

CURSO DE TCP/IP: 4ª ENTREGA

TCP (TRANSMISION CONTROL PROTOCOL)

2ª PARTE

Este mes nos metemos de lleno en los paquetes de Internet. Crearemos un paquete desde cero, nos adentraremos en el código binario y para rematar comprenderemos de una vez por todas el significado de algunos ataques ya conocidos por todos.

1. Fundamentos de la comunicación digital.

Cuando empecé con el curso de TCP/IP, cuya cuarta entrega tenéis ahora mismo en vuestras manos, ya os advertí de que, cansado de ver una y otra vez las mismas técnicas para explicar los conceptos en todos los libros y tutoriales, este curso pretendía ser una apuesta arriesgada, orientando la explicación de los mismos conceptos desde un punto de vista bastante diferente.

Siguiendo en esta línea un tanto experimental, voy a dar otro nuevo paso que no he visto en ningún libro sobre TCP/IP, para tratar de que os familiaricéis más aún con TCP/IP.

Lo que pretendo conseguir ahora es que convirtáis esos misteriosos paquetes TCP que tenéis rondando ahora mismo por vuestras cabezas, en algo tangible, de carne y hueso, o más bien debería decir de unos y ceros. 😊

Para ello, vamos a ver con un ejemplo cómo está construido exactamente un paquete TCP. Mi intención es que después de este ejemplo, los paquetes TCP dejen de ser al fin para vosotros unos entes teóricos de los cuales conocéis el funcionamiento, pero no su constitución "física".

Por supuesto, esta constitución física no podremos terminar de comprenderla hasta que lleguemos a la capa inferior de la jerarquía de capas de TCP/IP: el nivel físico. Así que habrá que esperar aún unos cuantos meses antes de que veamos qué son exactamente

esos ceros y unos de los que vamos a hablar ahora.

De momento, nos quedaremos con una simplificación de la idea de qué es exactamente un cero y un uno. Como sabemos, los datos que circulan en una red lo hacen siempre a través de un medio físico. Este medio, normalmente, será un cable eléctrico, pero puede ser también, por ejemplo, una onda de radiofrecuencia, un cable óptico, o cualquier otro medio físico empleado en la interconexión de máquinas. Quedémonos con el caso más común, el del cable eléctrico, aunque todo lo explicado se puede extrapolar, por supuesto, a cualquier otro medio físico.

Como es lógico, por un cable eléctrico circula electricidad. La electricidad por si misma no contiene mucha información. Un medio de transmisión de información será más versátil cuantos más parámetros posea.

Por ejemplo, una imagen puede transmitir una gran cantidad de información (ya se sabe que una imagen vale más que mil palabras), ya que posee muchísimos parámetros, como son los colores en cada punto, la iluminación, etc. En cambio, la electricidad posee pocos parámetros propios, como pueden ser la tensión eléctrica (voltaje), y la intensidad eléctrica (corriente). Para colmo, estos dos parámetros están directamente relacionados entre sí (¿recordáis la famosa ley de Ohm, o vosotros también suspendíais física en el cole? 😊)

Por tanto, una primera solución intuitiva para transmitir información por medio de la

electricidad sería hacer variar esos parámetros en función de lo que quisiéramos transmitir.

Por ejemplo, si queremos transmitir un número, podemos ajustar el voltaje del cable en relación directa con el valor de ese número. Por ejemplo, para transmitir un 5 pondríamos 5 voltios en el cable, y para transmitir un 7 pondríamos 7 voltios en el cable. Si, en cambio, queremos transmitir el número 250000, más nos vale no tocar el "cablecito", a no ser que queramos seguir este curso desde el más allá (allí también llega, desde que la revista cambió de distribuidor).

Supongo que las mentes más despiertas habrán descubierto ya aquí una de las más destructivas técnicas de hacking. Si algún día os encontráis por un chat al <malnacido> ese que os robó la novia, basta con que le enviéis un paquete que contenga un número tocho con ganas, como el 176874375618276543, y por su módem circulará tal tensión eléctrica, que en el lugar que ocupaba su casa veréis un cono atómico que ni el de Hiroshima.

Bueno, antes de que se me eche alguien encima por decir semejantes estupideces, tendré que reconocer que, como es lógico, las cosas no funcionan así en la práctica. ¡Pero no os creáis que sea algo tan descabellado! ¿Y si en lugar de una proporción 1/1 utilizamos otra clase de proporcionalidad para transmitir los números?

