

Garanticemos la disponibilidad de nuestra aplicación

TASA LIMITADA

Quizá haya llegado el momento de que prestemos atención a los chicos de la red. **POR KURT SEIFRIED**

Hace unos meses escribí sobre el ataque Slowloris a servidores web [1]. En resumen: El atacante se conecta, dejando la conexión abierta, usando muy pocos recursos en su extremo, pero afectando a todos los sockets disponibles e impidiendo conectarse a los usuarios legítimos. Desde entonces se han anunciado varios tipos de ataques de denegación de servicio contra servidores web, aplicaciones web y otros servicios. Lo que me llevó a pensar: ¿Cómo puede el programador lidiar con este tipo de problemas de un modo general para reducir su impacto?

La respuesta es, por supuesto, limitando la tasa. Lo más divertido es que en el mundo interconectado en que vivimos, la limitación de las tasas y el control de flujos son elementos críticos de las redes basadas en IP que llevan funcionando años literalmente, habiéndose resuelto una gran variedad de problemas. Por desgracia, la mayoría de programadores a nivel de aplicación ignoran estos trabajos o simplemente no saben por qué habríamos de preocuparnos por los límites de las tasas o el control de los flujos (no será porque no se trabaja lo suficiente en cuanto a peticiones de nuevas funcionalidades o a reporte de fallos).

Ejemplos de Ataques

La importancia de la limitación de la tasa depende del modo en que se lleva a cabo dicha limitación. Un ejemplo de ataque es el basado en la utilización de las funciones de búsqueda con varias expresiones, de forma que se consumen grandes cantidades de CPU y de memoria, así como recursos de la base de datos. Otro ataque típico a servidores FTP consiste en ejecutar un

comando al estilo de `ls */..*/..*/..*/..*/..*/..*/..*/..*/..*`, que tiene como consecuencia una ingente cantidad de consultas de directorios y, en algunos casos, incluso puede llegar a tirar el servidor.

Los sistemas de reserva de vuelos permiten elegir asientos, reservándolos durante un breve espacio de tiempo (por ejemplo, 10 ó 15 minutos) para que otros no puedan cogerlos mientras los pagamos. Un atacante podría iniciar el proceso de compra de asientos varias veces sin llegar a finalizarlos, impidiendo a los clientes legítimos la compra real de asientos.

Los spammers pueden descargar sitios web automáticamente, creando una copia de sus contenidos, para ponerlos en sus propios sitios web junto con ads y generar ingresos. También podríamos simplemente garantizar acceso a los servidores para los clientes que realizan abonos con preferencia sobre los usuarios que consumen servicios gratuitos.

¿Dónde Limitar la Tasa?

Una vez determinada la tasa a limitar, falta por decidir dónde tendrá lugar dicha limitación. No tiene sentido limitar el número de nuevas conexiones por dirección IP en caso de tener habilitado *HTTP Keep-Alive* (permitiendo a los clientes hacer más de una petición por conexión al servidor web). De manera similar, si lo que nos preocupan son las contraseñas débiles en cuentas de usuario (y por tanto los problemas de soporte relativos a cuentas comprometidas), simplemente se podrían limitar las probabilidades de éxito de los ataques de averiguación de contraseñas por fuerza bruta, permitiendo a la vez a los usuarios visitar tantas páginas web como deseen una vez autenticados.

En términos generales, se limitará la

tasa dentro de la aplicación para obtener la máxima flexibilidad y control. Sin embargo, si el problema viene dado por la gente que se conecta a la aplicación (por ejemplo, debido a un tiempo de inicio demasiado alto) o no se puede modificar la aplicación propiamente dicha, bien porque el código sea cerrado o bien porque sea demasiado extenso y enrevesado como para meterle mano sin mayores complicaciones, se puede considerar la posibilidad de interponer un proxy o de usar una capa adicional para proteger el sistema.

El Problema de los Cubos Agujereados

Entonces, ¿cómo se puede limitar la tasa? La forma más sencilla de hacerlo es mediante el algoritmo del cubo con agujeros [2] (Listado 1). Mediante este programa relativamente sencillo, simplemente definimos una tasa aceptable (como pueda ser una búsqueda cada doce segundos, o cinco búsquedas por minuto). Conforme van llegando las peticiones, se van disponiendo en "el cubo". Los trabajos se van tomando del cubo a una tasa previamente definida (suponiendo que las tareas vayan llegando más rápido de lo que se puede procesar), descartando los nuevos trabajos. De este modo se garantiza que el trabajo en curso nunca excederá la cantidad máxima.

Pero este esquema presenta un problema significativo: ¿qué ocurre cuando un usuario necesita llevar a cabo diez búsquedas rápidamente y sólo puede hacer una cada cierto tiempo? Seguro que se sentirá frustrado casi inmediatamente al ver que la sexta búsqueda falla y que tiene que esperar un minuto para hacer el resto. ¿Cómo podemos lidiar con las peticiones hechas a ráfagas al tiempo que limitamos la actividad de los usuarios para evitar daños serios?

La Ventaja de los Cubos Testigo

Con el cubo de testigos [3] se puede gestionar el tráfico a ráfagas (que en un momento es alto y al siguiente hay poco) permitiendo cierta tasa de tráfico (por

ejemplo, cinco búsquedas por minuto) y permitiendo también que dichas búsquedas se produzcan en períodos de un segundo. De este modo se fuerza al usuario a esperar sesenta segundos antes de que pueda realizar nuevas búsquedas.

