UNIVERSIDAD DE ZARAGOZA

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Escuela de Ingeniería y Arquitectura

Desarrollo de un 'sniffer' para la generación de listas blancas para Snort

Arturo Ruiz Mañas
OSNA Cyber Security Research Group www.osna-solutions.com

OSNA
OPEN SENSOR NETWORK AUTHENTICATION

Director: Dr. Michael Schukat
Ponente: Dr. José Luis Salazar Riaño

email: arruma2160@gmail.com
Zaragoza, Agosto de 2013

Resumen

A lo largo de esta memoria, voy a tratar sobre el complejo pero importante problema de la seguridad en redes de Control Industrial y sistemas SCADA (Supervisory Control And Data Acquisition). Para esto, previo a este trabajo, se me ha presentado abundante información sobre Hacking y técnicas de Intrusión ICS (Internet Connection Sharing). En el presente texto profundizaremos en una solución ante situaciones de “black-hat hacking”[1]para dichos entornos.

Lograr un nivel alto de seguridad en nuestros sistemas de información y señales de control es el objetivo de todo empleado en seguridad informática. Durante este texto presentaré el software que he creado durante mi estancia en el grupo OSNA en Irlanda, que busca precisamente ayudar a conseguir ese nivel alto de seguridad. Para ello me baso en la idea de “Deep Packet Inspection”[30]: se toma cada paquete que la interfaz de red detecta y se examinan campos concretos del paquete. De este modo el programa realizará un estudio de los valores que dichos campos toman, originando una representación lo más precisa posible de la información que recorre nuestro segmento de red a estudiar, en forma de reglas para Snort.

Además, otro objetivo añadido a mi diseño, es el combinar dos de los métodos tradicionales de seguridad informática: listas-blancas y listas-negras. Por un lado, nuestro enfoque mediante “Deep Packet Inspection”, comentado en el párrafo anterior, aporta el enfoque de “listado-blanco”, basado en listas-blancas, y por otro, el componente de listas-negras vendrá dado por la multitud de ficheros de reglas-Snort colgadas en la red, creadas por especialistas en temas de seguridad informática[Snus00], y que como “software libre” que es, podemos perfectamente descargar y utilizar a nuestro antojo. La integración en un sólo diseño de ambas filosofías, es por lo tanto, un punto interesante a tener en cuenta.

Sintetizando con pocas palabras, diré que, utilizando mi programa, cuyo fin será el de elaborar una descripción lo más exacta posible del tráfico de red (listas blancas), y pasando dicha representación a Snort, programa para la “Detección de Intrusiones”, junto con ficheros de listas-negras y preprocesadores para Snort ya presentes en la Web, estamos generando una herramienta que, sin duda alguna, va a ser muy útil en las tareas de seguridad de redes y que representa a mi parecer, un recurso muy interesante a tener entre el software de todo encargado de la seguridad en sistemas de comunicaciones informáticas.

1.Hacking se puede dividir en tres categorías diferentes: hacking de sombrero negro, hacking de sombrero blanco, y hacking de sombrero gris. Los nombres resultan muy representativos sobre sus significados. Los de sombrero negro son los “malos”, los de sombrero blanco son los “buenos”, y los de sombrero gris son lo que están entre medio.

Agradecimientos:

En especial a mi familia, por todo el apoyo recibido por su parte a lo largo de estos años.

Agradecimientos al grupo OSNA y en especial a Michael Schukat.

Debo también agradecer a José Luis su ayuda en la composición de este escrito.

Tabla de Contenidos

Tabla de figuras

Lista de tablas

1. Introducción y Exposición de Objetivos

1.1 Introducción a la Seguridad en Redes

A día de hoy nos vemos envueltos en un mundo sin igual, en el que las comunicaciones y asuntos diarios pasan a través de una “nube” formada por multitud de máquinas y dispositivos electrónicos. Nuestros emails alcanzan su destinatario, no sin antes haber atravesado la maraña de routers que conforman Internet; ahora ya podemos sobrevivir sin poner pie en un supermercado, ya que con tan sólo un ordenador, una conexión a Internet y un pequeño número de clicks podemos realizar nuestra compra semanal; transacciones bancarias, últimos libros, ropa, redes sociales... Todo puede ser realizado, y así lo es, desde el despacho en el trabajo, o desde el confort de tu sofá en casa.

Pero, ¿qué sucede en el ámbito de las empresas privadas? Datos personales, información y señales de control en entornos industriales, documentos de importancia... todo sale/llega desde/a un computador, viaja a lo largo del cable en una intranet, estando esta intranet muy probablemente conectada a la red de redes: Internet está más presente que nunca y todo el mundo debería entender el importante papel que juega la Seguridad en Redes en todo esto. Guste o no, nos encontramos inmersos de lleno en una era digital, en la cual todo es traducido a unos y ceros.

Dicho esto, diría que un buen punto de partida para este documento podría consistir en definir con pocas palabras lo que es la “Seguridad en Redes”. Así pues, ¿qué es la Seguridad en Redes?

- Según la Wikipedia [1]:

La seguridad en redes consiste en las medidas y políticas adoptadas por un administrador de red para prevenir y monitorizar accesos no autorizados, malos usos, modificaciones o restricciones a una red de computadores o recursos de red accesibles a través de la misma.

- La Webopedia dice [2]:

Un campo especializado dentro del “computer networking” que involucra la seguridad de una infraestructura de la red. La seguridad en redes es normalmente manejada por un administrador de red o administrador del sistema que implementa las políticas de seguridad, el software y el hardware necesitados para proteger una red y los recursos accedidos a través de la misma de accesos no autorizados.

- Una definición más simple la tenemos en la revista digital “ Magazine Encyclopedia”[3]:

Protección de sistemas de computadores en red de intrusiones no deseadas.

En definitiva, podemos concluir que la Seguridad en Redes previene de ataques y posibles amenazas, protegiendo los sistemas de computadores que posibilitan el desarrollo de nuestras actividades diarias. Pero también podríamos plantear esto desde otra perspectiva, considerando las implicaciones de la Seguridad de Redes, una vez las medidas de seguridad han sido vulneradas, y afirmar que se ocupa de controlar y monitorizar lo que dentro de la red sucede, de modo que se pueda verificar que todo está dentro de un orden, y en caso contrario poner medidas al respecto.

1.2 Ciberseguridad y Amenazas a CCI

Para explicar correctamente el contexto de este trabajo, me gustaría hablar sobre qué representa el término Ciberseguridad y qué son los Ataques a CCI (Conexión Compartida a Internet) para posteriormente realizar la exposición de objetivos.

En el mundo actual, ataques contra IC (Infraestructuras Críticas) de energía, gas, petróleo, agua... están creciendo, y no es extraño conocer que detrás de dichos ataques se encuentran organizaciones respaldadas económicamente tanto por empresas competidoras como incluso por

gobiernos de países[32].

Basta tan sólo para dar un marco real a este punto, con consultar en Internet artículos de periódicos, que en los meses que llevamos de año 2013, hacen referencia a ataques a la seguridad informática:

- HuffingtonPost [31] → Posted: 05/16/2013 :

"Syria faced an Internet blackout for eight hours on Wednesday, its second one in the past week and the sixth one of the two-year uprising against President Bashar al-Assad, a U.S. web trafficking firm reported. Phone lines into Damascus were also down."

- Informe Semanal [33] - Espionaje masivo → 15 jun 2013

"Edward Snowden era, hasta hace unos días, uno más de los miles de empleados anónimos que analizan la información para las agencias de inteligencia del Gobierno de Estados Unidos. Huido a Hong Kong y después en paradero desconocido, se ha convertido en uno de los hombres más buscados por el FBI. "

- esmateria.com [39]→ "La ciberguerra es inevitable" 04/06/2013

"Los expertos advierten de que las infraestructuras críticas dependen de sistemas vulnerables"

- BBC News Technology [40]→ 19 May 2013 Last updated at 23:52 GMT

"How to hack a nation's infrastructure"

La lista de noticias que hablan sobre hechos relacionados con ciberseguridad/ciberataques es larga, podemos sin duda encontrar una buena colección de páginas web que dan cuenta de estos hechos. Dedicar sólo unos instantes a leer alguna de estas noticias, nos ayuda a comprender por qué es tan importante invertir esfuerzos en asegurar y proteger las redes de datos y entornos como el que tratamos en este texto.

1.3 Seguridad en Sistemas SCADA

1.3.1 ¿Qué significa SCADA?

El acrónimo SCADA hace referencia a Supervisory Control And Data Acquisition. Los sistemas SCADA son un tipo de CCI, en concreto son sistemas controlados por computador cuya tarea es monitorizar y controlar sistemas industriales. Han estado presentes en nuestras vidas desde principios de los años 70, permitiéndonos controlar remotamente dispositivos distribuidos a lo largo de grandes extensiones.

Ejemplos de sistemas SCADA son los sistemas que permiten a operadores establecer y modificar condiciones que hacen saltar alarmas que controlan la temperatura en sistemas de control de temperatura por enfriamiento de agua en ciertos procesos industriales; o los sistemas que monitorizan los niveles alto y bajo en tanques de agua, y alertan cuando el nivel de agua ha alcanzado un cierto límite... Existen muchos ejemplos de sistemas SCADA que desarrollan importantes funciones en procesos que intervienen directamente en nuestro bienestar social y calidad de vida.

El diseño de los sistemas SCADA ha evolucionado mucho a lo largo de los años. Su arquitectura consiste en un sistema de computadores centrales que se comunican con otras máquinas usando una o más tecnologías de comunicación. Durante la última década, Internet también se ha incluido en el diseño de estos sistemas, de este modo, por un lado, se les está dotando de una mayor flexibilidad y funcionalidades extra, pero a su vez, por contra, resultan en sistemas mucho más vulnerables a ataques, siendo esta la motivación de este trabajo.

1.3.2 Nivel de Seguridad en los Sistemas SCADA

A día de hoy, la preocupación sobre como proteger estos sistemas está en aumento; ya que, a pesar del importante papel que desarrollan en nuestra vida (responsables del control y motorización de sistemas tales como los de distribución de agua, de tuberías de petróleo y gas, de red eléctrica), estos dispositivos aún poseen muchas vulnerabilidades [41] [42].

Muchos ciber-ataques en la actualidad, se centran en lograr el control de estos sistemas SCADA y otros sistemas CCI. Debemos tener en mente que si uno de estos ataques llega a buen puerto, puede desencadenar terribles consecuencias en términos de salud humana e incluso llegar a representar un riesgo para la vida. Ejemplos de estos ataques son: DoS (Denial of Service), mediante el cual dejamos la máquina fuera de servicio; robo de contraseñas, que otorgan privilegios al atacante en la máquina atacada; impersonalización, como una posible consecuencia del robo de contraseñas; falsificación de archivos o borrado de los mismos... Existen multitud de formas de ataque.

Algo que resultará muy útil a todo administrador de red encargado de poner medidas de protección ante estos ataques expuestos, será conocer su red. El conocimiento del funcionamiento de los dispositivos que existen en ella, junto con el conocimiento de “qué es normal” y que “anormal”, va a ser información muy importante de cara a reaccionar con velocidad ante un ataque que está ocurriendo. Es por este aspecto que la herramienta que en este documento se presenta resulta tan atractiva, ya que ayudará a detectar ataques lo antes posible, pudiendo establecer soluciones al respecto antes de que sea demasiado tarde.

1.4 Exposición de Objetivos

¿Qué medidas se pueden tomar para mejorar el nivel de seguridad contra ataques a CCI? Actualmente existen muchas herramientas en manos de los encargados de la seguridad informática: cortafuegos , antivirus... Aun así, con el creciente número de ataques y su diversificación, no se debe nunca bajar la guardia. Lo que un día fue un sistema seguro, hoy puede no serlo. La tecnología avanza pero los ataques se vuelven más sofisticados a su vez.

Desde el grupo OSNA, se propone combinar dos de las filosofías tradicionales de protección de sistemas informáticos: “black-listing” y “white-listing”.

- Black-listing o lista-negra

Consiste en un método cuya aproximación a la seguridad está basado en la comparación del tráfico observado con patrones que describen conductas que no deben ser permitidas, consintiendo sólo aquellas conexiones cuyos comportamientos no estén explícitamente descritos en una lista negra. El problema de este método es que sólo previene de comportamientos que han sido analizados y estudiados previamente, dejando un agujero de seguridad durante una cierta ventana de tiempo: desde el momento en el que surge una nueva forma de ataque, hasta cuando se dispone de un patrón que describa el comportamiento de tal amenaza.

Además, los atacantes estudian formas para burlar estas medidas de seguridad, inclusive estudian las listas negras. Sus métodos pasan desde dividir la carga del ataque entre diversos paquetes, de modo que las medidas de seguridad que buscan el ataque completamente contenido en un paquete, sean incapaces de saltar alarmas; hasta la representación de la información en el mismo paquete de un modo que no sea fácilmente reconocible; llegando a eludir este método de Black-listing.

- White-listing o lista-blanca

En la otra cara de la moneda tenemos la perspectiva contraria: white-listing. Esta filosofía hará saltar alarmas cada vez que se observe un comportamiento fuera de lo esperado. Dicho comportamiento

"correcto" estará completamente descrito dentro de las llamadas listas-blancas.

Aunque de nuevo, este método tampoco resulta 100% seguro. Su fortaleza, pero a la vez su debilidad, estriban en cómo de bien descrito esté el comportamiento "aceptable" dentro de sus listas de reglas blancas. Si no somos capaces de afinar bien una regla dentro de dichas listas, estaremos permitiendo comportamientos que aún estando dentro de lo aceptado, constituyen una amenaza para el sistema.

Existe a su vez, un tercer método de detección de ataques a redes de datos llamado Detección de Anomalías. Este método es una de las últimas aproximaciones a la Seguridad de Redes y se basa en el estudio de N-gramas[Anexo B], creados a partir de la información contenida en los paquetes. Realizando un estudio estadístico, la idea es sacar patrones del contenido que se observa en los paquetes, alertando cuando dicho contenido sea muy distinto del observado durante una fase de estudio del tráfico de red. Si esta aproximación no se ha añadido a nuestro diseño, es debido a que necesita de un gran trabajo para luego no lograr resultados demasiado positivos: a veces un patrón puede ocurrir en un campo concreto del paquete pero no en otros...

En definitiva, el objetivo del estudio que aquí desarrollo es el de presentar una herramienta que sea capaz de fundir las filosofías de protección de redes previamente expuestas. Por un lado, usando la idea de DPI (Deep Packet Inspection)[30], mi software será capaz de, durante una fase preliminar de estudio del tráfico de red, realizar un análisis estadístico sobre el tráfico en el segmento de red a controlar, creando listas con los comportamientos que se suponen normales (listas blancas). Por otro lado, haciendo uso de "listas negras" ya creadas por especialistas en el tema[9] y añadiéndolas a nuestro diseño, busco presentar, lo que sin duda alguna, da lugar a una herramienta muy completa.

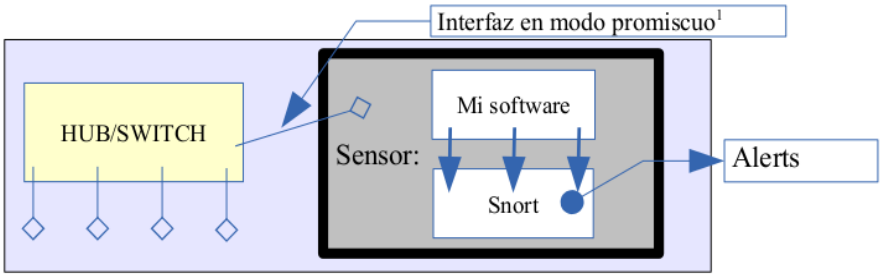


Fig.1 Representación de mi software + Snort en una red conformada por nodos de interconexión (hubs / switches)

2. Requisitos de Usuario

Con el fin de presentar de un modo lógico la problemática, iré desde lo más general a lo más particular, exponiendo al final los requisitos parciales de usuario que me he planteado y que me han ayudado a tener una guía de acción

2.1 Redes de Comunicación en el Entorno Industrial.

Un primer requisito de usuario, y muy importante a tener en cuenta, es el entorno de trabajo de las máquinas que vamos a monitorizar. Entornos sometidos a condiciones de trabajo más exigentes, ya que se ven sometidos a condiciones extremas, ya sea de temperatura, humedad, vibración... lo que conllevará modificaciones en los protocolos que ya conocemos, e incluso, el uso de nuevos estándares.

1. Una interfaz en modo promiscuo es capaz de recibir no sólo los paquetes destinados a su propia interfaz sino cualquier paquete que fluya en el segmento de red al que se conecta.

Ejemplos de estos entornos los podemos observar en cualquier central eléctrica (nucleares, eólicas, hidráulicas...). Para cualquiera que haya visitado alguna en su vida, le será fácil comprender que, los componentes que trabajan en estos, soportan condiciones que pueden llegar a exceder los rangos usuales de temperatura, vibración, presión... de los equipos IT. Para lidiar con estos problemas, se utilizan los llamados Protocolos de Red de Sistemas de Control Industrial[29]. Ejemplos de estos protocolos son: Ethernet Industrial[22], Modbus[23], ZigBee, EtherCAT... la lista es extensa y busca dar solución a problemáticas particulares derivadas del entorno de trabajo.

2.1.1 Modbus

Profundizando en los Protocolos de Red de Sistemas de Control Industrial, presento el protocolo con el que se me pidió que mi software debiera trabajar: Modbus. Se trata de un protocolo simple y robusto usado para comunicaciones en serie, publicado originalmente por Modicon (www.modicon.com) para su uso con PLCs (Programmable Logic Controller), que se ha consolidado como un importante protocolo de comunicaciones y un modo de conexión entre dispositivos electrónicos[ezTCP].

Modbus posibilita la comunicación de entre aproximadamente hasta 240 máquinas conectadas a la misma red. Se utiliza para conectar un computador supervisor con una unidad remota RTU (Remote Terminal Unit) en sistemas de control y adquisición de datos SCADA[23][24][25].

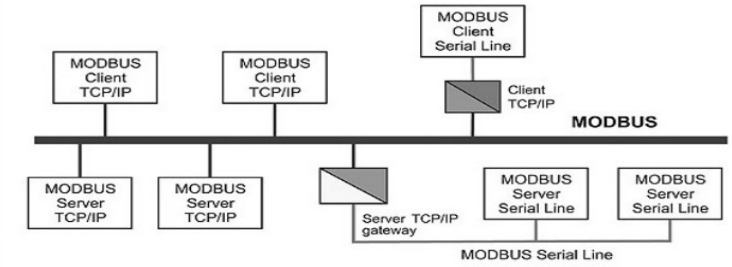


Fig.2 Ejemplo de set-up de una red Modbus

Existe software a disposición en Internet para simular el funcionamiento de redes Modbus. Algunos de los dispositivos en dicha red serán Master (Maestro) y otros serán Slave (Esclavo). También ha sido desarrollada una API Modbus que simplifica el proceso de creación de software más específico en la creación de un escenario virtual Modbus.

Existen varios tipos de Modbus: Modbus RTU, Modbus ASCII y Modbus TCP; mi software tiene como requisito de diseño trabajar con Modbus TCP. El hecho de trabajar en un software para un protocolo encapsulado sobre TCP, abre la posibilidad a futuras extensiones a otros protocolos de similar encapsulado.

Layer	ISO/OSI Function	Modbus Function
5,6,7	Application	Modbus Application Protocol
4	Transport	Transmission Control Protocol
3	Network	Internet Protocol
2	Data Link	IEEE 802.3
1	Physical	IEEE 802.3

Tabla 1 Pila de protocolos en Modbus TCP

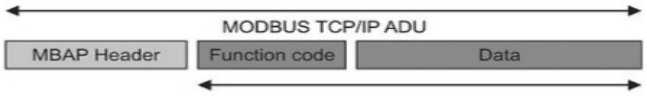


Fig.3 Modbus TCP – ADU

Transaction Identifier	Protocol Identifier	Length	Unit Identifier
2 bytes	2 bytes	2 bytes	1 byte

Fig.4 Cabecera en Modbus TCP

2.2 IDS / IPS (Intrusion Detection System / Intrusion Prevención System)

Este proyecto se enmarca dentro del campo de la Detección y Prevención de Intrusiones. El acrónimo IDS/IPS hace alusión a una aplicación software que es capaz de contrastar los paquetes que transitan en nuestro segmento de red con unos patrones predefinidos (listas blancas / listas negras). Más en concreto, como requisito de usuario, se me presentó con la herramienta IDS llamada Snort, herramienta de software libre con la que mi software debe de trabajar y que delimita mis opciones de diseño, como más adelante veremos.

2.2.1 Snort

Así pues, teniendo en mente el Detector de Intrusiones Snort, se me pidió idear un método que fuera capaz de crear un “mapa” informativo de/con las conexiones de nuestro segmento de red y “dárselo de comer a Snort”, de modo que se pueda automatizar el proceso de detección de dicho IDS. Pero, ¿qué es Snort en definitiva?

Snort es un programa de software libre desarrollado por Sourcefire[Snus00]. Se usa para detectar accesos no autorizados a ordenadores y redes de comunicación. Puede ponerse a funcionar de tres modos distintos: sniffer, logger, NIDS (Network Intrusion Detection System), siendo este tercer modo el que nos interesa debido a su carácter IDS.

Su funcionamiento se basa en módulos, los cuales trabajan sobre los paquetes que el sensor capta en su segmento de red. La siguiente imagen describe visualmente la relación entre los distintos módulos que conforman Snort.

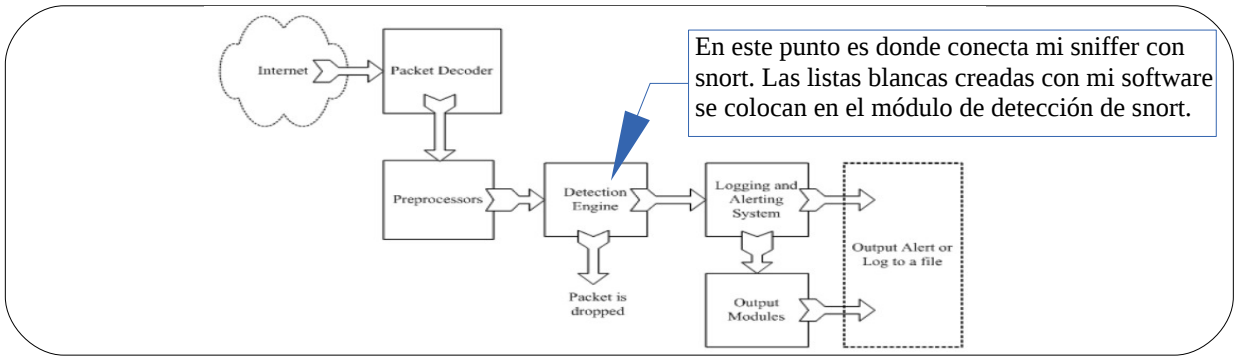


Fig.5 Flujo de los paquetes a través de los módulos de Snort

Vayamos paso por paso utilizando como apoyo la imagen anterior:

Primeramente, los paquetes, transmitiéndose por la red, son captados por nuestro sensor Snort, a través de su NIC (Network Interface Card). Acto seguido son decodificados (primer módulo),

permitiendo conocer qué protocolos utilizan las comunicaciones entre máquinas.

Tras ello, y con Snort trabajando como NIDS (Network Intrusion Detection System), los paquetes serán enviados a los diversos preprocesadores según haya sido establecido en el fichero de configuración de Snort. Los preprocesadores son plug-ins que Snort usará para realizar ciertas transformaciones sobre el paquete, de modo que no se nos escapen ataques más elaborados. Estos preprocesadores son usados en la escritura de las reglas.

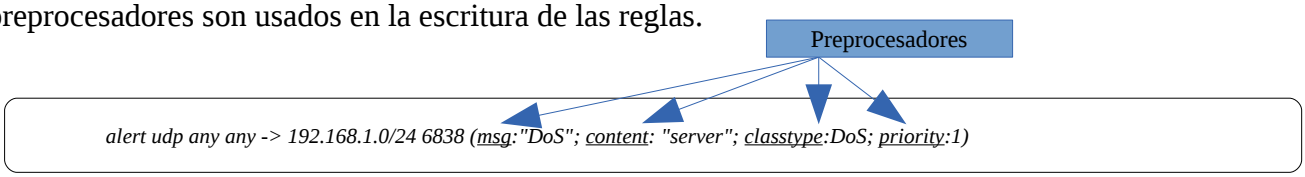


Fig.6 Ejemplo de regla Snort

Tercer módulo, “Detection Engine”, es donde están contenidas las reglas y se produce la acción propiamente de contraste entre paquetes y reglas. El resultado de este módulo será pasado a los componentes de alerta y “loggeo”, que generarán las alertas necesarias en caso de haber detectado algún comportamiento fuera de la norma. Una alerta generará una entrada en un fichero de logging de modo que pueda ser consultada en un posible futuro estudio del ataque.

Los módulos de Output establecen el modo en el que el administrador podrá ser “alertado”: desde un e-mail a su cuenta personal de correo, un mensaje a su móvil, un busca conectado al sensor Snort... las posibilidades son múltiples y variadas.

2.3 Requisitos para la Creación del Software

Ahora que tenemos una visión general de la situación puedo empezar a plantear una solución a la problemática expuesta. Voy a crear un software que analice tráfico Modbus y cree listas blancas para Snort basándose en la idea de DPI, aprovechando las alternativas que Snort nos brinda ya de por sí.

A fin de lograr una herramienta más completa se requiere, además, añadir al diseño un enfoque de black-listing, a través de listas negras ya publicadas en Internet (ej. www.Snort.org) con las que mi software trabajará a fin de juntar ambos enfoques en el mismo diseño.

Personalmente, he procurado descomponer el problema de diseño del software en varias etapas, de modo que pudiera ir aproximándome paso a paso al resultado final. Estas etapas las considero requisitos parciales de usuario y son las siguientes:

1. *Creación de un “sniffer”: programa que recoge los paquetes que fluyen por el segmento de red al cual se ha conectado la máquina sensor y lo descompone en sus campos.*
2. *Cuando ya tenemos un programa sniffer, debemos ser capaces de trabajar con los campos de interés. En un primer momento pensamos únicamente en los protocolos IP y TCP, almacenando en nuestra estructura de árbol dinámico, estructura explicada más adelante, los campos de interés: Ip fuente, Ip destino, puerto fuente, puerto destino.*
3. *Teniendo un algoritmo que ya es capaz de recoger esos campos y guardarlos como nos interesa, es momento de extender nuestro algoritmo de modo que también sea capaz de recoger información Modbus. Los campos Modbus que queremos almacenar, extendiendo aún más la idea de árbol dinámico, son tres: longitud del fragmento Modbus, identidad y código de función Modbus.*
4. *Ahora que ya tenemos la información guardada, deberemos representarla en ficheros para poder estudiar que recoge la información correctamente. Logrado esto podemos pasar a representar esta información en un modo que Snort pueda comprender y utilizar.*
5. *Ya tenemos los ficheros de reglas, pero Snort deberá saber donde están contenidos. Nuestro software moverá estos ficheros de reglas a las carpetas destinadas para tal efecto dentro del sistema de archivos de Snort y modificará el fichero de configuración de éste para que Snort sepa que debe tenerlos en cuenta.*

Tras estos 5 puntos, ya sólo queda poner a funcionar Snort en modo NIDS (Network Intrusion Detection System) y estar tranquilos sabiendo que ahora Snort, protegerá nuestro sistema de todo lo que no se adecue al “mapa de conexiones” que nuestro software a generado.

3. Análisis y Diseño

3.1 Entorno de Trabajo

La primera decisión a tomar ha sido sobre el sistema operativo, ¿comercial o software libre? Personalmente, la idea de software libre me gusta mucho, pero lo que más ha pesado en mi decisión es el hecho de que Snort sea en sí mismo software libre. Ya sólo por esto, mi decisión ha sido rápida: software libre; lo que me lleva a pensar en Linux[LinSP00].

Pero, ¿qué distribución Linux? Mandriva, Debian, CentOS, Fedora... existen tantas que resulta complicado extraer motivos por los cuales tomar una y descartar otra. Pensando en que lo importante será que nuestro sistema dedique, en su mayoría, recursos a lo que realmente interesa, que son las tareas de sensor y no pierda el tiempo en temas de entorno gráfico u otras florituras como muchos sistemas operativos actuales hacen, me han convencido dos versiones de la distribución Debian[48][49] que cumplen con esto: Xubuntu y Openbox. Openbox [46][47], con el entorno gráfico más ligero de todos los Linux, centrando su funcionamiento especialmente en su shell. Por otro lado Xubuntu[52], versión de Ubuntu, dedica muy pocos recursos a temas gráficos y que, personalmente, es la elección para mi ordenador personal. Será entonces, por motivos de reducción de la curva de aprendizaje y que cumple con los requisitos que le pido al sistema operativo, lo que me lleva a elegir Debian Xubuntu.

¿Qué lenguaje vamos a utilizar en la programación? Puedo elegir entre los lenguajes Java, C y C++; pero tenemos por otro lado un sistema operativo escrito enteramente en C y un programa IDS/IPS que a su vez, también está escrito enteramente en C: resulta una elección fácil por armonizar todo, elegir como lenguaje de programación el lenguaje C.

Además, uno de los placeres de trabajar en Linux es la programación de scripts shell [NeSt00]; facilitan las tareas de administración del sistema y pueden ser incluidos en cualquier código C a través de la llamada al sistema “system”. Ahora bien, ¿qué shell debo elegir? Existen varias versiones de shell: sh Bourne, csh, zsh, ksh, bash... La versión “bash” shell es la más común en sistemas Linux, esto hace que sea más probable que un administrador de sistema este familiarizado con esta versión, logrando que éste trabaje más cómodo desde un principio y podamos ahorrar en tiempo de aprendizaje sobre otros shells, ya que por otro lado, no aportan ventajas unos sobre otros. Así pues, elegir el “bash” shell parece la opción más acertada.

3.2 Librería PCAP

Pensando en la creación de un sniffer de modo que podamos ir cumpliendo con el punto 1 de los requisitos parciales de usuario y teniendo en cuenta que trabajamos en un entorno Linux y programamos en C, no disponemos de muchas alternativas de diseño, sólo nos queda la posibilidad de buscar una API para C. Una API (Application Program Interface) [4][5][6] define la interfaz a través de la cual componentes software se comunican entre sí a nivel código. Provee de un nivel de abstracción a través de un conjunto de interfaces, normalmente funciones, que un código puede invocar [LinSP00]. Resulta una importante ayuda para los desarrolladores de software de cara a poder olvidarse de las particularidades del hardware con el que trabajan y centrarse en su tarea de programación.

Las APIs contendrán rutinas, estructuras de datos, constantes, variables... que pueden ser usadas

en tus programas. El modo en el que se usan siempre es el mismo: comienzas usando una función “open”, que abre un flujo de datos sobre el cual puedes leer y/o escribir como si fuera un archivo normal. La idea de las API concuerda con la filosofía Linux (y previamente UNIX) de “hacerlo todo 'un archivo' ” (el sistema operativo “mapea” cualquier hardware conectado al PC como si fuera un archivo) resultando en un nivel de “abstracción” que facilita el trabajo.

PCAP (Packet CAPture) es precisamente una implementación de API que se usa para capturar tráfico de red. Es la base hoy en día para cualquier programa sniffer o cualquier Sistema de Detección de Intrusiones como Snort o sniffers tales como Wireshark. Conocidos paquetes software como “Aircrack-ng suite” [Hck01] también basan sus funcionalidades en PCAP. Los sistemas operativos tipo UNIX implementan PCAP en su librería libpcap,y podrá ser utilizada en tareas de programación C tras su inclusión mediante la sentencia *#include <pcap/pcap.h>*, donde pcap.h es el fichero de cabecera.

3.3 ¿Cómo nuestro programa trabaja con Snort?

Snort trabaja contrastando las reglas que describen comportamientos permitidos y no permitidos y que están contenidas en sus ficheros de reglas, con el tráfico que observa en su NIC (Network Interface Card). Esta NIC, por su parte, deberá estar en modo promiscuo, lo que significa que deberá de ser capaz de captar cualquier paquete que vea circulando en su segmento de red [Hck01] .

El aporte de una filosofía de “listas blancas” implementando la idea de “Deep Packet Inspection” (DPI) [30] va a ser nuestro aporte principal al campo de la Detección de Intrusiones; pero, ¿cómo podemos crear estas listas que describen el comportamiento permitido en nuestra red, sin necesidad de hacerlo a mano y afinar al máximo en la descripción de dicho comportamiento?

Snort posee un modo propio de descripción de los comportamientos que se emplea en la elaboración de reglas, que consiste en una estructuración concreta y forma de escritura particular de sus reglas, al cual, los archivos resultado de nuestro software, se deben adaptar, de modo que podamos entendernos con Snort.

Poniendo a correr mi software, durante una fase de estudio inicial del tráfico de red en el segmento a controlar, el sniffer que he elaborado, será capaz de generar reglas Snort que describan el comportamiento a nivel IP-TCP (referente a direcciones y puertos) y a nivel Modbus (recogiendo los campos de funciones, números de identidad, longitud paquete Modbus), automatizando el proceso de creación de reglas blancas y cumpliendo con los puntos 2 y 3 de los requisitos parciales de usuario.

Posteriormente, y una vez finalizado este proceso de estudio, también será capaz de interactuar con el sistema de archivos y ficheros de Snort, colocando estos ficheros de reglas (además de las reglas negras que se decidan incluir al diseño) dentro de las carpetas que Snort utiliza para almacenar los ficheros de este tipo y modificará el fichero de configuración de Snort mediante sentencias “include”, permitiendo al sensor conocer la existencia de nuestros nuevos ficheros de reglas.

En definitiva el proceso resulta muy automático y alivia en gran medida las tareas de cualquier administrador de red, facilitando el proceso de “ajuste” de un sensor IDS/IPS Snort.

3.4 Árbol de Listas Anidadas

Todo programa que se precie de realizar una función útil, deberá contener estructuras de datos donde almacenar el valor de sus variables. Mi programa necesita almacenar los datos con los que realizar estadísticas, la estructura que utilizaré la he llamado “Árbol de listas anidadas”.

De modo que podamos entender qué son y cómo he llegado a ello, muestro a continuación el pensamiento evolutivo que me ha llevado hasta dar con la idea del “Árbol de Listas Anidadas”:

1. *Primera idea: arrays; el principal problema que presenta un array es la delimitación del mismo: declarar el número de variables que lo componen. Por el tipo de problema al que nos enfrentamos, no podemos determinar a priori cuántas variables vamos a necesitar; si lo hiciéramos, estaríamos haciendo un uso ineficiente de memoria.*
2. *Necesitamos algo que pueda ser determinado a tiempo real. Las listas anidadas nos dan la facilidad de reservar espacio de memoria dentro del "heap" de un programa, y sólo, cuando sea necesario. Por contra, esta segunda opción no termina de cubrir nuestras necesidades, de algún modo perdemos una dimensión, no somos capaces de establecer una clasificación por niveles (direcciones IP, puertos, función, longitud, identidad). Esto hace que esta segunda idea no termine de encajar.*
3. *Pero, ¿qué tal si extendemos la idea de listas anidadas buscando obtener una dimensionalidad extra? Utilizando listas anidadas, pero añadiendo un segundo puntero (un puntero para los datos de un mismo nivel y otro puntero para los datos del nivel/tipo inmediatamente inferior) guardaríamos datos, siempre y cuando, fuera necesario mediante la asignación dinámica que caracteriza a las listas anidadas, utilizando de manera eficiente la memoria; pero además tendríamos la información almacenada por niveles obteniendo tal dimensionalidad buscada.*

Basándonos en un diseño de listas anidadas simple y unidimensional, lo primero que se me ocurre es presentar el "struct" que se usa para almacenar datos en este método de almacenamiento:

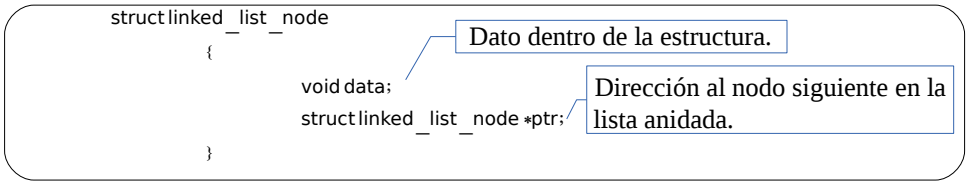


Fig.7 Nodo en una lista anidada

Ejemplificándolo, pudieran existir los 5 nodos siguientes en una lista anidada:

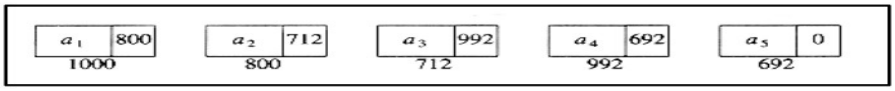


Fig.8 Ejemplo de lista anidada

Cada nodo estará representado por un "struct" como el anterior: cada a_i representa el dato dentro de la estructura y el número contenido en el segundo recuadro (800, 712, 992, 692, 0) es la dirección de la siguiente estructura dentro de la lista anidada. Los números bajo los recuadros internos son la dirección de la estructura de datos dibujada sobre ellos.

Las estructuras serán emplazadas en memoria mediante asignación dinámica[AlWe0], siendo esto la ventaja frente a la utilización de arrays: mediante asignación dinámica somos capaces de usar únicamente la memoria necesaria, sin desperdiciar, o hacer corto de la misma.

Volviendo a la idea de "Árbol" y uniendo diseño y requisitos de usuario: en el punto dos, se habla sobre la recogida de información de campos del paquete IP (direcciones IP) y segmento TCP (puertos); en el punto tres, incluimos Modbus al diseño, en cuyo caso, nuestro "Árbol de Listas Anidadas" se deberá extender varios niveles "hacia abajo" con el fin de dar cabida a tres campos extra (función, identidad y longitud). Así, vemos un cierto numero de niveles que se corresponden con el tipo de datos que la variable en sí contiene. Ya de aquí, extendiendo la idea de listas anidadas mediante la añadidura de un puntero más, obtenemos la idea de "Árbol de Listas Anidadas", que es capaz de cubrir a la perfección nuestras necesidades de almacenamiento.

La siguiente figura muestra una posible realización de un “Árbol” que acumula datos IP-TCP:

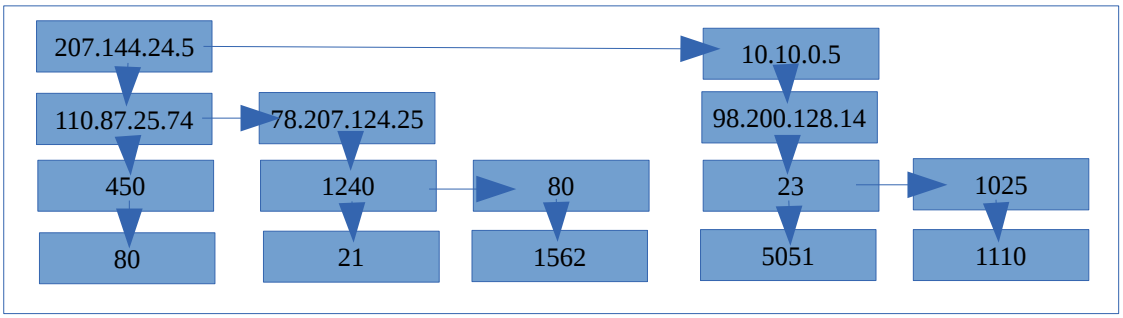


Fig.9 Representación “reducida” de un “Árbol de listas anidadas”

La idea es que cada vez que el programa encuentre en un paquete una nueva combinación de los campos *dirección Ip fuente, dirección Ip destino, número de puerto fuente y número de puerto destino*, respecto a combinaciones anteriormente vistas, deberá incluir nuevos nodos en el árbol.

Explicando la figura anterior en la que se mostraba un pequeño Árbol:

- Llega un primer paquete con direcciones Ip fuente 207.144.24.5, dirección Ip destino 110.87.25.74, número de puerto fuente 450 y número de puerto destino 80. Como la lista estaba vacía, esta combinación de datos no están aun almacenados, así que los guarda. Reservando para ello espacio y completando los campos de nuestra estructura de datos.
- Llega un segundo paquete con direcciones Ip fuente 207.144.24.5, dirección Ip destino 78.207.124.25, número de puerto fuente 1240, número de puerto destino 21. Como la dirección Ip fuente ya la tenía y el primer campo nuevo en aparecer es la dirección Ip destino; a partir de este nodo, se plantea una nueva conexión desde este nodo hacia una posición de memoria que almacene la nueva dirección Ip destino y, a partir de ésta, el resto de campos de información (número de puerto fuente y número de puerto destino).
- Siguiente paquete: Ip fuente 207.144.24.5, dirección Ip destino 78.207.124.25, número de puerto fuente 80 y número de puerto destino 1562. Los nuevos campos son los números de puerto. Desde el nodo que contiene la información de dirección Ip destino 78.207.124.25 se añade un nuevo nexo a otro nodo que contenga la información del nuevo número de puerto fuente y de este, un nexo a otro nodo con la información de número de puerto destino.

Y así, este proceso continuaría, comparando la información ya presente con la información que llega en el nuevo paquete. El árbol, puede ser tan extenso como nueva información vaya llegando. ¿Cuánta información debe ser almacenada? Eso dependerá del diverso número de máquinas y del número de conexiones diferentes que se establezcan entre ellas.

La estructura de datos que utiliza cada nodo del “Árbol” presentado es la siguiente:

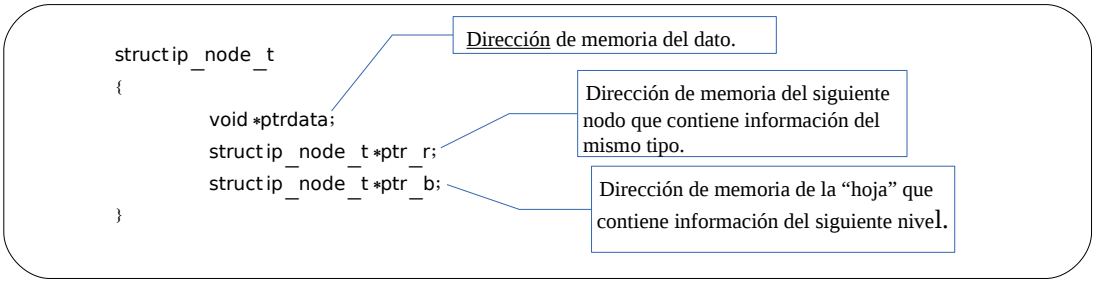
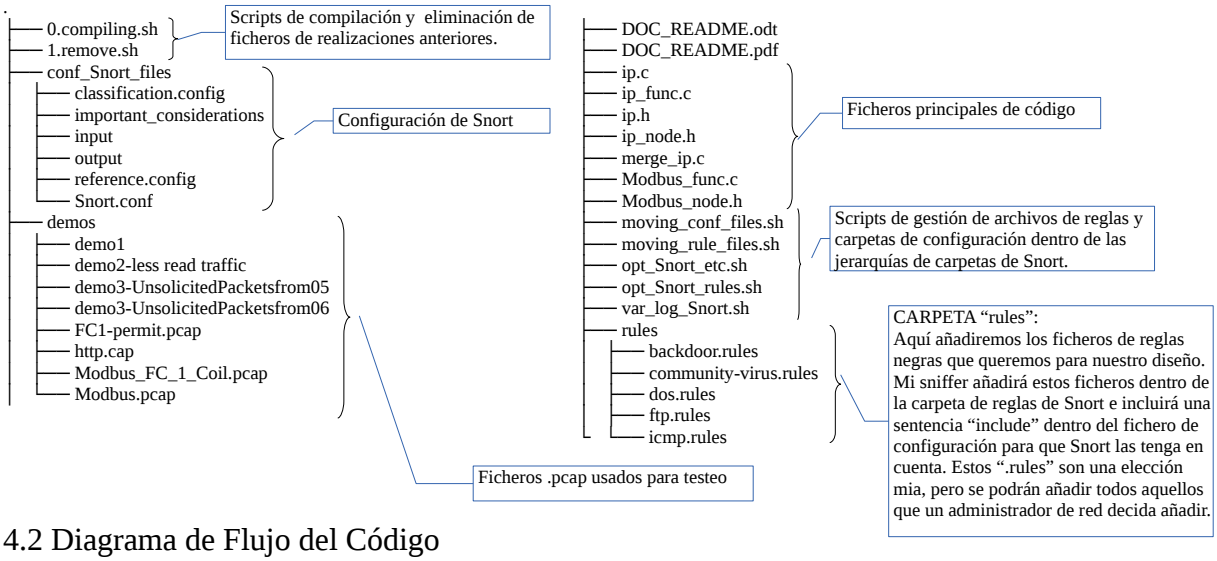


Fig.10 Nodo del Árbol de Listas Anidadas

4. Implementación

4.1 Código

Los archivos que componen el software creado durante estos meses para el proyecto:



4.2 Diagrama de Flujo del Código

¿Cómo conectan los archivos previamente expuestos?

