Intrusion Detection System

Ejercicio N° 2

Objetivos	• Diseño y construcción de sistemas orientados a objetos • Diseño y construcción de sistemas con procesamiento concurrente • Encapsulación de Threads y Sockets en clases • Protección de los recursos compartidos
Instancias de Entrega	**Entrega 1:** clase 6 (18/04/2017). **Entrega 2:** clase 8 (02/05/2017).
Temas de Repaso	• Threads en C++11 • Clases en C++11
Criterios de Evaluación	• Criterios de ejercicios anteriores • Orientación a objetos del sistema • Empleo de estructuras comunes C++ (string, fstreams, etc) en reemplazo de su contrapartida en C (char*, FILE*, etc) • Uso de **const** en la definición de métodos y parámetros • Empleo de constructores y destructores de forma simétrica • Buen uso del stack para construcción de objetos automáticos • Ausencia de condiciones de carrera e interbloqueo en el acceso a recursos • Buen uso de Mutex, Condition Variables y Monitores para el acceso a recursos compartido

Índice

Introducción

Un sistema para la detección de intrusos o en sus siglas en inglés IDS (Intrusion Detection System) es un software que pasivamente está a la espera de una actividad sospechosa.
Existen múltiples variantes de un IDS, y en este trabajo práctico se desarrollará un IDS para redes IP.

Descripción

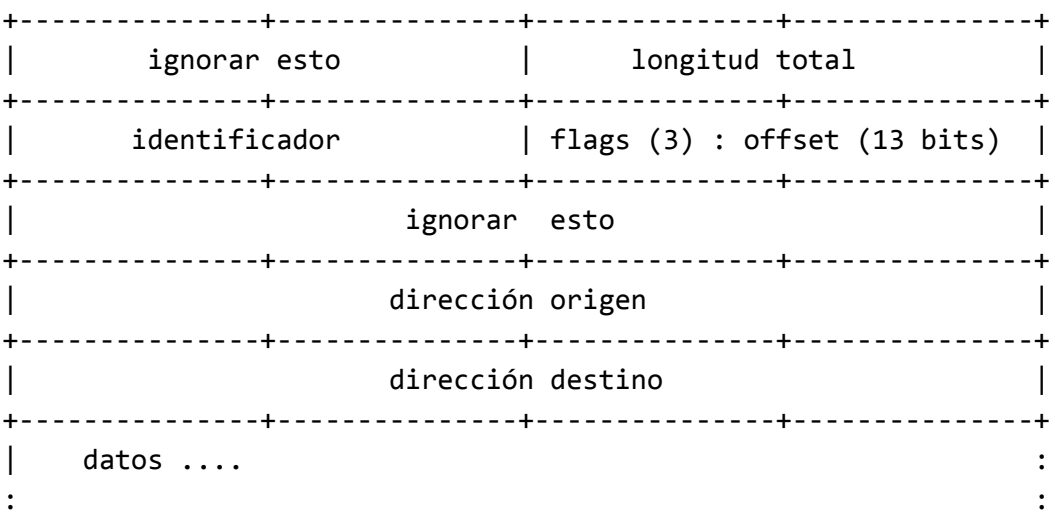
En líneas generales un IDS para redes IP consta de tres fases:

1. **Captura**: el IDS captura paquetes de red. En nuestro caso solamente paquetes IP y con el fin de simplificar el trabajo los paquetes seran leidos de un archivo en vez de ser capturados.
2. **Ensamblado**: los paquetes capturados son ensamblados y los datos que transportan son reinterpretados en protocolos de más alto nivel. En nuestro caso nos limitaremos a únicamente reconstruir paquetes IP fragmentados.
3. **Detección**: finalmente el IDS intenta detectar alguna actividad abnormal haciendo uso desde simples reglas lógicas hasta análisis de patrones y machine learning. En nuestro caso solo implementaremos unas reglas lógicas simples.

Captura

Como se comentó anteriormente, la captura o *sniffing* es el proceso de captura de los paquetes de red. Nuestro trabajo sin embargo solo leerá de un archivo binario que contiene una captura previa de solamente paquetes IP.

