to instantiate and interact with standard Java classes. When LiveConnect is used, the JVM pins the DNS after the webpage is loaded.

By coordinating a DNS change with the page load, an attacker can successfully launch an anti-DNS pinning attack without the DNS cache timeout required in the JavaScript / XHR method. More importantly the JVM supports full UDP[21] and TCP[22] sockets, and partially supports ICMP[23]. This means that virtually any application protocol can be supported: SSH, SSL, telnet, SNMP, database protocols, CIFS, etc.

A practical LiveConnect-based attack would be structured much like the XHR technique. Instead of an HTTP proxy accepting the attacker's requests, a SOCKS[24] proxy is used. To support a wide variety of attack tools, a generic SOCKS client[25] is tied to the attacker's IP stack. To improve performance, the socket reads and writes on the browser can be handled asynchronously[26,27] within JavaScript.

Adobe Flash

Because Flash does not implement any DNS-pinning[28], this is more properly considered a classic DNS-spoofing attack. Using Flash's socket functionality in ActionScript, it is possible to send arbitrary data over TCP. Two requirements significantly limit the flexibility of such an attack. First, in an odd throwback to old UNIX "security", Flash will only connect to TCP port numbers greater than 1023. Second, each response has to be in XML format and terminated by a null character, effectively making it a one-way transaction[29]. Despite these limitations, any action that can be performed during the first stage of data transmission is possible to implement. Some text-based protocols such as HTTP or SMTP are partially usable. Some exploits may allow the shell-code to be transmitted this way also.

Proxy Servers

If the victim browser is configured to use a proxy server, it will usually not resolve hostnames for requests, effectively disabling its DNS pinning functionality. Since all DNS resolution is performed by the proxy server, anti-DNS pinning attacks are product specific. However, proxy servers can easily run for months at a time, making it impractical to permanently pin the DNS cache. If a proxy server has access to the internal network, it can be used to perform XHR-based anti-DNS pinning attacks. It is conceivable that the HTTP CONNECT command could be used on a proxy to tunnel any TCP protocol with anti-DNS pinning. If the browser is configured to use a SOCKS v5 server, UDP protocols may be possible also.

Defense Against Anti-DNS Attacks

The most obvious defense against anti-DNS pinning attacks is to change the browsers' behavior so that DNS records are permanently pinned into the cache. Web servers do not go down often, and requiring the user to restart the browser

is not an unreasonable burden. However, this is not a panacea. It doesn't address browser-restart attacks, or any attacks using external DNS caches such as browser plug-ins or proxy servers.

Realistically, most companies cannot disable JavaScript across the board; too many common websites use it. Better web browser security policies would help with this. Firefox does support zone-based policies[30], but it is a well hidden feature that few know about, and it is lacking in granularity. Internet Explorer has better granularity for some features, but not for scripting. While IE allows XMLHTTPRequest to be globally disabled, it would be more useful to disable it for a single security zone without disabling all scripting.

The NoScript[31] add-on for Firefox presents some benefit, but is still lacking in granularity and is difficult for most users to manage. Deploying it across an enterprise would be very time consuming due to the high level of customization required.

Another Firefox add-on with JavaScript security features is LocalRodeo[32]. One of its features is a monitor of the browser's DNS cache; if an IP address changes (as part of an anti-DNS pinning attack), it will block it. It also attempts to detect and block JavaScript-based reconnaissance such as port scanning, fingerprinting, etc. However, LocalRodeo is a beta tool that most administrators should be reluctant to widely deploy in an enterprise. It also does not address browser plug-ins or proxy servers.

While it may become safer, it seems unlikely that downloading and executing code from anonymous strangers will ever be safe. As long as technologies like Java, JavaScript, ActiveX, browser plug-ins, and even email attachments exist, techniques will be available to bypass perimeter firewalls. More consistent application of security controls like host hardening & patching, strong authentication & encryption, and network segmentation will always provide significant protection.

[1] http://www.mozilla.org/projects/security/components/same-origin.html
[2] http://www.ietf.org/rfc/rfc2616.txt, section 15.3
[3] http://viper.haque.net/~timeless/blog/11/
[4] http://shampoo.antville.org/stories/1451301/
[5] http://msdn2.microsoft.com/en-us/library/ms175046.aspx
[6] http://www.cgisecurity.com/lib/XmlHTTPRequest.shtml
[7] https://bugzilla.mozilla.org/show_bug.cgi?id=297078
[8] https://bugzilla.mozilla.org/show_bug.cgi?id=302263
[9] http://www.w3.org/TR/html401/present/frames.html#h-16.5
[10] http://www.w3.org/TR/XMLHttpRequest/
[11] http://msdn2.microsoft.com/en-us/library/ms535874.aspx
[12] http://developer.mozilla.org/en/docs/XMLHttpRequest
[13] http://mgran.blogspot.com/2006/08/downloading-binary-streams-with.html
[14] http://www.gnucitizen.org/projects/backframe/
[15] http://www.bobbyvandersluis.com/articles/dynamicCSS.php
[16] http://www.irt.org/articles/js065/
[17] http://java.sun.com/sfaq/
[18] http://shampoo.antville.org/stories/1566124/
[19] http://developer.mozilla.org/en/docs/LiveConnect
[20] http://java.sun.com/products/plugin/1.3/docs/jsobject.html
[21] http://java.sun.com/j2se/1.5.0/docs/api/java/net/DatagramSocket.html
[22] http://java.sun.com/j2se/1.5.0/docs/api/java/net/Socket.html
[23] http://java.sun.com/j2se/1.5.0/docs/api/java/net/InetAddress.html#isReachable(int)
[24] http://tools.ietf.org/html/rfc1928
[25] http://www.hummingbird.com/products/nc/socks/index.html
[26] http://developer.mozilla.org/en/docs/DOM:window.setTimeout
[27] http://developer.mozilla.org/en/docs/DOM:window.setInterval
[28] http://www.jumperz.net/index.php?i=2&a=3&b=3
[29] http://www.adobe.com/support/flash/action_scripts/actionscript_dictionary/actionscript_dictionary867.html
[30] http://www.mozilla.org/projects/security/components/ConfigPolicy.html
[31] http://noscript.net/
[32] http://databasement.net/labs/localrodeo/