CentOS

Apache Web Server Administration

1 Introduction

A Web server is a server that is responsible for accepting HTTP requests from web clients and serving them HTTP responses, usually in the form of web pages containing static (text, images etc) and dynamic (scripts) content. The Apache Web server has been the most popular and widely used Web server for the last decade. It is used by approximately 50% of all websites. Apache is cross-platform, lightweight, robust, and used in small companies as well as large corporations. Apache is also free and open-source.

The Apache Web server has almost endless possibilities, due to its great modularity, which allows it to be integrated with numerous other applications. One of the most popular bundles is the LAMP Web server application stack, which includes the Apache Web server alongside MySQL, PHP, Perl, and Python. The Apache Web server is developed by the Apache Software Foundation. You can read more about Apache on Wikipedia.

Being able to configure and secure the Apache Web server is one of the most important tasks for a (Linux) system administrator. Almost every company has some sort of a website that advertises it, including intranet pages that are used by the company's workers. The Web interface is used for many tasks beside pure browsing, including tasks as simple as meal orders and shift rosters, but also important tasks like administration of databases. In most cases, a local web server is setup to accommodate these needs.

If you are working for a company that hosts public websites, the task becomes even more complicated. Web sites are used to serve content to billions of users daily. Whoever controls this content - controls the World Wide Web, from news and blogs to financial transactions. Web servers are hubs of information and power. Misconfigured or compromised servers can expose a large number of people to undesired content and potentially incur huge damages to involved parties.

Running a Web site is much more than opening a port and serving a few HTML pages. There are tremendous network usability and security considerations that must continuously be met, evaluated and improved in order to maintain a safe and effective Web server. In this Part of the Book, we will learn how to properly setup and run the Apache Web server, including the secure (HTTPS) server.

Note: Apache Web server configuration demonstrated on CentOS 8.4.

2 Basic Setup

In this chapter, we will setup a Web server that will serve pages on our internal network. We will perform the most basic setup with the minimum number of steps required to get the server running. Later, we will slowly expand, introducing new features and options.

2.1 Verify installation

First, we'll verify that Apache is indeed installed. If you get an empty prompt or a message saying the package is not installed, you will need to download and install it. If the shell displays the package name and version, you're good to go.

```
rpm -q httpd
```

2.2 Package files

Rule no. 1: don't panic! The list before you might seem intimidating at the moment, but that is simply because you are not yet familiar with Apache. But don't worry. For now, treat the list as a reference only. At this stage, you don't need to know anything or remember anything. We will slowly cover everything, step by step.

Now, let us overview the location and purpose of the files used by the Apache server. Please note that the list is partial and includes only the most important entries. We will slowly expand this list as we go through the document.

Table 1: Apache Web server files

File name	Description
/usr/sbin/httpd	server binary
/etc/httpd	directory containing server configuration files
/etc/httpd/conf	directory containing main configuration files
/etc/httpd/conf.d	directory containing configuration files for individually packaged modules, like ssl, php, perl etc
/etc/httpd/logs	symbolic link to /var/log/httpd
/etc/httpd/modules	symbolic link to /usr/lib/httpd/modules
/etc/httpd/run	symbolic link to /var/run
/usr/lib/httpd/modules	server modules
/var/log/httpd	server log
/var/run	runtime variables
/var/run/httpd.pid	server process ID
/var/www/html	public html files

2.3 Main configuration file(s)

The main configuration file is **/etc/httpd/conf/httpd.conf**. This file is well commented and self-explanatory. It contains quite a large number of settings, but we'll concentrate on just the few necessary to setup the server.

2.3.1 Backup

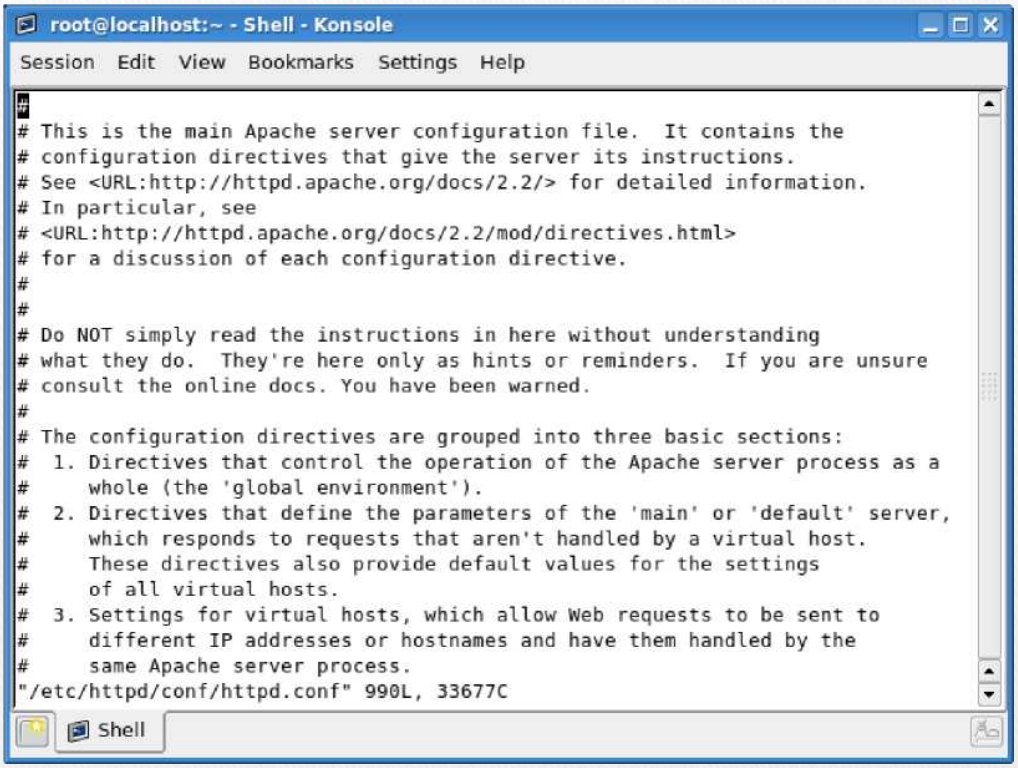
This is one of the most important things to remember. Always retain the copy of the original file so you can easily revert to the default. At the very least, do NOT delete default lines; instead, just comment them out so you'll be able to see what the original settings read and refer to them.

```
cp /etc/httpd/conf/httpd.conf
    /etc/httpd/conf/httpd.conf-default
```

2.4 Edit the httpd.conf configuration file

Let's open this file in *vim* text editor and review the most important entries. The file has many options - but we need only a few. In fact, you will need to change just a *single* line to create your server and get it running. However, you should be familiar with some other settings before launching the server. This is what the file looks like - at least the beginning of it:

Figure 1: Editing httpd.conf main configuration file



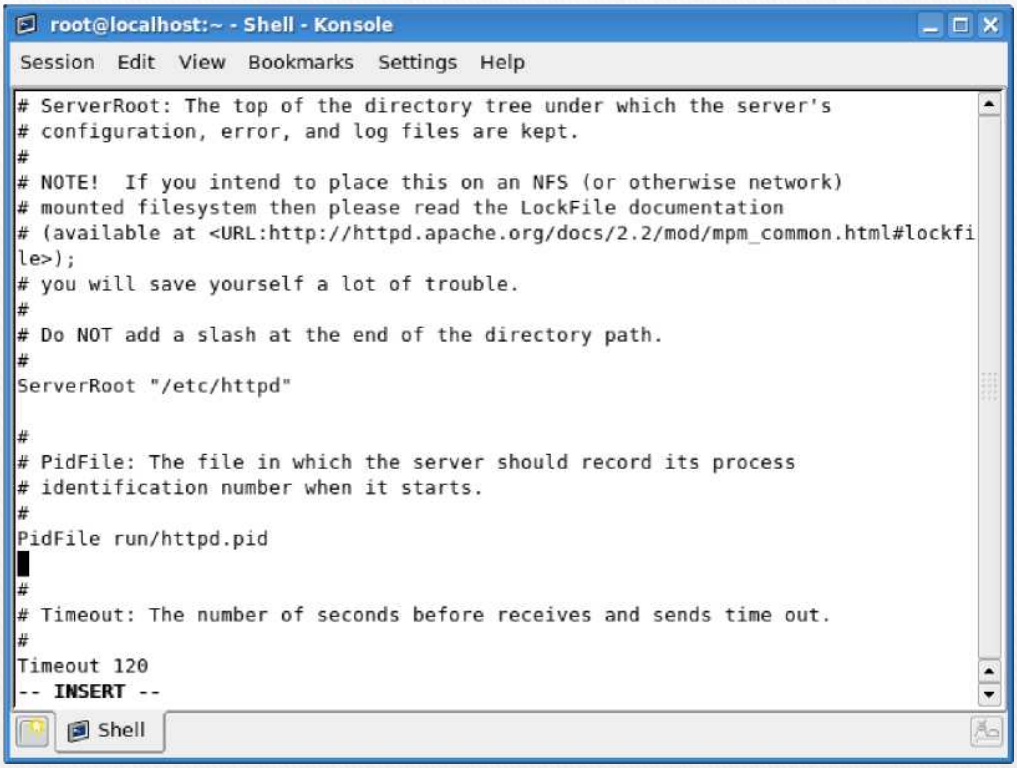
Let's go over the most important entries you should remember for now.

2.4.1 ServerRoot

ServerRoot is the path to the server's configuration, error and log files. It is possible to change this path, provided all the necessary files are copied to the new location accordingly. We will later review this concept as a part of the security measure known as the Chroot Jail, but more about that later. The default location is */etc/httpd.*

As you can see, the file is rich with easy-to-understand comments. Apropos the comments, please note that you should not place a trailing slash at the end of the specified path.

Figure 2: ServerRoot directive



2.4.2 PidFile

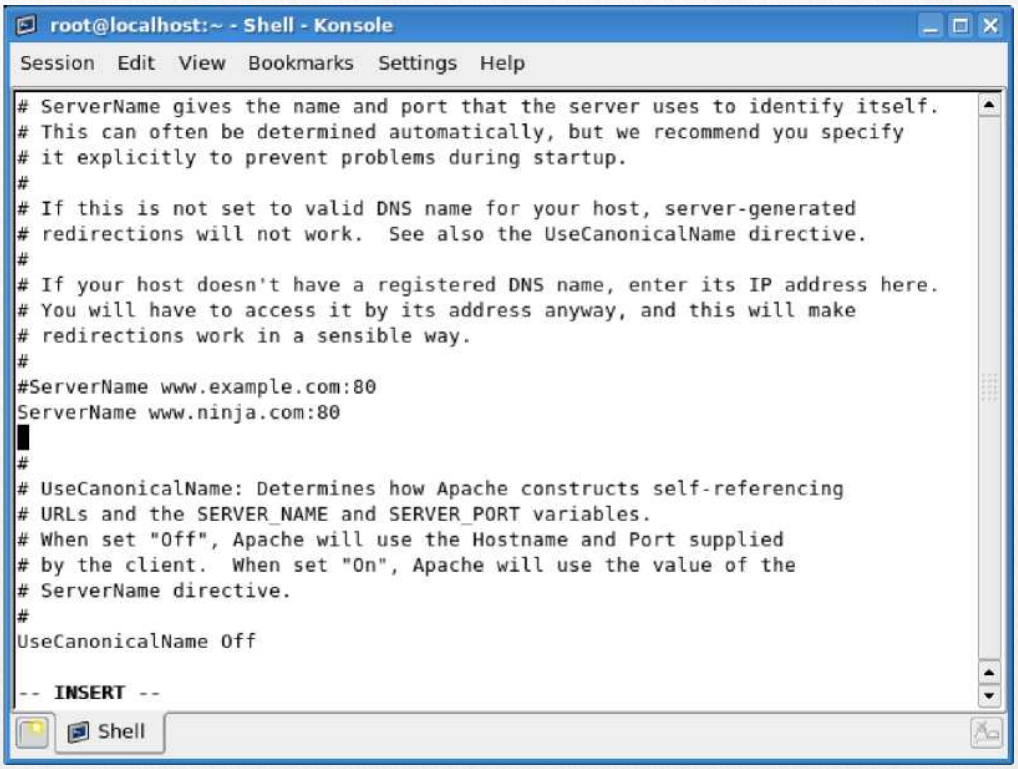
PidFile is the process identification number for the *httpd.* This process number is important, because Apache spawns numerous child processes when running to accommodate the web traffic. It allows you to monitor and manipulate your server processes. See image above.

2.4.3 ServerName

This is the one setting you will have to change to get your server running. This is where you declare the name of your website. I will use *www.ninja.com* - just a random name with no association whatsoever to the real site bearing this name. The generous comments in the file remind us that if we do not have a registered

DNS name, we should use an IP address. Indeed, we're going to use the *hosts* file to demonstrate the address-to-name translation.

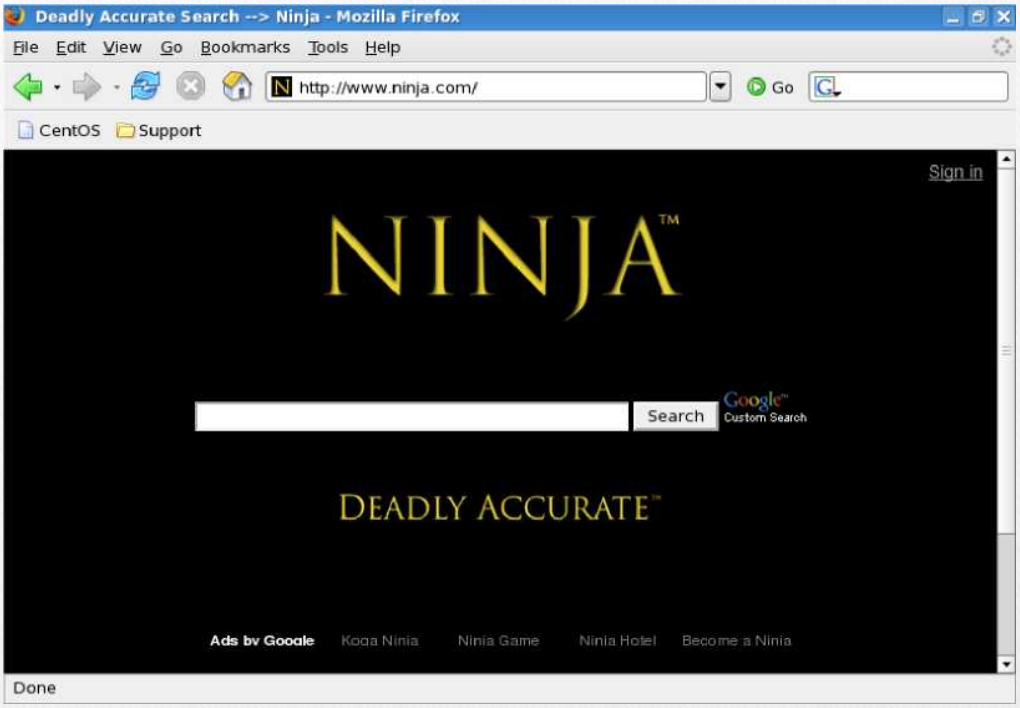
Figure 3: ServerName directive



2.4.4 Add site to /etc/hosts file

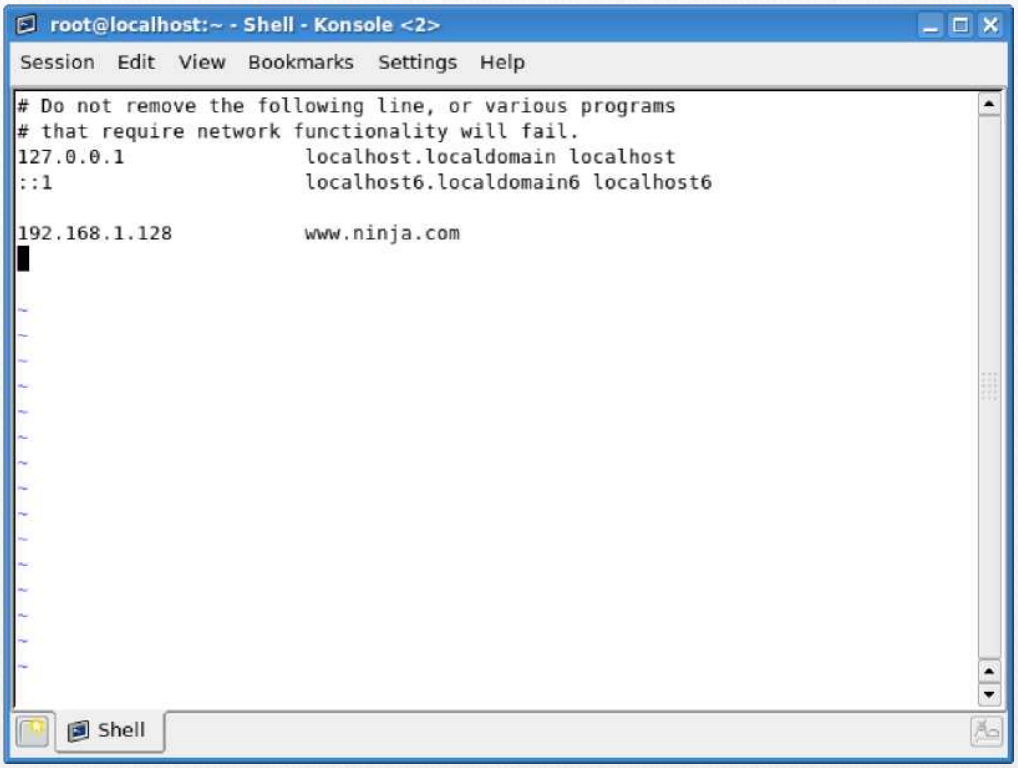
As you already know, the *hosts* file allows easy matching of names to IP addresses. In general, using the *hosts* file is a good way of testing your IP-to-name (or vice versa) configurations before committing these changes into a production environment. First, with no new entries added to the *hosts* file, typing *www.ninja.com* in the address bar of a web browser takes us to the site itself (on the Internet).

Figure 4: External web site loaded



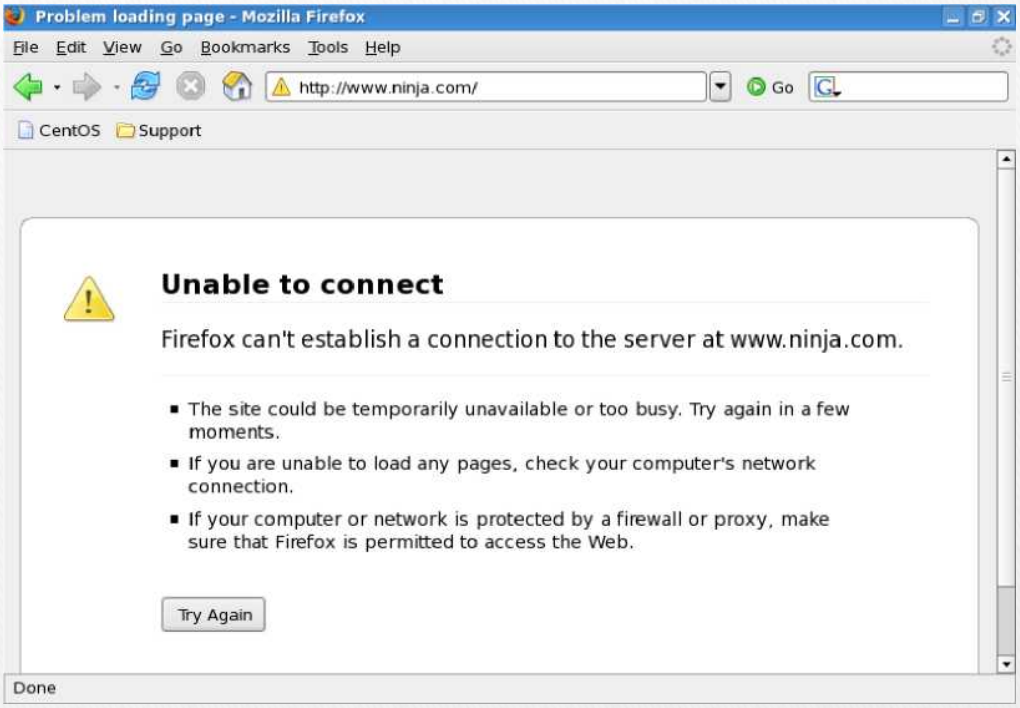
Now, we shall edit the file and add an entry, pointing *www.ninja.com* to a local IP address.

Figure 5: Editing /etc/hosts file



After saving the *hosts* file, we can no longer see the Internet site. Furthermore, we don't get any fancy results from our own Web server, because it is not running yet.

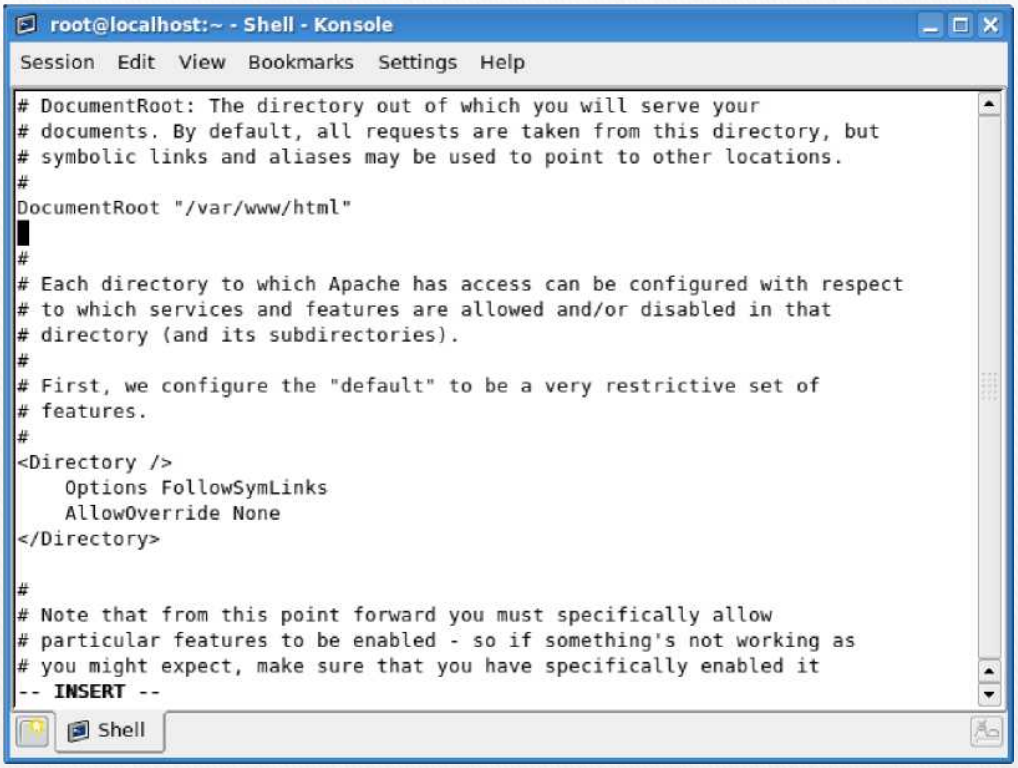
Figure 6: Unable to connect to site



2.4.5 DocumentRoot

DocumentRoot tells you where your web documents (html files, images etc) should be located. It is possible to reference files in other directories using aliases and symbolic links. The default directory is */var/www/html.*

Figure 7: DocumentRoot directive

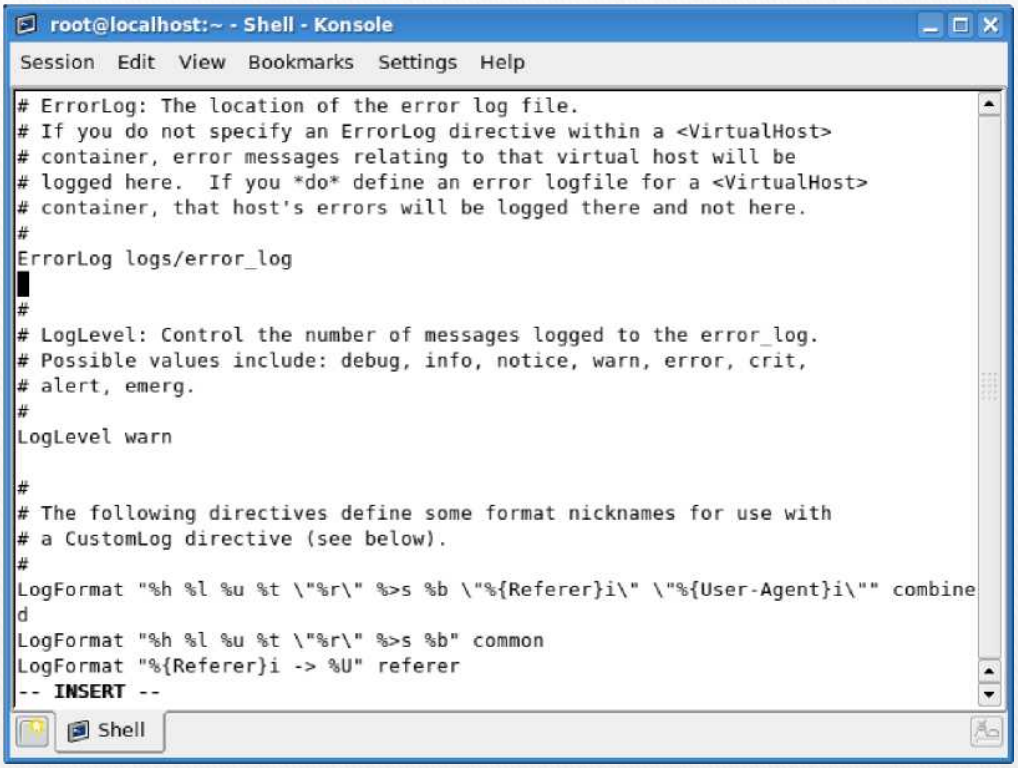


2.4.6 ErrorLog

ErrorLog tells you where the log containing all server errors is located. This file is critical for debugging and solving server misconfiguration problems and for proper traffic shaping. By default, all messages with the value of warning *(warn)* and higher will be logged. This is described in the *LogLevel* directive just below.

The default location is *logs/error_log.* Please note that this is relative to the *ServerRoot.* Therefore, our log file is*/etc/httpd/logs/error_log.* However, let us not forget that */etc/httpd/logs* is a symbolic link to */var/log/httpd.* Thus, finally, the actual error log is */var/log/httpd/error_log.*

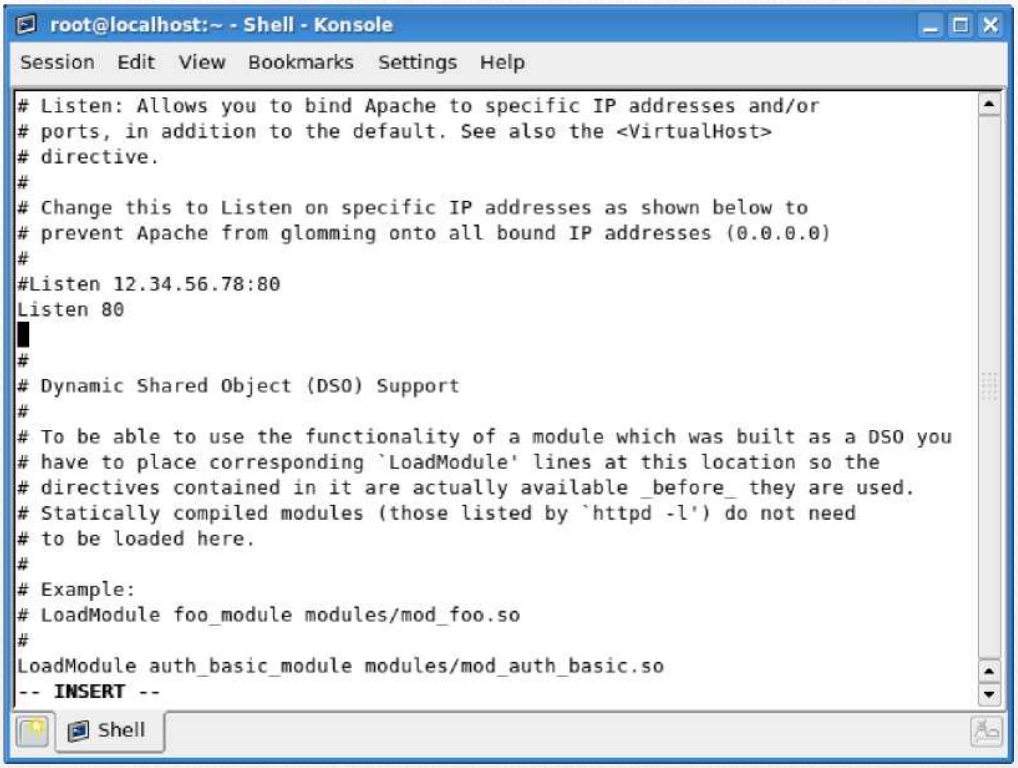
Figure 8: ErrorLog directive



2.4.7 Listen

The *Listen* command tells the Web server what ports to use for incoming connections. By default, port 80 is used, although any one or several can be used. The accepted conventions calls for using port 80 for non-secure web communications (without any encryption of traffic). Secure web communications are normally handled on port 443.

Figure 9: Listen directive

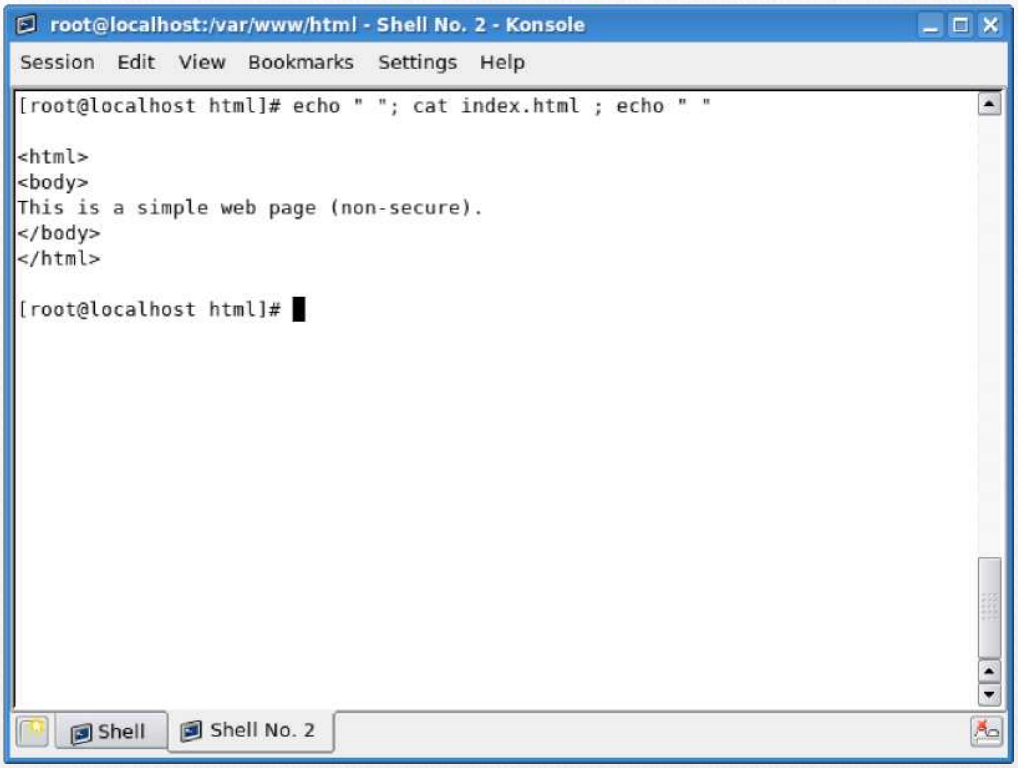


That's it. These are all the settings you need to know for now and tamper with in order to successfully launch the Web server.