Cada paquete IP tiene el siguiente formato (simplificado)



El presente trabajo no tiene como objetivo hacer un análisis en detalle del protocolo IP así que varios de los campos deben ser ignorados.
Para el resto, su definición es:

- **longitud total**: son 2 bytes que representan la longitud en bytes de los datos mas la cabecera.
- **identificador**: es un número de 2 bytes que representa a un paquete IP. Si este es fragmentado, todos los fragmentos comparten el mismo identificador.
- **flags**: son tres bits en donde solo nos interesa el tercero: el bit de **MF** o **More Fragments**.
- **offset**: son trece bits que determinan el offset para ensamblar los fragmentos.
- **dirección origen**: es la dirección IP del host origen de donde proviene el paquete. Puede ser vista como un número de 4 bytes en Big Endian.
- **dirección destino**: es la dirección IP del host destino hacia donde el paquete es enviado. Puede ser vista como un número de 4 bytes en Big Endian.
- **datos**: cero o más bytes completamente arbitrarios.

Los numeros deben interpretarse como numeros sin signo y en big endian.
La **longitud total** es la longitud de todo el paquete, esto es la cabecera (20 bytes) más los datos.
Ambas direcciones IP deben verse como simples números. *No se requiere ninguna conversion al dot-format como "192.168.0.1".*

Por ejemplo, este es el hexdump del archivo **one.cap**, un archivo de captura con un único paquete IP

```
00000000  45 00 00 1f 00 00 00 00  00 00 00 00 00 00 00 01  |E...............|
00000010  00 00 00 02 68 65 6c 6c  6f 20 77 6f 72 6c 64     |....hello world|
0000001f
```

Vemos que

- **longitud total**: "00 1f" o simplemente 31 bytes. Dado que la cabecera tiene 20 bytes, nos quedan 11 bytes para los datos.
- **identificador**: "00 00" o simplemente 0.
- **flags** & **offset**: "00 00", tanto los flags como el offset son cero.
- **dirección origen**: "00 00 00 01" o simplemente 1.

- **dirección destino**: "00 00 00 02" o simplemente 2.
- **datos**: "68 65 6c 6c 6f 20 77 6f 72 6c 64" o "hello world". Notese como son exactamente 11 bytes.

Ensamblado

Un paquete IP puede fragmentarse en múltiples paquetes IP. Nuestro IDS deberá ser capaz de reconstruir el paquete IP completo original a partir de los fragmentos.

Por ejemplo, el paquete del archivo **one.cap** podría haberse fragmentado en 2 como lo muestra el siguiente hexdump

```
00000000  45 00 00 18 00 00 20 00  00 00 00 00 00 00 00 01  |E..... .........|
00000010  00 00 00 02 68 65 6c 6c  45 00 00 1b 00 00 00 04  |....hellE.......|
00000020  00 00 00 00 00 00 00 01  00 00 00 02 6f 20 77 6f  |............o wo|
00000030  72 6c 64                                          |rld|
00000033
```

Vemos que hay dos paquetes IP.
Para el primero tenemos

- **longitud total**: "00 18" o 24 bytes (4 bytes de datos).
- **identificador**: "00 00" o 0
- **flags** & **offset**: "20 00" que visto como 16 bits tenemos 001 00000. Vemos entonces que el bit **MF** es 1 y el **offset** es 0.
- **dirección origen** y **destino**: 1 y 2 respectivamente
- **datos**: "68 65 6c 6c" o "hell"

Notamos que el bit de **MF** es 1 por lo tanto hay mas fragmentos (de ahi el nombre del flag).

Para el segundo tenemos:

- **longitud total**: "00 1b" o 27 bytes (7 bytes de datos)
- **identificador**: "00 00" o 0
- **flag** & **offset**: "00 04" que visto como bits es 000 00100. Vemos que el bit **MF** es 0 y el **offset** es 00100 en binario o simplemente 4.
- **dirección origen** y **destino**: 1 y 2 respectivamente
- **datos**: "6f 20 77 6f 72 6c 64" o "o world"

Notamos que el bit de **MF** es 0 pero que el **offset** es mayor a cero lo que indica que este es un fragmento tambien, el ultimo.

