GIAC Certification

GSEC Practical Assignment

Version 1.4b Option 2

SSL Web Proxy

A Secure and Inexpensive Remote Access Implementation

Prepared and Developed by David E. Culp

April 2, 2003

Abstract:

With our network perimeter expanding, the type of clients that need remote access changing, and IT budgets decreasing in today's economy; The demand for a secure, inexpensive, and multi-platform solution is gaining momentum. For years VPNs (Virtual Private Networks) using the IPSEC protocol have proven to be a stable and secure remote access solution. However, in addition to the initial cost of the implementation, the time and cost of maintaining the clients are becoming more apparent. Furthermore, the need to support clients other than 'Window PCs' has increased.

Within this paper, I will outline a system that I developed using the proven technology of SSL (Secure Socket Layer), and the combination of stable, secure Open Source software with some custom programming. The objective of this system is to allow external clients without any configuration changes to securely access our internal web applications via the Internet. For my implementation, I was given a set of criteria which are outlined in the following section. With the criteria declared, my objectives set and a vast sea of information via the Internet and the Open Source community before me, I was eager and confident that this journey would be successful. I have included all the paths that my research took me, because each path resulted in a great system. However for one reason or another, the system did not meet all of the criteria. But after many hours of research and development, a system was created that met the criteria and satisfied our needs.

SECTION: (Before) Setting the Stage:

The State of the Industry and the Growing Needs:

For secure remote access, companies have been using VPNs (Virtual Private Networks) for years. The technology is stable and secure. In practice, more and more network administrators are finding there are some drawbacks to VPN implementations. They include:

* The extra administrative workload involved in maintaining the remote clients. This includes changes to the remote VPN client version, issues with the remote client OS version, and issues with home and office firewalls.

* A limited range of client types are supported. Most of the VPN Vendor clients are written for the Windows platform. With the growing emergence of PDAs and Linux Desktops, this is becoming more of an issue. For Linux desktops, a solution can be found with FreeS /WAN (http://www.freeswan.org/).

In that past year, a new technology that offers secure remote access without the shortcomings listed above has been gaining industry attention and market share. This new technology is known as “SSL VPNs”. In turn the traditional VPNs are now referred to as “IPSEC VPNs”. Some of the phrases you may read from a commercial product’s marketing literature are ‘client–less” and “access from any Internet-Enabled device”. Aventail (www.aventail.com) is the current leader in this market with it’s *ASAP Platform*. Nortel (www.nortelnetworks.com) announced their *Alteon SSL* product line back in September 2002. While these products offer a rich set of features, they do not come cheap. On average, the base price usually starts around $25,000. So after researching the commercial products, I decided to develop a system using OSS (Open Source Software) and SSL on a Linux OS platform.

Current Remote Access Structure:

