"Web Server Vulnerabilities and a Defense in Depth Strategy Using the Squid Proxy"

GSEC Practical version 1.4b

Jim Brewer, February 19, 2004

Abstract:

This paper will present a "Defense in Depth" strategy for a web server environment that includes the use of Squid. Squid is an open source caching HTTP proxy server available from www.squid-cache.org. Squid includes a rich set of access controls that can be used to restrict what passes through it in much the same way as a network firewall restricts what IP traffic it passes.

Included in this document are:

- A review of vulnerabilities and categories of vulnerabilities that are known to occur in web servers and explain how Squid may be used to eliminate or mitigate the
risks associated with these vulnerabilities.

- A "how-to" guide on obtaining, installing, and configuring Squid to provide maximum protection for your web server.

It has been slightly more than a decade since the web paradigm began its lightning charge to the popularity it has today. It was Christmas day, 1990 when Tim Berners-Lee, a consultant at the nuclear research facility CERN (Conseil Européen pour la Recherche Nucleaire) finished writing what is considered the first browser which combined his work in "hyperlinking" of text with his ideas of being able to link to text on disparate computers over a communications network. Berners-Lee and co-worker Robert Cailliau proved the worth of the new technology by making the CERN telephone directory the world's first web application. Instructions to obtain and run a copy of the web server developed at CERN were posted by Berners-Lee to a Usenet group in August of 1991 and web servers started springing up worldwide.

Jump ahead to 2004. There are now tens of millions of web servers in use. The home page of the popular search engine www.google.com claims to have over 3.3 billion pages indexed.

Anyone who has ever written a computer program of maybe a few thousand lines of code knows how easy it is to have "bugs" in their program and have it not handle data in the way it was intended. Extrapolate this to today's web servers that have millions of lines of code and imagine the opportunities to have data not handled in the way it was intended. Some of the problems are outright coding errors that make the server programs abort when encountering certain data streams. Most of today's vulnerabilities with web servers, though, are attributed to the creativity of the hacker community to find ways for the web servers to handle data in ways the web server developer did not intend.

A query of the vulnerability database maintained by the CERT® Coordination Center (http://www.kb.cert.org/vuls/) using the search term "web server" returns over 140 hits on already discovered flaws and vulnerabilities. It is quickly apparent that these vulnerabilities fall into a few easily identifiable categories. There are two recent reports of web server vulnerabilities that handily identify these categories. The Open Web Application Security Project (OWASP) (www.owasp.org) published their annual "Top 10" vulnerabilities report in January 2004. In February 2004, a commercial enterprise, WebCohort (now called Impervia (www.imperva.com)) published a report based on vulnerability assessments done by their "Applications Defense Center". Of note in WebCohort's report is that 90 percent of the web applications subject to their assessments contained common vulnerabilities that put in jeopardy the servers on which they ran or left subject to corruption the data the application handled. Some of the more pervasive vulnerabilities mentioned in one or both reports are discussed below.

Un-validated Input

This problem rests squarely on the shoulders of the web developers and programmers. No number of firewalls, intrusion detection systems, or even up-to-date patches for the web server can replace careful consideration on how to evaluate and edit the data being received by the web-based program. A web programmer might easily take a cue from those who configure firewalls and edit their input data to only allow the input they can reasonably expect and deny everything else. This can be as simple as checking a phone number to see that it has only numerics and expected separators like "-" and the correct number of digits for the phone system in the country where the phone number is based.

Careful evaluations and editing of the data presented to the web application is also the last line off defense against other attacks like Cross Site Scripting, Injection, or Parameter Tampering (below).

Parameter Tampering

This is a subset of input validation errors. In parameter tampering, though, there is more effort required on the part of the attacker other than just putting unexpected data in an input box. For example, since the web's HTTP protocol is stateless, a programmer might store information needed in the next step of a process in form variables that use the <hidden> tag. All someone needs to do to provide unexpected input to the program is to save an HTML page containing hidden variables, edit the file, reload it into the browser and submit the modified form. If the data saved in the hidden values happens to be the price of an item being purchased over the Internet, the hacker just did a "name your own price" transaction. If the data in the hidden variable was some homebrew form of session identifier, the hacker could change the data and possibly hijack someone else's session.

Simple avoidance of some parameter tampering can be done if a programmer just rechecks that the parameters being returned make sense compared to other parameters. In other cases where it is imperative to maintain state and it is possible to corrupt data if the state is not known, the application will need to be moved to an application server environment where state is maintained and variables remain server-side where they are out of the hacker's reach.

Cookie Poisoning

A simplistic example of cookie poisoning might be a parallel to the parameter tampering vulnerability. If a programmer passes data from one step in a process to another via HTTP cookies, the data in the cookie may be edited before it is retrieved by the server, giving the program the hacker's data instead of what was intended. Most often though, the term cookie poisoning is used when a web application has taken the extra step to maintain state via a session, probably via an application server. The problem arises when the session token, a cookie value that identifies a browser's current session context on the server, has a value that can be reverse engineered and changed by a hacker to take over another session. Most, if not all, of the currently available application servers that create session tokens do so in a way that is not decipherable in the lifetime of the session. The problem often arises when a programmer tries to develop his/her own session machine. Unless the programmer has a PhD in mathematics and experienced in devising sequences that are not predictable, the algorithm they come up with is probably predictable in some way and can be cracked.