2.5 Create your HTM L documents

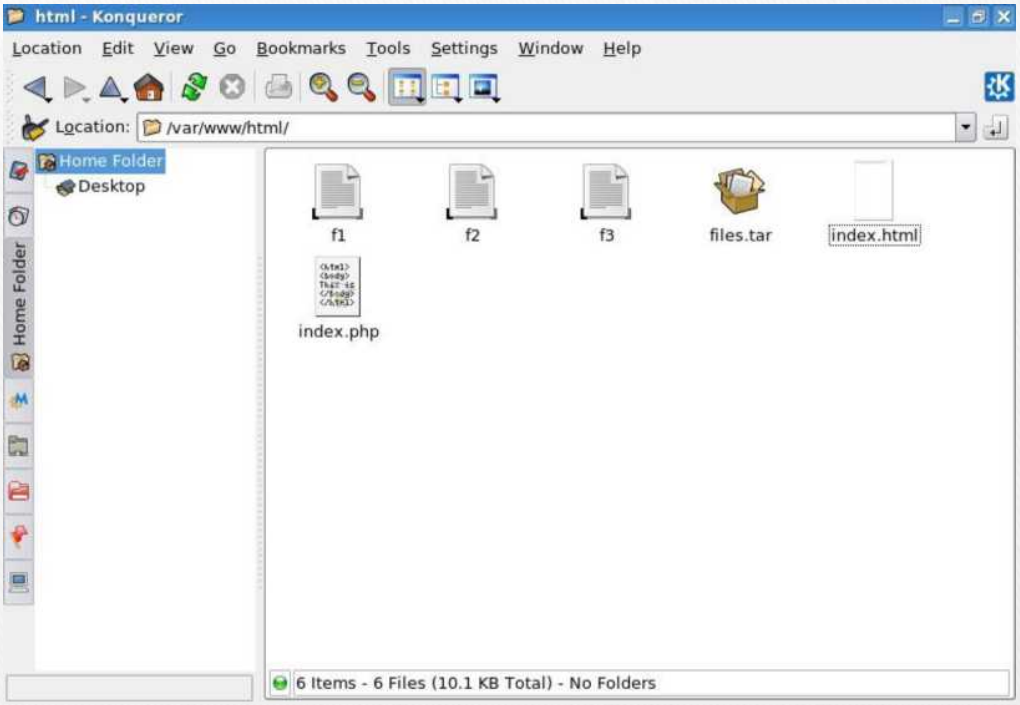
Now, just to make things more interesting, we shall create a number of files and place them in the *DocumentRoot* directory *(/var/www/html),* including a simple index.htm l file. Here's the source of our *index.html* file (the two echoes are used to make the output easier to read):

Figure 10: Creating sample HTML document



And here is the preview of files we have in the *html* directory:

Figure 11: Contents of /var/www/html directory



Now that we know what we have, it's time to power up the server.

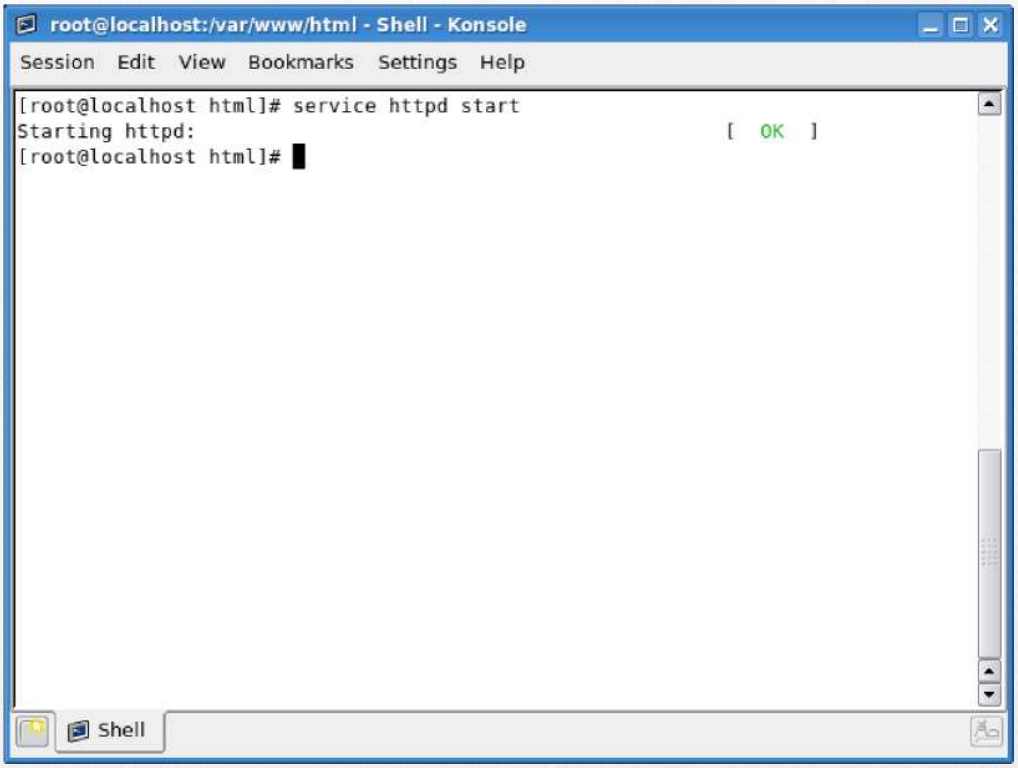
2.6 Start the Web Server

Start the *httpd* service:

```
service httpd start
```

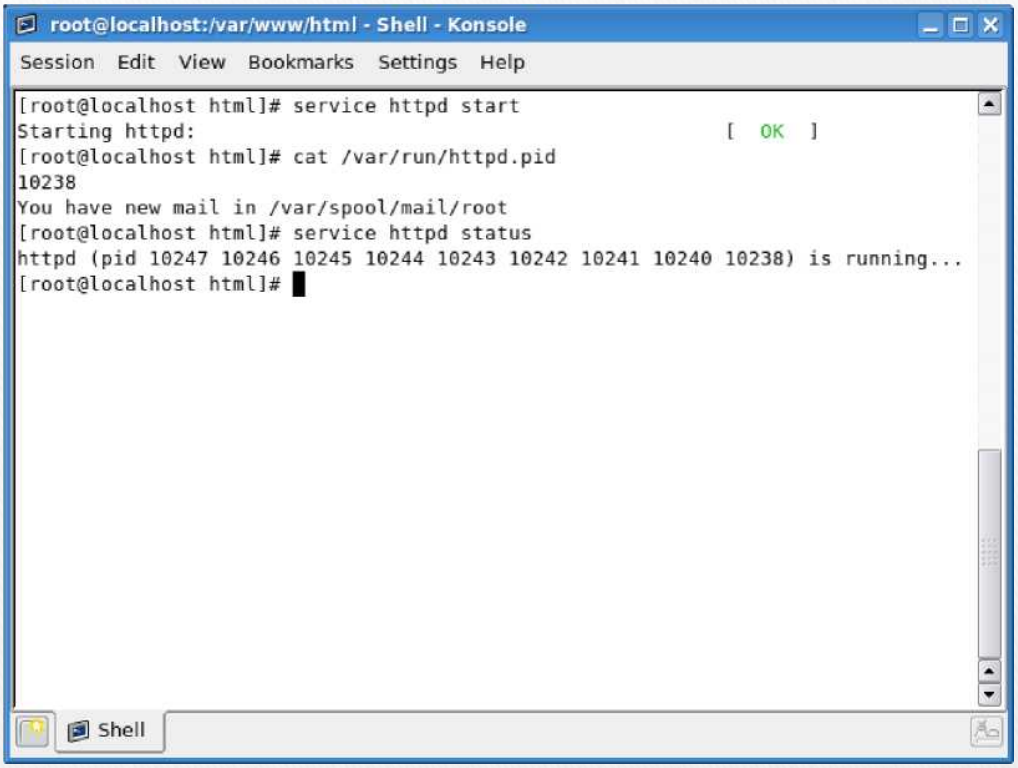
If everything worked out fine, the web server should start without any errors and you should see the following image:

Figure 12: Starting httpd service



Still, it does not hurt to check the status of the service or verify its process ID:

Figure 13: httpd service status

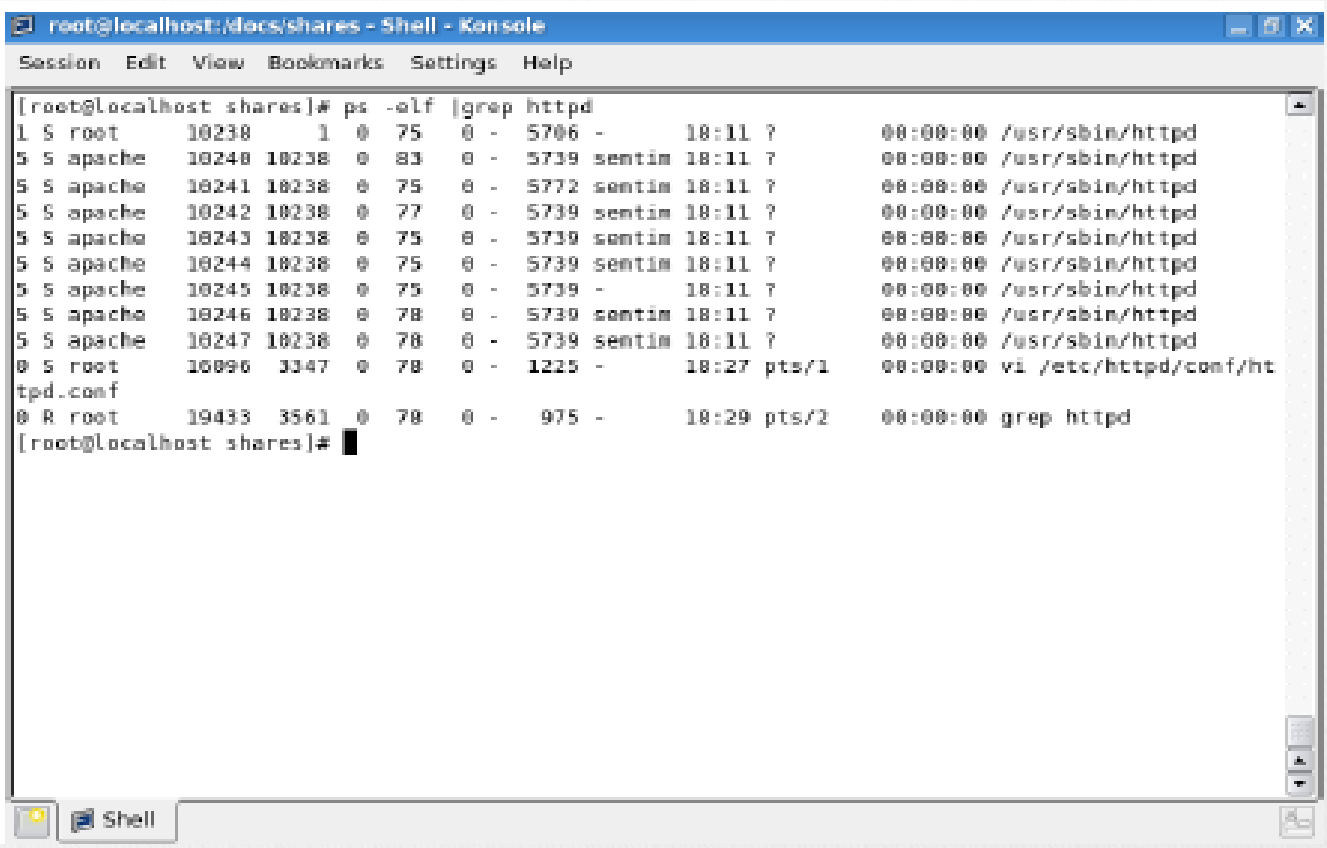


There are 9 processes running for Apache. This may be confusing, but there's a very simple explanation for this. In the *httpd.conf* file, you will find a directive called *StartServers.* This directive tells the Web server how many server processes to launch on startup. The default setting is 8 server processes.

Once started, the Web server dynamically kills and creates processes based on the traffic load, with the number of server processes fluctuating between *MinSpare-Servers* and *MaxSpareServers.* So far, everything figures out just nicely. Now, let's make another check.

The Apache Web server, if configured to listen on port 80 (or any "secure" port below 1024) must be started as root. Otherwise, it can also be started by regular (non-root) users. As a security precaution, the server processes spawned by Apache run as user *apache,* which belongs to the group *apache.* Indeed, we can easily verify that:

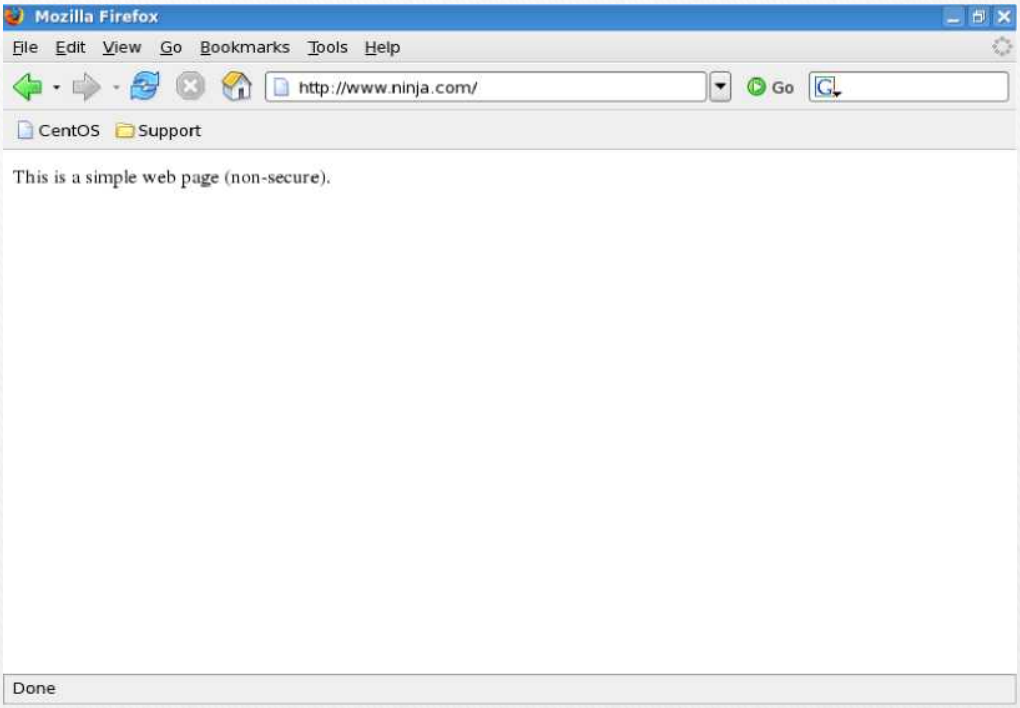
Figure 14: Process listing for httpd



2.7.1 Local access

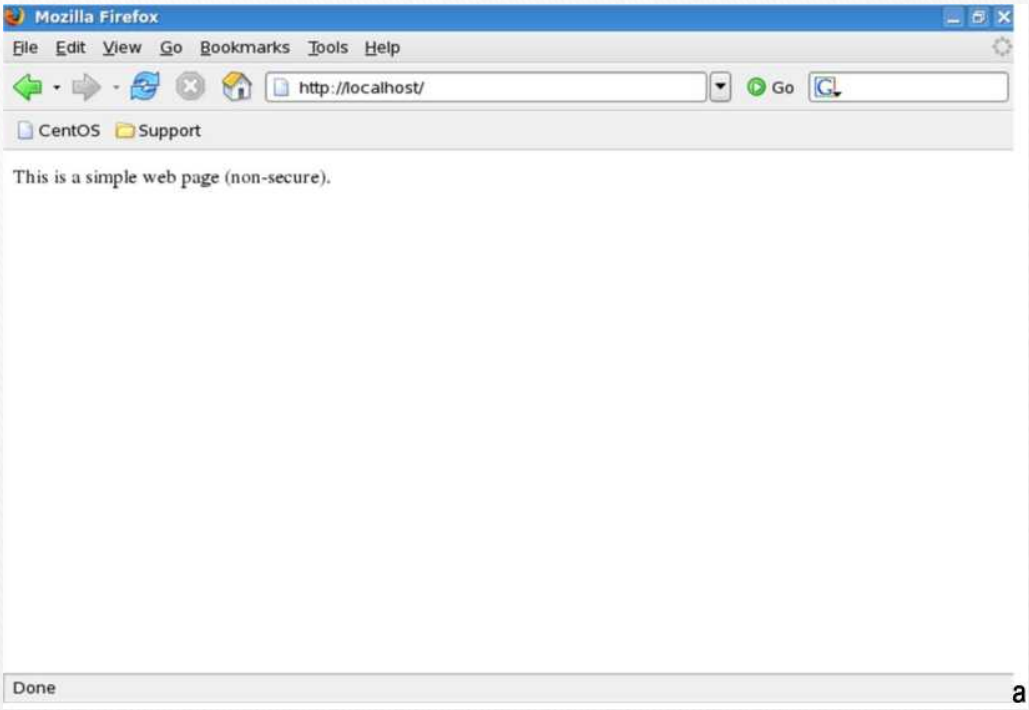
Now, let's access our homepage. Open a web browser and type *www.ninja.com* in the address bar. Earlier, we were unable to access it, even though we have specified the entry for our website in the *hosts* file. This was because the server was not running. But now, *www.ninja.com* resolves to our custom webpage.

Figure 15: Local access to website



Our server works. Alternatively, we could have simply accessed it by typing *localhost* in the web browser address bar.

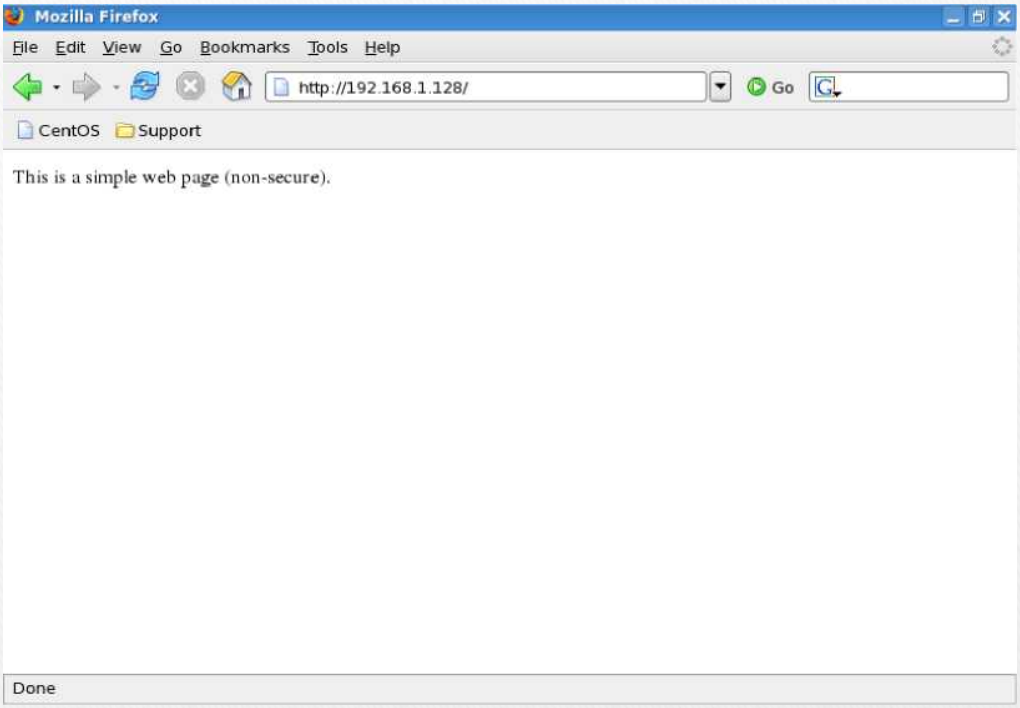
Figure 16: Local access via localhost



2.7.2 Internal & external access

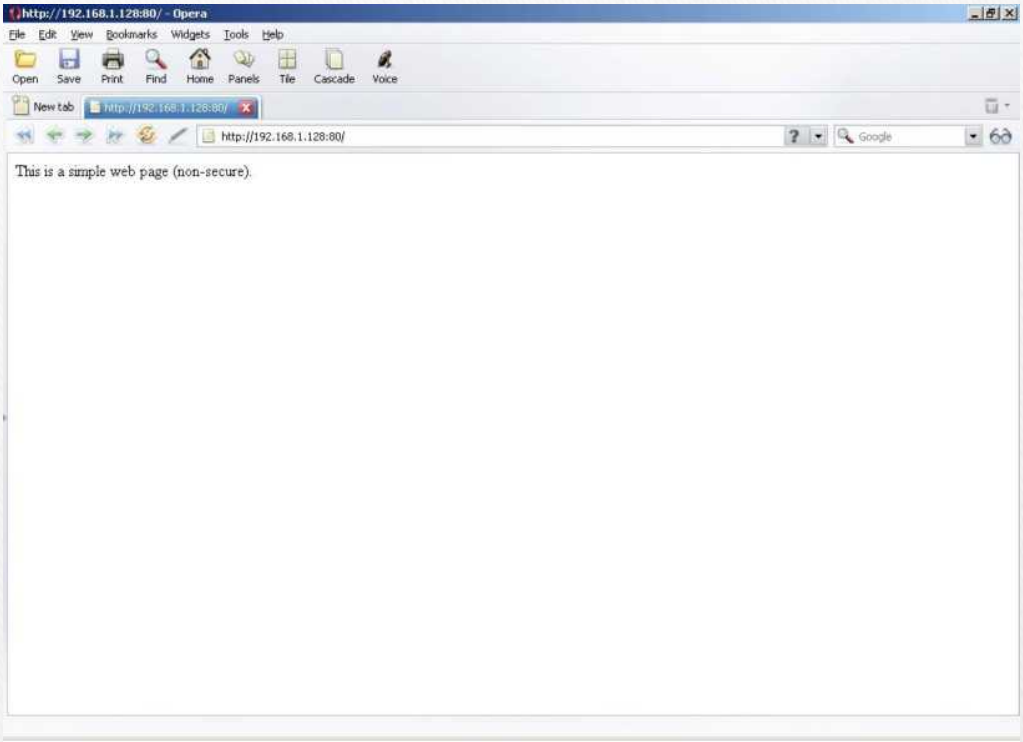
Accessing the web server locally is not the most challenging thing to do. Let's access it from other machines. Here's our webpage, seen in another CentOS machine, belonging to the same subnet:

Figure 17: Internal access to website



Here's our webpage, seen in the Opera browser running on a Windows machine:

Figure 18: Internal access from a Windows machine



As you might expect, using *www.ninja.com* on the other machines does not yield the desired result. This is because there is nothing telling these machines to match the name to the IP address of our server (192.168.1.128). On the server itself, we overcame the problem using the *hosts* file.

Theoretically, we could do the same thing on every host on our network, but this is slightly impractical and cumbersome. However, this will not solve the problem of accessibility from hosts that we have no control of, outside our local network. To overcome this monumental problem, we will need to use name resolution by configuring and running a Domain Name System (DNS) server[1].

Meanwhile, everything works as we've expected. Soon, we will go over some advanced configurations.

[1] **Discussed separately.**

2.8 Summary of basic setup

To make things simple and clear, here's an overview of the steps you will have to take to setup and launch Apache:

- Verify installation of the Apache RPM.
- Backup the */etc/httpd/conf/httpd.conf* main configuration file.
- Open it in the *vi* text editor and review the options listed therein.
- Setup the *DocumentRoot* directive (default */var/www/html*).
- Setup the *ServerName* directive (for example, *www.ninja.com).*
- Optionally setup other directives (like *ServerRoot, ErrorLog, Listen* etc).
- Configure the */etc/hosts* file so that you can access the website by name.
- Create a sample HTML file and place it in the *DocumentRoot* directory.
- Start the *httpd* server.
- T est the setup by accessing the web site.

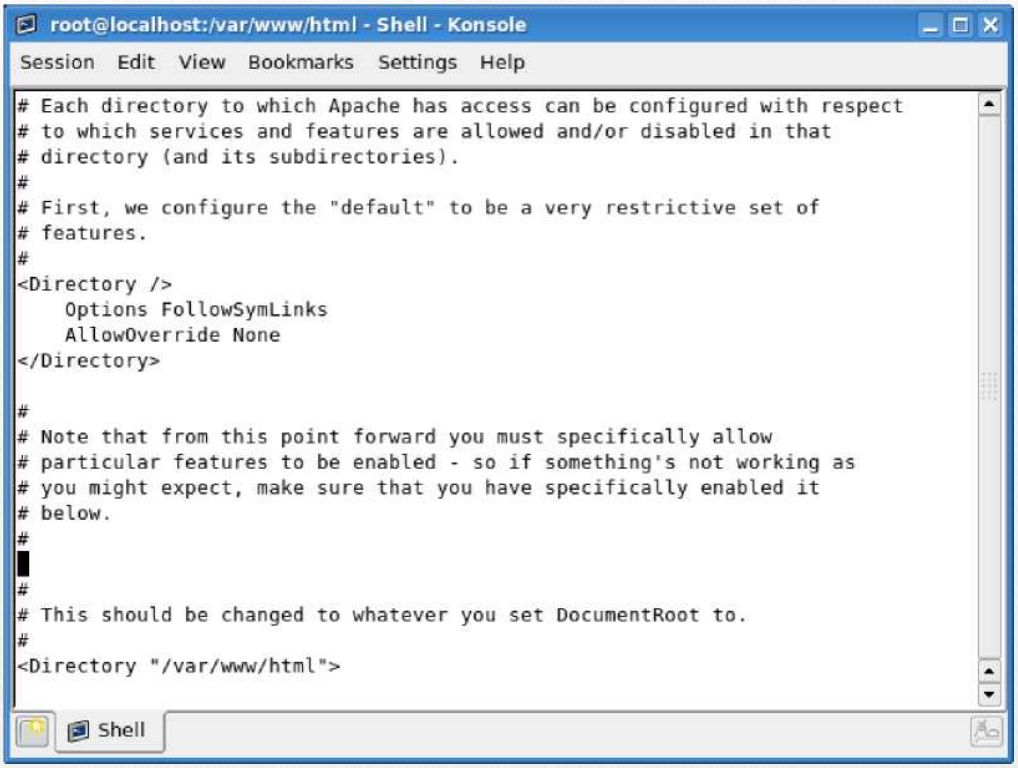
3 Advanced setup

The *httpd.conf* file can be extensively customized using a range of directives. We have studied a few and will now review several more. Please note that it is impossible to list every single directive here. Nevertheless, we will go over some of the more useful and practical directives, which will greatly enhance the usability (and also the security) of your web server.

3.1 Directory tags

Directory tags allow you to specify the configurations separately for each directory serving the web pages. If you are familiar with HTML and CSS, then using containers m ight be the simplest analogy. This allows you to serve content to specific IP ranges while denying other ranges, limit access to certain files, set the behavior of pages contained in these directories, and more.

Figure 19: Directory tags example



Just about any directory can be listed, although it is not necessary. The most sensible solution is to setup very restrictive parameters to the root (/) directory and custom, desired parameters to directories inside *DocumentRoot. Directory* tags take the following form:

<Directory directory_path> tag begins a block. Next, follows a series of options defining what users accessing web pages located in this directory can do. *</Directory>* tag closes the block. Again, this is very analogous to HTML tags. Here's a sample block, showing the default settings applied to the root (/) directory:

```
<Directory />
    Options FollowSymLinks
AllowOverride none
</Directory>
```

Let's try to understand what we have here:

<Directory /> declares the block for the root (/) directory and all subdirectories.

Options directive declares which server features are valid for the specified directory; FollowSymLinks i s one of the possible options - it allows webpages to use symbolic links to point to files located anywhere on the root (/) directory. Please note this is not the best configuration from the security point of view; however, it does demonstrate the functionality of the *Directory* tags. We will discuss the server security measures later in the Part.

AllowOverride directive governs the behavior of *.htaccess* files (more about them later). It tells whether the restrictions imposed by the *Options* can be overridden by specific settings inside the *.htaccess* files. The default behavior is set to *none* and should remain that way. This will prevent security breaches or nuisances due to misconfiguration.

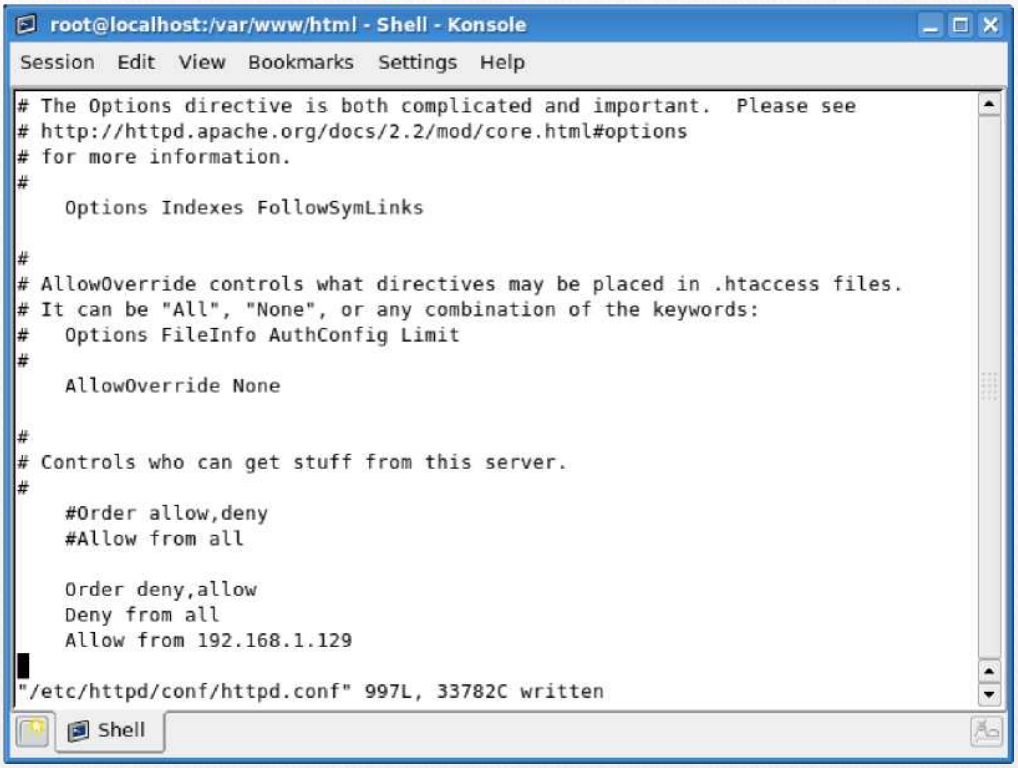
<Directory /> closes the block.

3.1.1 Order (allow, deny)

Allow and Deny directives govern the access to the directory declared (via the *Directory* tags). The *Order* directive specifies how the *allow* and *deny* directives are treated. The *Order* of *allow, deny* can be looked upon as default-allow or blacklist; only "bad" hosts or IPs are disallowed. The *Order* of *deny, allow* can be looked upon as default-deny or whitelist; only "good" hosts or IPs are allowed.

Possible declaration of allowed or denied clients can be via host name, domain name, IP address, partial IP address, and more. Here, we'll restrict access to the directory (or rather, the server) by denying access from all - and only permitting access from a single IP address, that of another machine on the LAN (in this case, 192.168.1.129).

Figure 20: Allow and Deny directives



Let's review the changes to the *httpd.conf* file:

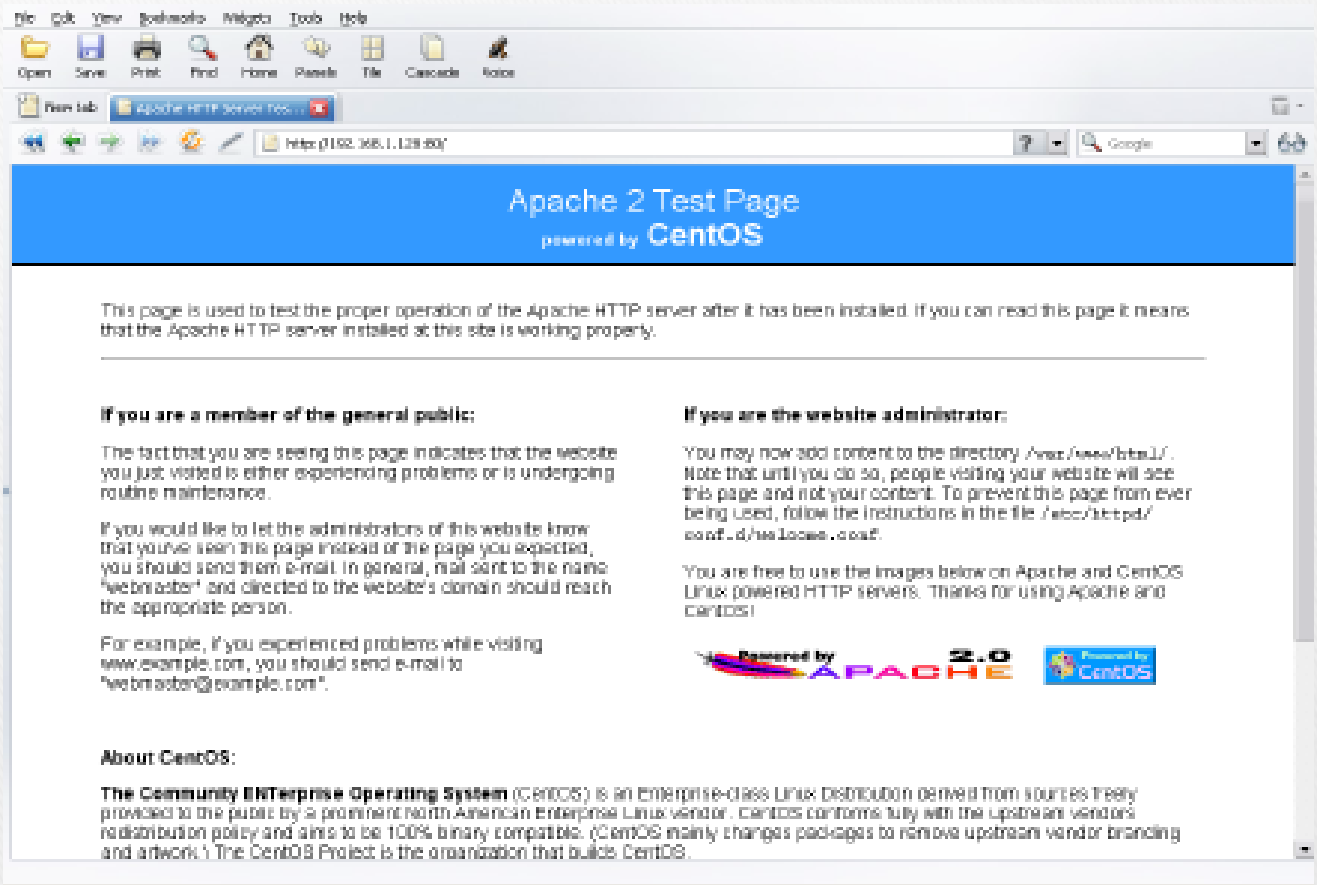
- I have commented out the original parameters, which allowed access from all hosts (or IPs).
- I have changed the order of *allow, deny* directives. Again, this is important, because the order defines the precedence of the rules. Thus, first, we'll deny everyone (this can be called default deny policy, so to speak) and then permit only specific hosts (or IPs). If the *Order* were reversed *(allow, deny* rather than *deny, allow*), no one would be able to access the server. This is critically important to remember when implementing *allow, deny* policies.