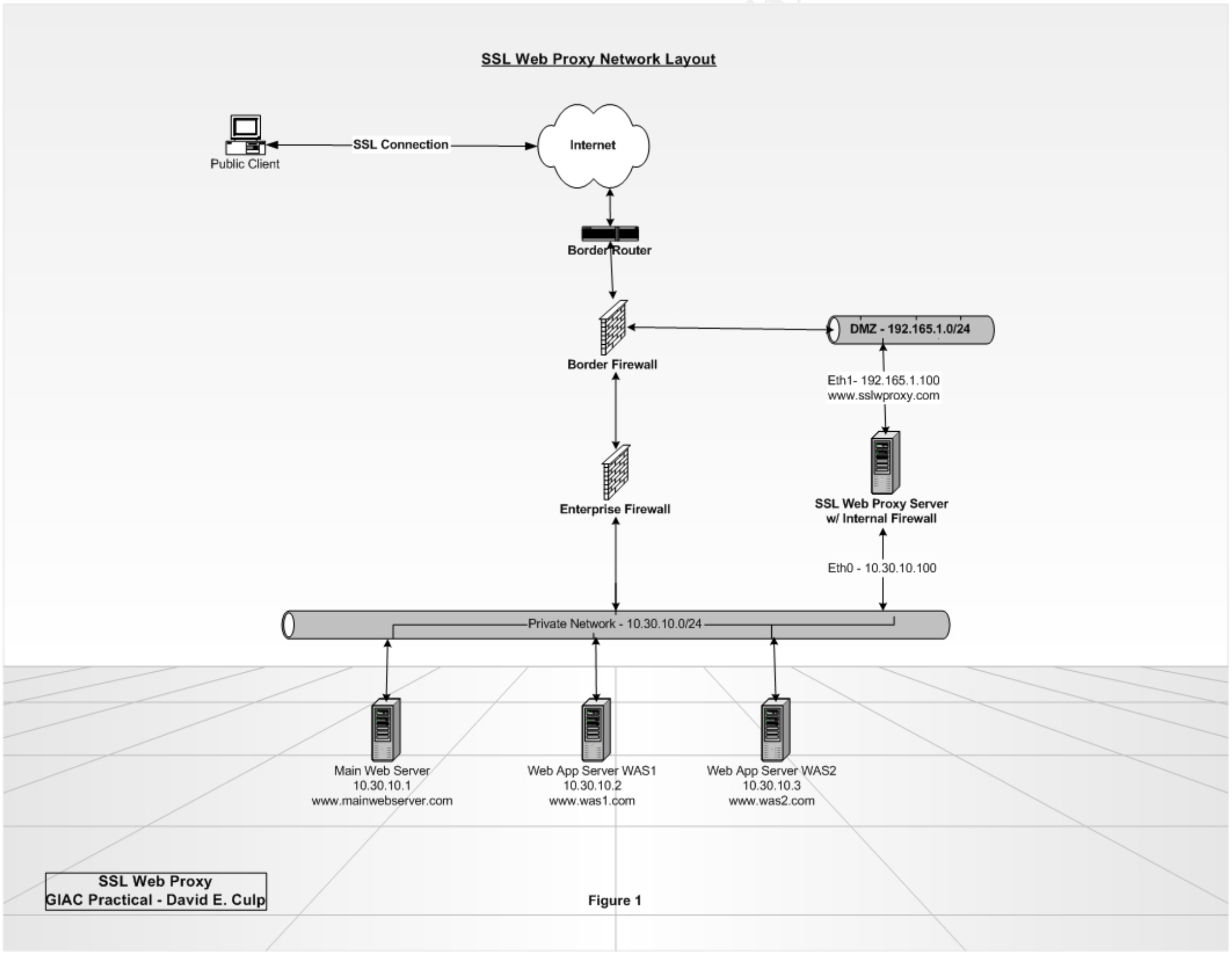
Our current Remote Access solution involves a combination of two Nortel Contivity VPN/Firewall switches for our broadband users and branch office tunnels. The number of clients and their types are increasing at a rapid pace. In addition we are beginning to experience the additional administrative workload of maintaining the remote clients: VPN client versions, and remote firewall issues. A “SSL Remote Access” solution can supplement our current infrastructure and help solve these problems as well as place us in our better position for future growth. The management of remote clients can be greatly reduced.

Criteria For Remote Access Solution:

a) Secure
b) Inexpensive
c) Allow any client (PC Windows, PDA, Linux) to be able to access our internal web applications. The only requirement is that the client has a browser capable of establishing a SSL connection.
d) No changes to our internal infrastructure (servers, web pages …)
e) No changes on the client, i.e. client drivers, proxy settings in their browsers
f) Authentication into our Intranet will be done via RSA SecurID tokens
Note: This SSL Web Proxy system can be used with other Authentication methods (see notes later in the document)

Overall Goal of the System:

Listed in *Figure 1* is the network layout of the proposed solution.



The overall goal of the system is to allow a public client to be able to access our private web applications through SSL. Initially the public client will establish a SSL connection to the public SSL Web Proxy server: www.sslwproxy.com. After Authenticating, our main Intranet web page will be presented to the client. Any embedded host links within the web page will have been *parsed* and *rewritten* to point back to the external SSL Web Proxy server. Using Figure 1, embedded links will be changed as follows:

www.mainwebserver.com	becomes	www.sslwproxy.com
www.was1.com	becomes	www.sslwproxy.com/srv1
www.was2.com	becomes	www.sslwproxy.com/srv2

Now when the public client selects one of the links associated with a private web application server, he is directed back to the public SSL Web Proxy server with a new URL associated with the private web application server.

The Research Paths Taken:

My first step of the research involved SSL. More specifically the widely available and secure OpenSSL (www.openssl.org). You can download and install the tarball from their website or use the version installed with your Linux distribution. **Important note**: Throughout this document, I will be giving http/ftp links to download the needed sources. You should always check the integrity and authenticity of the file. The file should have either an associated MD5 checksum, or a PGP signature file (you will need the author's PGP public key). Common programs used for this are *md5sum* and *gpgv*.

