Building a Secure Squid Web Proxy

By Mick Bauer

(Reprinted from the Linux Journal)

PART I

Consider the venerable Web proxy—back when the Internet was new to most of us, setting up a Web proxy was a convenient way to grant users of an otherwise non-Internet-connected network access to the World Wide Web. The proxy also provided a convenient point to log outbound Web requests, to maintain whitelists of allowed sites or blacklists of forbidden sites and to enforce an extra layer of authentication in cases where some, but not all, of your users had Internet privileges.

Nowadays, of course, Internet access is ubiquitous. The eclipsing of proprietary LAN protocols by TCP/IP, combined with the technique of Network Address Translation (NAT), has made it easy to grant direct access from "internal" corporate and organizational networks to Internet sites. So the whole idea of a Web proxy is sort of obsolete, right? Actually, no.

After last month's editorial, we return to technical matters—specifically, to the venerable but assuredly not obsolete Web proxy. This month, I describe, in depth, the security benefits of proxying your outbound Web traffic, and some architectural and design considerations involved with doing so. In subsequent columns, I'll show you how to build a secure Web proxy using Squid, the most popular open-source Web proxy package, plus a couple of adjunct programs that add key security functionality to Squid.

What Exactly Is a Web Proxy?

The last time I discussed proxies in this space was in my December 2002 article "Configuring and Using an FTP Proxy". (Where does the time go?) A quick definition, therefore, is in order.
The concept of a Web proxy is simple. Rather than allowing client systems to interact directly with Web servers, a Web proxy impersonates the server to the client, while simultaneously opening a *second* connection to the Web server on the client's behalf and impersonating the client to that server. This is illustrated in Figure 1.

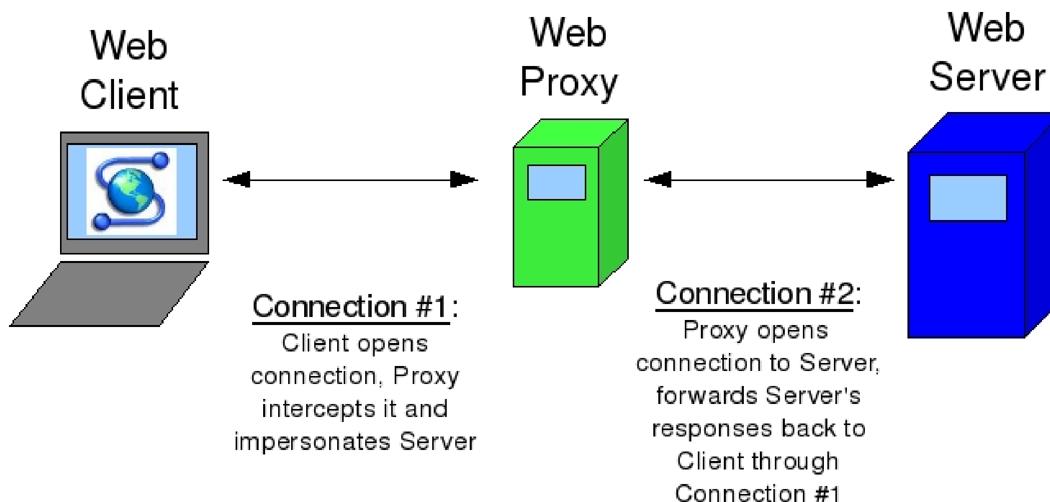

Figure 1
How Web Proxies Work

Because Web proxies have been so common for so long, all major Web browsers can be configured to communicate directly through Web proxies in a "proxy-aware" fashion. Alternatively, many Web proxies support "transparent" operation, in which Web clients are unaware of the proxy's presence, but their traffic is diverted to the proxy via firewall rules or router policies.

Why Proxy?

Just because nowadays it's *easy* to interconnect TCP/IP networks directly doesn't mean you always *should*. If a nasty worm infects systems on your internal network, do you want to deal with the ramifications of the infection spreading outward, for example, to some critical business partner with whom your users communicate over the Internet?

In many organizations, network engineers take it for granted that all connected systems will use a "default route" that provides a path out to the Internet. In other organizations, however, it's considered much less risky to direct all Web traffic out through a controlled Web proxy to which routes are internally published and to use *no default route whatsoever* at the LAN level.

This has the effect of allowing users to reach the Internet via the Web proxy—that is, to surf the Web—but not to use the Internet for non-Web applications, such as IRC, on-line gaming and so forth. It follows that what end users can't do, neither can whatever malware that manages to infect their systems.

Obviously, this technique works only if you've got other types of gateways for the non-Web traffic you need to route outward, or if the only outbound Internet traffic you need to deal with is Web traffic. My point is, a Web proxy can be a very useful tool in controlling outbound Internet traffic.

What if your organization is in a regulated industry, in which it's sometimes necessary to track some users' Web access? You can do that on your firewall, of course, but generally speaking, it's a bad idea to make a firewall log more than you have to for forensics purposes. This is because logging is I/O-intensive, and too much of it can impact negatively the firewall's ability to fulfill its primary function, assessing and dealing with network transactions. (Accordingly, it's common practice mainly to log "deny/reject" actions on firewalls and not to log "allowed" traffic except when troubleshooting.)

A Web proxy, therefore, provides a better place to capture and record logs of Web activity than on firewalls or network devices.

Another important security function of Web proxies is blacklisting. This is an unpleasant topic—if I didn't believe in personal choice and freedom, I wouldn't have been writing about open-source software since 2000—but the fact is that many organizations have legitimate, often critical, reasons for restricting their users' Web access.

A blacklist is a list of forbidden URLs and name domains. A good blacklist allows you to choose from different categories of URLs to block, such as social networking, sports, pornography, known spyware-propagators and so on. Note that not all blacklist categories necessarily involve restricting personal freedom per se; some blacklists provide categories of "known evil" sites that, regardless of whatever content they're actually advertising, are known to try to infect users with spyware or adware, or otherwise attack unsuspecting visitors.

And, I think a lot of Web site visitors do tend to be unsuspecting. The classic malware vector is the e-mail attachment—an image or executable binary that you trick the recipient into double-clicking on. But, what if you could execute code on users' systems without having to trick them into doing anything but visit a Web page?

In the post-Web 2.0 world, Web pages nearly always contain some sort of executable code (Java, JavaScript, ActiveX, .NET, PHP and so on), and even if your victim is running the best antivirus software with the latest signatures, it won't examine any of that code, let alone identify evil behavior in it. So, sure enough, the “hostile Web site” has become the cutting edge in malware propagation and identity theft.

Phishing Web sites typically depend on DNS redirection (usually through cache poisoning), which involves redirecting a *legitimate* URL to an attacker's IP address rather than that site's real IP, so they're difficult to protect against in URL or domain blacklists. (At any rate, none of the free blacklists I've looked at include a phishing category.) Spyware, however, is a common blacklist category, and a good blacklist contains thousands of sites known to propagate client-side code you almost certainly don't want executed on your users' systems.

Obviously, no URL blacklist ever can cover more than a tiny fraction of the actual number of hostile Web sites active at any given moment. The real solution to the problem of hostile Web sites is some combination of client/endpoint security controls, better Web browser and operating system design, and in advancing the antivirus software industry beyond its reliance on virus signatures (hashes of known evil files), which it's been stuck on for decades.

Nevertheless, at this very early stage in our awareness of and ability to mitigate this type of risk, blacklists add *some* measure of protection where presently there's very little else. So, regardless of whether you need to restrict user activity per se (blocking access to porn and so forth), a blacklist with a well-maintained spyware category may be all the justification you need to add blacklisting capabilities to your Web proxy. SquidGuard can be used to add blacklists to the Squid Web proxy.

Just How Intelligent Is a Web Proxy?