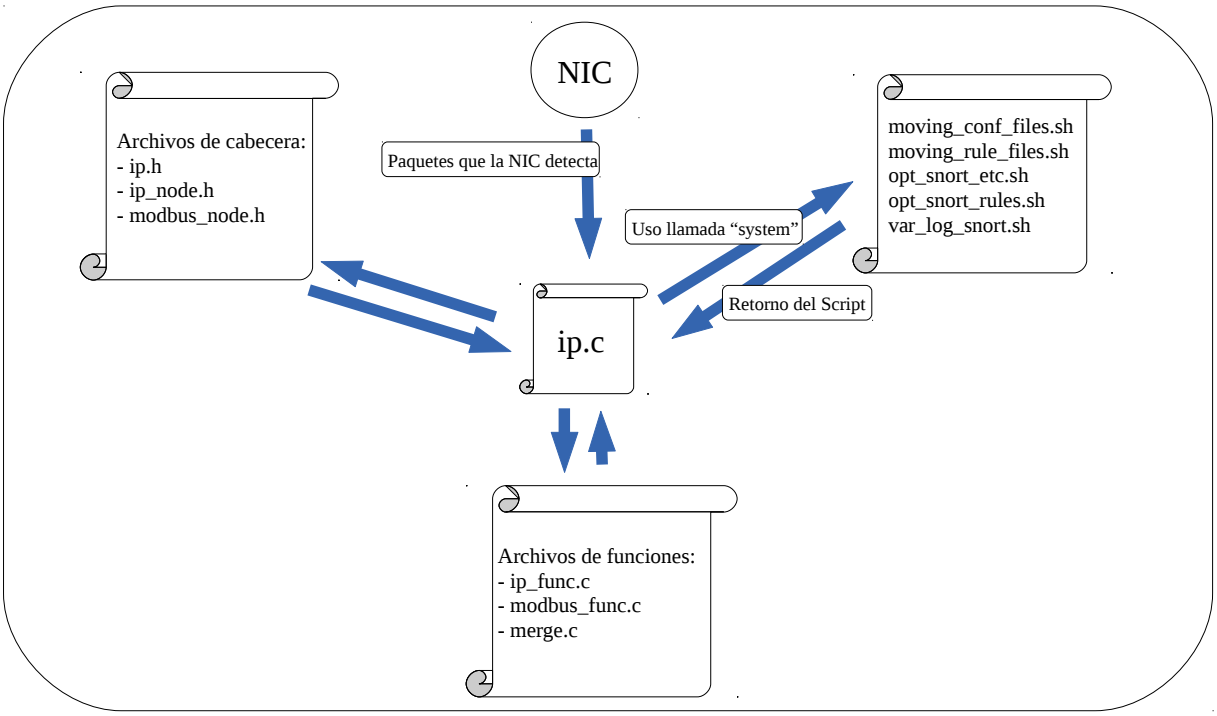


Fig.11 Visión global

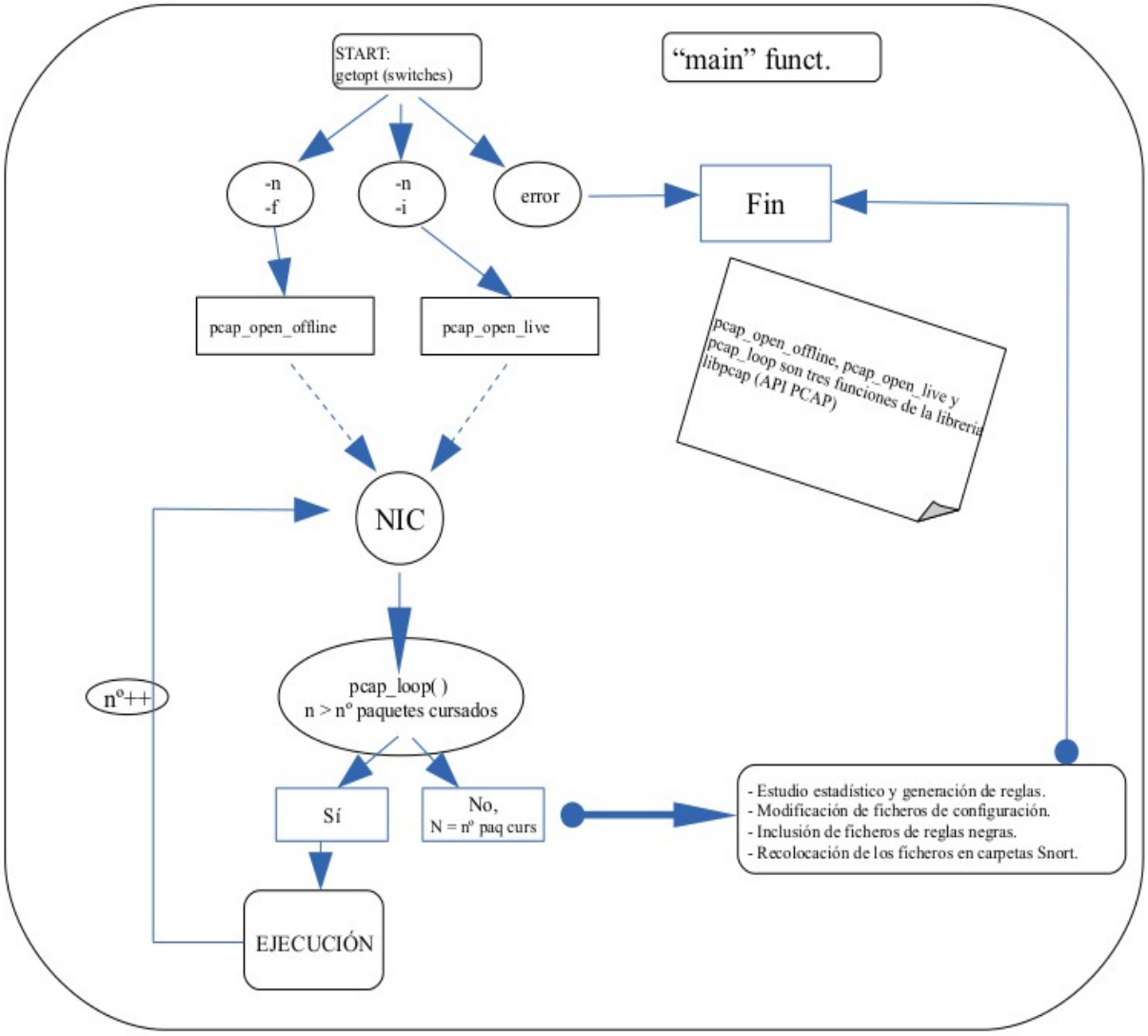


Fig.12 Pcap_loop

En este punto tenemos la estructura general del programa. Primeramente se comprueban los switches con los que se invoca al programa: -n, -i, -f. Con los switches -f, y -i se indica desde dónde va a "leer" el sniffer (desde fichero .pcap o desde interfaz de red); en el caso de leer desde .pcap, éste se "toma" como si fuera la misma interfaz de red (NIC), pero resulta interesante entender la distinción. Para la fase de testeo este switch -f es de especial importancia.

Usando las funciones de librería "pcap_open_" se ha abierto un descriptor de flujo que nos sirve para trabajar con el "interfaz" como si trabajásemos con un fichero cualquiera; es aquí donde comenzamos a hacer uso de la abstracción que la API PCAP nos brinda.

El switch -n indica el número de paquetes que conformarán la fase de estudio del segmento de red, el software deberá trabajar hasta que se hayan registrado/estudiado dicho número de paquetes. Una vez llegado a dicho número, se termina dicha fase de estudio y se comienza la organización de ficheros estadísticos y de reglas como se ha indicado en el recuadro inferior más a la derecha.

Profundizamos en la ejecución del programa:

- Volcado a estructuras de datos, uso ip_func.c y Modbus_func.c para generación de Árboles y obtención de primeras estadísticas (statistics.txt) y datos de “sniffing” (sniff_data.txt) :

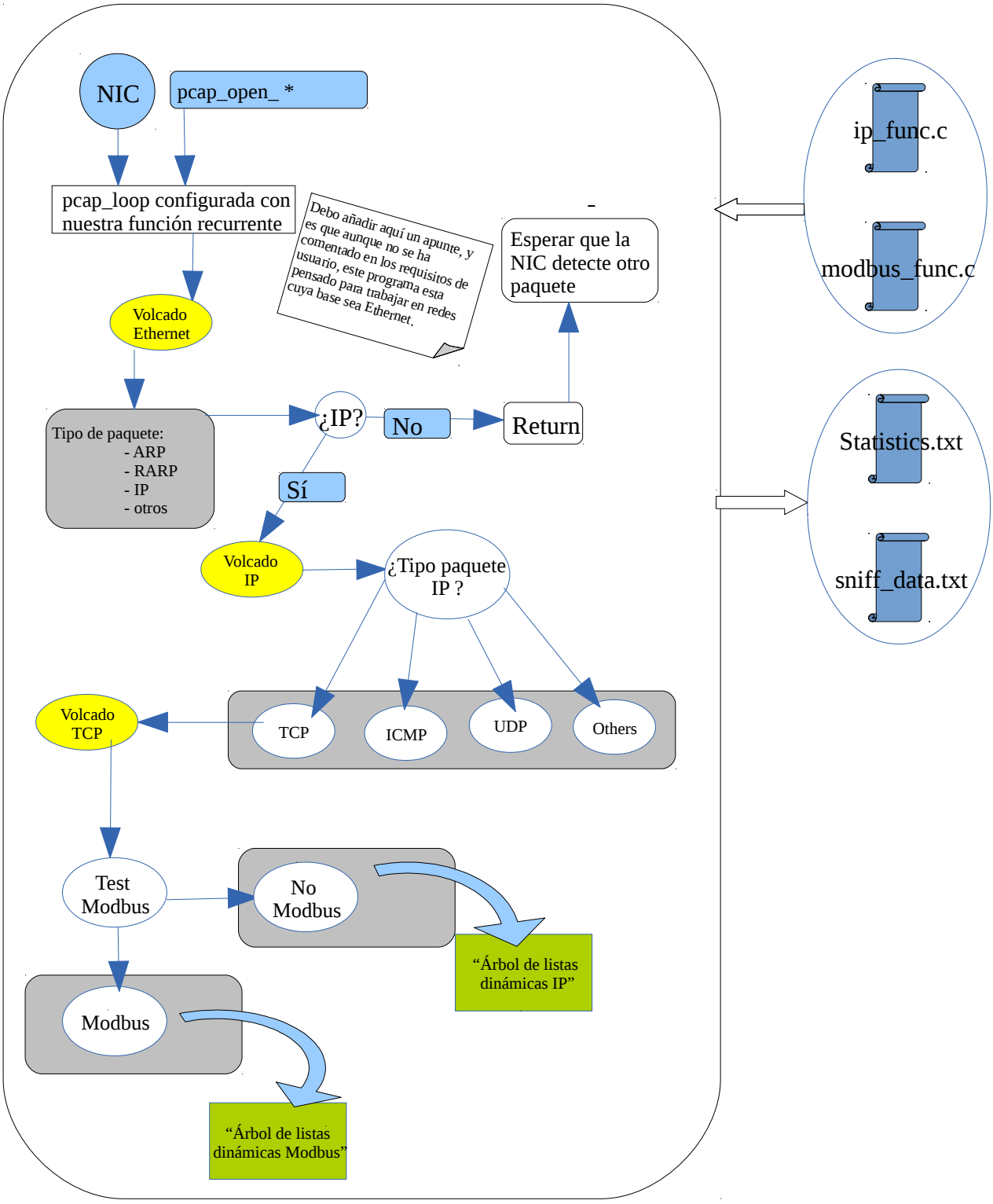


Fig.13 Carga en Árboles

- Uso de funciones en ip_func.c: “Árbol de listas dinámicas IP”.

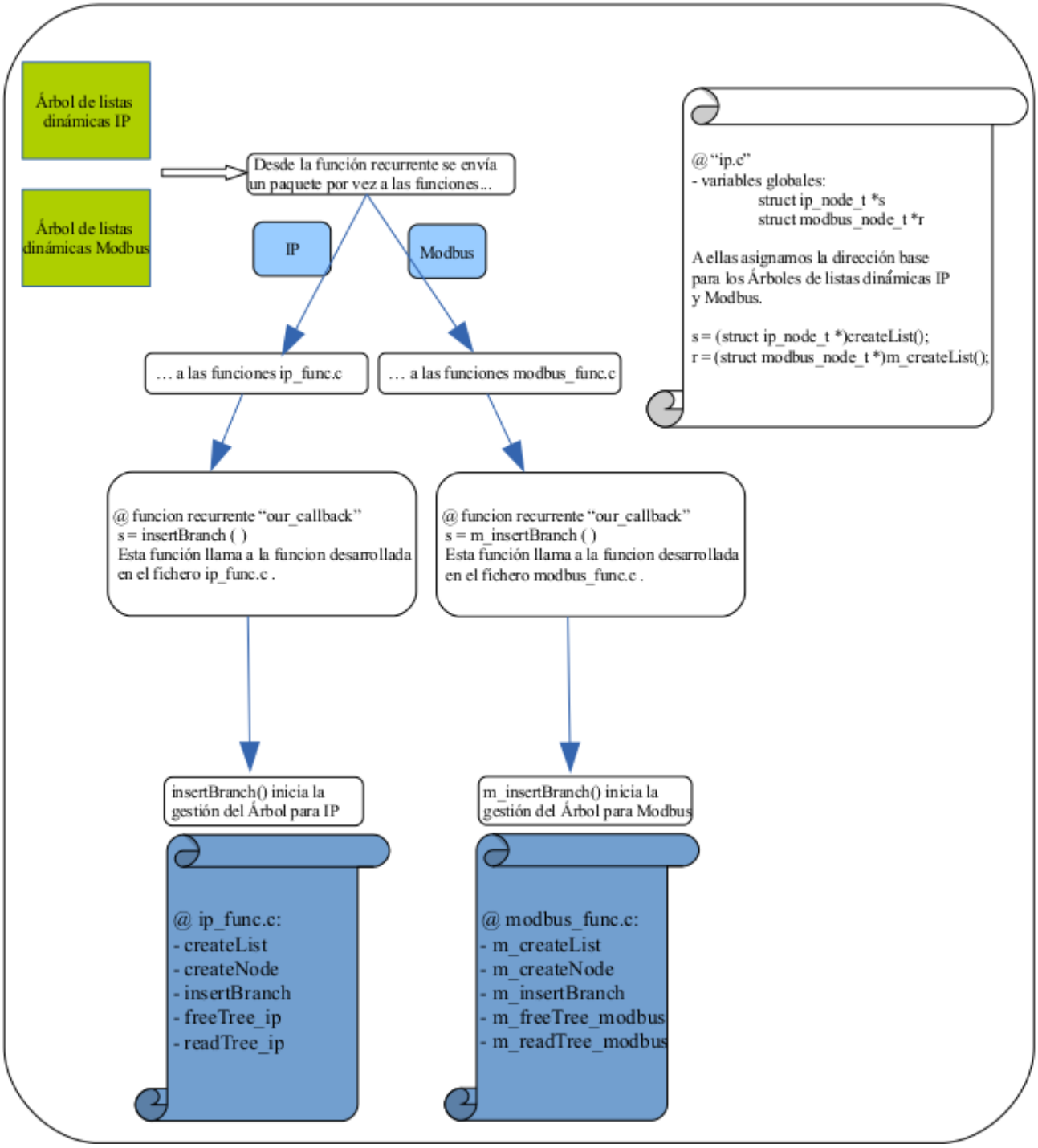


Fig.14 ip_func.c & Modbus_func.c

- Dentro de ip_func.c y Modbus_func.c:

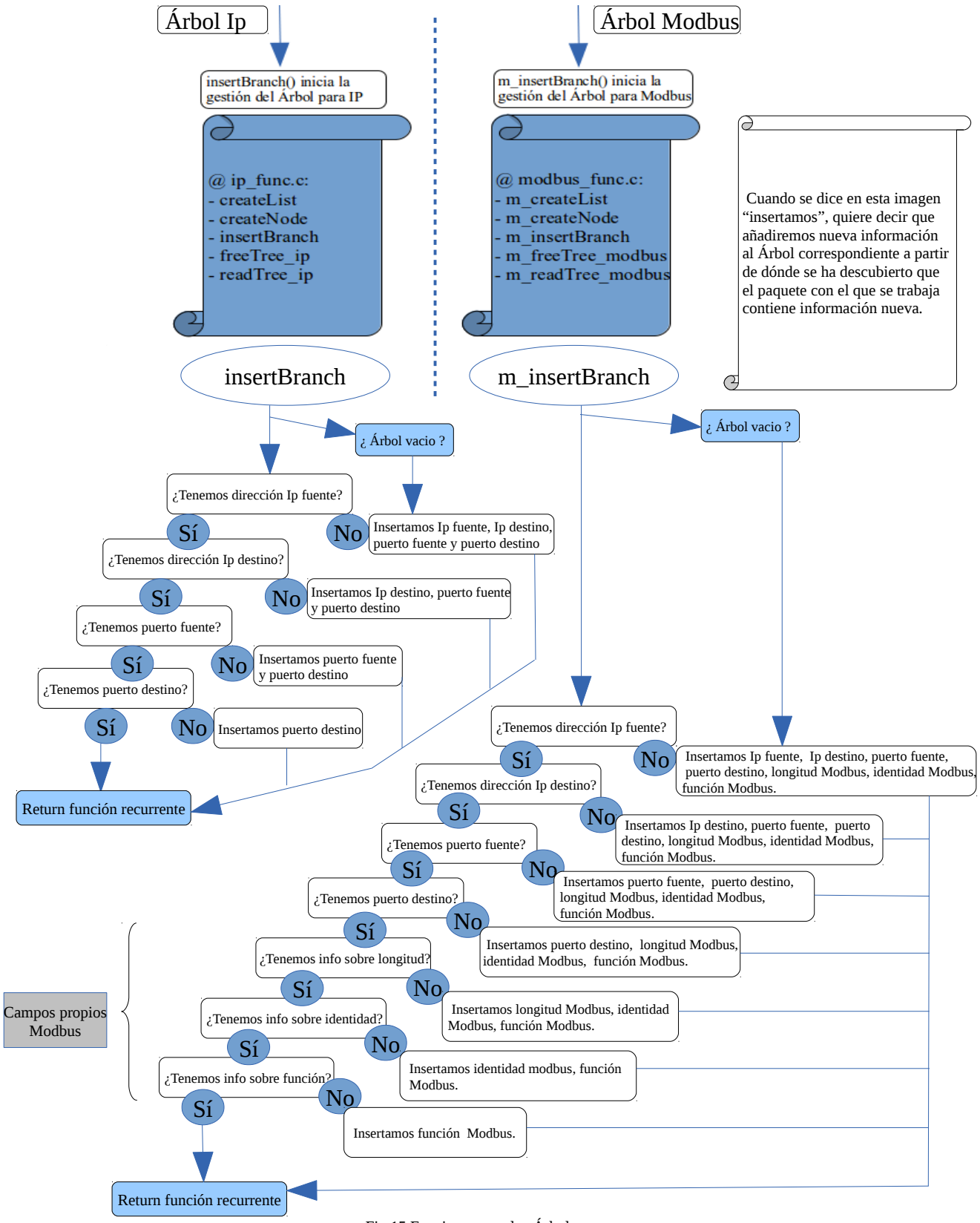


Fig.15 Funciones para los Árboles

- Funcionamiento general:

Entendidas las anteriores partes podemos conectarlas y tener una visión completa del funcionamiento.

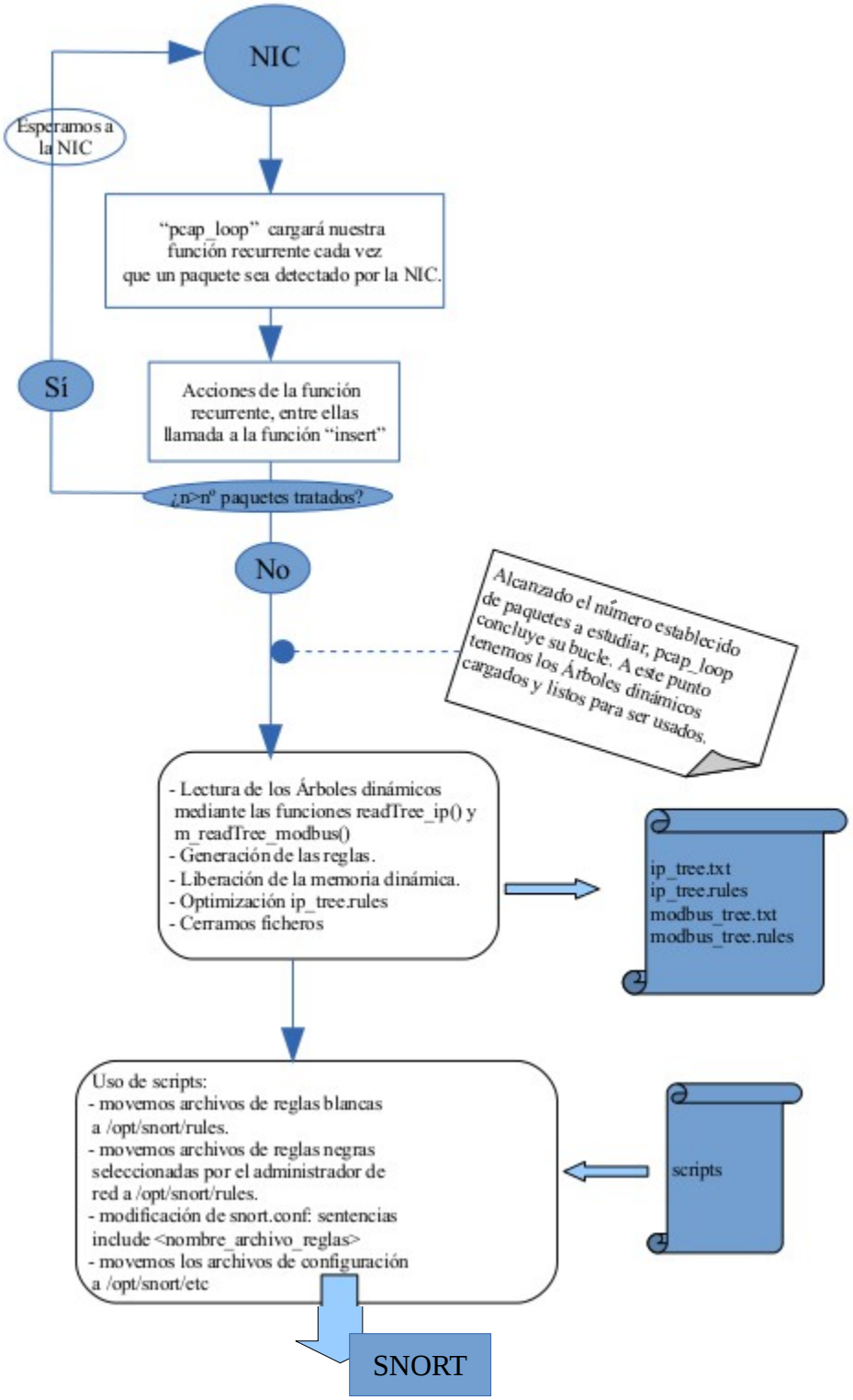


Fig.16 Visión del resultado final

4.3 Diagrama de Flujo del Funcionamiento del Programa.

Haremos uso de los scripts creados para borrar y compilar. Primeramente, realizamos un borrado de ficheros, resultado de alguna ejecución previa (1.remove.sh), tras lo cual ejecutaremos el script 0.compiling, que compila y enlaza ficheros “objeto” para crear el ejecutable: sniffer.out.

De este modo:

```
root@arturo-laptop:/home/arturo/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus# ./1.remove.sh
...remove.sh done
root@arturo-laptop:/home/arturo/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus# ./0.compiling.sh
...compilation process done
...usage:         ./sniffer -i <interface> -n <number of packets>
...usage:         ./sniffer -f <file_name>
root@arturo-laptop:/home/arturo/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus#
```

Fig.17 Ejecución scripts iniciales

Tras el proceso de compilación, vemos una pequeña información sobre cómo usar el programa. Como en puntos anteriores se hizo mención, el programa puede funcionar leyendo desde la interfaz de red o desde un fichero “pcap” que contenga información recogida en una captura anterior con programas del tipo wireshark o tcpdump.

Los switches son bastante autodescriptivos:

- “-i” : switch utilizado para indicar qué interfaz de red es la que se estudiará.
- “-n” : número de paquetes que implica el estudio. Una vez “sniffados” y estudiados “n” paquetes, el programa concluirá y comenzará la etapa de creación de ficheros de información y de reglas.
- “-f” : se usa para indicar la ruta del fichero “pcap” a estudiar.

Diagrama de flujo para “sniffer” en modo interfaz de red:

./sniffer -i wlan0 -n 1000 — Estudiaremos el tráfico relacionado con la interfaz wlan0, en concreto 1000 paquetes de información.

En una primera fase del programa se nos presentan banners informativos (sobre instalación de Snort e inputs y outputs del sniffer).

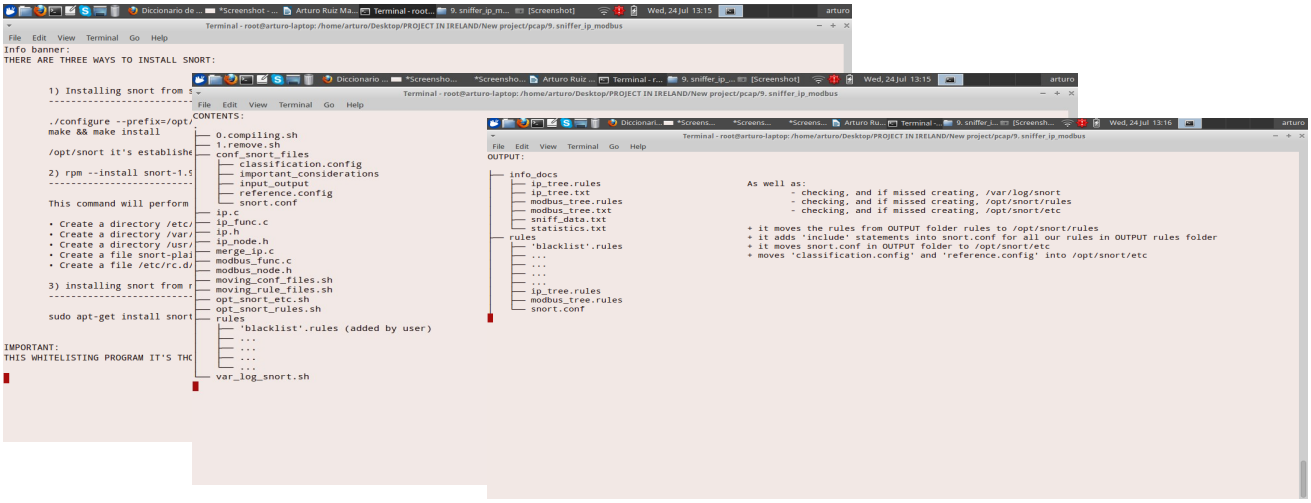


Fig.18 Banners de inicio

Comprobación de la existencia de carpetas importantes dentro del sistema de carpetas de Snort y que será donde almacenemos los outputs de nuestro sniffer.

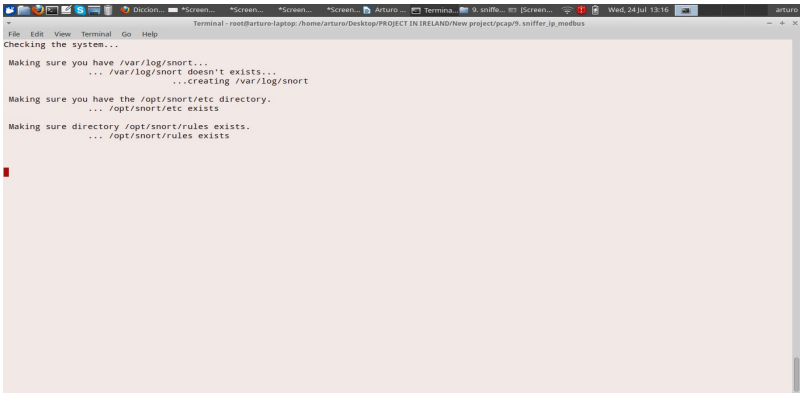


Fig.19 Comprobación existencia de carpetas

Comienza el proceso de “sniffado”: al concluir éste, se nos informa sobre cuánto tiempo ha necesitado para detectar y estudiar los “1000” paquetes que le hemos solicitado. Es ahora, cuando comienza el proceso de organización: movimiento de ficheros de reglas resultado del sniffing, movimiento de ficheros de configuración, modificación del fichero de configuración de Snort...

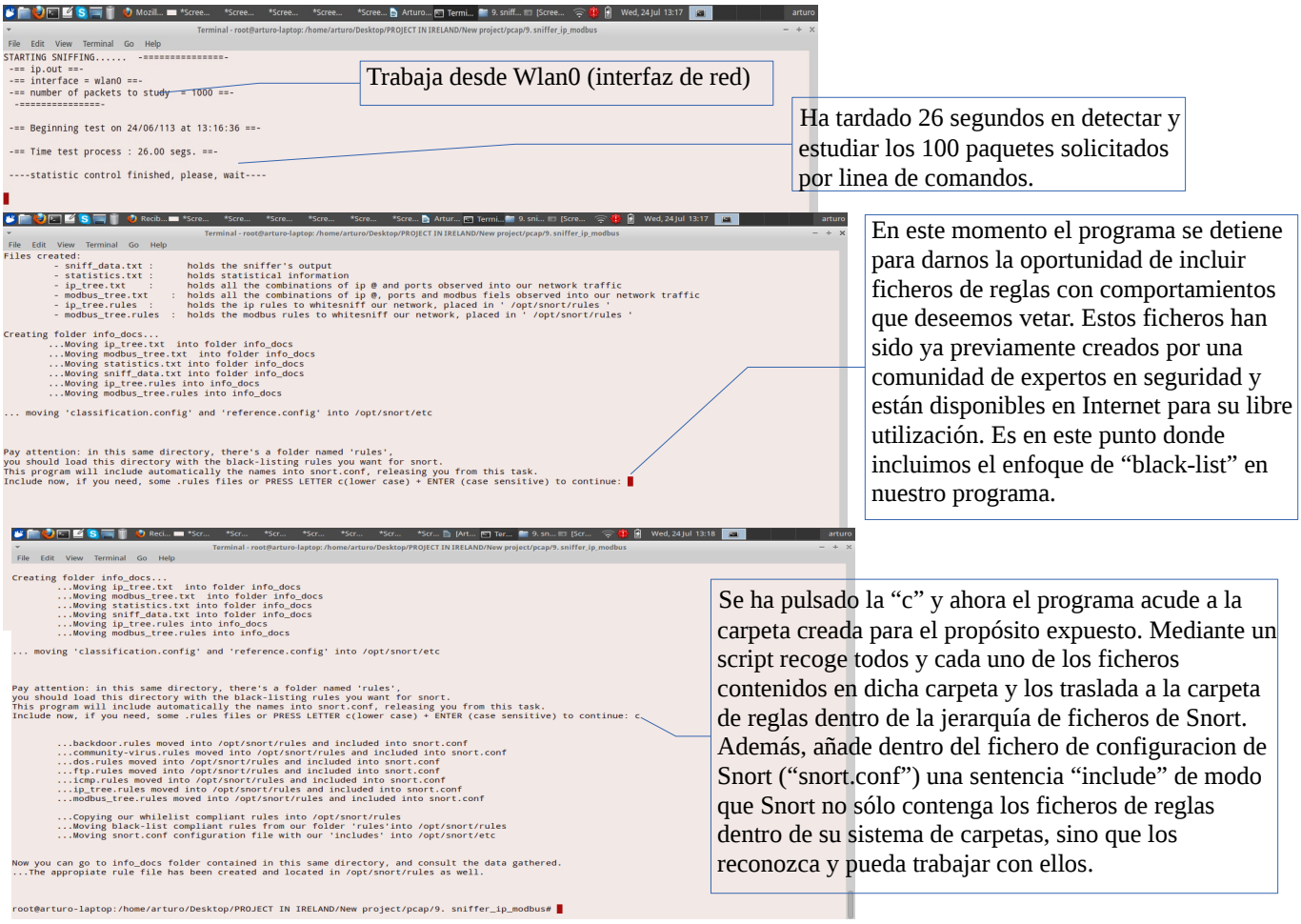


Fig.20 Distintas pantallas en el funcionamiento del programa

Volviendo sobre ficheros resultado de nuestro programa:

Fig.21 Output de nuestro programa

- ip_tree.txt: fichero que contiene la información rescatada en el árbol dinámico para IP-TCP.
- ip_tree.rules: contiene la misma información que el fichero anterior ".txt" pero expresada de la forma en que Snort será capaz de entender. Es este fichero, uno de los varios ficheros que copiaremos (a través de los scripts que conforman nuestro sniffer) dentro de la estructura de carpetas de Snort, que se empleará en el enfoque "white-listing".
- Modbus_tree.txt: equivalente fichero a ip_tree.txt pero con la información relativa al árbol dinámico para Modbus.
- Modbus_tree.rules: fichero de reglas para Modbus que el programa copiará dentro de los directorios Snort y que servirán para contribuir al enfoque de listas blancas.
- sniff_data.txt: para no llenar la pantalla de números y letras que ni siquiera vamos a ser capaces de leer, la visualización de los campos de interés de los paquetes, se envían a un fichero que se podrá visualizar en cualquier momento deseado.
- statistics.txt: en este fichero se incluyen estadísticas temporales. Contribuye a añadir información y completar el "dibujo" de la red. Podría resultar de bastante interés en el desarrollo de aplicaciones futuras dentro de este mismo programa.

5. Testeo

5.1 Eficacia que No Eficiencia

¿Cómo comprobar los límites de velocidad de un software, cuando estos están sujetos a la capacidad de un hardware? La respuesta a esta pregunta pasa por un estudio estadístico sobre el número de paquetes perdidos una vez instalado nuestro sniffer en distintos hardwares. Aún así, dicho estudio estadístico está sujeto a tantas variables (tarjeta de red, RAM, velocidad del procesador, …) que a mi parecer resultaría una pérdida de tiempo y no obtendríamos un resultado útil.

Las necesidades hardware son indudablemente un aspecto importante a tener en cuenta, pero no existe una guía que te diga cuántos gigahercios, gigabytes, megabits.. debe tener el computador donde instalamos el software [BeJS01]. Las dimensiones del hardware que éste debe tener, dependerán en gran medida de cuánto tráfico querremos controlar, o dicho de otro modo, cómo de "bulliciosa" va a ser nuestra red.

No debemos perder de vista que nos interesa "comprobar" el mayor, si no el total, número de paquetes que fluyen por nuestra red. Aquí es donde entra la idea de "eficiencia". Si el administrador de la red decide instalar el software en un hardware que no es capaz de estar a la altura del nivel de trabajo de la red que se propone controlar, perderá muchos paquetes y por lo tanto el nivel de seguridad ofrecido se reduce considerablemente. Las capacidades del hardware en definitiva, estarán sujetas al tráfico a monitorizar y vendrán determinadas mejor o peor, por lo mejor o peor que ese administrador

de red sea. Importante entonces será cuán bien el administrador conoce su red.

Como recomendación, no se debería instalar, ya sea nuestro software, como Snort, en un hardware antiguo; en su lugar, es una práctica muy común adquirir un computador de altas prestaciones e instalar el paquete software-sensor en él. Una vez que dicho computador no cubre las expectativas de "monitorización", se puede colocar a realizar otras funciones más generales dentro de la empresa.

5.2 Testeo del Software

Al principio de este texto, hemos expuesto unos requisitos parciales de usuario ("Requisitos para la Creación del Software"), etapas que yo mismo me he marcado; de modo que, dividiendo el problema grande en problemas más pequeños, podamos llegar a una solución final satisfactoria. A la hora de testear pues, debemos ir testeando a la finalización de cada una de dichas etapas, para comprobar que el resultado logrado es el esperado y que vamos en la dirección correcta.

5.2.1 Testeo del Sniffer Base y Recogida de Datos IP-TCP y Modbus

- Dos aspectos importantes a testear respecto al primer requisito parcial de usuario serán:

- Comprobamos que el programa conecta correctamente con la interfaz establecida en linea de comandos y que capta los datos correctamente. Esto se realizará a tiempo real a través del switch "-i <NIC>".
- a través de un fichero ".cap" usando el switch "-f": comprobación de un "sniffing" correcto. Para esto utilizaremos ficheros ".cap" que pueden ser a la vez abiertos con otros programas como Wireshark, de este modo comprobaremos que los resultados obtenidos por nuestro sniffer son correctos.

- Entrando ya en los requisitos parciales segundo y tercero:

Fue por motivos de testeo que añadí la alternativa de funcionamiento para mi "sniffer" desde ficheros pcap "*./sniffer -f <nombre_fichero>*". Y es que existen ya de por sí multitud de ficheros pcap creados para propósito en la red [7][8].

Primero quiero comprobar si mi software es capaz de discernir entre paquetes IP-TCP Modbus y no Modbus. Para esto descargamos un fichero pcap que contenga un cierto número de paquetes de todo tipo y lo abrimos con el Wireshark. Wireshark, como herramienta ya consolidada[], nos da la seguridad de realizar análisis completos y correctos sobre un tráfico.

Acudimos al menú "statistics" y dentro de él a la opción "Protocol Hierarchy Statistics": una ventana que nos informa de los protocolos que se ven en el tráfico junto con el número tanto de bytes como de paquetes de cada tipo. A nosotros nos interesa comparar el número de paquetes de cada tipo, con el número de paquetes que nuestro sniffer registra.

Ejemplo de prueba: 1. Invocamos wireshark con un fichero cualquiera previamente descargado, para este ejemplo "Modbus_FC_1_Coil.pcap". Se trata de un fichero pcap muy sencillo que simplemente me ayude a mostrar el método de testeo seguido para la comprobación del correcto funcionamiento del software.

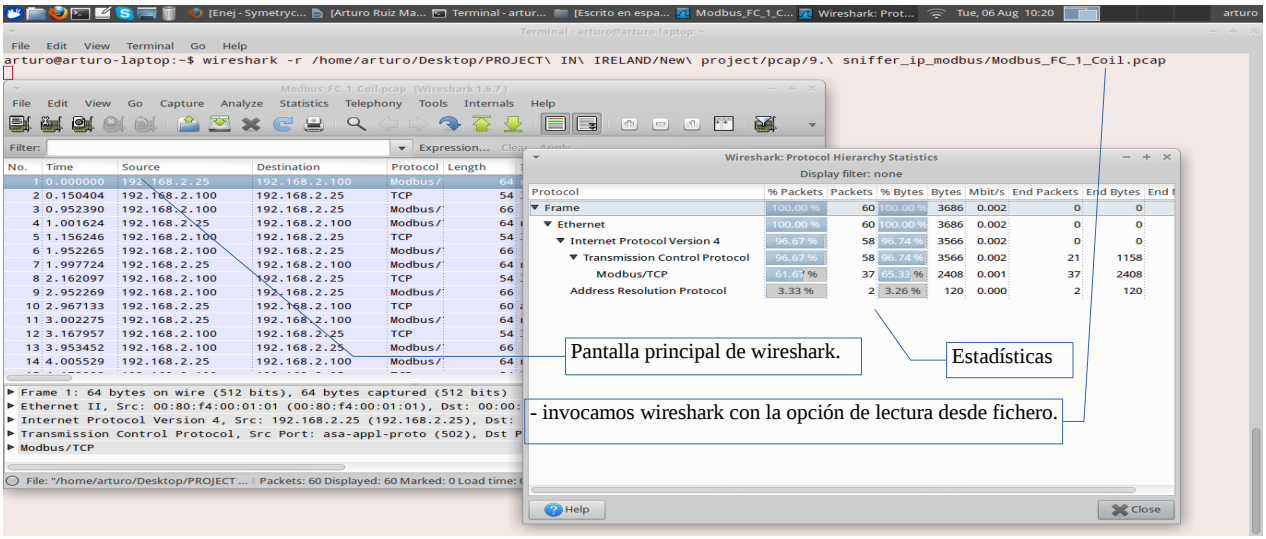


Fig.22 Estadísticas Wireshark

Invocamos nuestro programa de modo que lea el mismo fichero pcap (*./sniffer -f Modbus_FC_1_Coil.pcap*) y dejamos que el programa realice su función. Una vez finalizado, acudimos a la carpeta donde se recogen los resultados y abrimos el fichero de texto "statistics.txt".