SSL allows one to create a secure (encrypted) tunnel between a client and a server. In my case, I will be using the HTTP protocol (standard port 80) which is converted into the familiar HTTPS (standard port 443). Stunnel (www.stunnel.org) was the first system that I found to allow me to use SSL to create a secure tunnel for the HTTP protocol. Stunnel is basically a TCP wrapper for a variety of protocols (HTTP, POP, IMAP, etc). This is a great system that allows you to secure a tunnel over un-secure protocols without changing the code or the configuration of either the client or the server. I downloaded and installed the system. It worked as advertised. However I encounter a 'Design Problem' that the system could not solve, and would also direct the rest of my research.

The 'Design Problem':

In most of our internal web pages, there are embedded links to internal servers. The initial web page can be sent successfully to the client. However if the client selects one of the embedded links; he would be directed to a server that does not exist on the "public network". One option that was discussed was to setup our external DNS servers to refer back to the *SSL Web Proxy* server for all internal DNS names. This was saved as a last resort option, and thankfully was never used.

The next system that I reviewed was *SQUID* (http://www.squid-cache.org/) One of the best proxy / reverse-proxy servers available. Although I could definitely use the proxying / reverse-proxying features , it still did not help me with the "Design Problem". Getting somewhat frustrated, I then turned to one of my favorite research portals: SANS Reading Room (http://www.sans.org/rr/). Here I found two articles that gave me some good ideas and reference links, but still did not help with the problem of "*Rewriting the Embedded links*". The articles I reviewed were:

1. "A Reverse Proxy Is A Proxy By Any Other Name", Art Stricek 1/10/02
   http://www.sans.org/rr/web/reverse_proxy.php

2. "Perimeter Defense-In-Depth: Using Reverse Proxies and other tools to protect our internal assets", Lynda L. Morrison 2/18/02
   http://www.sans.org/rr/main/reverse_proxies.php

The next step in my research brought me to Apache 1.3.x (http://httpd.apache.org/) . The number one web server on the Internet is also an excellent proxy / reverse-proxy server. The existence of DSOs (Dynamic Shared Objects) makes it a flexible and highly customizable system. After reviewing the 'mod_proxy' and 'mod_rewrite' modules, I thought I had a solution. 'Mod_rewrite' is a powerful module that allows rewriting of URLS based on *regular expressions*. But as I found out while deep in the documentation, the Rewriting only occurs on URLS and not the HTML Body.

With many hours already spent on this project and still not close to solving the *'Rewriting the embedded links within the HTML page'* problem, I started looking at the new features of Apache 2.x. One of the new features was almost too good to be true: *mod_ext_filter* (http://httpd.apache.org/docs-2.0/mod/mod_ext_filter.html) This was exactly what I needed to get the project finally going. I downloaded the source for Apache 2 , configured it with the needed options, installed, and verified all was running OK. Next I added my own custom routines to the mod_ext_filter module. When I pointed my browser to the SSL Web Proxy Server, our internal web server's main web page was loaded with the embedded links Rewritten ! With this issue resolved, I then turned to the implementation of RSA SecurID as our authentication method.

Note: A quick note on Testing Methods:

The majority of testing for these systems was performed from a client on our private network. To compensate for this, I used two Sniffers (Packet Analyzers) to monitor the network traffic. The paths monitored were:

1) Between the client workstation and the "SSL Web Proxy" server. I verified that all network traffic between the two devices was encrypted.
2) Between the "SSL Web Proxy" server and our main Internal web server. I verified that the only traffic going to/from the web server was from the "SSL Web Proxy" server and not the client workstation.

Note: Authentication methods other than RSA SecurID.

The largest cost of implementing this system will be with using RSA SecurID's Authentication if you do not already have a RSA Authentication system in place.

Other Authentication methods besides RSA SecurID can be used with this system: BASIC , Auth_DBM, Auth_DBI, Auth_LDAP, etc … Authenticating using BASIC (plain text and password) is secure since all traffic will be passing through an encrypted tunnel (SSL).

Problem with RSA SecurID and Apache 2.x

The next part of this project was to add RSA's SecurID as our Authentication method. I reviewed the available options from RSA for Linux and Apache. I then downloaded the RSA ACE/Agent 5.1 for Apache. Installation was done via a simple ./install script. However I found out that only Apache 1.3.x and Stronghold 4.0 (Red Hat's Commercial Apache Server) were supported, not Apache 2.x. After opening a support ticket with RSA, I was informed that there were no plans at the moment to add compatibility with Apache 2.x ! After that setback, I was forced back to Apache 1.3.x .