The changes will only take effect after the Web server is restarted or the configuration file reloaded. This can be achieved by running either service restart or reload:

```
service http restart|reload
```

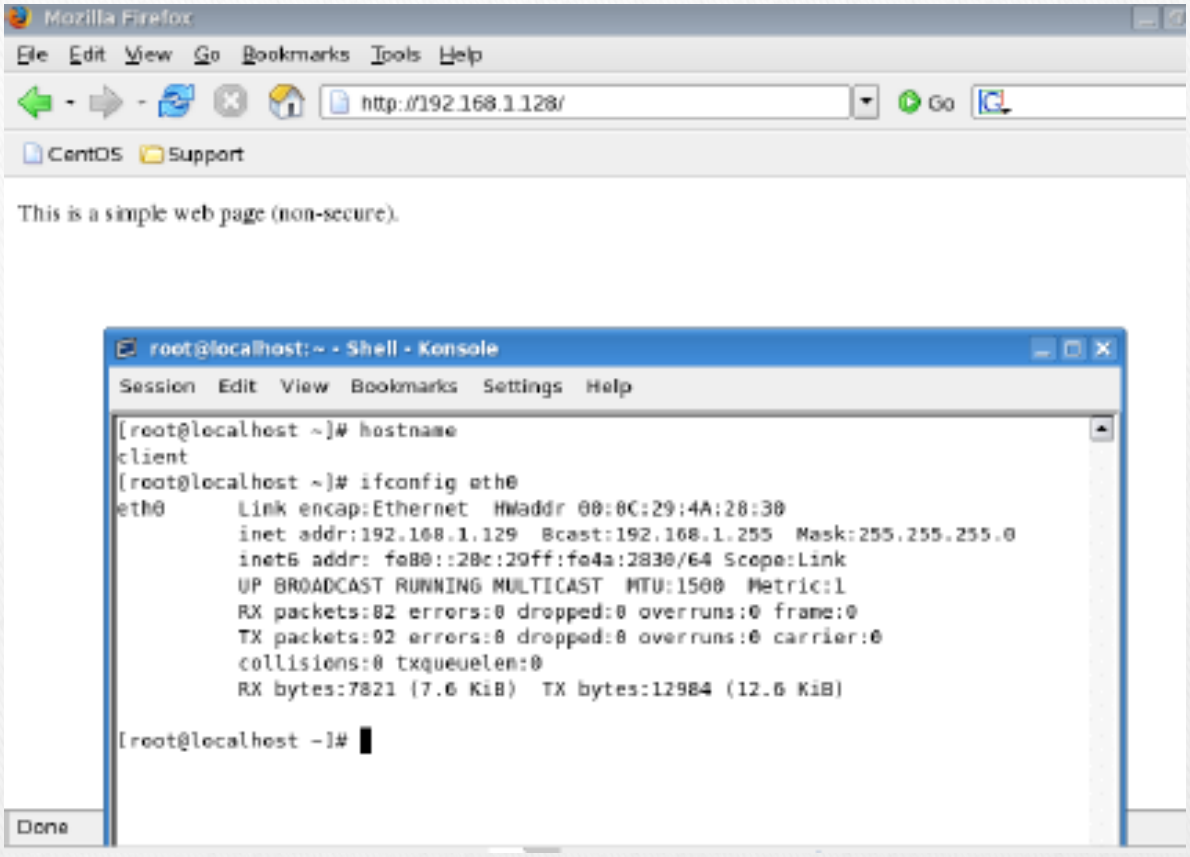
After *httpd* reads the new configuration file, the changes will take effect. Now, let's try to access the server from the Windows machine.

Figure 21: Accessing Web server from denied host



As you can see, we are denied access. But accessing from the CentOS client with the IP of 192.168.1.129 works fine.

Figure 22: Accessing Web server from allowed host



3.1.2 Indexes

Indexes directive tells the server whether to display the directory listing when asked. The behavior of this directive depends on another directive - the *DirectoryIndex.* The *DirectoryIndex* directive tells the server the name of the default page that it should serve when a user requests the listing of a directory.

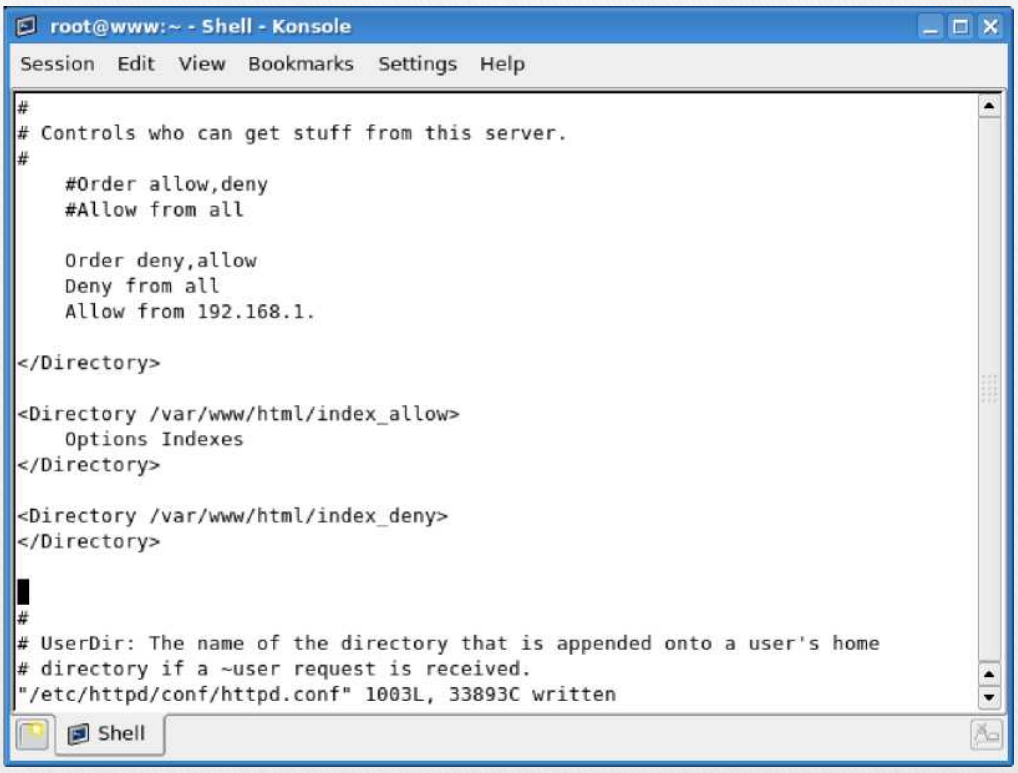
This is the typical everyday scenario. Users are trying to access webpages by simply typing their names, without typing the exact homepage (like *index.html, index.php* etc). Various file names specified under the *DirectoryIndex* are looked for and the first one found is presented to the user. If no file is found, the listing of the directory is then generated by the server.

This is something you may want to avoid, especially if there are files you do not wish your users to see. However, if the *Options Indexes* directives are used, then

directory listings will be generated. One solution is to place a dummy *index.html* file in every directory, but this is cumbersome. The more elegant approach is to disable the listing globally (remove Indexes from the *Options* directive under *DocumentRoot)* and then allow per-directory listing when you see fit.

The default configuration in the *httpd.conf* file specifies *Options Indexes* for the *Directory* tags of the default *DocumentRoot (/var/www/html).* We will change that. First, we will remove *Indexes* from the *Options* line for our *DocumentRoot.* Then, we will create two directories, called *index_allow* and *index_deny,* where only the first will have the *Options Indexes* specified. Both of these directories will contain some random files.

Figure 23: Indexes directive



This is the new configuration file. Save it, then restart *httpd.* Now, if we request the directory listing for each one from our clients, we'll get the following results:

Figure 24: index_allow example

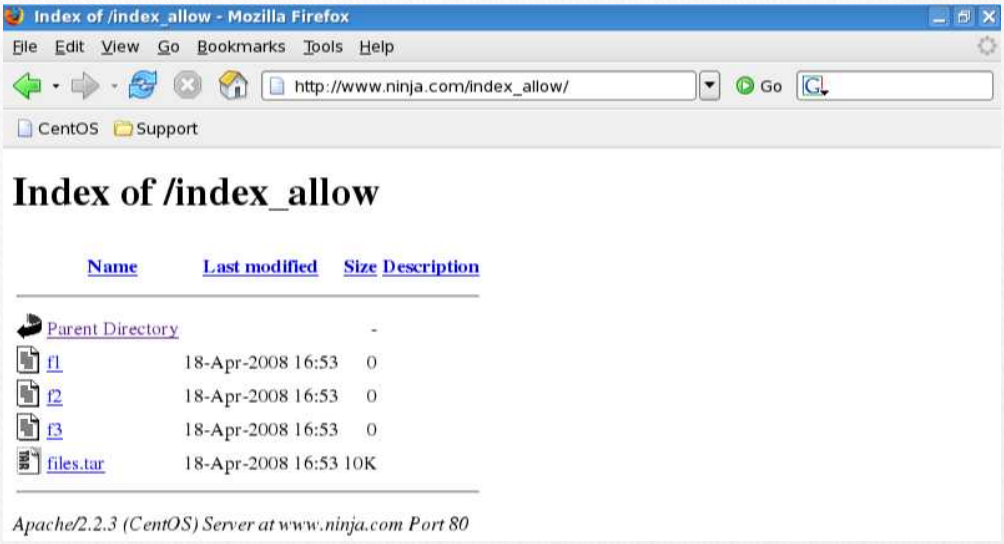
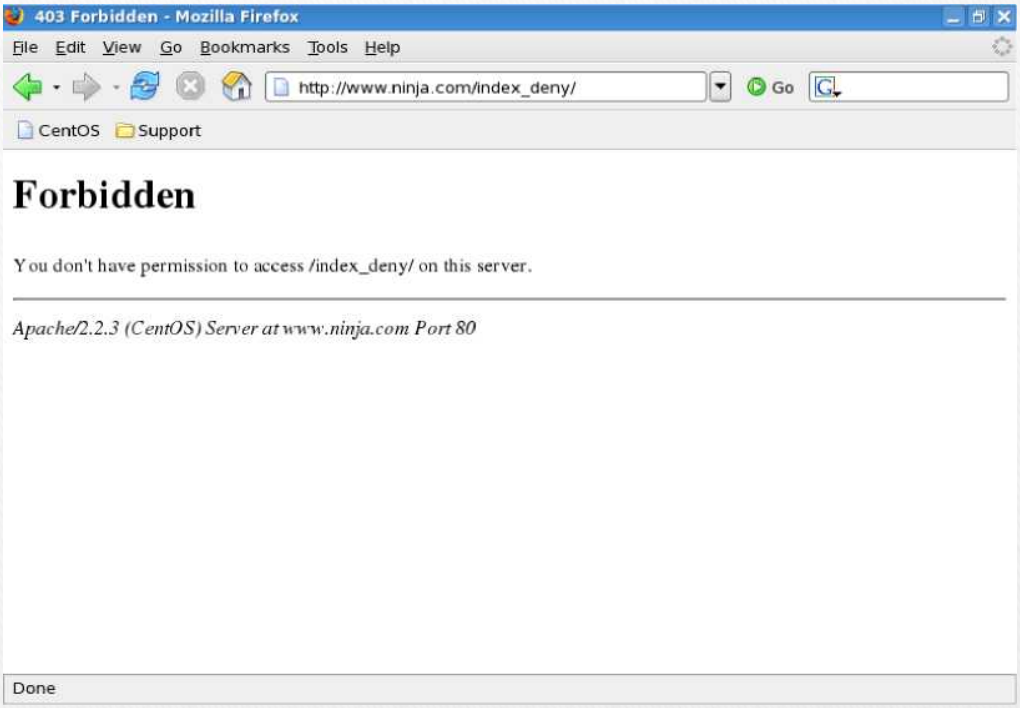


Figure 25: index_deny example



3.1.3 DirectoryMatch

Directives enclosed in the *Directory* tags will be indiscriminately applied to all sub-directories. If you require a more fine-tuned approach for several similar subdirectories, you will have to use the **DirectoryMatch** tags. The main difference is that the DirectoryMatch tags allow the use of regular expressions, allowing you to match several sub-directories inside a single rule. Again, for those familiar with HTML / CSS and the use of classes and ids, the idea is very much similar.

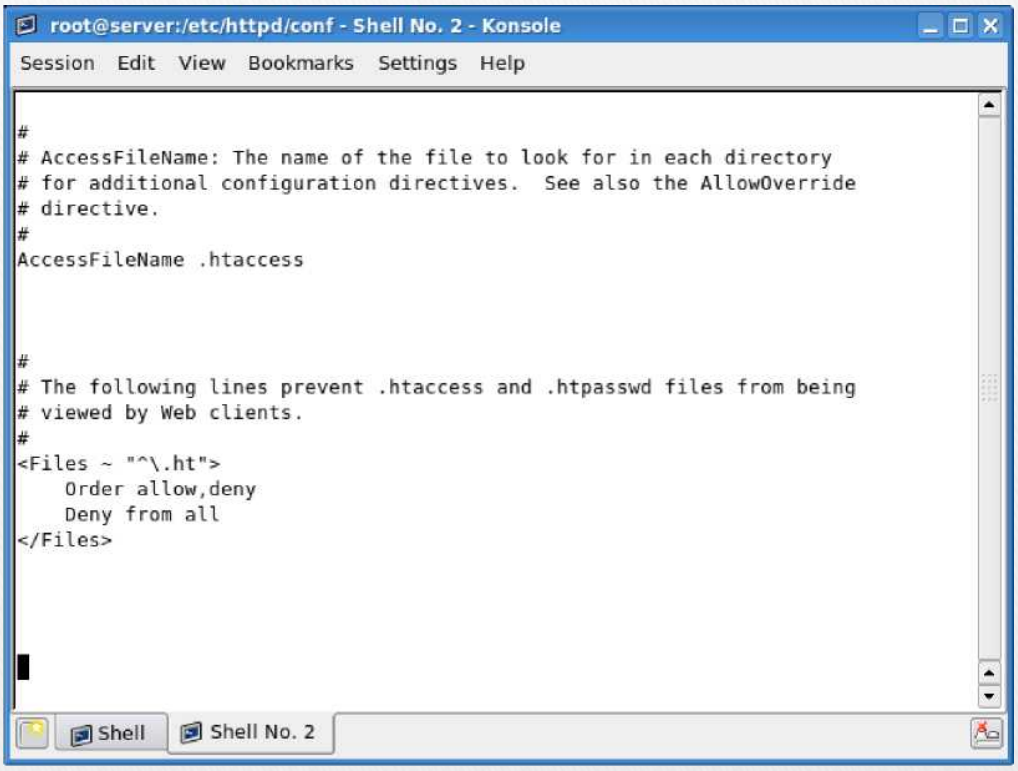
3.2 Files tags

Files tags are very similar to *Directory* tags. The major difference is that while *Directory* tags govern the scope of permissions (or restrictions) of the enclosed directives by directory name, *Files* tags do the same on the file name level. In

other words, *Files* tags can be used to configure the behavior of a single file - or a set of files that match a regular expression.

Here's an example, showing the restrictions applied to *.htaccess* and *.htpasswd* files, the files usually used in restricting access to certain directories (and/or files) by requiring users to authenticate before viewing the content:

Figure 26: Files tags example

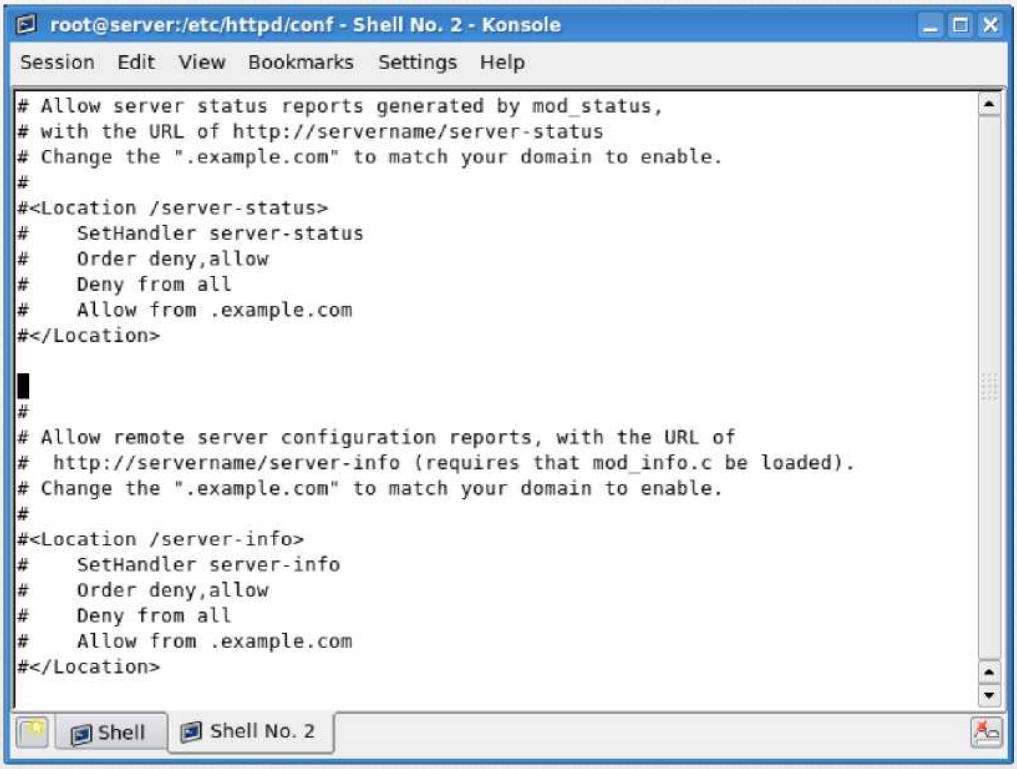


We will review this particular example later in this Part. In the above example, we've seen the use of regular expressions to allow multiple files to be covered by a single rule. However, a comparable directive, more suitable for handling multiple files and complex regular expressions is the **FilesMatch** directive.

3.3 Location tags

Again, **Location** tags are quite similar to the two mentioned above. The major difference is that *Location* tags are used to limit the scope of enclosed directives by URLs. In other words, the *Directory* and *Files* tags should be used to control content that resides on the system (like various files and images, within their sub-directories), while the *Location* tags should be used to control content that is located *outside* the system, like databases, for instance. Below, we can see a commented example included in the *httpd.conf* file. If enabled, this block would allow you to access your server statistics, but only if you connected from the server itself.

Figure 27: Location tags example



Here's an example (please disregard the actual URL):

Figure 28: Apache Server Status Location tags example

Again, for complex regular expressions, you should use **LocationMatch** directive.

3.4 Directory, Files and Location

Directory, Files and *Location* tags all perform a similar function: they categorize what restrictions are placed on content enclosed by each one. At first glance, there seems to be very little difference between them. However, just like the order of *allow* and *deny* directives is critical, so is the correct use of these tags.

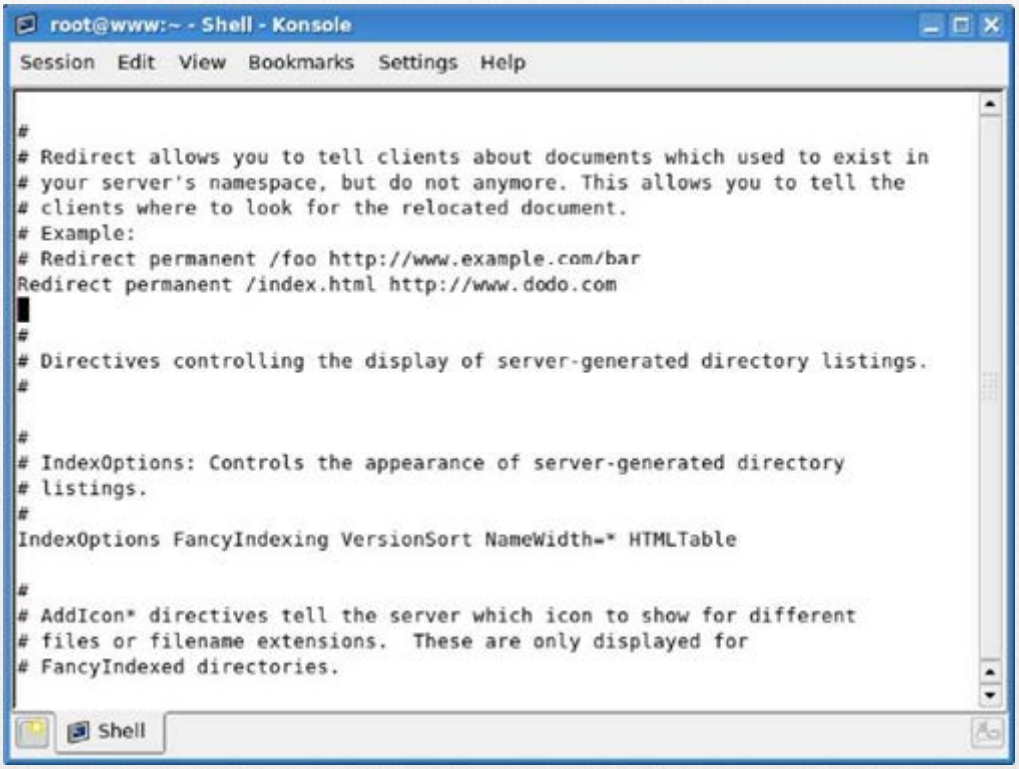
The configuration sections must be placed in a very particular order to make sure they behave as intended. The order of precedence of their execution by the server means that a misplaced section could compromise the security of the server - or not get executed at all.

For more details, please refer to Configuration Sections - Apache HTTP Server.

3.5 Redirect

The **Redirect** setting allows you to map an old webpage to a new URL. This could be the case if you changed domain, for example, or moved around a lot of files, renaming and deleting them. To demonstrate the directive, we'll map our server to point to dodo.com site. Now, it's hard to see the actual redirections, so please take my word for it.

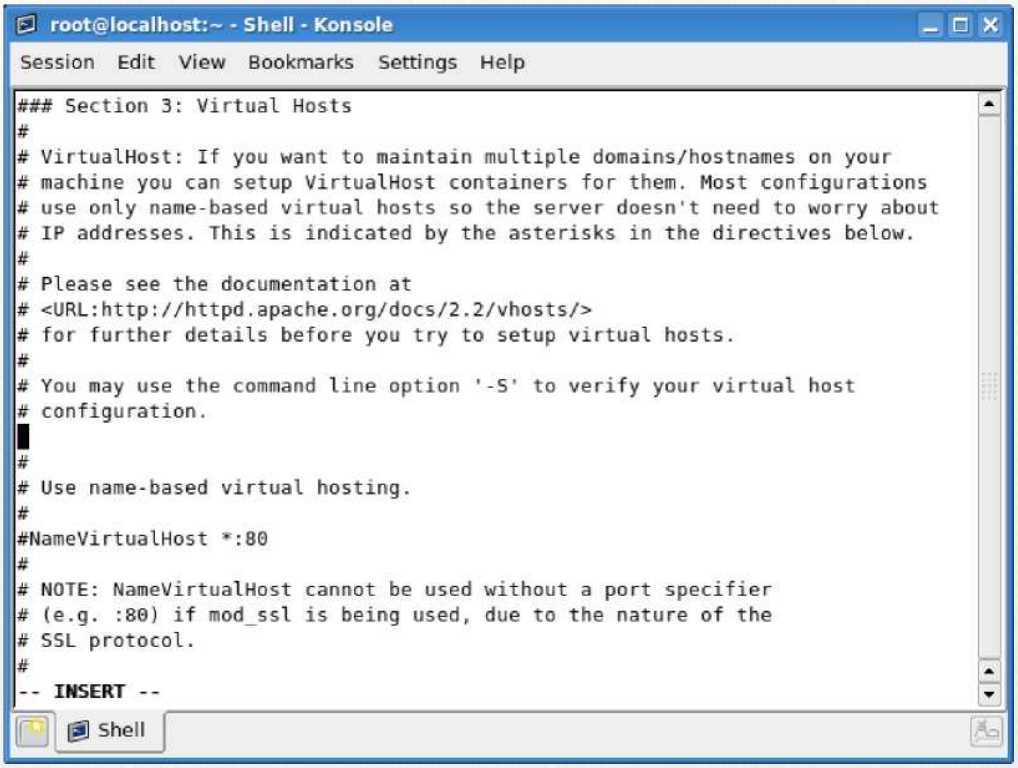
Figure 29: Redirect directive



3.6 Virtual Hosts

Virtual Hosts is an important, powerful feature that allows you to run several websites from a single computer. *Virtual Hosts* can be IP-based or named-based, offering a high level of customization (and flexibility).

Figure 30: Virtual Hosts example



Virtual Hosts can use almost any option normally used in the *httpd.conf* file. To make you better understand this, you can treat *Virtual Hosts* as individual customized *httpd.conf* files nested inside the main *httpd.conf* file. Let's review a sample *Virtual Host:*

```
<VirtualHost *:80>
        DocumentRoot /var/www/html/ninja-father
ServerName www.ninja-father.com # other directives
</VirtualHost>
```

What do we have here?

<VirtualHost *:80> declares the name or the IP address of the site (server) that should be served using the directives inside the *VirtualHost* block on port 80. If no port number is used, the default one specified under the *Listen* option is used. The default port is 80 (standard convention). Asterisk (*) can be replaced with any name (for example, *www.ninja.com)* or IP address (for example, 192.168.1.128), depending on your needs and requirements. Let see several simple examples:

- <*VirtualHost 192.168.1.128:80*> will apply the directives listed in the block below to all incoming connections aimed at 192.168.1.128 on port 80.
- <*VirtualHost 192.168.1.128*> will apply the directives listed in the block below to all incoming connections aimed at 192.168.1.128 on the default port (as specified in the *Listen* directive). If this port is 80, then this option is identical to the one above.
- <*VirtualHost planck.matter.com*> will apply the directives listed in the block below to all incoming connections aimed at *planck.matter.com* on the default port (which can be 80, 8080 or any other).
- <*VirtualHost ninja.com:8777*> will apply the directives listed in the block below to all incoming connections aimed at our site *ninja.com* on port 8777. This port must be specified under the *Listen* directive.

DocumentRoot /var/www/html/ninja-father declares the directory where you should place all files that you wish served when the *VirtualHost* is invoked (matching names or IPs and the port).

ServerName www.ninja-father.com is the name of the server. In other words, this is the address people will type in the web browser address name in order to get to your site. In order to successfully resolve this name to the IP address of the Web Server, we will need to use */etc/hosts* file like before or setup a DNS Server.

- **other directives** is just a comment specifying many other options can be used, including those we have not yet reviewed here. OK, now that we know what we're dealing with, let's create and test several scenarios.

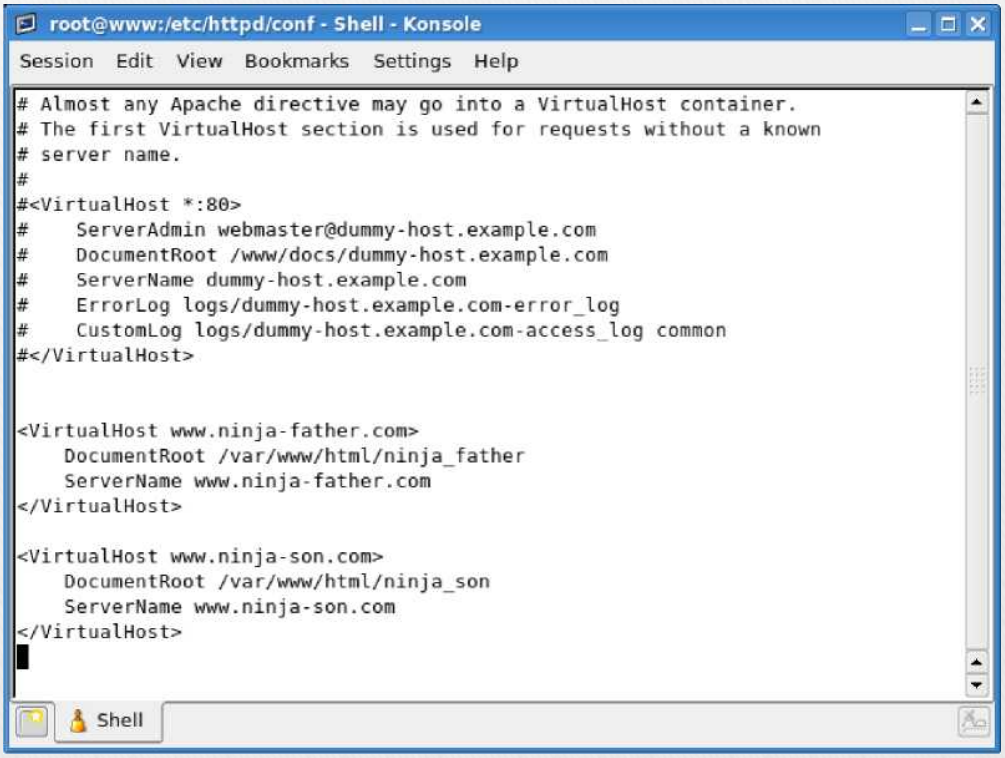
3.6.1 Single IP, two websites

This is one of the most common setups. We will create two websites - *www.ninja-father.com* and *www.ninja-son.com.* Both will reside on our server, which has the IP address 192.168.1.128. In order to make them both accessible to the world, we will create two *VirtualHost* blocks and declare their *DocumentRoot* and *ServerName* separately.

Here's what we need to do:

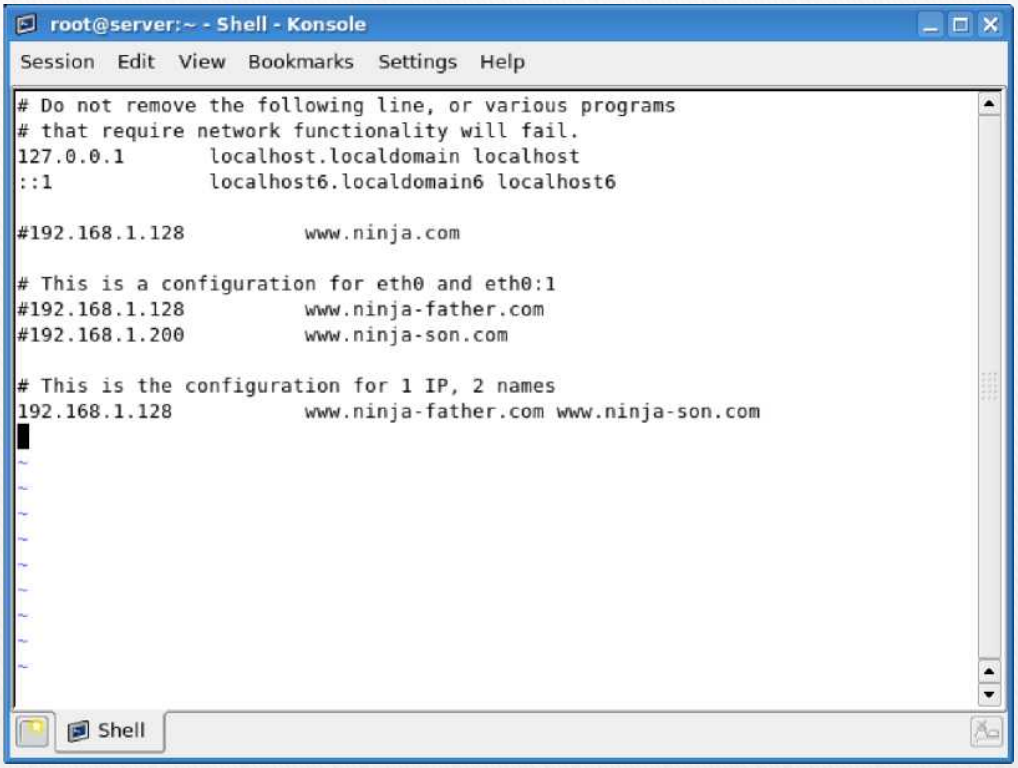
- Create directories inside */var/www/html* called *ninja-father* and *ninja-son.*
- Create simple *index. html* files for each.
- Edit *httpd.conf* and create our two *VirtualHost* blocks.