You should be aware of two important limitations in Web proxies. First, Web proxies generally aren't very smart about detecting evil Web content. Pretty much anything in the payloads of RFC-compliant HTTP and HTTP packets will be copied verbatim from client-proxy transactions to proxy-server transactions, and vice versa.

Blacklists can somewhat reduce the chance of your users visiting evil sites in the first place, and content filters can check for *inappropriate* content and perhaps for viruses. But, hostile-Web-content attacks, such as invisible iframes that tell an attacker's evil Web application which sites you've visited, typically will not be detected or blocked by Squid or other mainstream Web proxies.

Note that enforcing RFC compliance is nothing to sneeze at. It constitutes a type of input validation that could mitigate the risk of certain types of buffer-overflow (and other unexpected server response) attacks. But nonetheless, it's true that many, many types of server-side evil can be perpetrated well within the bounds of RFC-compliant HTTP messages.

Second, encrypted HTTPS (SSL or TLS) sessions aren't truly proxied. They're tunneled through the Web proxy. The contents of HTTPS sessions are, in practical terms, completely opaque to the Web proxy.

If you're serious about blocking access to sites that are inappropriate for your users, blacklisting is an admittedly primitive approach. Therefore, in addition to blacklists, it makes sense to do some sort of content filtering as well—that is, automated inspection of actual Web content (in practice, mainly text) to determine its nature and manage it accordingly. DansGuardian is an open-source Web content filter that even has antivirus capabilities.

What if you need to limit use of your Web proxy, but for some reason, can't use a simple source-IP-address-based Access Control List (ACL)? One way to do this is by having your Web proxy authenticate users. Squid supports authentication via a number of methods, including LDAP, SMB and PAM. However, I'm probably not going to cover Web proxy authentication here any time soon—802.1x is a better way to authenticate users and devices at the network level.

Route-limiting, logging, blacklisting and authenticating are all security functions of Web proxies. I'd be remiss, however, not to mention the main reason many organizations deploy Web proxies, even though it isn't directly security-related—performance. By caching commonly accessed files and Web sites, a Web proxy can reduce an organization's Internet bandwidth usage significantly, while simultaneously speeding up end-users' sessions.

Fast and effective caching is, in fact, the primary design goal for Squid, which is why some of the features I've discussed here require add-on utilities for Squid (for example, blacklisting requires SquidGuard).

Web Proxy Architecture

Suppose you find all of this very convincing and want to use a Web proxy to enforce blacklists and conserve Internet bandwidth. Where in your network topology should the proxy go? Unlike a firewall, a Web proxy doesn't need to be, nor should it be, placed “in-line” as a choke point between your LAN and your Internet's uplink, although it is a good idea to place it in a DMZ network. If you have no default route, you can force all Web traffic to exit via the proxy by a combination of firewall rules, router ACLs and end-user Web browser configuration settings. Consider the network shown in Figure 2.

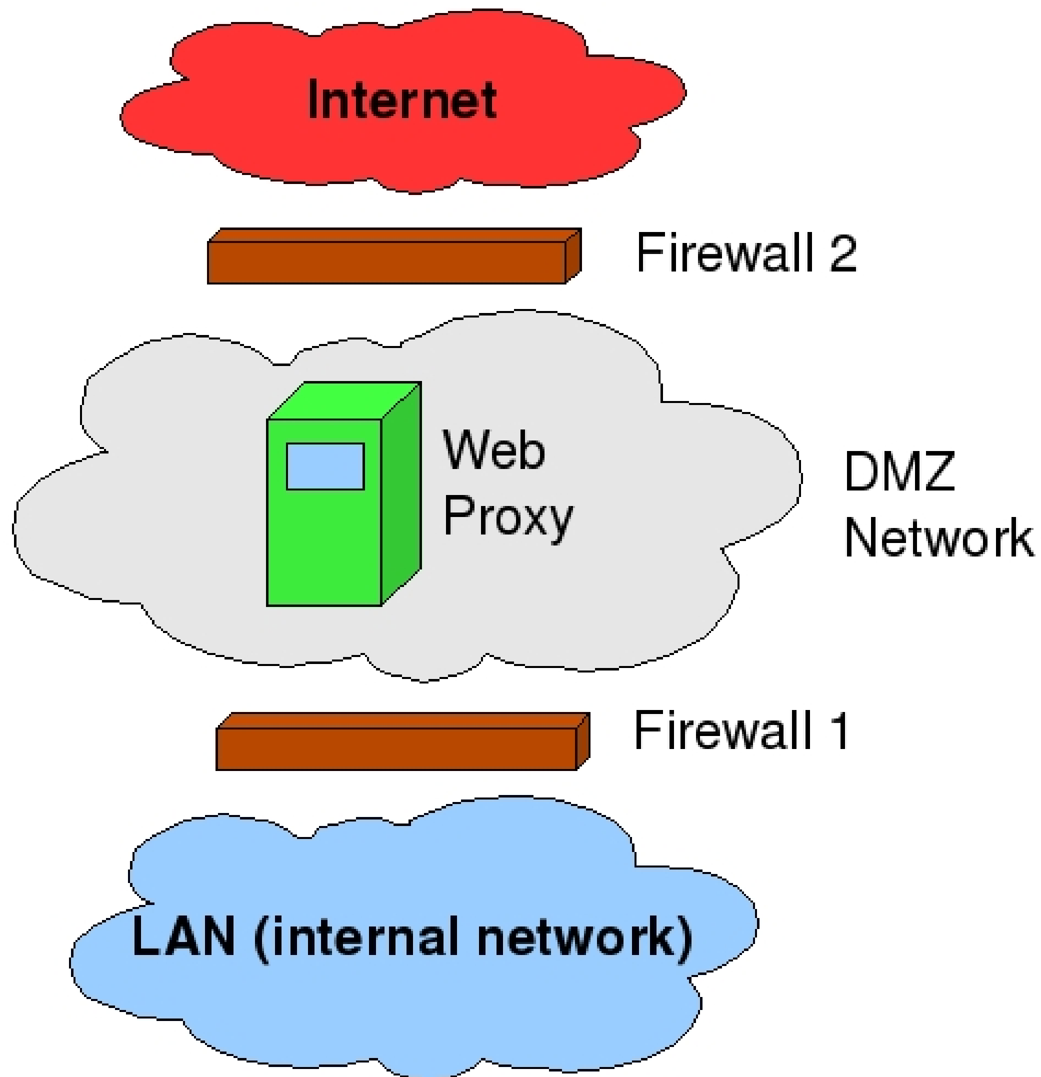

Figure 2
Web Proxy Architecture

In Figure 2, Firewall 1 allows all outbound traffic to reach TCP port 3128 on the proxy in the DMZ. It does *not* allow any outbound traffic directly from the LAN to the Internet. It passes only packets explicitly addressed to the proxy. Firewall 2 allows all outbound traffic on TCP 80 and 443 from the proxy (and only from the proxy) to the entire Internet.

Because the proxy is connected to a switch or router in the DMZ, if some emergency occurs in which the proxy malfunctions but outbound Web traffic must still be passed, a simple firewall rule change can accommodate this. The proxy is only a logical control point, not a physical one.

Note also that this architecture could work with transparent proxying as well, if Firewall 1 is configured to redirect all outbound Web transactions to the Web proxy, and Firewall 2 is configured to redirect all inbound replies to Web transactions to the proxy.

You may be wondering, why does the Web proxy need to reside in a DMZ? Technically, it doesn't. You could put it on your LAN and have essentially identical rules on Firewalls 1 and 2 that allow outbound Web transactions only if they originate from the proxy.

But, what if some server-side attacker somehow manages to get at your Web proxy via some sort of “reverse-channel” attack that, for example, uses an unusually large HTTP response to execute a buffer-overflow attack against Squid? If the Web proxy is in a DMZ, the attacker will be able to attack systems on your LAN only through *additional* reverse-channel attacks that somehow exploit user-initiated outbound connections, because Firewall 1 allows no DMZ-originated, inbound transactions. It allows only LAN-originated, outbound transactions.