Por ejemplo, supongamos que no queremos superar los 10 voltios y, en cambio, queremos transmitir números entre 1 y 1000. Pues basta con que establezcamos el convenio de que 10 voltios equivalen al número 1000 y, por tanto, 5 voltios al 500, 2'5 voltios al 250, etc., etc.

Esta solución no sólo es mucho más realista, si no que incluso ha sido el sistema de transmisión de información en el que se ha basado toda la electrónica hasta que llegó la revolución digital. Y el nombre de esto os sonará mucho, ya que a esto es a lo que se llama comunicación ANALÓGICA.

Siempre habréis escuchado el término analógico como opuesto al término digital. Vamos a ver ahora mismo en qué consisten las diferencias entre analógico y digital.

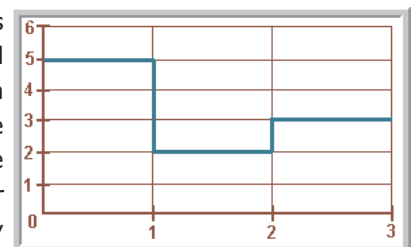
Si bien la tecnología analógica aprovecha la tensión eléctrica, uno de los parámetros que caracterizan la electricidad que circula por un cable, la tecnología digital no utiliza ninguno de los parámetros de la electricidad para transmitir la información.

¿Cómo puede transmitirse la información entonces? Evidentemente, siempre es imprescindible la presencia de una variable que se pueda modificar convenientemente para codificar la información que se desea transmitir. La diferencia es que en el caso de la transmisión digital el parámetro que se utiliza para portar la información no es inherente a la propia electricidad, si no que es un parámetro más sencillo: **el tiempo**.

Evidentemente, el tiempo es siempre un parámetro fundamental en toda comunicación, ya que es imprescindible que haya una sincronización temporal entre transmisor y receptor.

Volviendo al caso de la transmisión analógica, pensemos por ejemplo en el caso en el que se transmitan sólo números del 0 al 9 y, para conseguir representar números más altos lo que se hace es transmitir cada una de sus cifras por separado (unidades, decenas, centenas, etc.). Si, por ejemplo, quisiéramos transmitir el número 523, primero transmitiríamos 5 voltios, luego 2 voltios, y por último 3 voltios.

En la imagen podemos ver la transmisión del número 523 por una línea analógica. En el eje X (el horizontal) se representa el tiempo, por ejemplo, en segundos, y en el eje Y (el vertical) se representa la tensión en voltios.



Lógicamente, es necesario establecer un convenio entre el transmisor y el receptor para saber cuánto tiempo tiene que pasar entre la transmisión de cada cifra. Si no fuese así, imaginad lo que pasaría si tratásemos de transmitir el número 5551. Si la línea se mantiene en 5 voltios el tiempo necesario para transmitir las tres primeras cifras, ¿cómo podrá saber el receptor que en ese tiempo se han transmitido tres cincos, y no sólo uno, o doce?

Por tanto, ha de existir un convenio previo entre emisor y receptor que diga "cada segundo se transmitirá una nueva cifra". Por tanto, si pasado un segundo el voltaje no ha cambiado, significa que la siguiente cifra es igual a la anterior.

Lo que hace la tecnología digital es explotar al máximo este tipo de "convenios". Pero empezamos viendo el caso más simple de todos, que es igual al caso de la transmisión analógica.

Imaginemos que queremos transmitir números pero que sólo puedan ser 0 o 1. En ese caso, la cosa funcionaría igual: a cada segundo una nueva cifra, y no hay más misterio. La diferencia sería que la tensión sólo podría tener dos valores: 0 o 1 voltios.

La diferencia viene cuando queremos transmitir números más grandes, que es cuando hay que hacer "convenios raros".

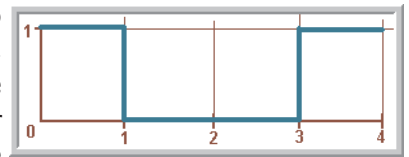
En realidad, estos convenios tienen poco de raro, al menos para un matemático, ya que no es ninguna invención, si no simplemente una aplicación directa de las matemáticas. Concretamente, lo que se aplica es la aritmética binaria.

Número decimal	Secuencia binaria
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Este "convenio" asigna una secuencia diferente para representar cada número decimal, que podemos ver en la tabla de la izquierda.

