Denial of service made easy

APACHE HTTPD

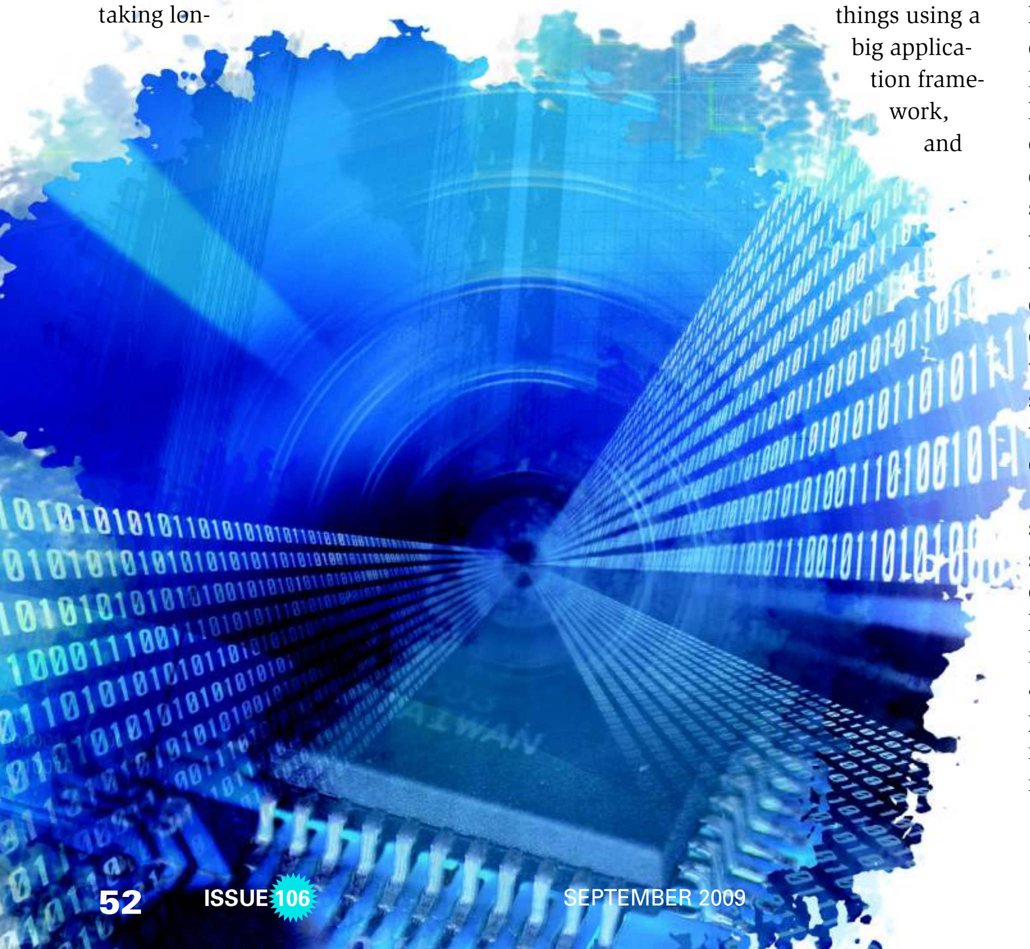
A slow death for the default configuration. **BY KURT SEIFRIED**

In a thought experiment that might not be historically accurate, but is close enough, you want to build a web server, so you program a socket-based server. A client connects, requests a file, you send the file, the client disconnects, and everyone is happy. The total transaction time is around 300 milliseconds. But you get a bug report that some guy's web server keeps getting slower and slower until it eventually dies. What to do? After diagnosing the problem, you figure out that some web clients aren't handling the connection properly. They create a connection but then do nothing with it, leaving it open indefinitely. The result is that the server slowly runs out of available connections until it can no longer service new requests. The fix is easy: Just stick in a default *TimeOut* directive so actions that tie up resources (like a connection) eventually get killed off if they aren't actually doing anything.

This works pretty well for a while, but you start noticing that web pages are taking longer and longer to load because they are no longer a single page but contain images, CSS files, JavaScript files, and so on. So being a smart programmer, you figure to let the client keep the connection alive and re-use it for additional requests. This solution is good because it avoids the setup cost of the connection (a TCP three-way handshake [1] takes time). Now the second (and third, and fourth) file request is a lot faster, and everyone is happy. Because you learned from your last mistake, you put a *KeepAliveTimeout* directive to prevent problems. Life is good, people download your web server, and pretty soon 60% or so of the web is using the Apache HTTPD server.

Sane Defaults

Defaults are one of the most annoying problems because, quite simply, no defaults work well for everyone. Site A could be serving millions of small images, whereas site B wants to serve things using a big application framework, and sites C through Z aren't quite sure what they're doing. However, the web server seems to work pretty well, so why worry about the defaults? The operating system vendors don't really want to change the defaults on all the software they ship unless they have a good reason to do so because it's one more thing to do. Additionally, it means your software could behave unexpectedly, resulting in support calls that no one wants to deal with, especially if they're a volunteer-driven organization. So you'll just have to trust the software project to choose sane defaults, which is probably for the best because they understand the software and what twiddling the knobs can break. Therefore, you end up with defaults that work for most people, assuming nothing strange happens – like 1,000 or more clients on really slow network connections hitting your server at the same time.