In contrast, if the Web proxy resides on your LAN, the attacker needs to get lucky with a reverse-channel attack only *once* and can scan for and execute more conventional attacks against your internal systems. For this reason, I think Web proxies are ideally situated in DMZ networks, although I acknowledge that the probability of a well-configured, well-patched Squid server being compromised via firewall-restricted Web transactions is probably low.

Yet to Come in This Series

I've explained (at a high level) how Web proxies work, described some of their security benefits and shown how they might fit into one's perimeter network architecture. What, exactly, will we be doing in subsequent articles?

First, we'll obtain and install Squid and create a basic configuration file. Next, we'll “harden” Squid so that only our intended users can proxy connections through it.

Once all that is working, we'll add SquidGuard for blacklisting, and DansGuardian for content filtering. I'll at least give pointers on using other add-on tools for Squid administration, log analysis and other useful functions.

PART II

What We're Doing (Review)

As you'll recall from last month, a Web proxy provides a control point for restricting which external Web sites your users can reach. It allows you to permit Web access without allowing non-Web traffic (or even publishing a default route to the Internet), and it provides a convenient place to perform content filtering and transaction logging.

As you also may recall, unlike a firewall, a Web proxy doesn't need to be a physical choke point through which all traffic must pass for a physical path to the outside. Instead, you can use firewall

rules or router ACLs that allow only Web traffic, as a means of ensuring your users will use the proxy. Accordingly, your Web proxy can be set up like any other server, with a single network interface.

This is the case with the Web server I show you how to build in this and subsequent columns. This month, we focus on Squid itself; we'll cover add-ons like SquidGuard in future columns.

Obtaining and Installing Squid

So, where do you get Squid software? Naturally, the Squid Web site (see Resources) is the definitive source. But, because Squid has been the gold standard for Linux Web proxies for so many years, chances are it's a fully supported package in your Linux distribution of choice. If so, that's how I recommend getting it; it's easier to keep it patched that way, and you'll have greater assurance of stability and compatibility with the other things on your system.

On Ubuntu and other Debian variants (not to mention Debian itself), you need the packages squid and squid-common. On Red Hat and its variants, you need the package squid. And, on SUSE and OpenSUSE systems, you need squid.

At the time of this writing, all three of these families of distributions (Debian, Red Hat and SUSE) are maintaining separate packages for Squid version 3; the packages cited above are for version 2. This is because although the Squid development team recently declared Squid 3.0 to be a stable release (in November 2008), at the time of these three distributions' most recent production releases, Squid 3.0 still was considered to be a beta code branch, with 2.6 or 2.7 as the preferred production versions.

On the one hand, by the time you read this, Squid 3.0 (maybe even 3.1, which is in beta right now) may be mainstreamed into your Linux distribution of choice. On the other hand, maybe not. So for now, I'm going to use examples from Squid 2.6.18, the version on my Ubuntu system. They still should be perfectly valid for later versions—generally, later versions have additional options and features, not replaced options. I can cover Squid 3.0 in a future column.

I leave it to you to use the package manager of choice to install Squid packages on your RPM-based system, but on Debian-based systems, the most direct way is usually with the command:

```
bash-$ sudo apt-get install squid
```

(apt-get automatically will determine that it also needs squid-common and will install that too.)

By the way, you do not need to install Apache or any other Web server package on your Squid server, unless, of course, you're also going to use it as a Web server or want to use some Web-based administration tool or another. Squid itself does not need any external Web server software or libraries in order to proxy and cache Web connections.

Configuring Squid: Basic Functionality

Creating a basic, working configuration for Squid isn't much harder than installing it. Like so much else in Linux, it's a matter of making small changes to a single text file, in this case, squid.conf. In all three distribution families I mentioned, its full path is /etc/squid/squid.conf.

To get started, first open a command window, and back up the default squid.conf file (non-Ubuntu users can su to root and omit the sudo from these examples):

```
bash-$ cd /etc/squid
bash-$ sudo cp squid.conf squid.conf.default
```

Next, open squid.conf with your text editor of choice. You actually may prefer a graphical editor, such as gedit, but I've always used vi for its simplicity and ubiquity—if it's UNIX-like, it's got vi.

(Note to the emacs-loving alpha geeks among you: yes, emacs is more powerful; it's written in LISP; God kills a kitten every time someone installs Gvim; you win! But, I still like vi.)

Believe it or not, all you need to do to get Squid running is add two lines to the ACL (Access Control List) section of this file: an object definition that describes your local network and an ACL allowing members of this object to use your proxy. For my network, these lines look like this:

```
acl mick_network src 10.0.2.0/24
http_access allow mick_network
```

The first line is the object definition. The acl signifies that I'm about to define an ACL object. mick_network is the name I've chosen for this object. src means that it represents the IP address or range of addresses of hosts initiating TCP transactions with my proxy (that is, proxy clients). Finally, 10.0.2.0/24 is my LAN's network address in CIDR notation, which in this case translates to “the range of IP addresses from 10.0.2.1 through 10.0.2.254”.

The second line declares an actual ACL: allow transactions involving the object mick_network—that is, transactions initiated by hosts having IP addresses in the range 10.0.2.1 through 10.0.2.254.

If more than one network address comprises your local network, you can specify them as a space-delimited list at the end of the acl statement, for example:

```
acl mick_network src 10.0.2.0/24 192.168.100.0/24
```

Because ACLs are parsed in the order in which they appear (going from top to bottom) in squid.conf, do not simply add these acl and http_access lines to the very end of squid.conf, which will put them after the default “http_access deny all” statement that ends the ACL portion of the default squid.conf file. On my Ubuntu system, this statement is on line 641, so I inserted my custom acl and http_access lines right above that.

In case you haven't guessed, all is a wild-card ACL object that means “all sources, all ports, all destinations” and so forth. Any transaction that is evaluated against any http_access statement containing any will match it, and in this case, will be dropped, unless, of course, it matches a preceding http_access line.

Now that you've created an object and ACL for your local network, you should save squid.conf and then restart Squid by typing this command (see earlier note about su root shells vs. sudo):

```
bash-$ sudo /etc/init.d/squid restart
```

In fact, if you're editing squid.conf from a sudo vi squid.conf session, you don't even need to leave your editing session; just do a :w to save your work, then type :! /etc/init.d/squid restart to restart Squid from within vi.

To test whether things are working, you need to configure a machine other than the proxy itself to use your proxy. (Squid comes configured by default to allow transactions from 127.0.0.1, the local loopback address, to be proxied.)

Figure 1 shows the dialog for setting up Firefox to use our example proxy.