Figure 31: Single IP, two websites example



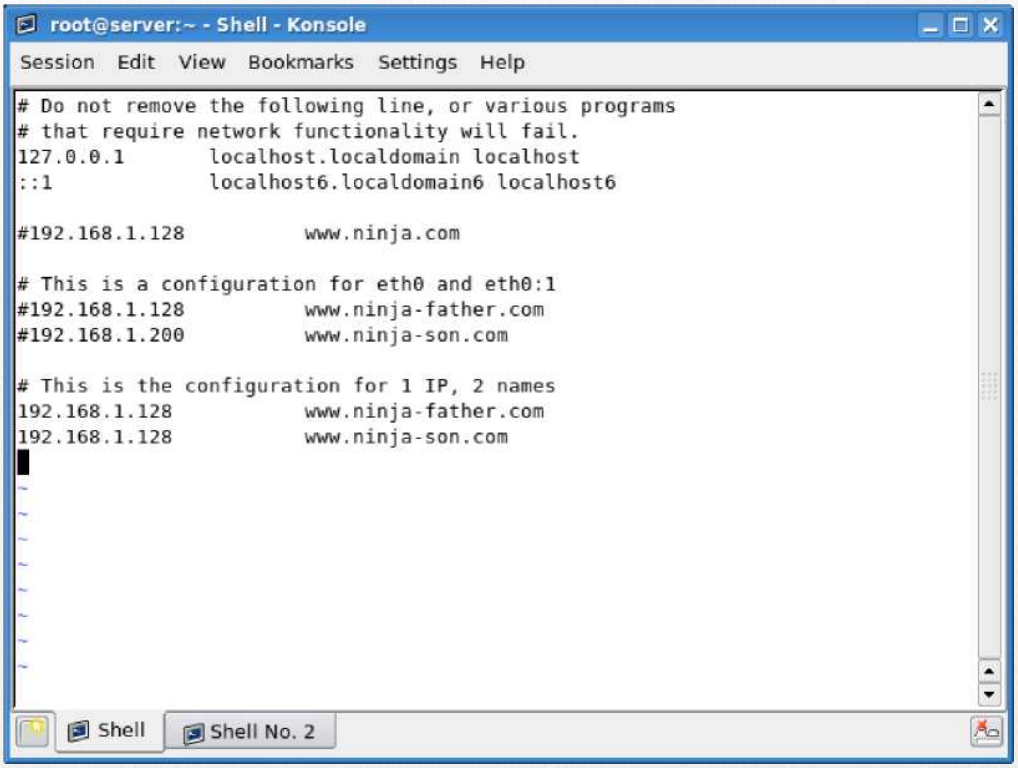
Now, for the sake of convenience, we will also use the */etc/hosts* file to allow name resolution to work. It is also imperative in our case, because using the IP address would always point to the first *VirtualHost* listed in the *httpd.conf* file.

Figure 32: Hosts file change matching Virtual Hosts



Please note that specifying an IP address in two different lines is **wrong**. The *hosts* file will always use only the first entry. You should list all names for a specific IP in a single line. For example, this is incorrect (although it would work in our case):

Figure 33: Wrong use of Hosts file entries

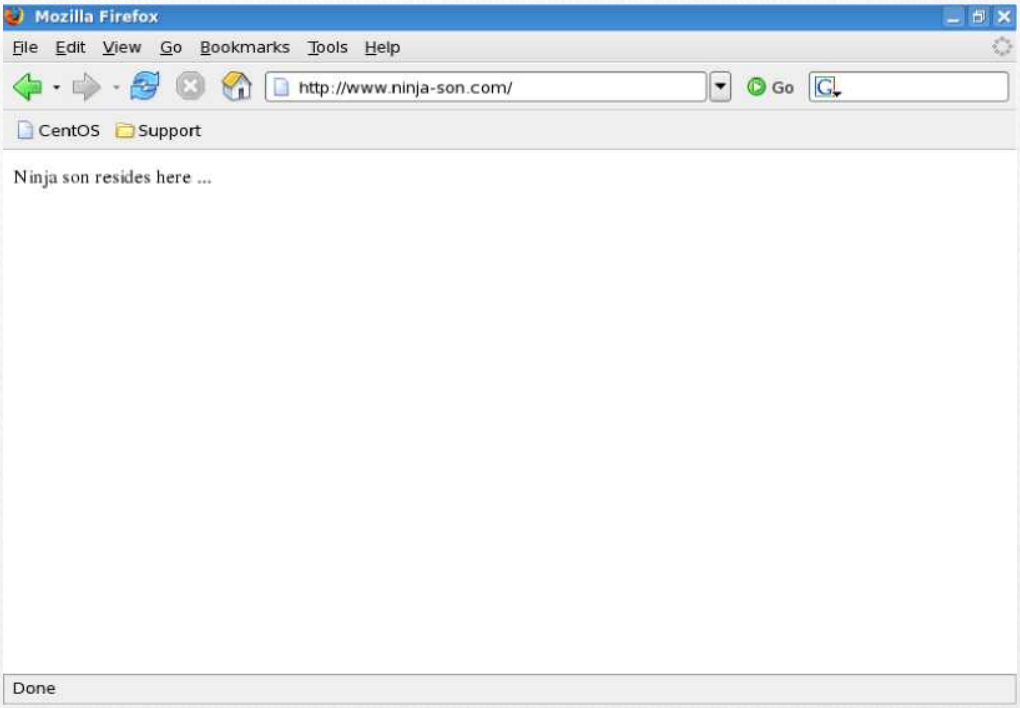


Don't mind the commented lines, they are used for other configurations: the first, our standard website; the second, for yet another *VirtualHost* scenario, which we will discuss soon. Now, we shall save the files (both *httpd.conf* and*/etc/hosts)* and restart *httpd.* Then, using Firefox, we will try to access each one.

Figure 34: www.ninja-father.com

Done

Figure 35: www.ninja-son.com



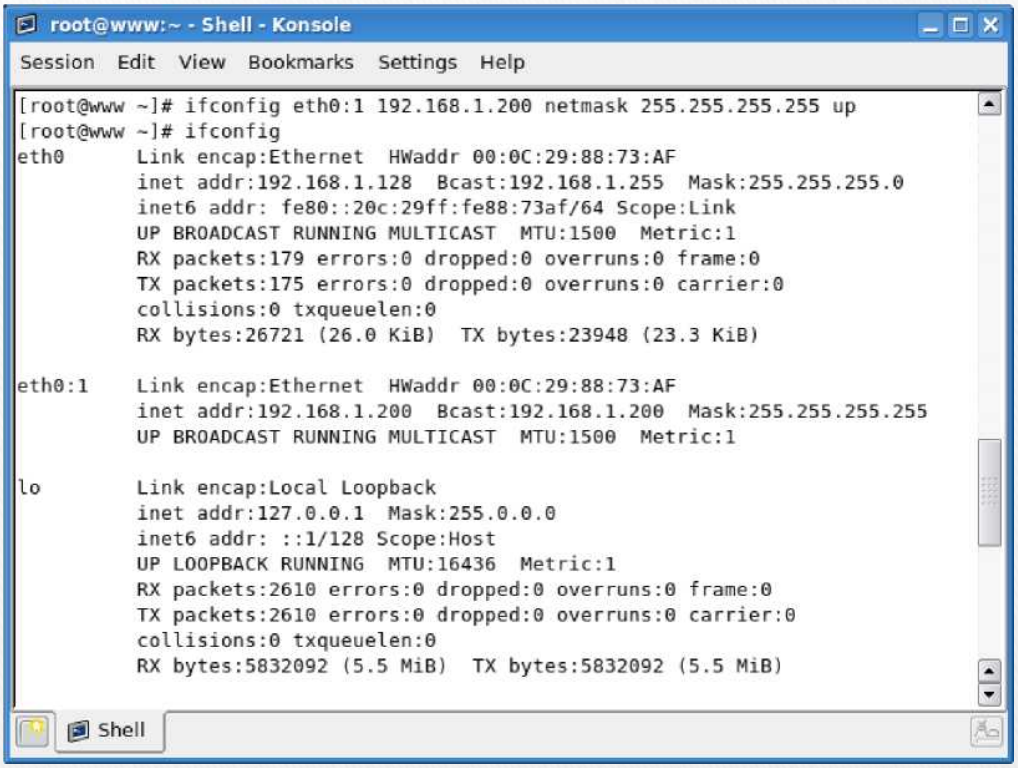
It works like magic. Best of all, the user has no idea that these two sites reside on the same machine.

3.6.2 Two IPs, two websites

This is another common scenario. You can assign a different IP to each website, avoid possible resolution mixups and simplifying your setup. However, this requires that you either use more than a single network adapter or create virtual adapters. If you have, let's say 14 websites, having 14 physical network devices plugged into your machines is not the best idea. Using virtual adapters is the most sensible choice here.

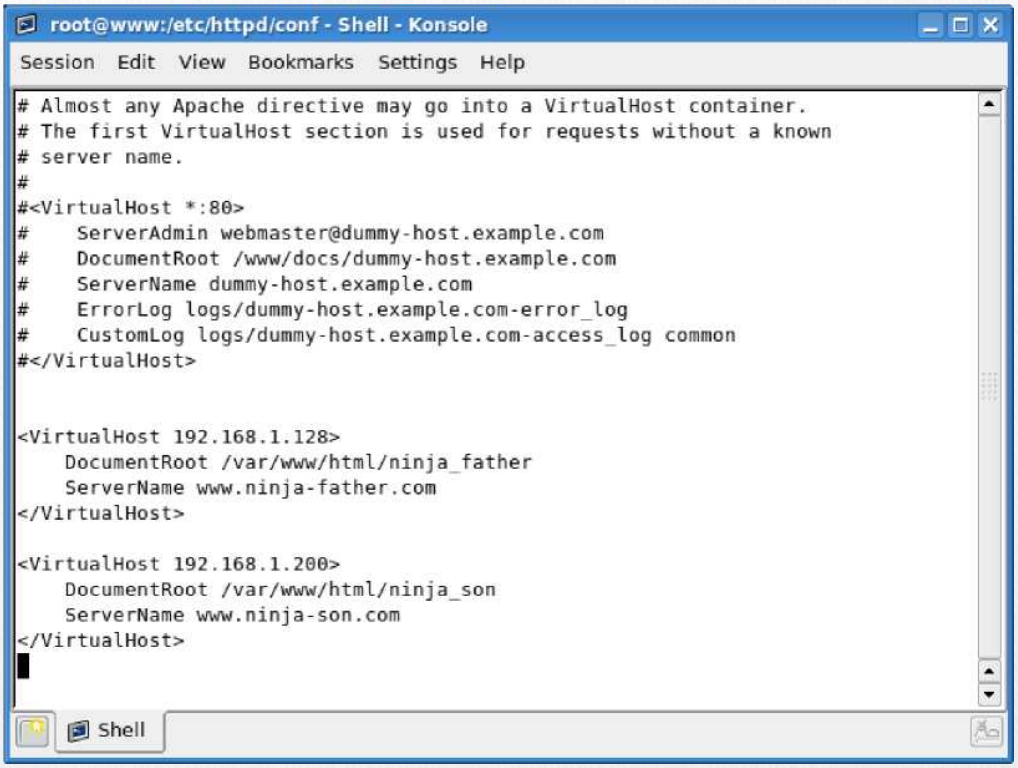
We already have our two websites ready. We just need to create a virtual network card and then change the *httpd.conf* file to reflect the changes. First, we will create a virtual adapter (eth0:1) with the IP address of 192.168.1.200.

Figure 36: Virtual network adapter



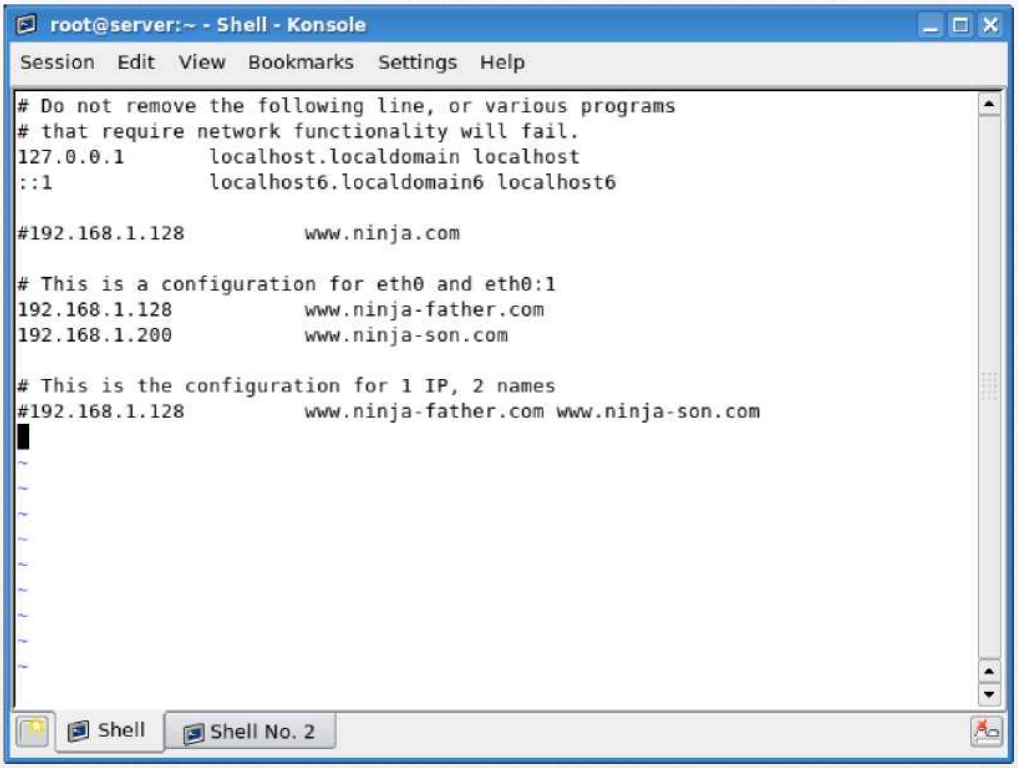
Then, we'll edit the *httpd.conf* file:

Figure 37: Two IPs, two websites example



Lastly, we'll edit the */etc/hosts* file:

Figure 38: Hosts file change



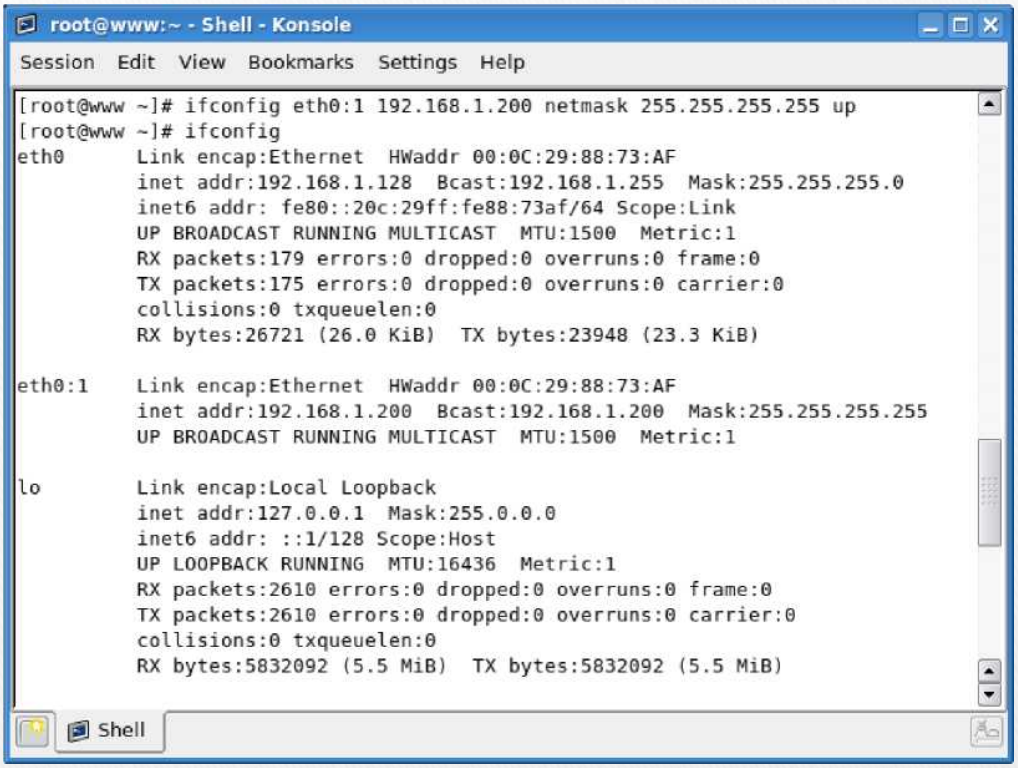
After restarting the server, we'll be able to get to our two sites easily. Again, the change is completely transparent to the user.

Two IPs, two websites

This is another common scenario. You can assign a different IP to each website, avoid possible resolution mixups and simplifying your setup. However, this requires that you either use more than a single network adapter or create virtual adapters. If you have, let's say 14 websites, having 14 physical network devices plugged into your machines is not the best idea. Using virtual adapters is the most sensible choice here.

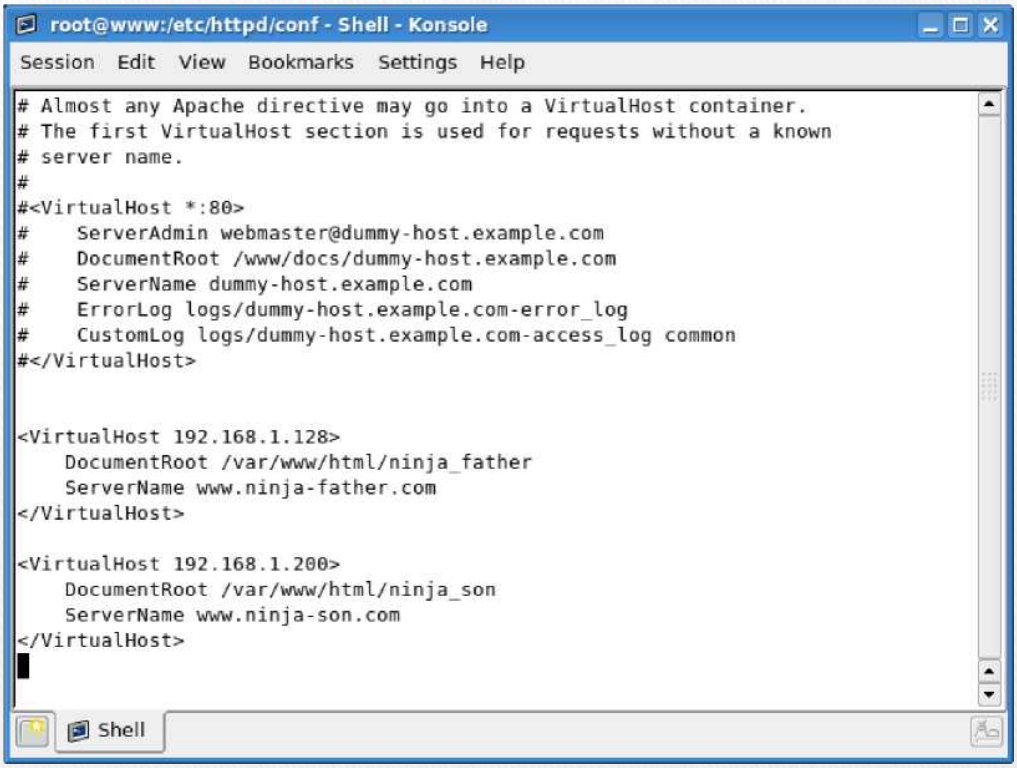
We already have our two websites ready. We just need to create a virtual network card and then change the *httpd.conf* file to reflect the changes. First, we will create a virtual adapter (eth0:1) with the IP address of 192.168.1.200.

Figure 36: Virtual network adapter



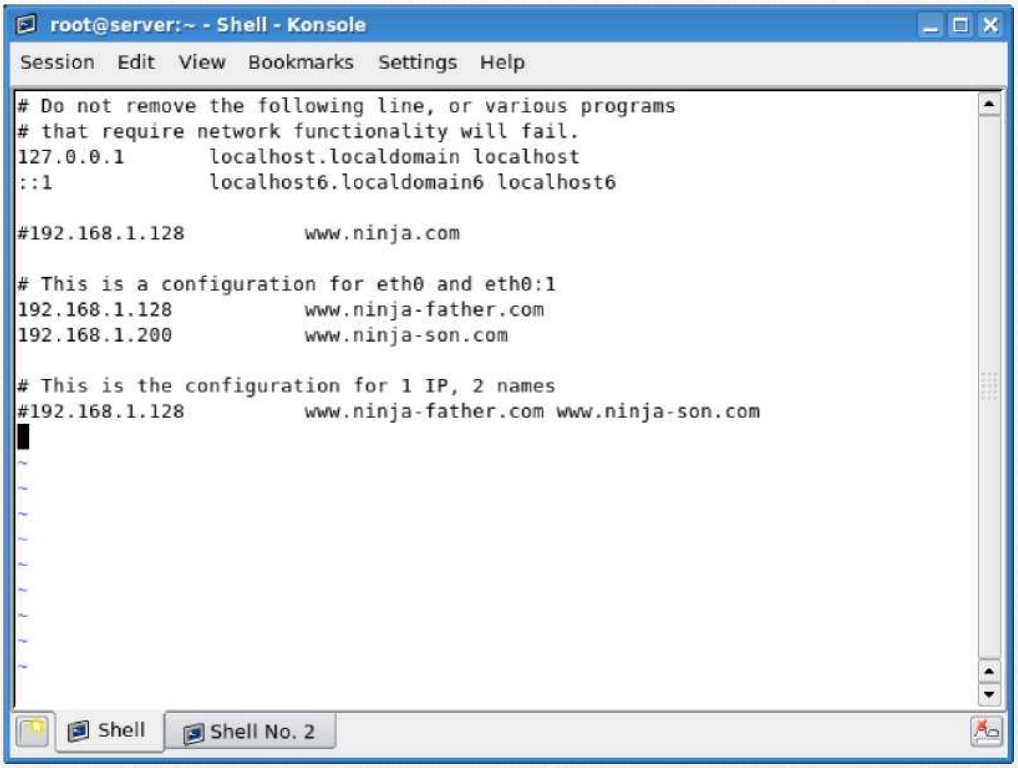
Then, we'll edit the *httpd.conf* file:

Figure 37: Two IPs, two websites example



Lastly, we'll edit the */etc/hosts* file:

Figure 38: Hosts file change



After restarting the server, we'll be able to get to our two sites easily. Again, the change is completely transparent to the user.

3.6.1 Other scenarios

Basically, the above two scenarios cover pretty much everything. Once you get the hang of *VirtualHost* setting, creating any which setup becomes a simple matter. Nevertheless, for the sake of clarity, I will demonstrate several more typical scenarios.

Different content for intranet and Internet: In practice, this scenario is very similar to having 2 different IPs serving two different websites, except that you will use one website but serve different parts of it to different customers.

Let's assume you wish to achieve the following: Allow users on the local network access to all content, but deny some to users on the Internet; Allow users on the local network to list directory index, but deny this feature to the Internet users; Display a different home page to local users and external customers; Allow certain custom scripts to be available only to external customers; Here's what a sample configuration in *httpd.conf* file would look like:

```
AddHandler cgi -script .cgi

NameVirtualHost 172.16.1.1:80
<VirtualHost 172.16.1.1:80>
DocumentRoot /www/intranet ServerName
www.our-company.com <Directory
/www/intranet>
      Option Indexes FollowSymLinks
   </Directory>
</VirtualHost>

NameVirtualHost 211.211.211.211:80
<VirtualHost 211.211.211.211:80>
DocumentRoot /www/web ServerName
www.our-company.com <Directory
/www/web>
      Options +ExecCGI FollowSymLinks
   </Directory>
</VirtualHost>
```

This examples introduces a number of concepts, so let's briefly review them:

NameVirtualHost allows you to map named-based incoming connections to specific IP addresses. You might ask yourselves why you need this, when we have seen perfectly good examples before, without this feature. Well, the answer is: if somehow a named-based request gets "lost" (due to DNS configuration, firewall rules or similar), it might not match any of the *VirtualHost* blocks. In that case, the default settings configured in *httpd.conf* will be applied to this request, which could be contrary to your needs. Using *NameVirtualHost* forces

all incoming connections to a certain IP address to point to a certain *VirtualHost* block. This request will also never fall back to the main configuration, allowing you a complete modularity in your setup. Thus, in our example, all requests to the internal IP address will go the *VirtualHost* with this IP address declared. We can also see that the users will be able to view directory listings and follow symbolic links.

+ExecCGI directive is listed under *Options* in the second block, which refers to the Internet customers. All incoming connections on the external IP will go to the *VirtualHost* with this IP declared. We can see the users won't be able to demand directory listings, but they will be able to follow symbolic links - and execute *.cgi* scripts located in this directory.

The *ExecCGI* directive tells the server to allow server-side scripting in the specified directory. The plus (+) signs signifies this *Option* is used in addition to all those *Options* already specified for the *root* directory. Similarly, the minus (-) sign can remove some of the privileges, compared to the Options already specified for the *root* directory. However, alone, this directive is insufficient to allow scripting in this directory.

AddHandler cgi-script.cgi allows scripts in non-default directories to be executed, by using the *+ExecCGI* option, as we've done before. In order to enable *.cgi* scripts to work outside the default script directory, a directive must be added to the *httpd.conf* configuration file.

The Apache Web server has many other options and features. You are welcome to try them all. For more information, please refer to:

Apache HTTP Server Version Documentation

Different websites on different ports: We've already discussed this before. Let's say you have a single IP address with multiple websites served. Using the *hosts* file or DNS resolution is a possibility, but this might not always work. Configuring the Web server to listen on several ports for incoming connections and then using *NameVirtualHost* feature to force the connections to specific *VirtualHost* blocks will force the server to behave as you desire. Here's an example:

```
Listen 192.168.1.128:80 Listen
192.168.1.128:9021

NameVirtualHost 192.168.1.128:80
<VirtualHost 192.168.1.128:80>
DocumentRoot /www/white-socks
ServerName www.white-socks.com
</VirtualHost>

NameVirtualHost 192.168.1.128:9021
<VirtualHost 192.168.1.128:9021>
DocumentRoot /www/black-socks
ServerName www.black-socks.com
</VirtualHost>
```

For more examples, please refer to VirtualHost Examples - Apache HTTP Server.

3.7 Modules

Modules are extensions that enhance the basic functionality of the Web server. The modules reflect the growth of the Web and the inclusion of dynamic content into the web pages. The static HTML can provide only so much functionality. In fact, many of the options we have seen and used above are provided by different modules. For example, the *Order* directive is provided by the *mod_access* module.

3.7.1 Module types

There are two types of modules:

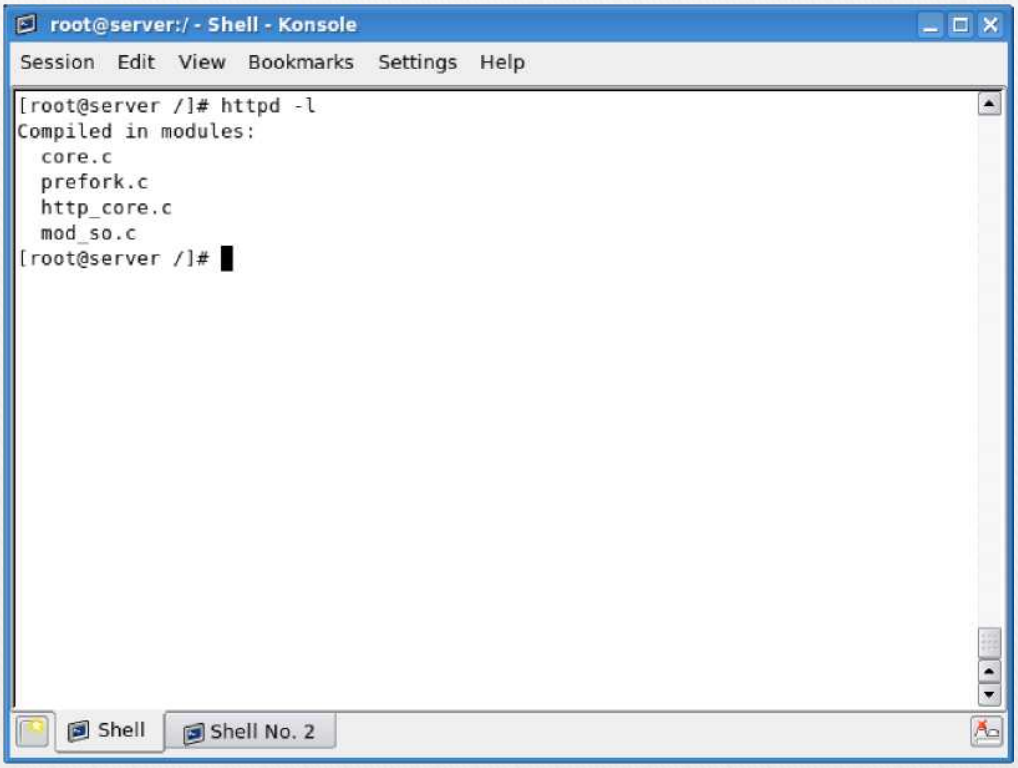
- Built-in modules, which are compiled into Apache and will load with the server any time it is started. Their functionality cannot be removed without recompiling the package. These modules are also known as *static.*
- Loadable modules, which can be loaded on and off as required. These are the *shared* modules.

3.7.2 View installed modules

You can always list the modules currently used by the server. The command below will display only the modules compiled into Apache.

```
httpd -l
```

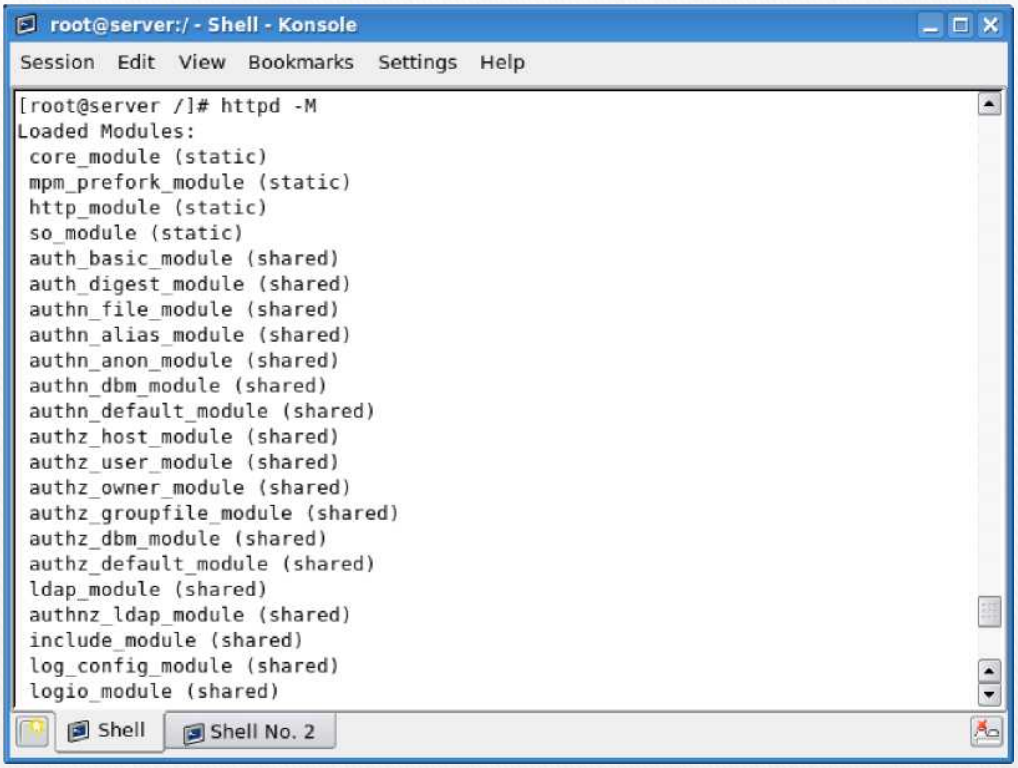
Figure 39: Listing compiled modules



The following command will list all modules, both *static* and *shared*:

```
httpd -M
```

Figure 40: Listing static and shared modules

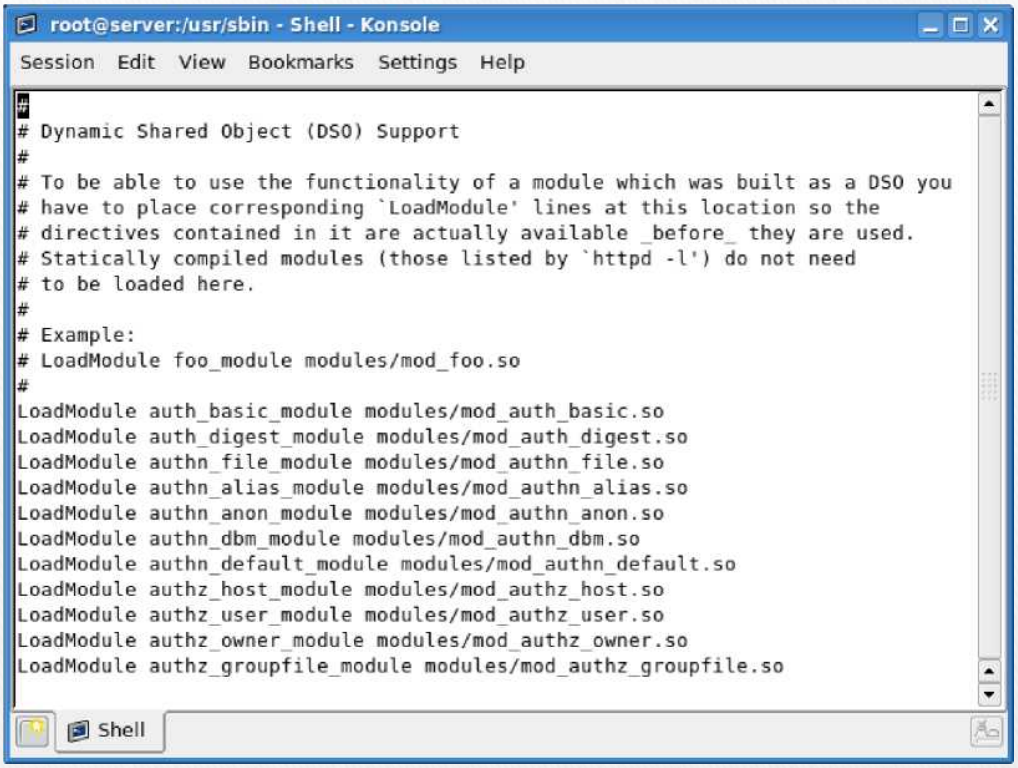


There is a wide range of modules available. We will review a number of more common ones. Please note that the list below is only partial and just briefly introduces the range of available modules.