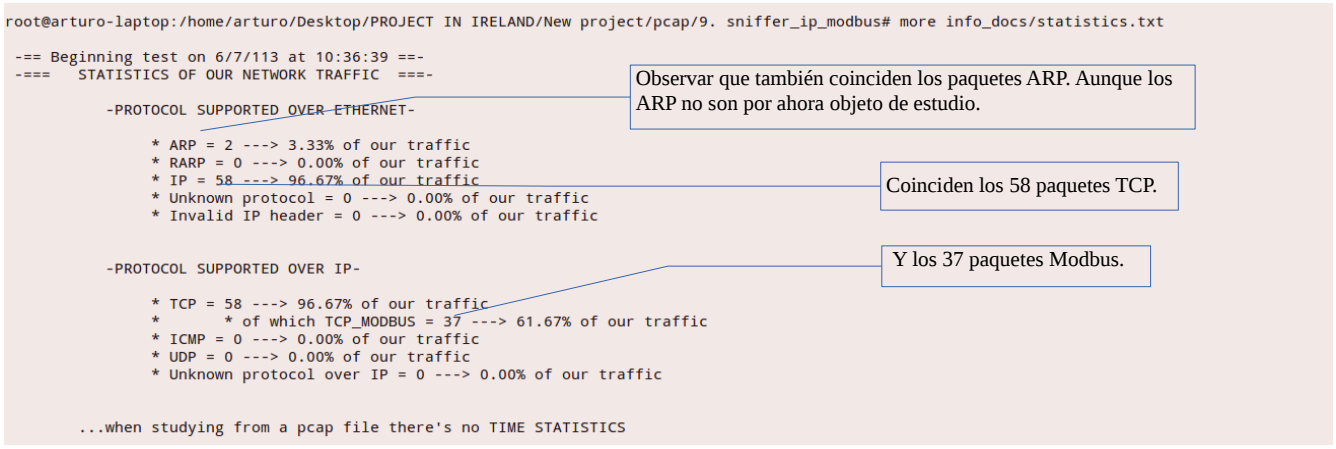


Fig.23 Fichero de estadísticas de nuestro sniffer

Ahora bien, deberemos repetir esta prueba un cierto número de veces, con el fin de detectar errores si los hubiera. ¿Cuántas veces? No existe un método que te de la seguridad de que realizando esta prueba "x" veces, te aseguras un 100% el correcto funcionamiento del software. Yo particularmente, he probado el software con 20 ficheros pcap diferentes. He procurado eso sí, que los ficheros pcap, contengan una cantidad suficiente de variaciones, representando diversos escenarios, verificando que el programa detecta todas ellas. De este modo, puedo estar tranquilo, sobre el correcto funcionamiento del programa. A lo largo del tiempo de vida de un programa pueden detectarse nuevos fallos conocidos como "bugs" y que van siendo corregidos en nuevas versiones del software. Por ahora, me siento muy satisfecho con las pruebas realizadas y puedo afirmar que el software funciona.

Continuando con el testeo a través de ficheros pcap, lo siguiente será comprobar que tenemos registradas las direcciones y puertos correctamente. De nuevo, Wireshark ofrece una opción para

rescatar que direcciones IP y puertos se ven sobre el tráfico: Menú “statistics” y opción “Endpoints”. Acudimos a la pestaña “TCP” donde veremos la dirección y el puerto.

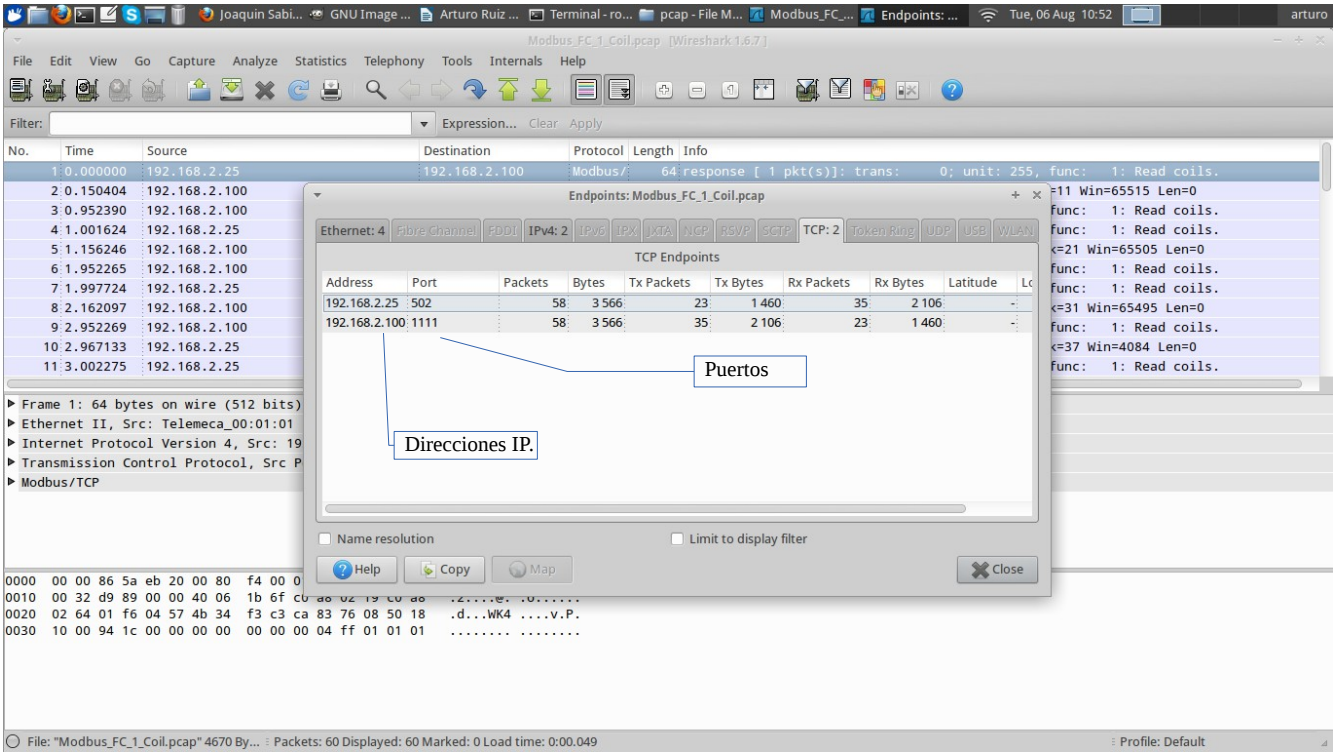


Fig.24 Wireshark estadísticas “endpoints”

Para verificar sobre este punto nuestro sniffer, abrimos el fichero “ip_tree.txt” generado con mi sniffer:

```
root@arturo-laptop:/home/arturo/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus# more info_docs/ip_tree.txt
    0: Ip source:   192.168.2.100, Ip destination:    192.168.2.25, Port Source:  1111, Port destination:   502
    1: Ip source:    192.168.2.25, Ip destination:   192.168.2.100, Port Source:   502, Port destination:  1111
```

Fig.25 Fichero de información IP-TCP

De nuevo podemos ver el mismo resultado en ambos programas.

En el requisito parcial de usuario cuarto, entramos en la elaboración de reglas Snort, comprobaremos que los datos obtenidos en “ip_tree.txt” y que han sido verificados como correctos, son traducidos a “reglas Snort”:

Una única regla que recoge ambas direcciones ip con los dos puertos. Esta regla permitirá el tráfico que nuestro sniffer ha registrado como “normal” durante su etapa de “entrenamiento”.

```
root@arturo-laptop:/home/arturo/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus# more info_docs/ip_tree.rules
pass ip 192.168.2.100 1111 <> 192.168.2.25 502
alert ip any any -> any any (msg:"communication out of our ip-white-list";)
```

Mientras que el resto del tráfico que no ha encajado con la regla “pass” debera ser alertado: “alert”.

Fig.26 Fichero de reglas IP-TCP

La comprobación de la correcta recogida de información relativa a Modbus es más compleja y wireshark parece no ofrecer estudios estadísticos al respecto sobre campos concretos de Modbus. Para resolver este problema, se ha ideado un entorno controlado que simule una red real de máquinas

funcionando bajo el protocolo Modbus-TCP.

5.2.2 Testeo mediante “Sniffing” en un Entorno Controlado

Para esta fase de testeo del proyecto, se ha recreado un pequeño entorno SCADA que simula el funcionamiento de una linea de manufacturación. Dicho entorno, está controlado por un PLC (Programmable Logic Controller) y un conjunto de módulos Modbus TCP.

La linea de manufacturación consiste en una cinta transportadora que transfiere ladrillos desde un cubo hasta un punto de recolección. En el punto de recolección un brazo robótico articulado recoge los ladrillos y los coloca en otra cinta transportadora que coloca los ladrillos en otro cubo de almacenamiento. En total tenemos 4 motores (dos para el brazo articulado y otros dos, uno por cinta transportadora) cuyo funcionamiento está controlado por switches que activan y desactivan relés.

En definitiva se trata de un diseño simple pero que utiliza el protocolo Modbus y que posibilita el testeo del “sniffer” tanto de modo interactivo como a través de ficheros pcap extraídos de su funcionamiento. Esta alternativa de creación de ficheros pcap que recogen el funcionamiento de una realización del entorno es algo muy positivo, ya que permite rescatar la información contenida en los campos de los paquetes para un estudio más exhaustivo de las conexiones entre máquinas.

Mediante este entorno, somos capaces de generar un gran número de variaciones en el funcionamiento de los dispositivos Modbus y comprobar que el software funciona en todos los casos. El sensor Snort junto con el programa sniffer, se instala en un computador a parte, con sistema operativo Linux, que se conecta por medio de una interfaz de red a dicho entorno. Se ponen a funcionar las máquinas, se generan una serie de variaciones y se comprueba que los datos recopilados por el programa son los que debieran ser.

5.2.3 Testeo del Funcionamiento de los Scripts

Ya por último, deberemos confirmar que los scripts realizan su cometido. Existen ciertas carpetas dentro de la jerarquía de carpetas de Snort, donde se deben guardar los ficheros de configuración y los archivos de reglas. Esta última comprobación consiste en tan sólo acceder a dichas carpetas y comprobar que contienen los archivos que deben contener.

En concreto, el archivo “Snort.conf” modificado por el software, deberá estar contenido en /opt/Snort/etc y por otro lado los archivos de reglas deberán estar en /opt/Snort/rules. Además, los “logs” generados por Snort, se almacenarán en /var/log/Snort, carpeta que deberá estar creada si no de instalación Snort, sí por mi programa.

En definitiva, se trata de un último paso de testeo pero que garantiza el completo y correcto funcionamiento del software en su totalidad.

6. Conclusiones

El motivo del presente proyecto nos ha llevado a diseñar un método para la realización automatizada de listas blancas para Snort basado en la idea de DPI (Deep Packet Inspection).

A su vez, y buscando diseñar una herramienta más completa, se busca añadir al método un enfoque de “listado-negro”, representado a través de listas negras ya publicadas en Internet, y que al ser de libre distribución, puedo usar a mi antojo.

Así pues, este método está “instanciado” a través del software que durante el desarrollo de este texto he presentado y que he probado de manera exitosa. Éste, se ha ido desarrollando siguiendo una

sucesión de etapas marcadas por los requisitos parciales de usuario y que culminan en un programa que es capaz de satisfacer la motivación del proyecto.

De cara a un futuro, se plantea realizar mejoras al software:

- Uso de un mayor número de preprocesadores en las reglas Snort que permitan una mejor definición del tráfico permitido.
- Generalización del software a otros protocolos de Entorno Industrial.
- Posibilidad de incluir en el diseño el desarrollo de reglas para otros IDS.
- Tratamiento y estudio del “payload” del paquete Modbus.

Respecto a la inclusión al diseño de técnicas para la Detección de Anomalías, se estudió la posibilidad de inclusión de estudio estadístico del paquete a través de N-gramas, alternativa que se descartó debido al gran trabajo que ésto necesitaría y lo imperfecto del método. El problema del estudio estadístico a través de N-gramas se basa en su incapacidad para distinguir entre tráfico “bueno” y “malo”, creando demasiados falsos positivos, ya que no tiene en cuenta la estructura del paquete y los diferentes campos de la cabecera del mismo. En su lugar, lo que mi diseño hace es un “Deep Packet Inspection”, comprobando las características de diversos campos del paquete, lo que además constituye una diferencia importante entre mi enfoque y otros enfoques.

Snort es una herramienta de software libre, cuyo código está abierto a posibles modificaciones: una de las alternativas de las que se dispone es, precisamente, la implementación de plug-ins que funcionen con Snort, pero realizando tareas concretas, para las cuales, dicho IDS software, aún no tiene herramientas específicas. Futuras inclusiones al diseño presentado durante este trabajo, pasarían entonces por programar preprocesadores que permitan realizar estudios más detallados del paquete, siendo usados directamente en las reglas, permitiendo unas reglas más concretas.

El campo de la Detección y Prevención de Intrusiones es un campo dentro de la Seguridad de Redes abierto a nuevas ideas y con un amplio horizonte de desarrollo para los jóvenes ingenieros. Es un campo complejo con continuas mejoras, pero un campo en el cual cada día surgen nuevos problemas. Un campo, que constituye un aspecto muy importante dentro de las Telecomunicaciones y que, a día de hoy, cobra una especial relevancia por lo presente que las redes de comunicaciones están en nuestras vidas, ya sea a través de nuestros ordenadores y móviles conectados a la Red, o a través de Sistemas Industriales y sistemas SCADA que buscan facilitar y mejorar nuestra calidad de vida.

ANEXO A: Código completo

```
#! /bin/sh
# 1.remove.sh

rm *.o 2> /dev/null
rm *~ 2> /dev/null
rm sniffer 2> /dev/null
rm -r ./info_docs 2> /dev/null
rm ./rules/Snort.conf 2> /dev/null
rm ./rules/ip_tree.rules 2> /dev/null
rm ./rules/Modbus_tree.rules 2> /dev/null
rm ./conf_Snort_files/*~ 2> /dev/null
echo "...remove.sh done"

#! /bin/sh
# 0.compiling.sh

gcc ip.c -c
gcc ip_func.c -c
gcc Modbus_func.c -c
gcc merge_ip.c -c
gcc ip.o ip_func.o Modbus_func.o merge_ip.o  -o sniffer -lpcap
echo "...compilation process done"
echo "...usage:\t ./sniffer -i <interface> -n <number of packets>"
echo "...usage:\t ./sniffer -f <file_name>"

//ip.h

#ifndef _IP_H_
#define _IP_H_

#include <time.h>
#include <sys/types.h>
#include <pcap/pcap.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>



#define BUFSIZE 2048
#define APP_NAME "ip.out"
#define SIZE_ETHERNET 14
#define ETHER_ADDR_LEN     6

enum eth_type  {
                ARP=0,
                RARP=1,
                IP=2,
                UNKNOWN=3,
                INV_IP_HEADER=4
};

enum upper_eth_type {
                TCP_NO_Modbus=0,
                TCP_Modbus=1,
                ICMP=2,
                UDP=3,
                UP_UNKNOWN=4,
};


/* data structures for IP */
```

```
#define ETHER_ADDR_LEN 6

/* Ethernet header */
struct sniff_ethernet {
        u_char ether_dhost[ETHER_ADDR_LEN];  /* Destination host address */
        u_char ether_shost[ETHER_ADDR_LEN];  /* Source host address */
        u_short ether_type;                  /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
        u_char ip_vhl;          /* version , header length */
        u_char ip_tos;          /* type of service */
        u_short ip_len;         /* total length */
        u_short ip_id;          /* identification */
        u_short ip_off;         /* fragment offset field */
        u_char ip_ttl;          /* time to live */
        u_char ip_p;            /* protocol */
        u_short ip_sum; /* checksum */
        struct in_addr ip_src;  /* source ip address */
        struct in_addr ip_dst; /* dest ip address */
};

#define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef uint32_t tcp_seq;

struct sniff_tcp {
      u_short th_sport;             /* source port */
      u_short th_dport;             /* destination port */
      tcp_seq th_seq;               /* sequence number */
      tcp_seq th_ack;               /* acknowledgement number */
      u_char  th_offx2;             /* data offset, rsvd */
            #define TH_OFF(th)     (((th)->th_offx2 & 0xf0) >> 4)
u_char  th_flags;                   /* flags */
u_short th_win;               /* window */
u_short th_sum;               /* checksum */
u_short th_urp;               /* urgent pointer */
};


/* Modbus-TCP header */
struct sniff_Modbus_tcp {
        u_short mtcp_trans_id;          /* synchronization */
        u_short mtcp_prot;     /* protocol identifier */
        u_short mtcp_len;      /* remaining! bytes in this frame */
        u_char      mtcp_iden;          /* identifier */
        u_char      mtcp_func;          /* function code */
};



/* protypes */

void our_callback(u_char *,const struct pcap_pkthdr* ,const u_char* );
void print_app_banner(char *,int);
void create_Statistics (struct tm *,struct tm *,double,int,int ) ;
void tail_banner(void);
void pantallazo(int);
void merge_ip(char *);


#endif

// ip.c

#include "ip.h"
```

```
#include "ip_node.h"
#include "Modbus_node.h"


/*global variables for statistics */

        int packet_type[5]={0,0,0,0,0};   /* statistics of ethernet frame type in our network */
        int info_type[5]={0,0,0,0,0};             /* statistics of info type inside the IP payload */
        FILE *statistics;
        FILE *ip_tree;
        FILE *sniff_data;
        FILE *ip_tree_rules;
        FILE *Modbus_tree;
        FILE *Modbus_tree_rules;
        struct ip_node_t *s;
        struct Modbus_node_t *r;



/*
 * MAIN
 *
 */

int
main (int argc, char **argv)
{
        /* vars */
        int     n_packets;                      /* number limit of packets we sniff */
        char    char_aux_i=0,char_aux_n=0;      /* checking flags from the terminal process */

        char    errbuf[PCAP_ERRBUF_SIZE];  /* holds the error string message in pcap functions */
        pcap_t  *handler;                       /* pcap handler */

        time_t  timer_init,timer_end;                   /* for time-stamps */
        struct  tm *st_timer_start,*st_timer_end;       /* for time-stamps */
        double  time_diff;                              /* holds the difference of time the program has used */

        int i;                                  /* for loops */

        //getting options for the program
        int flag_n=0, flag_i=0, flag_f=0;
        int c;
        char *nvalue = NULL;
        char *ivalue = NULL;
        char *fvalue = NULL;

        opterr = 0;

        while ((c = getopt (argc, argv, "n:i:f:")) != -1)
        {
                switch(c)
                {
                        case 'n':
                                flag_n = 1;
                                nvalue = optarg;
                                n_packets = atoi(nvalue);
                                break;
                        case 'i':
                                flag_i = 1;
                                ivalue = optarg;
                                break;
                        case 'f':
                                flag_f = 1;
                                fvalue = optarg;
                                break;
                        case '?':
                                fprintf(stderr,"usage: ./ip.out -i <interface> -n <number_of_packets>\n");
                                fprintf(stderr,"usage: ./ip.out -f <file_name>\n");
                                exit(1);
```

```
                        default:
                                fprintf (stderr,"Unknown option character `\\x%x'.\n",optopt);
                                fprintf(stderr,"usage: ./ip.out -i <interface> -n <number_of_packets>\n");
                                fprintf(stderr,"usage: ./ip.out -f <file_name>\n");
                                exit(1);
                }
}

// checking for a correct combination of switches
if( !( (flag_f && !flag_i && !flag_n  ) || (!flag_f && flag_i && flag_n ) ) )
{
                fprintf(stderr,"misuse of the program switches\n");
                fprintf(stderr,"usage: ./sniffer.out -i <interface> -n <number_of_packets>\n");
                fprintf(stderr,"usage: ./sniffer.out -f <file_name>\n");
                exit(1);
}else {
                if( !flag_f ) print_app_banner(ivalue,n_packets);
                else print_app_banner(fvalue,0);
}



// initialize linked list
s = (struct ip_node_t *)createList();
r = (struct Modbus_node_t *)m_createList();



// opening sniff_data to hold the sniffer's output
sniff_data = fopen("sniff_data.txt","w");
ip_tree_rules = fopen("ip_tree.rules","w");
Modbus_tree_rules = fopen("Modbus_tree.rules","w");



// stablishing handler for sniffing:
if(!flag_f)
{
                if( ( handler = pcap_open_live(ivalue,BUFSIZ,1,10000,errbuf)) == NULL )
                {
                                printf("\n%s %s: %s\n","Couldn't open device",ivalue,errbuf);
                                fprintf(stderr,"\n%s\n","exiting.....");
                                exit(1);
                }

} else if( ( handler = pcap_open_offline(fvalue, errbuf) ) == NULL )
                {
                                printf("\n%s %s: %s\n","Couldn't open device",ivalue,errbuf);
                                fprintf(stderr,"\n%s\n","exiting.....");
                                exit(1);
                }



// printing time stamp of beginning
timer_init=time(NULL);
st_timer_start=localtime(&timer_init);
printf("\n -== %s %02d/%02d/%d at %02d:%02d:%02d ==-\n","Beginning test on",
                st_timer_start->tm_mday,st_timer_start->tm_mon,st_timer_start->tm_year,
                st_timer_start->tm_hour,st_timer_start->tm_min,st_timer_start->tm_sec
                );



// opening files for the linked trees
ip_tree = fopen("ip_tree.txt","w");
Modbus_tree = fopen("Modbus_tree.txt","w");
```

```
	// entering in the loop
	if((pcap_loop(handler,n_packets,our_callback,NULL))==-1)
	{
		fprintf(stderr,"\n%s\n","error ocurred while in loop, exiting now...");
		exit(1);
	}




	// exiting,printing statistics and closing files
	timer_end = time(NULL);
	st_timer_end = localtime(&timer_end);
	printf("\n%s %2.2lf %s\n"," -== Time test process :",time_diff = difftime(timer_end,timer_init),"segs. ==-");



	//creating and closing statistics file
	pcap_close(handler);
	statistics = fopen("statistics.txt","w");
	for(i=0,n_packets=0;i<5;i++) n_packets += packet_type[i];
	create_Statistics(st_timer_start,st_timer_end,time_diff,n_packets,flag_f);

	fclose(statistics);
	fclose(sniff_data);



	//reading from the tree and free-ing the allocated space
	readTree_ip(s);
	m_readTree_Modbus(r);

	fclose(ip_tree);
	fclose(Modbus_tree);

	freeTree_ip(s);
	m_freeTree_Modbus(r);

	printf("\n%s\n\n"," ----statistic control finished, please, wait----");


	fprintf(ip_tree_rules,"alert ip any any -> any any (msg:\"communication out of our ip-white-list\";)");
	fclose(ip_tree_rules);
	fprintf(Modbus_tree_rules,"alert ip any any -> any any (msg:\"communication out of our Modbus-white-list\";)");
	fclose(Modbus_tree_rules);



	// merging together the rules in order to get more compact rule files
	merge_ip("ip_tree.rules");



	// creating folder info_docs and moving files into it
	tail_banner();
	exit(0);

}
```

```
/*
 * our functions
 */

void
our_callback(u_char *args,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
	static int count = 1;		/* packet counter */
	u_short eth_type;		/* ethernet type in host byte order for switch use */
	int Modbus_flag = 0;		/* Modbus_flag = 1 when it detects a Modbus protocol packet */
	char mybuff[50];		/* inet_ functions use statically allocated memory */


	/*aux variables to help in the use of tree_linked_list */
	u_short sportaux,dportaux;
	u_short m_lenaux;
	u_char m_idenaux,m_funcaux;


	/* declare pointers to packet headers */
	const struct sniff_ethernet *ethernet;		/* The ethernet header [1] */
	const struct sniff_ip *ip;		/* The IP header */
	const struct sniff_tcp *tcp;
	const struct sniff_Modbus_tcp *Modbus_tcp;		/* The Modbus TCP header */


	/* sizes of ip frame and tcp segment */
	int size_header_ip;
	int size_header_tcp;
	char test_Modbus = 0;


	fprintf(sniff_data,"\nPacket number %d:\n", count);
	fprintf(sniff_data,"---------------------- \n", count);
	count++;



	/* define ethernet header torrent*/
	ethernet = (struct sniff_ethernet*)(packet);

	eth_type=ntohs(ethernet->ether_type);

	switch (eth_type) {

		case(0x0806):
			fprintf(sniff_data,"\tARP packet\n");
			packet_type[ARP]++;
			return;
		case(0x8035):
			fprintf(sniff_data,"\tRARP packet\n");
			packet_type[RARP]++;
			return;
		case(0x0800):
			fprintf(sniff_data,"\tIP packet\n");
			packet_type[IP]++;
			break;
		default:
			fprintf(sniff_data,"\tnot an ARP/RARP/IP packet\n");
			packet_type[UNKNOWN]++;
			break;
	}



	/* define/compute ip header offset */
	size_header_ip = IP_HL(ip = (struct sniff_ip*)(packet + SIZE_ETHERNET))*4;
	if (size_header_ip < 20)
```

```
{
        packet_type[INV_IP_HEADER]++;
        packet_type[IP]--;
        fprintf(sniff_data,"\t Invalid IP header length: %u bytes\n", size_header_ip);
        return;
}

/* print source and destination IP addresses */
strcpy(mybuff,inet_ntoa(ip->ip_src));
fprintf(sniff_data,"\tFrom: %s To: %s\n", mybuff, inet_ntoa(ip->ip_dst));



/* determine protocol */
switch(ip->ip_p)
{
        case IPPROTO_TCP:
                fprintf(sniff_data,"\t-TCP protocol\n");
                info_type[TCP_NO_Modbus]++;
                break;
        case IPPROTO_UDP:
                fprintf(sniff_data,"\t-UDP protocol\n");
                info_type[UDP]++;
                return;
        case IPPROTO_ICMP:
                fprintf(sniff_data,"\t-ICMP protocol\n");
                info_type[ICMP]++;
                return;
        default:
                fprintf(sniff_data,"\t-Not a TCP/UDP/ICMP protocol\n");
                info_type[UP_UNKNOWN]++;
                return;
}


/* define/compute tcp header offset */
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_header_ip);
size_header_tcp = TH_OFF(tcp)*4;
if (size_header_tcp < 20)
{
        fprintf(sniff_data,"\t-invalid TCP header\n");
        return;
}

fprintf(sniff_data,"\t Src port: %d to Dst port: %d\n", ntohs(tcp->th_sport),ntohs(tcp->th_dport));



/* define/print Modbus header fields */
/* is it an IP packet carrying Modbus data ? */

if ( ( ntohs(ip->ip_len) - size_header_ip ) == size_header_tcp ) test_Modbus = 0;
else test_Modbus = 1;

Modbus_tcp = (struct sniff_Modbus_tcp*)(packet + SIZE_ETHERNET + size_header_ip + size_header_tcp);

test_Modbus = test_Modbus && (Modbus_tcp->mtcp_prot == 0)&&(Modbus_tcp->mtcp_iden != 0)&&(Modbus_tcp->mtcp_len != 0 );
test_Modbus = test_Modbus && (Modbus_tcp->mtcp_iden < 256) && (Modbus_tcp->mtcp_func < 256 );


if ( test_Modbus )
{
                info_type[TCP_Modbus]++;
                fprintf(sniff_data,"      - Modbus: \n");
                fprintf(sniff_data,"\t\t%s = %d \n","remaining bytes in this frame",ntohs(Modbus_tcp->mtcp_len) );
                fprintf(sniff_data,"\t\t%s = %d \n","identifier",Modbus_tcp->mtcp_iden);
                fprintf(sniff_data,"\t\t%s = %d \n","function code",Modbus_tcp->mtcp_func);
                Modbus_flag = 1;

} else {
```

```
                                fprintf(sniff_data,"\t\t%s \n"," ( not a Modbus_tcp protocol / or misformed Modbus_tcp packet )");
                                Modbus_flag = 0;
        }



        /* sending info to the linked tree list  */

        if (!Modbus_flag)
        {
                if ((eth_type == 0x0800)&&(ip->ip_p == IPPROTO_TCP))
                {
                        sportaux = ntohs(tcp->th_sport);
                        dportaux = ntohs(tcp->th_dport);
                        s = (struct ip_node_t *)insertBranch(s,(struct in_addr *)&ip->ip_src,(struct in_addr *)&ip->ip_dst,
                                (u_short *)&sportaux,(u_short *)&dportaux);
                }

        } else
                {
                        sportaux = ntohs(tcp->th_sport);
                        dportaux = ntohs(tcp->th_dport);
                        m_lenaux = ntohs(Modbus_tcp->mtcp_len);
                        m_idenaux = Modbus_tcp->mtcp_iden;
                        m_funcaux = Modbus_tcp->mtcp_func;
                        r = (struct Modbus_node_t *)m_insertBranch(r,(struct in_addr *)&ip->ip_src,(struct in_addr *)&ip->ip_dst,
                                (u_short *)&sportaux,(u_short *)&dportaux,(u_short *)&m_lenaux,(u_char *)&m_idenaux,(u_char
*)&m_funcaux);

                }
        return;
}

void
print_app_banner(char *v,int n)
{
        char c = '0' ;
        int i ;
        char *snt_scrp[] =
        {
                "\n Making sure you have /var/log/Snort...\n",
                "./var_log_Snort.sh ",
                "\n Making sure you have the /opt/Snort/etc directory. \n",
                "./opt_Snort_etc.sh ",
                "\n Making sure directory /opt/Snort/rules exists. \n",
                "./opt_Snort_rules.sh ",
                " \n",
                "echo \"\n\"",
                NULL
        };
        char *ord[] =
        {
                "more ./conf_Snort_files/important_considerations",
                "more ./conf_Snort_files/input",
                "more ./conf_Snort_files/output",
                NULL
        };


        pantallazo(1);
        printf("Info banner: \n");
        for(i = 0 ; ord[i] ; i++)
        {
                system(ord[i]);
                pantallazo(20);
        }

        printf("Checking the system...\n");
        for( i = 0 ; snt_scrp[i] ; )
```

```
		{
			printf("%s",snt_scrp[i++]);
			system(snt_scrp[i++]);
			sleep(3);
		}

		pantallazo(4);
		if (n == 0)
		{
			printf("STARTING SNIFFING.....");
			sleep(1);
			printf(" %s\n"," -================-");
			printf(" -== %s ==- \n", APP_NAME);
			printf(" -== %s = %s ==-\n","interface",v);
			printf(" %s\n"," -================-");
			sleep(1);
		}

		else{
			printf("STARTING SNIFFING.....");
			sleep(1);
			printf(" %s\n"," -================-");
			printf(" -== %s ==- \n", APP_NAME);
			printf(" -== %s = %s ==-\n","interface",v);
			printf(" -== %s = %d ==-\n","number of packets to study ", n );
			printf(" %s\n"," -================-");
			sleep(1);
		}
		return;
}

void
create_Statistics (struct tm *begin,struct tm *end,double td, int pkt,int f) {

		enum eth_type et_aux;					/* aux through for-loops*/
		enum upper_eth_type uet_aux;		/* aux through for-loops*/
		char *et_uet;
		int sum_et=0,sum_uet=0;
		float assess=0;

		fprintf(statistics,"\n -== %s %d/%d/%d at %d:%d:%d ==-\n","Beginning test on",
			begin->tm_mday,begin->tm_mon,begin->tm_year,begin->tm_hour,
			begin->tm_min,begin->tm_sec);


		fprintf(statistics," -=== %s ===- \n","  STATISTICS OF OUR NETWORK TRAFFIC ");

		fprintf(statistics,"\n\t%s  \n\n","   -PROTOCOL SUPPORTED OVER ETHERNET- ");

		for(et_aux=ARP; et_aux<=INV_IP_HEADER; et_aux++)
		{
			switch(et_aux)
			{
				case(ARP): et_uet="ARP";break;
				case(RARP): et_uet="RARP";break;
				case(IP): et_uet="IP";break;
				case(UNKNOWN): et_uet="Unknown protocol";break;
				case(INV_IP_HEADER):et_uet="Invalid IP header";break;
				default: fprintf(statistics,"\t!!debug needed!!\n");
			};
			assess=(float)(packet_type[et_aux])/pkt;
			fprintf(statistics,"\t\t* %s = %d ---> %2.2f%c of our traffic\n",
				et_uet,packet_type[et_aux],assess*100,'%');
		}

		fprintf(statistics,"\n\n");
		fprintf(statistics,"\t%s  \n\n","   -PROTOCOL SUPPORTED OVER IP- ");


		for(uet_aux=TCP_NO_Modbus ;uet_aux<=UP_UNKNOWN ;uet_aux++)
```

```
 {
		switch(uet_aux)
		{
			case(TCP_NO_Modbus): et_uet="TCP";break;
			case(UDP): et_uet="UDP";break;
			case(ICMP): et_uet="ICMP";break;
			case(TCP_Modbus): et_uet="\t* of which TCP_Modbus";break;
			case(UP_UNKNOWN): et_uet="Unknown protocol over IP";break;
			default: fprintf(statistics,"\t!!debug needed!!\n");
		}
		assess=(float)(info_type[uet_aux])/pkt;
		fprintf(statistics,"\t\t* %s = %d ---> %2.2f%c of our traffic\n",
			et_uet,info_type[uet_aux],assess*100,'%');
}

/* when data dumped from a file, there's no sense in time statistics, the file takes 0 secs for the program
to be examined, resulting in divisions by 0 in our next piece of code*/

if ( f )
{
		fprintf(statistics,"\n\n\t...when studying from a pcap file there's no TIME STATISTICS\n\n");
		return;
}

fprintf(statistics,"\n%s %f %s\n\n"," -== TIME STATISTICS : the tests has taken ",td," segs.  ==-");


fprintf(statistics,"\t%s \n","   -PROTOCOL SUPPORTED OVER ETHERNET- ");

for(et_aux=ARP; et_aux<=INV_IP_HEADER; et_aux++)
{
		switch(et_aux)
		{
			case(ARP): et_uet="ARP";break;
			case(RARP): et_uet="RARP";break;
			case(IP): et_uet="IP";break;
			case(UNKNOWN): et_uet="Unknown protocol";break;
			case(INV_IP_HEADER):et_uet="Invalid IP header";break;
			default: fprintf(statistics,"\t!!debug needed!!\n");
		}
		fprintf(statistics,"\t\t* %s has %2.3lf packets/sec.\n",et_uet,(double)(packet_type[et_aux])/(int)td);
}

fprintf(statistics,"\n\n");
fprintf(statistics,"\t%s  \n","   -PROTOCOL SUPPORTED OVER IP- ");


for(uet_aux=TCP_NO_Modbus ;uet_aux<=UP_UNKNOWN ;uet_aux++)
{
		switch(uet_aux)
		{
			case(TCP_NO_Modbus): et_uet="TCP";break;
			case(UDP): et_uet="UDP";break;
			case(ICMP): et_uet="ICMP";break;
			case(TCP_Modbus): et_uet="\t* of which TCP_Modbus";break;
			case(UP_UNKNOWN): et_uet="Unknown protocol over IP";break;
			default: fprintf(statistics,"\t!!debug needed!!\n");
		}
		fprintf(statistics,"\t\t* %s has %2.3lf packets/sec.\n",et_uet,(double)(info_type[uet_aux])/(int)td);
}

fprintf(statistics,"\n -== %s %02d/%02d/%d at %02d:%02d:%02d ==-\n\n -== %s %02d/%02d/%d at %02d:%02d:%02d ==-\n",
		"Test start:",begin->tm_mday,begin->tm_mon,begin->tm_year, begin->tm_hour,begin->tm_min,
		begin->tm_sec,"Test finish:",end->tm_mday,end->tm_mon,end->tm_year,end->tm_hour,end->tm_min,
		end->tm_sec);

fprintf(statistics,"\t -=== %s ===- \n"," ------ END OF OUR STATISTICS FILE ------- ");


return;
```

```
}

void
tail_banner(void)
{
	char c = '0';
	int i ;
	char *sentences[] =
	{
		"Files created: \n",
		"\t - sniff_data.txt :\t holds the sniffer's output\n",
		"\t - statistics.txt :\t holds statistical information\n",
		"\t - ip_tree.txt    :\t holds all the combinations of ip @ and ports observed into our network traffic\n",
		"\t - Modbus_tree.txt    :\t holds all the combinations of ip @, ports and Modbus fiels observed into our network traffic\n",
		"\t - ip_tree.rules  :\t holds the ip rules to whitesniff our network, placed in ' /opt/Snort/rules ' \n",
		"\t - Modbus_tree.rules  :\t holds the Modbus rules to whitesniff our network, placed in ' /opt/Snort/rules ' \n",
		NULL
	};
	char *orders[] =
	{
		"mkdir info_docs","\nCreating folder info_docs...\n",
		"mv ./ip_tree.txt ./info_docs/.","\t...Moving ip_tree.txt  into folder info_docs\n",
		"mv ./Modbus_tree.txt ./info_docs/.","\t...Moving Modbus_tree.txt  into folder info_docs\n",
		"mv ./statistics.txt ./info_docs/.","\t...Moving statistics.txt into folder info_docs\n",
		"mv ./sniff_data.txt ./info_docs/.","\t...Moving sniff_data.txt into folder info_docs\n",
		"mv ./ip_tree.rules ./info_docs/.","\t...Moving ip_tree.rules into info_docs\n",
		"mv ./Modbus_tree.rules ./info_docs/.","\t...Moving Modbus_tree.rules into info_docs\n\n",
		"./moving_conf_files.sh","... moving 'classification.config' and 'reference.config' into /opt/Snort/etc\n",
		NULL
	};
	char *more_sentences[] =
	{
		"\n\n\nPay attention: in this same directory, there's a folder named 'rules',\n",
		"you should load this directory with the black-listing rules you want for Snort. \n",
		"This program will include automatically the names into Snort.conf, releasing you from this task.\n",
		"Include now, if you need, some .rules files or PRESS LETTER c(lower case) + ENTER (case sensitive) to continue: ",
		NULL
	};
	char *last_sentences[] =
	{
		"\n\nNow you can go to info_docs folder contained in this same directory, and consult the data gathered.\n",
		"...The appropiate rule file has been created and located in /opt/Snort/rules as well.\n\n\n\n",
		NULL
	};


	pantallazo(5);
	system("rm -r ./info_docs 2> /dev/null");
	for (i = 0 ; sentences[i] ; i++)
	{
		printf("%s",sentences[i]);
		sleep(1);
	}
	for(i = 0; orders[i] ;  )
	{
		if(system(orders[i++]) != -1 ) printf("%s",orders[i++]);
		sleep(2);
	}

	for ( i = 0; more_sentences[i] ; i++ )
	{
		printf("%s",more_sentences[i]);
		sleep(2);
	}

	fflush(stdin);
	for ( ; c != 'c' ; )
	{
		scanf("%c",&c);
		if( c != 'c' ) printf("\nPRESS LETTER C (case sensitive) + ENTER to continue: ");
```

```
                fflush(stdin);
        }

        if(system("./moving_rule_files.sh") != -1)
        {
                        printf("\n\t...Copying our whilelist compliant rules into /opt/Snort/rules\n");
                        sleep(1);
                        printf("\t...Moving black-list compliant rules from our folder 'rules'into /opt/Snort/rules\n");
                        printf("\t...Moving Snort.conf configuration file with our 'includes' into /opt/Snort/etc \n");
        }
        sleep(2);

        for ( i = 0; last_sentences[i] ; i++ )
        {
                printf("%s",last_sentences[i]);
                sleep(2);
        }

        return;
}


void
pantallazo (int j)
{
        sleep (j);
        system("clear");
        return;
}

//ip_node.h

#ifndef IP_NODE_H_
#define IP_NODE_H_



struct ip_node_t
{
        void *ptrdata;
        struct ip_node_t *ptr_r;
        struct ip_node_t *ptr_b;
};

enum field_t
{
        ip_ip_from = 0,
        ip_ip_to = 1,
        ip_src_port = 2,
        ip_dst_port = 3
};


/* prototypes */

struct ip_node_t *createList (void);
struct ip_node_t *createNode(void *);
struct ip_node_t *insertBranch (struct ip_node_t *s,void *,void *,void *,void *);
void freeTree_ip(struct ip_node_t *);
void readTree_ip(struct ip_node_t *);

#endif

//ip_func.c

#include "ip.h"
#include "ip_node.h"

extern FILE *ip_tree;
extern FILE *ip_tree_rules;
```

```
struct ip_node_t
*createList (void)
{
	return NULL;
};

struct ip_node_t
*createNode(void *data)
{
	struct ip_node_t *s;
	s = malloc(sizeof(struct ip_node_t));

	if (s != NULL)
	{
		s->ptrdata = data;
		s->ptr_r = NULL;
		s->ptr_b = NULL;
	}
	return s;
}

struct ip_node_t
*insertBranch (struct ip_node_t *s,void *ob1,void *ob2,void *ob3,void *ob4)
{
	struct ip_node_t *aux,*aux1,*aux2,*aux3,*prev,*prev1,*prev2,*prev3;
	struct ip_node_t *aux_loop;
	void *ob;
	enum field_t level = ip_ip_from;
	int prev_int;



	for ( aux = s, prev = NULL ; aux != NULL ; prev = aux, aux = aux->ptr_r)
	{
		if(!memcmp(aux->ptrdata,ob1,sizeof(struct in_addr)))
		{
			level++;
			for( aux1 = aux->ptr_b, prev1 = NULL ; aux1 != NULL ; prev1 = aux1, aux1 = aux1->ptr_r)
			{
				if(!memcmp(aux1->ptrdata,ob2,sizeof(struct in_addr)))
				{

					level++;
					for( aux2 = aux1->ptr_b , prev2 = NULL ; aux2 != NULL ; prev2 = aux2, aux2 = aux2->ptr_r)
					{
						if(!memcmp(aux2->ptrdata,ob3,sizeof(u_short)))
						{
							level++;
							for( aux3 = aux2->ptr_b , prev3 = NULL ; aux3 != NULL ; prev3 =aux3,
aux3 = aux3->ptr_r)
							{
								if(!memcmp(aux3->ptrdata,ob4,sizeof(u_short)))
								{
									return s;
								}

							}
						}
					}
				}
			}

		}

	}


	if (prev == NULL) prev_int = 0; else prev_int = 1;
```

```
switch (prev_int) {

        case(0):

                ob = malloc(sizeof(struct in_addr));                    //ip source
                s = createNode(memcpy(ob,ob1,sizeof(struct in_addr)));

                ob = malloc(sizeof(struct in_addr));                    //ip destination
                s->ptr_b = createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                aux_loop = s->ptr_b;

                ob = malloc(sizeof(u_short));                           //src port
                aux_loop->ptr_b = createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                           //dst port
                aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));

                return s;

        default:

                switch(level) {

                        case (ip_ip_from):

                                ob = malloc(sizeof(struct in_addr));            //ip source
                                prev->ptr_r = createNode(memcpy(ob,ob1,sizeof(struct in_addr)));
                                aux_loop = prev->ptr_r;

                                ob = malloc(sizeof(struct in_addr));            //ip destination
                                aux_loop->ptr_b = createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //src port
                                aux_loop->ptr_b = createNode(memcpy(ob,ob3,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //dst port
                                aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));
                                return s;


                        case (ip_ip_to):

                                ob = malloc(sizeof(struct in_addr));            //ip destination
                                prev1->ptr_r = createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                                aux_loop = prev1->ptr_r;

                                ob = malloc(sizeof(u_short));                           //src port
                                aux_loop->ptr_b = createNode(memcpy(ob,ob3,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //dst port
                                aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));
                                return s;

                        case (ip_src_port):

                                ob = malloc(sizeof(u_short));                           //src port
                                prev2->ptr_r = createNode(memcpy(ob,ob3,sizeof(u_short)));
                                aux_loop = prev2->ptr_r;

                                ob = malloc(sizeof(u_short));                           //dst port
                                aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));
                                return s;

                        case (ip_dst_port):

                                ob = malloc(sizeof(u_short));                           //dst port
```

```
                                        prev3->ptr_r = createNode(memcpy(ob,ob4,sizeof(u_short)));
                                        return s;

                                default:

                                        printf("DEBUG NEEDED \n");
                                        exit(1);

                        }
                }
}


void
freeTree_ip(struct ip_node_t *s)
{
        struct ip_node_t *aux_i,*aux_j,*aux_k,*aux_r;

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                {
                                        free(aux_r->ptrdata);
                                        free(aux_r);
                                }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                        {
                                free(aux_k->ptrdata);
                                free(aux_k);
                        }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                {
                        free(aux_j->ptrdata);
                        free(aux_j);
                }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
        {
                free(aux_i->ptrdata);
                free(aux_i);
        }

        return;
}


void
readTree_ip(struct ip_node_t *s)
{
        struct ip_node_t *aux_i,*aux_j,*aux_k,*aux_r;
        int aux1,aux2;
        struct in_addr *in_addr_aux1, *in_addr_aux2;
        int count=0;
        char mybuff[50];                /* inet_ functions use statically allocated memory */

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
        {
                in_addr_aux1 = (struct in_addr *)aux_i->ptrdata;
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                {
                        in_addr_aux2 = (struct in_addr *)aux_j->ptrdata;
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                        {
                                aux1 = *(u_short *)aux_k->ptrdata;
```

```
                                        for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                        {
                                                aux2 = *(u_short *)aux_r->ptrdata;
                                                fprintf(ip_tree,"%5d: ",count++);
                                                fprintf(ip_tree,"Ip source: %15s, ",inet_ntoa(*in_addr_aux1));
                                                fprintf(ip_tree,"Ip destination: %15s, ",inet_ntoa(*in_addr_aux2));
                                                fprintf(ip_tree,"Port Source: %5d, ",aux1);
                                                fprintf(ip_tree,"Port destination: %5d \n",aux2);
                                                strcpy(mybuff,inet_ntoa(*in_addr_aux2));
                                                fprintf(ip_tree_rules,"pass ip %s %d <> %s %d \n",
                                                        inet_ntoa(*in_addr_aux1),aux1,mybuff,aux2);

                                        }
                                }
                        }
                }
                return;
}


//Modbus_node.h

#ifndef Modbus_NODE_H_
#define Modbus_NODE_H_



struct Modbus_node_t
{
                void *ptrdata;
                struct Modbus_node_t *ptr_r;
                struct Modbus_node_t *ptr_b;
};

enum m_field_t
{
                Modbus_ip_from = 0,
                Modbus_ip_to = 1,
                Modbus_src_port = 2,
                Modbus_dst_port = 3,
                Modbus_len = 4,
                Modbus_iden = 5,
                Modbus_func = 6
};


/* prototypes */

struct Modbus_node_t *m_createList (void);
struct Modbus_node_t *m_createNode(void *);
struct Modbus_node_t *m_insertBranch (struct Modbus_node_t *,void *,void *,void *,void *,void *,void *,void *);
void m_freeTree_Modbus(struct Modbus_node_t *);
void m_readTree_Modbus(struct Modbus_node_t *);

#endif

//Modbus_func.c

#include "ip.h"
#include "Modbus_node.h"

extern FILE *Modbus_tree;
extern FILE *Modbus_tree_rules;

struct Modbus_node_t
*m_createList (void)
{
                return NULL;
};
```

```
struct Modbus_node_t
*m_createNode(void *data)
{
	struct Modbus_node_t *s;
	s = malloc(sizeof(struct Modbus_node_t));

	if (s != NULL)
	{
		s->ptrdata = data;
		s->ptr_r = NULL;
		s->ptr_b = NULL;
	}
	return s;
}

struct Modbus_node_t
*m_insertBranch (struct Modbus_node_t *s,void *ob1,void *ob2,void *ob3,void *ob4,void *ob5,void *ob6,void *ob7)
{
	struct Modbus_node_t *aux,*aux1,*aux2,*aux3,*aux4,*aux5,*aux6,*prev,*prev1,*prev2,*prev3,*prev4,*prev5,*prev6;
	struct Modbus_node_t *aux_loop;
	void *ob;
	enum m_field_t level = Modbus_ip_from;
	int m_prev_int;



	for ( aux = s, prev = NULL ; aux != NULL ; prev = aux, aux = aux->ptr_r)
	{
	if(!memcmp(aux->ptrdata,ob1,sizeof(struct in_addr)))
		{
		level++;
		for( aux1 = aux->ptr_b, prev1 = NULL ; aux1 != NULL ; prev1 = aux1, aux1 = aux1->ptr_r)
		{
		if(!memcmp(aux1->ptrdata,ob2,sizeof(struct in_addr)))
			{
			level++;
			for( aux2 = aux1->ptr_b , prev2 = NULL ; aux2 != NULL ; prev2 = aux2, aux2 = aux2->ptr_r)
			{
			if(!memcmp(aux2->ptrdata,ob3,sizeof(u_short)))
				{
				level++;
				for( aux3 = aux2->ptr_b , prev3 = NULL ; aux3 != NULL ; prev3 =aux3, aux3 = aux3->ptr_r)
				{
				if(!memcmp(aux3->ptrdata,ob4,sizeof(u_short)))
					{
					level++;
					for( aux4 = aux3->ptr_b , prev4 = NULL ; aux4 != NULL ; prev4 =aux4, aux4 = aux4->ptr_r)
					{
					if(!memcmp(aux4->ptrdata,ob5,sizeof(u_short)))
						{
						level++;
						for( aux5 = aux4->ptr_b , prev5 = NULL ; aux5 != NULL ; prev5 =aux5, aux5 = aux5-
>ptr_r)
						{
						if(!memcmp(aux5->ptrdata,ob6,sizeof(u_char)))
							{
							level++;
							for( aux6 = aux5->ptr_b , prev6 = NULL ; aux6 != NULL ; prev6 =aux6,
aux6 = aux6->ptr_r)
							{
							if(!memcmp(aux6->ptrdata,ob7,sizeof(u_char)))
								{
									return s;
								}
							}
							}
						}
						}
					}
				}
			}
		}
	}
```

```
                                        }
                                        }
                                }
                                }
                        }
                        }

}


if (prev == NULL) m_prev_int = 0; else m_prev_int = 1;

switch (m_prev_int) {

        case(0):

                ob = malloc(sizeof(struct in_addr));                    //ip source
                s = m_createNode(memcpy(ob,ob1,sizeof(struct in_addr)));

                ob = malloc(sizeof(struct in_addr));                    //ip destination
                s->ptr_b = m_createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                aux_loop = s->ptr_b;

                ob = malloc(sizeof(u_short));                           //src port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                           //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                           //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                            //iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                            //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));

                return s;

        default:

                switch(level) {

                        case (Modbus_ip_from):

                                ob = malloc(sizeof(struct in_addr));            //ip source
                                prev->ptr_r = m_createNode(memcpy(ob,ob1,sizeof(struct in_addr)));
                                aux_loop = prev->ptr_r;

                                ob = malloc(sizeof(struct in_addr));            //ip destination
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //src port
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //dst port
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //len Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_b;
```

```
                ob = malloc(sizeof(u_char));                    //iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));

                return s;


        case (Modbus_ip_to):

                ob = malloc(sizeof(struct in_addr));            //ip destination
                prev1->ptr_r = m_createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                aux_loop = prev1->ptr_r;

                ob = malloc(sizeof(u_short));                   //src port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    // iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                return s;

        case (Modbus_src_port):

                ob = malloc(sizeof(u_short));                   //src port
                prev2->ptr_r = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_r;

                ob = malloc(sizeof(u_short));                   //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    // iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                return s;

        case (Modbus_dst_port):

                ob = malloc(sizeof(u_short));                   //dst port
                prev3->ptr_r = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_r;

                ob = malloc(sizeof(u_short));                   //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    // iden Modbus
```

```
                                        aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                                        aux_loop = aux_loop->ptr_b;

                                        ob = malloc(sizeof(u_char));                    //func Modbus
                                        aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                                        return s;

                                case(Modbus_len):

                                        ob = malloc(sizeof(u_short));                   //len Modbus
                                        prev4->ptr_r = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                                        aux_loop = prev4->ptr_r;

                                        ob = malloc(sizeof(u_char));                    // iden Modbus
                                        aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                                        aux_loop = aux_loop->ptr_b;

                                        ob = malloc(sizeof(u_char));                    //func Modbus
                                        aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));

                                        return s;

                                case(Modbus_iden):

                                        ob = malloc(sizeof(u_char));                    // iden Modbus
                                        prev5->ptr_r = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                                        aux_loop = prev5->ptr_r;

                                        ob = malloc(sizeof(u_char));                    //func Modbus
                                        aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                                        return s;

                                case(Modbus_func):

                                        ob = malloc(sizeof(u_char));                    //func Modbus
                                        prev6->ptr_r = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                                        return s;

                                default:

                                        printf("DEBUG NEEDED \n");
                                        exit(1);

                        }
        }
}


void
m_freeTree_Modbus(struct Modbus_node_t *s)
{
        struct Modbus_node_t *aux_i,*aux_j,*aux_k,*aux_r,*aux_s,*aux_t,*aux_u;

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                        for (aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
                                                for (aux_t = aux_s->ptr_b ; aux_t != NULL ; aux_t = aux_t->ptr_r)
                                                        for (aux_u = aux_t->ptr_b ; aux_u != NULL ; aux_u = aux_u->ptr_r)
                                                                {
                                                                free(aux_u->ptrdata);
                                                                free(aux_u);
                                                                }


        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
```

```
                                        for (aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
                                                for (aux_t = aux_s->ptr_b ; aux_t != NULL ; aux_t = aux_t->ptr_r)
                                                {
                                                        free(aux_t->ptrdata);
                                                        free(aux_t);
                                                }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                        for (aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
                                        {
                                                free(aux_s->ptrdata);
                                                free(aux_s);
                                        }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                {
                                        free(aux_r->ptrdata);
                                        free(aux_r);
                                }


        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                        {
                                free(aux_k->ptrdata);
                                free(aux_k);
                        }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                {
                        free(aux_j->ptrdata);
                        free(aux_j);
                }


        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
        {
                free(aux_i->ptrdata);
                free(aux_i);
        }



return;
}


void
m_readTree_Modbus(struct Modbus_node_t *s)
{
        struct Modbus_node_t *aux_i,*aux_j,*aux_k,*aux_r,*aux_s,*aux_t,*aux_u;
        int aux1,aux2,aux3;
        struct in_addr *in_addr_aux1, *in_addr_aux2;
        char aux_char1,aux_char2;
        int count=0;
        char mybuff[50];                /* inet_ functions use statically allocated memory */

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
        {
        in_addr_aux1 = (struct in_addr *)aux_i->ptrdata;
        for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
        {
```

```
                        in_addr_aux2 = (struct in_addr *)aux_j->ptrdata;
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                        {
                                aux1 = *(u_short *)aux_k->ptrdata;
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                {
                                        aux2 = *(u_short *)aux_r->ptrdata;
                                        for(aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
                                        {
                                                aux3 = *(u_short *)aux_s->ptrdata;
                                                for(aux_t = aux_s->ptr_b ; aux_t != NULL ; aux_t = aux_t->ptr_r)
                                                {
                                                        aux_char1 = *(u_char *)aux_t->ptrdata;
                                                        for(aux_u = aux_t->ptr_b ; aux_u != NULL ; aux_u = aux_u->ptr_r)
                                                        {
                                                        aux_char2 = *(u_char *)aux_u->ptrdata;

                                                        fprintf(Modbus_tree,"%5d: ",count++);
                                                        fprintf(Modbus_tree,"Ip src: %14s, ",inet_ntoa(*in_addr_aux1));
                                                        fprintf(Modbus_tree,"Ip dst: %14s, ",inet_ntoa(*in_addr_aux2));
                                                        fprintf(Modbus_tree,"Port src: %5d, ",aux1);
                                                        fprintf(Modbus_tree,"Port dst: %5d, ",aux2);
                                                        fprintf(Modbus_tree,"lenght_data: %5d, ",aux3);
                                                        fprintf(Modbus_tree,"ident: %5d, ",(u_char)aux_char1);
                                                        fprintf(Modbus_tree,"funct code: %5d \n",(u_char)aux_char2);
                                                        strcpy(mybuff,inet_ntoa(*in_addr_aux2));
                                                        fprintf(Modbus_tree_rules,"pass ip %s %d <> %s %d (Modbus_func: %d
;Modbus_unit: %d;)\n",
                                                                inet_ntoa(*in_addr_aux1),aux1,mybuff,aux2,(u_char)aux_char2,
(u_char)aux_char1);
                                                        }
                                                }

                                        }
                                }
                        }
                }
                }
                return;
}

// merge_ip.c

#include "ip.h"


void
merge_ip(char *name_f)
{
                FILE *origin_1;
                FILE *origin_2;
                FILE *copy_aux;
                char buff_1[200];
                char buff_2[100];
                char aux[100];
                char pass[10],ip[5],s_add[20],s_port[10],sym[5],d_add[20],d_port[10];
                fpos_t pos;
                char equal = 0;


                //opening files
                origin_1 = fopen(name_f,"r");
                origin_2 = fopen(name_f,"r");
                copy_aux = fopen("backup.txt","w");



                while (!feof(origin_1))
                {
                        fgets(buff_1,sizeof(buff_1),origin_1);
```

```
                        fgetpos(origin_1,&pos);                         //pos points to the next line
                        fscanf(origin_2,"%s %s %s %s %s %s %s",pass,ip,s_add,s_port,sym,d_add,d_port);
                        if(!strcmp(pass,"alert")) break;
                        fsetpos(origin_2,&pos);                         //origin_1 and origin_2 must point to the same position
                        sprintf(buff_2,"%s %s %s %s %s %s %s",pass,ip,d_add,d_port,sym,s_add,s_port);
                        while(strcmp(pass,"alert"))
                        {
                                fscanf(origin_1,"%s %s %s %s %s %s %s",pass,ip,s_add,s_port,sym,d_add,d_port);
                                if( !strcmp(pass,"alert") ) break;
                                sprintf(aux,"%s %s %s %s %s %s %s",pass,ip,s_add,s_port,sym,d_add,d_port);
                                if(!strcmp(buff_2,aux))
                                {
                                        equal = 1;
                                        break;
                                }
                        }
                        if ( !equal )
                        {
                                fprintf(copy_aux,"%s\n",buff_2);

                        }else equal = 0 ;
                        fsetpos(origin_1,&pos);
                }
                fprintf(copy_aux,"%s\n",buff_1);
                //closing files
                fclose(origin_1);
                fclose(origin_2);
                fclose(copy_aux);

                rename("backup.txt",name_f);


}

#! /bin/sh
# var_log_Snort.sh

if [ -d /var/log/Snort ]
then
        echo "\t\t... /var/log/Snort exists"
else
        echo "\t\t... /var/log/Snort doesn't exists..."
        mkdir /var/log/Snort
        echo "\t\t\t\t...creating /var/log/Snort"
fi

#! /bin/sh
# moving_conf_files.sh

if [ -d ./conf_Snort_files ]
        then
                cp ./conf_Snort_files/classification.config /opt/Snort/etc/.
                cp ./conf_Snort_files/reference.config /opt/Snort/etc/.
                cp  ./conf_Snort_files/Snort.conf ./rules/Snort.conf
        else
                echo " !! revise your sniffer folder, conf_Snort_files folder is missed !! "
                exit 1

fi

#! /bin/sh
# moving_rule_files.sh

if [ -d ./info_docs ]
        then
                cp ./info_docs/*.rules ./rules/.
                cp ./conf_Snort_files/Snort.conf ./rules/Snort.conf
                chown $SUDO_USER ./rules/Snort.conf
                cd rules
                echo "\n"
```

```
                for file in $(ls *.rules)
                        do
                                cp ./$file /opt/Snort/rules/$file
                                echo "\t...$file moved into /opt/Snort/rules and included into Snort.conf"
                                sleep 1
                                echo "include \$RULE_PATH/$file" >> ./Snort.conf
                        done
                cp ./Snort.conf /opt/Snort/etc/.
                cd ..
        else
                echo "debug needed"
                exit 1
fi

#! /bin/sh
#opt_Snort_rules.sh


if [ -d /opt/Snort/rules ]
then
echo "\t\t... /opt/Snort/rules exists"
else
echo "\t\t... /opt/Snort/rules doesn't exists ..."
mkdir /opt/Snort/rules
echo "\t\t\t...creating /opt/Snort/rules"
fi

#! /bin/sh
# opt_Snort_etc.sh

if [ -d /opt/Snort/etc ]
        then
                echo "\t\t... /opt/Snort/etc exists"
        else
                echo "\t\t... /opt/Snort/etc doesn't exists..."
                if [ -d /opt ]
                        then
                                cd /opt
                                if [ -d /opt/Snort ]
                                        then
                                                mkdir /opt/Snort/etc
                                        else
                                                mkdir /opt/Snort
                                                mkdir /opt/Snort/etc
                                fi
                        else
                                mkdir /opt
                                mkdir /opt/Snort
                        mkdir /opt/Snort/etc

                fi

        echo "\t\t\t...creating /opt/Snort/etc"
fi
```