Final Research Path

After searching the mailing lists for Apache and later mod_perl, I came upon a solution using the best tool on the Internet: *Perl* or specifically *mod_perl*: Perl embedded within Apache. Working directly with the Apache API, it seems that one can insert code at any of 11 stages of a HTTP transaction via a *Handler Subroutine*. I then researched deeper into mod_perl and Apache Handlers by reviewing the books: Writing Apache Modules with Perl and C, and The mod_perl Developers Cookbook. I was looking for the stages at which I can intercept the Response Object before it is sent back to the client. At this time, I could rewrite the embedded links. While researching these topic, I came across a system that was developed back in December, 1999 at ATT's Research Lab: *ABSENT* http://www.research.att.com/projects/absent/) . This system had the same goals of the system I need to develop. However there were parts of the system that would not work in our environment. One of the team members of the project: Christian Gilmore, transferred the majority of the *Rewrite* code into a perl module for Apache, *Apache::ProxyRewrite*. From the documentation for the *Apache::ProxyRewrite* module, I was certain that I finally had all the items needed to make this system work. This system will be using custom *handlers* to perform the *proxying* and the *rewriting*.

After installing the needed files (details in the following section). I pointed my browser to the 'SSL Web Proxy' server and our main internal web page came up; However the embedded links were not rewritten. After reviewing the code of *Apache::ProxyRewrite* and changing the *parse()* subroutine to handle the newer HTML tags; It worked! The next section details the Installation & Setup instructions of the *SSL Web Proxy* Server using Apache 1.3.x, mod_perl, mod_ssl and RSA SecurID.

SECTION: (During) Developing the SSL Web Proxy System :

Setting up the actual server was the first phase of this system. I chose Red Hat 7.3 as the operating system. During the installation, I chose the "Custom Installation" and installed the minimal amount of applications possible. Later on during the final phase of this implementation, I will remove additional programs. After the initial install, I ran Red Hat's Up2date agent and updated the needed applications as well as the kernel. OpenSSL, version 0.9.6b-28, was installed as two RPMS, the core *openssl-0.9.6b-28* and the needed header files *openssl-devel-0.9.6b-28.*

Next I removed Apache 1.3.x that was installed during the installation by the following commands:

```
rpm –e apache-devel
rpm –e --nodeps  apache
```

Installation of Apache 1.3x, mod_perl , and mod_ssl

Next I downloaded the source files and checksums to a temporary location. I used the standard location /usr/src/redhat/SOURCES. To extract the source files, I used the command: "tar –zxvf *filename*" which unzips and untars the files into a subdirectory.

For our system, we will need the following source files:

a). apache 1.3.27.tar.gz from http://httpd.apache.org/download.cgi
source directory: => ../apache_1.3.27

b). mod_ssl 2.8.12-1.3.27.tar.gz from http://www.modssl.org/
source directory: => ../mod_ssl-2.8.12-1.3.27

c) mod_perl mod_perl-1.0-current.tar.gz (choose mod_perl 1.0 Stable)
from http://perl.apache.org/download/source.html
source directory => ../mod_perl-1.27

Each of the systems will be installed via the standard sequence:

```
./configure, make, make install.
```

with some adjustments. The order of installation is important. Installing both mod_perl and mod_ssl will add the needed files into the Apache source directories and adjust some lower level *configure* scripts.

1) First we install mod_ssl: Here is my configure script for mod_ssl:

```
% cd ../mod_ssl-2.8.12-1.3.27
% ./configure   --with-apache=../apache_1.3.27 \
                --with-ssl=/usr/include/openssl
% make
% make install
```

2) Next we will install mod_perl. Here are my options and steps for mod_perl:

```
% cd  ../mod_perl-1.27
% perl Makefile.PL  APACHE_SRC=../apache_1.3.27/src NO_HTTPD=1 \
     USE_APACI=1 PREP_HTTPD=1 EVERYTHING=1
% make
% make install
```

This creates /updates the *src* subdirectory in the Apache source directory.