3.7.3 LoadModule

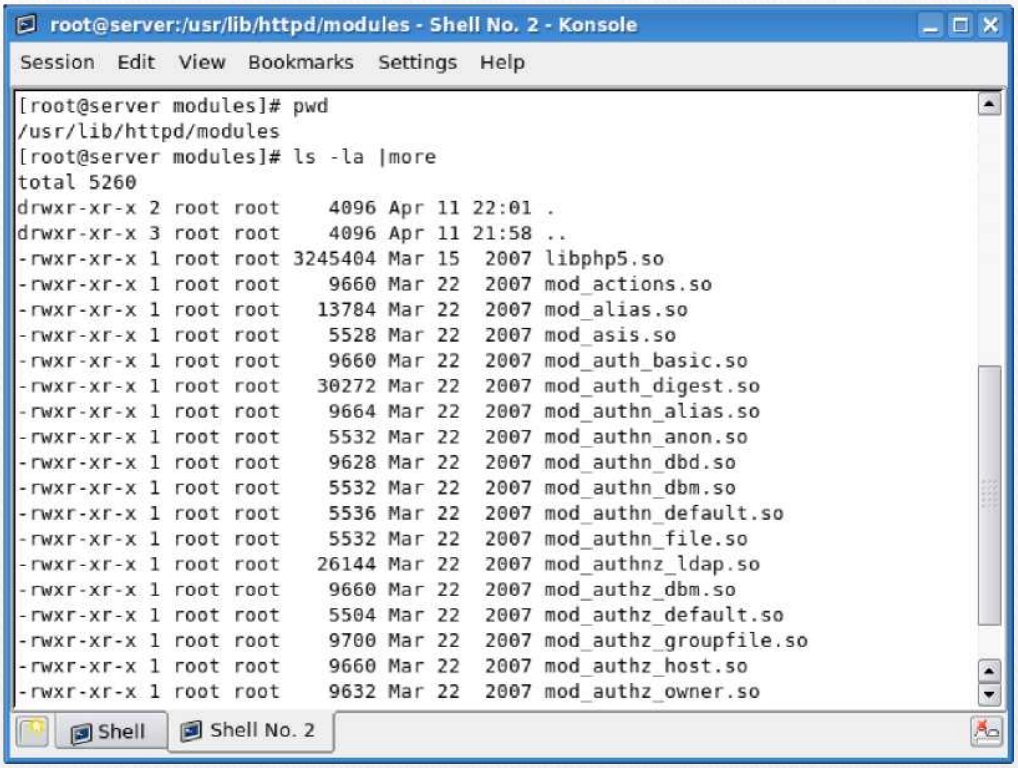
Shared modules are called by the Web server using the *LoadModule* directive in the *httpd.conf* file. If you do not wish to use a certain module, simply comment its line. However, you must remember this will remove the functionality that the module provides.

Figure 41: LoadModule directive



These modules are referenced by a symbolic link in the */etc/httpd/* directory, pointing to */usr/lib/httpd/modules.*

Figure 42: Contents of /usr/lib/httpd/modules



Let us go over some of the more interesting modules, just a sampling.

mod_access This module provides access control based on client host name, IP address, or other characteristics of the client request.

mod_dir This modules provides interface for redirects and serving directory indexes. We have reviewed quite a bit of its functionality in the previous sections.

mod_perl This module allows dynamic content produced by Perl scripts to be served to incoming requests without using the Perl interpreter every time, reducing overhead and system load. This is done by embedding a Perl interpreter into the Apache server. The module can also emulate a CGI environment, allowing

the reuse of Common Gateway Interface (CGI) scripts without any changes to the setup.

mod_python This allows integration of the Python programming language into the Apache server. It is intended to replace CGI as a method of executing Python scripts on a web server. It offers much faster execution and allows data to be maintained over multiple sessions.

mod_ssl This module provides an interface to the OpenSSL library, allowing the use of Secure Socket Layer (SSL) and Transport Layer Security (TSL) secure communication protocols. This allows you to run a Web server that will run encrypted sessions with clients, allowing a safe exchange of potentially sensitive data. We will discuss this module again when we setup a secure Web server (7.12).

For a detailed list of available modules and their functionality, please refer to: Apache HTTP Server Module Index.

4 .htaccess

.htaccess stands for hypertext access. This is the default name of the Apache directory-level configuration file. This file can be used to create security restrictions for particular directories. One of the most common uses is to require user authentication in order to serve certain web pages. Before we setup *.htaccess,* there are some things you should remember:

- *.htaccess* is not a replacement for a carefully laid out security plan. You should use the *httpd.conf* file to place restrictions on your server. Only then should you use *.htaccess,* to further restrict the already *allowed* users.
- Do not ever use *.htaccess* to handle secure or privileged content, like user data.
- *.htaccess* file is loaded every time a webpage is requested, incurring a performance loss.
- Using this file grants individual users an ability to make security modifications to your site, creating possible risks if not properly configured.

On the other hand, using *.htaccess* is useful if you run a multi-user hosting plan. These users do not have *root* access to the main configuration file and their only way of "shaping" traffic is by using the *.htaccess* file. In general, the use of the *.htaccess* file should be limited to non-root users only.

Before we can setup access-protected pages, we need to briefly overview the layout and syntax of the *.htaccess* files. Let's examine what a typical *.htaccess* file looks like. Then, we will combine it with our web content.

```
AuthType Basic
AuthName  “Restricted web page”
AuthUserFile
 “/etc/httpd/conf/.htpasswd require
valid-user
```

AuthType Basic defines the type of authentication. *Basic* means there is no encryption and the password hash is sent as clear text. This is one of the major reasons why *.htaccess* cannot be considered for protection of confidential user data.

"Restricted web page" is a window title string. When someone tries to access an .htaccess-protected page, a username & password window will pop in the web browser. This window will bear a title - this is the *AuthName.* It can be anything you like.

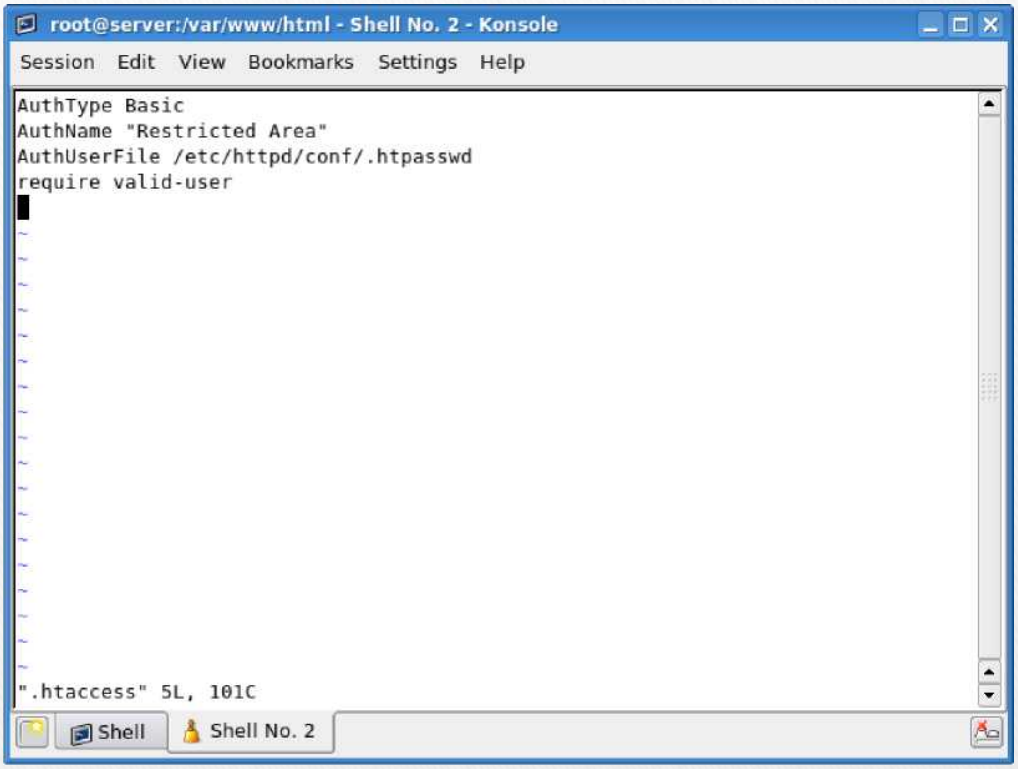
AuthUserFile /etc/httpd/conf/.htpasswd defines the path to a file where user credentials are stored. This file does not exist, but we will create it soon.

require valid-user indicates only successful authentication attempts will result in the loading of the page. Now that we know what we're about, we will:

- Create an *.htaccess* file similar to the one above.
- Create the *.htpasswd* file containing usernames & password necessary for the authentication.
- Place . *htaccess* in the directory we wish users to validate before accessing the content.
- Tell *httpd* to allow user authentication via *.htaccess* files.
- Restart the server.
- Test the results.

4.1 Create .htaccess file

Figure 43: Creating .htaccess file



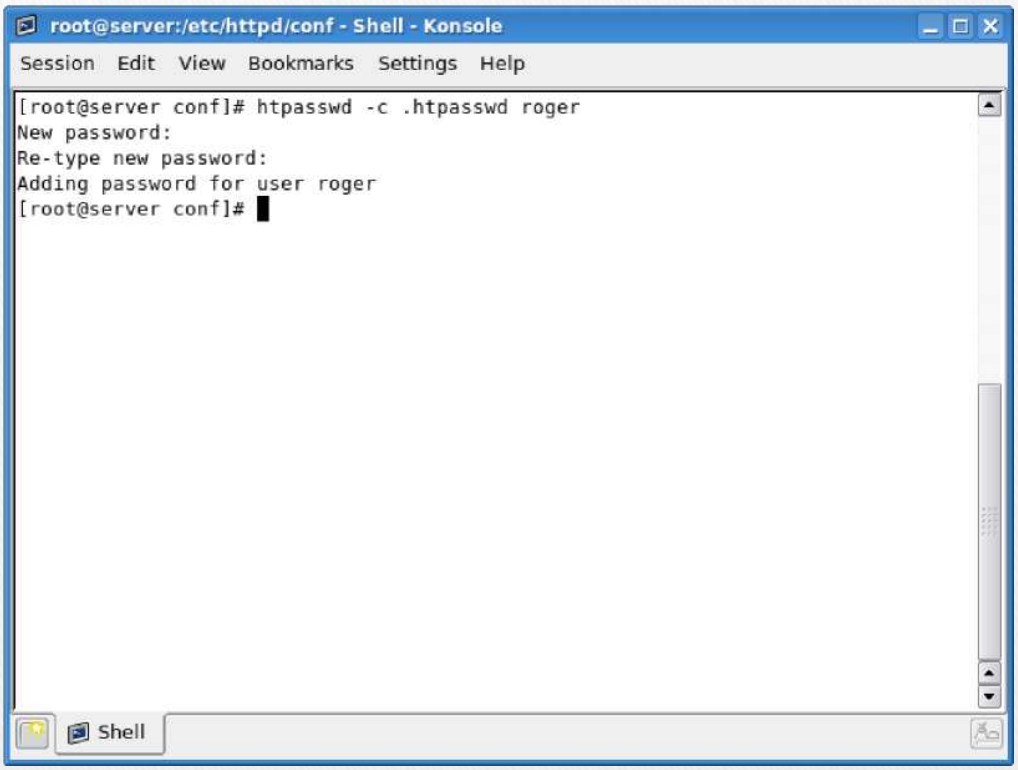
4.2 Create .htpasswd file

First, we will access the directory where we intend to place the file */etc/httpd/-conf.* It can be any directory, but it must be outside the *DocumentRoot,* so it so not viewable by your clients. Make sure **only** root can modify the *.htpasswd* file! It should have permissions set to 0644. Users and passwords are added to the file by running the *htpasswd* command.

```
htpasswd -c .htpasswd username
```

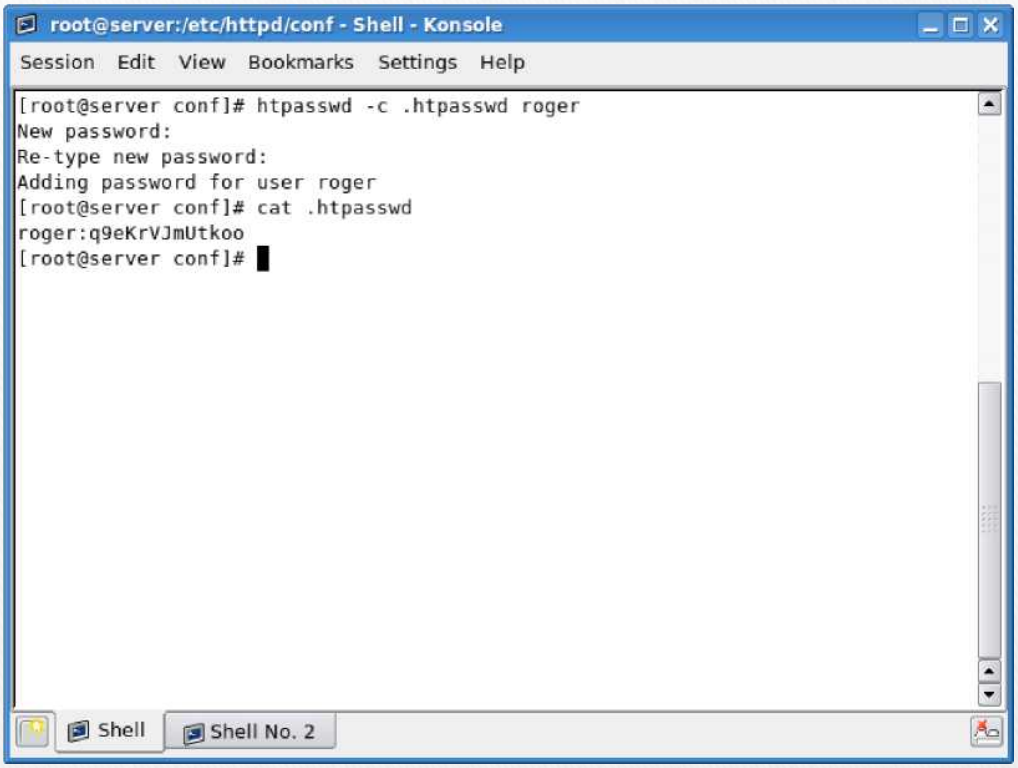
The name of the authentication file can be anything. You may consider changing it to something else.

Figure 44: Creating .htpasswd file



After you have finished adding the usernames (there can be one or more), you can see the contents of the *.htpasswd* file. The passwords are encrypted.

Figure 45: Contents of .htpasswd file



4.3 Copy .htaccess to restricted directory

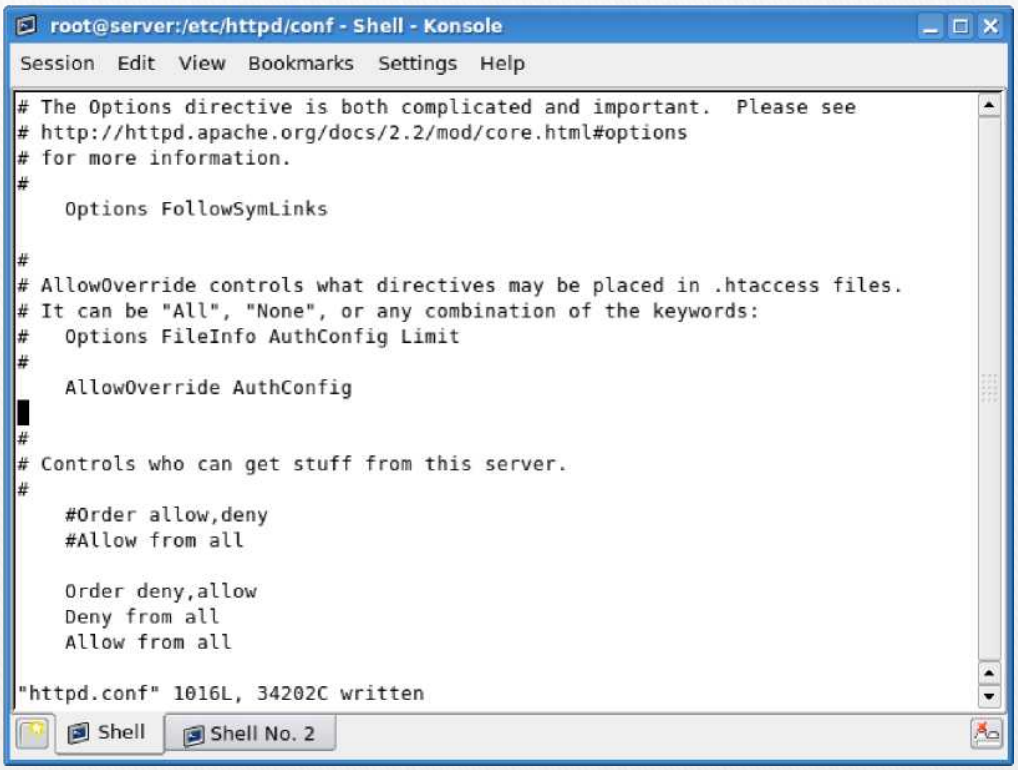
We will place the *.htaccess* file in our *DocumentRoot.* To make things interesting, we will also change the site and the homepage somewhat. Instead of *ninja.com,* we will serve *ourserver.com.* Again, no relation to any real site bearing this name.

4.4 Configure httpd.conf to allow authentication via .htaccess

By default, *.htaccess* files are given no control whatsoever. This is accomplished by the *AllowOverride* directive. This directive specifies what the *.htaccess* files can do - in addition and contrary to main configuration settings. Please note that this could pose a security risk. Badly configured *.htaccess* files can compromise

the security of your system. We will allow *.htaccess* to authenticate users. We will replace the original *AllowOverride none* to *AllowOverride AuthConfig.*

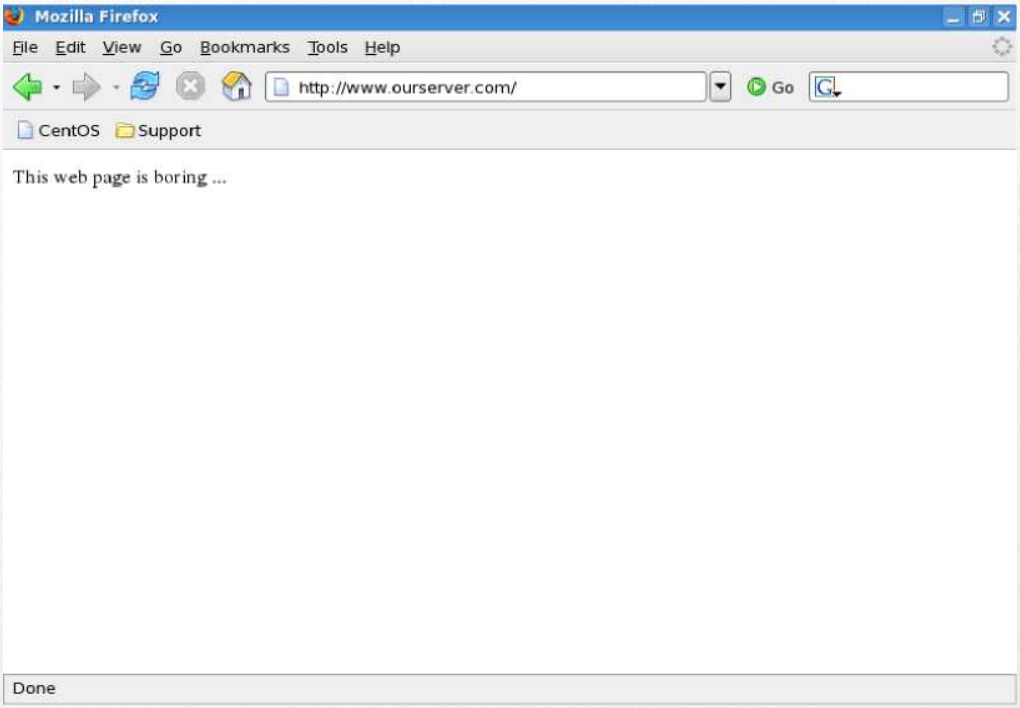
Figure 46: AllowOverride directive



4.5 Test setup

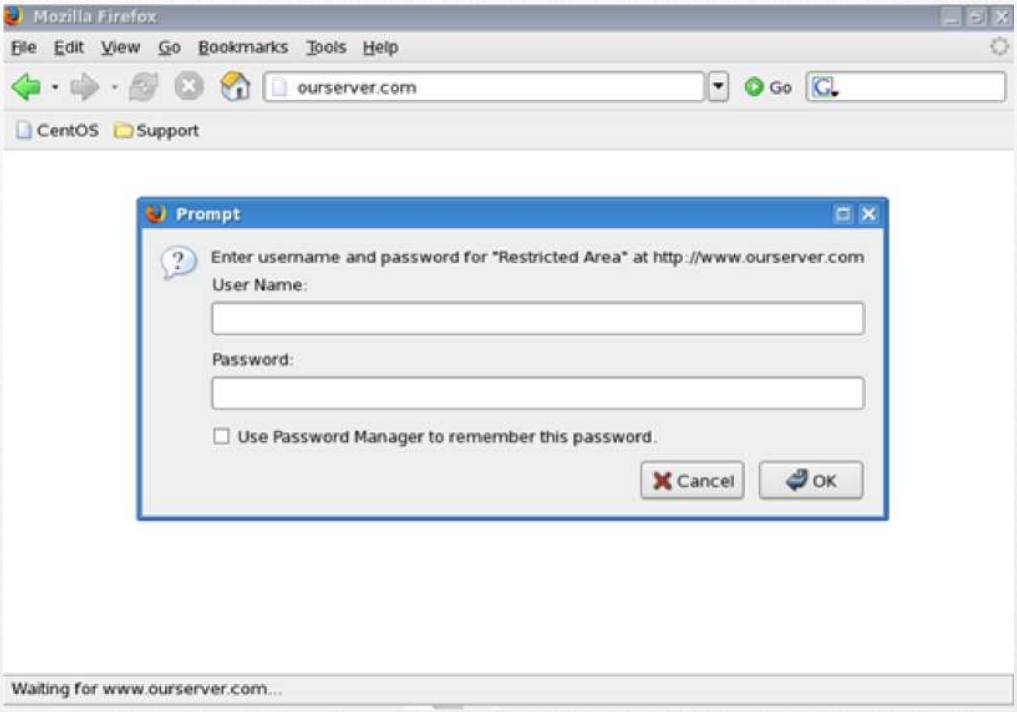
Restart the server and test. This is the webpage, seen without any restrictions.

Figure 47: Testing .htaccess, no file in place



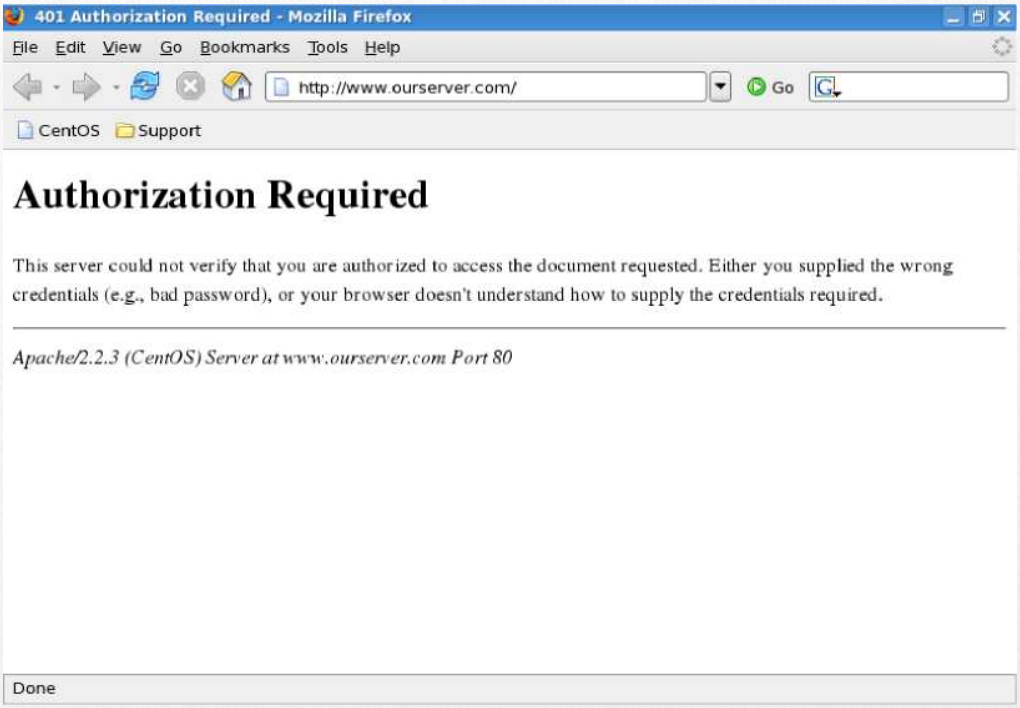
Now, after restarting the server, we will be asked for authentication credentials.

Figure 48: Testing .htaccess, file in place



If we succeed, we will reach the webpage, like before. If we enter the wrong username & password - or none, we will be rejected. You can customize the "reject" page, if you like.

Figure 49: Authorization required



4.6 Other configuration

While this pretty much covers the basic setup of *.htaccess* files, there are several more things you should remember.

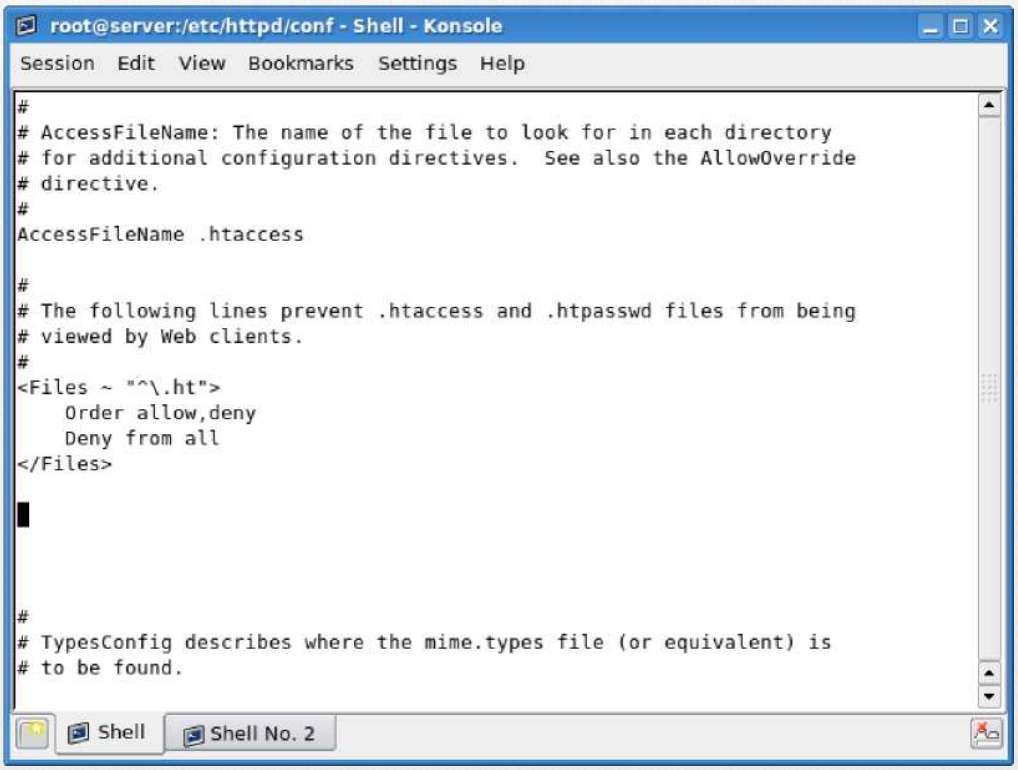
4.6.1 Inheritance & performance loss

Please remember that the *.htaccess* restrictions are inherited by all sub-directories that exist in the directory you have placed the file. This means that whenever one of your clients tries to access a page in one of the sub-directories, the server will have to make a recursive search up the directory tree until it finds the file. Furthermore, even if it does find the file, the server will have to check up every directory up the tree to create a complete set of restrictions.

4.6.2 Disable web access to .htaccess

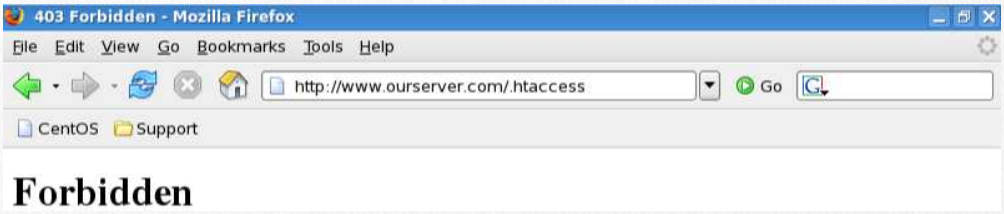
By default, Apache prevents any file beginning with letters *.ht* to be visible through the web browser. This is a m inor security consideration, which allows you to keep your *.htaccess* files safe from prying eyes, even though they are located in world-readable location *(DocumentRoot* directories and sub-directories). This behavior is governed by the combination of the *AccessFileName* and *Files* directives. We have seen this example earlier when we review the *Files* tag; now, we can see them in practical use.

Figure 50: Disabling web access to .htaccess



You can also setup other types of files - orjust specific files - from being accessible - or accessible only to certain hosts. Indeed, if we try to reach *.htaccess* through the web browser, we will be denied access.

Figure 51: Forbidden message on access to .htaccess



Done

5. Secure Web server

Running a secure Web server is something you should consider if the daily use of your websites will include an exchange of confidential, private information from your users. Regular Web servers send and receive traffic in unencrypted form. Unfortunately, this makes them vulnerable to man-in-the-middle attacks, where a potential attacker could use sniffer tools to log packets en route from clients to the server and derive sensitive information from them. This mode of security is completely unacceptable for websites that must deal in personal data, like bank accounts, medical or financial records, or others. The secure Web server eliminates this threat by offering two key advantages:

- It allows users to verify the identity of the server.
- It allows users to conduct safe transactions with your server by encrypting the authentication and the session.

To achieve this, the Apache Web server uses secure communication protocols like the Secure Socket Layer (SSL) or the Transport Layer Security (TLS) to protect the flow of data.

5.1 Encrypted session

Before we setup a secure server, we should first understand how encrypted communication between the server and the client is conducted. Let us outline the details of a typical secure session:

- A client tries to connect to port 443 on the secure Web server.
- The client sends a list of available encryption methods it supports; if the client cannot support encryption, for instance very old browsers, the connection attempt will be unsuccessful. Modern browsers support both SSL and TLS without any problems.
- The server will choose the strongest available encryption method that both sides can support.