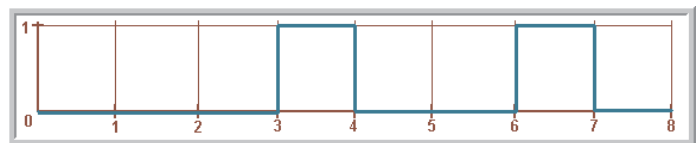
Por tanto, si queremos transmitir digitalmente el número 9, estos serán los voltajes que tendremos que poner en la línea:

Es decir, primero 1 voltio durante un segundo, luego 0 voltios durante dos segundos y, por último, 1 voltio durante el último segundo.



¿Cómo podemos entonces transmitir un número de dos cifras, como por ejemplo el 12?

Pues de nuevo hay que hacer otro convenio entre el emisor y el receptor y decir: "cada cifra decimal va a constar de 4 cifras binarias". Por tanto, si transmitimos la secuencia: "0001 0010" (ver la tabla anterior) estaremos transmitiendo el número 12, según el convenio preestablecido, tal y como vemos en la siguiente imagen.



De esta manera, a base de acuerdos entre el transmisor y el receptor, podemos transmitir cualquier información, con sólo transmitir ceros y unos.

¿Cuál es la ventaja de transmitir sólo dos valores en lugar de variar el voltaje en función del valor que se desea transmitir? Pues son varias las ventajas, pero la más obvia es la gran fiabilidad de la transmisión.

Imaginemos que nuestro cable lo hemos comprado en el *Todo a 100* y falla más que una escopeta de feria. Pretendemos transmitir 5 voltios y, en cambio, unas veces el cable transmite 4 voltios, otras veces 3,5, otras veces 2...

Estos fallos del cable serían críticos en una transmisión analógica, ya que el voltaje se traduce directamente en la información que transmitimos. En cambio, la transmisión digital

nos permite unos márgenes de error muy grandes. Al fin y al cabo, transmitir ceros y unos se limita tan sólo a diferenciar "no hay electricidad en el cable" de "sí hay electricidad en el cable".

Por tanto, podemos decir por ejemplo: "si hay menos de un voltio en el cable, consideramos que es un cero. Si hay más de un voltio, consideramos que es un uno". Así, todos esos fallos del cable de *Todo a 100* no afectarían a la transmisión digital, ya que incluso 2 voltios seguiría siendo considerado como "sí hay electricidad en el cable" y, por tanto, sería considerado como un 1.

Son muchas otras las ventajas de lo digital frente a lo analógico, como la mayor facilidad a la hora de almacenar y procesar la información, pero esas ventajas no se deducen de lo explicado hasta ahora, por lo que no entraremos ahora en más detalle. De momento, la idea con la que nos tenemos que quedar es con que a la hora de transmitir información, la transmisión digital tiene una mayor inmunidad al ruido y, en general, a cualquier error, que la transmisión analógica.

Por supuesto, nada de lo explicado hasta ahora es en la práctica tal y como lo he contado (para aquellos que sepan de qué va el tema y estén flipando), pero mi intención no es escribir un RFC sobre transmisión analógica vs. transmisión digital, si no tan sólo explicar conceptos, aunque tenga que ser faltando a la realidad en muchas ocasiones. Lo importante es que comprendáis los conceptos que subyacen a una transmisión de datos digitales, como es el caso de TCP/IP.

Para los más quisquillosos que sigan insistiendo en que las cosas no son así, y que realmente incluso las transmisiones digitales circulan de forma analógica, insisto en que no estoy tratando de explicar el nivel físico (cosa que ya haré dentro de unos cuantos artículos, y donde todo esto quedará finalmente aclarado), si no tan sólo abstraerme de los detalles para explicar los conceptos básicos. 😊



¿Y si el mundo...

¿Y si el mundo fuese distinto?

Muchas personas, cuando les dices que el mundo informático funciona con CEROS y UNOS, nunca llegan a entenderlo. Con lo sencillo que sería trabajar con el sistema decimal (0,1,2,3,4,5,6,7,8,9) e incluso con letras.

Acabamos de descubrir que aplicar el sistema decimal supondría 10 niveles de voltaje en un cable eléctrico, algo técnicamente posible pero MUY CARO. Como ya hemos dicho los cables deberían ser muy buenos y los dispositivos que detectasen los cambios de voltaje deberían ser muy precisos. Pero esto es hablando del mundo analógico... si nos vamos al digital la cosa se pone interesante.

Hemos dicho que en el MUNDO DIGITAL solo hay dos estados, es decir, "unos" (abierto, con electricidad, iluminado) y ceros (cerrado, sin electricidad, apagado). Pero ¿por qué? ¿por qué el hombre ha decidido que el mundo digital solo tenga dos estados en lugar de 10?