Cross Site Scripting (XSS)

The WebCohort report, mentioned above, says that 80% of web servers are vulnerable to XSS attacks. The OWASP report from above indicates that 30% of reported vulnerabilities are XSS related. Either number causes alarm.

Cross Site Scripting is a hacking technique where a web client is enticed to click on a link of the hacker's design that is a hyperlink to another site, possibly one that the uses would have no reason not to trust the content that it would provide. The hacker's link has HTML tags in it, though, that identify a browser-side scripting section (<SCRIPT>, or other tags that can collect and submit data or execute program statements. <FORM>, <OBJECT>, <APPLET>, and <EBMED> tags have all been identified as tags that might be used in XSS attacks). The supposedly trusted web server can sometimes be "tricked" into sending these tags to the user's browser. The following example of an example hyperlink that could be used to launch an XSS attack is from a February 2000 CERT advisory describing how such an attack could be mounted:

```
<A HREF="http://example.com/comment.cgi?
mycomment=<SCRIPT>malicious code</SCRIPT>"> Click here</A>
```
(from http://www.cert.org/advisories/CA-2000-02.html)

Here, example.com is a site in which the browser user has interest and has no reason not to trust. The comment.cgi program/page is one that the hacker has discovered will echo query parameters that are fed to it on a GET. The parameters are indeed echoed including the hacker-originated one containing malicious scripting.

The security policies written into browsers protect against script code having access to certain parameters that belong to a page from being read or set by a script from another site. Since, however, a site has been tricked into downloading an intruder's XSS code, the attacking code has access to parameters it otherwise could not see. This could be used to collect form data from the page, information about the session, information about the server which "owns" the page, or information about the user that is viewing the page. The script will forward the ill-gotten information to a location of the hacker's choice. Even if the attack via XSS does not collect data and forward it to another location, the technique can be used in an annoying fashion to alter the appearance of the page... a “web defacing” attack without having to hack into the originating web server and modify the HTML code.

Once more, an astute web programmer may design code to check inputs before using them to create a page. Newer versions of popular browsers are including checks against XSS. Some security products examine URLs before allowing them to be processed by the web server (more on that when we discuss Squid).

SQL Injection

Web applications have evolved from their original roots. Web pages designed in the mid-1990s were output-only. They displayed static data or images but accepted no input. Soon, web browsers were replacing client/server “fat” clients and doing interactive tasks. In the last few years, web browsers have become the interface for multi-tiered applications that interact with back-end databases. SQL injection is the “art” of placing SQL programming statements into what are intended to be data fields. If there are not sufficient checks on the input field before the data is plopped into a SQL statement which queries or updates a backend SQL-compliant database (MS-SQL Server, Sybase, Oracle, etc.), it is possible to pass SQL statements of the hacker's choice in place of the data intended to be in the input field. The key to the attack is to terminate a data input of a text field with a single quote character ('), the normal terminator for a text field in a SQL statement, insert SQL code to query or update the database, and follow it with a double dash sequence (--). Any data after the double dash is interpreted by SQL as a comment so the remainder of the legitimate portion of the SQL statement is ignored by the database server. So, a SQL statement that was intended to accept as input a customer number and return the customer's name....

```
SELECT name FROM customer_data WHERE cust_numb = '$custNum' ;
```

...where $custNum is an input from the POST of the browser form, can be used to do more than was intended. If, instead of just inserting a customer number in the custNum field of the form, a hacker were to fill in the field with a customer number (real or not, doesn't matter) followed by a SQL statement of his choosing and sufficient syntactical punctuation to make the query not blow up, the statement presented to the database server might look like this (bold data is what was entered in the HTML form)...

```
SELECT name FROM customer_data WHERE cust_numb = '12345';INSERT into
customer_data VALUES ('99876','Ima Hacker','myPassword');--' ;
```

The database server would return the originally intended “name” query but would also insert the bogus record into the database.

How did the hacker know the format of the customer_data table to insert his own record? Part of what is needed to make this hack work is error messages that are TMI-rich (TMI= too much information"). To get to the point where the hacker knew the customer_data table had three values... customer number, name and password, he had to be familiar with the types of error messages that the SQL access method of the web server would return and fashion queries that would exhort the server to return "helpful" error messages.

If this vulnerability were fixed at the application level, first the programmer should limit the size of the input field to the expected 5 characters of the cust_numb field. Further validation might be that if the cust_numb is expected to be all numeric characters, return an error if any alphabetics are included in the input. Alternatively, the entire POST could be rejected if the single quote or semicolon characters are in the input.

Improper Error Handling

In the SQL injection example above, the attack was made easier by the hacker being able to determine table structure from error messages. The more information an attacker has, the easier the web site is to hack. How often do you still see sites that do homebrew user authentication that have error messages that tell which parameter, user ID or password, was not accepted. Too much information! Ideally, from a security perspective, for any error that occurs, the connection should be terminated without an error message. Too little information though, and a help desk has to answer too many calls without enough information to help the user. Keep in mind, though, that more information than is necessary gives the nefarious user a road map of your infrastructure or data structures.