- The server will then send back to the client its certificate and the public encryption key. The certificate is a sort of an ID, telling the client important information about the server. To make this information credible, the certificate must be signed by a reputable Certificate Authority (CA), like EquiFax, Thawte or others. The public key will be used by the client to generate its own encryption hash should it choose to accept the server's certificate.
- The client receives the certificate. In most browsers, the certificate is first compared to an existing list of authorities. If the digital signature matches, the certificate will be accepted. If no match is found for the certificate, the browser might use the Online Certificate Status Protocol (OCSP) to connect to CAs in real time in an attem pt to verify the certificate. Generally, the use of OCSP is not enabled by default in most browsers, in order to speed up the authentication process. If no match is found still, the client will be issued a warning by the browser, informing it that the certificate could not be verifi The user now must decide whether he/she can take the risk and accept the certificate. In addition to being self-signed (i.e. no CA signature), the typical issues arising with certificate prompts include a mismatch between the site you are trying to access and the one registered in the certificate, dubious credentials or an expired certificate.
- Regardless of what may occur, if the client accepts the connection, it will send back a hash encrypted with the server's public key. This hash will be used to encrypt all communication between the server and the client throughout the session. Only the client will be able to decrypt the communications - or rather, anyone who possesses the private key. But if the client side is fairly secure and the server's certificate is valid, the communication is safe.

5.2 Requirements

We have already mentioned that the client must support some sort of encryption to able to establish secure connections to a server. On the server end, the server must also support the secure communication protocols. The Apache Web server uses the *mod_ssl* module, which provides an interface to the OpenSSL library, allowing the use of SSL and TLS.

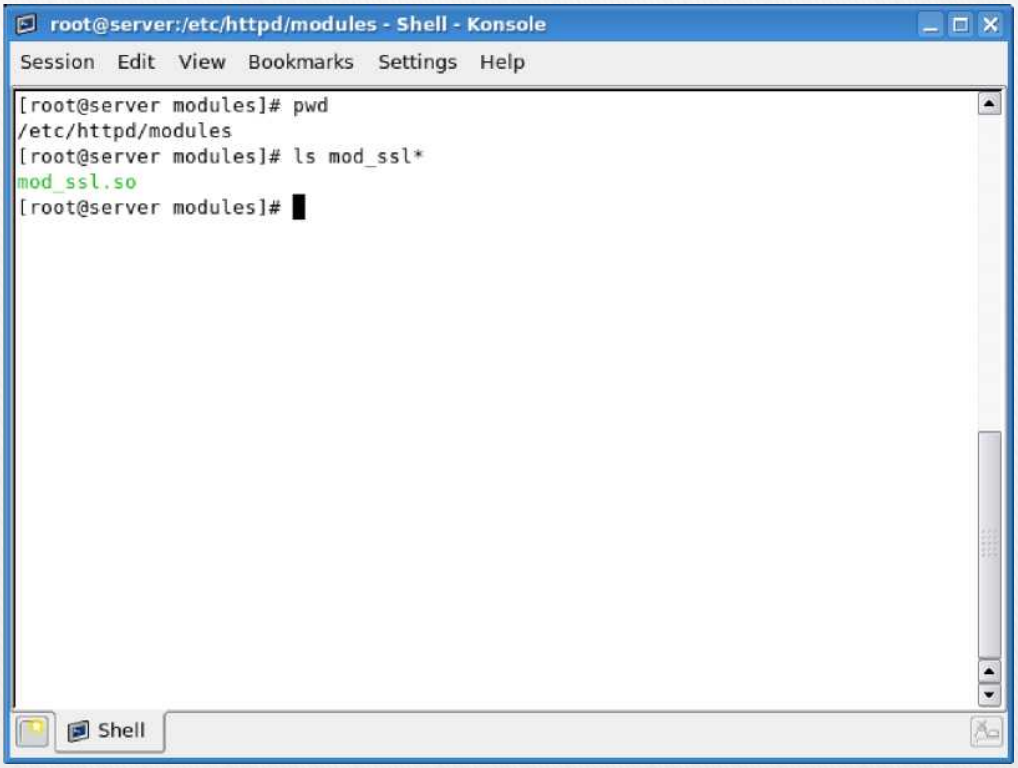
By default, most distributions today ship with the OpenSSL library installed and the Apache server compiled against the *mod_ssl* module. If your distro does not

include either one or both, you will have to obtain them before you can use a secure Web server. You can check if you have the OpenSSL library installed:

```
rpm -q openssl
```

And to certify if Apache uses *mod_ssl,* you should look for it in the*/etc/httpd/-modules* directory.

Figure 52: mod_ssl module under /etc/httpd/modules



5.3 Limitations

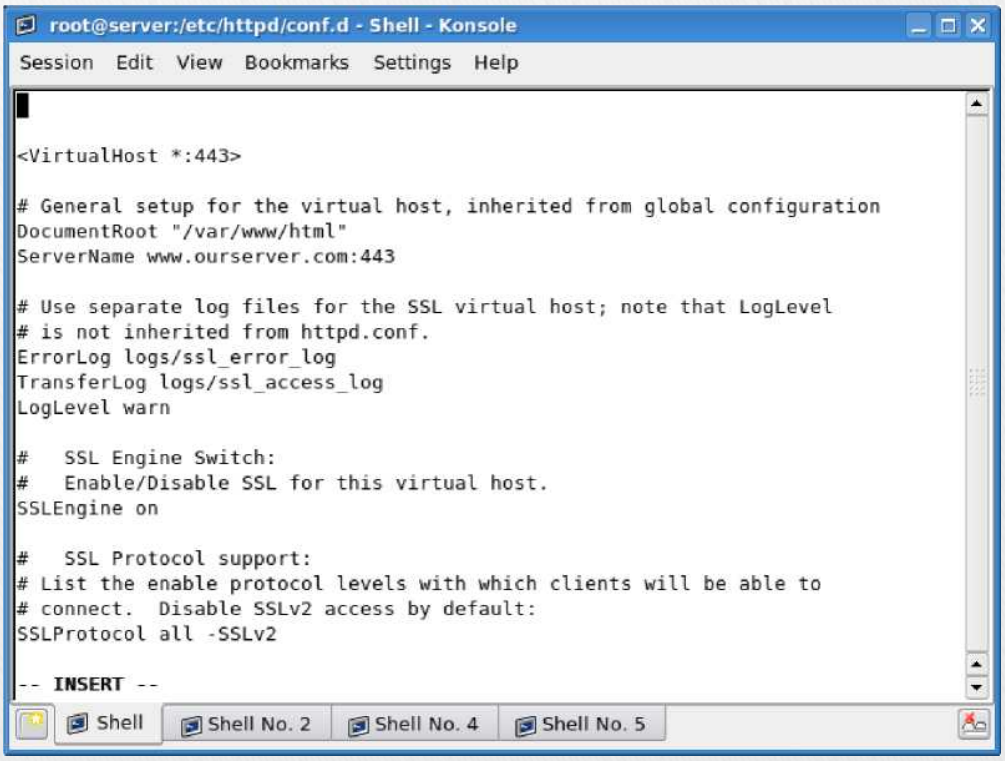
On one hand, the secure Web server offers verification of the server's identity and safe transactions. On the other hand, it is slower than the regular server.

Therefore, you should take into consideration the performance loss stemming from the use of encryption. You should not use the secure Web server for regular daily content that does not include any exchange of personal information.

5.4 Main configuration file(s)

The main configuration file for the secure Apache Web server is /etc/httpd/-conf.d/ssl.conf. This file is very similar to *httpd.conf,* except that it includes a number of special directives. But the principle remains the same. Basically, the configuration file contains a *VirtualHost* block, where all secure Web server directives should be listed. We will edit this block to suit our needs. Don't forget to backup!

Figure 53: ssl.conf configuration file



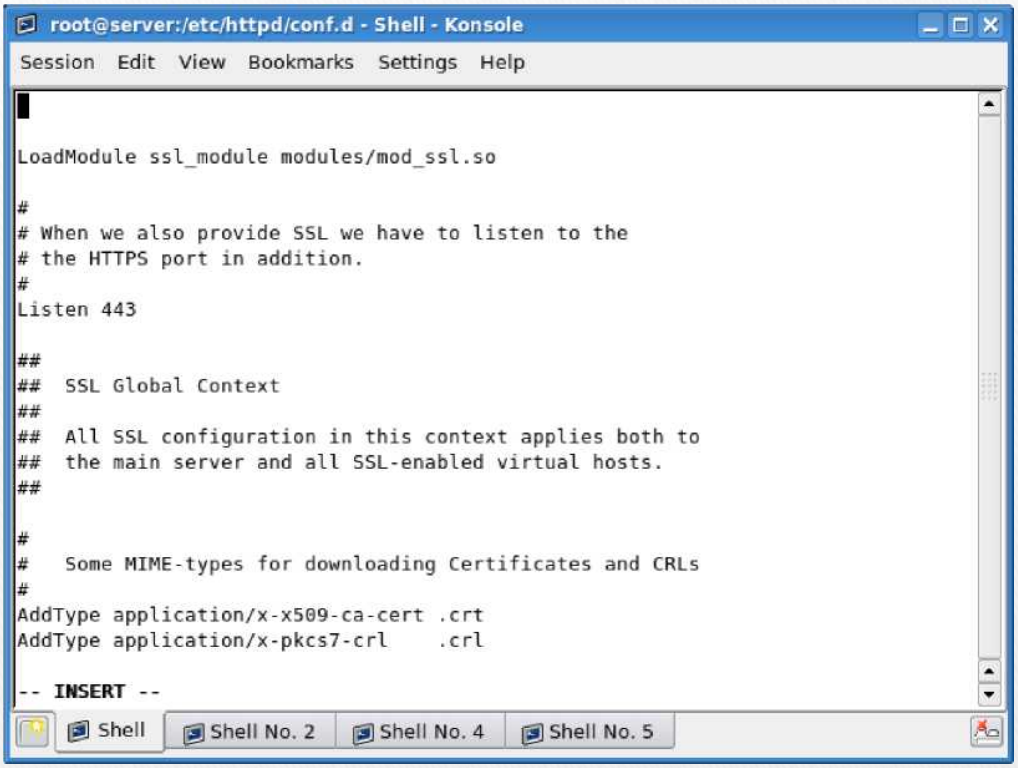
5.5 Edit the ssl.conf confi file - part 1

Again, we need to make a number of changes to get our server to work. However, before we can fully edit all of the necessary options, we will have to digitally sign our server. This includes creating the public key and signing it with a certificate from a known, reputable CA. However, since we do not have a certificate, it costs money and the process takes time, for the purpose of this exercise, we will create our own CA and use it to sign our server. But first, let us review the most important directives that we need to get our server started. The procedure is identical to what we have done earlier.

5.5.1 LoadModule

This directive instructs the server to use the *mod_ssl* module. The path is relative to the *ServerRoot* directive specified in the *httpd.conf* configuration file. Without loading the module, our encryption will not work.

Figure 54: Loading SSL module



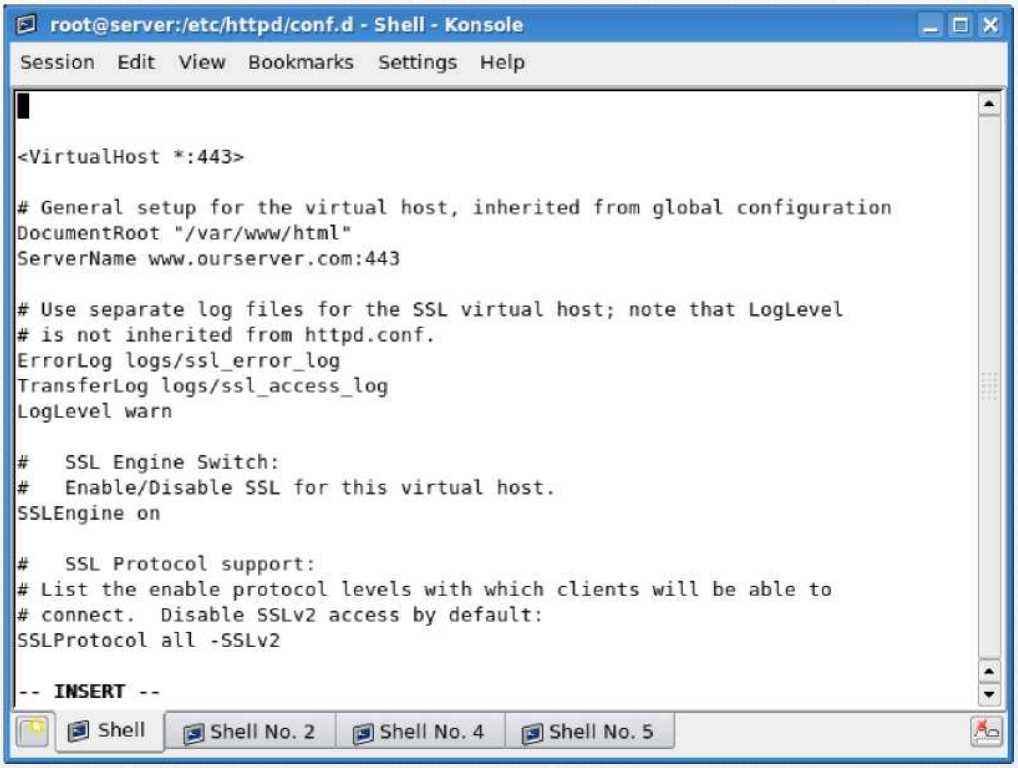
5.5.2 Listen

This directive instructs the server to listen for incoming connections on port 443. This is the accepted convention for secure Web communications (https). It is critical that this port be different from the port used by the regular server. See the image above.

5.5.3 VirtualHost

Here, we define our secure Web server. Using the *VirtualHost* block is the most elegant way of doing it. This allows you to create additional blocks and serve additional secure sites to your clients, allowing you an extra degree of flexibility and security.

Figure 55: VirtualHost block for secure site



Like we did before, we need to setup the *DocumentRoot,* the *ServerName* and other directives. Let us review the most important elements:

<VirtualHost *:443> tells our server to listen on all interfaces for incoming connections on port 443. You may consider narrowing down the range to specific IP addresses. Nevertheless, it is important to remember that you can only use IP addresses! The secure Web server does not permit named-based connections in its *VirtualHost* block. This is because the SSL handshake occurs before the HTTP request can identify the named-based virtual host.

> **Important note:** Use only IP-based *VirtualHost* directives in the *ssl.conf* configuration file! Name-based virtual hosts will fail.

DocumentRoot "/var/www/html" specifies the directory where all your web pages should be stored. It is recommended that you use a different root for

non-secure and secure pages. However, in our example, we will use the default selection. Just remember that this is NOT the optimal setting.

ServerName www.ourserver.com:443 defines the server name. If you do not use the *hosts* file or DNS server for name resolution, you will have to specify an IP address. We have solved this limitation earlier, so we can use the server name here. In a production setup, where your server is used by clients on the Internet, you will have to use DNS for name resolution. For study and testing and in small, private networks, the *hosts* file is an adequate solution. This covers the first part of our setup. Now we must create the certificate.

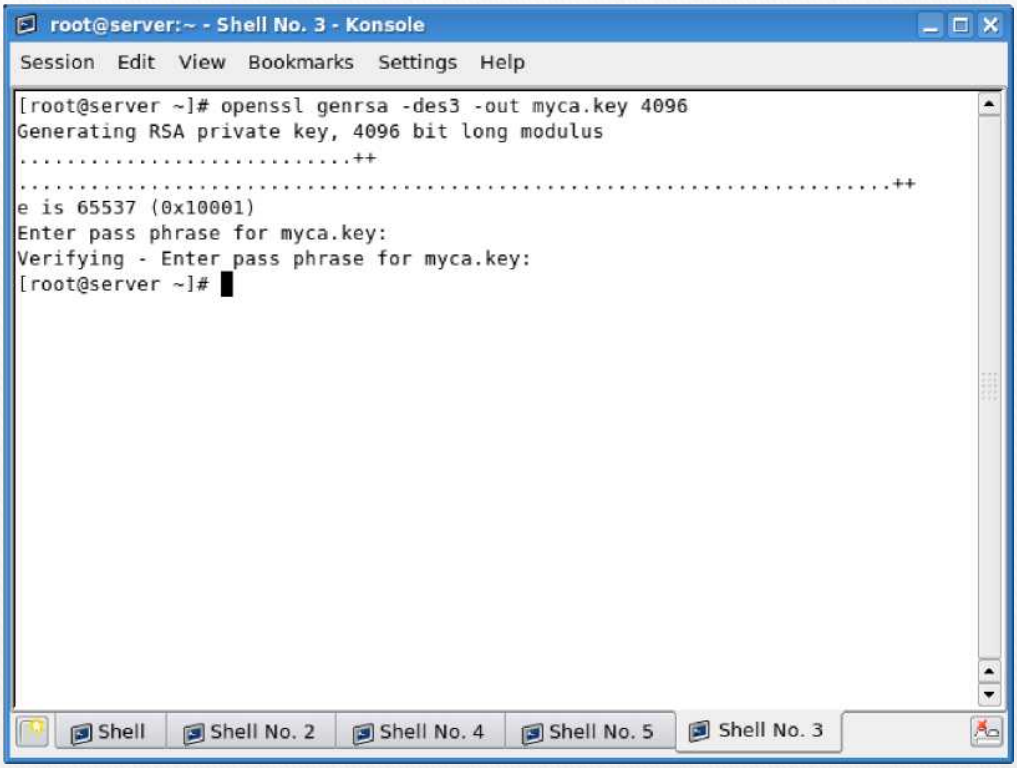
5.6 Create SSL certification

Like we said before, we will create a CA, create a server key and then sign the key with our self-created CA. In a production setup, this will not work. If you intend to run any semi-serious business, you will have to use a reputable, world-acknowledged CA to sign your certificates. Please note that the comparison between our setup and the real scenario can be slightly confusing. If you get lost, there's a table summary (5.9) at the end of this section, emphasizing the important differences between the two setups.

5.6.1 Create Certificate Authority (CA)

The first step is to create an encryption key, which we will use to sign our CA. Please note that you should use a meaningful name for the key. The best way to avoid confusion is to use the letters *ca* in the name of the CA key. Likewise, use the word *server* when creating the server key. I have chosen the name *myca.key,* so that we do not confuse this self-generated key (and the CA) with real keys.

Figure 56: Creating Certificate Authority key



Let us review the command:

```
openssl genrsa -des3 -out myca.key 4096
```

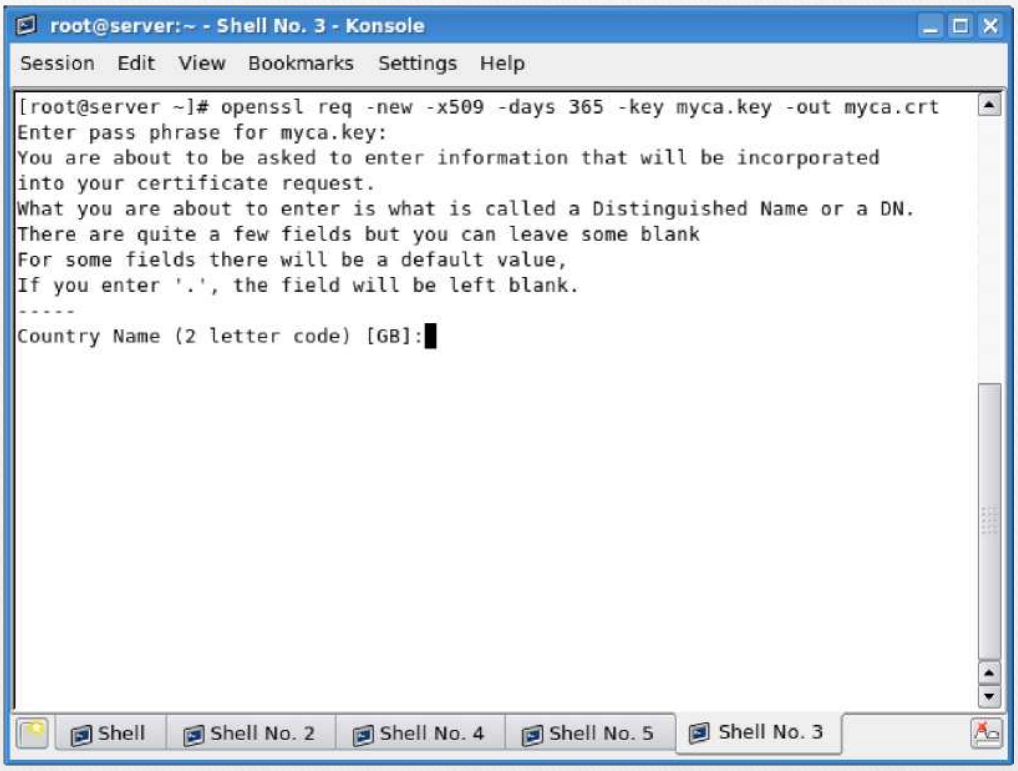
This OpenSSL command line tool will generate an RSA key, using the T riple-DES cypher. The *-out* flag signifies the output name. The number at the end of the command tells us how long the key will be; generally, the longer the better. A 4096-bit encryption is quite sufficient. Please refer to openssl man page for more details.

After the key is created, you will be asked to use a password. This means you won't be able to use this key without providing the password. While in theory,

this is an interesting security measure, it offers little actual benefit. We'll discuss this soon. Now that we have the key, we will create a CA.

```
openssl req -new -x509 -days 365 -key myca.key -out myca.crt
```

Figure 57: Creating new certificate

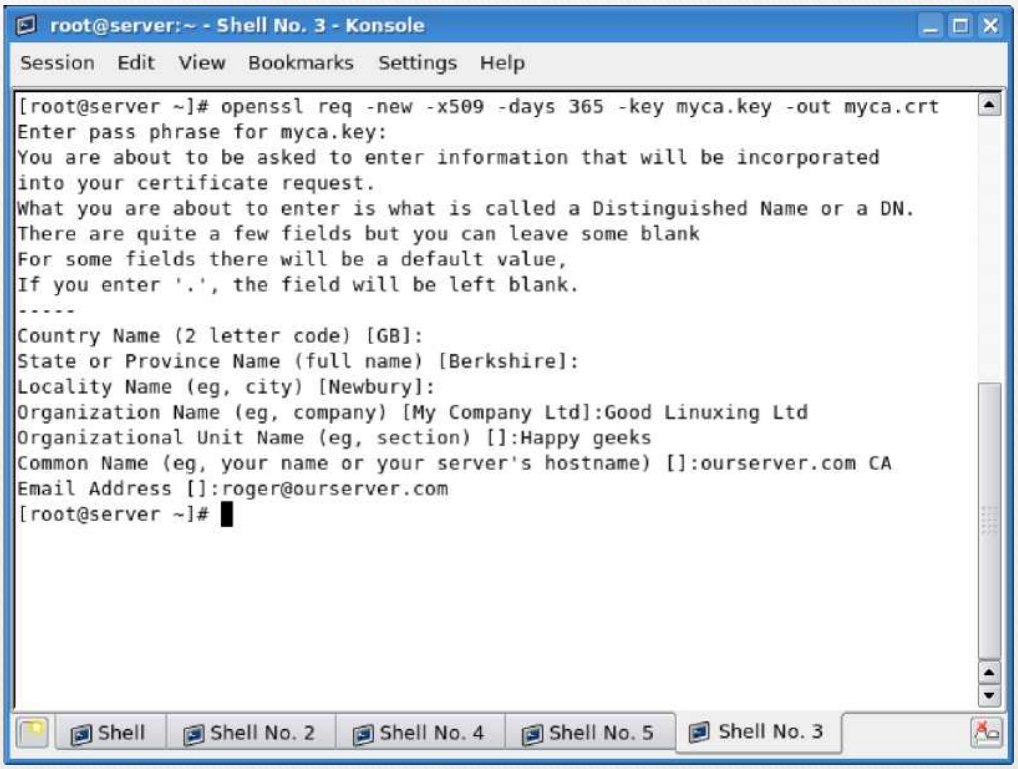


What do we have here? Well, basically, we are creating a certificate, using the key we have created earlier. Let us go over the details:

req -new -x509 tells us we want to issue a new X.509 Certificate Signing Request (CSR), where X.509 is an international standard for public key and privilege management infrastructures. In simple words, we want to create a certificate that will identify our CA.

-days 365 tells us how long the certificate will be valid. Security aspects of this parameter are examined in greater depth in the Security chapter.
-key myca.key -out myca.crt - we will use the key we have created earlier to sign the certificate for the CA. The command will invoke a guided text-interface wizard. We will have to provide the password for the certificate key before we can continue. After that, we will have to fill out an interactive form, including the basic credentials that will identify us as the CA.

Figure 58: Creating new certificate, continued



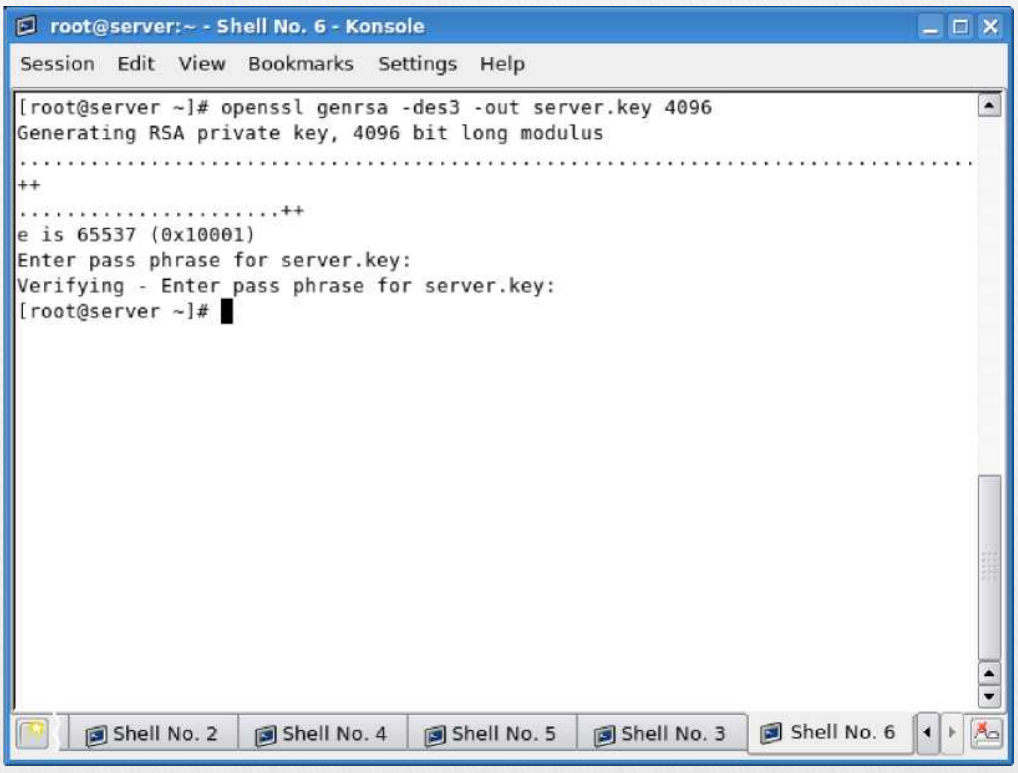
Please note that you should be careful when entering the *Common Name.* You should use meaningful entries that will allow you to easily distinguish your records, especially if you have several CAs. Most people will never have to bother with this setting, but should a need arise, here's a pair of simple rules that you should adhere to when creating CAs: For each CA, use the name of the site it will certify; in our case, ourserver.com (or

www.ourserver.com). Append the letters *CA* to the end of the *Common Name,* so you will know this is the CA entry. In a real life situation, your credentials would be replaced with those of an existing, reputable CA.

5.6.2 Create server key

We now have a certificate. It's time to create the server key. The principle is similar to what we've done before. The one thing you should remember is that the server key should be named *server.key,* in order to conform with Apache conventions.

Figure 59: Creating server key



After the encryption key is created, we will be asked to provide a password to make the use of our key impossible without knowing it. While this method is

somewhat effective, it is not considered a serious security measure. In fact, you are advised not to use it, since the benefits do not outweigh the shortcomings.

Since you must provide the password any time the server is restarted or reloaded, this means the secure server will not be able to start after unattended reboot and will require a presence of an administrator to activate. This is cumbersome and can even be impractical. On the other hand, should your system be compromised, the password will most likely present little challenge to the attacker. Furthermore, compromised systems cannot be trusted, whether passwords or other security methods are used.

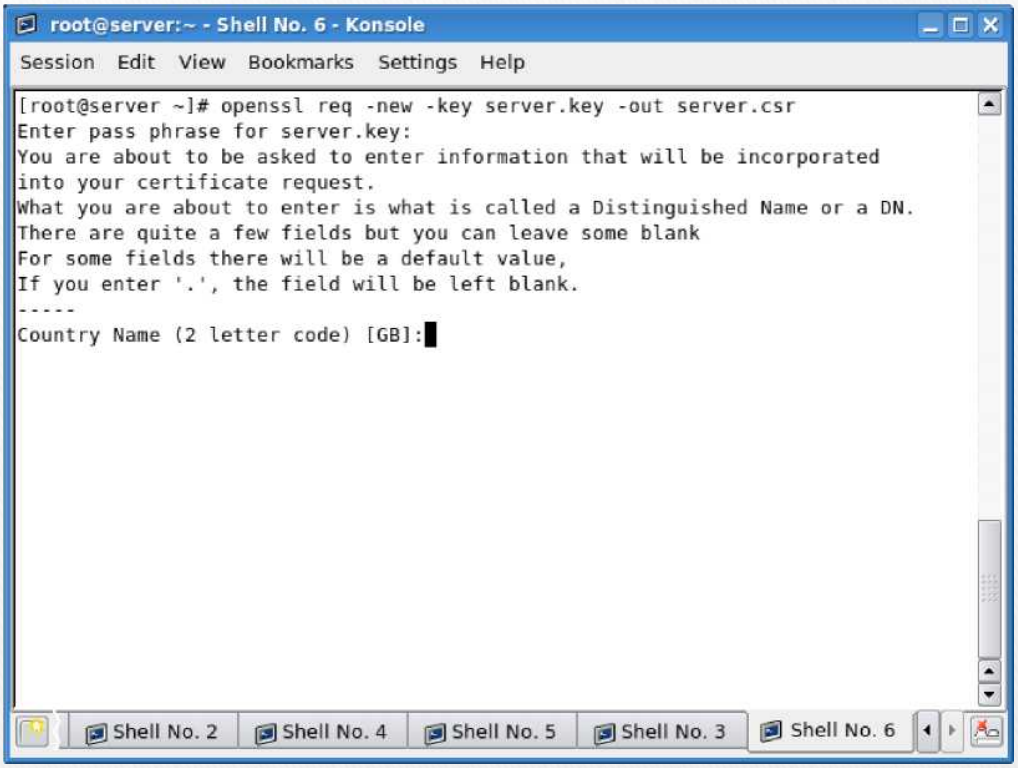
Nevertheless, we will demonstrate both methods, so you can learn and use both, should a need arise. We will begin with the password-protected key and then later, create another one, which uses no password.

5.6.3 Create Certificate Signing Request (CSR)

Now, we must "ask" our CA to sign our certificate. In a real life situation, you would receive the *server.csr* from an existing, established CA. Or you might even receive the signed key, with the information you have provided in an application form, for instance.

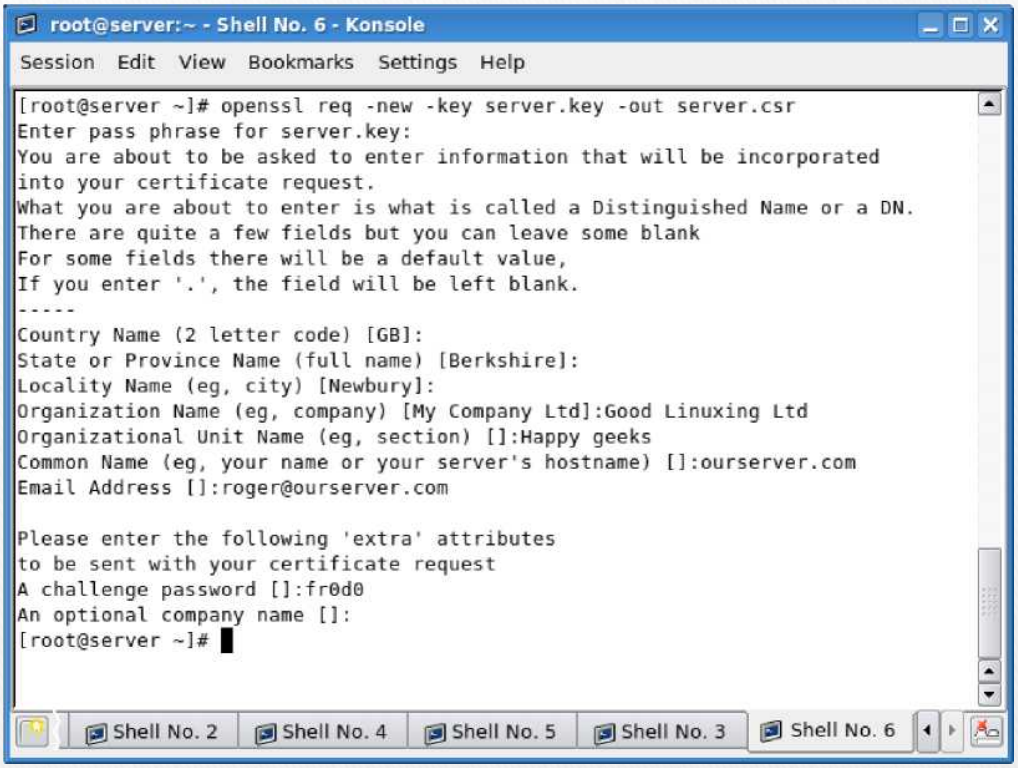
Again, we must provide a password before we can continue. Then again, we must go through an interactive form, providing details for our site. In a real life situation, a real CA would ask you for these details, whether via email, phone, an application form etc. For more details, please refer to Submission of CSR to CA sub-section below (5.11).