Notese como

- los fragmentos de un mismo paquete original comparten el mismo **identificador** y las mismas **direcciones de origen** y **destino**.
- el primer fragmento siempre tiene un **offset** igual a 0
- el último fragmento siempre tiene el bit **MF** igual a 0, el resto no.
- el **offset** de un fragmento es la suma de la cantidad de bytes de **datos** (no incluye la cabecera) de los fragmentos anteriores, o equivalentemente es la suma del **offset** y de la **(longitud total - 20)** del fragmento anterior.

Ahora bien, hay dos observaciones importantes

- dada una captura, nada garantiza que los fragmentos vengan *en orden*. El campo **offset** debera ser usado para el ordenamiento de ellos hasta que se tenga todos los fragmentos y se pueda ensamblar el paquete completo.
- dada una captura, nada garantiza que no *falten* fragmentos. Una vez más el campo **offset** y **MF** deberá ser usado para detectar esto. En este caso el IDS deberá esperar a que los fragmentos faltantes sean leídos o bien, si no hay más capturas, simplemente se descartan.

  Nota: investigue la estructura de datos de **std::map** y la función **std::sort** de **<algorithm>**.

Detección

El IDS deberá leer una serie de reglas desde un archivo de configuración. Estas tienen el siguiente formato:

```
src  dst  threshold  keyword  w1 w2 w3... ;
```

Donde

- **src** y **dst** son las direcciones origen y destino. Una regla aplica solamente si **src** y **dst** coinciden con las **direcciones origen** y **destino** del paquete a procesar.
- **src** y **dst** pueden valer 0 y en cuyo caso la dirección origen y/o la destino no son comparadas (funcionan como *wildcards*)
- **threshold**: es la cantidad de veces que una regla tiene que aplicar hasta que recién se empiecen a emitir alertas.
- **keyword**: puede ser *"always"*, *"all"* o *"any"*:
  - *"always"* ignora **w1 w2 w3...** y siempre aplica la regla.
  - *"all"* solo aplica la regla si todas las palabras **w1 w2 w3...** aparecen en los datos del paquete IP.
  - *"any"* solo aplica la regla si alguna de las palabras **w1 w2 w3...** aparecen en los datos del paquete IP.
- toda regla termina con un punto y coma.
- se garantiza que habrá al menos un espacio entre cada uno de los parámetros de la regla incluso entre el punto y coma.

Tanto **src**, **dst** como **threshold** son números sin signo escritos en notación hexadecimal.

Nota: investigue el **operador >>**, la biblioteca **<iomanip>** y el manipulador **std::hex**.

Una vez que una regla aplica y supera el threshold genera una alerta. El IDS deberá imprimir por salida estándar lo siguiente:

```
Rule num: ALERT! src -> dst: datos
```

Donde

- **num** es el número de la regla (contando desde 0)
- **src** y **dst** son las direcciones origen y destino del paquete que generó la alerta
- **datos** son los datos del paquete, cada byte separado por un espacio.

Tanto **src**, **dst** como los **datos** deben imprimirse en hexadecimal.

Veamos unos ejemplos:

Sea la captura (del ejemplo anterior)

```
00000000  45 00 00 18 00 00 20 00  00 00 00 00 00 00 00 01  |E..... .........|
00000010  00 00 00 02 68 65 6c 6c  45 00 00 1b 00 00 00 04  |....hellE.......|
00000020  00 00 00 00 00 00 00 01  00 00 00 02 6f 20 77 6f  |............o wo|
00000030  72 6c 64                                          |rld|
00000033
```

El IDS ensambla ambos fragmentos en un único paquete IP:
- src = 1
- dst = 2
- datos = "hello world"

y es este quien es sometido a la etapa de Detección.

Sean las reglas:

```
0 0 0 always ;
0 0 0 all hello hola ;
0 0 0 any hello hola ;
0 0 4 always ;
2 0 0 always ;
```

- La primer regla aplica, puede no hay restricciones de ningún tipo.
- La segunda no, ya que "hello world" no contiene a "hola"
- La tercera regla aplica.
- La cuarta regla aplica pero no genera una alerta ya que debe aplicar 4 veces (a 4 paquetes) antes de empezar a emitir alertas y solo aplico 1.
- La quinta regla tampoco aplica ya que el **src** de la regla (2) no coincide con el **src** del paquete (1).