Default error message should be edited. The site administrator should take out any information from a default error page or template that identifies infrastructure (example: server names or IP addresses). Take out any template parameters from canned applications or servers that return to the user the error number generated by the server or application. It's not much harder for the hacker to figure out the error if he looks up the error number in an API reference than if the error page template also fills in the text of the error code. Take out any code that echoes user information. A programmer should follow these same guidelines when generating custom error messages.

Keep in mind that as-complete-as-possible information should be kept in administrator accessible log files. Without sufficient and complete logging, there may not ever be enough forensic information to catch a cyber vandal or maybe even ever know the attack happened.

<u>Insecure Storage</u>

Just as in the case where a programmer tries to invent their own user authentication package, any attempt from a programmer to invent their own encryption algorithm is not time well spent. There are innumerable experts in cryptography that have spent their life devising encryption schemes that are uncrackable (in a time frame where the data would still be useful). There are available API packages that implement these schemes. That a programmer would use his dog's name as a hash key (this one was for real in my past) so that he had an easy symmetric encryption algorithm to store data in a cookie is just asking for a hack.

<u>Denial of Service (DoS)</u>

This is an extremely big category of computer attack. It encompasses any number of methods that keep a computing component from doing the job for which it is intended. Attackers can do this by flooding the device with data faster than it can be serviced, flooding a network segment with so much data that legitimate packets never get through, or exploiting a software flaw that will make a server or other device crash. Although DoS attacks get much immediate attention when they happen because the information system that is being attacked is obviously impaired or out of service, it is quite likely the previously mentioned attacks that work at the data level have more potential to allow a hacker steal something of value and be long gone by the time the breach is discovered.

Generally speaking, the best that can be done to avoid or reduce the occurrence of DoS attacks is to keep current with vendor patches that fix known coding flaws that will crash a server, and implement network device (i.e.: firewall) configurations that start dropping the types of packets used in an attack when their frequency reaches a certain threshold. And (how many times have you heard this) shut down all unnecessary services that leave a port open that can be used for an attack vector.

Now, I introduce Squid. The purpose of enumerating common vulnerabilities in the beginning of this paper was to provide a list of vulnerabilities that Squid may be able to mitigate.

Rather than find a way to paraphrase the Squid FAQ page on the question "what is Squid", here is a quote from the FAQ page.

> **Squid is a high-performance caching proxy server for web clients, supporting FTP, gopher, and HTTP data objects. Unlike traditional**

> caching software, Squid handles all requests in a single, non-blocking, I/O-driven process.
>
> Squid keeps metadata and especially hot objects cached in RAM, caches DNS lookups, supports non-blocking DNS lookups, and implements negative caching of failed requests.

This document will concern itself with only the HTTP capabilities of Squid. If anyone is running an FTP server exposed to the Internet, they may want to consider a copy of Squid to proxy FTP for the same reasons that will be outlined for using Squid for HTTP. As far as gopher services, this 1980's vintage predecessor to HTTP has all but vanished since HTTP on the WorldWideWeb became the popular standard.

Although Squid is more often used in a “traditional” forward proxy mode where it is used to allow clients blocked from the Internet by a firewall to access Internet resources, in the implementation described here, Squid will be used in what is colloquially known as a “reverse proxy” or in the terminology of RFC 3040 (“Internet Web Replication and Caching Taxonomy”), a “surrogate”, serving web pages on behalf of the actual origin server.

The primary reason for using a reverse proxy in your web server network is to offer additional layers of “Defense in Depth” between your web server and the hacker community of the Internet. Instead of a more traditional approach where the path of attack to a web server is Internet --> firewall --> web server where the web server lives in the DMZ, adding a reverse proxy makes the path to your web server Internet --> firewall --> reverse proxy --> firewall --> web server. Of course any backend application server or database server is also two more boxes distant from the Internet.

In return for any latency/delays that a reverse proxy may add, Squid, because it is also caching server, may actually improve response time for frequently used static web server objects. Once a page or image is cached in the reverse proxy, the originating web server is more lightly loaded because it no longer has to serve the static objects. When Squid is used in reverse proxy mode, the configuration files call this “accelerator” mode. Because of its highly indexed cache store structure and having the “hot”/frequently-used objects stored in RAM, Squid can serve pages faster than the origin web server.

An additional task that Squid can perform on behalf of the origin web server that may further lighten the load on the web server is Squid's capability of terminating Secure Sockets Layer (SSL) connections. In actuality, if you want Squid to perform the access control or redirection functions that will be described shortly, and your security needs dictate use of SSL, it is necessary for SSL connections to be terminated at Squid so the data in the packets can be analyzed. A further reality check of your security needs may indicate, though,

that you do not want to leave the data stream unencrypted between the reverse proxy and the origin server. Squid is also capability of also being an SSL client to keep the data encrypted on the path to/from the origin server.

A less technical reason for using a reverse proxy may be to shield a web server that is known to have security issues. Perhaps you have an older version of a web server for which there are a number of known vulnerabilities and you find your application doesn't run correctly on the new version, or, the newer web server requires newer hardware to accommodate its hunger for memory or CPU speed. A temporary partial mitigation of the risk may be accomplished by interposing a reverse proxy between the known bug-laden web server and the Internet.