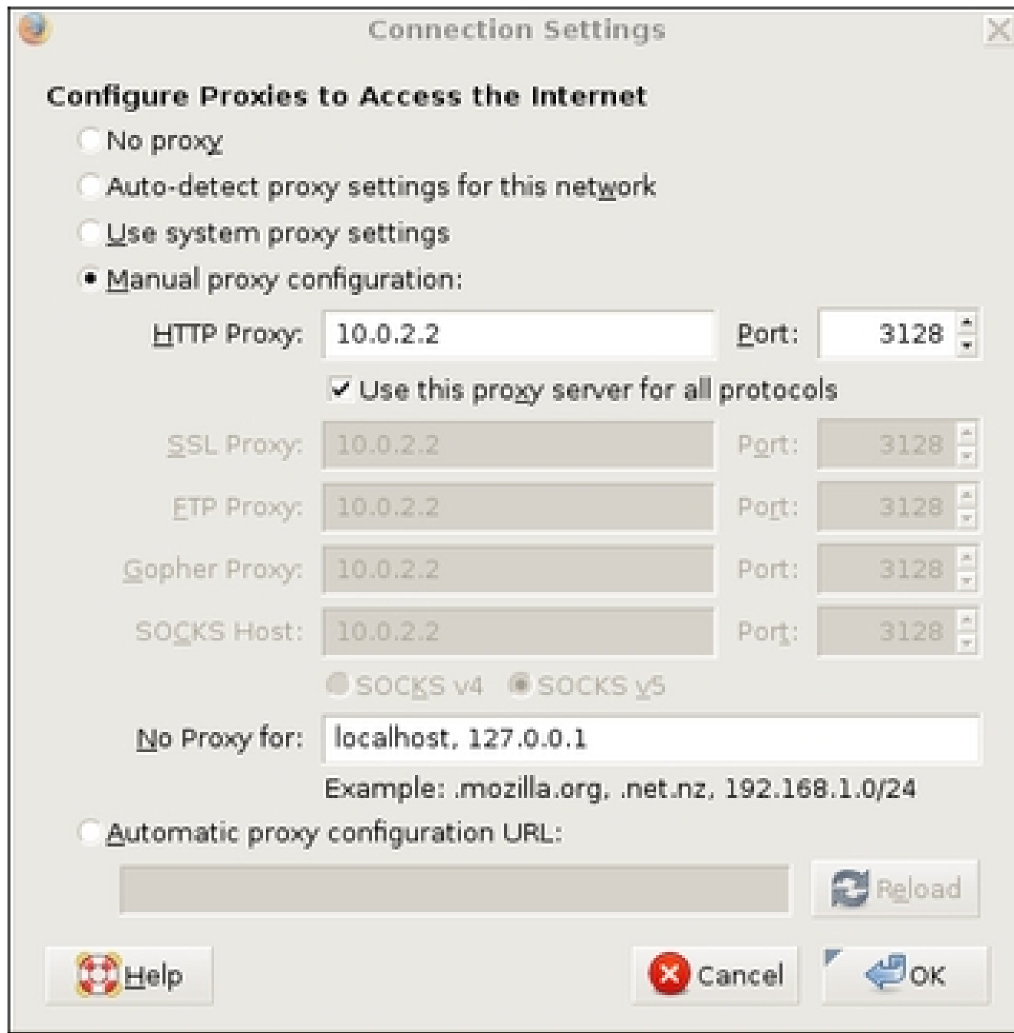

Figure 1
Setting Up Firefox to Use Proxies

In Figure 1, we've selected Manual proxy configuration and entered in an HTTP Proxy address (which can be either a hostname or IP address) of 10.0.2.2 and Port number 3128, which is Squid's default listening port for client connections. We've also selected the box to Use this proxy server for all protocols, resulting in the same values being copied automatically to the subsequent settings for other types of proxies.

We've left No Proxy for: at its default value of localhost, 127.0.0.1. The reason for not proxying connections to Web pages hosted locally on the client system is probably obvious, but you can additionally list URLs or IP addresses elsewhere on your local network that there's no need to use the proxy to reach.

At this point, you may be wondering, what does the connection between a client and a Web proxy look like? Is there some special protocol, or maybe a subset of HTTP commands or flags?

In fact, proxy connections are simpler than you may think. Normally, when you click on a hyperlink or enter a URL, your browser resolves the URL you typed or clicked on, using its own local DNS capabilities. It then takes the IP address and sends an HTTP/HTTPS request to that IP address, with the original (non-resolved) URL in the body of the request.

A proxied connection is the same without any DNS resolution. Your browser simply sends its HTTP/HTTPS request to the proxy server without trying to resolve the URL. The body of that request is identical to the one it would otherwise send directly to the Web server you're trying to reach.

Instead of configuring your Web browser's proxy settings directly, if you use the GNOME desktop on your client test system, you can set global proxy settings that can, in turn, be used by Firefox and other Internet applications. Note, however, that the proxy settings you set in GNOME will be applied only to applications that are, in turn, configured to use system settings—for example, by selecting the option Use system proxy settings shown in Figure 1. Other applications will continue to use either their own proxy settings or no proxy at all.

GNOME's Network Proxy Preferences applet, which should appear in your System→Preferences menu, is shown in Figure 2.

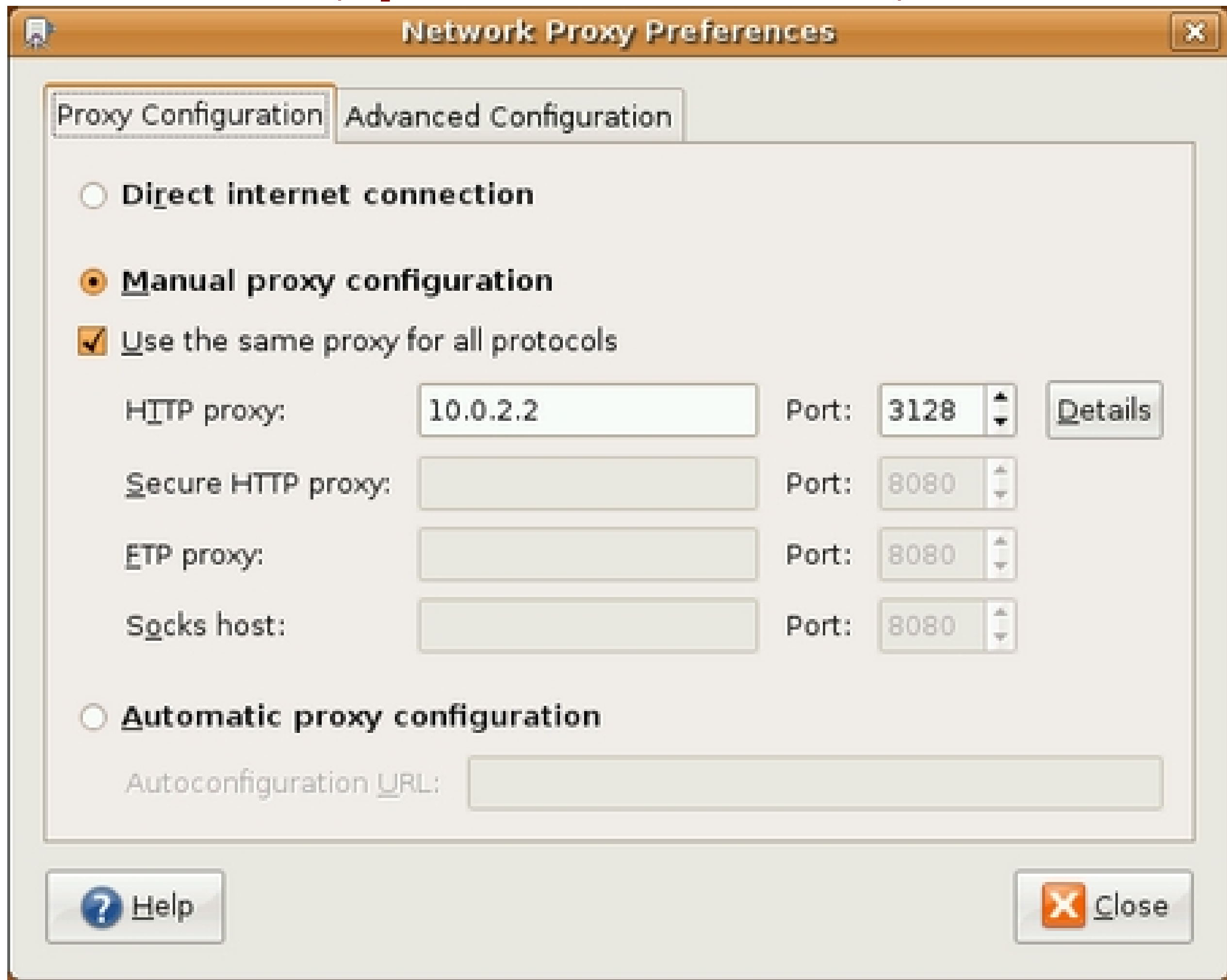

Figure 2
Setting Global Proxy Options in GNOME

It may seem like I'm spending a lot of ink explaining client-side configuration just for testing purposes, given that this is an article about building Squid servers. But, of course, the way you set up a proxy client for testing is the same as for one in production, so I would have had to explain this sooner or later anyhow.

