Detector de ataques DOS (Hashing con direccionamiento abierto)

© F.J. Madrid Cuevas (fjmadrid@uco.es)
Estructuras de Datos. Grado de Ingeniería Informática
Universidad de Córdoba. España

Objetivos

- Implementar una tabla hash con gestión de colisiones usando direccionamiento abierto.
- Aprender a usar los tipos de entero con tamaño de bit fijo definidos en `<cstdint>`.
- Aprender a usar la nueva forma de generación de números aleatorios con C++11 definidos en `<random>`.

Descripción

Un ataque “Denial of Service” (DOS) consiste en que el ordenador atacante (o ordenadores si es coordinado) inundan a un servidor (p. ej. web, email, dns …) con peticiones de acceso con tal velocidad que este servidor no puede atender a nadie más (legítimo o no) y en caso extremo hasta puede caer.

Normalmente los servidores informan al sistema operativo de los accesos anómalos utilizando alguna utilidad “Log” del sistema. Un detector de ataques DOS rastreará estos “Log” periódicamente buscando patrones de acceso anómalos y bloqueando (“ban”) las direcciones consideradas como maliciosas en el cortafuegos por lo que ya no accederán al servidor.

En esta práctica vamos a simular un detector de ataques DOS. El diseño del detector se muestra en los algoritmos 1 y 2.

El detector (algoritmo 2) analizará periódicamente (cada segundo) el Log del sistema (lo abstraemos como una secuencia de entradas “ `<tiempo> <dirección ip>` ”). Usaremos dos variables para definir la ventana temporal. La variable `first` indica la primera entrada del Log sin procesar y la variable `last` indica la entrada más antigua dentro de la ventana temporal.

En cada ciclo del algoritmo principal (`DOS_detector`) el detector analizará y avanzará la ventana temporal 1 minuto usando el algoritmo `update_counters` .

El algoritmo `update_counters` tiene dos pasos:

- El primero avanza el comienzo de la ventana temporal hasta el tiempo `System::now()` incrementando la variable `first` e incrementando el contador de accesos asociados a la IP de cada nueva petición registrada en el Log. Además, se controla cuántas veces una misma dirección IP ha realizado una petición al servidor en el último minuto. Si este número de peticiones supera un umbral especificado por el usuario (`max_acc`), se considerará esta dirección IP como maliciosa y será bloqueada (`System().ban_ip()`) en el cortafuegos.
- El segundo paso del algoritmo, avanza el límite posterior de la ventana temporal de un minuto (`System::now()-60`) incrementando la variable `last` y descuenta los accesos de las IPs que han quedado fuera de la ventana temporal, ya que sólo nos interesa los accesos en el último minuto.

El detector usará dos tablas hash: una para almacenar un contador de peticiones por IP activa en la ventana temporal y otra para mantener las IPs que han sido bloqueadas junto con el tiempo en el que el bloqueo termina. El sistema se encargará (dentro de la función `System::sleep()`) de sacar de la tabla de IPs bloqueadas aquellas para las que el tiempo de bloqueo se haya cumplido.

Algoritmo 1

```
Procedure DOS_detector(
  log:Log,//Array of pairs <Time, IP>
  max_acc:Integer)//Max. num. of acc.
Var
   first:Integer //First unprocessed line of log.
   last:Integer //Last line of Log in current temporal window.
   c:HashTable[IP,Integer] //Save a counter by active ip.
Begin
  first ← 0
  last ← 0
  While System::sleep(1) Do //sleep 1 second.
          update_counters(log, first, last, c, max_acc)
  End-While
End.
```

Algoritmo 2

```
Procedure update_counters(
  log:Log,//Array of pairs <Time, IP>
  Var first:Integer,//First unprocessed line of log.
  Var last:Integer, //Last line of Log in current temporal window.
  Var c:HashTable[IP, Integer],//Save a counter by active ip.
  max_acc:Integer //Max. num. of acc. allowed.
)
Begin
  //update new accesses.
  While log[first].time < system::now() Do
          increment_counter(log[first].ip, c)
          If n_acc(log[first].ip, c) >= max_acc Then
            System::ban_ip(log[first].ip)
          End-If
          first ← first + 1
  End-While
  //remove old accesses.
  While log[last].time < System::now()-60 Do
          decrement_counter(log[last].ip, c)
          last ← last + 1
  End-While
End.
```

Detalles de implementación

Uso de enteros con un tamaño de bit fijo

Para implementar la funciones `Ip2Int()` y la función hash se debe por un lado convertir una dirección IPv4 en un entero sin signo de al menos 32 bits y realizar las operaciones para hacer el "hashing" usando más de 32 bits para poder ser representadas.

Como el tamaño en bits de los tipos int, unsigned, long y unsigned long dependen de la arquitectura donde se compila y queremos realizar un código general, vamos a utilizar los tipos de enteros con tamaño fijo definidos en [1] como son std::uint8_t, std::uint32_t y std::uint64_t para asegurar un tamaño de bits apropiado.

Sin embargo, el estándar C++ no asegura que todas las arquitecturas puedan proporcionar estos tipos por lo tanto hay que comprobar esto al configurar el proyecto. En el fichero CMakeLists.txt se puede ver una forma de indicar que nuestro proyecto depende de que la arquitectura donde se compila proporciona el tipo std::uint64_t.

Sobre la generación de números aleatorios

El estándar C++11 añade un nuevo paquete para generación de números aleatorios accesible al

incluir el fichero de cabecera `#include <random>` [3].

Anteriormente se utilizaba la funciones `rand()` y `srand()`. El problema es que hay distintos algoritmos para generar números pseudo-aleatorios y la función `rand()` sólo implementa un de estos algoritmos. Además, "*estas funciones mantienen un estado interno que es global y no está protegido. Esto significa que las llamadas a rand() nunca son seguras en entornos multi hilo*" [4].

El estándar C++11 [3] proporciona un método para utilizar distintos algoritmos generadores de números pseudo-aleatorios, seguros en entornos multi-hilo. Además, el estándar también proporciona funciones para generar números aleatorios siguiendo las más conocidas distribuciones de probabilidad como: uniforme en rango flotante (`std::uniform_real_distribution<>`), uniforme en rango entero (`std::uniform_int_distribution<>`), normal, binomial, etc.

Sobre la simulación del sistema operativo

El código base entregado usará un “singleton” [2] accesible con “System()” para abstraer algunas operaciones del sistema operativo como son: `time()`, `sleep()`, `ban_ip()` y `banned_ips()`.

Evaluación

Superar todos los tests	**Puntos**
`test_ip_utils tests_ip_utils`	2
`test_hash_f tests_hash_function`	1
`test_hash_f tests_hash_function_lp`	0,5
`test_hash_f tests_hash_function_rp`	0,5
`test_hash_f tests_hash_function_qp`	0,5
`test_hash_f tests_hash_function_dh`	0,5
`test_hash_table tests_hash_table`	3
`test_dos_detector tests_dos_detector`	2

Referencias

[1] `<cstdint>` : https://en.cppreference.com/w/cpp/header/cstdint
[2] Patrón de diseño OO "Singleton": https://es.wikipedia.org/wiki/Singleton
[3] `<random>` : https://en.cppreference.com/w/cpp/numeric/random

[4] Funciones que no son seguras en entornos multi hilo: https://developer.arm.com/documentation/dui0475/m/the-c-and-c---library-functions-reference/c-library-functions-that-are-not-thread-safe