So, why Squid over any other reverse proxy product? There may be financial considerations. Squid is open source software freely distributed under the GNU General Public License (see http://www.gnu.org/copyleft/gpl.html if you are not familiar with GNU GPL) and is essentially free. Readers familiar with open source products will immediately remember that the Apache web server also works in reverse proxy mode and is free. But Apache is primarily a web server that may be configured to work in reverse proxy mode. What makes Squid ideally suited to a Defense in Depth strategy is its rich set of access controls. Of the nearly 200 configuration directives for Squid, about 25 of them are different types of Access Control Lists (ACLs) that can be applied to about 20 types of access control rules (the "nearly" and "about" qualifiers are because the configuration directives set changes with every new update of the Squid software). The usefulness of some of the built-in ACL types is a bit dubious. The "ident" ACL that probes the source client's port 113 ident service (see RFC 1413) to obtain the identity of the client seems too easy to spoof to be useful in a security context. The "arp" ACL is meant to allow or deny a list of link-level ethernet addresses (MAC addresses) to the proxy. In the accelerator mode, the only MAC address the reverse proxy should be seeing at its input is the firewall or router immediately adjacent to it. Perhaps you could use this to assure the only client MAC address is the adjacent network device, but don't forget to reconfigure the proxy if the firewall or router ethernet card ever changes. But that still leaves about 23 ACL to configure as sentinels at the door to your web server. The most likely to be useful to provide security lock-down in acceleration mode are:

src – limit the source IP addresses, either by individual address or by subnet. In accelerator mode, this is most likely to be useful if you only want a group of people at a known remote location (meaning, you know their subnet) to be granted or denied access.

dst - limit the destination IP address(es). This is mandatory. In accelerator mode, this would be set to allow only the destination address of your backend web server(s). Without this, the hacker community would have their way with

your proxy by proxying their hack attacks to any IP address through you. (Of course, you are probably smart enough to have your firewall configured to stop most or all outbound connections.)

srcdomain and srcdom_regex – Similar to src (above) but instead of the IP address, this ACL acts on the character string, or a regular expression for part of the string, of the fully qualified domain name (FQDN) of the remote client. In order for this to work, the clients address must be must be configured in DNS to respond to a reverse lookup.

dstdomain and dstdom_regex – similar to dst (above) but acts on the hostname in the supplied URL. A security reason for this might be to force connections to your server to be in a particular domain for cookie reasons. With a host file entry on the client computer, the host string in the URL can be anything. If your site works with cookies though, cookies can only be read from the domain (or server, depending on how the cookie was created) where they were created. Using this option will allow only connections to your domain so cookies will only be created in your cookie domain.

port – list the IP ports/sockets that you want to allow or deny. In the case of the http_access rule (described later), this might be the list of allowed destination ports. In accelerator mode, this should be a short list... only ports that your web server(s) is(are) listening on.

method – This refers to HTTP methods like GET, POST, etc. Since some methods have known vulnerabilities, this ACL will be explored further when a sample configuration is proposed.

proto – In a strictly http and/or https environment, the protocol ACL can be used to limit traffic to http and or https and stop attempts to use ftp, gopher, ldap, or other more obscure protocols from being processed by the proxy.

time – If your web site is out of service on a regular schedule, this would be useful to generate an error if a client tries to connect in the outage window. With proper configuration of a custom error message, the client would know in a more elegant manner that you are down instead of just getting the default “can't connect” message.

proxy_auth –Should you decide to authenticate users entering your proxy, this ACL must be used to allow or deny particular authenticated users. This ACL works with the proxy authentication information in the request headers.

url_regex and urlpath_regex – Here is some of the REAL power of Squid. In the upcoming example configuration, we will see how to thwart a number of attacks by using regular expressions to find signatures of attacks and stop them at the proxy. url_regex works on the entire URL urlpath_regex works on the URL after (to the right of) the host name.

One more access control feature that is powerful is the External ACL. Squid may be configured to pass information to a program that you write. The external program returns to Squid whether or not the external ACL's conditions have been met. Squid will pass to an external ACL program much of the same information that the built-in ACLs work with like source address, destination address, protocol, port, etc., but the external ACL is the only one that deals with header variables. An external program may look for certain header values and allow or stop access based on the header variable's value.

Access Control Lists are just that.... lists. In order to be used to control the flow of data though the proxy the ACLs must be used in Access Control Rules.

The rules for HTTP proxies are quite analogous to the rules used network firewalls. A proper rule set will allow a limited selection of packets through the proxy and deny all others. The following is an example of how to apply ACLs to Access Control Rules:

First, two ACLs...

```
		acl All src 0/0
		acl MyOriginServers dst 192.168.16.0/29
```

The 0/0 notation is shortcut notation for “any IP address in any subnet”, All ACL means “a connection from any IP address”. The dst ACL could also have been written with a list of the individual addresses of the origin servers in the subnet. The ACL could have been:

```
acl MyOriginServers dst 192.168.16.1 192.168.16.2 192.168.16.3 192.168.16.4  ---->
192.168.16.5 192.168.16.6 192.168.16.7
```

separating the arguments with spaces, and covered all possible server on the subnet. (Note: there is no line continuation character in the squid configuration file, squid.conf. The entire directive above would have to be on one line.) Of course, if there were only one or two origin server, you would only list the one or two addresses in the ACL.