ANEXO B: English report

Whitelisting Sniffer and Statistical Traffic Study for Snort (IDS)

Arturo Ruiz Mañas

Supervisors: Dr. Michael Schukat & Dr. Hugh Melvin
OSNA Cyber Security Research Group http://www.osna-solutions.com/

Abstract

Along this thesis, I am going to address the complex and important problem of network security in Industrial Control and SCADA (Supervisory Control And Data Acquisition) systems. For this, previous to this work, I have gone through a lot of information about hacking and penetration ICS (Internet Connection Sharing) techniques, and in the present text I will delve into a solution against black-hat hacking[1] in the said environments.

The different approaches taken around this issue, are never 100% effective working alone by themselves, but a combination of some of them can bring a good level of protection. This is why, employees in these fields, always use a combination of tools for their tasks, trying to cover as many gaps as possible.

During this work, I introduce our design for a combination of a couple of methods traditionally used in network security (whitelists and blacklists), and propose further steps into this research in order to add (anomaly detection)[2]. Ploughing and ploughing, the idea is to get a powerful tool against information system's threats.

Using a Network Intrusion Detection System known as Snort, a very famous open-source program to any system administrator, and adding my program to it, we are sure of offering good safety to networks.

Of course, not everything is done, and hackers nowadays are able to avert many and very good security systems, but our tool, is able to come up with a very good representation of what it happens in our network, in a sense that everything that is outside that behaviour, Snort using our results, will alert about it, and further measures could be taken about what it could be a likely attack.

1.Hacking can be divided into three different categories: black-hat hacking, white-hat hacking, and grey-hat hacking. Names are very representative of their meaning.

2.Although this last of anomaly detection is a bit controversial since what we are achieving with our design is a mix between whitelisting and anomaly detection, but all depends on the point of view we take.

Acknowledgments

I would like to take this opportunity to thank my supervisors Dr. Michael Schukat and Dr. Hugh Melvin. After this period in which I have been presented with many new tools, got to read many interesting books and received great direction I believe I got to understand what the term “network security” means. Your advice, suggestions, and continuous support have been of incredible help not only for the present moment but very surely for my near future career.

I would like also to thank Jonathan Hanley, who I have had good advice from as well, thank you Jonny. Stephanus Meiring and David Thornton, even though I have spent little time with you, you are in my thoughts too.

Gracias además a mi ponente, Dr. José Luis Salazar Riaño, tuve ya una muy buena experiencia en las asignaturas que él imparte de Comercio Electrónico y Seguridad y Criptografía.

A mi familia, por toda la ayuda recibida, especialmente en esos momentos en los que parecía quedarme estancado. Sin vuestro incondicional apoyo no habría llegado a este punto.

Dorotka, thanks for these last two years, good friend, and better partner, I have always found great support in you.

Y yayo, un recuerdo especial para ti. Gracias por todos los momentos vividos juntos, has sido y seras siempre una gran referencia. Te echo de menos.

Table of Contents

Table of figures

List of Tables

1. INTRODUCTION

1.1 Introduction to Network Security

Nowadays, we find ourselves absorbed in a world in which our communications and daily affairs pass through a large number of computer systems. Our emails, reach its destination not without previously go along the tangle of routers that conform Internet, at the moment you can “survive” without stepping in a supermarket, with just a computer, an Internet connection and after a little number of clicks you can order you weekly shopping; bank transactions, last books, clothes, social networks. All can be done, and is done, from your desk at work or in the comfort of your sofa at home. What about private companies? Personal details, industrial data or signal control, important documents... everything gets off from a computer, goes a long a cable in an intranet, and this intranet is very likely connected to the network of networks: Internet is here, and everybody should understand the important role that Network Security plays in all of this.

Whether like it or not, we are already in the digital era in which everything is translated into 1's and 0's. This is why it is so important to protect the information over this new platform. Thus, I believe a good starting point for this thesis would be a definition of “network security”. What does network security refers to?

According to wikipedia, network security consist on [1]:

*“**Network security** consists of the provisions and policies adopted by a network administrator to prevent and monitor unauthorized access, misuse, modification, or denial of a computer network and network-accessible resources.”*

Webopedia says [2]:

“*A specialised field in computer networking that involves securing a computer network infrastructure. Network security is typically handled by a network administrator or system administrator who implements the security policy, network software and hardware needed to protect a network and the resources accessed through the network from unauthorized access and also ensure that employees have adequate access to the network and resources to work.*”

One simpler definition would be the one in the digital magazine “Magazine Encyclopedia” [3]:

“*Protecting the computer systems in the network from unwanted intrusions.*”

In short, network security prevents from attacts and possible threats, protecting, computer systems that make possible the development of daily routines in Internet, from misuses. We could also

see it from the other side, and say as well that once we got hacked, network security controls and monitorizes these misuses against private networks in companies, personal businesses or your home.

1.2 Networks in Industrial Environment

Let's try to make it familiar to us. Since we have never talked during our University years about Industrial Networking Protocols, the best thing to start with this topic, will be speaking about Industrial Ethernet; after all, everybody working in Telematics has come into touch with Ethernet.

Industrial Ethernet refers to the use of Ethernet into industrial environments (connectors... switches...) for automation or process control. Components in these environments must work sometimes in extreme conditions of temperature, humidity or vibration; conditions, that on the other hand, exceed the ranges of usual information technology equipment [22].

This particular Ethernet reduces problems related to electrical noise and prevents from equipment damage. Although essentially both Ethernets share a common basis, there are some differences between them, for example: Industrial Ethernet uses deterministic delivery, whereas Ethernet uses collition detection. But I repeat that, essentially, they both share a common basis.

Some other examples of Industrial Protocols are: Modbus and its variants, ZigBee, EtherCAT, DeviceNet... it exists a large list.

To sum up, we should think of Industrial Protocols, as an adaptation of the Information Protocols to harsh environments. They help machines in situations very different to the ones we users have in our offices or homes, to develop their functionality.

2. LITERATURE REVIEW

2.1 Network Protocols

"A network protocol defines rules and conventions for communication between network devices. Protocols for computer networking all generally use packet switching techniques to send and receive messages in the form of *packets*." [26].

The list of Network Protocols can be huge, here we are going to make a little reference to some that we believe are important for this thesis:

2.1.1 IP

Internet Protocol (IP), about this protocol, we can find big amounts of information in Internet, and with reason, since is the fundamental pillar on which Internet is sustained. Without this protocol we could not have interconnections further than our local network, it is because of this that the IP protocol represents such a fundamental intruduction into networking technologies, and resulted in the net of networks we all know nowadays. It's been so important that it has given its name to a whole protocols stack: TCP/IP.

IP supports unique addressing for computers on a network, IP addresses (IPv4 and IPv6). There aren't two devices in this world, that could have the same IP address, we could compare it to the postal address of our homes. Data on the other hand, is organized into packets and all of them include both a header (with information about source and destination) and the payload with the info itself.

The protocol IP, works in the 3rd layer of the OSI model. It can therefor run on top of different link layers interfaces: Ethernet, Wifi, Frame Relay, ATM …

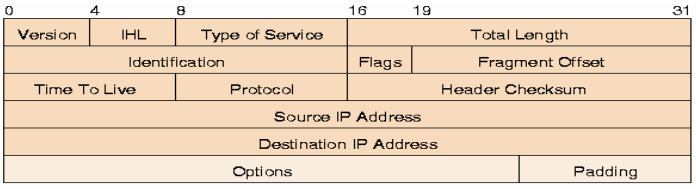


Fig.1 IP frame structure

We could speak much further about this protocol [SteRi01], and although we could fill many pages just talking about it, this is not the point of this thesis. Let's move on, and continue with another important protocol: TCP.

2.1.2 TCP

Transmission Control Protocol (TCP) is one of the main protocols in the TCP/IP stack. TCP functions are about reliability, packet ordering, error-checking delivery of a stream of bytes between programs, that established a session between, running on computers connected to a network.

This protocol refers to the transport layer of OSI model.

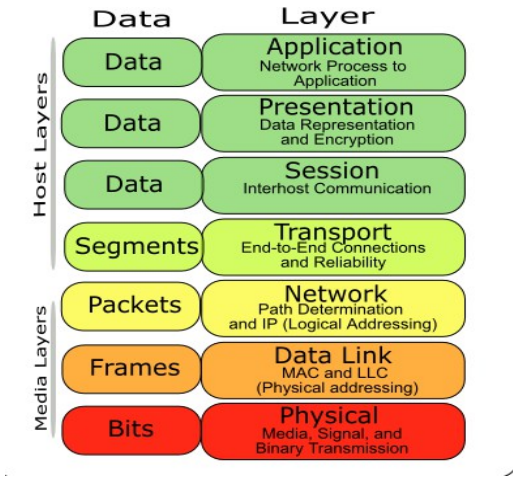


Fig. 2 OSI model

Some important concepts on TCP are:

- TCP segment structure (TCP header)
- session: data transfer.
- connection diagrams: connection establishment and termination.
- maximum segment size

Many things could be said within TCP, and there is a lot of bibliography written about this protocol [SteRi01] and about its “unreliable” protocol partner UDP [SteRi02]. It is important to understand very well these two protocols if you want to work on networking and topics related to it.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Fig.3 TCP header

2.1.3 Other IP Protocols

There are other protocols included into the IP-TCP protocols stack and not only the ones named before and there are many other protocols that are gradually adapting their technology to be included inside an IP frame or an TCP segment.

Just to see the IP-TCP protocols stack, we include the next figure. In it, we can differenciate between levels: application level would be the one on the top, later downward transport layer, network layer[1] and link layer.

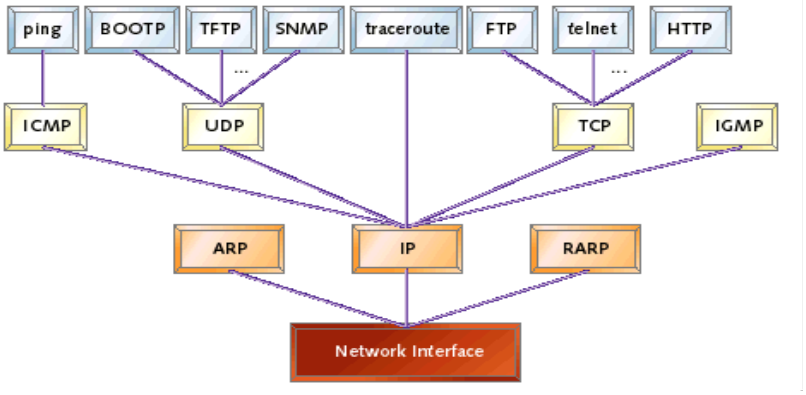


Fig.4 OSI model / DARPA and TCP/IP Protocol Suite

HTTP: HyperText Transfer Procotol used by the WWW (World Wide Web). Defines messages' format and how this messages are transmitted. It also defines how Web-servers and browsers should respond to commands.

1. The protocols ICMP and IGMP, although they are at the same high as UDP and TCP, they are not part of the transport level. They are encapsuled inside IP and that's why in the image they are drawn near the latter protocols.

FTP: File Transfer Protocol, built on a client-server protocol architecture, it is used to exchange files over the Internet.

SMTP: Simple Mail Transfer Protocol, used for sending e-mails between servers. E-mails can be retrieved with an e-mail client.

ARP: Address Resolution Protocol, a network layer protocol used to translate IP network addresses into link layer addresses (MAC addresses).

UDP: a connectionless protocol associated to the transport layer in the OSI model, running on top of IP. Provides very few error recovery services, leaving this tasks normally for protocols that run on top of him.

2.1.4 Modbus

As this is a completely new protocol for us in our Telecommunications Engineering education, we will delve a bit deeper into it than with the previous protocols, and since in this work, we are to deal with a solution for Industrial Control System Protocol networks (and more precisely with Modbus), a brief but good intruction to Modbus is required.

Modbus is a simple and robust serial communications protocol originally published by Modicon for its use with its programmable logic controllers(PLCs). It has become an important standard communications protocol, and a commonly available way to connect electronic devices [ezTCP].

It renders possible communication between many devices, approximately 240, connected to the same network, a common example would be a system that measures temperature and humidity and communicates the results to a computer. Modbus is usally used to connect a supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition (SCADA) systems [23][24] [25].

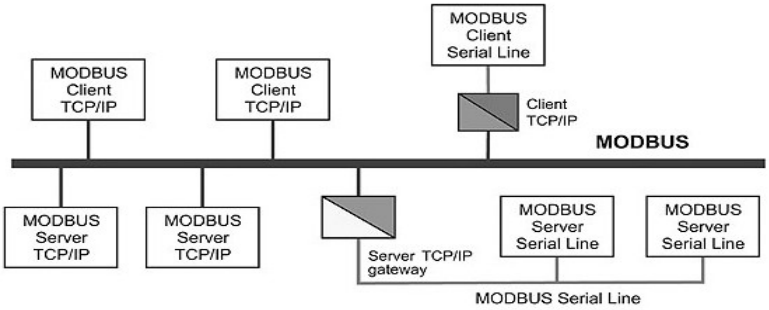


Fig.5 Setup example of a Modbus Network

There is software available in Internet in order to emulate set ups that work with Modbus devices, some devices are master some others slave. It's been developed as well an API Modbus that simplifies the process of creating more especific sofware and virtualization of a whole Modbus scenario. As a matter of fact, for this project, it was created a virtual network, using Virtualbox for this purpose.

In this network, it was configured three virtual machines: a virtual machine representing the Modbus master and another machine for the Modbus slave, finally another machine plays the role of NIDS (Network Intrusion Detector System) sensor. There is an annexe at the end of this text explaining the process of virtualization and the Modbus software employed in it.

To make things clearer, let's make a little reference to the terminology "master" and "slave". Nowadays we know this as the server-client model. A master-device works as a client, who sends requests for the slave (server) to process them and produce answers or replies. If you know how the model server-client works, this master-slave terminology shouldn't be a problem [SteRi02].

There are several types of Modbus: Modbus RTU, Modbus ASCII, and Modbus TCP. Our solution works on TCP-Modbus. Encapsulation of Modbus inside a segment TCP it's been a great advantage to our design, since it makes easier to work with it in a much wider environment, and our design can be comfortably extendable to other protocols that are included into the IP-TCP model as application layer SDU, resulting very straightforward to add functionalities following the same procedure seen in our program code.

Layer	ISO/OSI Function	Modbus Function
5,6,7	Application	Modbus Application Protocol
4	Transport	Transmission Control Protocol
3	Network	Internet Protocol
2	Data Link	IEEE 802.3
1	Physical	IEEE 802.3

Table 1 Stack's layer in Modbus TCP

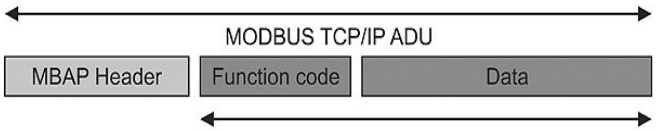


Fig.6 Modbus TCP/IP ADU

Transaction Identifier	Protocol Identifier	Length	Unit Identifier
2 bytes	2 bytes	2 bytes	1 byte

Fig. 7 Modbus-TCP header

2.1.5 ZigBee

Though it is not important for our thesis, learning a bit about ZigBee can help us to see clearer the idea of Industrial Network Protocols:

ZigBee, is a specification for a suite of high level communication protocols using small, low-power digital radios based on an IEEE 802 standard for personal area networks. Devices are often used in mesh network form to transmit data over longer distances, passing data through intermediate devices to reach more distant ones. This allows ZigBee networks to be formed ad-hoc, with no centralized control or high-power transmitter/receiver able to reach all of the devices. Any ZigBee device can be tasked with running the network.

The list of uses its quite extensive, but some examples to consider are:

- Home Entertainment and Control – Smart homes.
- Wireless sensor networks
- Industrial control
- Embedded sensing
- Medical data collection
- Smoke and intruder warning
- Building automation

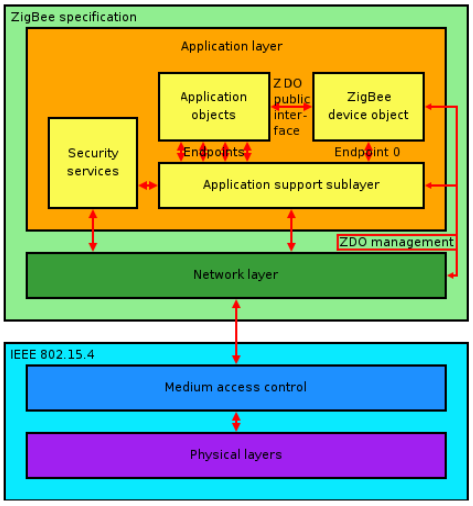


Fig.8 ZigBee specificiation's layers

ZigBee builds upon the physical layer and medium access control defined in IEEE standard 802.15.4 . As we can see in the previous figure, ZigBee stack architecture consists of Physical and Medium Access control layer, that gives support to the ZigBee network layer. For its part, the ZigBee specification (network) layer, consists of the APS sub-layer, the ZDO (containing the ZDO management plane), and the manufacturer-defined application objects.

2.2 Cybersecurity / Threats on ICS

To contextualise this work, let's talk a little about cybersecurity and threats on ICS (Internet Connection Sharing). In the world we're living in, attacks against IC (Critical Infrastructure) of energy, gas, oil and water are increasing, and it's not weird to know that these attacks are well-funded by organizations, competitors or even governments.

Just to put this into the right context, let's take some examples of attacks in the recent year. Although we shouldn't believe everything we read in the Internet, we can get a good smattering of what it is possible nowadays with just a computer and an Internet connection and how vulnerable are all the systems we base our daily routines (electricity, water, heating) on:

- HuffingtonPost → Posted: 05/16/2013 11:17 am EDT | Updated: 05/16/2013 11:23 am EDT [31]:

  “Syria faced an Internet blackout for eight hours on Wednesday, its second one in the past week and the sixth one of the two-year uprising against President Bashar al-Assad, a U.S. web *trafficking firm reported*. Phone lines into Damascus were also down.”

- BBC → Posted: 20 May 2013 Last updated at 10:13 GMT [32]:

  “State-sponsored hackers have renewed attacks on the US after a three-month hiatus, the New York Times reports.”

- Infosecurity-magazine.com → Posted: 12 April 2013 [33]:

  “The latest issue of the ICS-CERT Monitor has described two similar hacks that happened last year where attackers used a weak credentials vulnerability to gain access to buildings’ energy management system (EMS), Tridium Niagara.”

The list of reported attacks during 2013 would be large, and it's just been four months since the beginning of the year. So “security”, such an important thing, but a thing on the other hand, that we usually don't put too much effort in.

2.2.1 Security on SCADA Systems

SCADA stands for Supervisory Control And Data Acquisition. But what is a SCADA system and what is it used for? The term SCADA refers to a type of Industrial Control Systems (ICS). These ICS are computer controlled systems whose task is to monitor and control industrial processes. Examples of SCADA systems could be those systems that allow operators to change and enable alarm conditions related to temperature in systems designed to control the flow of cooling water in some industrial processes or those systems that monitor high and low levels in water tanks and alarm when a certain level is reached. There are plenty of examples and they develop important functions in many processes that influence in our welfare.

2.2.1.1 Attacks on SCADA Systems

In these days, worries about how to protect SCADA systems are increasing. With the ever-growing threat of “cyber terrorism” (specially after the 11S of New York) [35], specialists in information security issues are becaming more concerned about vulnerabilities in SCADA systems since as I explained before, these systems are responsible for controlling and monitoring our water distribution systems, oil and gas pipelines or electrical grid...

The design of such systems has evolved during the years, providing them with extra flexibility and functionalities, but turning them more vulnerable as well. SCADA systems have been present since earlies 1970, allowing us to monitor and remotely control devices distributed along wide extensions. The architecture of these systems, consists of a central computer system that communicate with other machines using one or more telecommunication technologies. During the last decade, Internet and other Internet-based techonologies have been included into the SCADA systems' design.

Attackers nowadays are determined to get control over SCADA and other ICS devices, for this they use different techniques. We must see these systems as real-time control system on which a successful attack could bring very serious and terrible consequences (in terms of human health or even life), and attacks against them doesn't seem to be very different from attacks to devices inside a common information network.

Good examples of attacks are: DoS (Denial of Service), passwords stealing, impersonalization, forgery of documents or deletion of them, not properly an attack but the prelude to one could be

scanning of ports of our machines... there is a wide range of threats to our systems and it is a good practice for any administrator to be aware about all of them.

Just to illustrate a little bit more this, threats such DoS (Denial of Services) against SCADA systems (or other informatic systems), consists on attacks in which attackers generate a high number of requests to our machines in such a level that they stop being able to give service and collapse [37].

Another example of attack to SCADA systems could be different types of malware that take advantage over vulnerabilities in the software of these devices. A known one is “a new type of malware that uses the .lnk vulnerability in Microsoft Windows and Siemens SCADA systems” [34].

There are many ways in which our system could be compromised, and a constant revision of our network devices is highly recommendable. On this respect, any tool that could automatise the process will be always a great help.

2.2.1.2 Solutions

What can we do to alleviate this? There are many tools and ideas out there to help security employees to deal with these threats. Of course, we should never lower the guard, and have always an eye on new threats. Today we have a safe system, but maybe tomorrow we are in troubles...

One idea, although not a solution in itself, is the use of honeypots [36]. A honeypot is a very smart way to get to know if there are people interested in your network and who they are. It consists of a “dummy” device that accurately expose the same characteristics as any of those devices controlling our network and that results in an easy target for attackers. Leaving there this device “unattended”, and monitoring attacks against it, can offer to system administrators with a very good source of information about those who intend to break into our system.

Any other form of security against for example DoS attacks in normal informatic systems, could do the job as well in SCADA systems. For example firewalls, such us Iptables or some other commercial ones, or other tools to control the connections flow would work just fine.

There are tools and solutions for many threats nowadays, the problem is how to use them in the most accurately manner as possible and for this, if you are in charge of these security aspects, you need to know your network.

Here is where it comes our idea. The solucion we propose to secure these systems; systems based on protocols as Modbus, is using Deep Packet Inspection [30]. Being able to come with a good

description of our network's behavings is a big advantage for any administrator, and it is on this aspect that our research in OSNA and in particular the research for this thesis is focused in.

2.3 IDS / IPS (Intrusion Detection System / Intrusion Prevention System)

The term Intrusion Detection System/Intrusion Prevention System (IDS/IPS) refers to a software application that is able to contrast the packets that flow in our network with some predefined patterns. Sometimes this patterns refer to allowed traffic, and some others to traffic that should be disallowed. Thus, we come across with two very important concepts which we will be speaking about along this text: blacklisting versus whitelisting.

Actually, to be more precise, there are three main traditional detection methods for network based attacks:

- blacklisting,

- whitelisting,

- and anomaly detection.

Let's explain them:

- Blacklisting

Blacklisting is a method whose approach to security is signature-based. It allows through all elements except those explicitly mentioned. But... this method, can only prevent from previously analyzed threads and is easy for attackers to dupe these protections. For example, by doing modifications in the packet's payloads; this modifications can range from fragmenting and spreading the payload of a single packet into different smaller packets, to the representation of the info contained inside the packet's payload in a manner that can completely avert the packet's inspection procedures of any Intrusion Detection System or an antivirus program.

Many other techniques can be used to avoid being detected... Furthermore, it may happen that every time a new threat is discovered, till a new signature is developed and distributed to all of our systems, it could pass weeks or even months before a complete updating against them, leaving us completely exposed to these new attacks.

- Whitelisting

On the other side of the coin, we have whitelisting, this would be the very opposite approach to the previous one. Instead of having the program checking for “bad-known” behaviours and alert about them, with a whitelisting approach, what we have is an approach to security that just allows “good-known” behaviours in the network traffic: all packets that don't match with what is listed into our whitelist, will be alerted about and taken apart for further study as a posible threat.

But... once again, this approach turns out not to be perfect, since we should be able to fine-tune as best as we can our detection system in order not to allow possible attacks, or possibly even allow actions that had previously been disallowed by a blacklist.

Additionally to this, hackers can study our whitelist, and construct packets that, though they are conformed according to what is considered to be “good”, they disguise what is the last virus or trojan, and our whitelisting measures would be useless.

- Anomaly detection

The last method, known as well as qualitative anomaly detection, is one of the last approaches to network security, and offer protection by analising one by one the packets, and rising an alert everytime the content of a packet is “too different” from “the norm”. For this approach to be possible, we should be able to seize network traffic's intrinsic characteristics. They would use N-gram analysis. N-grams are sequences of N-consecutive bytes extracted from the payload of the packets, for later compare these N-grams to the N-gram allowed models.

The problem we find in N-gram statistical study is that it does not allow to distinguish between good and bad traffic or creates too many false positives since N-gram does not take into account packet structure and different header fields. Instead, what the design here explained does is Deep Packet Inspection, looking into the characteristics of network fields, constituing with it an important difference between other approaches and ours. With my program, during an initial training phase, we could build statistics about N-grams present in normal network traffic and according to them we conform traffic models. The packets later sniffed from the network will be compared to these models to rise alerts in case of a certain deviation from this normal behaviour.

In this thesis, we will be presenting our desing and code for a new program. The program began being a simple sniffer to which we have added new features in order to implement a whitelisting

approach. Although we have focused specially in whitelisting, blacklisting features are included as well, in such a way that we could have the best of both worlds.

However, in OSNA, we believe that the combination of the three policies, and not only the first two, is what is desirable, and a possible modification to this thesis for future developments would pass through including a more thorough statistical study of traffic characteristics. This would complete what we are sure to be a very powerful tool for system administrators.

To discover unauthorized access to a computer network, IDS analyze traffic on the network for signs of malicious activity. They base their functionality in comparing well-established packet patterns to the ones it sees in the network. This packet patterns are known as “signatures”. Normaly, this “signatures” are part of this schema of blacklisting, although you can find as well some “pass-rules” that would exemplify the whitelisting approach. As we commented in our abstract, there exists a period of time then, in which from the moment of the creation and expansion of a new threat, till the moment in which we have its pattern registered and updated in all of our devices, we are completely unprotected from this malware. And this is why the combination of both policies offers a better proctection, on one side blacklisting protect from already known attackts, from other a well fine-tuned whitelist can avoid most recent attacks during those periods of unprotection.

2.3.1 Snort

Our program will transmit useful information to Snorts, an open source Network Intrusion Prevention and Detection System (IDS/IPS) developed by Sourcefire. It combines the benefits of signature, protocol, and anomaly-based inspection. It is used to detect non-authorised accesses to computers or networks. This non-authorized accesses vary from skilful attacks from nimble hackers to the well-known script kiddies used by teenagers to gain access to social network accounts of their partners in highschool.

To have a better grasp of what is intended here, we should do a little description of how Snort works. After it, the idea we are chasing and what we are seeking with it will be much clearer.

Snort can work in three different manners: as a packet sniffer itself, as a packet logger, or it can work as a Packet Intrusion Detection System tool. Its functionality is based in several modules that manage the information a single packet at a time, and after all the needed transformations carried out by these modules, the result is handed to an “alerting and logging component” which, if it's the case, it

will fire off an alert and log the packet.

Let's go step by step: to illustrate the path followed by a packet, once it arrives to the computer's network interface, I'm including the next figure.