Figure 60: Creating Certificate Signing Request



Since our CA and our website are one and the same, the form will differ little from what we have done when creating the CA. This can be confusing. Therefore, you should remem ber that the *Common Name* for your CA should include the letters *CA* (or similar), to distinguish it from the server record. Lastly, you can provide an additional password for the server key, to make misuse more difficult.

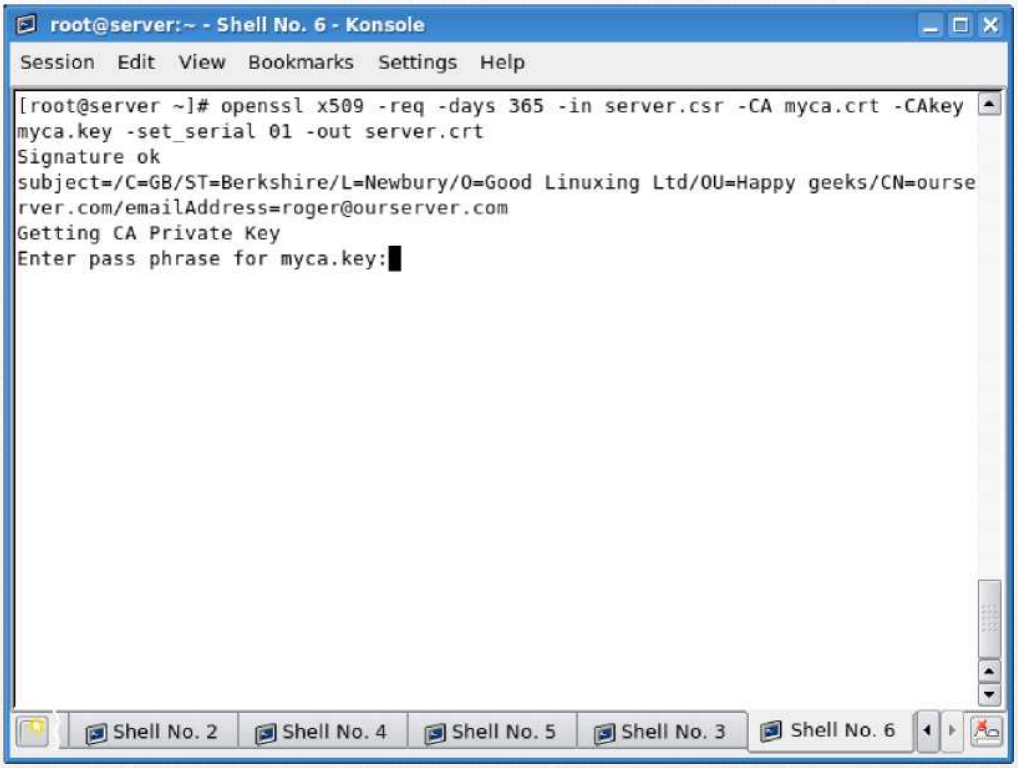
Figure 61: Creating Certificate Signing Request, continued



5.6.4 Sign Certificate Signing Request (CSR) with Certificate Authority (CA)

What remains to be done is to sign the CSR with the CA we have created. Once we do that, our certificate will be valid for the coming year. After that, we will have to renew it.

Figure 62: Signing CSR with CA



You should be familiar with the syntax by now:

-CA myca.crt instructs *openssl* to use our CA certificate.

-CAkey myca.key instructs *openssl* to sign the certificate with the CA key.

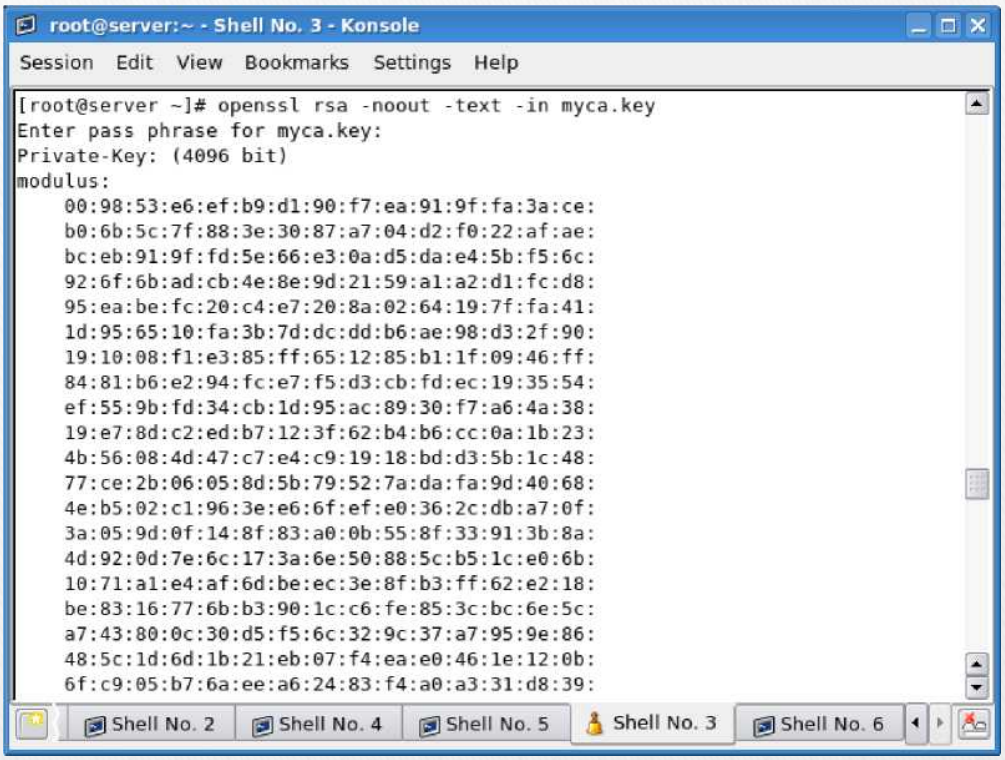
-set_serial 01 option is used to create a serial number when outputting a selfsigned certificate. This allows you to track the changes done to the certificate. This covers the creation and signing of the SSL certificates.

5.6.5 Verify certificate

Let's examine the certificates we have just created. This can help you see if there are any problems with your files.

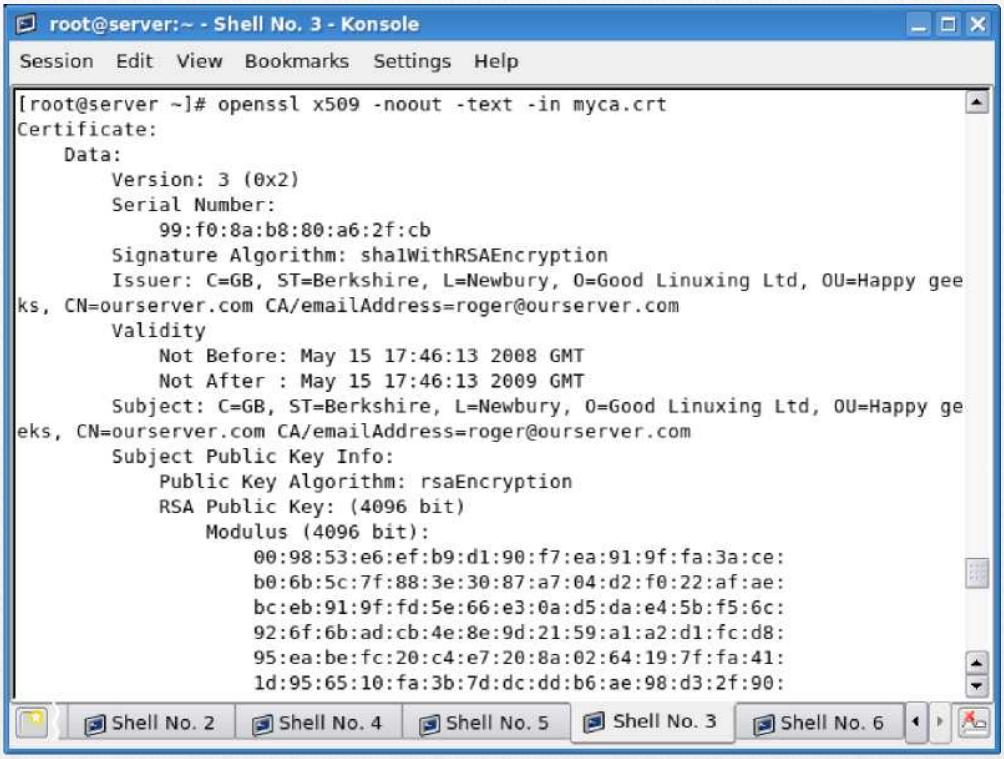
openssl rsa -noout -text -in myca.key

Figure 63: Verify myca.key



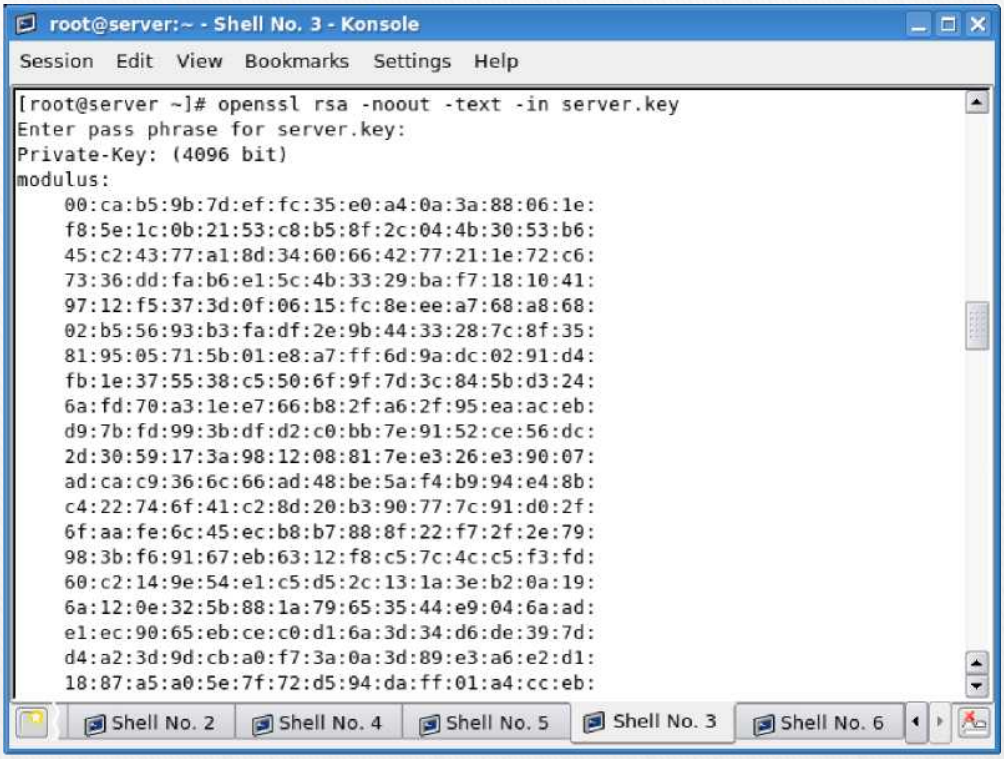
openssl x509 -noout -text -in myca.crt

Figure 64: Verify myca.crt



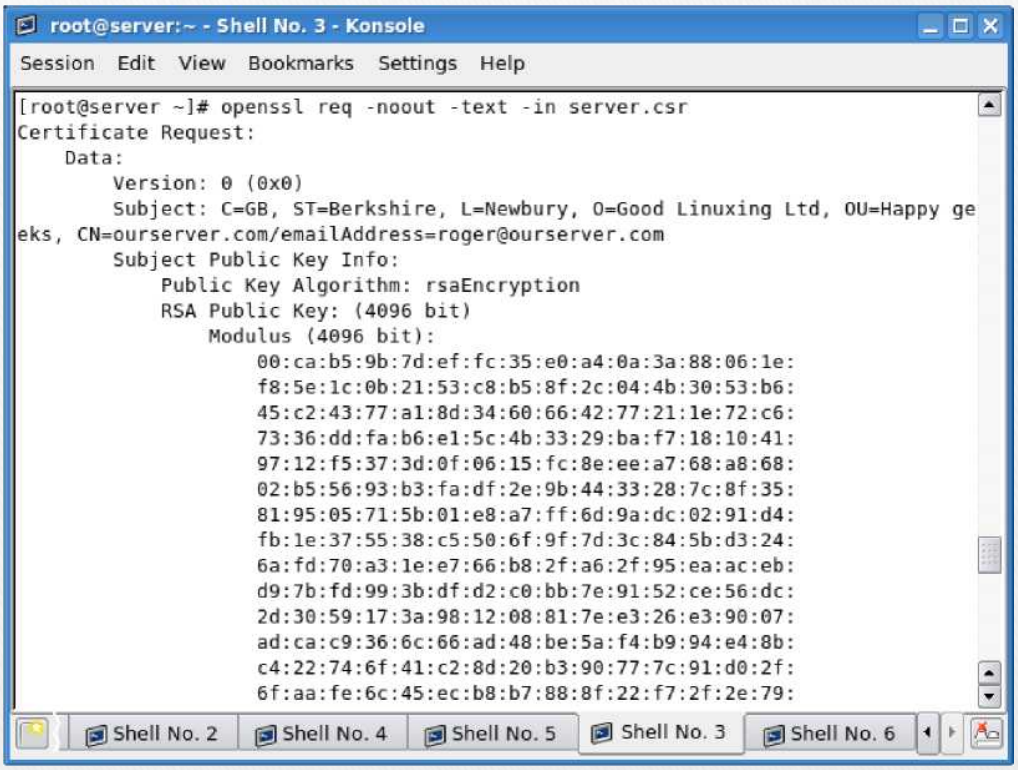
openssl rsa -noout -text -in server.key

Figure 65: Verify server.key



openssl x509 -noout -text -in server.csr

Figure 66: Verify server.csr



Everything looks good. Now, we can finish editing the *ssl.conf* file.

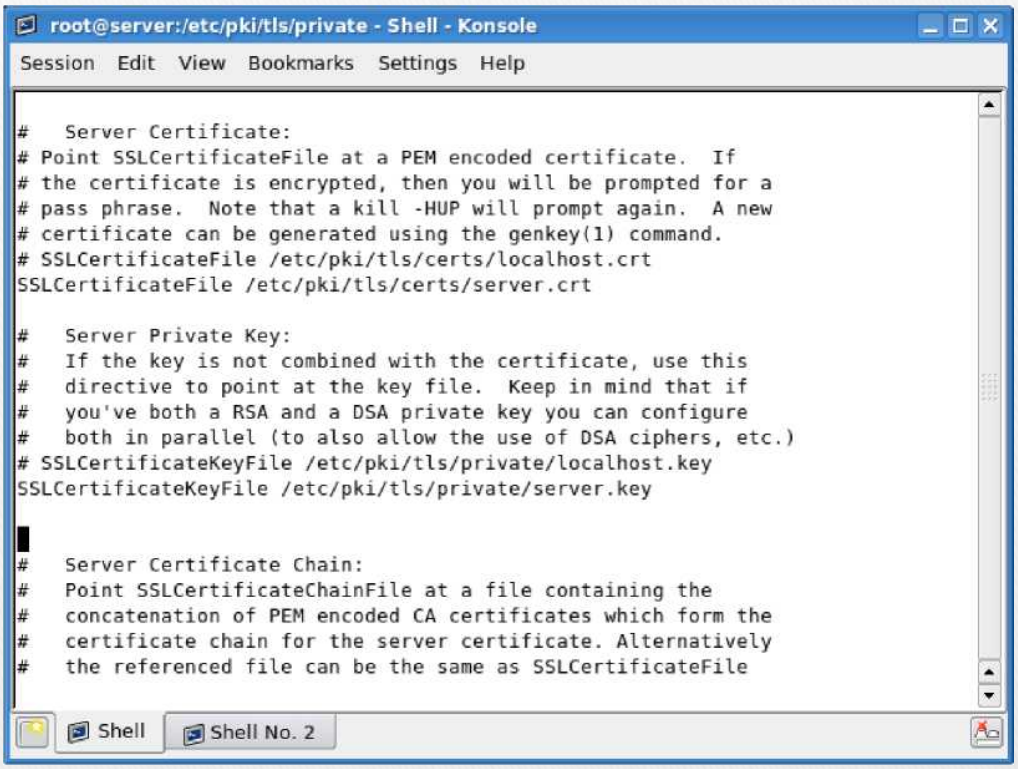
5.7 Edit ssl.conf configuration file - part 2

We now need to specify the location of our certificates and the keys in the *ssl.conf* file so the server can find and use them. We will comment out the sample entries in the file and use our own. Necessarily, we will have to copy the relevant files to their right location.

5.7.1 Server Certificate

This directive specifies the location of the server certificate (*server.crt*). On CentOS 5, the default location is */etc/pki/tls/certs.* We will use the same directory. Your choice may vary. The important thing to remember is to make the files unavailable to anyone but root.

Figure 67: Server Certificate location



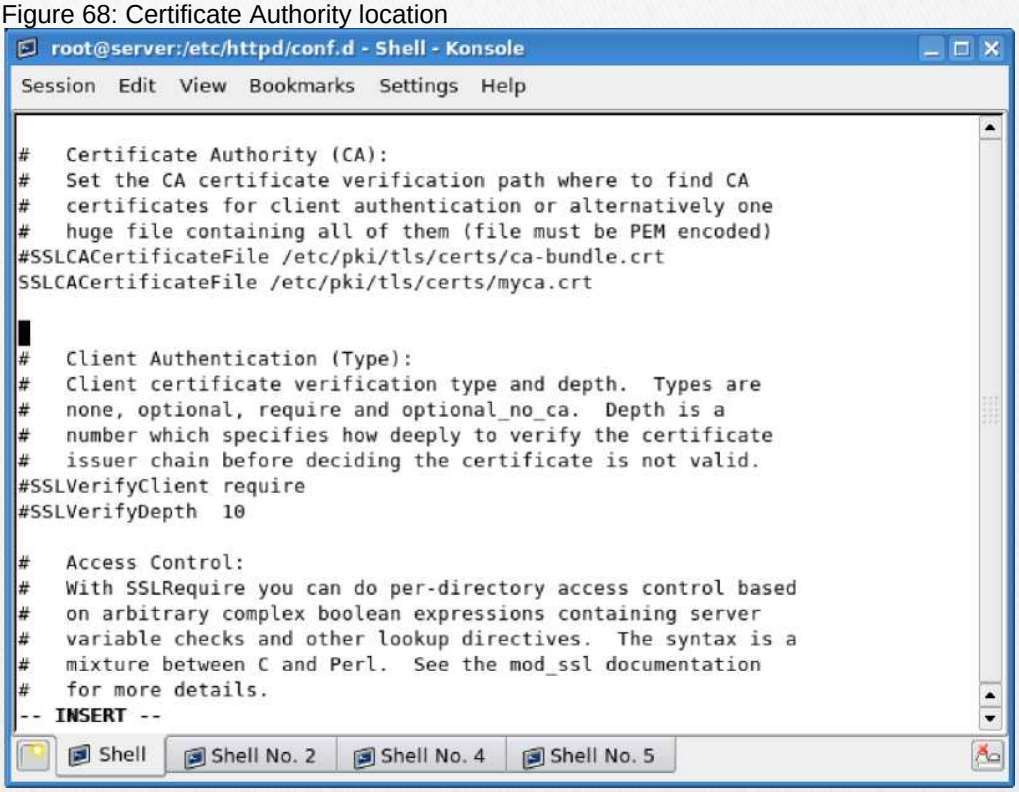
5.7.2 Server Private Key

This directive points to the location of the server key *(server.key*). Again, your choice should reflect your needs. See image above.

5.7.3 Certificate Authority

This directive specifies the location of the CA certificate.

Figure 68: Certificate Authority location



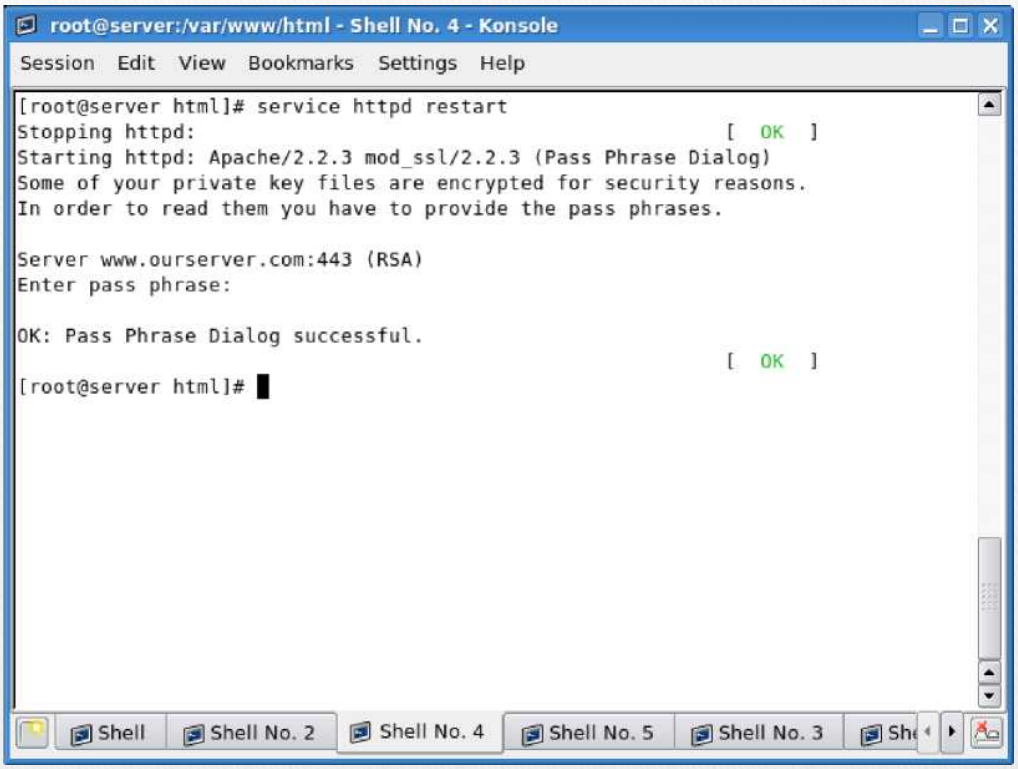
Now, let us copy the files to their relevant locations:

cp server.key /etc/pki/tls/private/server.key cp server.crt /etc/pki/tls/certs/server.crt cp myca.crt /etc/pki/tls/certs/myca.crt

5.8 Test setup

We are ready. Let's test our setup. After saving the *ssl.conf* file, we need to restart our server.

Figure 69: Restart server with password



As you can see, we must provide a password before we can continue. Now, we will try to access our server by typing *https://www.ourserver.com* in the address line of a web browser. You will most likely receive a warning message.

Figure 70: Unknown certificate website warning

Let us examine the certificate before we accept it.

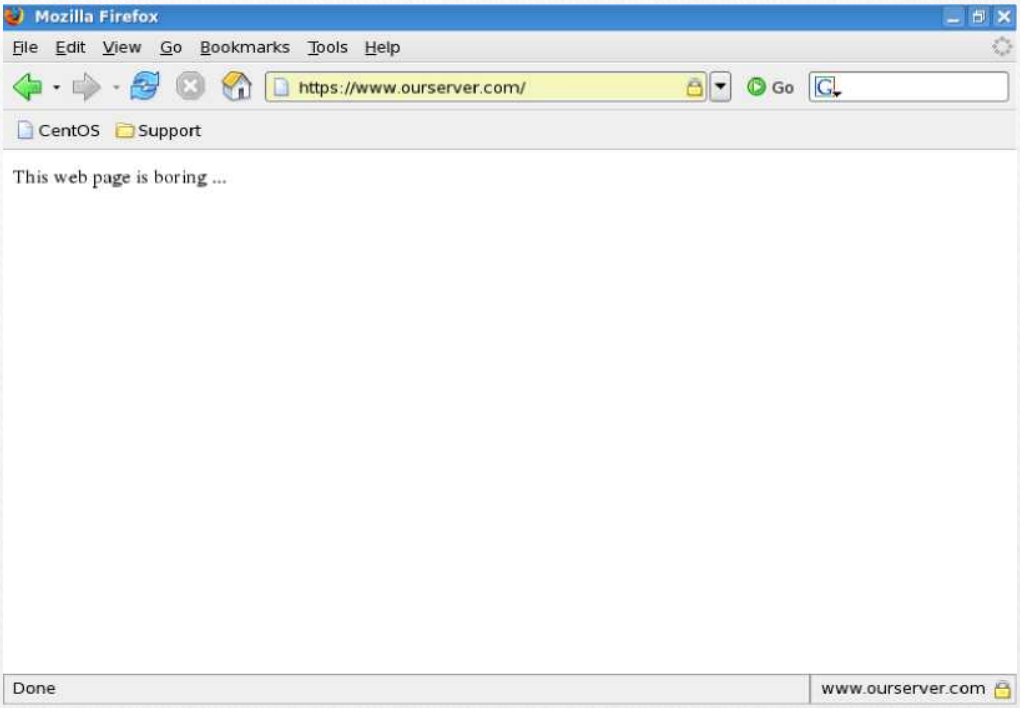
Figure 71: Certificate Viewer

Indeed, everything looks fine. On the Web, though, very few people would be convinced by this certificate. But in our setup, it serves well. After accepting the certificate (either permanently or temporarily for this session only), you will hit yet another warning.

Figure 72: Domain Name Mismatch error

This time, there's a mismatch between the domain name (www.ourserver.com) and the certificate *(ourserver.com).* This should not be an issue if you are using the DNS server, but we will discuss this separately in the next Part. For now, we will accept the certificate. After that, we should reach our site safely. Our setup works.

Figure 73: Secure site loaded



5.9 Mini-summary

Setting up the secure Web server might seem a little confusing. Therefore, here's a mini summary that should clarify the setup process.

5.9.1 Names

This is a short list of file names used in this section:

Table 2: Certificate and key names

File name	Description
myca.key	CA key
myca.crt	CA certificate
server.key	serverkey
server.csr	server CSR
server.crt	server certificate, signed by CA

5.9.2 Commands

Below, you can find a summarized list of commands you will need to run to create your certificate. Please note that the names I have used are generic and might not suit your needs. However, it is important that you use the name *server.key* for the server key file, to conform with Apache standards.

Table 3: openssl commands

Command	Description
openssl genrsa -des3 -out myca.key 4096	Create CA key
openssl req -new -x509 -days 365 -key myca.key -out myca.crt	Create CA certificate
openssl genrsa -des3 -out server.key 4096	Create server key
openssl req -new -key server.key -out server.csr	Create CSR
openssl x509 -req -days 365 -in server.csr -CA myca.crt -CAkey myca.key -set_serial 01 -out server.crt	Sign CSR
5.9.3 Difference between self-signed and CA-signed certifi	

This will help you better understand the differences between our exercise and a real, production setup.

Table 4: Difference between self-signed and CA-signed certificates

Step	Self-signed CA	Real CA
Create CA key	Yes	No need
Create CA	Yes	No need
Create CSR	Yes	As instructed by CA
Create serverkey	Yes	Maybe: 1. CA creates key, signs it and sends to customer 2. CA creates CSR only and sends to customer, who then creates server key and signs with CA
Sign server key	Yes	Maybe: 1. CA sends signed key to customer 2. CA sends CSR; customer signs the key by himself/herself

5.9.4 Verification

You will have to run these commands to check your certificates and keys:

```
openssl rsa -noout -text -in server.key openssl x509 -
noout -text -in server.crt openssl rsa -noout -text -in
myca.key openssl x509 -noout -text -in myca.crt
```

5.9.5 File names and locations

These are the locations of relevant files:

Table 5: Secure Web server file names and locations

Full path and name	Description
/etc/httpd/conf.d/ssl.conf	Main configuration file
/etc/httpd/modules	Location of all modules
/etc/httpd/modules/mod_ssl.so	Location of *mod_ssl* module
/etc/pki/tls/certs	Location of server and CA certificates
/etc/pki/tls/private	Location of server keys

5.10 Extras

Now that we have our secure Web server running, let us review a number of other options.

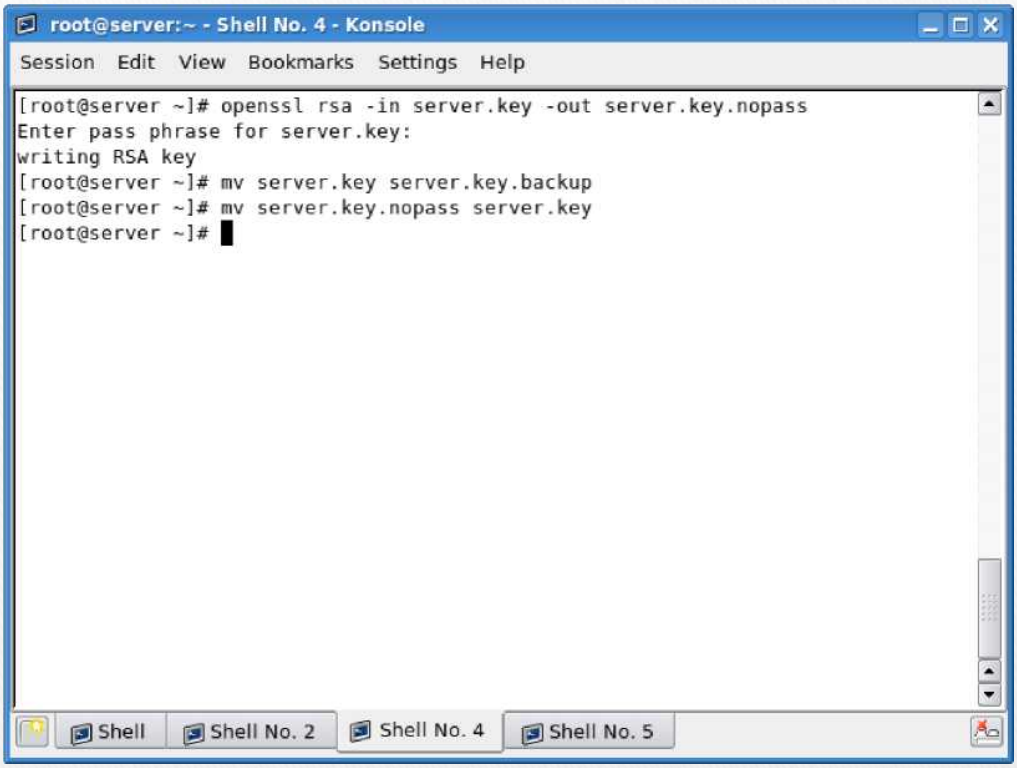
5.10.1 Do not use password-protected server keys

As said before, the password protection for server keys is rather impractical, without significantly contributing to the server security. Therefore, you are encouraged not to use them. Of course, you must make sure that your server security is ensured by other means.