Now to apply the ACLs to rules:

```
		http_access allow MyOriginServers
		http_access deny All
```

If there were no other rules in the configuration, the above rule set would allow any packet with a destination of the origin servers and stop all other packets that it saw. An alternate ACL would be “ acl MyOriginServers destdom_regex -i .mydomain.net$” interpreted as “any FQDN ending in .mydomain.net”. The Access Control Rule using the MyOriginServers ACL would remain the same. (If

you are unfamiliar with the syntax or regular expressions, try a text such as Jeffrey Friedl, Mastering Regular Expressions, O'Rielly, 2002. For a more lightweight source, most PERL programming texts include a chapter on regular expressions.)

When Squid is trying to determine whether to allow or deny a packet, it will descend through each rule in squid.conf and stop as soon as a packet matches a rule. That one rule that was the first one to match the incoming packet is the only one that will decide and access or deny. If no rule matches, Squid will take the current default. Huh...??? Current default...??? A default is not always the default??? A peculiarity of Squid's Access Control Rules is that there is no one default rule of accept or deny. At the end of processing a list of rules, if none has matched, Squid will assume the opposite of whatever the last rule was that was checked. If the last rule was an "allow" rule, the default behavior will be to deny the packet. If the last rule was a "deny" rule, the default will be to accept the packet. Of course the best way to keep this apparent ambiguity from being in play is to always end the rule set as it was done in the trivial example above and make the last rule of a set to be a "deny All". That way there will be at least one match to every packet and there will be no default behavior.

It was mentioned earlier that Squid will terminate SSL connections. If you are used to dealing with server SSL certificates using the GUI wizards in commercial web server products like IIS or Sun One, you will be in for a shock. The tool you are most likely to be using to manage server certs for Squid is the command line "OpenSSL" toolkit. OpenSSL comes packaged with many Linux distros and can be obtained as source code to compile on any other *nix platform. Windows users can venture further into the world of open source by installing the CygWin Linux-like shell on their Windows computer and use the OpenSSL tools that are in CygWin. If you have no experience with OpenSSL and find yourself needing to use it try a book like the one from authors Viega, Messier, & Chandra, Network Security with OpenSSL,O'Reilly & Associates, June 15, 2002. Most of this book deals with using OpenSSL's APIs to write your own SSL-enabled networking tools, but one chapter is a very helpful cookbook for using command line OpenSSL to manage certificates. Your Squid will need at least one server cert signed by a recognized certificate authority (Verisign, Thawte, etc.) if Internet users will be connecting to your reverse proxy with https protocol. If your Squid will be hosting multiple URL FQDNs, a separate server cert will be needed for each FQDN host name and you will need to design multiple https_ports directives in squid.conf (more on that later) that define IP address, host name, and port combinations that match each cert.

[For the adventurous only: You may be able to devise a scheme where one FQDN host name is shared among multiple origin servers by using a different URI to denote which origin server is to receive the packets. This would most likely require the addition of a "redirector" to your Squid to steer the different

URIs to their corresponding origin servers. Use of a redirector is described in Squid documentation and will not be engaged here.]

[For the extremely adventurous only: You may be able to use conversion routines in the OpenSSL tool to convert the PKCS#7 certificates and their private keys most likely already in use on your origin server(s) into the PEM format needed for Squid. Before attempting that, though, check your contract with the certificate authority that signed your server cert to see if that use of the cert is allowed.]

Now, let's actually build a system.

Hardware:

Squid is not a CPU killer. It can be implemented on a box of moderate power. A computer that has become underpowered and cast aside from another server duty may be quite suitable for Squid. Squid is also not extensively multi-threaded so a multi-CPU box will give little additional kick. What is important for Squid is lots of memory/RAM and fast disk access. The above-mentioned cast aside server probably has SCSI disks and either a lot of RAM or slots to put in more RAM. Do not try to use RAID to make squid fault tolerant. If you have an intelligent disk controller that will give you better read performance if you mirror your data, then do so, but if your Squid box should suffer a catastrophic disk crash, no irreplaceable data will have been lost. Your Defense in Depth strategy most certainly includes having a standby box to replace a smoked box. Plug in the standby, get your service back on line and let the new Squid start re-populating its cache as objects from the origin server get accessed.

When deciding on a disk configuration for your Squid server, take a page from database server wisdom. Put the cache storage on the largest, fastest disk configuration you can muster (but no bigger than some percentage over your calculations of the maximum space needed to cache all your cachable pages). Put the logs on the biggest, cheapest disk configuration you can, as long as it is separate from drive(s) used for cache storage (to avoid disk contention). From a security perspective, you should be doing extensive logging. Make sure the log disk will hold a few months of logs or make sure you have a good strategy for archiving older logs.

Network-wise, the squid server should have at least two Network Interface Cards. One of them will be the path from the Internet-facing firewall. The other will be the path to the firewall out of the DMZ toward the web server. It may also be prudent to include a third NIC that is used strictly for administrative tasks like backups and SSH access to the shell. The administrative network will have no (!!) access to or from of the Internet.