In fact, future installments in this series may go further in covering client configuration topics. Autoproxy.pac files, for example (which is what Figure 1's Automatic proxy configuration URL setting is for), can be very handy in managing very complex or very highly scaled proxy environments.

Once you've configured your test client system to use your Squid proxy, you can attempt to navigate to some Web page to see if everything works. It's a good idea to tail Squid's access log simultaneously. To do so, enter this command on your Squid system:

```
bash-$ sudo tail -f /var/log/squid/access.log
```

If browsing works but nothing zings by in this log-tailing session, your client-side configuration is incorrect—it isn't actually using the proxy. If browsing doesn't work, you may see some useful server-side message in the log-tailing session. Squid usually returns fairly useful messages directly to client browsers as well.

If things don't work, your browser session is simply timing out and nothing is showing up in access.log, try using the ping command from your client to your proxy and vice versa. If pinging doesn't work, the problem is at the network level and has nothing to do with Squid.

Squid's Performance Benefits

The Paranoid Penguin is a security column, so naturally, security is our primary focus in dealing with Squid (or it will be, once I've walked you through the basics of getting it up and running). But, you should be aware that Squid is not a security application per se. Squid's main purpose in life is to cache commonly accessed Web and FTP content locally, thereby both reducing Internet bandwidth usage and speeding up end users' download times.

The negative side of this is that Squid doesn't have as rich of a security feature set built in to it as commercial security-oriented Web proxies, such as BlueCoat and Sidewinder. In fact, Squid (years ago) used to ship with a default configuration that allowed completely open access.

The good side is that Squid can be configured, especially along with add-ons like SquidGuard, to provide some of the most important Web proxy security features. And, even if those features are your main reason for deploying Squid, you'll still enjoy the performance benefits of having commonly accessed Web content cached locally by Squid.

Seldom, in the security business, do we enhance end users' experience when we add security controls.

Conclusion

With any luck, at this point, chances are that everything works! Your Squid proxy software is installed, configured to accept only client connections from itself and from hosts on your local network, and it's hard at work proxying your users' connections and caching commonly accessed content. Not a bad day's work!

Not difficult, was it? Like most server applications, Squid's default configuration file is designed to maximize your chances for success, while minimizing the odds of your shiny-new Squid server being hacked. But, also like other server applications, there's certainly more that you can and should do to secure your Squid proxy than the default settings will do for you.

That will be our starting point next month. Among other things, we'll delve much deeper into Squid's Access Control List features to further harden Squid. Until then, be safe!

Building a Secure Squid Web Proxy

By Mick Bauer

(Reprinted from the Linux Journal)

PART III

We've been building a secure Squid Web Proxy the past few months, and we'll continue to do so for a couple more. Last time [May 2009], we got Squid installed, running and restricted to serve only local clients (based on their IP addresses). This month, we delve deeper into Squid's Access Control List (ACL) capabilities and other built-in security features.

ACL Review

As you may recall from my last column, all we had to do to get Squid running on a standard Ubuntu 8.04 system was add two lines to the file /etc/squid/squid.conf:

```
acl mick_network src 10.0.2.0/24
http_access allow mick_network
```

We inserted those two lines, which allow outbound proxy connections from clients whose IP addresses fall within the network 10.0.2.0/24 (that is, addresses 10.0.2.1 through 10.0.2.254), right above Squid's default “deny all” ACL, which looks like this:

```
http_access deny all
```

You can correctly infer from this that, by default, Squid denies proxy connections from all clients. This is a refreshing change in default server application configurations during the past few years. Whereas in the past, many applications had default configurations that would “just work”, which is a very user-friendly but also excessively open stance, nowadays few network applications will do much of anything without some administrative intervention. This is only sensible. Connecting things to the Internet that you don't even know how to configure is the way of pain.

Getting back to our example ACL, the acl statement itself is fairly self-explanatory: acl tells Squid we're defining an ACL; mick_network is its name; src indicates it matches the client's source IP address or network address; and 10.0.2.0/24 is the network address in CIDR notation that will match this ACL.

This is the simplest type of ACL and still one of the most useful. In February 2002, if the New York Times had had a simple source-IP/network ACL correctly configured on its Internet-facing corporate Web proxies, the rogue hacker Adrian Lamos couldn't have gained access quite so easily to its editorial-page contributor database or its Lexus-Nexus portal.

ACLs in More Depth

Besides clients' (source) IP addresses, Squid also can match a great deal of other proxy transaction characteristics. Note that some of these deal with arcane HTTP headers and parameters, many of which are minimally useful for most Squid users anyhow.

Table 1
Complete List of ACL Types Supported in Squid 2.6

ACL Type	Description
src	Client (transaction source) IP address or network address.
dst	Server (transaction destination) IP address or network address.
myip	Local IP address on which Squid is listening for connections.
arp	Client's Ethernet (MAC) address (matches local LAN clients only).
srcdomain	Client's domain name as determined by reverse DNS lookup.
dstdomain	Domain portion of URL requested by client.
srcdom_regex	Regular expression matching client's domain name.
dstdom_regex	Regular expression matching domain in requested URL.
time	Period of time in which transaction falls.
url_regex	Regular expression matching entire requested URL (not just domain).
urlpath_regex	Regular expression matching path portion of requested URL.
urllogin	Regular expression matching requested URL's “login” field.
port	Requested site's (destination) TCP port.
myport	Local TCP port on which Squid is listening for connections.
proto	Application-layer protocol of request (HTTP, HTTPS, FTP, WHOIS or GOPHER).
method	Request's HTTP method (GET, POST or CONNECT).
browser	Matches the client's browser, per HTTP “User-Agent” header.
referer_regex	Regular expression matching the unreliable HTTP “Referer” header (that is, the supposed URL of some page on which the user clicked a link to the requested site).
ident	Matches specified user name(s) of user(s) running client browser, per an “ident” lookup. Note that ident replies, which often can be spoofed, should not be used in lieu of proper authentication.
ident_regex	Regular expression defining which client user names to match per ident lookup.
src_as	Matches client IP addresses associated with the specified Autonomous System (AS) number, usually an ISP or other large IP registrant.
dst_as	Matches destination-server IP addresses associated with the specified AS number.
proxy_auth	Matches the specified user name, list of user names or the wild card REQUIRED (which signifies any valid user name).
proxy_auth_regex	Regular expression defining which user names to match.
snmp_community	For SNMP-enabled Squid proxies, matches client-provided SNMP community string.
maxconn	Matches when client's IP address has established more than the specified number of HTTP connections.
max_user_ip	Matches the number of IP addresses from which a single user attempts to log in.
req_mime_type	Matches a regular expression describing the MIME type of the client's request (not the server's response).
req_header	Matches a regular expression applied to all known request headers (browser, referer and mime-type) in the client's request.
rep_mime_type	Matches a regular expression describing the MIME type of the server's response.

ACL Type	Description
rep_header	Matches a regular expression applied to all known request headers (browser, referer and mime-type) in the server's response.
external	Performs an external ACL lookup by querying the specified helper class defined in the external_acl_type tag.
urlgroup	Matches a urlgroup name, as defined in redirector setups.
user_cert	Matches specified attribute (DN, C, O, CN, L or ST) and values against client's SSL certificate.
ca_cert	Matches specified attribute (DN, C, O, CN, L or ST) and values against client certificate's issuing Certificate Authority certificate.
ext_user	Matches specified user name(s) against that returned by an external ACL/authentication helper (configured elsewhere in squid.conf).
ext_user_regex	Matches a regular expression describing user names to be matched against that returned by an external ACL/authentication helper.