Luego, este sería la salida esperada

```
Rule 0: ALERT! 1 -> 2: 68 65 6c 6c 6f 20 77 6f 72 6c 64
Rule 2: ALERT! 1 -> 2: 68 65 6c 6c 6f 20 77 6f 72 6c 64
```

Multithreading

Nuestro IDS podrá recibir múltiples capturas y deberá procesar cada una de ellas en **paralelo** con un thread separado. Es importante aclarar que los fragmentos IP pueden estar distribuidos en las múltiples captura por lo que los threads deberán coordinar entre si para el ensamblado.

> **Nota:** como en cualquier trabajo, la mejor estrategia es separar el trabajo en bloques, implementar de a uno a la vez, testearlo y luego continuar con el siguiente bloque.
>
> Una posible estrategia para este trabajo práctico es implementar primero la captura y parseo de paquetes y testear que el parser funcione correctamente; luego las reglas y testear que funcionen correctamente (con esto varios casos de pruebas deberían pasar) Luego el ensamblador (más casos de pruebas deberían pasar); y finalmente el multithreading.
>
> Trabajar de a bloques, verificando y testeando cada bloque antes de avanzar con el siguiente bloque es una estrategia sumamente efectiva. NO intente programar todo el trabajo y testear/debuggear al final.

Formato de Línea de Comandos

```
./tp reglas.cfg captura1.cap captura2.cap ...
```

El primer argumento es el archivo de configuración de las reglas. El resto son las capturas.

Códigos de Retorno

El IDS deberá retornar 1 si no hay parámetros suficientes o 0 en otro caso.

Entrada y Salida Estándar

No se hará uso de la entrada estándar. Por el otro lado el IDS emitira las alertas por la salida estándar.

Ejemplo de Ejecución

Sea el archivo de captura **cap.cap** cuyo contenido es:

```
00000000  45 00 00 24 00 00 20 0a  00 00 00 00 00 00 00 03  |E..$.. .........|
00000010  aa bb cc dd 20 61 6e 64  20 62 65 6e 64 20 74 68  |.... and bend th|
00000020  65 20 73 70 45 00 00 1e  00 00 20 00 00 00 00 00  |e spE..... .....|
00000030  00 00 00 03 aa bb cc dd  44 6f 20 6e 6f 74 20 74  |........Do not t|
00000040  72 79 45 00 00 1b 05 39  00 6a 00 00 00 00 00 00  |ryE....9.j......|
00000050  00 03 aa bb cc dd 6e 6f  74 68 69 6e 67 45 00 00  |......nothingE..|
00000060  3e 00 00 20 40 00 00 00  00 00 00 00 03 aa bb cc  |>.. @...........|
00000070  dd 79 20 74 72 79 20 74  6f 20 72 65 61 6c 69 7a  |.y try to realiz|
00000080  65 20 74 68 65 20 74 72  75 74 68 3a 20 74 68 65  |e the truth: the|
00000090  72 65 20 69 73 20 6e 6f  20 73 70 45 00 00 1b 00  |re is no spE....|
000000a0  2a 00 6a 00 00 00 00 00  00 00 03 aa bb cc dd 6e  |*.j............n|
000000b0  6f 74 68 69 6e 67 45 00  00 18 00 00 00 6a 00 00  |othingE......j..|
000000c0  00 00 00 03 aa bb cc dd  6f 6f 6e 2e 45 00 00 00  |.........oon.E...|
000000d0  00 3a 00 00 20 1a 00 00  00 00 00 00 00 03 aa bb  |.:.. ...........|
000000e0  cc dd 6f 6f 6e 2e 20 54  68 61 74 27 73 20 69 6d  |..oon. That's im|
000000f0  70 6f 73 73 69 62 6c 65  2e 20 49 6e 73 74 65 61  |possible. Instea|
00000100  64 2e 2e 2e 20 6f 6e 6c                           |d... onl|
00000108
```

El primer paquete tiene:

- **longitud total** = 36
- **identificador** = 0
- **MF** = 1
- **offset** = 10
- **datos** = `" and bend the sp"` (16 bytes)

El segundo tiene:

- **longitud total** = 30
- **identificador** = 0
- **MF** = 1
- **offset** = 0
- **datos** = `"Do not try"` (10 bytes)

Nótese como por tener un **offset** de 0 este paquete debe ser necesariamente el primer fragmento y como **MF** es 1 sabemos que hay más fragmentos.