El algoritmo también se puede modificar para incluir un límite máximo de capacidad, permitiendo el almacenamiento de hasta 10 testigos, por ejemplo, con un límite de cinco por minuto. De este modo, los usuarios pueden realizar inicialmente hasta diez búsquedas durante el primer minuto, pero pasadas las diez primeras búsquedas, sólo podrán hacer cinco por minuto (hasta que den un minuto o dos al servidor para que se recupere).

Esta limitación se puede efectuar de modo generalizado, por direcciones IP (a través de un array de direcciones con sus correspondientes límites especificados), o incluso por usuarios (por ejemplo, un cubo de testigos por usuario), o también combinando los distintos métodos con, por ejemplo, un límite de cinco búsquedas por minuto y usuario con un total de 50 búsquedas por minuto en el sistema entre todos los usuarios (dando por hecho que un número de búsquedas mayor ralentizaría demasiado el sistema).

Cubos de Testigos Distribuidos

Un fallo común en estos limitadores que se da en aplicaciones con más de un servidor (como es el caso de la mayoría de las apli-

caciones hoy día), es que los distintos servidores y componentes del sistema no comparten la información sobre estados adecuadamente. Por ejemplo, un sitio con interfaces en cinco servidores web diferentes puede implementar un límite en la tasa de descarga o limitar a los clientes el número de intentos de acceso, pero si no se comunican los unos con los otros, un atacante podría multiplicar por cinco la tasa de descarga realizando peticiones a los cinco servidores simultáneamente. Por supuesto, el problema se resuelve compartiendo la información sobre los estados. Dos posibles soluciones pasan por: a) usar una base de datos con bloqueos a nivel de fila (reduciendo la cantidad de conflictos a la hora de comprobar o actualizar la información sobre los estados); b) usar algo muy ligero y rápido, como pueda ser *memcached* [4]. El servidor *memcached* no sólo soporta el almacenamiento de claves junto con los valores asociados a éstas, sino que puede albergar también un contador (un entero de 64 bits) para aumentarlo o disminuirlo atómicamente (sin que se pisen unos procesos a otros) mediante los comandos *incr* y *decr*. De este modo, se puede introducir una clave con un valor equivalente al nombre de usuario, la dirección IP, o cualesquiera datos que vayamos a usar para mantener el contador. Con cada intento de acceso se incrementa el contador, reduciéndolo de nuevo a intervalos regulares para garantizar la expiración de los intentos.

Lockouts vs. Timeouts

Otro aspecto de la limitación de la tasa es la capacidad de reducir los problemas derivados del uso de *lockouts* (sistemas que tratan de protegerse a sí mismos bloqueando al usuario tras un número determinado de intentos fallidos, por ejemplo). Los *lockouts* son propensos a sufrir ataques de falseamiento (*spoofing*). Si un atacante consigue adoptar la apariencia de un usuario legítimo (digamos, probando un nombre de usuario legítimo para acceder), puede llegar a bloquear la cuenta de dicho usuario al alcanzar el límite máximo de intentos de acceso. Alternativamente, si tenemos usuarios o clientes que se encuentran tras servidores proxy, el atacante podría acabar bloqueando el acceso a más de un usuario legítimo a la vez.

El uso de *timeouts* durante períodos de tiempo incrementales (por ejemplo, esperando el doble de tiempo tras cada nuevo fallo), puede ser tan efectivo como el de los *lockouts*, con la característica añadida de que cuando el ataque cese, el *timeout* se reseteará eventualmente, permitiendo empezar de nuevo a lo que fuera que provocó el *timeout*. Claro está que dependiendo de nuestras necesidades, podremos establecer los *timeouts* adecuadamente para permitir a los usuarios acceder a sus cuentas o bien bloquear a los spammers durante períodos de tiempo extendidos.

¿Qué Buscamos?

Una opción final en conjunción con la limitación de la tasa es permitir a los usuarios de un sistema probar que no tienen intenciones hostiles (a pesar de su “mal comportamiento”). Por ejemplo, si enviamos un gran número de peticiones similares en un breve espacio de tiempo, puede que recibamos un mensaje del estilo “Disculpe las molestias” que nos lleve a introducir un *CAPTCHA* con el que demostrar nuestra naturaleza humana y que no somos ningún programa automatizado o malicioso. Siempre será mucho mejor que toparnos con una página en blanco o una que simplemente no responda. ■

Listado 1: Pseudo Código para el Cubo Agujereado

```
01 searches=5
02 per_second=60
03 current_allowed=searches
04 last_check = time()
05 while process(search_terms):
06 else:
07 # tenemos al menos un testigo
08 do_search()
09 # y lo empleamos (el testigo)
10 current_allowed = current_allowed - 1
11 # determinamos cuántos segundos han transcurrido desde
12 # la última búsqueda
13 time_now=time()
14 time_passed = time_now - checked_at
15 # y definimos el momento en que ésta se produjo
16 checked_at=time_now
17 # añadimos testigos al cubo:
18 current_allowed += time_passed * (searches / per_second)
19 # comprobamos si hay testigos
20 if (current_allowed > searches):
21 # hemos alcanzado el número máximo de testigos
22 current_allowed = searches
23 if (current_allowed < 1):
24 #testigo parcial, se ignora la búsqueda
25 discard_search()
```

RECURSOS

- [1] “Apache HTTPD” por Kurt Seifried, Linux Magazine Número 57, página 8.
- [2] Cubo Agujereado: http://en.wikipedia.org/wiki/Leaky_bucket
- [3] Cubo de Testigos: http://en.wikipedia.org/wiki/Token_bucket
- [4] Memcached: <http://memcached.org/>