3) We will now install Apache 1.3.x. You can change the layout of the directory and file locations via the *-- with-layout=x* compile option. I prefer the default layout which places everything under one directory tree: /usr/local/apache

Configure Script (configure compiler and make options)

This is my <u>final</u> *configure* script for Apache. After everything was working, I went back and disabled all the modules that were not needed. This reduces the *footprint* of Apache as well as enhances the security

```
% cd ../apache_1.3.27

 % ./configure --activate-module=src/modules/perl/libperl.a --enable-module=so  \
--enable-module=ssl  --disable-module=asis --disable-module=cgi --disable-module=imap \
--disable-module=status --disable-module=userdir --disable-module=actions \
--disable-module=autoindex --disable-module=negotiation --disable-module=alias \

% make
% make certificate
```

This will generate the initial server key and a demo certificate needed for SSL. You will be prompted for a PEM passphrase. Make sure you record this, it will be needed elsewhere. Also you will be entering information needed for your certificate. The template for this information is stored in the OpenSSL configuration file (openssl.conf) usually located in */usr/share/ssl/*.

% *make install*

4) The final item to install is the Apache perl module, Apache::ProxyRewrite.
You can download the source tarball from CPAN
http://search.cpan.org/author/CGILMORE/Apache-ProxyRewrite-0.17/
Then run the following commands:

```
a) % tar –zxvf Apache-ProxyRewrite-0.17.tar.gz
b) % cd Apache-ProxyRewrite-0.17
c) % perl Makefile.PL
d) % make
e) % make install
```

This will install the package file (ProxyRewrite.pm) into one of your Perl Library paths. To find all your Perl Library paths (from @INC), issue the command: *perl –le 'print join "\n", @INC'* In my case, it was *usr/lib/perl5/site_perl/5.6.1* and then in the Apache subdirectory.

Note: You may be able to use the module without modifications. Review the man {documentation} page on setting the variables for your environment via the PerlSetVar. I had to change the *parse()* subroutine within the package to make it work.

Starting Apache 1.3x and the needed changes to *httpd.conf*

Start apache via the command: *../bin/apachectl startssl* . You will need to enter the PEM passphrase that you setup earlier (Did you write it down?).If you get any errors due to the associated modules not being loaded, comment out the affected line(s) in the Apache configuration file (httpd.conf).

The additions to your Apache configuration file (httpd.conf) to use the *ProxyRewrite* module are:

```
PerlModule Apache::ProxyRewrite
 # Now within A Directory, Location or Virtual Host Block you can include for example:
<Location / >
        SetHandler      perl-script
        PerlHandler     Apache::ProxyRewrite
        PerlSetVar      ProxyTo         http://myinternal.com
        PerlSetVar      ProxyAuthInfo "BASIC aGb2c3ewenQ6amF4szzmY3b="
        PerlSetVar      ProxyAuthRedirect    On
        PerlSetVar      ProxyRewrite "http://myinternal.com/ => http://pubserver.com"
</ Location >
```

Using the example outlined in Figure 1 and with a modified ProxyRewrite module (SSLProxy.pm), here are the additions to my httpd.conf file.

```
PerlModule Apache::SSLProxy

# Main Web Server
<Location />
        SetHandler  perl-script
        PerlHandler Apache::SSLProxy
        PerlSetVar      ProxyTo         http://www.mainwebserver.com
</ Location>

# Web Application Server WAS1
<Location /srv1>
        SetHandler  perl-script
        PerlHandler Apache::SSLProxy
        PerlSetVar      ProxyTo         http://www.was1.com
</ Location>

# Web Application Server WAS2
<Location /srv2>
        SetHandler  perl-script
        PerlHandler Apache::SSLProxy
        PerlSetVar      ProxyTo         http://www.was2.com
</ Location>
```

I handled the “Embedded Links Rewriting” logic within the modified *parse()* routine, so I did not need to include the *PerlSetVar ProxyRewrite* parameters.

Installing and Configuring RSA SecurID

Before installing the RSA Webagent for Apache, you will need to setup the agent host on the RSA ACE Server.

1) Add a new Agent Host (type Unix), use the internal DNS name
2) Change the *Name Locking* feature to match your environment
   Note: this must be the same for the ACE server and the RSA WebAgent
3) Uncheck the "sent node secret" field
4) Generate a configuration file (sdconf.rec) for the Agent Host
5) Transfer this file to the SSL Web Proxy Server:
   make a directory /var/ace and put the file there.
6) change ownership of the directory /var/ace to the user & group that
   Apache will run as: e.g. chown nobody:nobody /var/ace

Next we will need to install the RSA WebAgent for Apache. You can download it from their website (www.rsasecurity.com). You will need to fill out a general form before downloading. The file that I used was *WebAgent5.1.1.tar.gz*. Place this file in your source directory and unzip it into its subdirectory. You will next run the installation script: *./install*.
Note: You will need to change the script to take into account Apache 1.3.27. There is a test in there for 1.3.26. Just add ' | 1.3.27' to the condition.
Then follow the prompts and be sure that it found your /var/ace and Apache directories.
The installation makes a small change to your httpd.conf. It adds a few lines near the bottom; The most important is the *include* statement.

