Making sure your application is available

RATE LIMITING

Maybe we should listen to those network guys. **BY KURT SEIFRIED**

A few months ago I wrote about the Slowloris attack on web servers [1], the short version of which is: Attackers connect and hold connections open, using very few resources on their end, but soaking up all your available sockets and preventing any legitimate users from connecting. Since then, a number of other denial-of-service types of attack have been announced against web servers, web applications, and other services. This got me thinking: How can programmers deal with these issues in a generic way to reduce their impact?

The answer, of course, is rate limiting. The funny thing is that in the network world we live in, rate limiting and flow control are critical elements of IP-based networks that have been worked on literally for decades, and a number of great solutions have been found for the various problems. Unfortunately, most application-level programmers are unaware of this work or simply don't know why you would want to bother with rate limiting and flow control (and it's not like there isn't enough work dealing with feature requests and bug reports).

Examples of Attacks

Rate limiting is just as important as how you rate limit. One example of an attack is using search functionality with multiple expressions to consume large amounts of CPU and memory, as well as database resources. Another classic attack against FTP servers is to execute a command such as *ls */../*/../*/../*/../*/../*/../*/..*, which causes an insane number of directory lookups and, in some cases, could even cause the server to crash.

Airline reservation systems let you select a seat, then they hold it and make it unavailable to others for a small period of time (e.g., 10 to 15 minutes) to make sure someone else doesn't buy it while you pay for it. An attacker can repeatedly start a transaction to buy seats on the plane but never finish them, preventing anyone else from reserving a seat.

Spammers that crawl through your website automatically can make an entire copy of your content so they can put it on their website with ads and make some money. Or you might simply want to ensure that your paying customers have preferred access to the servers that also host your free users.

Where to Rate Limit

Once you've determined what you want to rate limit, you need to figure out where to do it. If you're worried about people screen-scraping a website, for example, it does no good to limit the number of new connections per IP address if you have HTTP Keep-Alives enabled (allowing clients to make more than one request per connection to the web server). Similarly, if your biggest concern is weak passwords on user accounts (and thus the support issues of dealing with hacked accounts), you might simply want to limit the effectiveness of brute force password-guessing attacks yet allow users to view as many web pages as they want once they are logged in.

Generally speaking, you will want to rate limit within the application because this gives you the most flexibility and control. However, if your concern is with people connecting to the application (i.e., it has a significant start-up time) or you do not have the ability to modify the application (it's closed source or simply too big and convoluted to be modified easily), you might want to consider proxying access or using an additional layer to protect the system.

The Problem with Leaky Buckets

So how exactly can you rate limit something? The simplest way is to use the leaky bucket [2] algorithm (Listing 1). With this fairly simple program, you simply define an acceptable rate – such as one search every 12 seconds (or five per minute). As work comes in, it is placed in a "bucket." At a rate you determine, work is taken out of the bucket. If the bucket is full (i.e., work is coming in faster than it can be processed), you simply discard the new work. This ensures that work being processed never exceeds a maximum amount.

But this scheme has a significant problem: What if a client needs to execute more than 10 searches quickly and only does this once in a while? Almost immediately they're going to get annoyed with you when their sixth search fails and they have to wait a minute to do the rest. So how can you deal with bursty behavior, but still rate limit what people are doing to prevent serious damage?

The Advantage of Token Buckets

The token bucket [3] can handle bursting traffic (busy now, quiet later) by allowing a certain rate of traffic (say, five searches every 60 seconds) and allowing those searches to happen in a period of

one second, then making the user wait 60 seconds before he can do any more searches.

The algorithm can be further modified to include a higher maximum capacity, allowing up to 10 tokens to be stored, for example, with a rate limit of five per minute, allowing users to initially do 10 searches in the first minute but only five per minute afterward (until they give the system a minute or two to recover).

This limiting can be done on a global basis, a per-IP address basis (simply have an array list of IPs and their current allowed searches and last time), or even a per-user basis (i.e., one leaky bucket per user), or the methods can be combined, allowing each user five searches per minute and a system total of 50 searches per minute (assuming more would make your system become too slow to use).

Distributed Token Buckets

A common flaw made with rate-limiting systems that occurs in applications with more than one server (which is almost all major applications nowadays) is that the various servers and system components do not properly share state. For example, a site with five front-end web servers may implement a download rate limit or a limit on login attempts for clients, but if they do not communicate with each other, an attacker would be able to execute five times the rate-limited downloads or attacks by hitting each server at the same time. Of course, the solution is to use a shared state. Two possible solutions are to use a database with row-level locking (reducing the amount of contention for checking and updating the shared state information) or use something very lightweight that is also extremely fast, such as memcached [4]. The memcached server not only supports storing keys and values associated with the keys but can also store a counter (a 64-bit integer) that can be incremented or decremented atomically (i.e., multiple processes won't step on each others' toes) with the *incr* and *decr* commands. In this way, you can simply insert a key with a value equivalent to the user name, IP address, or whatever you are using and then maintain a counter. For each attempt to log in, you increment the counter, and you regularly decrement the counter by a control process to ensure that attempts are expired.

Lockouts vs. Timeouts

Another aspect of rate limiting is the ability to reduce problems caused by lockouts – systems that try to protect themselves by locking out users after a certain number of failures, for example. Lockouts are prone to spoofing attacks. If attackers can make themselves look like legitimate users (i.e., by trying to use the victim's username to log in), they might be able to lock out that user (assuming you lock an account out after three bad passwords). Alternatively, if you have customers or users coming from behind proxy servers, you might end up with one bad user blocking access for a group of legitimate users.

The use of increasingly long timeouts (such as doubling the waiting period each attempt) can be just as effective as a lockout but has the added (dis)advantage that when the attack stops, the timeout will eventually reset, allowing whatever caused the timeout to start again. Of course, depending on what you want, you can set the timeouts appropriately to either allow users into their accounts or block spammers for extended periods of time.

What Is Your Quest

A final option available in conjunction with rate limiting is allowing users of a system to prove that they do not have hostile intentions (despite their "bad" behavior). For example, if you send too many similar-looking queries to Google in a short time period you might get a web page saying "Sorry" that directs you to enter a CAPTCHA string to prove you are human and not an automated program or malicious program. All of which is much better than a user staring at a blank and non-responsive web page. ■

Listing 1: Leaky Bucket Pseudo-Code

```
01 searches=5
02 per_second=60
03 current_allowed=searches
04 last_check = time()
05 while process(search_terms):
06         # we determine how many seconds since the last search
07         time_now=time()
08         time_passed = time_now - checked_at
09         # and set when our last search was
10         checked_at=time_now
11         # add tokens to bucket:
12         current_allowed += time_passed * (searches / per_second)
13         # check if we have any tokens
14         if (current_allowed > searches):
15                 # we have reached our max. tokens
16                 current_allowed = searches
17         if (current_allowed < 1):
18                 #partial token, ignore search
19                 discard_search()
20         else:
21                 # we have at least one token
22                 do_search()
23                 # and we "spend" the token
24                 current_allowed = current_allowed - 1
```

INFO

[1] "Apache HTTPD" by Kurt Seifried, *Linux Magazine*, September 2009, pg. 52, *http://www.linuxpromagazine.com/Issues/2009/106/APACHE-HTTPD*

[2] Leaky Bucket: *http://en.wikipedia.org/wiki/Leaky_bucket*

[3] Token Bucket: *http://en.wikipedia.org/wiki/Token_bucket*

[4] Memcached: *http://memcached.org/*

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.