Detector de ataques DOS

Objetivos.

- Aprender a implementar una tabla hash con gestión de colisiones usando encadenamiento.
- Aprender a usar los tipos definidos en <cstdint>

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 detector analizará periódicamente (cada segundo) el Log del sistema (lo abstraemos como una secuencia de entradas <tiempo> <dirección ip>). En cada ciclo el detector analizará, en una ventana temporal de 1 minuto, cuántas veces una misma dirección ip ha accedido al servidor. Si este número de veces supera un umbral especificado por el usuario, se considerará esta dirección ip como maliciosa y será bloqueada (“banned”) en el cortafuegos.

El detector usará dos tablas hash una para almacenar un contador de acceso por IP activa en la ventana temporal y otra para mantener el tiempo de la primera vez que una IP ha sido bloqueada.

El diseño del detector será el siguiente:

```
Algorithm DOS_detector(
  log:Log,//Array of pairs <Time, IP>
  maxAcc:Integer)//Max. num. of acc.
Local
   i:Integer //First unprocessed line of log.
   j:Integer //First line in current temporal window.
   c:HashTable[IP,Integer] //Save a counter by active ip.
Begin
  i ← 0
  j ← 0
  while system::sleep(1) do //sleep 1 second.
     updateCounters(log, i, j, c, maxAcc)
```

```
  end-while
end.

Algorithm updateCounters(
  log:Log,//Array of pairs <Time, IP>
  Var i:Integer,//First unprocessed line of log.
  Var j:Integer,//First line in current temporal window.
  Var c:HashTable[IP, Integer],//Save a counter by active ip.
  maxAcc:Integer //Max. num. of acc. allowed.
)
Begin
  //update new accesses.
  while log[i].time < system::now() do
     increment(log[i].ip, c)
     if nAcc(log[i].ip, c) >= maxAcc then
        system::banIP(log[i].ip])
     end-if
     i ← i + 1
  end-while
  //remove old accesses.
  while log[j].time < system::now()-60 do
     decrement(log[j].ip, c)- 1
     j ← j + 1
  end-while
end.
```

Así, en esta práctica tendrás que implementar en primer lugar el tipo HashTable[K,V] usando cadenas para gestionar las colisiones y después usando tablas hash el detector de DOS.

Detalles de implementación.

Sobre las cadenas y el cursor.

Como vamos a usar el mecanismo de encadenamiento para gestionar las colisiones, se utilizará el tipo `std::list` de la STL [3] para representar las cadenas.

Para implementar el cursor de la tabla, deberás combinar un índice (`size_t`) de tabla junto con un iterador a lista (`std::list<>::iterator`) que representa la cadena colisiones de esta entrada.

Así al mover el cursor se deben actualizar convenientemente estos dos atributos que lo representan. Por ejemplo, partiendo de una posición válida, para pasar al siguiente elemento (`goto_next()`), primero avanzaremos el iterador de la lista y si alcanzamos el

final de la lista, avanzamos el índice de tabla para buscar la siguiente entrada no vacía y posicionamos el iterador en el primer elemento de la lista de esta entrada.

Para eliminar un elemento en `std::list` dado un iterador al mismo utiliza la operación `std::list<>::erase()`.

Sobre el uso de enteros con un tamaño de bit independiente de la arquitectura.

Para implementar la funciones Ip2Int() y 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” que requieren más de 32 bits para poder ser representadas.

Como el tamaño en bits de los tipos integrales enteros dependen de la arquitectura donde se compila y queremos realizar un código general, se recomienda utilizar los tipos definidos en <cstdint> [1] como son uint_8, utint_32 y uint_64 para asegurar un tamaño de bits apropiado.

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 uint64_t con 8 bytes.

Sobre la simulación del sistema operativo.

Por otro lado, el código base entregado usará un “singleton” [2] para abstraer las operaciones del sistema operativo como son System().time(), System().sleep(), System().ban_ip() y System().banned_ips().

Referencias.

[1] <cstdint>: http://www.cplusplus.com/reference/cstdint/

[2] Singleton: https://es.wikipedia.org/wiki/Singleton

[3] std::list: http://www.cplusplus.com/reference/list/list/list/