One problem that I encountered that cost me a few hours was an issue with the handling of MIME types (PDFS, DOC , XLS …). When I selected a link that pointed to one of these document types, a message box came up asking me to save with no option to view. However when attempting to save, another error occurred. To solve this issue, you will need to change a configuration setting via the command: *config* which is located in the rsawebagent directory. Change the following setting from 'enabled' to 'disabled':

Attempt to prevent Browser to cache protected pages [disabled]

Now restart Apache.

You can verify that the RSA SecurID Agent is working properly from the command line.

To verify the settings: % ../rsawebagent/acestatus
To test Authentication: % ../rsawebagent/acetest
Enter your user ID, PIN # + {current token passcode }

To customize the Web Templates used by RSA or change any of the other settings, review the documents (pdfs) supplied with the agent software.

SECTION: (After) Implementing the SSL Web Proxy Server:

Before this system can be used by our remote users, I need to make sure the system is as secure as possible. The overall goal of this system is to provide secure access for our remote users. I do not want to introduce any new vulnerability to our existing network infrastructure.

Securing the *SSL Web Proxy* Server:

In addressing the issue of securing the SSL Web Proxy server, I took a multi-layer approach. The inner layer of security will be the securing of the Apache web server. The next layer of security will be the “hardening” of the Linux server that runs the SSL Web Proxy. The final layer of security involves the network placement and verification of the “upstream network security”.

Securing the Inner Layer: Apache

If you used the compile options from above, then a lot of the work has already been done. By disabling such modules as mod_cgi, mod_userdir, etc … we have eliminated potential vulnerabilities which could be exploited. A lot of the compromises come through CGI scripts. Since we enabled the use of DSO (Dynamic Shared Objects) and to be 100 % certain, you will need to verify that your configuration file does not contain any commands that could load the modules dynamically. For example: Loadmodule cgi_module modules/mod_cgi.so …
Addmodule mod_cgi.c

You can check which modules have been compiled into Apache by the command: *httpd –l* . To get the available *Configuration Directives* for your compiled version of Apache, use the command: *httpd –L*.

Also verify that the user/group defined in the Apache configuration file has limited rights. When Apache starts, it will always run as *root* in order to setup the environment and then spawn its child processes. It is the child processes that do all the work of handling the HTTP transactions. The child processes run as the user/group specified in the configuration file (httpd.conf). Following the principle of ‘Least Privilege’, ensure the user/group specified in your configuration has limited permissions and no shell.

Changing the port(s) that the server will listen on will also help. Since this server is functioning as a SSL proxy only. I changed the *Listen* Directive to only use Port 443 (Standard Secure HTTP port). A simple port scan can verify that indeed only that port is open.

Another step is to make your Containers (Directory, Location, Files) as restrictive as possible by including the following in each Container.

```
Options none
AllowOverride none
```

To improve security of the Directory and File Hierarchy, make sure that *root* is the owner of all directories except *./htdocs*. Also change the mode of all directories to 755. File modes within the directories should be set to 644, except in *./cgi-bin* (750), *./libexec* (755), and the main executable *httpd* set to 511.

Securing the Next Layer: Hardening the Server

In securing the server, I first examined the list of installed programs by the command: *% rpm –qa* | *sort > curprogs.lst* . From this list, I removed the unnecessary programs via the command: $ rpm –e 'package'. Note: This procedure will not find programs installed without the RPM utility.

I next reviewed the increasingly popular mechanism for hardening Linux servers: *Bastille Linux* (http://www.sans.org/projects/bastille_linux.htm). This collection of perl scripts automates and takes a lot of the complexity out of identifying and resolving the security issues associated with setting up a Linux server. In addition to hardening your server, Bastille-Linux is also a great educational tool. Detailed explanations are given during each step of the 'Hardening' process. For the Bastille user interface, I chose the ncurses (text) and not the X (Tk) GUI. I used the following files: *perl-Curses-1.05-10.i386.rpm*, and *Bastille-2.0.4-1.0.i386.rpm.* During the *Bastille-Linux Hardening* scripts, I chose to setup *SysLog Forwarding* to one of our centralized SysLog servers on the private network.