5.10.2 Create server key without password

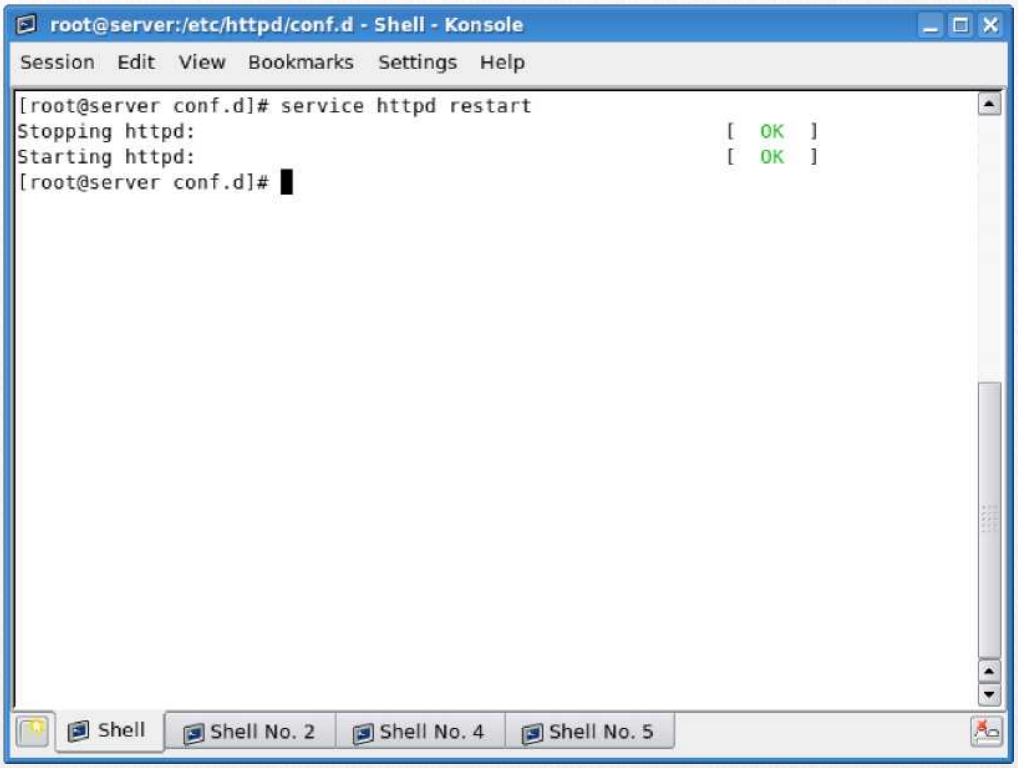
We will “filter” the old key (*server.key*) into a new one (*server.key.nopass*), which will not include a password. Then, we will swap between the old and the new one. You are advised to keep a backup copy of the original key, just in case.

Figure 74: Allow secure site restart without password



Next time we restart the server, we won't be prompted for a password. All other settings remain unchanged.

Figure 75: Restarting Web server without password prompt



5.11 Submission of CSR to CA

It is difficult to say what method the CA will use to certify your site. They m ight ask you for your credentials over a form or through an online form. Or they might ask you to submit a CSR, which they will sign, returning the key to you. Here, we will review a typical option. Again, this may not be the case you'll encounter.

5.11.1 Create CSR

Assuming the CA wants all records in the purely digital form, you will have to create a CSR. We have already done that. Optionally, they might ask you to convert the file into a Privacy Enhanced Mail (PEM) format. Please note that this format is not widely used and this will probably not be necessary.

```
openssl x509 -inform crt -in server.crt -out server.pem
```

Please note that you should consult your CA for detailed instructions regarding the conversions, if at all required.

5.11.2 Send CSR to CA

You will now have to submit the CSR file (or PEM) to the CA. Once the CA processes your application, you will receive the certificate back. The file will most likely be sent in the PEM format, so you will have to convert it back to CSR format. It will also most likely bear a different name from what you are used to, so you should rename it to *server.crt,* to conform with Apache conventions.

5.11.3 Verify certificate

Now, you should verify the certificate, against the relevant CA's file. You should receive this file from your CA. Alternatively, if your distro includes a list of CAs, you m ight try that one. On CentOS 5, a bundle containing a list of known CAs is located under */etc/pki/tls/certs.*

```
openssl verify -CAfile ca-bundle.crt -purpose sslserver
server.crt
```

Next, you should verify that the certificate corresponds to the private key. Please make sure the names match.

```
openssl x509 -noout -modulus -in server.pem | openssl md5
openssl rsa -noout -modulus -in server.crt | openssl md5
```

Once you have completed the above steps, you will have to edit the *ssl.conf* file, restart the server and test your setup. We're back on familiar grounds. For more details regarding the different commands and verifi please consult verify man page and OpenSSL: Documents, req.

5.12 General considerations

Here's a number of settings that you should remember.

5.12.1 Use secure server only

If you want your server to serve only secure pages, simply comment the *Listen 80* directive from the *httpd.conf* file. Sites dedicated to clients privacy and security should not run any other, non-secure content.

5.12.2 Use only IP-based virtual hosts

It is impossible to use named-based virtual hosts with the secure Web server. This is because the SSL handshake occurs before the HTTP request can identify the named-based virtual host. Using names will result in errors. You may only use IP addresses in the *VirtualHost* directives inside the ssl.conf configuration file.

5.12.3 Use server.key as file name for the server key

It is not strictly necessary, but it is good practice and in line with Apache conventions to use the *server.key* name for the server key file. If for some reason you require paranoid security, then you might change this name to something less obvious.

Authentication and Authorization

Authentication is any process by which you verify that someone is who they claim they are. Authorization is any process by which someone is allowed to be where they want to go, or to have information that they want to have.

Related Modules and Directives

There are three types of modules involved in the authentication and authorization process. You will usually need to choose at least one module from each group.

- Authentication type (see the AUTHTYPE directive)
  - MOD _AUTH .BASIC
  - MOD _AUTH .DIGEST
- **Authentication provider (see the AUTHBASICPROVIDER and AUTHDIGESTPROVIDER directives)**
  - MOD _AUTHN _ANON
  - MOD _AUTHN _DBD
  - MOD _AUTHN _DBM
  - MOD _AUTHN .FILE
  - MOD _AUTHNZ _LDAP
  - MOD _AUTHN _SOCACHE
- Authorization (see the REQUIRE directive)
  - MOD _AUTHNZ _LDAP
  - MOD _AUTHZ _DBD
  - MOD _AUTHZ _DBM
  - MOD _AUTHZ .GROUPFILE
  - MOD _AUTHZ .HOST
  - MOD _AUTHZ .OWNER
  - MOD _AUTHZ .USER

Introduction

If you have information on your web site that is sensitive or intended for only a small group of people, the techniques in this article will help you make sure that the people that see those pages are the people that you wanted to see them.

This article covers the "standard" way of protecting parts of your web site that most of you are going to use.

Note:

If your data really needs to be secure, consider using MOD SSL in addition to any authentication.

The Prerequisites

The directives discussed in this article will need to go either in your main server configuration file (typically in a < DIRECTORY > section), or in per-directory configuration files (`.htaccess` files).

If you plan to use `.htaccess` files, you will need to have a server configuration that permits putting authentication directives in these files. This is done with the ALLOWOVERRIDE directive, which specifies which directives, if any, may be put in per-directory configuration files.

Since we're talking here about authentication, you will need an ALLOWOVERRIDE directive like the following:

```
AllowOverride AuthConfig
```

Or, if you are just going to put the directives directly in your main server configuration file, you will of course need to have write permission to that file.

And you'll need to know a little bit about the directory structure of your server, in order to know where some files are kept. This should not be terribly difficult, and I'll try to make this clear when we come to that point.

You will also need to make sure that the modules MOD AUTHN CORE and MOD AUTHZ CORE have either been built into the httpd binary or loaded by the httpd.conf configuration file. Both of these modules provide core directives and functionality that are critical to the configuration and use of authentication and authorization in the web server.

Getting it working

Here's the basics of password protecting a directory on your server.

First, you need to create a password file. Exactly how you do this will vary depending on what authentication provider you have chosen. More on that later. To start with, we'll use a text password file.

This file should be placed somewhere not accessible from the web. This is so that folks cannot download the password file. For example, if your documents are served out of `/usr/local/apache/htdocs`, you might want to put the password file(s) in `/usr/local/apache/passwd`.

To create the file, use the `htpasswd` utility that came with Apache. This will be located in the `bin` directory of wherever you installed Apache. If you have installed Apache from a third-party package, it may be in your execution path.

To create the file, type:

```
htpasswd -c /usr/local/apache/passwd/passwords rbowen
```

`htpasswd` will ask you for the password, and then ask you to type it again to confirm it:

```
# htpasswd -c /usr/local/apache/passwd/passwords rbowen
New password: mypassword Re-type new password: mypassword
Adding password for user rbowen
```

If `htpasswd` is not in your path, of course you'll have to type the full path to the file to get it to run. With a default installation, it's located at `/usr/local/apache2/bin/htpasswd`

Next, you'll need to configure the server to request a password and tell the server which users are allowed access. You can do this either by editing the `httpd.conf` file or using an `.htaccess` file. For example, if you wish to protect the directory `/usr/local/apache/htdocs/secret`, you can use the following directives, either placed in the file `/usr/local/apache/htdocs/secret/.htaccess`, or placed in `httpd.conf` inside a < Directory "/usr/local/apache/htdocs/secret" > section.

```
AuthType Basic
AuthName "Restricted Files"
# (Following line optional) AuthBasicProvider file
AuthUserFile
"/usr/local/apache/passwd/passwords"
Require user rbowen
```

Let's examine each of those directives individually. The AUTHTYPE directive selects that method that is used to au- thenticate the user. The most common method is `Basic`, and this is the method implemented by MOD AUTH BASIC. It is important to be aware, however, that Basic authentication sends the password from the client to the server un- encrypted. This method should therefore not be used for highly sensitive data, unless accompanied by MOD SSL. Apache supports one other authentication method: `AuthType Digest`. This method is implemented by MOD AUTH DIGEST and was intended to be more secure. This is no longer the case and the connection should be encrypted with MOD SSL instead.

The AUTHNAME directive sets the *Realm* to be used in the authentication. The realm serves two major functions. First, the client often presents this information to the user as part of the password dialog box. Second, it is used by the client to determine what password to send for a given authenticated area.

So, for example, once a client has authenticated in the `"Restricted Files"` area, it will automatically retry the same password for any area on the same server that is marked with the `"Restricted Files"` Realm. Therefore, you can prevent a user from being prompted more than once for a password by letting multiple restricted areas share the same realm. Of course, for security reasons, the client will always need to ask again for the password whenever the hostname of the server changes.

Letting more than one person in

The directives above only let one person (specifically someone with a username of `rbowen`) into the directory. In most cases, you'll want to let more than one person in. This is where the AUTHGROUPFILE comes in.

If you want to let more than one person in, you'll need to create a group file that associates group names with a list of users in that group. The format of this file is pretty simple, and you can create it with your favorite editor. The contents of the file will look like this:

```
GroupName:    rbowen dpitts sungo rshersey
```

That's just a list of the members of the group in a long line

separated by spaces. To add a user to your already existing

password file, type:

```
htpasswd /usr/local/apache/passwd/passwords dpitts
```

You'll get the same response as before, but it will be appended to the existing file, rather than creating a new file. (It's the `-c` that makes it create a new password file).

Now, you need to modify your `.htaccess` file or <DIRECTORY> block to look like the following:

```
AuthType Basic
AuthName "By Invitation Only"
# Optional line:
AuthBasicProvider file AuthUserFile
"/usr/local/apache/passwd/passwords"
AuthGroupFile
"/usr/local/apache/passwd/groups" Require group GroupName
```

Now, anyone that is listed in the group `GroupName`, and has an entry in the `password` file, will be let in, if they type the correct password.

There's another way to let multiple users in that is less specific. Rather than creating a group file, you can just use the following directive:

```
Require valid-user
```

Using that rather than the `Require user rbowen` line will allow anyone in that is listed in the password file, and who correctly enters their password.

Possible problems

Because of the way that Basic authentication is specified, your username and password must be verified every time you request a document from the server. This is even if you're reloading the same page, and for every image on the page (if they come from a protected directory). As you can imagine, this slows things down a little. The amount that it slows things down is proportional to the size of the password file, because it has to open up that file, and go down the list of users until it gets to your name. And it has to do this every time a page is loaded.

A consequence of this is that there's a practical limit to how many users you can put in one password file. This limit will vary depending on the performance of your particular server machine, but you can expect to see slowdowns once you get above a few hundred entries, and may wish to consider a different authentication method at that time.

Alternate password storage

Because storing passwords in plain text files has the above problems, you may wish to store your passwords somewhere else, such as in a database.

MOD AUTHN DBM and MOD AUTHN DBD are two modules which make this possible. To select a dbm file rather than a text file, for example:

```
<Directory
"/www/docs/private">
AuthName "Private"
AuthType Basic AuthBasicProvider dbm AuthDBMUserFile
"/www/passwords/passwd.dbm" Require valid-user </Directory>
```

Other options are available. Consult the MOD AUTHN DBM documentation for more details.

Using multiple providers

With the introduction of the new provider based authentication and authorization architecture, you are no longer locked into a single authentication or authorization method. In fact any number of the providers can be mixed and matched to provide you with exactly the scheme that meets your needs. In the following example, both the file and LDAP based authentication providers are being used.

```
<Directory
"/www/docs/private">
AuthName "Private"
AuthType Basic AuthBasicProvider file ldap
AuthUserFile "/usr/local/apache/passwd/passwords"
AuthLDAPURL ldap://ldaphost/o=yourorg Require valid-user
</Directory>
```

In this example the file provider will attempt to authenticate the user first. If it is unable to authenticate the user, the LDAP provider will be called. This allows the scope of authentication to be broadened if your organization implements more than one type of authentication store. Other authentication and authorization scenarios may include mixing one type of authentication with a different type of authorization. For example, authenticating against a password file yet authorizing against an LDAP directory.

Just as multiple authentication providers can be implemented, multiple authorization methods can also be used. In this example both file group authorization as well as LDAP group authorization is being used.

```
<Directory
"/www/docs/private">
AuthName "Private"
AuthType Basic AuthBasicProvider file
AuthUserFile "/usr/local/apache/passwd/passwords" AuthLDAPURL
ldap://ldaphost/o=yourorg
AuthGroupFile "/usr/local/apache/passwd/groups" Require group
GroupName
Require ldap-group cn=mygroup,o=yourorg </Directory>
```

Beyond just authorization

The way that authorization can be applied is now much more flexible than just a single check against a single data store. Ordering, logic and choosing how authorization will be done is now possible.

Applying logic and ordering

Controlling how and in what order authorization will be applied has been a bit of a mystery in the past. In Apachen 2.2 a provider-based authentication mechanism was introduced to decouple the actual authentication process from authorization and supporting functionality. One of the side benefits was that authentication providers could be configured and called in a specific order which didn't depend on the load order of the auth module itself. This same provider based mechanism has been brought forward into authorization as well. What this means is that the REQUIRE directive not only specifies which authorization methods should be used, it also specifies the order in which they are called. Multiple authorization methods are called in the same order in which the REQUIRE directives appear in the configuration.

With the introduction of authorization container directives such as <REQUIREALL> and <REQUIREANY>, the configuration also has control over when the authorization methods are called and what criteria determines when access is granted. See Authorization Containers for an example of how they may be used to express complex authorization logic.

By default all REQUIRE directives are handled as though contained within a < REQUIREANY > container directive. In other words, if any of the specified authorization methods succeed, then authorization is granted.

Using authorization providers for access control

Authentication by username and password is only part of the story. Frequently you want to let people in based on something other than who they are. Something such as where they are coming from.

The authorization providers `all`, `env`, `host` and `ip` let you allow or deny access based on other host based criteria such as host name or ip address of the machine requesting a document.

The usage of these providers is specified through the REQUIRE directive. This directive registers the authorization providers that will be called during the authorization stage of the request processing. For example:

```
Require ip address
```

where *address* is an IP address (or a partial IP address) or:

```
Require host domain_name
```

where *domain name* is a fully qualified domain name (or a partial domain name); you may provide multiple addresses or domain names, if desired.

For example, if you have someone spamming your message board, and you want to keep them out, you could do the following:

```
<RequireAll>
Require all granted
Require not ip 10.252.46.165
</RequireAll>
```

Visitors coming from that address will not be able to see the content covered by this directive. If, instead, you have a machine name, rather than an IP address, you can use that.

```
<RequireAll>
Require all granted

Require not host host.example.com

</RequireAll>
```

And, if you'd like to block access from an entire domain, you can specify just part of an address or domain name:

```
<RequireAll>
Require all granted
Require not ip 192.168.205
Require not host phishers.example.com more.example

Require not host ke

</RequireAll>
```

Using <REQUIREALL> with multiple < REQUIRE > directives, each negated with `not`, will only allow access, if all of negated conditions are true. In other words, access will be blocked, if any of the negated conditions fails.

Firewall rules

Here's the most basic pair of rules to allow HTTP traffic, including both secure and non-secure:

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
iptables -A INPUT -p tdp --dport 443 -j ACCEPT
```

However, you can restrict the traffic even more. For example, you can allow incoming connections only to a certain interface:

```
iptables -A INPUT -p tcp --dport 80 -i eth0 -j ACCEPT
iptables -A INPUT -p tdp --dport 443 -i eth0 -j ACCEPT
```

Finally, you can restrict the traffic to a specific subnet, allowing only certain machines to connect:

```
iptables -A INPUT -p tcp --dport 80 -s 192.168.1.0/24 -i
eth0 -j ACCEPT
iptables -A INPUT -p tdp --dport 443 -s 192.168.1.0/24
-i eth0 -j ACCEPT
```

Advanced firewall rules

Sometimes, your Web server might not be an external client; it will connect to the Internet through a dedicated, firewalled gateway. This brings about several issues, which include the Network Address Translation (NAT) and port forwarding. Let's assume that your Web server is a local machine, with a local IP address. It serves both internal and external clients. The internal setup is rather simple. We need to make sure external clients can connect, too.

We will need to allow traffic destined to ports 80 and 443 of our external IP address (let's assume 1.1.1.1) to be forwarded to ports 80 and 443 of our Web server, which resides on a local address (192.168.1.128). Furthermore, interfaces are *eth0* for the Internet and *eth 1* for the intranet. Our setup will include several steps:

- We will have to "forward" our web ports, so that clients behind the gateway will be able to accept incoming communications.
- We will have to setup some sort of masquerading, only this time we will use methods slightly different from a typical DHCP server.

Port forwarding

Our gateway is configured to forward communications between internal and external hosts, with new, established and related connections permitted outbound and established and related connections permitted inbound. This is a great setup for a server-less network, but it won't do for Apache. Here are the original rules, which are used to setup the DHCP server to act as a gateway:

```
iptables -A FORWARD -t filter -i eth1 -m state -state
ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -t filter -o eth1 -m state -state
NEW,ESTABLISHED,RELATED -j ACCEPT
```

Now, we need these rules to make it work:

```
iptables -I FORWARD -p tcp -i eth1 -o eth0 -d 192.168.1.128
-dport 80 -m state -state NEW -j ACCEPT iptables -I FORWARD
-p tcp -i eth1 -o eth0 -d 192.168.1.128 -dport 443 -m state
-state NEW -j ACCEPT
```

If you want to tighten the rules some more, you can also specify the source ports:

```
iptables -I FORWARD -p tcp -i eth1 -o eth0 -d 192.168.1.128
-dport 80 -sport 1024:65535 -m state -state NEW -j ACCEPT
```

We have placed the rules on the top of the chain, so they would be processed before the existing rules, which only allow new outbound connections. Basically, these rules are sufficient if your gateway is servicing a number of local networks, all of which can fully resolve one another's IP addresses. They are not good enough for the Internet, though.

Destination NAT

Using masquerading the way we did when we configured the DHCP server might not be good enough for us, because it will point all traffic to the default network interface. We have to use a more sophisticated method, which is the DNAT. Now, we need to allow new incoming connections to our Web server to properly resolve to the right client, on the right ports.

```
iptables -t nat -A PREROUTING -p tcp -i eth1 -d 1.1.1.1
-dport 80 -to-destination 192.168.1.128:80 -j DNAT
iptables -t nat -A PREROUTING -p tcp -i eth1 -d 1.1.1.1
-dport 443 -to-destination 192.168.1.128:443 -j DNAT
```

Of course, you can use non-default ports on the Web server, like 8080 or anything alike, which makes the idea of port forwarding even more meaningful. To reiterate, IP masquerading is good enough for "normal" browsing, but servers behind the firewall also require port forwarding. Casual peer-to-peer (P2P) home users behind routers often have to do this to make their programs work.

Static NAT

If you have more than one publicly visible IP address, you won't be able to use IP masquerading. This is because masquerading forces all traffic to the default network interface on the firewalled gateway, resulting in a single usable external IP address.

However, it is quite likely that you will want to run your servers on separate hosts, with different both internal and external addresses, both to shape your traffic in a more orderly fashion and reduce the workload on specific hosts. To this end, you will have to use SNAT rather than IP forwarding.

The basic principle remains the same, except that you use separate external IP addresses for individual hosts, groups of hosts or the entire local network, as you see fit. In our example, we will demonstrate SNAT by creating a private rule for the Web server and a general rule for all other clients. Let's assume the Web server will use a public IP address of 1.1.1.1, while all other clients will use 1.1.1.2.

Web server rules:

These two rules are required to allow DNAT and SNAT for the client running the Web server. Please note that these two rules do not specify what kind of servers are running on the particular client. This grants you extra flexibility, if you need to run more than one server on a particular machine.

```
iptables -t nat -A PREROUTING -d 1.1.1.1 -i eth1 -to-
destination 192.168.1.128 -j DNAT iptables -t nat -A
POSTROUTING -s 192.168.1.128 -o eth1 -to-source 1.1.1.1
-j SNAT
```

As said, the forwarding rules from before remain valid, both the specific rules for the Web server, which permit new inbound connections, and the general rules, which permit only new outbound connections.

General rules:

This rule applies to all local network clients, trying to communicate with the external network.

```
iptables -t nat -A POSTROUTING -s!      192.168.1.128 -o
eth1
-to-source 1.1.1.2 -j SNAT
```

Enable Web server on startup

You will most likely want your Apache server to run on startup. The simplest way to enable this is to use the *chkconfig* utility.

```
chkconfig -levels 5 httpd on
```

Security

Web server security is one of the most important things in your setup. If your server becomes compromised, you run the risk of serving malicious, fraudulent or simply tasteless pages to hundreds and thousands of your visitors. Furthermore, you risk exposing the privacy of your clients and users. Forum names and passwords, email addresses, sensitive records, credit card numbers, and other information could all be leaked out, creating an identity theft nightmare. It is paramount that you keep your Web server in tiptop shape at all times. This requires lots of work, attention and responsibility and is not a trivial task. Running a good server takes time and patience.

It is also important that you be constantly aware of what goes about on your server. If you have several users uploading material to their individual directories, you are advised to make sure that they do not post potentially dangerous content. Most Linux users are oblivious to the web threats, but a large percentage of Windows users have a hard time with sites loaded with malicious payload. As the server owner and administrator, it is your responsibility to make sure that your visitors are not at risk. Now, let us review some of the most crucial settings that you should pay attention to make sure both your server and your clients are secure.

Updates

Keep your server up to date at all times. Make sure you patch new vulnerabilities instantly. You are advised to subscribe to the Apache HTTP Server Mailing Lists for information about new bugs, updates, features, and more.

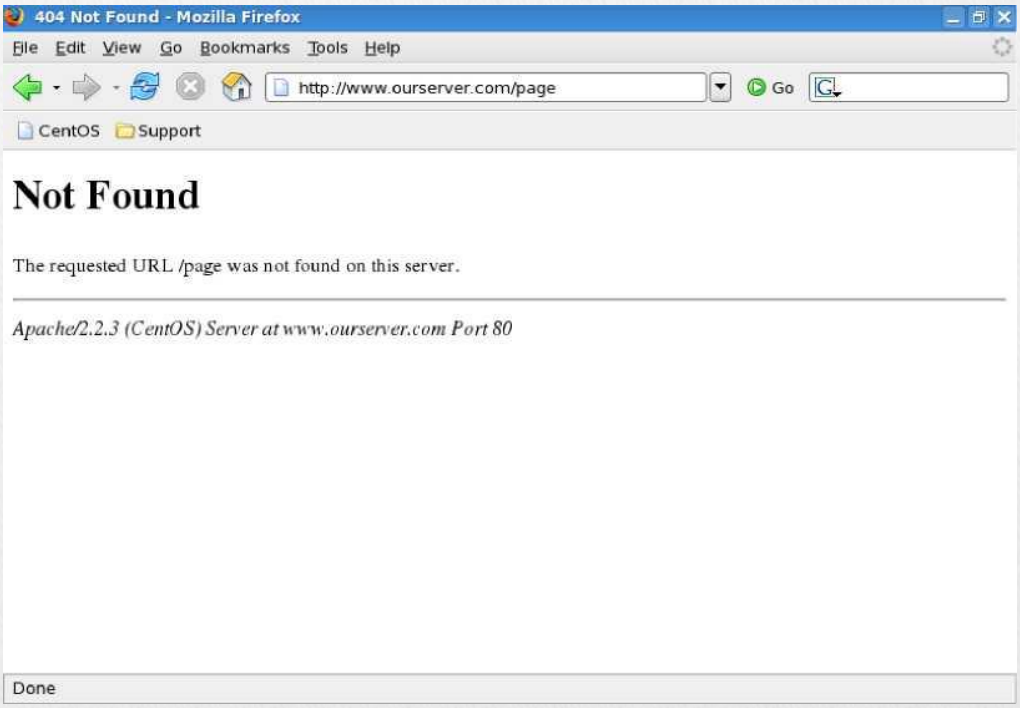
Hide your server version

This is a "security through obscurity" measure. Nevertheless, it does not hurt to use it. At the very least, this will annoy and delay a potential attacker, by making his attempts to harvest server information more difficult. To remove server information, you will need to use these two directives:

```
ServerSignature Off
ServerTokens Prod
```

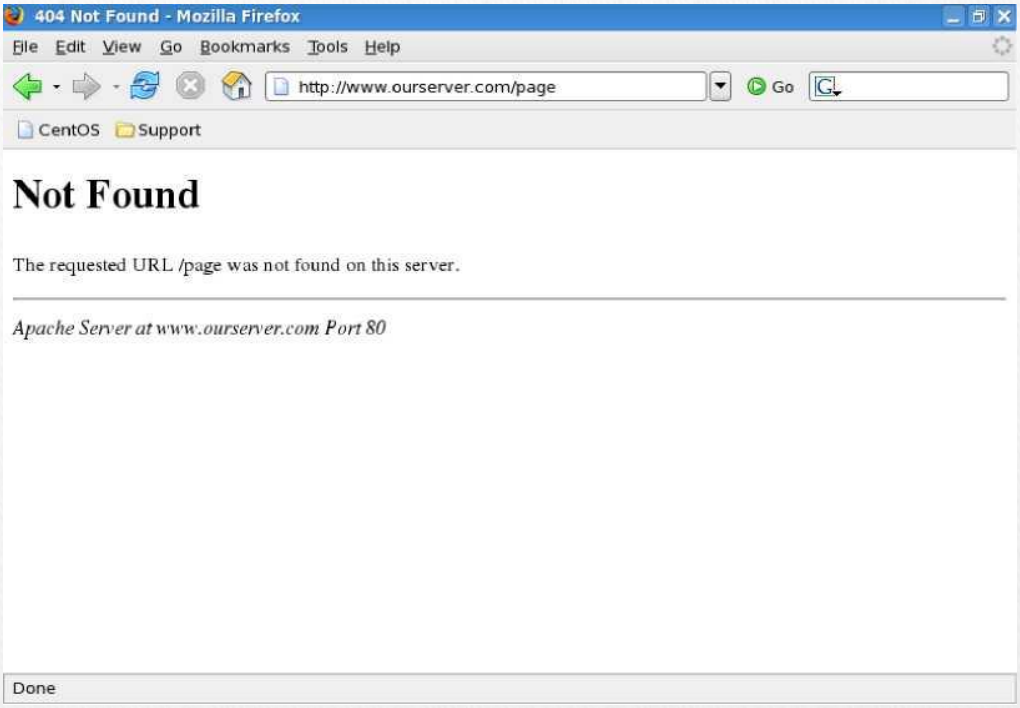
The first directive, **ServerSignature**, will remove the server version information from the pages generated by the server, like error pages (403 Forbidden, 404 Not found), directory listings and others. The second directive, **ServerTokens**, will change the server's HTTP Response Header. By default, with the directive set to *OS,* the header will disclose both the version and the operating system. Set to *Prod,* the header will merely report *Apache.* Here's an example without these directives.

Figure: Server version displayed



And here's with the directives in place. The version is disguised.

Figure: Server version hidden



Logs

You should check your logs at least daily. This may be tedious and boring, but it is vital that you discover any potential breaches as quickly as possible. Keep an eye on things and look for suspicious directories and files.

Permissions

Badly implemented permissions can ruin your entire security. It is critical that you make sure the executables, configuration files, logs, access files and private keys are located outside the public HTML directories and writable only by root. The web pages should be readable and possibly executable by your visitors but only writable by their respective owners. Let us review the necessary permissions:

Table: Apache Web server recommended file permissions

Location	Permissions
/usr/sbin/httpd	F: 511
/etc/httpd	D: 751
/etc/httpd/conf	D: 751, F: 644 / 600
/etc/httpd/conf.d	D: 751, F: 644 / 600
/etc/httpd/logs	symbolic link, 755/711
/etc/httpd/modules	symbolic link, 755/711
/etc/httpd/run	symbolic link, 755/711
/usr/lib/httpd/modules	D: 751, F: 644 / 600
/var/log/httpd	D: 751, F: 644 / 600
/var/run	D: 751, F: 644 / 600
/var/www/html	D: 755, F: 755

If you are really paranoid, then you should ONLY allow root access to the binaries and configuration files. It really depends on your setup and needs. Last but not the least, let's not forget that system files MUST be owned by root.

Access to root (*/*)

You m ust not allow anyone to access the *root* directory. Therefore, you should implement a default deny policy for the root and all sub-directories and then partially allow access to specific locations, like the public HTML directories of your users.

```
<Directory />
   Order Deny, Allow
Deny from all
</Directory>
```

AllowOverride

This directive specifies if options used in the *.htaccess* files can conflict (and thus override) the settings configured for the particular directory. In general, you should set this directive to *none* and only perm it specific tasks to a small number of trusted users. If you lease your server to numerous clients who must have some sort of protection for their content, then you can allow them to use the *.htaccess* file for authentication, as we have shown before (4).

```
AllowOverride none
```

Disable public access to .ht files

You must not allow any public user to be able to load the *.htaccess* file through the browser window. Furthermore, the *.htpasswd* file, which contains the user names and passwords, must also be protected from public access. Again, we have discussed this before, but it doesn't hurt to repeat it.

```
<Files   "*.ht" >
Order allow, deny
Deny from all
</Files>
```

Dynamic content

You need to be very careful with dynam ic pages and scripts, since they allow server to perform a variety of operations that static HTM L files cannot do. You should think twice before you allow any user to run scripts from his/her own public directory. In general, scripts are only allowed in special directories, as defined by the *ScriptAlias* directive.

Disable CGI

You should disable scripts in user directories. If you really must permit them, use the *Options +ExecCGI* directive.

```
Options -ExecCGI
```

Disable Server Side Includes (SSI)

SSI is a simple scripting language that allows web servers to display variables or execute other programs from within web pages. This introduces a significant load on the server and poses a security risk. You should not allow SSI to function on your server unless absolutely necessary.

```
Options -Includes
```

Disable unnecessary modules

By default, Apache is very permissive when it comes to the functionality it offers. You should comment out any modules you do not need or wish to use from the */etc/httpd/conf/httpd.conf* file.

Use ModSecurity (mod_security) module

ModSecurity is a powerful Web Application Filter (WAF), which allows you to greatly enhance the security of your server by to detecting and preventing attacks before they reach web applications. ModSecurity can perform a variety of tasks, including detection of HTTP protocols violation, detection against common web attacks, detection of bots and crawlers, detection of Trojan horses, filtering based on existing rulesets, policies or regular expressions, and more. The application relies on generic rules to detect and prevent exploits and does not rely on blacklists or signatures.

Furthermore, it will not throttle the traffic throughput. And best of all, ModSecurity is very easy add to an existing and running Apache server. Setting up and configuring ModSecurity is outside the scope of this document.

Chroot Jail

Like with many other services, it is possible to "sandbox" *httpd* to run in a virtual prison. The service will think its files are located in the default directories. In reality, they will reside inside a Chroot Jail, which will mimic the layout of the real directories, except that many of the files normally found under the real root won't be there, preventing possible privilege escalation risks. For more information, please read: Apache in a chroot jail.

Secure web server only

The secure Web servers will offer their certificates to any client that asks for them. This means that you do not care about who your clients are. However, if your setup also requires that only a limited number of clients be allowed to access the secure content, you m ight consider using any one or several of the following methods: Kerberos, firewall rules, TCP wrappers, *allow* & *deny* directives, *.htaccess* files, client certificates, and more.

Different DocumentRoot

You are advised to use a separate directory for the secure pages. This allows you to fully separate normal content from secure, privileged data and might even mitigate potential exposure in case of an attack.

Permissions

The *server.crt* and the *server.key* file must only be readable by root. You should even disallow the root user from making any changes to the files. Set the permissions for these two files to 0400. The permissions for the *ssl.conf* should be in line with your policy, which should be either640 or600.

Duration of certification

Normally, certificates are issued for a year. However, if you are running an ultraparanoid setup, you may want to make the certificates expire after only a few weeks. This means that potential attackers will always have only a limited access to data for a short period of time, in case they succeed in decrypting your ciphers. You will force them to work all over again, trying to decode your data. Needless to say, you will most likely use self-signed certificates for this type of work.