El tercero tiene:

- **longitud total** = 27
- **identificador** = 1337
- **MF** = 0
- **offset** = 106
- **datos** = `"nothing"` (7 bytes)

Nótese cómo este paquete es un fragmento pero no está relacionado con los 2 anteriores (a pesar de tener las mismas **direcciones de origen** y **destino**, el **identificador** es distinto).

El cuarto tiene:

- **longitud total** = 62
- **identificador** = 0
- **MF** = 1
- **offset** = 64
- **datos** = `"y try to realize the truth: there is no sp"` (42 bytes)

El quinto tiene:

- **longitud total** = 27
- **identificador** = 42
- **MF** = 0
- **offset** = 106
- **datos** = `"nothing"` (7 bytes)

El sexto tiene:

- **longitud total** = 24
- **identificador** = 0
- **MF** = 0
- **offset** = 106
- **datos** = `"oon."` (4 bytes)

Vemos que el flag **MF** es 0 por lo tanto debe ser el último fragmento.

El septimo tiene:

- **longitud total** = 58
- **identificador** = 0
- **MF** = 1
- **offset** = 26
- **datos** = `"oon. That's impossible. Instead... onl"` (38 bytes)

Nótese que con este fragmento ya podemos ensamblar los fragmentos, esto es porque si ordenamos los fragmentos por **offset** vemos que:

```
 offset  (longitud total - 20)  MF  datos
--------------------------------------------------------------------------------
 0              10              1   "Do not try"
 10             16              1   " and bend the sp"
 26             38              1   "oon. That's impo"
 64             42              1   "y try to realize the truth: there is no sp"
 106            4               0   "oon."
```

Nótese como no nos falta ningún fragmento ya que cada **offset** es la suma del **offset** y de la **(longitud total - 20)** del fragmento anterior. Y que el último fragmento tiene **MF** igual a 0.

Finalmente podemos rearmar el paquete original cuyo datos seran
```
 "Do not try and bend the spoon. That's impossible. Instead... only try to realize the truth: there is no spoon."
```

Sea el archivo de reglas **reglas.cfg**:

```
0 0 0 all spoon ;
9 9 0 aways ;
```

Al ejecutar

```
 ./tp reglas.cfg cap.cap
```

debería emitir lo siguiente

```
Rule 0: ALERT! 3 -> aabbccdd: 44 6f 20 6e 6f 74 20 74 72 79 20 61 6e 64 20 62 65 6e 64 20 74 68 65 20 73 70 6f 6f 6e 2e 20 54 68 61 74 27 73 20 69 6d 70 6f 73 73 69 62 6c 65 2e 20 49 6e 73 74 65 61 64 2e 2e 2e 20 6f 6e 6c 79 20 74 72 79 20 74 6f 20 72 65 61 6c 69 7a 65 20 74 68 65 20 74 72 75 74 68 3a 20 74 68 65 72 65 20 69 73 20 6e 6f 20 73 70 6f 6f 6e 2e
```

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C++11.
2. Está prohibido el uso de variables globales.
3. Debe haber al menos una jerarquía de clases y polimorfismo.
4. Debe haber al menos una implementación del constructor y operador asignación por movimiento.
5. Debe haber al menos una clase con el constructor y operador asignación por copia borrados.
6. Cada archivo de captura debe procesarse en un hilo distinto.
7. Debe haber al menos una implementación de un objeto monitor.
8. Cada vez que el Ensamblador esté en condiciones de reconstruir un paquete, este debe dárselo inmediatamente al Detector.

Referencias

[1] Paquete IP: https://en.wikipedia.org/wiki/IPv4#Packet_structure
[2] Fragmentacion y ensamblado: https://en.wikipedia.org/wiki/IPv4#Fragmentation_and_reassembly
[3] Containers de C++ (buscar std::map): http://www.cplusplus.com/reference/stl/
[4] Streams de C++ (buscar operador >> y std::hex): http://www.cplusplus.com/reference/iolibrary/