Building a Secure Web Server

Burke Libbey

April 10, 2008

Abstract

With the rate at which the internet is growing, it is becoming increasingly important to secure all aspects of an internet-connected machine. This paper will outline some steps that can be taken to prevent unauthorized access to a Linux machine at all levels. By following these techniques, a reasonably secure machine is simple to set up and maintain.

Introduction

Security for a web server can be sliced up into four general areas: Physical, Local, Remote, and Application security. Physical security protects against people with physical access to the machine, local security protects against authenticated users, remote security protects against unauthenticated users, and application security protects your application data. I'll go though each area in detail, outlining things that a malicious user may try to exploit, and how to protect your server against them.

Physical Security

Many system administrators neglect physical security somewhat, spending more effort on software-based security. While it is true that physical security breaches are much less common than other types, they have the potential to have much more severe implications. Physical access to your machine should always be restricted to the people that need access. Room locks and case/rack locks are always a good idea.

If an attacker manages to gain physical access to the machine, they can reboot it with a Linux LiveCD and have full access to your filesystem. This will typically allow them to change your root password (not a good thing) or whatever else they decide to do. To counter this, it's a good idea to set a BIOS password and remove everything but your primary boot drive from the BIOS startup list. It's worth keeping in mind that many BIOSes have well-known override passwords, and the rest can be reset by a determined hacker regardless.[1] Keep this in mind when purchasing locks.

Local Security

Local security refers to protecting the system from malicious and/or careless but legitimate users of the system. In the context of a web server, this scope is fairly narrow, since there will typically only be a handful of users, and the majority of their activity is limited to uploading files.

Setting correct file permissions is very important for any Linux system. You should always check that users cannot access anything they aren't supposed have access to. For example, users should not be able to execute files in '`/sbin`', '`/usr/sbin`', or '`/usr/local/sbin`'. For the most part, they shouldn't be allowed to write to any files outside of '`$HOME/`'. One notable exception to this rule will be '`/var/www`' or some subdirectory thereof to allow access to the web root. By correctly setting file permissions and managing user groups, you can go a long way to ensuring protection from local exploits.

Privilege escalation exploits are occasionally found in software that runs as the root user. These allow an unprivileged user to execute arbitrary commands as a more privileged user. This can be catastrophic, but there's not much you can do about it, other than carefully selecting the software you install on the system, and updating it regularly. When an exploit is found, it is usually fixed quickly. By carefully evaluating the requirements of your server, you can trim down the software you install to a bare minimum.

Remote Security

Remote Security is the most often assaulted front for web servers. They're usually reasonably secure physically, with a small group of trusted users, but they're almost always connected directly to the internet, with a domain name advertising their presence. Most web servers are frequently probed by would-be attackers. By setting up proper remote security, it isn't difficult to prevent most attackers from gaining access.

Far and away the most common 'exploit' you'll have to worry about is people (read: bots) trying to brute-force their way past your SSH authentication. You'll probably see this if you examine the contents of '`/var/log/sshd/current`', or wherever your SSH logs are kept. The simple solution to this problem is to blacklist IPs after they've guessed incorrectly a certain number of times. A package called '`denyhosts`' will take care of this for you with about a minute's worth of configuration.[2]

A firewall is also a very important part of any secure server. This is commonly accomplished on a Linux system through '`iptables`'. `iptables` lets you filter both incoming and outgoing packets. For our web server context, we are only concerned with the former.

If a local user is running a program that listens for connections on a given port, it's typically not desirable to have that port be accessible from the internet (at least, not from the system administrator's point of view). You should configure `iptables` to drop all packets by default, making exceptions for explicitly allowed ports.

A malicious user can craft 'broken' packets that cause poorly-designed applications to behave unexpectedly. This is another problem firewalls can help with, by automatically dropping any invalid packet, even if it's sent as part of an established connection. `iptables` categorizes packets based on their 'state'. Invalid packets have a state of '`INVALID`', while good packets are typically '`NEW`', '`RELATED`', or '`ESTABLISHED`'.[3] If we set the firewall to automatically accept any packet with the '`RELATED`' or '`ESTABLISHED`' state, then specify that each unblocked port accept packets of state '`NEW`', '`INVALID`' packets will be dropped by default.

```
Chain INPUT (policy DROP)
target  prot opt source   destination
ACCEPT  all  --  anywhere anywhere    state RELATED,ESTABLISHED
ACCEPT  tcp  --  anywhere anywhere    state NEW tcp dpt:ssh
ACCEPT  tcp  --  anywhere anywhere    state NEW tcp dpt:http
```

The above configuration is a good place to start. As with anything else related to security, it's important to configure your firewall from a whitelist perspective – drop all packets by default, but let them through if they match certain criteria. Blacklist policies are usually susceptible to cleverness.