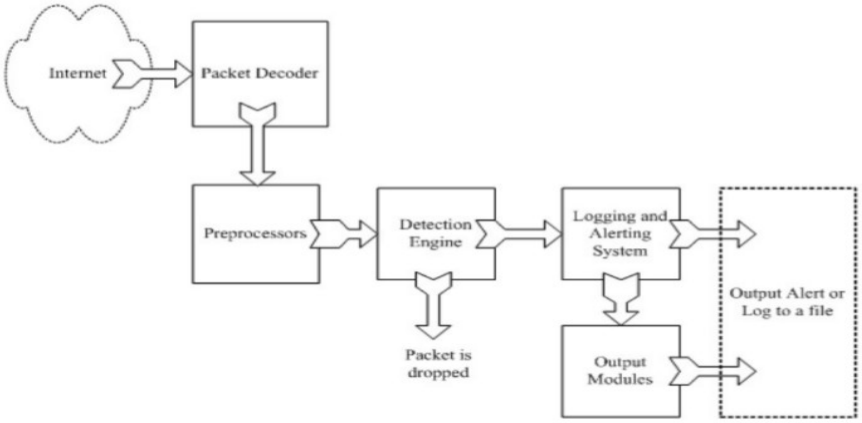


Fig.9 Snort's inner workings

As we see in this image, Snort uses a very well defined set of behaviors and for its purposes it uses several modules .

Let's describe the inner workings of Snort:

First, the packets arrive to the device's NIC and are decoded off the wire by the packet decoder, which will determine what protocol is in use for a given packet. Then, when using Snort as a NIDS, after the incoming packets are parsed by the packet decoders, data is sent through any preprocessor you may have enabled in your Snort.conf file. It continues through the detection engine which matches it against the rules in any ruleset enabled in your Snort.conf file. “Snort.conf”, being a configuration file, is a very important file to bear in mind then, as it is where we can configure Snort's behaviours.

Afterwards, matches are sent to the alerting and logging components, to be passed through whatever plug-ins you have selected, alerting and logging the data as it has been configured to do.

Some of this modules would need some further explanation as we do next:

- The packet decoders

  Note: here alerts can be generated based on malformed protocol headers:

  - overly long packets

  - unusual or incorrect TCP options

- and other such behavior.

- Preprocessors

Plug-ins to Snort that allow you to parse incoming data in different ways useful for the alerting modules. Without preprocessors you will only look at each individual packet as it comes in over the wire, missing some modern attacks:

- overwriting data in overlapping fragments.

- deliberated IDS evasion techniques like putting part of a milicious application request in one packet and the rest in another packet.

- and other similar practices.

After the data is returned from the preprocessors, it is passed to the detection engine.

- The detection engine

It's the component of Snort that takes data from the packet decoder and preprocessors (if any enabled) and compares it against the rules in your Snort.conf.

First, the detection engine will try to determine what rulesets it ought to be matching against for a given piece of data. It classifies this first by protocol: TCP, UDP, ICMP, or IP.

For TCP or UDP this is source and destination port number.

For ICMP: it's the ICMP type.

For plain old IP packets ...

- Rules and matching

IMPORTANT: in general, “alert” rules will fire before “pass” rules. However, if you would rather have this behavior reversed, you can specify the -o option to Snort on the command line, making the order “pass”, “alert”, “log” instead. Since we intend to provide a whilisting feature to this detection tool, we must be careful what's the order in which Snort execute its actions otherwise we could be alerting that is intended to be allower. So, this is a very importan note to have in mind.

Eventually, we can understand that the whole idea about Snort goes around files that describe network correct or wrong behavings. All Snort's modules described in this section carry out their functions having a common direction: work on the packets Snort picks from the net, and transform them in a way that alert and logging modules (next explained module) can perfectly alert and log about possible threats. If we could run a program that is able to come up with what is suppose to be acceptable, or with what is suppose to be allowable, we can automatize this powerful IDS tool and make system administrators' duties much easier.

- Alerting and Logging components

After the rules have been matched against the data, we have the alerting and logging components.

- The logging mechanism in Snort will archive the packets that triggered Snort rules.

- While the alerting mechanism is used to notify the analyst that a rule has fired.

– Pass rules will allow some behaviours, seen as acceptable-behaviours, if we have established correctly the order to be checked in the previous module.

2.4 Aim of Thesis / Motivation of the Project

As we were speaking before, both security approaches, whitelisting and blacklisting, have their pros and cons separatelly, and this is why we intend to build up a solution in order to merge them into a single program and get the best from both perspectives. Hence, we propose a solution that we believe will be a very powerful tool in any system administrator's resources.

During a study phase in which the user runs our program, the idea is to come up with all the possible sessions established among devices in a network, collecting its IP addresses and ports and collecting as well information about functions, identificators and lengths in Modbus packets. With this information, the application will generate file rules, being this file rules the basis for a whilelisting approach.

Normally, networks in private companies, can consist in hundreds of devices, exchanging in just little time thousands of packets among them. For any system administrator, trying to register all these exchanges would represent a daunting task. In OSNA, we want to make this task much simpler for that workers, and come up with a complete study of these connections. What in a beginning can seem a very complex network, our program is designed to summarize in a file, all those sessions stablished between machines.

For this, we have modified the “idea” of sniffer program, including on it some new features. A sniffer or packet analyzer is a computer program that can intercept and log traffic passing over a digital network. Ours, is able to create different files, some of them containing pass-rules, according to the network traffic it sees in the segment of the network it is connected to, for afterwards transmitting them to the Snort file system.

Snort will use its own features to alert on any behaviour out of its blacklist rules file that has been set in its configuration file, and on the other hand, it will use the “pass rules” created by our

sniffer, allowing what is consider to be a “good-known” behavior. Merging then both security methods: black and white -listing.

To understand the whole schema, we should get at least some smattering of Snort, and once then, we will see our design's idea a bit clearer. Since it is not our intention to speak too deeply about Snort in this thesis, an annexe talking about it has been included at the end of this text. But to get a general concept of it, we will continue into the next point with a little review on Snort.

3. DESIGN

3.1 Working Environment

During this project I have been working in a Linux environment, using a Debian Linux distribution in its Xubuntu version. The programming language used is C and shell programming. The shell program used is the Bash Unix shell of the GNU project and the compiler it's been gcc - GNU project C and C++ compiler-.

Steps in our work till the final result:

The development of our sniffer has gone through several steps. From a simple sniffer able to sniff the packets in promiscuous mode and dump the information contained in the packet's header, to the final product in which it is able to implement all the functionality we had in mind, leaving it open to further plug-ins.

Let's describe the different steps taken in our development process of our program.

3.2 PCAP Library

An important concept to understand, and that I came to understand when faced this project is “what is an API?”

API stands for Application Program Interface, is a protocol created to be used as an interface by software components in order to communicate among them. It's an important help for software developers allowing them to forget about particularities of the hardward they are working with.

The bottom line is that once you use an API, everything can be used according to the Linux philosophy of “all is a file in Linux”. Is a library that have routines, data structures, object classes and variables that you can use into your programs. One example of it would be the PCAP (Packet CAPture) library [4] [5] [6].

Pcap (packet capture) consists of an API used for capturing network traffic, is the basis of any sniffer nowadays or Intrusion Detection Systems such as Snort. Unix-like systems implement pcap, in the libpcap library. During the section dedicated to our design's code, it is seen the statement `#include <pcap/pcap.h>` which “pcap.h” is the header file that allows us to use the functions, constants, macros... available in libpcap, inside our program.

3.3 How our program works with Snort

The program has been designed to bring some help to Snort tasks. What firstable was a hard task for any system administrator, our program is thought to make things easier for our employees working in securiting our information systems.

The idea is to take advantage from the workings of Snort, which is able to contrast the packets that arrive to the NIC (Network Interface Card) with some predefined patterns. If we could be able to come up with a representation of our segment network's behaviours and express it into a file, after moving this file into Snort's file system, we could count with a very powerful tool.

Well then, the program designed during this period is able to create that file (among some other more files) with what is called “whitelist”, then takes responsibilities in checking if some important folders are within Snort's file system, and if they don't exist, our program creates them, for later on, moving the whitelists into this folders.

It handles as well, some of the configuration issues within Snort. Snort bases its configuration on a configuration file called “Snort.conf”. In this files, you can find variables, defines and some other statements related to configuration of our Snort sensor.

The sniffer here proposed, includes the defines statements into this file, for later moving this file into Snort folders, substituing the previous configuration file for this new one. A new one on the other hand, that is already fixed in order to indicate Snort where the rules' files are.[1]

3.4 Tree Linked List

The Tree-Linked list is the meat and potatoes of our sniffer's intentions. At this point of the project we're not taking in to consideration if it's a Modbus packet or any other TCP packet, what we are going to focus on, is to rescue IP addresses source, IP addresses destination, TCP source ports and TCP destination ports of the packets sniffed. This point give us the starting point for further ploughing.

The whole point of it, is to save all this relevant information about IPs and ports, to later elaborate (the program will) the rules that will be transmitted into Snort's file system. We should bear in mind, that it is very important to restrict the amount of data we are taking from the network. We don't

1. Rules must be contained in /opt/Snort/rules. The program creates this folder and places the blacklisting rule files and whitelisting rule files in here.

want to put unnecessary extra rules, or rules that refers to previous rules' especifications, because it will make Snort much slower. Thus, we are here dealing with a trade-off: max security versus accuracy and amount of rules.

To take just the necessary info then, we will design functions that will load a modified linked lists with the combinations of IP addresses and ports of queries and replies, with repeating any. The reason to use linked lists and not arrays is because if we used arrays, we would not know how much space we should allocate for it, and since we are picking packets as they flow in the wire, we can't dynamically establish the length of an array, some times could be 1000 elements others 10000 should be necessary...

We will introduce my Tree- Linked list here: We will start simple. What is a linked list? A linked list is a nested list of structs linked by addresses.

The struct used is as follow:

```
struct linked_list_node
        {
                void ptr_data;
                struct linked_list_node *ptr;
        }
```

Space for data.

Next node in the list.

An accurate visual representation would be:

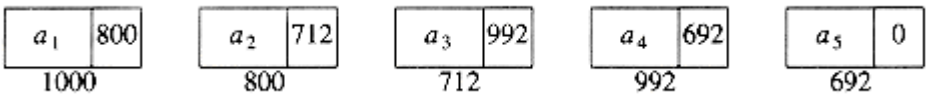


Fig.10 A single dimension linked list

where 1000, 800, 712, 992 and 692 are the memory addresses of these structures containing the data and a pointer to the next structure. a1, a2, a3, a4 and a5 are the data itself. The type void allows to store any kind of data in it.

These structures will be dynamically allocated, allowing users to keep big amounts of data, allowing the user to forget about stablishing sizes in arrays.

But the linked list shower before is not what we have in mind, a one-dimensional linked list doesn't help us to cope with our intentions. We will develop a Tree-Linked list. At the moment, I repeat,

we are dealing just with IP header information, we haven't got yet to the point of dealing with Modbus protocol, and the “linked list of linked lists” is consisting just of information related to IP-TCP.

The structure I'll be using for our purposes is this:

```
struct ip_node_t
{
        void *ptrdata;
        struct ip_node_t *ptr_r;
        struct ip_node_t *ptr_b;
}
```

Address to node's data.

Next node in the same level's list.

Next node into the downwards' list.

What better way to see the idea than through an ilustration of our tree? The next figure will clarify the idea, and later we explain the fields in my structure:

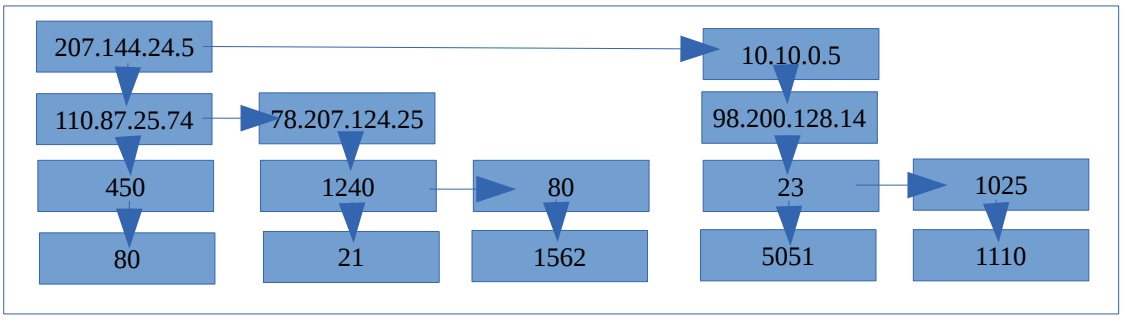


Fig.11 Part of a possible sample of a Tree-Linked list

The data structure then is composed by:

- void *ptrdata → is a pointer to data allocated dynamically by my code.
- struct ip_node_t *ptr_r → pointer to the next node in the same level (horizontal arrows in the graph).
- struct ip_node_t *ptr_b; → pointer to the node in the lower level (arrows pointing downwards).

Let's explain the previous image:

It could happen that the sniffer has picked up a packet with:

IP source address: 207.102.1.5

IP destination address: 10.10.1.37

Port source address: 345

Port destination address: 80

Later, in any moment, it picks up another one:

IP source address: 207.102.1.5

IP destination address: 10.10.1.115

Port source address: 3295

Port destination address: 80

Another packet any time later:

IP source address: 154.245.0.7

IP destination address: 10.10.1.37

Port source address: 1245

Port destination address: 21

Maybe after...

IP source address: 207.102.1.5

IP destination address: 10.10.1.115

Port source address: 80

Port destination address: 3987

And this would continue till the moment it reaches the number of packets we have tell the sniffer to sniff in the command line of our terminal.

4. IMPLEMENTATION

At this point we build the basis for the project, it consists just of a normal sniffer that picks the packets from the network in promiscuous mode. Promiscuous mode means that our NIC is able to pick not only the packets addressed to itself but packets addressed to any machine connected to that segment of the network. For this, the computer's NIC should be configurated, but normally Snort does it himself. To configure oneself the NIC in promiscuous mode we can do it through Linux terminal.

```
arturo@arturo-laptop:~$ sudo su
root@arturo-laptop:/home/arturo# ifconfig wlan0 down
root@arturo-laptop:/home/arturo# iwconfig wlan0 mode monitor
root@arturo-laptop:/home/arturo# iwconfig wlan0
wlan0     IEEE 802.11bgn  Mode:Monitor  Tx-Power=15 dBm
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:off
```

To put back your NIC in managed mode (otherwise you'll not be able to use your browser):

```
arturo@arturo-laptop:~$ sudo su
root@arturo-laptop:/home/arturo# ifconfig wlan0 down
root@arturo-laptop:/home/arturo# iwconfig wlan0 mode managed
root@arturo-laptop:/home/arturo# ifconfig wlan0 up
root@arturo-laptop:/home/arturo# iwconfig wlan0
wlan0     IEEE 802.11bgn  ESSID:"UPC943203"
          Mode:Managed  Frequency:2.437 GHz  Access Point: 70:71:BC:00:83:01
          Bit Rate=54 Mb/s   Tx-Power=15 dBm
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:off
          Link Quality=50/70  Signal level=-60 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:298   Missed beacon:0
```

Functionality: the packets' header information of any packet sniffed in our network segment, are dumped to the screen. Is it an IP packet? TCP over IP? Modbus over TCP-IP? Malformed IP packet? ARP? To illustrate this, next we have included a sample of the sniffed data during one of our tests:

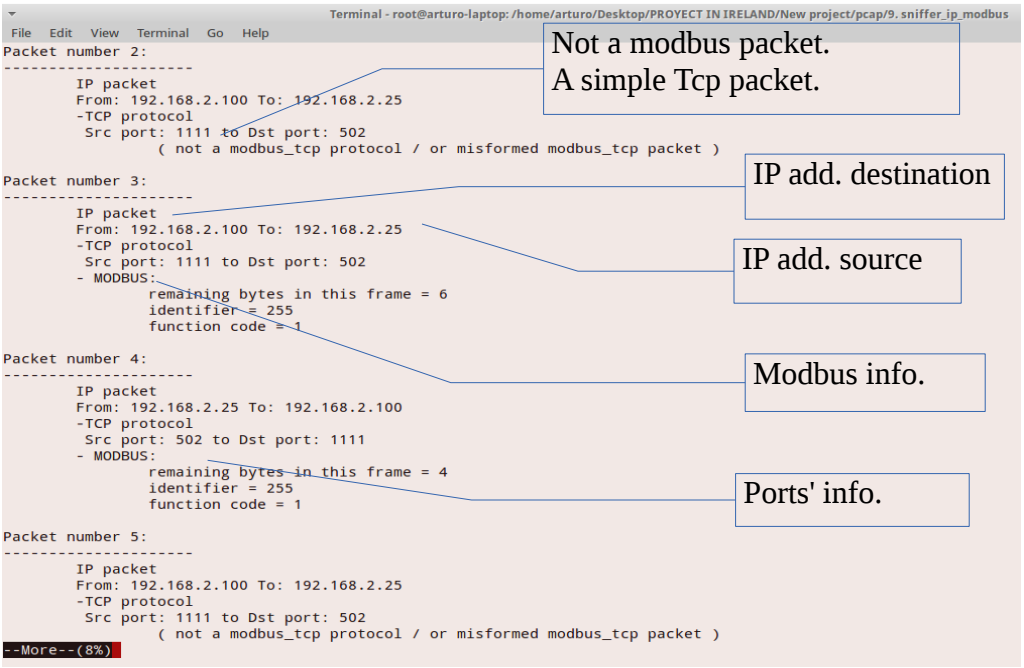


Fig.12 Screen sample of our sniffer's output

Another interesting feature to speak a bit about, is that, at this stage, we have started to implement the statistical study. Just a simple one: how many packets of one type has seen, what percentage represents from the total... from here we could develop future plug-ins to this sniffer as N-Gram study, but at the moment this is just an idea.

Since there are two ways our sniffer can work (sniffing from a pcap file or from a NIC), it can happen that when it picks packets from a pcap file, the time stadistics are useless, since the processing time of the pcap file usually is “0.00”, and then statistical time study leads to divisions by “0”, making the program to show this result as “-nan” or any other type of error.

A more thorough statistical study could be a great asset for our program as commented before, and could represent a next step to take in its design. It could be used for whitelisting information inside the payload, resulting in a very powerful tool for securing networks.

This next screen capture shows the output of a statistical study from a pcap file:

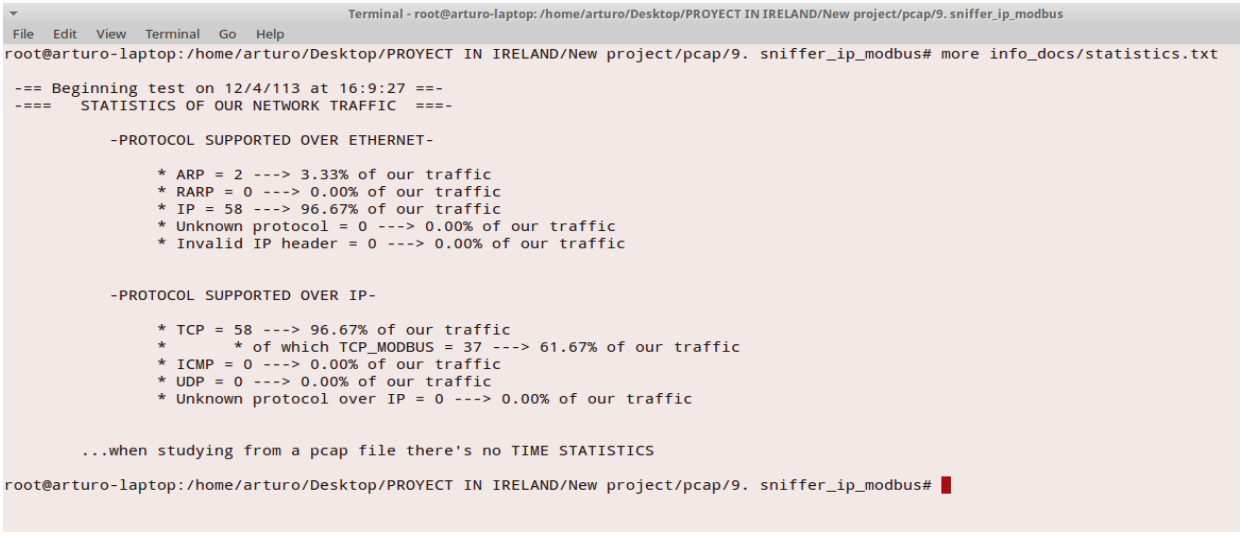


Fig.13 Screen sample of an example of statistical study

4.1 Tree-Linked List for IP

We create a code file called ip_func.c with its header ip_node.h. Let's explain this files:

ip_node.h → header file for ip linked list functions:

```
#ifndef IP_NODE_H_
#define IP_NODE_H_


struct ip_node_t
{
        void *ptrdata;
        struct ip_node_t *ptr_r;
        struct ip_node_t *ptr_b;
};

enum field_t
{
        ip_ip_from = 0,
        ip_ip_to = 1,
        ip_src_port = 2,
        ip_dst_port = 3
};

/* prototypes */

struct ip_node_t *createList (void);
struct ip_node_t *createNode(void *);
struct ip_node_t *insertBranch (struct ip_node_t *s,void *,void *,void *,void *);
void freeTree_ip(struct ip_node_t *);
```

Our node for the Tree – linked list.

Enumeration type that we will use to reference in which level of the Tree – linked list the program is. It can be understood by looking into the "insertBranch" function.

```
void readTree_ip(struct ip_node_t *);
#endif
```

Ip_func.c → file with the functions to use for the linked ip list.

```
#include "ip.h"
#include "ip_node.h"

extern FILE *ip_tree;
extern FILE *ip_tree_rules;


struct ip_node_t
*createList (void)
{
        return NULL;
};
```

This function it's not totally necessary but it makes the code more readable.

```
struct ip_node_t
*createNode(void *data)
{
        struct ip_node_t *s;
        s = malloc(sizeof(struct ip_node_t));

        if (s != NULL)
        {
                s->ptrdata = data;
                s->ptr_r = NULL;
                s->ptr_b = NULL;
        }
        return s;
}
```

This function will create a node and will be used in the next function in order to create nodes.
It also starts up the fields in the structure.

This is the function that we will use in the main program function. It go across the linked tree list and compares the info in the nodes. If the info is already in the tree, it returns, if not, it stores it in the last position of the level it corresponds to.

```
struct ip_node_t
*insertBranch (struct ip_node_t *s,void *ob1,void *ob2,void *ob3,void *ob4)
{
        struct ip_node_t *aux,*aux1,*aux2,*aux3,*prev,*prev1,*prev2,*prev3;
        struct ip_node_t *aux_loop;
        void *ob;
        enum field_t level = ip_ip_from;
        int prev_int;


        for ( aux = s, prev = NULL ; aux != NULL ; prev = aux, aux = aux->ptr_r)

        {
                if(!memcmp(aux->ptrdata,ob1,sizeof(struct in_addr)))
                {
                        level++;
                        for( aux1 = aux->ptr_b, prev1 = NULL ; aux1 != NULL ; prev1 = aux1, aux1 = aux1->ptr_r)
                        {
                                if(!memcmp(aux1->ptrdata,ob2,sizeof(struct in_addr)))
                                {

                                        level++;
                                        for( aux2 = aux1->ptr_b , prev2 = NULL ; aux2 != NULL ; prev2 = aux2,
```

During this big and nested for loop, the program runs along the Tree-list (if already created) and checks if the information is already contained in the it.

```
aux2 = aux2->ptr_r)
{
if(!memcmp(aux2->ptrdata,ob3,sizeof(u_short)))
{
        level++;
        for( aux3 = aux2->ptr_b , prev3 = NULL ; aux3 != NULL ;
                prev3 =aux3, aux3 = aux3->ptr_r)
        {
                if(!memcmp(aux3->ptrdata,ob4,sizeof(u_short)))
                {
                        return s;
```

If the Tree has not been created yet, this case(0) will add the first “branch”.

```
        }}}}}}}
}

if (prev == NULL) prev_int = 0; else prev_int = 1;

switch (prev_int) {

        case(0):

                ob = malloc(sizeof(struct in_addr));                    //ip source
                s = createNode(memcpy(ob,ob1,sizeof(struct in_addr)));

                ob = malloc(sizeof(struct in_addr));            //ip destination
                s->ptr_b = createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                aux_loop = s->ptr_b;

                ob = malloc(sizeof(u_short));                           //src port
                aux_loop->ptr_b = createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                           //dst port
                aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));

                return s;
```

At this point there's at least one node in our tree.

```
        default:

                switch(level) {
```

Here we see the enumerated type data. In this case is an Ip source.

The name of the function “insertBranch” is very reprensentative of what it does:
whenever it finds some new data, and therefore, non-stored data, it adds a whole “branch” to the Tree. A branch larger or smaller depending on What's the new item found. If the new item is an IP source... it will add IP source and destination and source and destiny ports.

```
case (ip_ip_from):
        ob = malloc(sizeof(struct in_addr)); //ip source
        prev->ptr_r = createNode(memcpy(ob,ob1,sizeof(struct in_addr)));
        aux_loop = prev->ptr_r;

        ob = malloc(sizeof(struct in_addr)); //ip destination
        aux_loop->ptr_b = createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
        aux_loop = aux_loop->ptr_b;

        ob = malloc(sizeof(u_short));                           //src port
        aux_loop->ptr_b = createNode(memcpy(ob,ob3,sizeof(u_short)));
        aux_loop = aux_loop->ptr_b;

        ob = malloc(sizeof(u_short));                           //dst port
        aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));
        return s;
```

Now the new item found is an Ip destination. So it adds the whole branch of information: Ip destination and ports. For this the enumeration type is useful.

```
case (ip_ip_to):
        ob = malloc(sizeof(struct in_addr)); //ip destination
        prev1->ptr_r = createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
        aux_loop = prev1->ptr_r;

        ob = malloc(sizeof(u_short));                           //src port
        aux_loop->ptr_b = createNode(memcpy(ob,ob3,sizeof(u_short)));
```

```
                        aux_loop = aux_loop->ptr_b;

                        ob = malloc(sizeof(u_short));                   //dst port
                        aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));
                        return s;

                case (ip_src_port):

                        ob = malloc(sizeof(u_short));                   //src port
                        prev2->ptr_r = createNode(memcpy(ob,ob3,sizeof(u_short)));
                        aux_loop = prev2->ptr_r;

                        ob = malloc(sizeof(u_short));                   //dst port
                        aux_loop->ptr_b = createNode(memcpy(ob,ob4,sizeof(u_short)));
                        return s;

                case (ip_dst_port):

                        ob = malloc(sizeof(u_short));                   //dst port
                        prev3->ptr_r = createNode(memcpy(ob,ob4,sizeof(u_short)));
                        return s;

                default:

                        printf("DEBUG NEEDED \n");
                        exit(1);

                }
        }
}
```

Malloc is the function used to get heap memory space. We allocate memory both for the data itself and for the node in the Tree-Linke list.

```
void
freeTree_ip(struct ip_node_t *s)
{
        struct ip_node_t *aux_i,*aux_j,*aux_k,*aux_r;

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                {
                                        free(aux_r->ptrdata);
                                        free(aux_r);
                                }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                        {
                                free(aux_k->ptrdata);
                                free(aux_k);
                        }

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                {
                        free(aux_j->ptrdata);
                        free(aux_j);
                }


        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
        {
```

The allocated memory, must be deallocated. This is the function for that purpose.

Freeing both data and linked list node.

```
                free(aux_i->ptrdata);
                free(aux_i);
        }

        return;
}



void
readTree_ip(struct ip_node_t *s)
{
        struct ip_node_t *aux_i,*aux_j,*aux_k,*aux_r;
        int aux1,aux2;
        struct in_addr *in_addr_aux1, *in_addr_aux2;
        int count=0;
        char mybuff[50];            /* inet_ functions use statically allocated memory */

        for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
        {
                in_addr_aux1 = (struct in_addr *)aux_i->ptrdata;
                for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
                {
                        in_addr_aux2 = (struct in_addr *)aux_j->ptrdata;
                        for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
                        {
                                aux1 = *(u_short *)aux_k->ptrdata;
                                for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
                                {
                                        aux2 = *(u_short *)aux_r->ptrdata;
                                        fprintf(ip_tree,"%5d: ",count++);
                                        fprintf(ip_tree,"Ip source: %15s, ",inet_ntoa(*in_addr_aux1));
                                        fprintf(ip_tree,"Ip destination: %15s, ",inet_ntoa(*in_addr_aux2));
                                        fprintf(ip_tree,"Port Source: %5d, ",aux1);
                                        fprintf(ip_tree,"Port destination: %5d \n",aux2);
                                        strcpy(mybuff,inet_ntoa(*in_addr_aux2));
                                        fprintf(ip_tree_rules,"pass ip %s %d <> %s %d \n",
                                                inet_ntoa(*in_addr_aux1),aux1,mybuff,aux2);

                                }
                        }
                }
        }
        return;
}
```

This function will write down both files:
- ip_tree.txt
- ip_tree.rules

Will be used in the main process of the program after the Tree has been completely loaded.

Moving along the Tree

Writing into files.

A little note about “inet_ntoa”: the string that is returned, is returned in a statically allocated buffer, which subsequent calls will overwrite.

In reference to files “ip_tree.txt” and “ip_tree.rules”:

Ip_tree.txt is the file in which our program will write in a readable format all the information previously stored into our Tree-linked list. An example of a line in this file:

1: Ip source: 192.168.2.25, Ip destination: 192.168.2.100, Port Source: 502, Port destination: 1111

Ip_tree.rules is the file in which the rules are written so that Snort can use them to compare it with the traffic in our network. This file will be stored, moved, later through some shell-scripts that we plug into the program's structure. An example of a line in this file:

pass ip 192.168.2.100 1111 <> 192.168.2.25 502

4.2 Implementation in Modbus

Right now we have a good understanding of how we can collect the info, keep it in dynamically allocated memory, and create files to store in a readable format both the info and the rules. It is time then to jump into the Modbus part.

For this we have again two files: Modbus_node.h and Modbus_node.c.

Modbus_node.h → header file for our Modbus linked list functions.

```
#ifndef Modbus_NODE_H_
#define Modbus_NODE_H_
```

```
struct Modbus_node_t
{
        void *ptrdata;
        struct Modbus_node_t *ptr_r;
        struct Modbus_node_t *ptr_b;
}
```

We have here a similar structure for the Tree-linked list node, but in this time it will reference modbus information.

```
enum m_field_t
{
        Modbus_ip_from = 0,
        Modbus_ip_to = 1,
        Modbus_src_port = 2,
        Modbus_dst_port = 3,
        Modbus_len = 4,
        Modbus_iden = 5,
        Modbus_func = 6
}
```

Now we have included three new types to the enumerated types: representing length of the data packet + 2, identification and function.

```
/* prototypes */

struct Modbus_node_t *m_createList (void);
struct Modbus_node_t *m_createNode(void *);
struct Modbus_node_t *m_insertBranch (struct Modbus_node_t *,void *,void *,void *,void *,void *,void *,void *);
void m_freeTree_Modbus(struct Modbus_node_t *);
void m_readTree_Modbus(struct Modbus_node_t *);

#endif
```

And the Modbus_func.c → with all the functions we will use for the Modbus Tree-linked list:

```
#include "ip.h"
#include "Modbus_node.h"

extern FILE *Modbus_tree;
extern FILE *Modbus_tree_rules;

struct Modbus_node_t
*m_createList (void)
{
        return NULL;
};
```

Function to create the list's first node's address for modbus

```
struct Modbus_node_t
*m_createNode(void *data)
{
        struct Modbus_node_t *s;
        s = malloc(sizeof(struct Modbus_node_t));

        if (s != NULL)
        {
                s->ptrdata = data;
                s->ptr_r = NULL;
                s->ptr_b = NULL;
        }
        return s;
}
```

Function to create list's modbus nodes

```
struct Modbus_node_t
*m_insertBranch (struct Modbus_node_t *s,void *ob1,void *ob2,void *ob3,void *ob4,void *ob5,void *ob6,void *ob7)
{
        struct Modbus_node_t *aux,*aux1,*aux2,*aux3,*aux4,*aux5;
        struct Modbus_node_t *aux6,*prev,*prev1,*prev2,*prev3,*prev4,*prev5,*prev6;
        struct Modbus_node_t *aux_loop;
        void *ob;
        enum m_field_t level = Modbus_ip_from;
        int m_prev_int;


         for ( aux = s, prev = NULL ; aux != NULL ; prev = aux, aux = aux->ptr_r)
        {
        if(!memcmp(aux->ptrdata,ob1,sizeof(struct in_addr)))
                {
                level++;
                for( aux1 = aux->ptr_b, prev1 = NULL ; aux1 != NULL ; prev1 = aux1, aux1 = aux1->ptr_r)
                {
                if(!memcmp(aux1->ptrdata,ob2,sizeof(struct in_addr)))
                        {
                        level++;
                        for( aux2 = aux1->ptr_b , prev2 = NULL ; aux2 != NULL ; prev2 = aux2, aux2 = aux2->ptr_r)
                        {
                        if(!memcmp(aux2->ptrdata,ob3,sizeof(u_short)))
                                {
                                level++;
                                for( aux3 = aux2->ptr_b , prev3 = NULL ; aux3 != NULL ; prev3 =aux3, aux3 =
                                aux3->ptr_r)
                                {
                                if(!memcmp(aux3->ptrdata,ob4,sizeof(u_short)))
                                        {
                                        level++;
                                        for( aux4 = aux3->ptr_b , prev4 = NULL ; aux4 != NULL ; prev4 =aux4,
                                                aux4= aux4->ptr_r)
                                        {
                                        if(!memcmp(aux4->ptrdata,ob5,sizeof(u_short)))
                                                {
                                                level++;
                                                for( aux5 = aux4->ptr_b , prev5 = NULL ; aux5 != NULL ;
                                                        prev5 =aux5, aux5 = aux5->ptr_r)
                                                {
                                                if(!memcmp(aux5->ptrdata,ob6,sizeof(u_char)))
                                                        {
                                                        level++;
                                                        for( aux6 = aux5->ptr_b , prev6 = NULL ; aux6 != NULL ;
                                                                prev6 =aux6, aux6 = aux6->ptr_r)
                                                        {
                                                        if(!memcmp(aux6->ptrdata,ob7,sizeof(u_char)))
```

Same function as “insertBranch” in the ip parallel function. Here three more steps are included in order to take into account the three new info data fields: id, func, len.

```
                                                {
                                                        return s;
                                                }
                                        }
                                        }
                                }
                                }
                        }
                        }

                }
                }
        }
        }

}

if (prev == NULL) m_prev_int = 0; else m_prev_int = 1;

switch (m_prev_int) {

        case(0):

                ob = malloc(sizeof(struct in_addr));                    //ip source
                s = m_createNode(memcpy(ob,ob1,sizeof(struct in_addr)));

                ob = malloc(sizeof(struct in_addr));            //ip destination
                s->ptr_b = m_createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                aux_loop = s->ptr_b;

                ob = malloc(sizeof(u_short));                           //src port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                           //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                           //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                            //iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                            //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));

                return s;

        default:

                switch(level) {

                        case (Modbus_ip_from):

                                ob = malloc(sizeof(struct in_addr)); //ip source
                                prev->ptr_r = m_createNode(memcpy(ob,ob1,sizeof(struct in_addr)));
                                aux_loop = prev->ptr_r;

                                ob = malloc(sizeof(struct in_addr)); //ip destination
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_short));                           //src port
```

Here we include the three new fields for Modbus

New enumerated type.

```
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));

                return s;


        case (Modbus_ip_to):

                ob = malloc(sizeof(struct in_addr)); //ip destination
                prev1->ptr_r = m_createNode(memcpy(ob,ob2,sizeof(struct in_addr)));
                aux_loop = prev1->ptr_r;

                ob = malloc(sizeof(u_short));                   //src port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    // iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                return s;

        case (Modbus_src_port):

                ob = malloc(sizeof(u_short));                   //src port
                prev2->ptr_r = m_createNode(memcpy(ob,ob3,sizeof(u_short)));
                aux_loop = aux_loop->ptr_r;

                ob = malloc(sizeof(u_short));                   //dst port
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_short));                   //len Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    // iden Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                aux_loop = aux_loop->ptr_b;

                ob = malloc(sizeof(u_char));                    //func Modbus
                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                return s;
```

```
                        case (Modbus_dst_port):

                                ob = malloc(sizeof(u_short));                   //dst port
                                prev3->ptr_r = m_createNode(memcpy(ob,ob4,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_r;

                                ob = malloc(sizeof(u_short));                   //len Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_char));                    // iden Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_char));                    //func Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                                return s;

                        case(Modbus_len):

                                ob = malloc(sizeof(u_short));                   //len Modbus
                                prev4->ptr_r = m_createNode(memcpy(ob,ob5,sizeof(u_short)));
                                aux_loop = prev4->ptr_r;

                                ob = malloc(sizeof(u_char));                    // iden Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                                aux_loop = aux_loop->ptr_b;

                                ob = malloc(sizeof(u_char));                    //func Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));

                                return s;

                        case(Modbus_iden):

                                ob = malloc(sizeof(u_char));                    // iden Modbus
                                prev5->ptr_r = m_createNode(memcpy(ob,ob6,sizeof(u_char)));
                                aux_loop = prev5->ptr_r;

                                ob = malloc(sizeof(u_char));                    //func Modbus
                                aux_loop->ptr_b = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                                return s;

                        case(Modbus_func):

                                ob = malloc(sizeof(u_char));                    //func Modbus
                                prev6->ptr_r = m_createNode(memcpy(ob,ob7,sizeof(u_char)));
                                return s;

                        default:

                                printf("DEBUG NEEDED \n");
                                exit(1);

                }
        }
}
```

Three new steps into this version of The function

```
void
m_freeTree_Modbus(struct Modbus_node_t *s)
{
        struct Modbus_node_t *aux_i,*aux_j,*aux_k,*aux_r,*aux_s,*aux_t,*aux_u;
```

```
for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
		for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
			for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
				for (aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
					for (aux_t = aux_s->ptr_b ; aux_t != NULL ; aux_t = aux_t->ptr_r)
						for (aux_u = aux_t->ptr_b ; aux_u != NULL ;
							aux_u = aux_u->ptr_r)
						{
							free(aux_u->ptrdata);
							free(aux_u);
						}


for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
		for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
			for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
				for (aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
					for (aux_t = aux_s->ptr_b ; aux_t != NULL ; aux_t = aux_t->ptr_r)
					{
						free(aux_t->ptrdata);
						free(aux_t);
					}

for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
		for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
			for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
				for (aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
				{
					free(aux_s->ptrdata);
					free(aux_s);
				}

for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
		for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
			for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
			{
				free(aux_r->ptrdata);
				free(aux_r);
			}


for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
		for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
		{
			free(aux_k->ptrdata);
			free(aux_k);
		}

for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r)
	{
		free(aux_j->ptrdata);
		free(aux_j);
	}


for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r)
{
	free(aux_i->ptrdata);
	free(aux_i);
}
```

```
return;
}

void
m_readTree_Modbus(struct Modbus_node_t *s)
{
	struct Modbus_node_t *aux_i,*aux_j,*aux_k,*aux_r,*aux_s,*aux_t,*aux_u;
	int aux1,aux2,aux3;
	struct in_addr *in_addr_aux1, *in_addr_aux2;
	char aux_char1,aux_char2;
	int count=0;
	char mybuff[50];		/* inet_ functions use statically allocated memory */

	for( aux_i = s ; aux_i != NULL ; aux_i = aux_i->ptr_r) {
	in_addr_aux1 = (struct in_addr *)aux_i->ptrdata;
	for( aux_j = aux_i->ptr_b ; aux_j != NULL ; aux_j = aux_j->ptr_r) {
		in_addr_aux2 = (struct in_addr *)aux_j->ptrdata;
		for (aux_k = aux_j->ptr_b ; aux_k != NULL ; aux_k = aux_k->ptr_r)
		{
			aux1 = *(u_short *)aux_k->ptrdata;
			for (aux_r = aux_k->ptr_b ; aux_r != NULL ; aux_r = aux_r->ptr_r)
			{
				aux2 = *(u_short *)aux_r->ptrdata;
				for(aux_s = aux_r->ptr_b ; aux_s != NULL ; aux_s = aux_s->ptr_r)
				{
					aux3 = *(u_short *)aux_s->ptrdata;
					for(aux_t = aux_s->ptr_b ; aux_t != NULL ; aux_t = aux_t->ptr_r)
					{
						aux_char1 = *(u_char *)aux_t->ptrdata;
						for(aux_u = aux_t->ptr_b ; aux_u != NULL ; aux_u = aux_u->ptr_r)
						{
						aux_char2 = *(u_char *)aux_u->ptrdata;

						fprintf(Modbus_tree,"%5d: ",count++);
						fprintf(Modbus_tree,"Ip src: %14s, ",inet_ntoa(*in_addr_aux1));
						fprintf(Modbus_tree,"Ip dst: %14s, ",inet_ntoa(*in_addr_aux2));
						fprintf(Modbus_tree,"Port src: %5d, ",aux1);

						fprintf(Modbus_tree,"Port dst: %5d, ",aux2);
						fprintf(Modbus_tree,"lenght_data: %5d, ",aux3);
						fprintf(Modbus_tree,"ident: %5d, ",(u_char)aux_char1);
						fprintf(Modbus_tree,"funct code: %5d \n",(u_char)aux_char2);
						strcpy(mybuff,inet_ntoa(*in_addr_aux2));
						fprintf(Modbus_tree_rules,"pass ip %s %d <> %s %d
						(Modbus_func: %d ;Modbus_unit: %d;)\n",
							inet_ntoa(*in_addr_aux1),aux1,mybuff,aux2,
						(u_char)aux_char2,(u_char)aux_char1);
						}
					}

				}
			}
		}
	}
	}
	return;
}
```

Walking through this part of the code, we can see the similarities between the functions for IP and for Modbus, actually Modbus' functions are just an extension including the three more fields to study in this protocol, but the algorithm in whiche they are based is exactly the same.

4.3 Scripts shell

The operative system we have been working in during this project as stated before is Linux, to be more precise it is a Debian in its Xubuntu version. Including shell scripts and using them in the code of our sniffer it's been relatively easy thanks to the system call “system()”. The shell program is one of the possitive aspects of working in Linux and it has brought useful and good results.

System( ) 's man page : “man system” in our shell program.

NAME
system - execute a shell command

SYNOPSIS
#include <stdlib.h>

*int system(const char *command);*

DESCRIPTION
system() executes the command specified in command by calling “ /bin/sh -c command ”, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

We wanted to make automatic the inclusion of the rule files created by our program into Snort's file system, and add the possibility for our users to include, through our program as well, other blacklisting rule files into Snort's workings. For this, and considering that Snort has been installed through a process of “configure && install” from a source code, our sniffer will detect if the proper folders exists in Snort's file system and will copy the rules files inside. At the same time, it will add automatically the corresponding “include” states in Snort configuration file allowing the system administrator to forget about his issues.

This scripts are:

```
#! /bin/sh

if [ -d ./conf_Snort_files ]
        then
                cp ./conf_Snort_files/classification.config /opt/Snort/etc/.
                cp ./conf_Snort_files/reference.config /opt/Snort/etc/.
                cp  ./conf_Snort_files/Snort.conf ./rules/Snort.conf
        else
                echo " !! revise your sniffer folder, conf_Snort_files folder is missed !! "
                exit 1
fi
```

Folder “conf_snort_files” inside our program's structure. It contains configuration files for Snort and some other files related to information about how to use our sniffer.