Muy sencillo, POR DINERO!!! ¿Cómo? ¿qué?... En el diminuto universo que conocemos, nuestro planeta, es muy sencillo (y barato) encontrar sustancias que sean capaces de tener dos estados claramente diferenciados, que puedan ser capaces de pasar de un estado a otro muy rápidamente, que lo hagan de forma barata y para colmo que sea muy fácil detectar esos estados.

Al igual que una bombilla puede estar encendida o apagada y todos podemos percibirlo mirándola (luz/oscuridad) o tocándola (calor/frío), en el caso de la informática EL DIOS ES EL SILICIO. Simplificando mucho el tema, podemos decir el silicio deja pasar la electricidad o no (ceros y unos), lo hace rapidísimamente (todos queremos tener un PC ultrarrápido) y detectar el estado ("cargado" o "descargado") es tecnológicamente sencillo y por lo tanto barato.

¿Y si el mundo fuese distinto?

Imagina otro elemento, por ejemplo el agua. Todos conocemos tres de sus estados, el sólido, el líquido y el gaseoso... si el agua se pareciese al silicio (si fuese sencillo pasar de un estado a otro, lo hiciese a la velocidad del rayo y fuese sencillo detectar esos cambios de estado)... nuestro ordenador estaría basado en un procesador de agua y, en ese caso... sería MUCHO más potente y rápido que los que ahora tenemos basados en silicio.

¿Qué? ¿Cómo? Sí, porque tendríamos un sistema de TRES estados (apagado -hielo-, neutro -líquido- y encendido -gaseoso-)

en lugar de DOS estados (apagado/encendido). Eso significa que en lugar del código BINARIO utilizaríamos el código TRINARIO (el nombre me lo acabo de inventar). Al haber tres estados en lugar de dos podríamos crear convenios mucho más optimizados, es decir, transmitir información de forma mucho más optimizada y por lo tanto mucho más rápida.

Deja volar tu imaginación... si un buen día alguien encuentra (o fabrica, o trae de otra galaxia) un material parecido al silicio pero que pudiese tener 600 estados en lugar de dos... buff... el mundo informático daría un salto astronómico en velocidad de cálculo. Para la humanidad sería comparable al paso de la edad de Piedra a la edad de Hierro.

Actualmente, a falta de materiales nuevos los científicos intentan utilizar elementos conocidos pero difíciles de "controlar". Unos basados en dos estados y otros en multiestados... por ejemplo los átomos.

Al final, la orgullosa humanidad depende de los materiales que su "humilde" entorno proporciona.

2.- Codificación binaria.

Llegados a este punto, posiblemente ya habréis perdido un poco de miedo a eso de los ceros y los unos. Lo que nos queda por ver es cómo se codifica realmente la información en binario (utilizando tan sólo ceros y unos), es decir, cuáles son los "convenios" reales de los que hemos hablado, que permiten que un transmisor y un receptor se puedan entender.

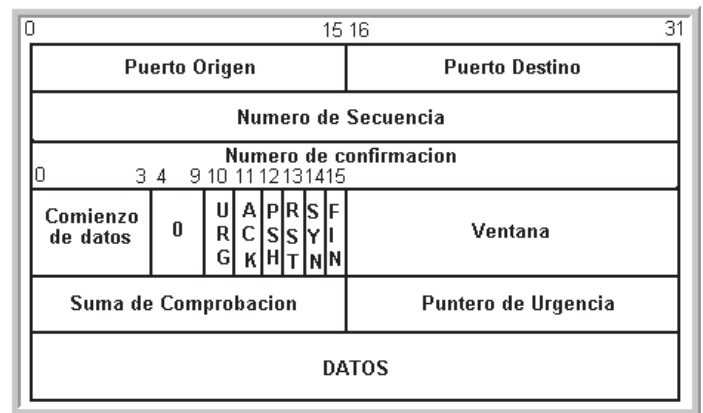
Es aquí donde entra el concepto fundamental de **palabra (word)**. La forma en que se representa la información depende del tamaño de la palabra utilizada. En el ejemplo anterior, donde representábamos cada número decimal con 4 cifras binarias, es decir, con 4 **bits** (bit es simplemente el nombre dado a una cifra binaria, es decir, un número que sólo puede ser un cero o un uno), teníamos una **palabra de 4 bits** para representar los números decimales.

La palabra más comúnmente utilizada en informática es el octeto que, en el caso de

los ordenadores personales, llamamos **byte**. Un byte es simplemente una palabra de 8 bits, es decir, de 8 cifras binarias (cero o uno). Si quisiéramos representar los números decimales con un byte, ésta podría ser la tabla correspondiente:

Número decimal	Secuencia binaria
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001

Vamos al fin a ver algo directamente relacionado con TCP. Volvamos al último artículo de la revista, y repasemos la cabecera TCP. Por si no lo tenéis a mano, aquí os la vuelvo a mostrar.