So What Happens When …?

If 1,000 clients on really slow network links (or one client with 1,000 connections pretending to be on a slow network link) hit your server all at once, it turns out that your server stops working. Rather, it still works, but it is limited by how many connections it can serve, So even if serving 1,000 slow connections doesn't take a lot of resources, your server has no more available connections to serve other legitimate clients. To the world, your server appears to be dead. This situation is a problem because a user with a clever piece of software like Slowloris [2] can attack a large site from a single computer on a relatively small network link (i.e., DSL or a cable modem).

But shouldn't this be easy to fix by simply limiting how many connections a single IP address or a network block can create and hold open? If you set this limit low, you might block users that are forced to use web proxies. AOL, for example, forces all users through web proxies (which saves them a ton of money on bandwidth). In many legitimate cases, a single IP address or a group of IP addresses in a small network might open a lot of connections (e.g., those IPs might be every single AOL user in the American Midwest). The problem here is that you need to find a limit that will prevent damage to your server yet is high enough that it is unlikely to block legitimate users. My advice is to set the limit as high as possible (i.e., where it affects your server but doesn't completely kill it) to avoid blocking as many users as possible.

Figure 1: Slowloris is named for a slow-moving primate with a very tight grip.

One generic way to rate limit connections per IP address is to use the iptables rate-limiting facility, which can be done selectively on single ports. Also, you can specify a block of time and the maximum number of connections that can be established in that time frame. The following code creates a 60-second block with a maximum of five connections. On the sixth or more, it will simply *DROP* the packets, causing the client to retry. Once an earlier connection closes, the new one will be allowed.

```
iptables -I INPUT -p tcp --dport 80 ⤦
  -m state --state NEW -m recent --set
iptables -I INPUT -p tcp --dport 80 ⤦
  -m state --state NEW -m recent ⤦
  --update --seconds 60 --hitcount 6 ⤦
  -j DROP
```

Of course, a determined attacker will simply use more machines, eventually saturating your attempt to rate limit, but you can make their job significantly harder.

Trading Performance for Survivability

If you're unwilling or unable to spend money and deploy more hardware and software to soak up denial of service attacks, chances are you can trade performance for added survivability. To deal with the Slowloris attack, the quickest and somewhat effective action is to set the *TimeOut* value from its default (typically 300 seconds, or five minutes) to a much shorter five seconds – or less if need be. Additionally, to prevent abuse of http keep-alive, you can simply disable it by setting the *KeepAlive* directive to off. Please note that neither of these workarounds will actually fix the problem in a very meaningful way, but against attackers with limited means, it should help [3]. Also note that this attack doesn't just affect the Apache HTTPD server: The Squid web proxy and a number of other web servers are also vulnerable to the Slowloris attack.

Long-Term Needs

Although you will never be able to completely mitigate the risk and effect of denial of service attacks (in a worst-case scenario, a botnet sends legitimate requests that soak up your available resources), you can build systems that can survive small attacks and force attackers to spend more resources on attacks, which you hope will discourage them. The best long-term solution appears to be building better rate-limiting functionality into applications and, most importantly, allowing these applications to change their settings as needed if they come under attack (e.g., reduce the connection timeout as they become busier and start kicking off slow hosts if they get maxed out). In this way, you will give applications the best chance of surviving not only denial of service attacks, but heavy workloads. For example, as I write this, CNN.com is looking slightly broken, probably because of Michael Jackson's death, but it's loading the page text so as not to be completely useless.

An example of this is a third-party patch from Andreas Krennmair [4]. His patch adds load percentage monitoring with the use of the Apache HTTPD scoreboard. As load increases, the timeout is adjusted. At 60% load, it halves the timeout; at 70%, it quarters it; and so on. Although simplistic, it is a good example of building some intelligence and a "survival" instinct into the application; unfortunately, it cannot close existing connections, so with enough resources, an attacker can still cause the machine to become unresponsive.

To Infinity and Beyond!

The true irony of these slow denial of service attacks that take up connection handling resources is that they don't actually cause the server to run slowly in most cases. They simply prevent legitimate clients from being able to connect to the server because no connections are available. And if any do become available, the attacker can aggressively attempt to connect to them, beating legitimate clients. The additional benefit to fixing applications so that they can deal gracefully with these denial of service attacks is that it will also help them handle higher loads of legitimate traffic – a win for everybody. ■

INFO

[1] TCP three-way handshake: *http://en.wikipedia.org/wiki/Transmission_Control_Protocol*

[2] Slowloris http DoS: *http://ha.ckers.org/slowloris/*

[3] Apache Security Tips: *http://httpd.apache.org/docs/trunk/misc/security_tips.html#dos*

[4] Anti-Slowloris patch for Apache HTTPD: *http://synflood.at/tmp/anti-slowloris.diff*

THE AUTHOR

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.