Once the Bastille scripts had completed, and the needed adjustments performed on the server, I proceeded to adjust my internal Firewall (iptables/netfilter). I took a conservative approach and *denied all* initially. I then trusted my l0 and eth0 (private) interfaces. Finally I enabled just TCP traffic on port 443 (HTTPS) on the eth1 (public) interface.

The next step was to obtain the latest version of *TripWire* and install it. I used the steps outlined in SANS's GSEC Security Toolkit in setting up *Tripwire*. See Chapter 3 on the steps and notes on setting up Tripwire. I then created the policy, the initial checksum database, and performed some quick tests. Once I was satisfied with the policy, I created a cron job to run it daily.

The final step that I performed before activating the 'public' interface was to change the *Run Level* of the server. Not only will this enhance security but will also increase the available resources. I changed the *Run Level* from the common default of 5 (multi-user / X Windows) to 3 (no X Windows).

Introducing the *SSL Web Proxy* Server to the Internet

After securing the SSL Web Proxy Server and adjusting the firewall (iptables) filters for the public interface, I then enabled the "public" interface (eth1). Next I performed a quick verification that the system was working as planned. From a 'public' client, I entered the external DNS name of the *SSL Web Proxy* server into my browser: The RSA SecurID Login page was displayed. After entering the needed RSA SecurID parameters, our internal Intranet home page was loaded in the browser. Thus without any changes to the client's configuration and without loading any drivers, I was able to access our company's main internal web page from the Internet in a secure manner.

Before spending the time to verify all of our Intranet's web page links, I proceeded to perform some security scans of the SSL Web Proxy Server using the following tools:

a) For Port Scans I used *NMAP* (www.insecure.org/nmap/)
b) For Vulnerability Scans I used *Nessus* (www.nessus.org)

These security scans were done on the server's 'public' interface.

Still feeling somewhat paranoid, I next added two rules to our NIDS (network Intrusion Detection System): *Snort* (www.snort.org*)*. I created temporary rules to log all traffic going to and from the SSL Web Proxy server. In addition, I set up a *Sniffer* to monitor the traffic on the 'private' interface. The final item for monitoring that I set up was *SysLog Forwarding* to our central syslog server located on our private network.

From the results of the security scans, I had to make some minor adjustments to the server and firewall rules. After the adjustments, I proceeded to perform more detail testing of the SSL Web Proxy server. The majority of testing involved selecting all the links on our Intranet web pages and modifying the *parse()* subroutine of the Apache perl module to handle the new internal hosts. In addition to the high-level testing, I also examined the network traffic between the 'public client' and the *SSL Web Proxy* server. All traffic was as expected: encrypted.

Now that the system is in place and being used by a subset of our remote users, my attention turns to the operational tasks of the system as well as the next phase of the system. Some of the day to day tasks that I foresee with this system are:

1) Contacting remote users to check on any performance issues or areas where web pages are not working.
2) Monitoring of Logs:
    a) Central Log Server
    b) Snort Logs/Alerts
    c) Tripwire Alerts
3) Adding new internal hosts to the *Rewriting/Proxy* modules

From time to time during the operation of this system, you may have the need to modify the Server Key and/or Certificates. I have included some general information concerning OpenSSL and Apache in Appendix A. Besides the important files and where they reside, I have included useful commands to view and maintain server keys and certificates.

Recent Upgrades Because of Vulnerabilities:

One needs to keep abreast of any security alerts associated with the components we have assembled here: Apache 1.3.x , mod_ssl , mod_perl and also the core OS .

For instance, (CAN-2003-0078, CAN-2003-0147 , CAN-2003-0131) were issued within the last month. These are concern with the recently discovered OpenSSL vulnerabilities. In addition Red Hat issued an alert a few weeks ago concerning a vulnerability in the kernel with *ptrace* that could lead to elevated privileges. *RHSA-2003:098 - Security Advisory.*

Because of these recent security alerts, Red Hat has released a new version of their kernel and patches for OpenSSL. In addition, there is an updated tarball for mod_ssl (2.8.14-1.3.27).

Future Enhancements for *SSL Web Proxy* Server:

The next stage of this system will include the ability for public clients to run true Client-Server applications. A prime example in our environment is the thin client: Citrix. Initial research on this has directed me to using JAVA and *JSSE* (Java Secure Socket Extension). I believe Java will allow me to go deeper into the network protocol layer to provide a secure tunnel. I should be able to continue to use Apache with Java , i.e. *TomCat.*

Feel free to contact me to see how this stage is progressing.