The target operating system for Squid is Unix. Most any *nix variant will do. If you already have support in-house for Solaris or AIX, by all means use that O/S. If you are budget minded, use Linux. Since Squid is distributed as source code and you will be custom compiling it for your server, use whatever operating system fits best in your environment. But what if your web server is already on Unix. Defense in Depth says you may want to put Squid on a Windows ® box to keep the hacker from using the same vulnerability to traverse multiple layers. There is a member of the open source community that is providing Windows compatible installations of Squid. See Squid's Internet site at www.squid-cache.org for more information.

(Authors note: To set up Squid to learn it for this project, I was able to purchase a couple of 400 Mhz. dual processor Dell servers with 256 Meg. of RAM each via eBay for just over $100 each. I loaded RedHat 9 Linux on one box for Squid and loaded Windows 2003 with IIS 6 on the other box to act as my origin server. My general-purpose personal computer was commandeered to act a the Internet-facing firewall running IPTables on RedHat 9. Although I, of course, never benchmarked this configuration for speed, everything worked out fine for this as a "sandbox". If I were using the Dell boxes in a production environment, I would probably add more RAM and add more disk than the two 9 Gig. SCSI drives they came with and see how the load testing turned out.)

⑩Operating System

Whatever O/S you decide to use, don't let it be the weak link in your defenses. Immediately patch it to the current vendor patches ("vendor" being used loosely in terms of Linux) and harden it. At a minimum use the hardening benchmarks from Center for Internet Security (CIS) appropriate for your system (www.CISecurity.org). Think you have it sucure....?? Throw the Nessus security scanner at it to see what holes it finds (www.nessus.org).

⑩Compiling:

If you are a veteran of using open source software, you are already familiar with the "gunzip, tar xvf, configure, make, make install" mantra that must be recited with every package you wish to compile into your system. If all you've ever done is point/click a GUI, it is now time to get back to computers' roots and get with the command line.

Squid is downloaded from www.squid-cache.org or one of its mirror sites in gzip (.gz) format. Use "gunzip <zipfilename>" to extract the tar file and "tar xvf <tarfilename>" to create the distribution directory structure. (For the aforementioned point/click-only GUI hounds, this is the time to become familiar with the "man" page system in Unix. Type "man tar" and/or "man gunzip" from a command window to view the manual pages on tar and gunzip. If you need to step even further back, type "man man" to learn how to use man.) Change your working directory into the directory created by un-tar'ing the Squid distribution. You will remain in this directory to do all the work to compile.

This how-to is not trying to build the most complex proxy. Of the nearly fifty configure-time options that can be set, the only one that will be recommended is to add SSL support to the default configuration. Even if you do not initially plan to implement SSL, it is something that is likely to be needed later, so do it now so you don't need to start from scratch later. So, the first command to build the software for Squid will be

"./configure --enable-ssl" (that is a dash, dash in front of "enable").

If the configure script did not find all the needed prerequisites and generated errors, you will have to deal with those before moving on. Having had luck with the configure step, remain in the same directory and type "make". On my test system this step took the better part of a half hour to compile all the Squid source code, so take some time to browse the Squid FAQ pages and gain confidence in your skills with Squid. When that completes successfully, the final command is "make install" which installs Squid in the default location of /usr/local/squid.

⑩squid.conf

As previously mentioned, the main purpose of this how-to is to describe implementation of the security aspects of Squid. Of the nearly 200 directives that can be in squid.conf, we will only deal with the ones to get an accelerator running and define the best access controls to mitigate the most vulnerabilities. There will be no attempt here to optimize the configuration or to set up Squid to work in the sibling/parent hierarchy of a proxy array.

The configuration directives will be described in the order they appear in the default squid.conf that was dropped by the install process. Be sure to make a back of the default file before making changes.

Change directory to /usr/local/squid/etc. Use your favorite editor to open squid.conf (uh.... no, notepad is <u>not</u> an option here. don't worry, you'll learn "vi" after a while).

http_port This is the listening/input IP address and port for your accelerator. The likely choice for this is port 80 on the interface from the Internet-facing firewall. If you will be listening on more than one port, the additional ports will be listed on the same line with a space separator. The configuration of this directive is different between versions 2.x (2.5 is the current 2.x version) and the 3.0 version (still actually considered beta/pre-stable). This paper was written using a Squid 3.0 test system. Where possible, the version 2.5 differences will be noted. In 3.0 a few options have been moved to the http_port directive that

used to be separate directives.

```
http_port 192.168.8.2:80 accel defaultsite=web1.mydomain.com
```

(Listens for http requests on port 80 on a particular interface, enables acceleration to the origin server or web1.mydomain.com)

(in ver 2.5, the accelerator options were on the httpd_accel directive)

https_port   Remember, to configure Squid to listen for https SSL requests, you must have a server cert in pem format.

```
https_port 192.168.8.2:443 cert=/usr/local/etc/secure/server.pem
defaultsite=web1.mydomain.com cipher= MEDIUM HIGH
```

(Remember, all config directives must be on one line. In some cases the lines is these examples may line wrap but that is not allowed in the actual squid.conf) (Listens on port 443, uses the CA provided server cert in a file named server.pem, accelerates to the origin server web1.mydomain.com, and will allow only 128-bit and above ciphers to be negotiated (remember, anything less that 128-bit ciphers may be broken in a short amount of time. Considering the speed of today's computers' speeds and the technology available to crackers, 40 and 56 bit ciphers are considered extremely weak.) To further fine tune the allowed ciphers, see the "man ciphers" manpage to see what OpenSSL supports.)