If this info_docs exists then it carries out the commands listed here, if not, there's some missed stuff and you should revise the contents of your programm

```
#! /bin/sh

if [ -d ./info_docs ]
        then
                cp ./info_docs/*.rules ./rules/.
```

Once the program has finished, folder “info_docs” should have been created, containing info texts and rules files created by it.

```
                cp ./conf_Snort_files/Snort.conf ./rules/Snort.conf
                chown $SUDO_USER ./rules/Snort.conf
                cd rules
                echo "\n"
                for file in $(ls *.rules)
                        do
                                cp ./$file /opt/Snort/rules/$file
                                echo "\t...$file moved into /opt/Snort/rules and included into Snort.conf"
                                sleep 1
                                echo "include \$RULE_PATH/$file" >> ./Snort.conf
                        done
                cp ./Snort.conf /opt/Snort/etc/.
                cd ..
        else
                echo "debug needed"
                exit 1
fi
```

Folder “rules” is a folder created to keep the rules generated by our sniffer and some other blacklisting rules add by the user.

This for-loop is an interesting feature:
It goes along all the files in “rules” folder copying them into the right place into Snort file system and including an “include” statement into snort.conf configuration file.

```
#! /bin/sh

if [ -d /opt/Snort/etc ]
        then
                echo "\t\t... /opt/Snort/etc exists"
        else
                echo "\t\t... /opt/Snort/etc doesn't exists..."
                if [ -d /opt ]
                        then
                        cd /opt
                        if [ -d /opt/Snort ]
                                then
                                        mkdir /opt/Snort/etc
                                else
                                        mkdir /opt/Snort
                                        mkdir /opt/Snort/etc
                                fi
                        else
                                mkdir /opt
                                mkdir /opt/Snort
                        mkdir /opt/Snort/etc

                fi

        echo "\t\t\t...creating /opt/Snort/etc"
fi
```

The next three scripts are run at the beginning of the main code of our sniffer, they will check for the existence of:

- /opt/snort/etc
- /opt/snort/rules
- /var/log/snort

This are important folders into Snort's file system, and they will house the files related to configuration, rules and logs. This is why is so important to check if they exists and if they don't, this scripts will create them. Actually, the previous scripts copy their files into these folders.

```
#! /bin/sh

if [ -d /opt/Snort/rules ]
then
echo "\t\t... /opt/Snort/rules exists"
else
echo "\t\t... /opt/Snort/rules doesn't exists ..."
mkdir /opt/Snort/rules
echo "\t\t\t...creating /opt/Snort/rules"
fi


#! /bin/sh
```

```
if [ -d /var/log/Snort ]
then
        echo "\t\t... /var/log/Snort exists"
else
        echo "\t\t... /var/log/Snort doesn't exists..."
        mkdir /var/log/Snort
        echo "\t\t\t\t...creating /var/log/Snort"
fi
```

4.4 Putting it all together: Main Code and Callback Function

First we show the header file: ip.h

```
#ifndef _IP_H_
#define _IP_H_


#include <time.h>
#include <sys/types.h>
#include <pcap/pcap.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>


#define BUFSIZE 2048
#define APP_NAME "ip.out"
#define SIZE_ETHERNET 14
#define ETHER_ADDR_LEN 6

enum eth_type
{
        ARP=0,
        RARP=1,
        IP=2,
        UNKNOWN=3,
        INV_IP_HEADER=4
};

enum upper_eth_type
{
        TCP_NO_Modbus=0,
        TCP_Modbus=1,
        ICMP=2,
        UDP=3,
        UP_UNKNOWN=4,
};

/* data structures for IP */

        #define ETHER_ADDR_LEN 6

        /* Ethernet header */
        struct sniff_ethernet {
                u_char ether_dhost[ETHER_ADDR_LEN];     /* Destination host address */
                u_char ether_shost[ETHER_ADDR_LEN];     /* Source host address */
                u_short ether_type;                     /* IP? ARP? RARP? etc */
        };
```

Includes with all header files needed.

Constants.

Enumeration data type for protocols over the link layer.

Enumeration data type for protocols over IP.

Data structure for Ethernet, this sniffer is designed to work over Ethernet.

```
/* IP header */
struct sniff_ip {
        u_char ip_vhl;      /* version , header length */
        u_char ip_tos;      /* type of service */
        u_short ip_len;     /* total length */
        u_short ip_id;      /* identification */
        u_short ip_off;     /* fragment offset field */
        u_char ip_ttl;      /* time to live */
        u_char ip_p;        /* protocol */
        u_short ip_sum; /* checksum */
        struct in_addr ip_src;        /* source ip address */
        struct in_addr ip_dst; /* dest ip address */
};

#define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef uint32_t tcp_seq;


struct sniff_tcp {
         u_short th_sport;           /* source port */
         u_short th_dport;           /* destination port */
         tcp_seq th_seq;             /* sequence number */
         tcp_seq th_ack;             /* acknowledgement number */
         u_char  th_offx2;           /* data offset, rsvd */
        #define TH_OFF(th)     (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;            /* flags */
        u_short th_win;             /* window */
        u_short th_sum;              /* checksum */
        u_short th_urp;             /* urgent pointer */
};

/* Modbus-TCP header */
struct sniff_Modbus_tcp {
        u_short mtcp_trans_id;      /* synchronization */
        u_short mtcp_prot;          /* protocol identifier */
        u_short mtcp_len;/* remaining! bytes in this frame */
        u_char    mtcp_iden;        /* identifier */
        u_char    mtcp_func;        /* function code */
};
```

Data Ip structure: it will house the different fields an Ip packet must have.

Macros: the first field in the Ip Structure is double, I mean, it contains the fields for “version” and “header length”. Header length will be useful along the program.

Data for TCP structure containing the different fields in a TCP segment.

TCP – Modbus data structure. These fields were previously commented when the introduction to Modbus.

```
/* protypes */

void our_callback(u_char *,const struct pcap_pkthdr* ,const u_char* );
void print_app_banner(char *,int);
void create_Statistics (struct tm *,struct tm *,double,int,int ) ;
void tail_banner(void);
void pantallazo(int);
void merge_ip(char *);


#endif
```

And now the main code with the call back function and some other secundary functions such as the statistical study: ip.c

```
#include "ip.h"
#include "ip_node.h"
#include "Modbus_node.h"
```

It's in the main code and the call-back function that we use the functions previously explained, for this we need to include their header files: ip_node.h and modbus_node.h along with ip.h

```
/*global variables for statistics */

        int packet_type[5]={0,0,0,0,0};     /* statistics of ethernet frame type in our network */
        int info_type[5]={0,0,0,0,0};               /* statistics of info type inside the IP payload */
        FILE *statistics;
        FILE *ip_tree;
        FILE *sniff_data;
        FILE *ip_tree_rules;
        FILE *Modbus_tree;
        FILE *Modbus_tree_rules;
        struct ip_node_t *s;
        struct Modbus_node_t *r;



/*
 * MAIN
 *
 */

int
main (int argc, char **argv)
{
        /* vars */

        int     n_packets;                      /* number limit of packets we sniff */
        char    char_aux_i=0,char_aux_n=0;      /* checking flags from the terminal process */

        char    errbuf[PCAP_ERRBUF_SIZE];       /* holds the error string message in pcap functions */
        pcap_t  *handler;                       /* pcap handler */

        time_t  timer_init,timer_end;                   /* for time-stamps */
        struct  tm *st_timer_start,*st_timer_end;   /* for time-stamps */
        double  time_diff;                              /* holds the difference of time the program has used */

        int i;                          /* for loops */


        //getting options for the program
        int flag_n=0, flag_i=0, flag_f=0;
        int c;
        char *nvalue = NULL;
        char *ivalue = NULL;
        char *fvalue = NULL;

        opterr = 0;

        while ((c = getopt (argc, argv, "n:i:f:")) != -1)
        {
                switch(c)
                {
                        case 'n':
                                flag_n = 1;
                                nvalue = optarg;
                                n_packets = atoi(nvalue);
                                break;
                        case 'i':
                                flag_i = 1;
                                ivalue = optarg;
                                break;
                        case 'f':
                                flag_f = 1;
                                fvalue = optarg;
                                break;
                        case '?':
                                fprintf(stderr,"usage: ./ip.out -i <interface> -n <number_of_packets>\n");
                                fprintf(stderr,"usage: ./ip.out -f <file_name>\n");
                                exit(1);
                        default:
```

Global variables

Getopt is used to break up (parse) options in command lines for easy parsing by shell procedures, and to check for legal options.

```
                        fprintf (stderr,"Unknown option character `\\x%x'.\n",optopt);
                        fprintf(stderr,"usage: ./ip.out -i <interface> -n <number_of_packets>\n");
                        fprintf(stderr,"usage: ./ip.out -f <file_name>\n");
                        exit(1);
        }
}


// checking for a correct combination of switches

if( !( (flag_f && !flag_i && !flag_n  ) || (!flag_f && flag_i && flag_n ) ) )
{
        fprintf(stderr,"missuse of the program switches\n");
        fprintf(stderr,"usage: ./ip.out -i <interface> -n <number_of_packets>\n");
        fprintf(stderr,"usage: ./ip.out -f <file_name>\n");
        exit(1);
}else {
        if( !flag_f ) print_app_banner(ivalue,n_packets);
        else print_app_banner(fvalue,0);
}


// initialize linked list

s = (struct ip_node_t *)createList();
r = (struct Modbus_node_t *)m_createList();


// opening sniff_data to hold the sniffer's output

sniff_data = fopen("sniff_data.txt","w");
ip_tree_rules = fopen("ip_tree.rules","w");
Modbus_tree_rules = fopen("Modbus_tree.rules","w");


// stablishing handler for sniffing:

if(!flag_f)
{
        if( ( handler = pcap_open_live(ivalue,BUFSIZ,1,10000,errbuf)) == NULL )
        {
                printf("\n%s %s: %s\n","Couldn't open device",ivalue,errbuf);
                fprintf(stderr,"\n%s\n","exiting.....");
                exit(1);
        }

} else if( ( handler = pcap_open_offline(fvalue, errbuf) ) == NULL )
        {
                printf("\n%s %s: %s\n","Couldn't open device",ivalue,errbuf);
                fprintf(stderr,"\n%s\n","exiting.....");
                exit(1);
        }


// printing time stamp of beginning

timer_init=time(NULL);
st_timer_start=localtime(&timer_init);
printf("\n -== %s %02d/%02d/%d at %02d:%02d:%02d ==-\n","Beginning test on",
```

Are we sniffing from a pcap file or from the network interface?

Starting our Tree-Linked lists:
- one for ip alone.
- another for Modbus.
- following their algorithm we could perfectly add new Tree-Linked list and create a whole information data base about our network.

Are we sniffing from our NIC? In that case we will use “pcap_open_life”.

Are we sniffing from a file? In this case we will use “pcap_open_offline”.

```
            st_timer_start->tm_mday,st_timer_start->tm_mon,st_timer_start->tm_year,
            st_timer_start->tm_hour,st_timer_start->tm_min,st_timer_start->tm_sec );


// opening files for the linked trees

ip_tree = fopen("ip_tree.txt","w");
Modbus_tree = fopen("Modbus_tree.txt","w");



// entering in the loop

if((pcap_loop(handler,n_packets,our_callback,NULL))==-1)
{
            fprintf(stderr,"\n%s\n","error ocurred while in loop, exiting now...");
            exit(1);
}

// exiting,printing statistics and closing files

timer_end = time(NULL);
st_timer_end = localtime(&timer_end);
printf("\n%s %2.2lf %s\n"," -== Time test process :",time_diff = difftime(timer_end,timer_init),"segs. ==-");


//creating and closing statistics file

pcap_close(handler);
statistics = fopen("statistics.txt","w");
for(i=0,n_packets=0;i<5;i++) n_packets += packet_type[i];
create_Statistics(st_timer_start,st_timer_end,time_diff,n_packets,flag_f);

fclose(statistics);
fclose(sniff_data);


//reading from the tree and free-ing the allocated space

readTree_ip(s);
m_readTree_Modbus(r);

fclose(ip_tree);
fclose(Modbus_tree);

freeTree_ip(s);
m_freeTree_Modbus(r);

printf("\n%s\n\n"," ----statistic control finished, please, wait----");


fprintf(ip_tree_rules,"alert ip any any -> any any (msg:\"communication out of our ip-white-list\";)");
fclose(ip_tree_rules);
fprintf(Modbus_tree_rules,"alert ip any any -> any any (msg:\"communication out of our Modbus-white-list\";)");
fclose(Modbus_tree_rules);

            // merging together the rules in order to get more compact rule files
merge_ip("ip_tree.rules");


// creating folder info_docs and moving files into it
tail_banner();
exit(0);

}
```

Establish call-back function for everytime a packet hits our NIC or a new packet is detected in a pcap file.

Handler is our call-back function, inside of it, we use the Tree-linked list functions.

Creating file for statistics in packet types and time.

At this point we have all the data saved into our Trees and it's time for our program to read it and create the corresponding files: .txt and .rules

The allocated space in the heap must be freed.

Added feature: most of the information is double. There are “queries” and “replies”. They both are going to give us the same information, and if we don't remove one of the copies, we have a doble extension in our ip.rules file.

```
/*
 * our functions
 */

void
our_callback(u_char *args,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
        static int count = 1;                   /* packet counter */
        u_short eth_type;               /* ethernet type in host byte order for switch use */
        int Modbus_flag = 0;                    /* Modbus_flag = 1 when it detects a Modbus protocol packet */
        char mybuff[50];                /* inet_ functions use statically allocated memory */


        /*aux variables to help in the use of tree_linked_list */
        u_short sportaux,dportaux;
        u_short m_lenaux;
        u_char m_idenaux,m_funcaux;


        /* declare pointers to packet headers */
        const struct sniff_ethernet *ethernet;          /* The ethernet header [1] */
        const struct sniff_ip *ip;                      /* The IP header */
        const struct sniff_tcp *tcp;
        const struct sniff_Modbus_tcp *Modbus_tcp;              /* The Modbus TCP header */


        /* sizes of ip frame and tcp segment */
        int size_header_ip;
        int size_header_tcp;
        char test_Modbus = 0;
        fprintf(sniff_data,"\nPacket number %d:\n", count);
        fprintf(sniff_data,"---------------------- \n", count);
        count++;



        /* define ethernet header torrent*/
        ethernet = (struct sniff_ethernet*)(packet);

        eth_type=ntohs(ethernet->ether_type);


        switch (eth_type) {
                case(0x0806):
                        fprintf(sniff_data,"\tARP packet\n");
                        packet_type[ARP]++;
                        return;
                case(0x8035):
                        fprintf(sniff_data,"\tRARP packet\n");
                        packet_type[RARP]++;
                        return;
                case(0x0800):
                        fprintf(sniff_data,"\tIP packet\n");
                        packet_type[IP]++;
                        break;
                default:
                        fprintf(sniff_data,"\tnot an ARP/RARP/IP packet\n");
                        packet_type[UNKNOWN]++;
        }

         /* define/compute ip header offset */
        size_header_ip = IP_HL(ip = (struct sniff_ip*)(packet + SIZE_ETHERNET))*4;
        if (size_header_ip < 20)
        {
                packet_type[INV_IP_HEADER]++;
                fprintf(sniff_data,"\t Invalid IP header length: %u bytes\n", size_header_ip);
                return;
        }
```

CALL-BACK FUNCTION: everytime a new packet arrives to the NIC or in a pcap file a new packet is read, this functions is called.

Initial byte of a new packet. Establishing-filling in the structures.

What is it?
- ARP
- RARP
- IP
- UNKOWN

- maybe an invaled IP packet

```
/* print source and destination IP addresses */

strcpy(mybuff,inet_ntoa(ip->ip_src));
fprintf(sniff_data,"\tFrom: %s To: %s\n", mybuff, inet_ntoa(ip->ip_dst));


/* determine protocol */
switch(ip->ip_p)
{
        case IPPROTO_TCP:
                fprintf(sniff_data,"\t-TCP protocol\n");
                info_type[TCP_NO_Modbus]++;
                break;
        case IPPROTO_UDP:
                fprintf(sniff_data,"\t-UDP protocol\n");
                info_type[UDP]++;
                return;
        case IPPROTO_ICMP:
                fprintf(sniff_data,"\t-ICMP protocol\n");
                info_type[ICMP]++;
                return;
        default:
                fprintf(sniff_data,"\t-Not a TCP/UDP/ICMP protocol\n");
                info_type[UP_UNKNOWN]++;
                return;
}
 /* define/compute tcp header offset */
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_header_ip);
size_header_tcp = TH_OFF(tcp)*4;
if (size_header_tcp < 20)
{
        packet_type[INV_IP_HEADER]++;
        return;
}

fprintf(sniff_data,"\t Src port: %d to Dst port: %d\n", ntohs(tcp->th_sport),ntohs(tcp->th_dport));

 /* define/print Modbus header fields */
/* is it an IP packet carrying Modbus data ? */

if ( ( ntohs(ip->ip_len) - size_header_ip ) == size_header_tcp ) test_Modbus = 0;
else test_Modbus = 1;

Modbus_tcp = (struct sniff_Modbus_tcp*)(packet + SIZE_ETHERNET + size_header_ip + size_header_tcp);

test_Modbus = test_Modbus && (Modbus_tcp->mtcp_prot == 0);
test_Modbus = test_Modbus &&(Modbus_tcp->mtcp_iden != 0)&&(Modbus_tcp->mtcp_len != 0 );
test_Modbus = test_Modbus && (Modbus_tcp->mtcp_iden < 256) && (Modbus_tcp->mtcp_func < 256 );


if ( test_Modbus )
{
                info_type[TCP_Modbus]++;
                fprintf(sniff_data,"            - Modbus: \n");
                fprintf(sniff_data,"\t\t%s = %d \n","remaining bytes in this
                        frame",ntohs(Modbus_tcp->mtcp_len) );
                fprintf(sniff_data,"\t\t%s = %d \n","identifier",Modbus_tcp->mtcp_iden);
                fprintf(sniff_data,"\t\t%s = %d \n","function code",Modbus_tcp->mtcp_func);
                Modbus_flag = 1;

} else {
                fprintf(sniff_data,"\t\t%s \n"," ( not a Modbus_tcp protocol / or misformed Modbus_tcp
                packet )");
                Modbus_flag = 0;
}
```

What is the IP packet keeping?

How do we know that it is Modbus what is inside the TCP packet...??? there are several tests to carry on...

```
	/* sending info to the linked tree list  */

	if (!Modbus_flag)
	{
		if ((eth_type == 0x0800)&&(ip->ip_p == IPPROTO_TCP))
		{
			sportaux = ntohs(tcp->th_sport);
			dportaux = ntohs(tcp->th_dport);
			s = (struct ip_node_t *)insertBranch(s,(struct in_addr *)&ip->ip_src,
				(struct in_addr *)&ip->ip_dst, (u_short *)&sportaux,(u_short *)&dportaux);
		}

	} else
		{
			sportaux = ntohs(tcp->th_sport);
			dportaux = ntohs(tcp->th_dport);
			m_lenaux = ntohs(Modbus_tcp->mtcp_len);
			m_idenaux = Modbus_tcp->mtcp_iden;
			m_funcaux = Modbus_tcp->mtcp_func;
			r = (struct Modbus_node_t *)m_insertBranch(r,(struct in_addr *)&ip->ip_src,
				(struct in_addr *)&ip->ip_dst, (u_short *)&sportaux,(u_short *)&dportaux,
				(u_short *)&m_lenaux,(u_char *)&m_idenaux,(u_char *)&m_funcaux);

		}
	return;
}

void
print_app_banner(char *v,int n)
{
	char c = '0' ;
	int i ;
	char *snt_scrp[] =
	{
		"\n Making sure you have /var/log/Snort...\n",
		"./var_log_Snort.sh ",
		"\n Making sure you have the /opt/Snort/etc directory. \n",
		"./opt_Snort_etc.sh ",
		"\n Making sure directory /opt/Snort/rules exists. \n",
		"./opt_Snort_rules.sh ",
		" \n",
		"echo \"\n\"",
		NULL
	};
	char *ord[] =
	{
		"more ./conf_Snort_files/important_considerations",
		"more ./conf_Snort_files/input",
		"more ./conf_Snort_files/output",
		NULL
	};


	pantallazo(1);
	printf("Info banner: \n");
	for(i = 0 ; ord[i] ; i++)
	{
		system(ord[i]);
		pantallazo(20);
	}

	printf("Checking the system...\n");
	for( i = 0 ; snt_scrp[i] ; )
	{
		printf("%s",snt_scrp[i++]);
		system(snt_scrp[i++]);
		sleep(3);
	}
```

At this point we have an IP packet carrying a simple TCP protocol that doesn't wrap inside any Modbus data.

In case of Modbus, we call the modbus Tree-linked list functions.

Secondary function that is used to display a banner at the beginning of our program when run.

Use of “system” system call.

Using the scripts shell.

```
	pantallazo(4);
	if (n == 0)
	{
		printf("STARTING SNIFFING.....");
		sleep(1);
		printf(" %s\n"," -================-");
		printf(" -== %s ==- \n", APP_NAME);
		printf(" -== %s = %s ==-\n","interface",v);
		printf(" %s\n"," -================-");
		sleep(1);
	}

	else{
		printf("STARTING SNIFFING.....");
		sleep(1);
		printf(" %s\n"," -================-");
		printf(" -== %s ==- \n", APP_NAME);
		printf(" -== %s = %s ==-\n","interface",v);
		printf(" -== %s = %d ==-\n","number of packets to study ", n );
		printf(" %s\n"," -================-");
		sleep(1);
	}
	return;
}

void
create_Statistics (struct tm *begin,struct tm *end,double td, int pkt,int f) {

	enum eth_type et_aux;			/* aux through for-loops*/
	enum upper_eth_type uet_aux;		/* aux through for-loops*/
	char *et_uet;
	int sum_et=0,sum_uet=0;
	float assess=0;

	fprintf(statistics,"\n -== %s %d/%d/%d at %d:%d:%d ==-\n","Beginning test on",
		begin->tm_mday,begin->tm_mon,begin->tm_year,begin->tm_hour,
		begin->tm_min,begin->tm_sec);


	fprintf(statistics," -=== %s ===- \n","  STATISTICS OF OUR NETWORK TRAFFIC ");

	fprintf(statistics,"\n\t%s  \n\n","   -PROTOCOL SUPPORTED OVER ETHERNET- ");

	for(et_aux=ARP; et_aux<=INV_IP_HEADER; et_aux++)
	{
		switch(et_aux)
		{
			case(ARP): et_uet="ARP";break;
			case(RARP): et_uet="RARP";break;
			case(IP): et_uet="IP";break;
			case(UNKNOWN): et_uet="Unknown protocol";break;
			case(INV_IP_HEADER):et_uet="Invalid IP header";break;
			default: fprintf(statistics,"\t!!debug needed!!\n");
		};
		assess=(float)(packet_type[et_aux])/pkt;
		fprintf(statistics,"\t\t* %s = %d ---> %2.2f%c of our traffic\n",
			et_uet,packet_type[et_aux],assess*100,'%');
	}

	fprintf(statistics,"\n\n");
	fprintf(statistics,"\t%s  \n\n","   -PROTOCOL SUPPORTED OVER IP- ");


	for(uet_aux=TCP_NO_Modbus ;uet_aux<=UP_UNKNOWN ;uet_aux++)
	 {
		switch(uet_aux)
		{
			case(TCP_NO_Modbus): et_uet="TCP";break;
```

This function creates:
- Time statistics.
- Packet number statistics.

```
                        case(UDP): et_uet="UDP";break;
                        case(ICMP): et_uet="ICMP";break;
                        case(TCP_Modbus): et_uet="\t* of which TCP_Modbus";break;
                        case(UP_UNKNOWN): et_uet="Unknown protocol over IP";break;
                        default: fprintf(statistics,"\t!!debug needed!!\n");
                }
                assess=(float)(info_type[uet_aux])/pkt;
                fprintf(statistics,"\t\t* %s = %d ---> %2.2f%c of our traffic\n",
                        et_uet,info_type[uet_aux],assess*100,'%');
        }

        /* when data dumped from a file, there's no sense in time statistics, the file takes 0 secs for the program
        to be examined, resulting in divisions by 0 in our next piece of code*/

        if ( f )
        {
                fprintf(statistics,"\n\n\t...when studying from a pcap file there's no TIME STATISTICS\n\n");
                return;
        }

        fprintf(statistics,"\n%s %f %s\n\n"," -== TIME STATISTICS : the tests has taken ",td," segs.  ==-");


        fprintf(statistics,"\t%s \n","   -PROTOCOL SUPPORTED OVER ETHERNET- ");

        for(et_aux=ARP; et_aux<=INV_IP_HEADER; et_aux++)
        {
                switch(et_aux)
                {
                        case(ARP): et_uet="ARP";break;
                        case(RARP): et_uet="RARP";break;
                        case(IP): et_uet="IP";break;
                        case(UNKNOWN): et_uet="Unknown protocol";break;
                        case(INV_IP_HEADER):et_uet="Invalid IP header";break;
                        default: fprintf(statistics,"\t!!debug needed!!\n");
                }
                fprintf(statistics,"\t\t* %s has %2.3lf packets/sec.\n",et_uet,(double)(packet_type[et_aux])/(int)td);
        }

        fprintf(statistics,"\n\n");
        fprintf(statistics,"\t%s  \n","   -PROTOCOL SUPPORTED OVER IP- ");


        for(uet_aux=TCP_NO_Modbus ;uet_aux<=UP_UNKNOWN ;uet_aux++)
        {
                switch(uet_aux)
                {
                        case(TCP_NO_Modbus): et_uet="TCP";break;
                        case(UDP): et_uet="UDP";break;
                        case(ICMP): et_uet="ICMP";break;
                        case(TCP_Modbus): et_uet="\t* of which TCP_Modbus";break;
                        case(UP_UNKNOWN): et_uet="Unknown protocol over IP";break;
                        default: fprintf(statistics,"\t!!debug needed!!\n");
                }
                fprintf(statistics,"\t\t* %s has %2.3lf packets/sec.\n",et_uet,(double)(info_type[uet_aux])/(int)td);
        }

        fprintf(statistics,"\n -== %s %02d/%02d/%d at %02d:%02d:%02d ==-\n\n -== %s %02d/%02d/%d at %02d:%02d:
%02d ==-\n",
                "Test start:",begin->tm_mday,begin->tm_mon,begin->tm_year, begin->tm_hour,begin->tm_min,
                begin->tm_sec,"Test finish:",end->tm_mday,end->tm_mon,end->tm_year,end->tm_hour,end->tm_min,
                end->tm_sec);

        fprintf(statistics,"\t -=== %s ===- \n"," ------ END OF OUR STATISTICS FILE ------- ");


        return;
}
```

When the program has kept all the important information into the Tree lists, now it's moment to sort out all the different files into the program's file system and specially in Snort's file system.

```
void
tail_banner(void)
{
        char c = '0';
        int i ;
        char *sentences[] =
        {
                "Files created: \n",
                "\t - sniff_data.txt :\t holds the sniffer's output\n",
                "\t - statistics.txt :\t holds statistical information\n",
                "\t - ip_tree.txt    :\t holds all the combinations of ip @ and ports observed into our network traffic\n",
                "\t - Modbus_tree.txt    :\t holds all the combinations of ip @, ports and Modbus fiels observed into our
                network traffic\n",
                "\t - ip_tree.rules  :\t holds the ip rules to whitesniff our network, placed in ' /opt/Snort/rules ' \n",
                "\t - Modbus_tree.rules  :\t holds the Modbus rules to whitesniff our network, placed in ' /opt/Snort/rules
                ' \n",
                NULL
        };
        char *orders[] =
        {
                "mkdir info_docs","\nCreating folder info_docs...\n",
                "mv ./ip_tree.txt ./info_docs/.","\t...Moving ip_tree.txt  into folder info_docs\n",
                "mv ./Modbus_tree.txt ./info_docs/.","\t...Moving Modbus_tree.txt  into folder info_docs\n",
                "mv ./statistics.txt ./info_docs/.","\t...Moving statistics.txt into folder info_docs\n",
                "mv ./sniff_data.txt ./info_docs/.","\t...Moving sniff_data.txt into folder info_docs\n",
                "mv ./ip_tree.rules ./info_docs/.","\t...Moving ip_tree.rules into info_docs\n",
                "mv ./Modbus_tree.rules ./info_docs/.","\t...Moving Modbus_tree.rules into info_docs\n\n",
                "./moving_conf_files.sh","... moving 'classification.config' and 'reference.config' into /opt/Snort/etc\n",
                NULL
        };
        char *more_sentences[] =
        {
                "\n\n\nPay attention: in this same directory, there's a folder named 'rules',\n",
                "you should load this directory with the black-listing rules you want for Snort. \n",
                "This program will include automatically the names into Snort.conf, releasing you from this task.\n",
                "Include now, if you need, some .rules files or PRESS LETTER c(lower case) + ENTER (case sensitive) to
                continue: ",
                NULL
        };
        char *last_sentences[] =
        {
                "\n\nNow you can go to info_docs folder contained in this same directory, and consult the data
                gathered.\n",
                "...The appropiate rule file has been created and located in /opt/Snort/rules as well.\n\n\n\n",
                NULL
        };


        pantallazo(5);
        system("rm -r ./info_docs 2> /dev/null");
        for (i = 0 ; sentences[i] ; i++)
        {
                printf("%s",sentences[i]);
                sleep(1);
        }
        for(i = 0; orders[i] ;  )
        {
                if(system(orders[i++]) != -1 ) printf("%s",orders[i++]);
                sleep(2);
        }

        for ( i = 0; more_sentences[i] ; i++ )
        {
                printf("%s",more_sentences[i]);
                sleep(2);
        }
```

```
	fflush(stdin);
	for ( ; c != 'c' ; )
	{
		scanf("%c",&c);
		if( c != 'c' ) printf("\nPRESS LETTER C (case sensitive) + ENTER to continue: ");
		fflush(stdin);
	}

	if(system("./moving_rule_files.sh") != -1)
	{
			printf("\n\t...Copying our whilelist compliant rules into /opt/Snort/rules\n");
			sleep(1);
			printf("\t...Moving black-list compliant rules from our folder 'rules'into /opt/Snort/rules\n");
			printf("\t...Moving Snort.conf configuration file with our 'includes' into /opt/Snort/etc \n");
	}
	sleep(2);

	for ( i = 0; last_sentences[i] ; i++ )
	{
		printf("%s",last_sentences[i]);
		sleep(2);
	}

	return;
}


void
pantallazo (int j)
{
	sleep (j);
	system("clear");
	return;
}
```

Clears the screen to keep writing information to the user.

5. DEPLOYMENT AND TEST

Our program's file system is composed by 2 directories and 31 files . From this 31 files, 4 are pcap files that are useful to test the functionality of the whitelisting-sniffer.

```
.
├── 0.compiling.sh
├── 1.remove.sh
├── conf_Snort_files
│   ├── classification.config
│   ├── important_considerations
│   ├── input
│   ├── output
│   ├── reference.config
│   └── Snort.conf
├── DOC_README.odt
├── DOC_README.pdf
├── FC1-permit.pcap
├── Modbus_FC_1_Coil.pcap
├── http.cap
├── Modbus.pcap
├── ip.c
├── ip_func.c
├── ip.h
├── ip_node.h
├── merge_ip.c
├── Modbus_func.c
├── Modbus_node.h
├── rules
│   ├── backdoor.rules
│   ├── community-virus.rules
│   ├── dos.rules
│   ├── ftp.rules
│   └── icmp.rules
├── moving_conf_files.sh
├── moving_rule_files.sh
├── opt_Snort_etc.sh
├── opt_Snort_rules.sh
└── var_log_Snort.sh
```

Folder conf_snort_files

Folder: rules

Explanation of the different files:

- 0.compiling.sh : script shell to compile the program. The program must be compiled previous use.
- 1.remove.sh : script shell that remove not needed files in the program's file system. It's positive to use it before to compile the program but not after it. It would erase the

executable files and you would have to compile back.

- Folder “conf_Snort_files ” : it contains important configuration program files. The most important among them is “Snort.conf”. Over this file, our sniffer with write the “include” statements, and after that, it will move this file into Snort's file system, overwriting the previous existing Snort.conf in there.
- DOC_README.odt && DOC_README.pdf : contain information about the whitelisting-sniffer.
- Next “.pcap” files: FC1-permit.pcap , Modbus_FC_1_Coil.pcap, http.cap, Modbus.pcap are files downloaded from Internet in order to test the correct function of the program.
- ip.c , ip_func.c, ip.h, ip_node.h, merge_ip.c, Modbus_func.c, Modbus_node.h have already been explained along this text in previous points.
- Folder “rules” : it conforms an important feature to our whitelisting program. During its execution, at the end of it, there's a moment in which the program informs you that rule files created are going to be moved into Snort's file system, advicing you and giving time to include inside this folder some other blacklisting file rules that the user could consider to be useful for Snort to use.
- moving_conf_files.sh , moving_rule_files.sh, opt_Snort_etc.sh , opt_Snort_rules.sh, var_log_Snort.sh have also already been explained. They interact with Snort's file system contributing to a more automatic use of this program with Snort.

Let's go through an example of execution:

1. Let's lists the contents of our folder:

```
arturo@arturo-laptop:~/...../pcap/9. sniffer_ip_Modbus$ ls -l
total 272
-rwxr--r-x 1 arturo arturo   289 May  6 13:39 0.compiling.sh
-rwxr--r-x 1 arturo arturo   284 May  3 16:03 1.remove.sh
drwxrwxr-x 2 arturo arturo  4096 May  6 13:10 conf_Snort_files
-rw-rw-r-- 1 arturo arturo 20850 May  6 13:28 DOC_README.odt
-rw-rw-r-- 1 arturo arturo 29415 May  6 13:29 DOC_README.pdf
-rw-rw-r-- 1 arturo arturo  3138 May  2 18:48 FC1-permit.pcap
-rw-r--r-- 1 arturo arturo 25803 May  2 13:37 http.cap
-rw-r--r-- 1 arturo arturo 17864 May  6 13:15 ip.c
-rw-r--r-- 1 arturo arturo  5846 May 11 16:28 ip_func.c
-rw-rw-r-- 1 arturo arturo  4204 May 14 20:25 ip_func.o
-rw-r--r-- 1 arturo arturo  2561 May  6 13:02 ip.h
-rw-r--r-- 1 arturo arturo   481 May  1 19:11 ip_node.h
-rw-rw-r-- 1 arturo arturo 17572 May 14 20:25 ip.o
-rw-r--r-- 1 arturo arturo  1340 May  3 15:33 merge_ip.c
-rw-rw-r-- 1 arturo arturo  2464 May 14 20:25 merge_ip.o
-rw-rw-r-- 1 arturo arturo  4670 May  2 18:08 Modbus_FC_1_Coil.pcap
-rw-r--r-- 1 arturo arturo 11957 May 11 16:35 Modbus_func.c
-rw-rw-r-- 1 arturo arturo  6976 May 14 20:25 Modbus_func.o
-rw-r--r-- 1 arturo arturo   633 Apr 30 15:11 Modbus_node.h
-rw-rw-r-- 1 arturo arturo  8337 May  2 16:24 Modbus.pcap
-rwxr--r-x 1 arturo arturo   322 Apr 30 18:28 moving_conf_files.sh
-rwxr--r-x 1 arturo arturo   501 May  3 15:47 moving_rule_files.sh
-rwxr--r-x 1 arturo arturo   433 May 11 21:48 opt_Snort_etc.sh
```

```
-rwxr--r-x 1 arturo arturo   211 May 11 21:48 opt_Snort_rules.sh
drwxrwxr-x 2 arturo arturo  4096 May 13 23:00 rules
-rwxrwxr-x 1 arturo arturo 29950 May 14 20:25 sniffer
-rwxr--r-x 1 arturo arturo   208 May  3 16:06 var_log_Snort.sh
```

2. Use of the removing script shell:

```
arturo@arturo-laptop:~/...../pcap/9. sniffer_ip_Modbus$ ./1.remove.sh
...remove.sh done
```

3. Compilation process:

```
arturo@arturo-laptop:~/...../pcap/9. sniffer_ip_Modbus$ sudo su
[sudo] password for arturo:
root@arturo-laptop:/home/arturo/...../pcap/9. sniffer_ip_Modbus# ./0.compiling.sh
...compilation process done
...usage:  ./sniffer -i <interface> -n <number of packets>
...usage:  ./sniffer -f <file_name>
```

4. Execution of “sniffer” in the form of ./sniffer -f <file_name>

```
root@arturo-laptop:/home/arturo/..../pcap/9. sniffer_ip_Modbus# ./sniffer -f Modbus_FC_1_Coil.pcap
```

Modbus_FC_1_Coil.pcap is a pcap file downloaded from Internet that contains precollected information from a sniffing session.

5.1 Different Screens in the Execution Process

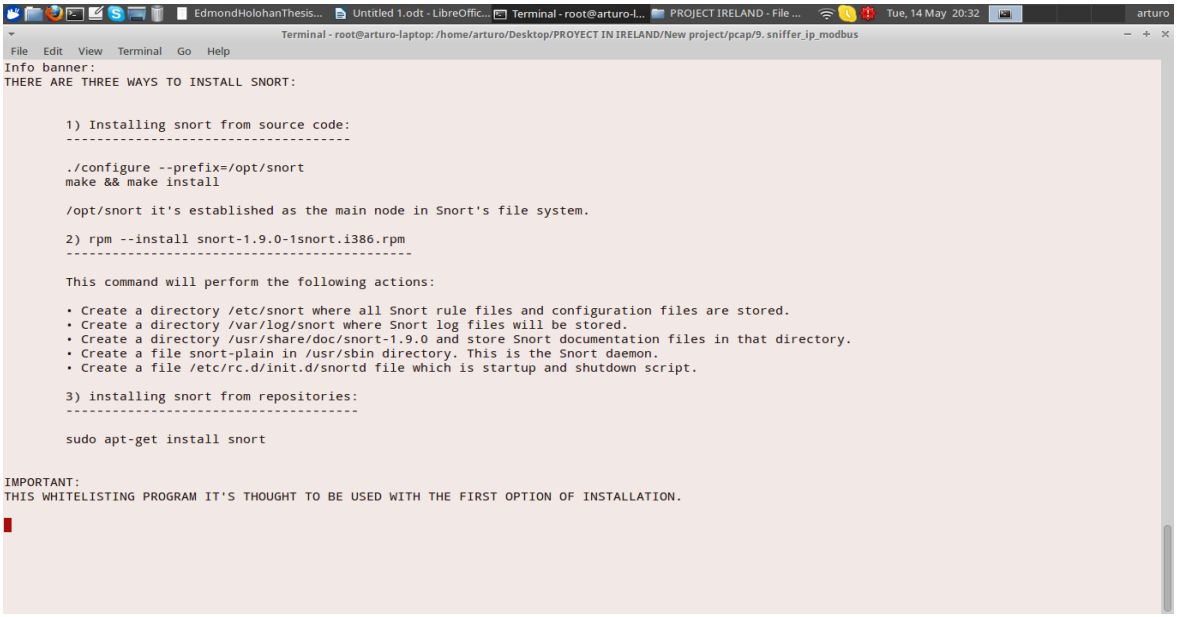


Fig. 14 Brief note about Snort's intallation

Fig. 15 Contents of the program's file system

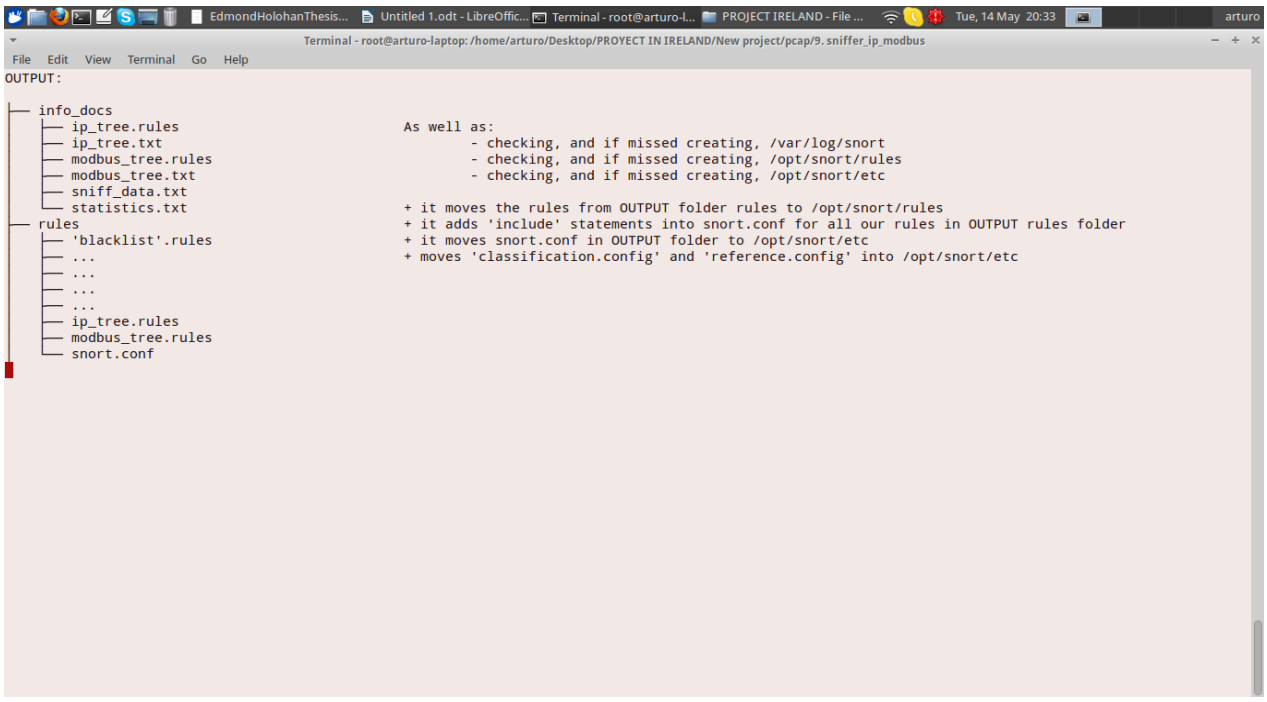


Fig.16 Files resulting from the execution of the sniffer

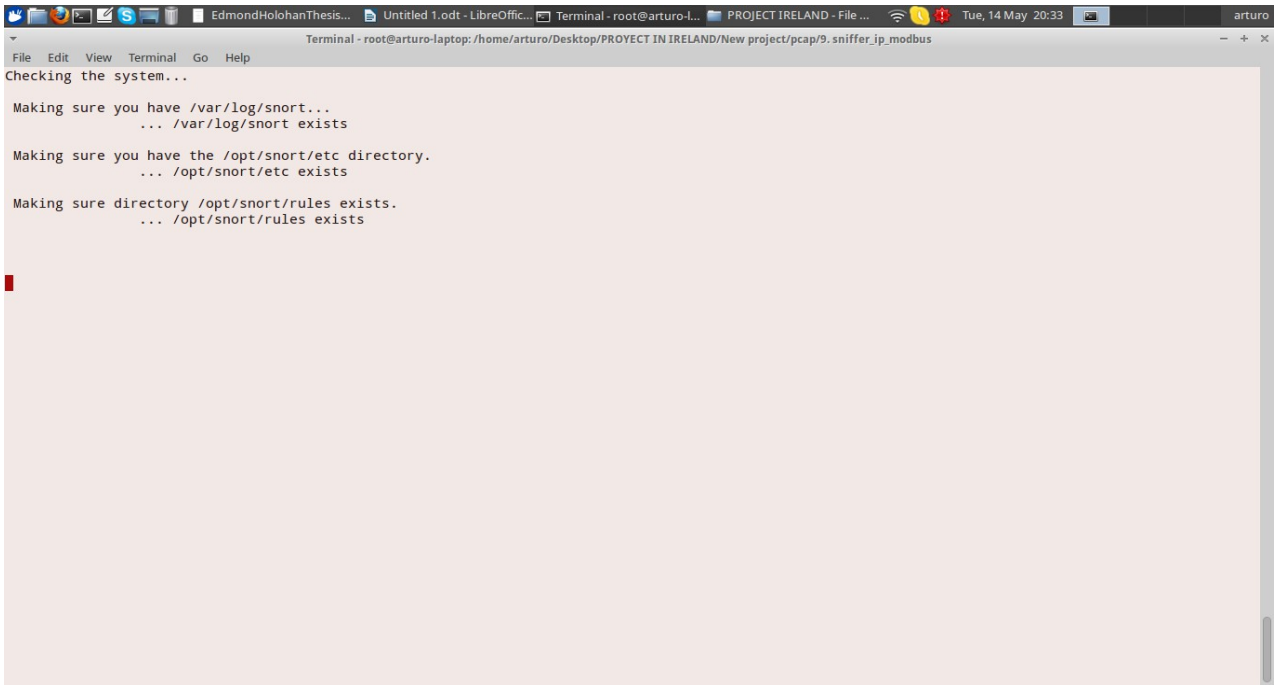


Fig.17 Checking / creating folders in Snort's file system

Fig.18 Sniffing process (no results dumped directly into screen)