Application Security

Application security usually becomes an issue in stateful web applications (ie. should be a concern for applications that interface with a database, not so much for static HTML sites). The most common and dangerous exploit is SQL Injection. CSRF and XSS are also widely used, and I will cover them as well.

SQL Injection is possible when the back end uses user data (usually form submissions) to build a SQL query. A malicious user can craft a string to run any arbitrary command on the database if he can correctly guess a few simple details of the environment.

Picture a username and password submission form. If the application is naïve, it likely generates the SQL query something like this: "`SELECT * FROM users WHERE name='$username' AND password='$password'`". This is not good.[4] If instead of a password, the user enters something like "`asdf' OR 1=1`", the query will return all users, which will likely validate the user.

Even worse, if the user enters something to the effect of "`asdf'; DROP TABLE users;-- `", an unsecured application will immediately delete every user record in the database. This is obviously undesirable.

There are a few simple ways to secure against SQL Injection. First and foremost, use your language's built-in string escape commands to bullet-proof something for entry into a SQL database. In PHP, it's usually sufficient to run '`addslashes()` on a value before entering it into the database and '`stripslashes()`' when retrieving it. Another better solution – though not mutually exclusive – is to use bind variables. Using this method, the above query would look like "`SELECT * FROM users WHERE name=?  AND password=?`" *(Sidenote: store passwords as hashes, not in cleartext).* A second parameter would then be passed to the query function containing the user input in an array. Using this method, your database backend will intercept any attempts at SQL Injection.

XSS stands for Cross-Site Scripting. When a user is allowed to submit information that will eventually be displayed back to other users, a malicious user can employ XSS by embedding javascript in the information. This javascript will usually submit data about the logged in user available only to scripts run from the same domain to some external host. Depending on the data stored in your cookies, this can be a very bad thing. At the very least, your users may be redirected to some other site, due to unwanted javascript content. To protect against XSS, just remove script elements from the user input and keep your cookie data to a minimum. Usually a session id suffices.

CSRF stands for Cross-Site Request Forgery. Let's say you're logged in on a web service. Facebook, for the sake of argument. Let's pretend Facebook had a page you could go to that would remove everyone from your friends list. Obviously, this would only be available when you're logged in. It would be pretty trivial, however, for an external site to redirect you to this page without any interaction from you. Further, it could make use of the `XMLHttpRequest` javascript object to load that page in the background without you even realizing what happened.[5] This all relies on the cookie stored on your machine indicating that you're logged in on Facebook. The only way to prevent CSRF is to include a session id in the cookie and send it as part of every request. The cookie should be accessible only from the originating domain, so external sites won't be able to craft a URL including your session id. If you can get in the habit of requiring session ids, CSRF becomes a non-issue.

Conclusion

While it's never possible to be completely secure, you can achieve an acceptable level of security by following the guidelines set out above. It won't be enough to stop most determined and skilled of hackers (nothing ever is...), but it will surely be enough to stop your machine from becoming part of a botnet overnight.

References

[1] Unknown, "Removing a bios - cmos password." http://www.dewassoc.com/support/bios/bios_password.htm, 2000. [Online; accessed 09-Apr-2008].

[2] Unknown, "Denyhosts frequently asked questions." http://denyhosts.sourceforge.net/faq.html, 2008. [Online; accessed 09-Apr-2008].

[3] O. Andreasson, "Iptables tutorial 1.2.2." http://iptables-tutorial.frozentux.net/iptables-tutorial.html, 2006. [Online; accessed 09-Apr-2008].

[4] Various, "Sql injection." http://en.wikipedia.org/wiki/SQL_injection, 2008. [Online; accessed 09-Apr-2008].

[5] Various, "Cross-site request forgery." http://en.wikipedia.org/wiki/Cross-site_request_forgery, 2008. [Online; accessed 09-Apr-2008].