Serving websites as unique users off a single server

Made to Serve

Creating secure websites with their own privileges on a single server. *By Kurt Seifried*

Most virtualization now focuses on the server level. This approach makes a lot of sense because you can use whatever applications you want and pretty much any operating system, and you don't have to worry about whether or not the applications will behave appropriately. The problem for virtualizing most services is that the protocols (SSH, FTP, etc.) are stateful, and very few are designed with load balancing and failover specifically in mind. Sharing state between servers is a non-trivial problem; sharing the state is simple, but do you take action before or after you have confirmed that the state was shared?

If you take action before confirming the state was shared and the server fails, you might end up with a mismatch when the client tries to talk to the other server. If you share state and confirm it was shared before taking action, you introduce latency: Every single action a user takes will need to be acknowledged by all the servers before it can be processed. Luckily, one of the most popular protocols is not only stateless but also supports virtualization (based on host names). HTTP 1.1 specifically supports not just requesting objects but requesting them from a specific host. This is old news. Literally. HTTP 1.1 was released and support for it become widespread in the mid-1990s, so why didn't everyone virtualize their websites at the application level using Apache HTTPD to serve hundreds or thousands of websites from a single server? Well, we did. This is exactly how hosting providers were able to bring rates down from hundreds of dollars per month to single digits.

However, widespread need for SSL (because people wanted to sell stuff online, protect user logins, etc.) broke all this. At the time, SSL and TLS had a chicken and egg problem: You had to establish a connection first, but until the connection was established you couldn't specify which host you wanted to connect to. The short-term solution was to bind an IP address for each site to the server and serve each site off of a unique IP address (which is exactly what happened before HTTP 1.1 to serve multiple sites off of a single box using HTTP).

Server Name Identification

Server Name Identification (SNI) is an extension to SSL and TLS that allows the client to request a specific certificate from a server (e.g., for *www.example.org*). If the server has such a certificate (or perhaps a wildcard certificate like **.example.org*), it replies with it. The client then examines it, and, if accepted, it establishes an encrypted connection with the server. The client can then request content as it normally would and get the specific site it wants. SNI was great because it allowed you to stop pay-

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

ing providers the extra US$ 5 or US$ 10 a month for each IP address needed to host multiple SSL-encrypted websites. Additionally, when you combine SNI with wildcard certificates, you can quickly create secure websites with arbitrary names under your domain, reducing the amount of time and money you need to spend dealing with certificate authorities.

To use SNI, you'll need a recent copy of Apache HTTPD (2.2.12 or later) and OpenSSL (0.9.8j or later) [1]. In general, if your server is up to date, this shouldn't be a problem (this has been well supported since 2009-2010). Configuration is easy: Just add the `NameVirtualHost` and `SSLStrictSNIVHostCheck` directives and then specify one or more virtual hosts:

```
Listen 443
NameVirtualHost *:443
SSLStrictSNIVHostCheck off

<VirtualHost _default_:443>
 ServerName default.example.org:443
 ... directives go here
</VirtualHost>

<VirtualHost *:443>
ServerName www.example.org:443
 ... directives go here
</VirtualHost>
```

Additionally, I like to specify `SSLStrictSNIVHostCheck` as *off*. If it is set to *on*, any clients that do not support SNI will be unable to connect at all. In general, this isn't an issue because any web browser old enough to lack support for SNI is pretty much obsolete at this point. However, I have noticed that several load-balancing and web health-checking devices and services do not support SNI properly, so to err on the side of caution, I allow clients that do not support SNI and direct them to a "default" website that notifies people they should update their web client. This also makes it easy to check the logfiles for clients that don't support SNI.

Separating the Websites

After I cheerfully deployed SNI and went back to having all my websites served off of a single web server, I realized I had a problem. Apache HTTPD runs as user *apache* by default. This meant that I had multiple sites (mostly WordPress and MediaWiki sites) all running as the same user. They all had writable cache directories and upload directories, as well as the WordPress upgrade directory. The result was that a compromise of one site would allow an attacker potentially to modify the cache or uploaded files in the other sites (meaning a single site getting compromised would result in a huge mess). So, how best to separate all these sites and have them run as unique users so they can't modify each other?

My search first led to Apache2-MPM-ITK [2], which is a fork of the standard MPM (multiprocessing module) that allows you to configure each virtual host to run as a separate UID/GID. However, it is not actively maintained (the last update was March 2011) and has not been extensively tested.

Separate Servers

My next thought was to use separate servers by setting up a copy of Apache HTTPD for each site running as a specific user on a different port (e.g. 8000, 8001, 8002) then setting up a single "master" to listen on ports 80/443, having it handle the SSL connections, and then proxying the request for a site to the appropriate web server on port 8000, or whichever one it is. The benefit of this is that you not only separate out the user accounts for each website, but you also can apply resource limits to users. Setting up the master process is easy because Apache supports proxy directives within a `VirtualHost` directive [3].

suPHP

In some cases, setting up a separate server for each user, especially for low-traffic sites, will be resource intensive. Another option for splitting up PHP-based sites so they run as separate users is suPHP [4]. suPHP is very simple to install and set up (just install the package). One really nice feature is that you can enable or disable suPHP by virtual host, so if you want a mix of "normal" sites and per-user sites, you can do this. To get suPHP to work properly, you will most likely need to set `allow_file_group_writeable` to *true*; otherwise, any PHP files that are group writable will result in an error message.

suPHP offers three options for running PHP files as a specific user and group. The first is by `owner`. Simply put, the PHP script is run with the user and group permissions the file is set with. To prevent a file being executed as root, for example, suPHP supports a `min_uid` and `min_gid`, so you can ensure that files are not executed with too much privilege.

The second option is called `force` and allows you to specify the user and group the PHP files are run with via the Apache configuration file. To use this method, simply set the `suPHP_UserGroup` variable within a `Directory` or `Location` directive.

With the third option, called `paranoid` (which is the default), suPHP uses the user and group the PHP file is owned by to run the PHP file, and it also checks the Apache configuration: If `suPHP_UserGroup` is set, it must match the file ownership.

Of course, once you have the websites all locked down to separate users, you will either have to give the users access to these accounts, or you can use Linux Access Control Lists (ACLs) to give specific users access to their files.

To enable ACLs, you need to modify `/etc/fstab` to include an {{acl}} at the beginning of the options for the filesystems on which you need ACLs. Once remounted, you'll be able to set specific users and groups on files using the `setfacl` command:

```
setfacl -m u:someuser:rw somefile.php
```

For more information, the man pages for `setfacl` and `getfacl` contain all the details you need.

Even if you're not running sites for multiple users, it's generally a good idea to split up the privileges that each site runs as. The most effective method is to use multiple servers, but if you want to avoid additional overhead, I suggest using suPHP. ■■■

INFO

[1] Name-based virtual hosts: *http://wiki.apache.org/httpd/NameBasedSSLVHosts*

[2] Apache MPM-ITK: *http://mpm-itk.sesse.net/*

[3] Apache VirtualHost examples: *http://httpd.apache.org/docs/2.4/vhosts/examples.html*

[4] suPHP: *http://www.suphp.org/*