REFERENCES

Bibliography

- Network Security with OpenSSL , by John Viega, Matt Messier , and Pravir Chandra
  O' Reilly & Associates , 1st Edition June 2002, ISBN: 059600270X

- Apache: The Definitive Guide, 3rd Edition, by Ben Laurie and Peter Laurie
  O' Reilly & Associates , 3rd Edition December 2002, ISBN: 0596002033

- Writing Apache Modules with Perl and C , by Lincoln Stein and Doug MacEachern
  O' Reilly & Associates , 1st Edition March 1999, ISBN: 156592567X

- mod_perl Developer's Cookbook , by Geoffrey Young, Paul Lindner, and Randy Kobes
  Sams, 1st Edition January 2002, ISBN: 0672322404

- Perl & LWP, by Sean M. Burke
  O' Reilly & Associates , 1st Edition June 2002, ISBN: 0596001789

- Building Secure Servers With Linux , by Michael D. Bauer
  O' Reilly & Associates , 1st Edition October 2002, ISBN: 0596002173

- GSEC Security Essentials Toolkit , by Eric Cole, Mathew Newfield, John M. Millican
  SANS Press, 1st Edition March 2002, ISBN: 0789727749

Online Resources

- Apache 1.3.x http://httpd.apache.org/docs/

  Mailing List Archive: http://marc.theaimsgroup.com/?l=apache-httpd-users

- Mod_SSL http://www.modssl.org/docs/2.8/
  http://www.modssl.org/docs/2.8/ssl_intro.html.

- Mod_Perl http://perl.apache.org/docs/1.0/guide/index.html
  http://modperl.com:9000/

  Mailing List Archive:
  http://marc.theaimsgroup.com/?l=apache-modperl&r=1&w=2#apache-modperl

- Articles from SANS Reading Room
  1. "A Reverse Proxy Is A Proxy By Any Other Name" , Art Stricek 1/10/02
     http://www.sans.org/rr/web/reverse_proxy.php

  2. "Perimeter Defense-In-Depth: Using Reverse Proxies and other tools to protect our internal assets", Lynda L. Morrison 2/18/02
     http://www.sans.org/rr/main/reverse_proxies.php

- *Bastille Linux:* http://www.sans.org/projects/bastille_linux.htm

- Web Security: http://www.w3.org/Security/faq/
  http://www.apacheweek.com/security/

Appendix A

Using OpenSSL to View and Maintain the Server Keys and Certificates

Files created after requesting a PassPhrase and entering your company's information:

{Apache Directory}

.../conf/ssl.key/server.key	Your private key file
.../conf/ssl.crt/server.crt	Your X.509 certificate file
.../conf/ssl.csr/server.csr	The PEM encoded X.509 certificate-signing request file. It is this file to send to a CA to get a real server server certificate to replace "server.crt"

Some command commands:

To view the contents of a private key file in plain text (PassPhrase may be needed):

```
% openssl rsa -noout -text -in <name>.key
```

To view the ingredients of a particular certificate file in plan text

```
% openssl x509 -noout -text -in <name>.crt
```

To view the ingredients of a particular CSR file (Certificate Signed Request) in plan text

```
% openssl req -noout -text -in <name>.csr
```

Creating Keys – Certificates – Self Assigning

Note: The following are excerpts from the mod_ssl (faq):

(1) Create a RSA private key for your Apache server (will be Triple-DES encrypted and PEM formatted):

$ openssl genrsa -des3 -out server.key 1024

Remember PEM PassPhrase

(2) Create a Certificate Signing Request (CSR) with the server RSA private key (output will be PEM formatted):

$ openssl req -new -key server.key -out server.csr

{Enter required / optional info as outlined in *openssl.conf*: org name, FQDN … It is important that the FQDN matches your server name}

(3) To be your own CA (Certificate Authority)

(a) **$ openssl genrsa -des3 -out ca.key 1024** (Remember PEM PassPhase)

(b) Create a self-signed CA Certificate (X509 structure) with the RSA key of the CA (output will be PEM formatted):

$ openssl req -new -x509 -days 365 -key ca.key -out ca.crt

Enter info for your CA (MyCertificateAuthority)

(c) Sign via script within mod_ssl distribution pkg.contrib/ {sign.sh}

./sign.sh server.csr This creates a server.crt file