I've presented the full range of possible ACL types to give you a taste for how rich Squid's ACL functionality is. Needless to say, however, I can't cover usage scenarios for (or even adequately explain) all of these. ViServe's “Squid 2.6 Configuration Manual” (see Resources) gives complete syntax and usage examples for all.

Many, if not most, Squid installations don't go much beyond a few src ACLs, along with perhaps a few simple dstdomain blacklist entries thrown in for good measure. Many of the other most useful ACL types, such as myip, time, port, proto, method, dst_mime_type and rep_mime_type, should be reasonably self-explanatory (or at least easy enough to understand from the examples shown in squid.conf's comments).

One category of less-intuitive ACL types is particularly powerful and useful: the ones that enable Squid to authenticate client users via external authentication authorities. Before we tackle authentication, however, we should give a little more attention to ACL operators, the tags that perform some action (most commonly, to allow or deny a request) based on a matched ACL.

By far, the most important ACL operator is http_access, which specifies whether Squid should allow the transaction matching the specified ACL to proceed. Going back to the example ACL/operator pair from the beginning of this section, after we defined the ACL mick_network as all transactions involving client/source IP addresses within 10.0.2.0/24, we operated on it with this line:

```
http_access allow mick_network
```

This is simple enough to understand: “allow HTTP requests matching the ACL named mick_network.” The most common use of ACLs is to specify a list of ACLs and http_access statements, ending (as we've seen) with a “drop by default” line, like this:

```
http_access deny all
```

This has the effect of creating a “whitelist”— a list of types of transactions that are allowed, with all others being denied.

Squid recognizes a number of additional ACL operators besides http_allow, including no_cache, ident_lookup_access, always_direct, never_direct and snmp_access. Because most of these concern cache performance, HTTP redirects and communications with other Squid servers rather than security per se, I'll leave it to you to explore those (or not) as your particular needs dictate. The Squid User's Guide referenced in the Resources section is a good source of information about Squid's various ACL operators.

Squid Authentication

As I mentioned previously, one of Squid's most handy capabilities is its ability to authenticate proxy users by means of a variety of external helper mechanisms. One of the simplest and probably most commonly used helper applications is ncsa_auth, a simple user name/password scheme that uses a flat file consisting of rows of user name/password hash pairs. The HOWTO by Vivek Gite and, to a lesser extent, the Squid User's Guide, explain how to set this up (see Resources).

Briefly, you'll add something like this to /etc/squid/squid.conf:

```
auth_param basic program /usr/lib/squid/ncsa_auth /etc/squid/squidpasswd
auth_param basic children 5
auth_param basic realm Squid proxy-caching web server at Wiremonkeys.org
auth_param basic credentialsttl 2 hours
auth_param basic casesensitive off
```

And, in the ACL section:

```
acl ncsa_auth_users proxy_auth REQUIRED
http_access allow ncsa_auth_users
```

The block of auth_param tags specifies settings for a “basic” authentication mechanism:

- program is the helper executable ncsa_auth, using the file /etc/squid/squidpassd as the user name/password hash list (created previously).
- children, the number of concurrent authentication processes, is five.
- realm, part of the string that greets users, is “Squid proxy-caching Web server at Wiremonkeys.org”.
- credentialsttl, the time after authentication that a successfully authenticated client may go before being re-authenticated, is two hours.
- casesensitive, which determines whether user names are case-sensitive, is off.

In the ACL section, we defined an ACL called ncsa_auth_users that says the proxy_auth mechanism (as defined in the auth_param section) should be used to authenticate specified users. Actually in this case, instead of a list of user names to authenticate, we've got the wild card REQUIRED, which

expands to “all valid users”. The net effect of this ACL and its subsequent http_access statement is that only successfully authenticated users may use the proxy.

The main advantages of the NCSA mechanism are its simplicity and its reasonable amount of security (only password hashes are transmitted, not passwords proper). Its disadvantage is scalability, because it requires you to maintain a dedicated user name/password list. Besides the administrative overhead in this, it adds yet another user name/password pair your users are expected to remember and protect, which is always an exercise with diminishing returns (the greater the number of credentials users have, the less likely they'll avoid risky behaviors like writing them down, choosing easy-to-guess passwords and so forth).

Therefore, you're much better off using existing user credentials on an external LDAP server (via the ldap_auth helper) on an NT Domain or Active Directory server (via the msnt_auth helper) or the local Pluggable Authentication Modules (PAM) facility (via the pam_auth helper). See Resources for tutorials on how to set up Squid with these three helpers.

Note that Squid's helper programs are located conventionally under /usr/lib/squid. Checking this directory is a quick way to see which helpers are installed on your system, although some Linux distributions may use a different location.

Other Squid Defenses

Access Control Lists really are Squid's first line of defense—that is, Squid's primary mechanism for protecting your network, your users and the Squid server itself. There are a couple other things worth mentioning, however.

First, there's the matter of system privileges. Squid must run as root, at least while starting up, so that, among other things, it can bind to privileged TCP ports such as 80 or 443 (although by default it uses the nonprivileged port 3128). Like other mainstream server applications, however, Squid's child processes—the ones with which the outside world actually interacts—are run with lower privileges. This helps minimize the damage a compromised or hijacked Squid process can do.

By default, Squid uses the user proxy and group proxy for nonprivileged operations. If you want to change these values for effective UID and GID, they're controlled by squid.conf's cache_effective_user and cache_effective_group tags, respectively.

Squid usually keeps its parent process running as root, in case it needs to perform some privileged action after startup. Also, by default, Squid does not run in a chroot jail. To make Squid run chrooted, which also will cause it to kill the privileged parent process after startup (that is, also will cause it to run completely unprivileged after startup), you can set squid.conf's chroot tag to the path of a previously created Squid chroot jail.

If you're new to this concept, chrooting something (changing its root) confines it to a subset of your filesystem, with the effect that if the service is somehow hacked (for example, via some sort of buffer overflow), the attacker's processes and activities will be confined to an unprivileged “padded cell” environment. It's a useful hedge against losing the patch rat race.

Chrooting and running with nonroot privileges go hand in hand. If a process runs as root, it can trivially break out of the chroot jail. Conversely, if a nonprivileged process nonetheless has access to other (even nonprivileged) parts of your filesystem, it still may be abused in unintended and unwanted ways.

Somewhat to my surprise, there doesn't seem to be any how-to for creating a Squid chroot jail on the Internet. The world could really use one—maybe I'll tackle this myself at some point. In the meantime, see Resources for some mailing-list posts that may help. Suffice it to say for now that as with any other chroot jail, Squid's must contain not only its own working directories, but also copies of system files like /etc/nsswitch.conf and shared libraries it uses.

Common Squid practice is to forego the chroot experience and to settle for running Squid partially unprivileged per its default settings. If, however, you want to run a truly hardened Squid server, it's probably worth the effort to figure out how to build and use a Squid chroot jail.

Conclusion

Setting ACLs, running Squid with nonroot privileges most or all of the time and running Squid in a chroot jail constitute the bulk of Squid's built-in security features. But, these are not the only things you can do to use Squid to enhance your network and end-user systems' security.

Next time, I'll show you how to use add-on tools such as SquidGuard to increase Squid's intelligence in how it evaluates clients' requests and servers' replies. I'll also address (if not next time then in a subsequent column) some of the finer points of proxying TLS/SSL-encrypted sessions. Until then, be safe!

PART IV

In my previous three columns [April, May and July 2009], I described the concept, benefits and architectural considerations of outbound Web proxies (Part I); discussed basic Squid installation, configuration and operation (Part II); and explained Squid Access Control Lists (ACLs), its ability to run as an unprivileged user and provided some pointers on running Squid in a chroot jail (Part III).