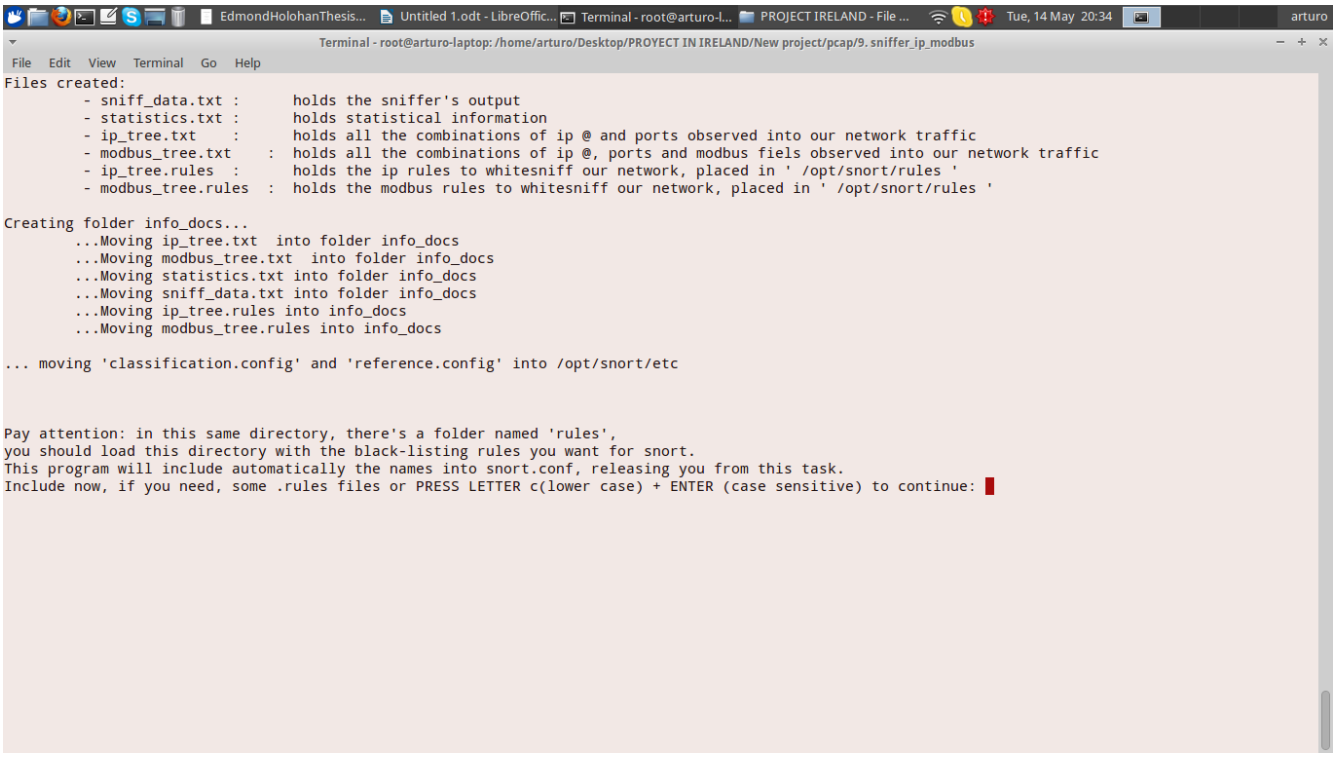


Fig.19 Results explained and further reconfiguration

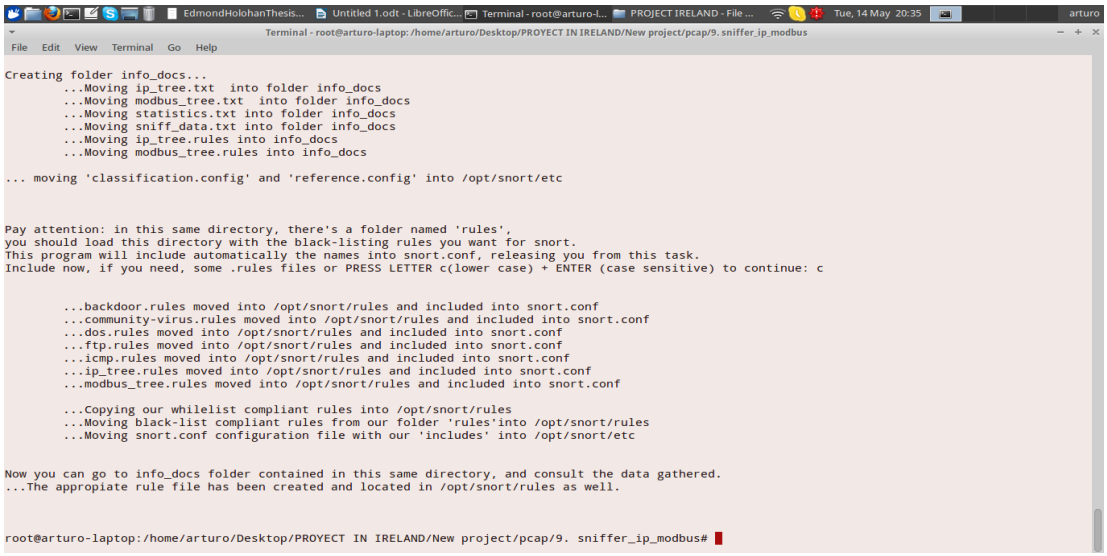


Fig.20 Last screen

6. “info_docs” folder:

root@arturo-laptop:/home/arturo/...../pcap/9. sniffer_ip_Modbus# cd info_docs/ && tree

.
├── ip_tree.rules
├── ip_tree.txt
└── Modbus_tree.rules

├── Modbus_tree.txt
├── sniff_data.txt
└── statistics.txt

```
arturo@arturo-laptop:~/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus/info_docs$ more ip_tree.rules
pass ip 192.168.2.100 1111 <> 192.168.2.25 502
alert ip any any -> any any (msg:"communication out of our ip-white-list";)
arturo@arturo-laptop:~/Desktop/PROJECT IN IRELAND/New project/pcap/9. sniffer_ip_modbus/info_docs$ more modbus_tree.rules
pass ip 192.168.2.25 502 <> 192.168.2.100 1111 (modbus_func: 255 ;modbus_unit: 255;)
pass ip 192.168.2.25 502 <> 192.168.2.100 1111 (modbus_func: 1 ;modbus_unit: 255;)
pass ip 192.168.2.25 502 <> 192.168.2.100 1111 (modbus_func: 5 ;modbus_unit: 255;)
pass ip 192.168.2.100 1111 <> 192.168.2.25 502 (modbus_func: 1 ;modbus_unit: 255;)
pass ip 192.168.2.100 1111 <> 192.168.2.25 502 (modbus_func: 5 ;modbus_unit: 255;)
alert ip any any -> any any (msg:"communication out of our modbus-white-list";)
```

Fig.21 Example of IP.rules & Modbus.rules file

7. “rules” folder:

root@arturo-laptop:/home/arturo/...../pcap/9. sniffer_ip_Modbus/info_docs# cd ../rules/ && tree
.
├── backdoor.rules
├── community-virus.rules
├── dos.rules
├── ftp.rules
├── icmp.rules
├── ip_tree.rules
├── Modbus_tree.rules
└── Snort.conf

6. CONCLUSIONS

It is impossible to come up with a tool that secures 100% any information system. Hackers always find a way to dupe security systems, this is why is so important to adopt a proactive attitude when working in this issues. This is why it is so important for any security administrator to know the network he is working in and disallow anything different from what they very well know it is normal.

Speaking about tools, any administrator will always try a combination of them. Some tools are very good in some environments and later they work poorly in others. Some, propose solutions based on one of the three methods presented during the abstract of this text (blacklisting, whitelisting or anomaly detection) and they loose sight over some parts of reality, leaving important security gaps. So... how to achieve a good level of security without spending big amounts of money?

Nowadays, it exists a lot of opensource programs that help to you to to achieve your desired security level. In this thesis we have propose a solution for securing networks, specially those working with Industrial Control System Protocol like Modbus, taking advantage of Snort's features, a IDS available as opensource in www.Snort.org that has become very famous among system administrators.

The design we propose in OSNA is based on whitelists and blacklists. It is known very well that just using a single approach of these ones alone, brings to quite disastrous results. We have been working to build up a solution that could merge whitelisting methods and blacklisting methods, leaving the door open to near future inclussions addressing methods of anomaly detection based in N-grams. This total-approach would constitute a very powerful resource to maintain a clean system and control possible intrusions.

Specifically, the blacklisting approach is achieved through Snort, we will be taking full advantage of its functionality, and we'll be adding our whitelisting sniffer to it in order to contribute with whitelisting features, making Snort even more complete and making the whole process very automatic for any user.

Our whitelisting method, comes up with all the information about the network segment we connect the program in and during a test period in which we run the code, it creates files that describe what it is known to be the correct behaving of the devices' interconnections. Everything that is out of this files, will be consider as a threat and counter measures taken against it.

The tool in this thesis proposed is then a very handy one, allowing to automatize the whole process of representation of a network in files reducint the amount of work any aministrator would have to use to create a whitelist of sessions connections among the devices. We are already thinking of future plug-ins into our program, working in this same direction of “describing the net”, we believe our sniffer, will be a very important resource for security information employees.

6.1 Further research

A nice step to take for this research, could consist in a graphical interface. This program together with Snort, works from terminal linux, making difficult to work for those ones that have no idea of Linux or have never interacted with a terminal where all commands are written instead of “clicked”.

Annexe A: Snort.

This annexe consists of a summarize of the open source book Intrusion Detection Systems with Snort Advanced IDS Techniques Using Snort, Apache, MySQL, PHP, and ACID. [InPe01]

A.1 Introduction to Intrusion Detection and Snort

Intrusion Detection methods started appearing in the last few years. Using Intrusion Detection methods, you can collect and use information from known types of attacks and find out if someone is trying to attack your network or particular hosts.

A comprehensive security system consists of multiple tools, including:

- Firewalls: used to block unwanted incoming as well as outgoing traffic of data.

- Intrusion Detection systems (IDS): used to find out if someone has gotten into or is trying to get into your network.

- Vulnerability assessment tools: used to find and plug security holes present in your network. Information collected from vulnerability assessment tools is used to set rules on firewalls so that these security holes are safeguarded from malicious Internet users.

These tools can work together and exchange information with each other. Some products provide complete systems consisting of all of these products bundled together.

What is Intrusion Detection?

Intrusion Detection Systems fall into two basic categories: signature-based Intrusion Detection Systems and anomaly detection systems. Intruders have signatures, like computer viruses, that can be detected using software. You try to find data packets that contain any known intrusion-related signatures or anomalies related to Internet protocols. Based upon a set of signatures and rules, the

detection system is able to find and log suspicious activity and generate alerts. Snort is primarily a rule-based IDS, but input plug-ins are present to detect anomalies in protocol headers as well.

Snort uses rules stored in text files that can be modified by a text editor. Rules are grouped in categories. Rules belonging to each category are stored in separate files. These files are then included in a main configuration file called Snort.conf. Snort reads these rules at the start-up time and builds internal data structures to apply these rules to captured data. Finding signatures and using them in rules is a tricky job, since the more rules you use, the more processing power is required to process captured data in real time. It is important to implement as many signatures as you can using as few rules as possible.

Signatures:

Signature is the pattern that you look for inside a data packet. A signature is used to detect one or multiple types of attacks. For example, the presence of “scripts/iisadmin” in a packet going to your web server may indicate an intruder activity. Signatures may be present in different parts of a data packet depending upon the nature of the attack. For example, you can find signatures in the IP header, transport layer header (TCP or UDP header) and/or application layer header or payload. You will learn more about signatures later in this book.

Alerts :

Alerts are any sort of user notification of an intruder activity. When an IDS detects an intruder, it has to inform security administrator about this using alerts. Alerts may be in the form of pop-up windows, logging to a console, sending e-mail and so on. Alerts are also stored in log files or databases where they can be viewed later on by security experts.

Logs :

The log messages are usually saved in file. By default Snort saves these messages under /var/log/Snort directory. However, the location of log messages can be changed using the command line switch when starting Snort. Log messages can be saved either in text or binary format. The binary files can be viewed later on using Snort or tcpdump program. Logging in binary format is faster because it saves some formatting overhead. In high-speed Snort implementations, logging in binary mode is

necessary.

Sensor :

The machine on which an Intrusion Detection System is running is also called the sensor in the literature because it is used to “sense” the network.

Components of Snort:

Snort is logically divided into multiple components. These components work together to detect particular attacks and to generate output in a required format from the detection system. A Snort-based IDS consists of the following major components:

- Packet Decoder
- Preprocessors
- Detection Engine
- Logging and Alerting System
- Output Modules

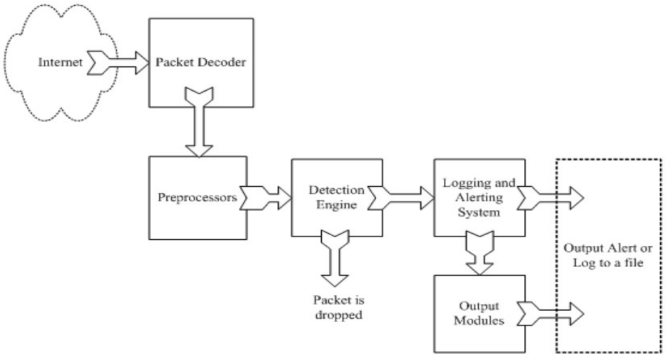


Fig. 22 Snort's inner workings schema

Packet Decoder : takes packets from different types of network interfaces and prepares the packets to be preprocessed or to be sent to the detection engine. The interfaces may be Ethernet, SLIP, PPP …

Preprocessors : components or plug-ins that can be used with Snort to arrange or modify data packets before the detection engine does some operation to find out if the packet is being used by an intruder. Some preprocessors also perform detection by finding anomalies in packet headers and generating alerts. They are very important for any IDS to prepare data packets to be analyzed against rules in the detection engine.

Hackers use different techniques to fool an IDS in different ways. For example, you may have created a rule to find a signature “scripts/iisadmin” in HTTP packets. If you are matching this string exactly, you can easily be fooled by a hacker who makes slight modifications to this string. For

example:

- “scripts/./iisadmin”
- “scripts/examples/../iisadmin”
- “scripts\iisadmin”
- “scripts/.\iisadmin”

To complicate the situation, hackers can also insert in the web Uniform Resource Identifier (URI) hexadecimal characters or Unicode characters which are perfectly legal as far as the web server is concerned. Note that the web servers usually understand all of these strings and are able to preprocess them to extract the intended string “scripts/ iisadmin”. However if the IDS is looking for an exact match, it is not able to detect this attack. A preprocessor can rearrange the string so that it is detectable by the IDS.

Preprocessors are also used for packet defragmentation. Receiving systems are capable of reassembling these smaller units again to form the original data packet. On IDS, before you can apply any rules or try to find a signature, you have to reassemble the packet. For example, half of the signature may be present in one segment and the other half in another segment. To detect the signature correctly you have to combine all packet segments. Hackers use fragmentation to defeat Intrusion Detection Systems.

The preprocessors are used to safeguard against these attacks. Preprocessors in Snort can defragment packets, decode HTTP URI, re-assemble TCP streams and so on. These functions are a very important part of the Intrusion Detection System.

The Detection Engine : is the most important part of Snort. Its responsibility is to detect if any intrusion activity exists in a packet. The detection engine employs Snort rules for this purpose. The rules are read into internal data structures or chains where they are matched against all packets. If a packet matches any rule, appropriate action is taken; otherwise the packet is dropped. Appropriate actions may be logging the packet or generating alerts.

This is the time-critical part of Snort. Depending upon some factors, it may take different amounts of time to respond to different packets or you may even drop some packets and may not get a true real-time response:

- Number of rules
- Power of the machine on which Snort is running
- Speed of internal bus used in the Snort machine
- Load on the network

The detection system can dissect a packet and apply rules on different parts of the packet. These parts may be:

- The IP header of the packet.
- The Transport layer header. (TCP, UDP) or other transport layer headers. It may also work on the ICMP header.
- The application layer level header. Application layer headers include, but are not limited to, DNS header, FTP header, SNMP header, and SMTP header. You may have to use some indirect methods for application layer headers, like offset of data to be looked for.
- Packet payload. This means that you can create a rule that is used by the detection engine to find a string inside the data that is present inside the packet.

In Snort version 2 all rules are matched against a packet before generating an alert. After matching all rules, the highest priority rule is selected to generate the alert.

Logging and Alerting System: depending upon what the detection engine finds inside a packet, the packet may be used to log the activity or generate an alert. Logs are kept in simple text files, tcp-dump-style files or some other form. All of the log files are stored under /var/log/ Snort folder by default. You can use –l command line options to modify the location of generating logs and alerts.

Output Modules : output modules or plug-ins can do different operations depending on how you want to save output generated by the logging and alerting system of Snort. Depending on the configuration, output modules can do things like the following:

- Simply logging to /var/log/Snort/alerts file or some other file
- Sending SNMP traps
- Sending messages to syslog facility
- Logging to a database like MySQL or Oracle.
- Generating eXtensible Markup Language (XML) output
- Modifying configuration on routers and firewalls.
- Sending Server Message Block (SMB) messages to Microsoft Windows-based machines

Other tools can also be usedto send alerts in other formats such as e-mail messages or viewing alerts using a web interface.

Name	Description
Packet Decoder	Prepares packets for processing.
Preprocessors or Input Plugins	Used to normalize protocol headers, detect anomalies, packet re-assembly and TCP stream re-assembly.
Detection Engine	Applies rules to packets.
Logging and Alerting System	Generates alert and log messages.
Output Modules	Process alerts and logs and generate final output.

Table 2 Snort's modules summarize.

A.2 Setting up of our Snort sensor

Depending upon the type of switches used, you can use Snort on a switch port. Some switches, allow you to replicate all ports traffic on one port where you can attach the Snort machine. These ports are usually referred to as spanning ports. The best place to install Snort is right behind the firewall or router so that all of the Internet traffic is visible to Snort before it enters any switch or hub.

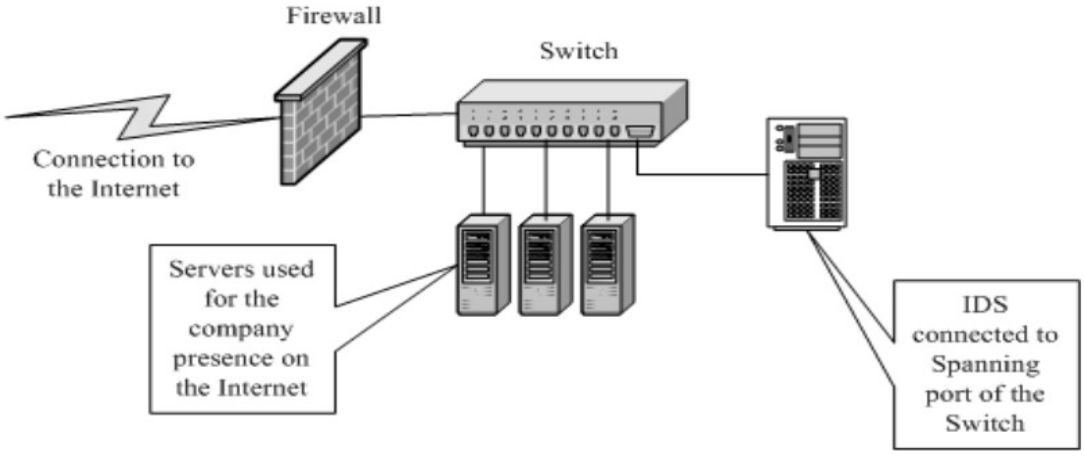


Fig 23 IDS behind the firewall

You can also connect the IDS to a small HUB or a Network TAP right behind the firewall, i.e., between firewall and the switch.

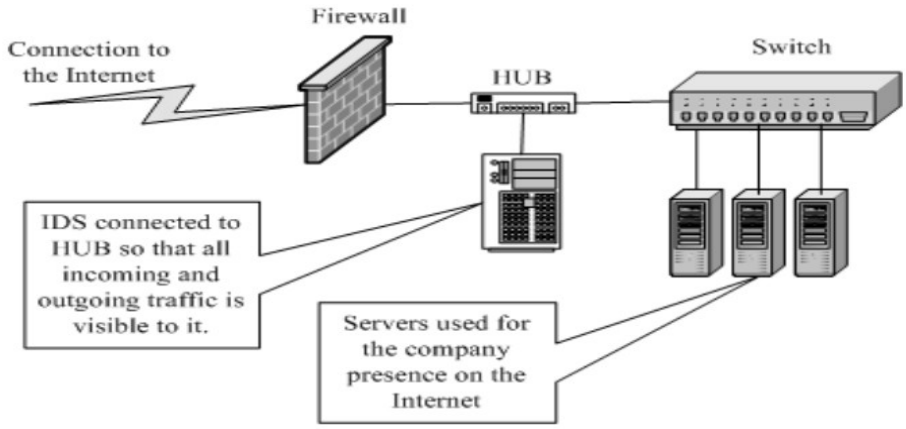


Fig.24 Likely scenario for a Snort sensor

Note that when the IDS is connected as shown in this last figure, data flowing among the company servers is not visible to the IDS. The IDS can see only that data which is coming from or going to the Internet. This is useful if you expect attacks from outside and the internal network is a trusted one.

Supported Platforms :

Snort is supported on a number of hardware platforms and operating systems. Currently Snort is available for the following operating systems:

- Linux
- OpenBSD
- FreeBSD
- NetBSD
- Solaris (both Sparc and i386)
- HP-UX
- AIX
- IRIX

• MacOS
• Windows

For a current list of supported platforms, refer to the Snort home page at http:// www.Snort.org.

How to Protect IDS Itself:

One major issue is how to protect the system on which your Intrusion Detection software is running. If security of the IDS is compromised, you may start getting false alarms or no alarms at all. The intruder may disable IDS before actually performing any attack. There are different ways to protect your system, starting from very general recommendations to some sophisticated methods. Some of these are mentioned below.

• The first thing that you can do is not to run any service on your IDS sensor itself. Network servers are the most common method of exploiting a system.
• New threats are discovered and patches are released by vendors. This is almost a continuous and non-stop process. The platform on which you are running IDS should be patched with the latest releases from your vendor. For example, if Snort is running on a Microsoft Windows machine, you should have all the latest security patches from Microsoft installed.
• Configure the IDS machine so that it does not respond to ping (ICMP Echo- type) packets.
• If you are running Snort on a Linux machine, use netfilter/iptable to block any unwanted data. Snort will still be able to see all of the data.
• You should use IDS only for the purpose of intrusion detection. It should not be used for other activities and user accounts should not be created except those that are absolutely necessary.

Following are two special techniques that can be used with Snort to protect it from being attacked:

- Snort on Stealth Interface .
- Snort with no IP Address Interface .

The advantage is that when the Snort host doesn’t have an IP address itself, nobody can access

it. You can configure an IP address on eth1 that can be used to access the sensor itself.

A.3 Installing of Snort and Getting Started

A simple Snort installation consists of a single Snort sensor run from terminal or from system start up as a daemon. To install Snort for this purpose, you can get a pre-compiled version or compile it yourself from the source code: http://www.Snort.org

Putting the sensor behind a router or firewall will enable you to detect the activity of intruders into the system. However, if you are really interested in scanning all Internet traffic, you can put the sensor outside the firewall as well.

Single Sensor with Database and Web Interface : the most common use of Snort should be with integration to a database. The data-base is used to log Snort data where it can be viewed and analyzed later on, using a web-based interface. A typical setup of this type consists of three basic components:

1. Snort sensor
2. A database server
3. A web server

Snort logs data into the database. You can view the data using a web browser connected to the sensor.

Different types of database servers like MySQL, PostgresSQL, Oracle, Microsoft SQL server and other ODBC-compliant databases can be used with Snort. This setup provides a very good and omprehensive IDS which is easy to manage and user friendly.

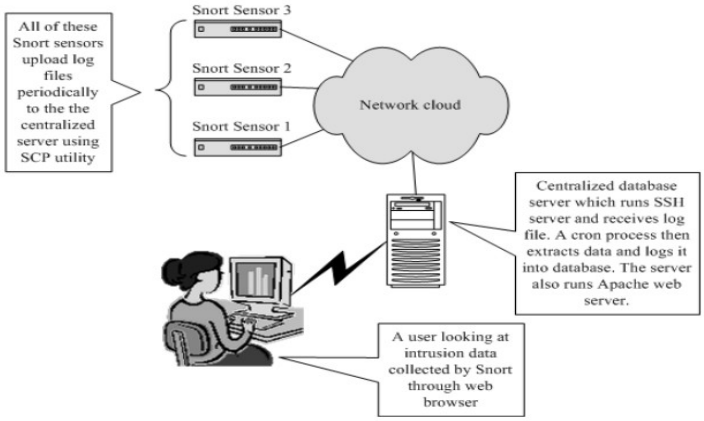


Fig.25 Our company's system administrator.

Installation:

- Snort's installation from sources:

* some development tools & libraries we need:

- flex
- bison
- checkinstall
- libpcap0.8
- libnet1.0

```
apt-get install flex bison build-essential checkinstall libpcap0.8-dev libnet1-dev
```

* we must download the next components for our installation:

- libpcap1.3 (updated)
- daq0.2
- pcre8.32
- libdnet1.12 (updated)
- zlib1.2.7
- Snort2.9.4

```
cd libpcap1.3
      ./configure && make && checkinstall
      dpkg -i libpcap0.8*.deb

cd daq-0.2
      ./configure && make && checkinstall
      dpkg -i daq_0.2-1_i386.deb

cd pcre-8.32
      ./configure && make && checkinstall
      dpkg -i pcre*.deb
```

```
cd libdnet-1.12
      ./configure && make && checkinstall
      dpkg -i libdnet*.deb

cd zlib-1.2.7
      ./configure && make && checkinstall
      dpkg -i zlib*.deb
```

And now we are already able to install our Snort sensor:

```
cd Snort-2.9.4
      ./configure && make && checkinstall
      dpkg -i Snort_*.deb
```

Available command line options with the configure script can be listed using the "./configure –help" ,

"./configure –prefix=/opt/Snort --enable-smbalerts --enable-flexresp --with-mysql --with-snmp --with-openssl " would an example of how to enable build in support for mysql database or snmp.

Automatic Startup and Shutdown

You can configure Snort to start at boot time automatically and stop when the system shuts down. On UNIX-type machines, this can be done through a script that starts and stops Snort. The script is usually created in the /etc/init.d directory on Linux. A link to the startup script may be created in /etc/rc3.d directory and shutdown links may be present in /etc/rc2.d, /etc/rc1.d and /etc/rc0.d directories.

Snort Command Line Options

Snort has many command line options that are very useful for starting Snort in different

situations. As you have already seen, command line options are helpful in running multiple versions of Snort on the same system. You can use “Snort -?” command to display command line options.

Snort Modes

Snort operates in two basic modes: packet sniffer mode and NIDS mode. It can be used as a packet sniffer, like tcpdump or snoop. When sniffing packets, Snort can also log these packets to a log file. The file can be viewed later on using Snort or tcpdump. No Intrusion Detection activity is done by Snort in this mode of operation. Using Snort for this purpose is not very useful as there are many other tools available for packet logging. For example, all Linux distributions come with the tcpdump program which is very efficient. When you use Snort in network Intrusion Detection (NIDS) mode, it uses its rules to find out if there is any network Intrusion Detection activity.

Logging Snort Data in Text Format

You can log Snort data in text mode by adding -l <directory name> on the command line. The following command logs all Snort data in /var/log/Snort directory in addition to displaying it on the console: *Snort -dev -l /var/log/Snort*

Logging Snort in Binary Format

On high-speed networks, logging data in ASCII format in many different files may cause high overhead. Snort allows you to log all data in a binary file in tcpdump format and view it later on. In this case, Snort logs all data to a single file in raw binary form. A typical command for this type of log is :

Snort -l /tmp -b Snort will create a file in /tmp directory.

To view this raw binary data, you can use Snort. The -r command line switch is used to specify a file name with Snort. The following command will display the captured data from file Snort.log.1037

Snort *-dev -r /tmp/Snort.log.1037 | more*

The output of this command will show data in exactly the same way if you are looking at it on

the console in real time. You can use different switches to display different levels of detail with this data.

You can also display a particular type of data from the log file. The following command displays all TCP type data from the log file: *Snort -dev -r / tmp/Snort.log.1037840339 tcp* Similarly, ICMP and UDP types of data can also be displayed.

You can also use the tcpdump program to read files generated by Snort when logging in this mode. The following command reads the Snort files and displays captured packets in the file:

tcpdump -r /tmp/Snort.log.1037

Network Intrusion Detection Mode

In Intrusion Detection mode, Snort does not log each captured packet as it does in the network sniffer mode. Instead, it applies rules on all captured packets. If a packet matches a rule, only then is it logged or an alert is generated. If a packet does not match any rule, the packet is dropped silently and no log entry is created. When you use Snort in Intrusion Detection mode, typically you provide a configuration file on the command line: *Snort -c /opt/Snort/etc/Snort.conf*

This configuration file contains Snort rules or reference to other files that contain Snort rules. In addition to rules, the configuration file also contains information about input and output plug-ins. The typical name of the Snort configuration file is Snort.conf.

Other command line options and switches can be used when Snort is working in IDS mode. For example, you can log data into files as well as display data on the command line. The following command will log data to /var/log/Snort directory and will display it on the console screen in addition to acting as NIDS: *Snort -dev -l /var/log/Snort -c /etc/Snort/Snort.conf* However in most real-life situations, you will use -D command line switch with Snort so that it does not log on the console but runs as a daemon. In a typical scenario, you will also want to log Snort data into a database. Logging data into MySQL database could be an example.

Snort Alert Modes

When Snort is running in the Network Intrusion Detection (NID) mode, it generates alerts when a captured packet matches a rule. Snort can send alerts in many modes. These modes are configurable

through the command line as well as through Snort.conf file.

Whenever an alert is fired off, Snort captures the packet that fired off the rule and creates an alert. The amount of information logged with the alert depends on the particular alerting mode.

Fast Mode

The fast alert mode logs the alert with following information:

• Timestamp
• Alert message (configurable through rules)
• Source and destination IP addresses
• Source and destination ports

To configure fast alert mode, you have to use “-A fast” command line option. This alert mode causes less overhead for the system.

Full Mode

This is the default alert mode. It prints the alert message in addition to the packet header. We configure the full alert mode with “-A full”.

Other modes: UNIX Socket Mode , No Alert Mode , Sending Alerts to Syslog , Sending Alerts to SNMP , Sending Alerts to Windows

A.4 Working with Snort rules

Like viruses, most intruder activity has some sort of signature. These signatures may be present in the header parts of a packet or in the payload. Snort’s detection system is based on rules. These rules in turn are based on intruder signatures. Snort rules can be used to check various parts of a data packet .

Most of the rules are written in a single line. However you can also extend rules to multiple lines by using a backslash character at the end of lines. Rules are usually placed in a configuration file,

typically Snort.conf. You can also use multiple files by including them in a main configuration file.

This point provides information about different types of rules as well as the basic structure of a rule. I'll expose many examples of common rules for Intrusion Detection activity and together with the next points we should have enough information to set up Snort as a basic Intrusion Detection System .

Snort rules operate on network (IP) layer and transport (TCP/UDP) layer protocols. However there are methods to detect anomalies in data link layer and application layer protocols.

The Firsts Bad Rule

Here is the first (very) bad rule. In fact, this may be the worst rule ever written, but it does a very good job of testing if Snort is working well and is able to generate alerts.

alert ip any any -> any any (msg: "IP Packet detected";)

You can use this rule at the end of the Snort.conf file the first time you install Snort. The rule will generate an alert message for every captured IP packet. It will soon fill up your disk space if you leave it there! This should be your first test to make sure that Snort is installed properly .

alert icmp any any -> any any (msg: "ICMP Packet found";)

It generates alerts for all captured ICMP packets.

Structure of a Rule

Now that you have seen some rules which are not-so-good but helpful in a way, let us see the structure of a Snort rule. All Snort rules have two logical parts: rule header and rule options.

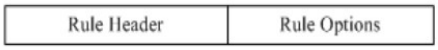


Fig. 26 General structure of a rule.

The rule header contains information about what action a rule takes. It also contains criteria for matching a rule against data packets. The options part usually contains an alert message and information about which part of the packet should be used to generate the alert message.

General structure of a Snort rule header:

Fig.27 General structure of a rule's header.

example: *alert icmp any any -> any any (msg: "Ping with TTL=100"; ttl: 100;)*

The part of the rule before the starting parenthesis is called the rule header. The part of the rule that is enclosed by the parentheses is the options part.

Rule Headers :

** Rule Actions:*

An action is taken only when all of the conditions mentioned in a rule are true. There are five predefined actions (however, you can also define your own actions as needed):

-Pass : this action tells Snort to ignore the packet. This action plays an important role in speeding up Snort operation in cases where you don’t want to apply checks on certain packets.

- Alert : used to send an alert message when rule conditions are true for a particular packet. An alert can be sent in multiple ways. For example, you can send an alert to a file or to a console. The functional difference between Log and Alert actions is that Alert actions send an alert message and then log the packet. The Log action only logs the packet.

- Activate : used to create an alert and then to activate another rule for checking more conditions. Dynamic rules, as explained next, are used for this purpose. The activate action is used when you need further testing of a captured packet.

- Dynamic : dynamic action rules are invoked by other rules using the “activate” action. In normal circumstances, they are not applied on a packet. A dynamic rule can be activated only by an “activate” action defined in another role.

- User Defined Actions : you can define your own actions. These new action types are defined in the configuration file Snort.conf. A new action is defined in the following general structure:

ruletype action_name
{
action definition
}

For example, an action named smb_db_alert that is used to send SMB pop-up window alert messages to hosts listed in workstation.list file and to MySQL database named “Snort” is defined below:

ruletype smb_db_alert
{
type alert
output alert_smb: workstation.list
*output database: log, mysql, user=rr password=rr *
dbname=Snort host=localhost
}

Theses types of rules will be discussed latter on in detail. Usually they are related to configuration of output plug-ins.

** Protocols:*

The second part of a Snort rule. The protocol part of a Snort rule shows on which type of packet the rule will be applied. Currently Snort understands the following protocols:

• IP
• ICMP
• TCP
• UDP

If the protocol is IP, Snort checks the link layer header to determine the packet type. If any other type of protocol is used, Snort uses the IP header to determine the protocol type. The options part instead can check parameters in other protocol fields as well.

** Address*

There are two address parts in a Snort rule. These addresses are used to check the source from which the packet originated and the destination of the packet. The address may be a single IP address or a network address. You can use the “any” keyword to apply a rule on all addresses. Or we can express an address followed by a slash character and number of bits in the netmask. For example:

- 192.168.2.0/24 represents C class network
- 192.168.2.0 with 24 bits in the network mask.

alert tcp any any -> 192.168.1.10/32 80 (msg: "TTL=100"; ttl: 100;)

Snort provides a mechanism to exclude addresses by the use of the negation symbol !, an exclamation point.

alert icmp ![192.168.2.0/24] any -> any any (msg: "Ping with TTL=100"; ttl: 100;)

This rule is useful, for instance, when you want to test packets that don’t originate from your home network (which means you trust everyone in your home network!).

You can also specify list of addresses in a Snort rule.

alert icmp ![192.168.2.0/24,192.168.8.0/24] any -> any any (msg: "Ping with TTL=100"; ttl: 100;)

** Port Number*

The port number is used to apply a rule on packets that originate from or go to a particular port or a range of ports. For example, you can use source port number 23 to apply a rule to those packets that originate from a Telnet server.

You can use the keyword any to apply the rule on all packets irrespective of the port number. Port number is meaningful only for TCP and UDP protocols. If you have selected IP or ICMP as the protocol in the rule, port number does not play any role.

alert tcp 192.168.2.0/24 23 -> any any (content: "confidential"; msg: "Detected confidential";)

The same rule can be applied to traffic either going to or originating from any Telnet server in the network by modifying the direction to either side as shown below:

alert tcp 192.168.2.0/24 23 <> any any (content: "confidential"; msg: "Detected confidential";)

Port numbers are useful when you want to apply a rule only for a particular type of data packet. For example, if a vulnerability is related to only a HTTP (Hyper Text Transfer Protocol) web server, you can use port 80 in the rule to detect anybody trying to exploit it. This way Snort will apply that rule only to web server traffic and not to any other TCP packets. Writing good rules always improves the performance of IDS.

- Port Ranges : *alert udp any 1024:2048 -> any any (msg: “UDP ports”;)*
- Upper and Lower Boundaries : for example, a range specified as :1024 includes all port numbers up to and including port 1024. A port range specified as 1000: will include all ports numbers including and above port 1000.
- Negation Symbol : *log udp any !53 -> any any log udp*

You can’t use comma character in the port filed to specify multiple ports. For example, specifying 53,54 is not allowed. However you can use 53:54 to specify a port range.

* *Direction:* determines the source and destination addresses and port numbers in a rule. The following rules apply to the direction field:

• A ->
• A <-

• A <> symbol shows that the rule will be applied to packets traveling on either direction. This symbol is useful when you want to monitor data packets for both client and server.

Rule options:

Rule options follow the rule header and are enclosed inside a pair of parentheses. There may be one option or many and the options are separated with a semicolon. If you use multiple options, these options form a logical AND. The action in the rule header is invoked only when all criteria in the options are true.

In general, an option may have two parts: a keyword and an argument.

msg: "Detected confidential";

In this option msg is the keyword and “Detected confidential” is the argument to this keyword.

**** The ack Keyword :*** the TCP header contains an Acknowledgement Number field which is 32 bits long. The field shows the next sequence number the sender of the TCP packet is expecting to receive. This field is significant only when the ACK flag in the TCP header is set.

Tools like nmap use this feature of the TCP header to ping a machine. For example, among other techniques used by nmap, it can send a TCP packet to port 80 with ACK flag set and sequence number 0. Since this packet is not acceptable by the receiving side according to TCP rules, it sends back a RST packet. When nmap receives this RST packet, it learns that the host is alive. This method works on hosts that don’t respond to ICMP ECHO REQUEST ping packets. To detect this type of TCP ping, you can have a rule like the following that sends an alert message:

alert tcp any any -> 192.168.1.0/24 any (flags: A; ack: 0; msg: "TCP ping detected";)

This rule shows that an alert message will be generated when you receive a TCP packet with the A flag set and the acknowledgement contains a value of 0. Generally when the A flag is set, the ACK value is not zero.

**** The classtype Keyword :*** rules can be assigned classifications and priority numbers to group and distinguish them. To fully understand the classtype keyword, first look at the file classification.config which is included in the Snort.conf file using the include keyword. Each line in the classification.config file has the following syntax:

config classification: name,description,priority
for example: *config classification: DoS,Denial of Service Attack,2*

To fully understand the classtype keyword, first look at the file classification.config which is included in the Snort.conf file using the include keyword.

Now let us use this classification in a rule. The following rule uses default priority with the classification DoS:

alert udp any any -> 192.168.1.0/24 6838 (msg:"DoS"; content: "server"; classtype:DoS;)

The following is the same rule but we override the default priority used for the classification.

alert udp any any -> 192.168.1.0/24 6838 (msg:"DoS"; content: "server"; classtype:DoS; priority:1)

Using classifications and priorities for rules and alerts, you can distinguish between high- and low-risk alerts. This feature is very useful when you want to escalate high-risk alerts or want to pay attention to them first.

Classifications are used in ACID , if you look at the ACID browser window, you will see the classification screens.

**** The content Keyword :*** One important feature of Snort is its ability to find a data pattern inside a packet. The pattern may be presented in the form of an ASCII string or as binary data in the

form of hexadecimal characters. Like viruses, intruders also have signatures and the content keyword is used to find these signatures in the packet.

The following rule detects a pattern “GET” in the data part of all TCP packets that are leaving 192.168.1.0 network and going to an address that is not part of that network.

alert tcp 192.168.1.0/24 any -> ![192.168.1.0/24] any (content: "GET"; msg: "GET matched";)

alert tcp 192.168.1.0/24 any -> ![192.168.1.0/24] any (content: "|47 45 54|"; msg: "GET matched";)

Hexadecimal number 47 is equal to ASCII character G, 45 is equal to E, and 54 is equal to T.

**** The offset Keyword :*** the offset keyword is used in combination with the content keyword. Using this keyword, you can start your search at a certain offset from the start of the data part of the packet. Use a number as argument to this keyword. The following rule starts searching for the word "HTTP" after 4 bytes from the start of the data.

alert tcp 192.168.1.0/24 any -> any any (content: "HTTP"; offset: 4; msg: "HTTP matched";)

**** The depth Keyword :*** The depth keyword is also used in combination with the content keyword to specify an upper limit to the pattern matching. Using the depth keyword, you can specify an offset from the start of the data part. Data after that offset is not searched for pattern matching. If you use both offset and depth keywords with the content keyword, you can specify the range of data within which pattern matching should be done.

The following rule tries to find the word "HTTP" between characters 4 and 40 of the data part of the TCP packet.

alert tcp 192.168.1.0/24 any -> any any (content:
"HTTP"; offset: 4; depth: 40; msg: "HTTP matched";)

This keyword is very important since you can use it to limit searching inside the packet. For example, information about HTTP GET requests is found in the start of the packet. There is no need to search the entire packet for such strings.

**** The content-list Keyword :*** the content-list keyword is used with a file name. The file name, which is used as an argument to this keyword, is a text file that contains a list of strings to be searched

inside a packet. Each string is located on a separate line of the file.

For example, a file named “porn” may contain the following three lines:

“porn”

“hardcore”

“under 18”

The following rule will search these strings in the data portion of all packets matching the rule criteria.:

alert ip any any -> 192.168.1.0/24 any (content-list: "porn"; msg: "Porn word matched";)

You can also use the negation sign ! with the file name if you want to generate an alert for a packet where no strings match.

**** The dsize Keyword:*** the dsize keyword is used to find the length of the data part of a packet. Many attacks use buffer overflow vulnerabilities by sending large size packets. Using this keyword, you can find out if a packet contains data of a length larger than, smaller than, or equal to a certain number.