access_log   Since logging is important in a secure environment, make sure this param is set and that it points to the special log file system set up to hold weeks worth of logs.

```
access_log /logfs/access.log
```

cache_peer   I only mention this directive because in version 2.5, Squid was not able to open an SSL connection directly to an origin server. The workaround was to use the cache_peer directive pointing at the origin server and treating it a a parent cache.

no_cache This is actually a specialized access control rule for keeping certain URIs from being cached. The original intent was to use it to not cache dynamic pages. Additionally, if there are any objects for which you may consider it a risk having a copy of them sitting in the cache of a box only a firewall away from the Internet, you should define a regex ACL that describes that object and include it in a no_cache rule.

```
acl QUERY urlpath_regex cgi-bin \?
no_cache deny QUERY
```

(This example is stolen directly from the default squid.conf. You may want to copy this syntax for another ACL/rule pair for jsp or other dynamic pages, or as mentioned above to keep an especially critical page from being held in cache.)

acl **OK, here is the promised meat/potatoes of locking down your origin server with Squid.**

First, some housekeeping with the default configuration. Since the more common use of Squid is as a forward proxy, there are ACLs and rules included by default that allow actions that are common to forward proxies. The default rules allow access to a few low port numbers where http services are generally hosted and the high "ephemeral" ports where unsafe services are less likely (?) to run. We need to change that default behavior so that the only allowed destination ports are the ones for our origin server(s). So create an ACL that defines the http port on the origin server:

```
acl origin_srv_port port 6780          # the dest port to our origin server
```

(In my test setup, I put the IIS instance on 6780 because it is an oddball unassigned port that has "80" in it.)

Now scroll down to the default "http_access" tags and find the line...

```
http_access deny !Safe_ports
```

(This line reads "deny access to any port that is not in the list of "Safe_ports" (the ! makes it a "not".))

Either delete this line or put a # in front of it to make it a comment-only line. Add a line that implements our ACL for our origin server port being the only allow port.

```
http_access deny !origin_srv_ports
```

Since we are not including this accelerator in a proxy mesh, all our requests will be handled directly with our origin server(s). There is an access rule called "always_direct" that must be set to let Squid forward to the origin box. The following example is done using the dstdomain ACL....

```
acl direct-servers dstdomain .mydomain.com
always_direct allow direct-servers
```

Since we set our default acceleration host to web1.mydomain.com in the http_ports and/or the https_ports directive(s), using the domain suffix as a filter works fine.

Another housekeeping item related to forward proxies, since SSL connections can't be broken open by the forward proxy to see what's in them, the only thing to do is to allow the http CONNECT method to tunnel through the proxy. Since in accelerator mode we want no tunneling, we need to disable this "feature". Find the line...

```
http_access deny CONNECT !SSL_ports
```

This rule incorporates the logical AND of two ACLs, one named CONNECT and the other named SSL_ports. What we want is to allow nobody to use the CONNECT method, so just clip the second qualifier off the rule so it now reads...

```
http_access deny CONNECT
```

(That hole is now plugged.)

We next would like to limit ingress to the accelerator so that only the NATed IP addresses coming from our firewall will be allowed source access. Find the comment line in squid.conf that says "# INSERT YOUR OWN RULE(S) HERE TO ALLOW ACCESS FROM YOUR CLIENTS". Scroll down to the empty space under the default example of an ingress rule and enter something of the following flavor...

```
acl local_net src 192.168.8.0/29
http_access allow local_net
```

Granted, the rather limited subnet mask in my test setup will only allow 7 incoming connections. When you pick a NAT pool in the firewall to your accelerator, you'll have to estimate how many connections you'll need and subnet accordingly.

One of the first things learned about a secure environment is that services with open sockets never run as "root". Squid nicely enforces that by, if root is the user that starts Squid, the effective user is setuid'd to the user "nobody". If your security plan calls for running as some other effective user, configure the cache_effective_user and cache_effective_group to a user and group of your choosing. File system access controls will have to be set so that your chosen effective user owns all the files and directories.

If, by some stroke of misfortune or carelessness, your proxy did get cracked by a hacker, one way to minimize the damage is to use the "chroot" directive. This passes a new file system root to the system level "chroot" command and Squid can no longer see anything but its own file space. Make sure, though, that any path-oriented configuration directives, for example, paths to log files, are changed to reference the chroot'ed path and not the path from the operating system's root.

OK. Now let's start locking this proxy down. Returning to the list of common vulnerabilities, let's start with “Cross Site Scripting” and build an ACL/rule combination that will thwart it.

The root of evil in XSS attacks is that a server gets tricked into relaying HTML tags back to the browser. If we can somehow block HTML tags from entering the server in the HTTP stream, the can never get bounced to the browser. Well, any HTML tag looks like <some_tag>. If we block the less-than and/or the greater-than symbols, XSSing is avoided. How about...

```
acl xss_zap urlpath_regex > <
http_access deny xss_zap
```

Zap!!!!! One threat mitigated.