Although by no means exhaustively detailed, those articles nonetheless cover the bulk of Squid's built-in security functionality (ACLs, running nonroot and possibly running chrooted). This month, I conclude this series by covering an important Squid add-on: squidGuard.

squidGuard lets you selectively enforce “blacklists” of Internet domains and URLs you don't want end users to be able to reach. Typically, people use squidGuard with third-party blacklists from various free and commercial sites, so that's the usage scenario I describe in this article.

Introduction to squidGuard

Put simply, squidGuard is a domain and URL filter. It filters domains and URLs mostly by comparing them against lists (flat files), but also, optionally, by comparing them against regular expressions.

squidGuard does not filter the actual contents of Web sites. This is the domain of appliance-based commercial Web proxies such as Blue Coat, and even products like that tend to emphasize

URL/domain filtering over actual content parsing due to the high-computing (performance) cost involved.

You may wonder, what have URL and domain filtering got to do with security? Isn't that actually a form of censorship and bandwidth-use policing? On the one hand, yes, to some extent, it is.

Early in my former career as a firewall engineer and administrator, I rankled at management's expectation that I maintain lists of the most popular URLs and domains visited. I didn't think it was my business what people used their computers for, but rather it should be the job of their immediate supervisors to know what their own employees were doing.

But the fact is, organizations have the right to manage their bandwidth and other computing resources as they see fit (provided they're honest with their members/employees about privacy expectations), and security professionals are frequently in the best “position” to know what's going on. Firewalls and Web proxies typically comprise the most convenient “choke points” for monitoring or filtering Web traffic.

Furthermore, the bigger domain/URL blacklists frequently include categories for malware, phishing and other Web site categories that do, in fact, have direct security ramifications. For example, the free Shalla's Blacklists include more than 27,600 known sources of spyware!

Even if you don't care about conserving bandwidth or enforcing acceptable-use policies, there's still value in configuring squidGuard to block access to “known dangerous” Web sites. That's precisely what I'm going to show you how to do.

Getting and Installing squidGuard

If you run a recent version of Fedora, SUSE, Debian or Ubuntu Linux, squidGuard is available as a binary package from your OS's usual software mirrors (in the case of Ubuntu, it's in the universe repositories). If you run RHEL or CentOS, however, you need to install either Dag Wieers' RPM of squidGuard version 1.2, Excalibur Partners' RPM of squidGuard version 1.4, or you'll have to compile squidGuard from the latest source code, available at the squidGuard home page (see Resources for the appropriate links).

Speaking of squidGuard versions, the latest stable version of squidGuard at the time of this writing is squidGuard 1.4. But, if your Linux distribution of choice provides only squidGuard 1.2, as is the case with Fedora 10 and Ubuntu 9.04, or as with OpenSUSE 11.1, which has squidGuard 1.3, don't worry. Your distribution almost certainly has back-ported any applicable squidGuard 1.4 security patches, and from a functionality standpoint, the most compelling feature in 1.4 absent in earlier versions is support for MySQL authentication.

Hoping you'll forgive my Ubuntu bias of late, the command to install squidGuard on Ubuntu systems is simply:

```
bash-$ sudo apt-get squidguard
```

As noted above, squidGuard is in the universe repository, so you'll either need to uncomment the universe lines in /etc/apt/sources.list, or open Ubuntu's Software Sources applet, and assuming it isn't already checked, check the box next to Community-maintained Open Source software (universe), which will uncomment those lines for you.

Besides using apt-get from a command prompt to install squidGuard, you could instead use the Synaptic package manager. Either of these three approaches automatically results in your system's downloading and installing a deb archive of squidGuard.

If you need a more-current version of squidGuard than what your distribution provides and are willing to take it upon yourself to keep it patched for emerging security bugs, the squidGuard home page has complete instructions.

Getting and Installing Blacklists

Once you've obtained and installed squidGuard, you need a set of blacklists. There's a decent list of links to these at squidguard.org/blacklists.html, and of these, I think you could do far worse than Shalla's Blacklists (see Resources), a free-for-noncommercial-use set that includes more than 1.6 million entries organized into 65 categories. It's also free for commercial use; you just have to register and promise to provide feedback and list updates. Shalla's Blacklists are the set I use for the configuration examples through the rest of this article.

Once you've got a blacklist archive, unpack it. It doesn't necessarily matter where, so long as the entire directory hierarchy is owned by the same user and group under which Squid runs (proxy:proxy on Ubuntu systems). A common default location for blacklists is /var/lib/squidguard/db.

To extract Shalla's Blacklists to that directory, I move the archive file there:

```
bash-$ cp mv shallalist.tar.gz /var/lib/squidguard/db
```

Then, I unpack it like this:

```
bash-$ sudo -s
bash-# cd /var/lib/squidguard/db
bash-# tar --strip 1 -xvzf shallalist.tar.gz
bash-# rm shallalist.tar.tz
```

In the above tar command, the --strip 2 option strips the leading BL/ from the paths of everything extracted from the shallalist tar archive. Without this option, there would be an additional directory (BL/) under /var/lib/squidguard/db containing the blacklist categories, for example, /var/lib/squidguard/db/BL/realestate rather than /var/lib/squidguard/db/realestate. Note that you definitely want to delete the shallalist.tar.gz file as shown above; otherwise, squidGuard will include it in the database into which the contents of /var/lib/squidguard/db will later be imported.

Note also that at this point you're still in a root shell; you need to stay there for just a few more commands. To set appropriate ownership and permissions for your blacklists, use these commands:

```
bash-# chown -R proxy:proxy /var/lib/squidguard/db/
bash-# find /var/lib/squidguard/db -type f | xargs chmod 644
bash-# find /var/lib/squidguard/db -type d | xargs chmod 755
bash-# exit
```

And with that, your blacklists are ready for squidGuard to start blocking. After, that is, you configure squidGuard to do so.

Configuring squidGuard

On Ubuntu and OpenSUSE systems (and probably others), squidGuard's configuration file squidGuard.conf is kept in /etc/squid/, and squidGuard automatically looks there when it starts. As root, use the text editor of your choice to open /etc/squid/squidGuard.conf. If using a command-line editor like vi on Ubuntu systems, don't forget to use sudo, as with practically everything else under /etc/, you need to have superuser privileges to change squidGuard.conf.

squidGuard.conf's basic structure is:

- Options (mostly paths)
- Time Rules
- Rewrite Rules
- Source Addresses
- Destination Classes
- Access Control Lists

In this article, my goal is quite modest: to help you get started with a simple blacklist that applies to all users, regardless of time of day, and without any fancier URL-rewriting than redirecting all blocked transactions to the same page. Accordingly, let's focus on examples of Options, Destination Classes and ACLs. Before you change squidGuard.conf, it's a good idea to make a backup copy, like this:

```
bash-$ sudo cp /etc/squid/squidGuard.conf
/etc/squid/squidGuard.conf.def
```

Now you can edit squidGuard.conf. First, at the top, leave what are probably the default values of dbhome and logdir, which specify the paths of squidGuard's databases of blacklists (or whitelists, you also can write ACLs that explicitly allow access to certain sites and domains) and its log files, respectively. These are the defaults in Ubuntu:

```
dbhome /var/lib/squidguard/db
logdir /var/log/squid
```

These paths are easy enough to understand, especially considering that you just extracted Shalla's Blacklists to /var/lib/squidguard/db. Whatever you do, do not leave a completely blank line at the very top of the file; doing so prevents squidGuard from starting properly.

Next, you need to create a Destination Class. This being a security column, let's focus on blocking access to sites in the spyware and remotecontrol categories. You certainly don't want your users'

systems to become infected with spyware, and you probably don't want users to grant outsiders remote control of their systems either.