The following rule generates an alert if the data size of an IP packet is larger than 6000 bytes:

alert ip any any -> 192.168.1.0/24 any (dsize: > 6000; msg: "Large size IP packet detected";)

**** The flags Keyword :*** The flags keyword is used to find out which flag bits are set inside the TCP header of a packet.

Flag	Argument character used in Snort rules
FIN or Finish Flag	F
SYN or Sync Flag	S
RST or Reset Flag	R
PSH or Push Flag	P
ACK or Acknowledge Flag	A
URG or Urgent Flag	U
Reserved Bit 1	1
Reserved Bit 2	2
No Flag set	0

Table 3 Flags' keywords

You can also use !, +, and * symbols just like IP header flag bits ! symbol is used for NOT, + is used for AND, and * is used for OR operation.

alert tcp any any -> 192.168.1.0/24 any (flags: SF; msg: "SYNC-FIN packet detected";)

**** The fragbits Keyword :*** The IP header contains three flag bits that are used for fragmentation and reassembly of IP packets.

- DF: Don't Fragment Bit
- MF: More Fragments Bit

Sometimes these bits are used by hackers for attacks and to find out information related to your network. For example, the DF bit can be used to find the minimum and maximum MTU for a path from source to destination. Using the fragbits keyword, you can find out if a packet contains these bits set or cleared.

The following rule is used to detect if the DF bit is set in an ICMP packet:

alert icmp any any -> 192.168.1.0/24 any (fragbits: D; msg: "Don't Fragment bit set";)

In this rule, D is used for DF bit. You can use R for reserved bit and M for MF bit.

You can also use the negation symbol ! in the rule. The following rule detects if the DF bit is not set, although this rule is of little use.

alert icmp any any -> 192.168.1.0/24 any (fragbits: !D; msg: "Don't Fragment bit not set";)

Value	Type of ICMP Packet
0	Echo reply
3	Destination unreachable
4	Source quench
5	Redirect
8	Echo request
11	Time exceed
12	Parameter problem
13	Timestamp request
14	Timestamp reply
15	Information request
16	Information reply

Table 4 Type of ICMP packet

**** The icmp_id Keyword :*** The icmp_id option is used to detect a particular ID used with ICMP packet.

Read texts related to ICMP header for further information.

For example: *alert icmp any any -> any any (icmp_id: 100; msg: "ICMP ID=100";)*

**** The icmp_seq Keyword***

for example: *alert icmp any any -> any any (icmp_seq:100; msg: "ICMP Sequence=100";)*

**** The itype Keyword :*** The ICMP header comes after the IP header and contains a type field.

for example:

alert icmp any any -> any any (itype: 4; msg: "ICMP Source Quench Message received";)

alert icmp any any -> any any (itype: 4; msg: "ICMP Source Quench Message received";)

**** The icode Keyword :*** In ICMP packets, the ICMP header comes after the IP header. It contains a code field . The type field in the ICMP header shows the type of ICMP message.

• If code field is 0, it is a network redirect ICMP packet.

• If code field is 1, it is a host redirect packet.

• If code is 2, the redirect is due to the type of service and network.

• If code is 2, the redirect is due to type of service and host.

The icode keyword in Snort rule options is used to find the code field value in the ICMP header. The following rule generates an alert for host redirect ICMP packets.

alert icmp any any -> any any (itype: 5; icode: 1; msg: "ICMP ID=100";)

Both itype and icode keywords are used. Using the icode keyword alone will not do the job because other ICMP types may also use the same code value.

**** The id Keyword :*** The id keyword is used to match the fragment ID field of the IP packet header. Its purpose is to detect attacks that use a fixed ID number in the IP header of a packet. Its format is as follows: id: "id_number"

If the value of the id field in the IP packet header is zero, it shows that this is the last fragment of an IP packet (if the packet was fragmented). The value 0 also shows that it is the only fragment if the

packet was not fragmented. The id keyword in the

Snort rule can be used to determine the last fragment in an IP packet.

**** The ipopts Keyword :*** A basic IPv4 header is 20 bytes long. You can add options to this IP header at the end. The length of the options part may be up to 40 bytes. These options can be used by some hackers to find information about your network.

Using Snort rules, you can detect such attempts with the ipopts keyword. The fol lowing rule detects any attempt made using Loose Source Routing:

alert ip any any -> any any (ipopts: lsrr; msg: "Loose source routing attempt";)

**** The ip_proto Keyword :*** The ip_proto Keyword The ip_proto keyword uses IP Proto plug-in to determine protocol number in the IP header.

alert ip any any -> any any (ip_proto: ipip; msg: "IP-IP tunneling detected";)

For further information about protocol numbers, consult the /etc/protocols file in your linux system.

**** The logto Keyword :*** The logto keyword is used to log packets to a special file.

The general syntax is as follows:

logto:logto_log

Consider the following rule:

alert icmp any any -> any any (logto:logto_log; ttl: 100;)

This rule will log all ICMP packets having TTL value equal to 100 to file logto_log, a file that later you can open with your “more”, “cat” or any other tool used to display a file into console.

**** The msg Keyword :*** The msg keyword in the rule options is used to add a text string to logs and alerts.

You can add a message inside double quotations after this keyword. The msg keyword is a common and useful keyword and is part of most of the rules.

The general form for using this keyword is as follows:

msg: "Your message text here";

If you want to use some special character inside the message, you can escape them by a backslash character.

**** The nocase Keyword :*** The nocase keyword is used in combination with the content keyword. It has no arguments. Its only purpose is to make a case insensitive search of a pattern within the data part of a packet .

**** The priority Keyword :*** The priority keyword assigns a priority to a rule. Priority is a number argument to this keyword. Number 1 is the highest priority. The keyword is often used with the classtype keyword.

alert ip any any -> any any (ipopts: lsrr; msg: "Loose source routing attempt"; priority: 10;)

The priority keyword can be used to differentiate high priority and low priority alerts.

**** The react Keyword :*** The react keyword is used with a rule to terminate a session to block some sites or services. Not all options with this keyword are operational.

The following rule will block all HTTP connections originating from your home network 192.168.1.0/24. To block the HTTP access, it will send a TCP FIN and/or FIN packet to both sending and receiving hosts every time it detects a packet that matches these criteria.

The rule causes a connection to be closed:

alert tcp 192.168.1.0/24 any -> any 80 (msg: "Outgoing HTTP connection"; react: block;)

In the above rule, “block” is the basic modifier. You can also use the “warn” modifier to send a visual notice to the source. You can also use the additional modifier “msg ” which will include the msg string in the visual notification on the browser. The following is an example of this additional modifier.

alert tcp 192.168.1.0/24 any -> any 80 (msg: "Outgoing HTTP connection”; react: warn, msg;)

Note: *In order to use the react keyword, you should compile Snort with --enable- flexresp command line option in the configure script. For a discussion of the compilation process. The react should be the last keyword in the options field.*

**** The reference Keyword :***

**** The resp Keyword:*** The resp keyword is a very important keyword. It can be used to knock down hacker activity by sending response packets to the host that originates a packet matching the rule. The keyword is also known as Flexible Response or simply FlexResp and is based on the FlexResp plug-in. The plug-in should be compiled into Snort using the command line option (--with-flexresp) in the configure script.

The following rule will send a TCP Reset packet to the sender whenever an attempt to reach TCP port 8080 on the local network is made.

alert tcp any any -> 192.168.1.0/24 8080 (resp: rst_snd;)

You can send multiple response packets to either sender or receiver by specifying multiple responses to the resp keyword. The arguments are separated by a comma. The list of arguments that can be used with this keyword is found in the following table.

Argument	Description
rst_snd	Sends a TCP Reset packet to the sender of the packet
rst_rcv	Sends a TCP Reset packet to the receiver of the packet
rst_all	Sends a TCP Reset packet to both sender and receiver
icmp_net	Sends an ICMP Network Unreachable packet to sender
icmp_host	Sends an ICMP Host Unreachable packet to sender
icmp_port	Sends an ICMP Port Unreachable packet to sender
icmp_all	Sends all of the above mentioned packets to sender

Table 5 List of arguments

**** The rev Keyword :***

**** The rpc Keyword :***

**** The sameip Keyword :*** The sameip keyword is used to check if source and destination IP addresses are the same in an IP packet. It has no arguments. Some people try to spoof IP packets to get information or attack a server. The following rule can be used to detect these attempts:

alert ip any any -> 192.168.1.0/24 any (msg: "Same IP"; sameip;)

**** The seq Keyword :*** The seq keyword in Snort rule options can be used to test the sequence number of a TCP packet. The argument to this keyword is a sequence number. The general format is as follows:

seq: "sequence_number";

Sequence numbers are a part of the TCP header.

**** The flow Keyword :*** The flow keyword is used to apply a rule on TCP sessions to packets flowing in a particular direction. You can use options with the keyword to determine direction. The following options can be used with this keyword determine direction:

- to_client
- to_server
- from_client
- from_server

Other options are also available which are used to apply the rule to different states of a TCP connection.

- The *stateless* option is used to apply the rule without considering the state of a TCP session.
- The *established* option is used to apply the rule to established TCP sessions only.
- The *no_stream* option enables rules to be applied to packets that are not built from a stream.

• The *stream_only* option is used to apply the rules to only those packets that are built from a stream.

**** The session Keyword :*** The session keyword can be used to dump all data from a TCP session. It can dump all session data or just printable characters. The following rule dumps all printable data from POP3 sessions:

log tcp any any -> 192.168.1.0/24 110 (session: printable;)

If you use “all” as argument to this keyword, everything will be dumped. Use the logto keyword to log the traffic to a particular file.

**** The sid Keyword :*** The sid keyword is used to add a “Snort ID” to rules. Output modules or log scanners can use SID to identify rules.

**** The tag Keyword :*** The tag keyword is another very important keyword that can be used for logging additional data from/to the intruder host when a rule is triggered. The additional data can then be analyzed later on for detailed intruder activity. The general syntax of the keyword is as follows:

tag: <type>, <count>, <metric>[, direction]

The following rule logs 100 packets on the session after it is triggered:

*alert tcp 192.168.2.0/24 23 -> any any (content: "boota"; msg: "Detected boota"; *

tag: session, 100, packets;)

Argument	Description
Type	You can use either “session” or “host” as the type argument. Using session, packets are logged from the particular session that triggered the rule. Using host, all packets from the host are logged.
Count	This indicates either the number of packets logged or the number of seconds during which packets will be logged. The distinction between the two is made by the metric argument.
Metric	You can use either “packets” or “seconds” as mentioned above.
Direction	This argument is optional.You can use either “src” to log packets from source or “dst” to log packets from the destination.

Table 6 Tag's arguments

**** The tos Keyword :*** The tos keyword is used to detect a specific value in the Type of Service (TOS) field of the IP header. The format for using this keyword is as follows:

tos: 1;

**** The ttl Keyword :*** The ttl keyword is used to detect Time to Live value in the IP header of the packet. The keyword has a value which should be an exact match to determine the TTL value. This keyword can be used with all types of protocols built on the IP protocol, including ICMP, UDP and TCP. The general format of the keyword is as follows:

ttl: 100;

Note: The traceroute utility uses TTL values to find the next hop in the path. The traceroute sends UDP packets with increasing TTL values. The TTL value is decremented at every hop. When it reaches zero, the router generates an ICMP packet to the source.

Using this ICMP packet, the utility finds the IP address of the router. For example, to find the fifth hop router, the traceroute utility will send UDP packets with TTL value set to 5. When the packet reaches the router at the fifth hop, its value becomes zero and an ICMP packet is generated.

Using the ttl keyword, you can find out if someone is trying to traceroute through your network. The only problem is that the keyword needs an exact match of the TTL value.

**** The uricontent Keyword :*** The uricontent keyword is similar to the content keyword except that it is used to look for a string only in the URI part of a packet.

A.5 The Snort Configuration File

Snort uses a configuration file at startup time. A sample configuration file Snort.conf is included in the Snort distribution. You can use any name for the configuration file, however Snort.conf is the conventional name. You use the -c command line switch to specify the name of the configuration file. The following command uses /opt/Snort/Snort.conf as the configuration file.

/opt/Snort/Snort -c /opt/Snort/Snort.conf

Snort.conf contains six basic sections:

• Variable definitions.
• Config parameters.
• Preprocessor configuration.
• Output module configuration.
• Defining new action types.
• Rules configuration and include files.

Using Variables in Rules

you can define a variable HOME_NET in the configuration file: *var HOME_NET 192.168.1.0/24*

Later on you can use this variable HOME_NET in your rules:

*alert ip any any -> $HOME_NET any (ipopts: lsrr; *
msg: "Loose source routing attempt"; sid: 1000001;)

As you can see, using variables makes it very convenient to adapt the configuration file and rules to any environment. For example, you don't need to modify all rules when you copy rules from one network to another.

Using a List of Networks in Variables

var HOME_NET [192.168.1.0/24,192.168.10.0/24]

Using Interface Names in Variables

var HOME_NET $eth0_ADDRESS
var EXTERNAL_NET $eth1_ADDRESS

Using the any Keyword

The any keyword can also be a variable: *var EXTERNAL_NET any*

There are many variables defined in the Snort.conf file that come with the Snort distribution. While installing Snort, you need to modify these variables according to your network.

The config Directives (¡!)

The config directives in the Snort.conf file allow a user to configure many general settings for Snort. Examples include the location of log files, the order of applying rules and so on.

Preprocessor Configuration

Preprocessors or input plug-ins operate on received packets before Snort rules are applied to them. The preprocessor configuration is the second major part of the configuration file. Detailed information about each preprocessor is found in manuals.

The general format of configuring a preprocessor is as follows:
preprocessor <preprocessor_name>[: <configuration_options>]

The following is an example of a line in the configuration file for IP defragmentation preprocessor frag2.

preprocessor frag2

Output Module Configuration

Output modules, also called output plug-ins, manipulate output from Snort rules. For example, if you want to log information to a database or send SNMP traps, you need output modules. The following is the general format for specifying an output module in the configuration file.

output <output_module_name>[: <configuration_options>]

For example, if you want to store log messages to a MySQL database, you can configure an output module that contains the database name, database server address, user name and password.

*output database: alert, mysql, user=rr password=boota *
dbname=Snort host=localhost

There may be additional steps to make the output module work properly. In the case of MySQL database, you need to setup a database, create tables, create user, set permissions and so on.

Defining New Action Types

You already know that the first part of each Snort rule is the action item. Snort has predefined action types; however, you can also define your own action types in the configuration file. A new action type may use multiple output modules.

The following action type creates alert messages that are logged into the database as well as in a file in

the tcpdump format.

ruletype dump_database
{
type alert
*output database: alert, mysql, user=rr dbname=Snort *

host=localhost

output log_tcpdump: tcpdump_log_file

}

This new action type can be used in rules just like other action types.

dump_database icmp any any -> 192.168.1.0/24 any (fragbits: D; msg: "Don't Fragment bit set";)

When a packet matches the criteria in this rule, the alert will be logged to the database as well as to the tcpdump_log_file.

Rules Configuration

The rules configuration is usually the last part of the configuration file. You can create as many rules as you like using variables already defined in the configuration file. The rules configuration is the place in the configuration file where you can put your rules. However the convention is to put all Snort rules in different text files. You can include these text files in the Snort.conf file using the "include" keyword. Snort comes with many predefined rule files. The names of these rule files end with *.rule.*

All files in the Snort distribution whose name ends with .rules contain rules and they are included in the Snort.conf file. These rule files are included in the main Snort.conf file using the "include" keyword. The following is an example of including myrules.rules file in the main configuration file.

include myrules.rules

Note: It is not necessary that the name of the rules file must end with .rule. You can use a name of your choice for your rule file.

A.6 Plugins, Preprocessors and Output Modules

Preprocessors

When a packet is received by Snort, it may not be ready for processing by the main Snort detection engine and application of Snort rules. For example, a packet may be fragmented. Before you can search a string within the packet or determine its exact size, you need to defragment it by assembling all fragments of the data packet. The job of a preprocessor is to make a packet suitable for the detection engine to apply different rules to it.

Configuration parameters for different preprocessors are present in the *Snort.conf* file. Using the file, you can enable or disable different preprocessors.

All enabled preprocessors operate on each packet. There is no way to bypass some of the preprocessors based upon some criteria. If you have enabled a large number of preprocessors, you may slow down Snort detection process. Therefore you should be careful when enabling preprocessors.

The general format of enabling a preprocessor is as follows:
preprocessor <name of preprocessor>[: parameters]
Brief description of different preprocessors:

HTTP Decode : The Hyper Text Transfer Protocol (HTTP) allows Intrusion Detection Systems to use hexadecimal characters in URI to defeat known attacks. For example, this can be done by inserting something like %3A%2F%2F in the URI to replace :// characters. A large number of attacks on web servers are carried by obfuscating URI characters using hexadecimal numbers in the URI. The HTTP decode blocks any such attempts by converting them to the actual URI.

Port Scanning : The first step in any intruder activity is usually to find out what services are running on a network. Once an intruder has found this information, attacks for known vulnerabilities for these services are tried. The portscan preprocessor is designed to detect port scanning activities.

You can also use another preprocessor in conjunction with this preprocessor. This preprocessor is portscan-ignorehosts, which can be used to ignore some hosts if any port scanning activity is detected from them.

The frag2 Module : With frag2, you can configure timeout and memory limits for packet defragmenta-

tion. By default, the preprocessor uses 4 MB of memory and a 60-second timeout period. If a packet assembly is not successful within this time period, previously collected fragments are discarded.

The stream4 Module : It provides two basic functions:

1. TCP stream reassembly
2. Stateful inspection

You must configure two preprocessors in the Snort.conf file for Stream4 to work properly. These modules are “stream4” and “stream4_reassemble.” Both of these take a number of arguments. If you don’t specify an argument, a default value is used instead.

ARP Spoofing : Address Resolution Protocol (ARP) is used to find a MAC address when an IP address is known.

Note: ARP is needed when a host wants to send an IP packet to another host on the local network. The sending host broadcasts an ARP packet on the network asking, “Who has this IP address?” The host who has that IP address will respond with its MAC address. After that, the sending host will send the data packet (usually called a frame at the link layer level) to the destination host.

The arpspoof preprocessor detects anomalies in ARP packets.

Output Modules

Output modules are used to control the output from Snort detection engine. By default, the

output from alerts and logs go into files in the /var/log/Snort directory. Using output modules, you can process output and send output messages a number of other destinations.

Output modules can be defined in the Snort configuration file and some of them can also be configured on the command line as well. The general format for defining the output module inside the configuration file is as follows: *output <module_name>[: arguments]*

For example, if you want to log messages to MySQL database called “Snort” using database user name “rr” and password “rr” located on the same machine where Snort is running, you use the following line in Snort.conf file.

*output database: log, mysql, user=rr password=rr *
dbname=Snort host=localhost

However when you use an output module in the configuration file, alerts will not go into the alert file. Once you place this line in the Snort.conf file, all alerts will go into the MySQL database. There are ways to send alerts to multiple destinations.

Sometimes you may want to send alerts to multiple locations. Defining your own action using the ruletype keyword is a good idea.

Sometimes you may want to send alerts to multiple locations. Defining your own action using the ruletype keyword is a good idea. For example, the following lines in the Snort.conf file will define an action type called “smb_db_alert” that will cause alerts to be sent to both the database and SMB pop-up windows for rules that use this action type.

ruletype smb_db_alert
{
type alert
output alert_smb: workstation.list
*output database: log, mysql, user=rr password=rr *
dbname=Snort host=localhost

}

The following rule uses this new action type. Alerts generated by this rule will go to MySQL database as well as to the Windows machine in the form of pop-up windows.

smb_db_alert icmp any any -> 192.168.1.0/24 any (fragbits: D; msg: "Dont Fragment bit set";)

You can also use command line options with some output modules. For example, you can use -s option to log alerts to Syslog.

* The alert_syslog Output Module : The alert_syslog module allows you to send alerts to the syslog facility.

* The alert_full Output Module : The alert_full module logs full alert messages in a file. The following line will log all alert messages to alert_detailed file under the Snort logging directory.

output alert_full: alert_detailed

However, enabling full alerts consumes a significant amount of time to log data into a file, causing some packets to be ignored by the detection engine.

* The alert_fast Output Module : Like alert_full, alert_fast also takes as an argument a file name for storing data. It is fast compared to full alerting. Packet headers are not saved in the alert file. The fol-

lowing line in the Snort.conf file enables one-line alert messages to be stored in alert_quick file.

output alert_fast: alert_quick

* The alert_smb Module : SMB alerts are sent to Microsoft Windows-based workstations using the smb client program which is part of the SAMBA client package on Linux machines. To send these alerts, the smbclient must be present in the PATH variable.

* The log tcpdump Output Module : This module is used to store alert data in a tcpdump format file that can be viewed later on using tcpdump or some other tool. This method is quick for heavily loaded networks where you want to offload processing from the Snort system and analyze data using some other mechanism. Following is the general format for using this module in Snort.conf file.

output log_tcpdump: <filename>

Typical entries in the Snort.conf file may look like the following:

output log_tcpdump: /var/log/Snort/Snort_tcpdump.log

Each time you start Snort, a new file is created.

Now you can display the contents of this file (the captured data) using the tcpdump command as follows:

tcpdump -v -r /var/log/Snort/ Snort_tcpdump.log.1039971287 since the file created is in rcpdump format.

* Logging to Databases : Databases are used with Snort to store log and alert data. Logging data to files in the disk is fine for smaller applications. However, keeping log data in disk files is not appropriate when you have multiple Snort sensors or you want to keep historical data as well. Databases also allow you to analyze data generated by Snort sensors.

For example, if you want to find the top 15 alerts that are generated most frequently, you can use SQL statements for the database. Finding the same information from log files is difficult. Similarly, if you want to find the most active attackers in the month of November 2002, it is very easy to find out that information from a database.

You can use multiple types of databases with Snort including Oracle and MySQL.

*output database: log, mysql, user=rr password=rr *

dbname=Snort host=localhost

To enable support of databases, you need to compile Snort with database support enabled. The following configure script enables MySQL database support in Snort.

./configure --prefix=/opt/Snort --with-mysql=/usr/lib/mysql

There are some other output modules, but are beyond the scope of this project:

* CSV Output Module .
* Unified Logging Output Module .
* SNMP Traps Output Module .
*Log Null Output Module .

A.7 Using Snort with MySQL

All systems need some type of efficient logging feature, usually using a database at the backend. Snort can be made to work with MySQL or Oracle for example.You already know from the discussion of output modules in the previous point that you can save logs and alerts to a database.

Logging to a database is very useful for maintaining history data, generating reports and analyzing information. By using other tools like Analysis Control for Intrusion Detection (ACID), discussed in next, you can get very useful information from the database about attack patterns. For example, you can get a report about the last fifteen unique attacks, information about hosts that are continuously attacking your network, the distribution of attacks by different protocols, and so on.

Since MySQL is a freely available database and works perfectly well on Linux and other operating systems, this is a natural choice for Snort.

There exists different scenarios when dealing with Snort and databases, but when you are running only one sensor and don’t have any pre-existing database server, it is a natural choice to install the database on the Snort machine itself.

Before you start logging to MySQL database, you have to create a database on the database server for Snort. After creating the database, you have to create tables where Snort data is logged. However, you don’t need to create tables manually because Snort comes with a script that will do the entire job for you. To work with MySQL, you may have to recompile Snort with MySQL support.

After going through this point, you should be able to install Snort and MySQL so that all of the Snort activity is logged to the database.

A.8 Using ACID with Snort

Analysis Console for Intrusion Databases (ACID) is a tool used to analyze and present Snort data using a web interface. It is written in PHP. It works with Snort and databases like MySQL.

ACID consists of many Pretty Home Page (PHP) scripts and configuration files that work together to collect and analyze information from a database and present it through a web interface. A user will use a web browser to interact with ACID. You have to have a web server, database server, PHP and some other tools installed on your system to make it work.

Annexe B: Virtual scenario for Modbus software.

I'm going to be using virtualbox for this.

B.1 Introduction Modbus IP

References:

1. “Modbus for Field Technicians” by Peter Chipkin.
2. programming the Modbus: http://pes.free.fr/libModbus.html
3. Modbus and Snort: pag. 131 in Snort_manual.pdf
4. coils and registers, slave (server) client (master) http://www.control.com/thread/1230731691
5. Modbus_protocol.pdf
6. Modbus_wiki.pdf
7. Snort-intrusion-detection-Modbus-tcp-ip-communications.pdf

B.2 Modbus IP , simulated Master, simulated Slave, Snort with fixed Modbus ruleset

VIRTUAL MACHINES:

Installing the virtual machines:

2 x Xubuntu/Openbox Modbus master & slave
1 x Xubuntu : Snort sensor.

Linux- Snort_Sensor:

Name: LinuxXubuntu-Snort sensor
OS Type: Ubuntu
Base Memory: 512 Mb
Start-up Disk: LinuxXubuntu - Snort sensor.vdi (Normal, 4.00 GB)
Network: Adapter 1: Intel PRO/1000 MT Desktop (Bridged adapter, wlan0)

Modbus – Master:

Name: Modbus - Master
OS Type: Ubuntu
Base Memory: 512 MB
Start-up Disk: Modbus - Master.vdi (Normal, 4.00 GB)
Network: Adapter 1: Intel PRO/1000 MT Desktop (Bridged adapter, wlan0)

Modbus – Slave:

Name: Modbus - Slave
OS Type: Ubuntu
Base Memory: 512 MB
Start-up Disk: Modbus - Slave.vdi (Normal, 4.00 GB)
Network: Adapter 1: Intel PRO/1000 MT Desktop (Bridged adapter, wlan0)

→ note: less than 4.00 GB for storage is not possible, the installation doesn't run.

- VirtualBox: how to install our Debian

How to create a virtual machine from an .iso file
http://www.pentest.ro/install-a-clean-debian-on-virtualbox/

IMPORTANT: for further reboots, pay attention to the configuration of the boot order...
- place the “hard disk” as the first option, otherwise you'll get the installation routine every time you reboot the virtual machine.

CREATING OUR VIRTUAL NETWORK:

References:
- http://www.virtualbox.org/manual/ch06.html
- https://blogs.oracle.com/fatbloke/entry/networking_in_virtualbox1
- Virtualbox: Virtual networking by Ravikiran Dighade
  http://www.csee.umbc.edu/~kalpakis/Courses/621/project/VirtualBox-VirtualNetworking.pdf

XUBUNTU && OPENBOX:

- Xubuntu-12.10
From .iso, we create a virtual machine in which we install our Snort following the instructions in the previously signaled website.

- remove unnecessary: games, chat-irc, open-office... through “Ubuntu Sofware Center”, it will make faster our system.

- By now, we leave any development tool, but when setting off our Snort in a real environment, remember to strip off any compiler, unnecessary libraries or other useful stuff to a posible intruder.

- Alternatively you can install openbox, one of the most lightweight window managers available:

```
sudo apt-get install openbox openbox-themes obconf obmenu
```

It will add an openbox session to the login menu.

```
sudo apt-get remove xfce*
```

It will supress the graphical environment for the xfce just leaving the options of Xubuntu graphical interface and Openbox when rebooting our VM, from which we choose Openbox.

With this new environment (Openbox) our machine works faster.

The environment is so simple that we don't even have a task bar with the windows we have in use, for that “Alt + tab” will move you from one window to the other.

- Snort's installation from sources:

* some development tools & libraries we need:

- flex
- bison
- checkinstall
- libpcap0.8
- libnet1.0

apt-get install flex bison build-essential checkinstall libpcap0.8-dev libnet1-dev

* we must download the next components for our installation:

- libpcap1.3 (updated)
- daq0.2
- pcre8.32
- libdnet1.12 (updated)
- zlib1.2.7
- Snort2.9.4

```
cd libpcap1.3
        ./configure && make && checkinstall
        dpkg -i libpcap0.8*.deb


cd daq-0.2
        ./configure && make && checkinstall
        dpkg -i daq_0.2-1_i386.deb

cd pcre-8.32
        ./configure && make && checkinstall
        dpkg -i pcre*.deb


cd libdnet-1.12
        ./configure && make && checkinstall
        dpkg -i libdnet*.deb

cd zlib-1.2.7
        ./configure && make && checkinstall
        dpkg -i zlib*.deb
```

And now we are already able to install our Snort sensor:

```
cd Snort-2.9.4
        ./configure && make && checkinstall
        dpkg -i Snort_*.deb
```

ERROR when running for the first time Snort:

"Snort: error while loading shared libraries: libdnet.1: cannot open shared object file: no such file"
Meaning that Snort does not find libdnet location.

Solution:
LD_LIBRARY_PATH=/usr/local/lib
export LD_LIBRARY_PATH
But this is a tiresome solution, since you have to add this any time you start off Snort...

Must find any "fix" solution...
http://www.linuxquestions.org/questions/linux-newbie-8/Snort-error-while-loading-shared-libraries-libdnet-1-cannot-open-shared-object-fil-901530/
Using a manually installed "libdnet-1.11" (Installed to /usr/local/) :
cd /usr/lib/
sudo ln -s /usr/local/lib/libdnet.1.0.1 libdnet.1

This solution adds a soft link to the libraries directory, possibiliting Snort to find the library that he missed.

Modbus:

www.Modbusdriver.com/modpoll.html → master simulator
www.Modbusdriver.com/diagslave.html → slave simulator

- download the program for slave and master: diagslave, modpoll.
- Enter into the linux folder where we can find the binary.
  - */Downloads/xxxxx/linux
  - The binary contained in has no execution rights: we must give them to it.
    - chmod u+x diagslave && ./diagslave
    - chmod u+x modpoll && ./modpoll

- In our slave:
  ifconfig → to get the IP address xxx.xxx.xxx.xxx

./diagslave -m tcp -a 1

- put our server waiting for requests in Modbus tcp.

- In our master:

./modpoll -m tcp -a 1 -r 100 -c 5 -l /dev/ttyS0 xxx.xxx.xxx.xxx

- where

./modpoll is our program's name
/dev/ttyS0 is our communication port in linux (in windows it would be COM1, COM2 …)
192.160.1.60 is our Server's address (slave's address)

The process:

1) we put our server to listen (slave)
2) send the request from our master (client)
3) Snort must be listening our network in order to capture the trafic Snort -dev (in mode verbose)

Bibliography:

Books:

- [SteRi01] Stevens, W.Richard " TCP/IP Illustrated, Volume 1 The Protocols " Addison-Wesley Professional Computing Series
  Publication Date: December 31, 1993 | ISBN-10: 0201633469 | ISBN-13: 978-0201633467

- [SteRi02] Stevens, W.Richard "UNIX Network Programming: Networking APIs: Sockets and XTI; Volume 1"

  ISBN-10: 013490012X | ISBN-13: 978-0134900124

- [Hck01] Cache, Jonnhy and Liu, Vincent "Hacking Exposed Wireless: Wireless Security Secrets & Solutions" McGraw-Hill Osborne Media
  Publication Date: March 26, 2007 | ISBN-10: 0072262583

- [CmRe01]Schildt, Herbert " C++: The Complete Reference " McGraw-Hill Osborne Media
  Publication Date: August 1, 1998 | ISBN-10: 0078824761 | ISBN-13: 978-0078824760

- [ProgC01]Kelley, Al and Pohl, Ira " A Book on C: Programming in C" Addison-Wesley Professional
  Publication Date: January 8, 1998 | ISBN-10: 0201183994 | ISBN-13: 978-0201183993

- [BeJS01]Beale, Jay and R.Baker, Andrew "Snort 2.1 Intrusion Detection" Syngress
  Publication Date: May 2004 | ISBN-10: 1931836043 | ISBN-13: 978-1931836043

- [Snus00]The Snort Project May 23, 2012 "Snort Users Manual 2.9.3 "
  Open source: http://www.Snort.org/assets/166/Snort_manual.pdf

- [NeSt00]Matthew, Neil and Stones, Richard "Beginning Linux Programming " Wrox
  Publication Date: November 5, 2007 | ISBN-10: 0470147628 | ISBN-13: 978-0470147627

- [InPe01]Perens, Bruce "Intrusion Detection Systems with Snort Advanced IDS Techniques Using Snort, Apache, MySQL, PHP, and ACID "
  Open source:
  http://ptgmedia.pearsoncmg.com/imprint_downloads/informit/perens/0131407333.pdf

- [AlWe0]Allen Weiss, Mark "Data Structures and Algorithm Analysis in C" Addison-Wesley
  Publication Date: September 19, 1996 | ISBN-10: 0201498405 | ISBN-13: 978-0201498400

- [MiLaw0]H.Miller, Lawrence and E.Quilici, Alexander "The Joy of C" Wiley
  Publication Date: January 30, 1997 | ISBN-10: 047112933X | ISBN-13: 978-0471129332

Open source references (pdf):

- [AcroM0] Acromag Technical Reference – Modbus TCP/IP INTRODUCTION TO Modbus TCP/IP
  http://www.dee.hcmut.edu.vn/vn/ptn/sch/download/Network_Architecture/intro_ModbusTCP.pdf

- [ezTCP] Technical Document Modbus/TCP of ezTCP Version 1.3
  http://www.eztcp.com/documents/application/an_Modbus_tcp_en.pdf

- [MoSe00] Modbus over Serial Line Specification and Implementation Guide V1.02
  http://www.Modbus.org/docs/Modbus_over_serial_line_V1_02.pdf

Web:

- About network security:
  - [1] at wikipedia: http://en.wikipedia.org/wiki/Network_security (2013,May 5th)
  - [2] at webopedia: http://www.webopedia.com/TERM/N/network_security.html (2013,May 5th)
  - [3] at Magazine Encyclopedia: http://www.pcmag.com/encyclopedia/term/47911/network-security (2013,May 5th)

- About pcap:
  - [4] http://yuba.stanford.edu/~casado/pcap/section1.html
  - [5] http://code.google.com/p/pcapsctpspliter/issues/detail?id=6
  - [6] http://www.tcpdump.org/pcap.htm

- Pcap samples:
  - [7] http://wiki.wireshark.org/SampleCaptures
  - [8] http://www.pcapr.net/home

- About Snort:
  - [9] http://www.Snort.org/
  - [10] http://manual.Snort.org/node1.html
  - [11] http://oreilly.com/pub/h/1393
  - [12] http://insidetrust.blogspot.ie/2010/12/how-to-use-Snort-on-backtrack-4-basic.html
  - [13] http://www.aboutdebian.com/Snort.htm
  - [14] http://bailey.st/blog/2010/10/06/compiling-Snort-2-9-0/

- About linux and C programming:

  - [15] http://www.freeos.com/guides/lsst/
  - [16] http://linuxcommand.org/writing_shell_scripts.php
  - [17] http://www.cprogramming.com/
  - [18] http://www.tenouk.com/Module40c.html
  - [19] http://www.thegeekstuff.com/2011/12/c-socket-programming/
  - [20] http://www.gnu.org/software/libc/manual/html_node/Getopt.html
  - [21] http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.15.html

- About Modbus and other Industrial Protocols or networking in general:

  - [22] http://en.wikipedia.org/wiki/Industrial_Ethernet
  - [23] http://en.wikipedia.org/wiki/Modbus
  - [24] http://www.Modbus.org/
  - [25] http://www.rtaautomation.com/Modbustcp/
  - [26] http://compnetworking.about.com/od/networkprotocols/g/protocols.htm
  - [27] http://en.wikipedia.org/wiki/Transmission_Control_Protocol

- About cybersecurity in Industrial Control and SCADA systems.

  - [28] http://www.technologyreview.com/view/511671/cybersecurity-risk-high-in-industrial-control-systems/
  - [29] http://en.wikipedia.org/wiki/Control_system_security
  - [30] http://www.tofinosecurity.com/
  - [31] http://www.huffingtonpost.com/2013/05/16/anonymous-telecomix-syria-internet-blackout_n_3279626.html?utm_hp_ref=technology
  - [32] http://www.bbc.co.uk/news/technology-22594140
  - [33] http://www.infosecurity-magazine.com/view/31793/icscert-reports-two-hacks-on-building-management-systems/
  - [34] https://www.cert.be/pro/attacks-scada-systems
  - [35] http://www.electricenergyonline.com/?page=show_article&article=181
  - [36] http://threatpost.com/attacks-scada-ics-honeypots-modified-critical-operations-031913/
  - [37] http://www.prweb.com/releases/2013/3/prweb10580258.htm

Revisión de la Bibliografía

Libros :

- [SteRi01] Stevens, W.Richard " TCP/IP Illustrated, Volume 1 The Protocols " Addison-Wesley Professional Computing Series , Publication Date: December 31, 1993 | ISBN-10: 0201633469 | ISBN-13: 978-0201633467

- [SteRi02] Stevens, W.Richard "UNIX Network Programming: Networking APIs: Sockets and XTI; Volume 1" , ISBN-10: 013490012X | ISBN-13: 978-0134900124

- [Hck01] Cache, Jonnhy and Liu, Vincent "Hacking Exposed Wireless: Wireless Security Secrets & Solutions" McGraw-Hill Osborne Media , Publication Date: March 26, 2007 | ISBN-10: 0072262583

- [CmRe01]Schildt, Herbert " C++: The Complete Reference " McGraw-Hill Osborne Media Publication Date: August 1, 1998 | ISBN-10: 0078824761 | ISBN-13: 978-0078824760

- [ProgC01]Kelley, Al and Pohl, Ira " A Book on C: Programming in C" Addison-Wesley Professional , Publication Date: January 8, 1998 | ISBN-10: 0201183994 | ISBN-13: 978-0201183993

- [BeJS01]Beale, Jay and R.Baker, Andrew "Snort 2.1 Intrusion Detection" Syngress Publication Date: May 2004 | ISBN-10: 1931836043 | ISBN-13: 978-1931836043

- [Snus00]The Snort Project May 23, 2012 "Snort Users Manual 2.9.3 " ,
  Open source: http://www.Snort.org/assets/166/Snort_manual.pdf

- [NeSt00]Matthew, Neil and Stones, Richard "Beginning Linux Programming " Wrox Publication Date: November 5, 2007 | ISBN-10: 0470147628 | ISBN-13: 978-0470147627

- [LinSP00]Love, Robert "Linux System Programming" O'Reilly
  Publication Date: September 2007 | ISBN-10: 0-596-00958-5 | ISBN-13: 978-0-596-00958-8

- [InPe01]Perens, Bruce "Intrusion Detection Systems with Snort Advanced IDS Techniques Using Snort, Apache, MySQL, PHP, and ACID "
  Open source:
  http://ptgmedia.pearsoncmg.com/imprint_downloads/informit/perens/0131407333.pdf

- [AlWe0]Allen Weiss, Mark "Data Structures and Algorithm Analysis in C" Addison-Wesley Publication Date: September 19, 1996 | ISBN-10: 0201498405 | ISBN-13: 978-0201498400

- [MiLaw0]H.Miller, Lawrence and E.Quilici, Alexander "The Joy of C" Wiley
  Publication Date: January 30, 1997 | ISBN-10: 047112933X | ISBN-13: 978-0471129332

Open source references (pdf):

- [AcroM0] Acromag Technical Reference – Modbus TCP/IP INTRODUCTION TO Modbus TCP/IP
  http://www.dee.hcmut.edu.vn/vn/ptn/sch/download/Network_Architecture/intro_ModbusTCP.pdf

- [ezTCP] Technical Document Modbus/TCP of ezTCP Version 1.3
  http://www.eztcp.com/documents/application/an_Modbus_tcp_en.pdf

- [MoSe00] Modbus over Serial Line Specification and Implementation Guide V1.02
  http://www.Modbus.org/docs/Modbus_over_serial_line_V1_02.pdf

Web:

- About network security:
  - [1] at wikipedia: http://en.wikipedia.org/wiki/Network_security (2013,May 5th)
  - [2] at webopedia: http://www.webopedia.com/TERM/N/network_security.html (2013,May 5th)
  - [3] at Magazine Encyclopedia: http://www.pcmag.com/encyclopedia/term/47911/network-security.html (2013,May 5th)

- About pcap:
  - [4] http://yuba.stanford.edu/~casado/pcap/section1.html
  - [5] http://code.google.com/p/pcapsctpspliter/issues/detail?id=6
  - [6] http://www.tcpdump.org/pcap.htm

- Pcap samples:
  - [7] http://wiki.wireshark.org/SampleCaptures
  - [8] http://www.pcapr.net/home

- About Snort:
  - [9] http://www.Snort.org/
  - [10] http://manual.Snort.org/node1.html
  - [11] http://oreilly.com/pub/h/1393
  - [12] http://insidetrust.blogspot.ie/2010/12/how-to-use-Snort-on-backtrack-4-basic.html
  - [13] http://www.aboutdebian.com/Snort.htm
  - [14] http://bailey.st/blog/2010/10/06/compiling-Snort-2-9-0/

- About linux and C programming:

  - [15] http://www.freeos.com/guides/lsst/
  - [16] http://linuxcommand.org/writing_shell_scripts.php
  - [17] http://www.cprogramming.com/
  - [18] http://www.tenouk.com/Module40c.html
  - [19] http://www.thegeekstuff.com/2011/12/c-socket-programming/
  - [20] http://www.gnu.org/software/libc/manual/html_node/Getopt.html
  - [21] http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.15.html

- About Modbus and other Industrial Protocols or networking in general :

  - [22] http://en.wikipedia.org/wiki/Industrial_Ethernet
  - [23] http://en.wikipedia.org/wiki/Modbus
  - [24] http://www.Modbus.org/
  - [25] http://www.rtaautomation.com/Modbustcp/
  - [26] http://compnetworking.about.com/od/networkprotocols/g/protocols.htm
  - [27] http://en.wikipedia.org/wiki/Transmission_Control_Protocol

- About cybersecurity in Industrial Control and SCADA systems.

  - [28]http://www.technologyreview.com/view/511671/cybersecurity-risk-high-in-industrial-control-systems/
  - [29] http://en.wikipedia.org/wiki/Control_system_security
  - [30] http://www.tofinosecurity.com/
  - [31]http://www.huffingtonpost.com/2013/05/16/anonymous-telecomix-syria-internet-blackout_n_3279626.html?utm_hp_ref=technology
  - [32] http://www.bbc.co.uk/news/technology-22594140
  - [33]http://www.rtve.es/alacarta/videos/informe-semanal/informe-semanal-espionaje-masivo/1875087
  - [34]http://www.infosecurity-magazine.com/view/31793/icscert-reports-two-hacks-on-building-management-systems/
  - [35] https://www.cert.be/pro/attacks-scada-systems
  - [36] http://www.electricenergyonline.com/?page=show_article&article=181
  - [37] http://threatpost.com/attacks-scada-ics-honeypots-modified-critical-operations-031913/
  - [38] http://www.prweb.com/releases/2013/3/prweb10580258.htm
  - [39] http://esmateria.com/2013/06/04/la-ciberguerra-es-inevitable/
  - [40] http://www.bbc.co.uk/news/technology-22524274)
  - [41] http://www.datacenterdynamics.es/focus/archive/2012/01/los-ataques-se-incrementa%C3%A1n-sobre-los-sistemas-scada-en-2012
  - [42] http://www.eset.es/soporte/315

- About Linux:
  - [43]http://en.wikipedia.org/wiki/Linux
  - [44]http://distrowatch.com/
  - [45]http://slashdot.org/
  - [46]http://openbox.org/
  - [47]http://en.wikipedia.org/wiki/Openbox
  - [48]http://www.debian.org/
  - [49]http://en.wikipedia.org/wiki/Debian
  - [50]http://en.wikipedia.org/wiki/Ubuntu_%28operating_system%29
  - [51]http://xubuntu.org/
    - [52]http://xubuntu.org/about/