Next, let's tackle SQL injection attacks. The operant characters there would appear to be the punctuations. Without the punctuation syntax it would not be possible to terminate the real SQL statement and insert anything of the hacker's whim. Let's pick the single quote character and the semicolon. Let's not include the dash character because it is a common character in URLs.

```
acl sql_inj_zap urlpath_regex ' ;
http_access deny sql_inj_zap
```

It would be helpful to also block the URL encoded versions of the above ACLs. That would make the XSS ACL...

```
acl xss_zap urlpath_regex > < %3c %3C %3e %3E
```

Notice that we have not yet hit any of regular expression's metacharacters. If any the following . ^ $ * + ? { [ ] \ | ( ) were part of our regex, they would have to be escaped with a backslash.

On a slightly different slant, with web technology still being developed, there are parts of the standards who's implementations are still young and having their share of bugs. In that vein, here are two ACLs that will be useful.

There have been a few attacks floating around involving flaws in Microsoft's FrontPage extensions. It is unlikely that anyone from the Internet will be legitimately accessing FrontPage extensions, so we can block their ingress with...

```
acl FP_ext_zap urlpath_regex vti_bin vti_pvt
```
(plus add an associated http_access rule)

A similar vulnerability that manifests itself as a DoS is in IIS implementations of the WebDAV (Distributed Authoring and Versioning) standard. The easiest way to block this would be to deny the HTTP methods that it uses, PROPFIND, SEARCH, MOVE, and BMOVE. Earlier we denied the CONNECT method. These methods would be treated similarly...

```
acl webdav_methods  method  PROPFIND SEARCH MOVE BMOVE
(and a deny rule)
```

If the box really needed to be locked down, all methods from RFC 2616 except for GET and POST may be able to be denied in a manner similar to above.

By now, you have the idea of how to protect against any vulnerability that is triggered by something in the in the URL path. To close out the topic of ACLs and rules, here are a few things to remember. It was mentioned earlier that order matters in access control. The first match found for an incoming packet allows or denies it. So, put rules that will be matched often near the top of the list and rules that will seldom be matched at the bottom to save search time.

So, let's test this configuration and see if this flies. Squid provides a syntax checker. Change to the /usr/local/squid/sbin directory and enter "./squid -k parse". Make note of and repair any problems.

The first time Squid is run, an option must be used to tell Squid to create its cache directories. (All of the following commands are run from the /usr/local/squid/sbin directory.) Enter "./squid -z". Squid will take a few minutes to initialize his directories. When it's done, you are ready, finally to run Squid. To run Squid in the foreground with debug output to the screen, enter "./squid -N -d2". Numbers higher or lower than 2 may be used in the -d option to get more or less debug information. Squid is stopped with the command " ./squid –k shutdown ".

At this point, you should be able to use a browser to connect through the accelerator to the origin server. The debug output on the screen from where you started Squid will show the progress of each request through the access control rules. When you are satisfied all is running correctly, start Squid with the option to run it as a daemon process "./squid -s".

That is it!! That is an introduction to the Squid cache proxy and how it fits into a defense in depth plan for your web servers. The more experience you have with the access controls, the better you will get at blocking potential attacks so that you have time for the more fun things in life.... like teaching users not to leave Post-It notes on their desks with passwords written on them.

Bibliographic references:

Duane Wessels, Squid: The Definitive Guide,O'Reilly, January 2004

Stanger, et. al,Hack Proofing Linux: A Guide to Open Source Security, Syngress, 2001 : Chapters 2, 9, and 10

RFC 2616,"Hypertext Transfer Protocol – HTTP/1.1",http://www.ietf.org/rfc/rfc2616.txt

RFC 3040,"Internet Web Replication and Caching Taxonomy", http://www.ietf.org/rfc/rfc3040.txt

"CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests", http://www.cert.org/advisories/CA-2000-02.html, Feb. 2000

"Denial of Service Attacks", http://www.cert.org/tech_tips/denial_of_service.html

" CERT/CC Vulnerability Notes Database", http://www.kb.cert.org/vuls/

"HTTP: A protocol for networked information: Predefined Methods", http://www.w3.org/Protocols/HTTP/Methods.html, 1992

Jeff Forristal,"Proxies Add a Protective Shield",http://www.nwc.com/1404/1404f4.html , March 2003

Impervia, Inc., "Only 10% of Web Applications are Secured Against Common Hacking Techniques http://www.imperva.com/company/news/2004-feb-02.html , February 2004

"OWASP top 10 vulnerabilities", http://unc.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf , Jan 2004

Edward Hurley, "Dangerous, familiar application vulnerabilities top list", http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci946187,00.html , January 2004

Chris Anley,"Advanced SQL injection in SQL Server Applications, http://www.nextgenss.com/papers/advanced_sql_injection.pdf , January 2004

Duane Wessels, "SQUID Frequently Asked Questions", http://www.squid-cache.org/Doc/FAQ/ , 2001

"Web History – Browsers", http://livinginternet.com/w/wi_browse.htm

"NGSSoftware Insight Security Research Advisory - WebDAV", http://www.nextgenss.com/advisories/iis_webdav_dos.txt , December 2003

Impervia, Inc., "What is Parameter Tampering", http://www.imperva.com/application_defense_center/glossary/parameter_tampering.html