Destination Classes that describe these two categories from Shalla's Blacklists look like this:

```
dest remotecontrol {
        domainlist      remotecontrol/domains
        urllist         remotecontrol/urls
}

dest spyware {
        domainlist      spyware/domains
        urllist         spyware/urls
}
```

As you can see, the paths in each domainlist and urllist statement are relative to the top-level database path you specified with the dbhome option. Note also the curly bracket {} placement: left brackets always immediately follow the destination name, on the same line, and right brackets always occupy their own line at the end of the class definition.

Finally, you need an ACL that references these destinations—specifically, one that blocks access to them. The ACL syntax in squidGuard is actually quite flexible, and it's easy to write both “allow all except...” and “block all except...” ACLs. Like most ACL languages, they're parsed left to right, top to bottom. Once a given transaction matches any element in an ACL, it's either blocked or passed as specified, and not matched against subsequent elements or ACLs.

You can have multiple ACL definitions in your squidGuard.conf file, but in this example scenario, it will suffice to edit the default ACL. A simple default ACL that passes all traffic unless destined for sites in the remotecontrol or spyware blacklists would look like this:

```
acl {
        default {
                pass !remotecontrol !spyware all
                redirect http://www.google.com
        }
}
```

In this example, default is the name of the ACL. Your default squidGuard.conf file probably already has an ACL definition named default, so be sure either to edit that one or delete it before entering the above definition; you can't have two different ACLs both named default.

The pass statement says that things matching remotecontrol (as defined in the prior Destination Class of that name) do not get passed, nor does spyware, but all (a wild card that matches anything that makes it that far in the pass statement) does. In other words, if a given destination matches anything in the remotecontrol or spyware blacklists (either by domain or URL), it won't be passed, but rather will be redirected per the subsequent redirect statement, which points to the Google home page.

Just to make sure you understand how this works, let me point out that if the wild card all occurred before !remotecontrol, as in “pass all !remotecontrol !spyware”, squidGuard would not block anything, because matched transactions aren't compared against any elements that follow the element they matched. When constructing ACLs, remember that order matters!

I freely admit I'm being very lazy in specifying that as my redirect page. More professional system administrators would want to put a customized “You've been redirected here because...” message onto a Web server under their control and list that URL instead. Alternatively, squidGuard comes with a handy CGI script that displays pertinent transaction data back to the user. On Ubuntu systems, the script's full path is /usr/share/doc/squidguard/examples/squidGuard.cgi.gz.

This brings me to my only complaint about squidGuard: if you want to display a custom message to redirected clients, you either need to run Apache on your Squid server and specify an http://localhost/ URL, or specify a URL pointing to some other Web server you control. This is in contrast to Squid itself, which has no problem displaying its own custom messages to clients without requiring a dedicated HTTP dæmon (either local or external).

To review, your complete sample squidGuard.conf file (not counting any commented-out lines from the default file) should look like this:

```
dbhome /var/lib/squidguard/db
logdir /var/log/squid

dest remotecontrol {
        domainlist      remotecontrol/domains
        urllist         remotecontrol/urls
}

dest spyware {
        domainlist      spyware/domains
        urllist         spyware/urls
}

acl {
        default {
                pass !remotecontrol !spyware all
                redirect http://www.google.com
        }
}
```

Now that squidGuard is configured and, among other things, knows where to look for its databases, you need to create actual database files for the files and directories under /var/lib/squidGuard/db, using this command:

```
bash-$ sudo -u proxy squidGuard -C all
```

This imports all files specified in active Destination Class definitions in squidGuard.conf, specifically in this example, the files remotecontrol/domains, remotecontrol/urls, spyware/domains and spyware/urls, into Berkeley DB-format databases. Obviously, squidGuard can access the blacklists much faster using database files than by parsing flat text files.

Configuring Squid to Use squidGuard

The last thing you need to do is reconfigure Squid to use squidGuard as a redirector and tell it how many redirector processes to keep running. The location of your squidGuard binary is highly distribution-specific; to be sure, you can find it like this:

```
bash-$ which squidGuard
/usr/bin/squidGuard
```

As for the number of redirector processes, you want a good balance of system resource usage and squidGuard performance. Starting a lot of redirectors consumes resources but maximizes squidGuard performance, whereas starting only a couple conserves resources by sacrificing squidGuard performance. Ubuntu's default of 5 is a reasonable middle ground.

The squid.conf parameters for both of these settings (redirector location and number of processes) are different depending on with which version of Squid you're using squidGuard. For Squid versions 2.5 and earlier, they're redirect_program and redirect_children. For Squid versions 2.6 and later, they're url_rewrite_program and url_rewrite_program.

For example, on my Ubuntu 9.04 system, which runs Squid version 2.7, I used a text editor (run via sudo) to add the following two lines to /etc/squid/squid.conf:

```
url_rewrite_program /usr/bin/squidGuard
url_rewrite_children 5
```

As with any other time you edit /etc/squid/squid.conf, it's probably a good idea to add custom configuration lines before or after their corresponding comment blocks. squid.conf, you may recall, is essentially self-documented—it contains many lines of example settings and descriptions of them, all in the form of comments (lines beginning with #). Keeping your customizations near their corresponding examples/defaults/comments both minimizes the chance you'll define the same parameter in two different places, and, of course, it gives you easy access to information about the things you're changing.

By the way, I'm assuming Squid itself already is installed, configured and working the way you want it to (beyond blacklisting). If you haven't gotten that far before installing squidGuard, please refer to my previous three columns (see Resources).

Before those changes take effect, you need to restart Squid. On most Linux systems, you can use this command (omitting the sudo if you're already in a root shell):

```
bash-$ /etc/init.d/squid reload
```

If you get no error messages, and if when you do a ps -axuw |grep squid you see not only a couple Squid processes, but also five squidGuard processes, then congratulations! You've now got a working installation of squidGuard.

But is it actually doing what you want it to do? Given the filters we just put in place, the quickest way to tell is, on some client configured to use your Squid proxy, to point a browser to http://www.gotomypc.com (a site in the remotecontrol blacklist). If everything's working correctly, your browser will not pull up gotomypc, but rather Google. squidGuard is passive-aggressively encouraging you to surf to a safer site!

Conclusion

squidGuard isn't the only Squid add-on of interest to the security-conscious. squidtaild and squidview, for example, are two different programs for monitoring and creating reports from Squid logs (both of them are available in Ubuntu's universe repository). I leave it to you though to take your Squid server to the next level.

This concludes my introductory series on building a secure Web proxy with Squid. I hope you're off to a good, safe start!

Resources

squidGuard home page, featuring squidGuard's latest source code and definitive documentation: squidguard.org.

OpenSUSE's squidGuard page: en.opensuse.org/SquidGuard.

squidGuard 1.2 RPMs for Fedora, CentOS and RHEL from Dag Wieers: dag.wieers.com/rpm/packages/squidguard.

squidGuard 1.4 RPM for CentOS 5, from Excalibur Partners LLC: www.excaliburtech.net/archives/46.

The Debian Wiki's “Rudimentary squidGuard Filtering” page: wiki.debian.org/DebianEdu/HowTo/SquidGuard.

Wessels, Duane: Squid: The Definitive Guide. Sebastopol, CA: O'Reilly Media, 2004 (includes some tips on creating and using a Squid chroot jail).

The Squid home page, where you can obtain the latest source code and binaries for Squid: www.squid-cache.org.

The Ubuntu Server Guide's Squid chapter: https://help.ubuntu.com/8.10/serverguide/C/squid.html.

The Squid User's Guide: www.deckle.co.za/squid-users-guide/Main_Page.

Shalla's Blacklists are available at www.shallalist.de (the most current blacklist archive is always at www.shallalist.de/Downloads/shallalist.tar.gz).