Como vemos, en la cabecera TCP se manejan distintos tamaños de palabra. En primer lugar, para los campos **puerto origen** y **puerto destino** contamos con 16 bits, después, para los **números de secuencia y de confirmación** tenemos 32 bits, 4 bits para el campo **comienzo de datos**, 1 bit para cada uno de los **flags**, etc.

Para saber cuántos valores se pueden representar con cada tamaño de palabra, basta con hacer la **potencia de 2 con el tamaño de la palabra en bits**. Es decir, con una

palabra de 4 bits podemos representar $2^4 = 16$ valores diferentes, con una palabra de 8 bits $2^8 = 256$ valores, con 16 bits $2^{16} = 65536$, etc.

Ahora podéis comprender por qué todo lo relacionado con la tecnología digital sigue siempre estos números. La memoria RAM que compras para el PC, las tarjetas de memoria para la cámara digital, la velocidad del ADSL, siempre son los mismos números; 8, 16, 32, 64, 128, 256, 512, 1024, 2048... Todos esos números son las diferentes potencias de 2.

El caso más sencillo es el de 1 único bit, donde tenemos $2^1 = 2$, es decir, se pueden representar sólo 2 valores con 1 bit. Estos 2 valores son, por supuesto: cero, y uno.

En el caso, por ejemplo, de 3 bits, tenemos $2^3 = 8$ valores diferentes, que son todas las posibles combinaciones de 3 cifras, donde cada cifra puede ser un uno o un cero, tal y como vemos en la tabla.

Número decimal	Secuencia binaria
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Como vemos, no quedan más posibles combinaciones de ceros y unos con sólo 3 cifras, y esto nos permite representar tan sólo los números del 0 al 7.

Como ya dije antes, para un matemático estos convenios para representar los números decimales mediante cifras binarias no son ningún misterio, ya que basta con aplicar las bases de la aritmética modular.

Voy a tratar de explicar rápidamente en qué consisten estas fórmulas porque, aunque al principio os puedan parecer complicadas, en realidad son realmente sencillas y, como todo, es sólo cuestión de práctica el aplicarlas de forma natural.

2.1. Pasando de binario a decimal

Vamos a ver en primer lugar cómo traducir una secuencia de ceros y unos en algo comprensible para nuestras mentes decimales.

En la base decimal, que es la que nosotros utilizamos, llamamos a cada cifra de una manera diferente según el orden que ocupa: unidades, decenas, centenas, etc. Como nos explicaron en los primeros años del cole, para calcular un número a partir de su representación decimal, tenemos que sumar las unidades a las decenas multiplicadas por diez, las centenas multiplicadas por 100, etc., etc. Es decir: $534 = 5 * 100 + 3 * 10 + 4 * 1$.

En realidad, 100 es una potencia de 10 ($10^2 = 100$). Y por supuesto 10 también es una potencia de 10 ($10^1 = 10$). pero también el 1 lo es, ya que 1 es potencia de cualquier número, pues $X^0 = 1$, donde en este caso, es $X = 10$, es decir, 10 elevado a cero es uno.

Por tanto, el número 534 se puede representar como: $534 = 5*10^2 + 3*10^1 + 4*10^0$.

Esta regla se puede aplicar a cualquier otra base que no sea 10. Volvamos a la tabla anterior, y veremos que el número 7 se representa como 111 en base 2.

Si aplicamos la fórmula anterior, pero en este caso utilizando base 2, tendremos: $1 * 2^2 + 1*2^1 + 1*2^0 = 7$. En efecto, se cumple, ya que $2^2 = 4$, $2^1 = 2$, y $2^0 = 1$, luego: $1*4 + 1*2 + 1*1 = 7$.

Con esta sencilla fórmula de las potencias de 2 se puede convertir cualquier número binario a su equivalente en decimal. Por ejemplo, vamos a traducir a decimal el número 10011010.

Empezamos aplicando la fórmula: $1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$. Ahora, sabiendo los valores de cada potencia de dos (cualquier geek que se precie tiene que conocer como mínimo todas las

potencias de 2 con exponentes de 0 a 16), podemos traducir esa fórmula en: $1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$. Es decir, nos queda la siguiente suma: $128 + 16 + 8 + 2 = 154$. Por tanto, el número binario 10011010 representa al número 154 en decimal.

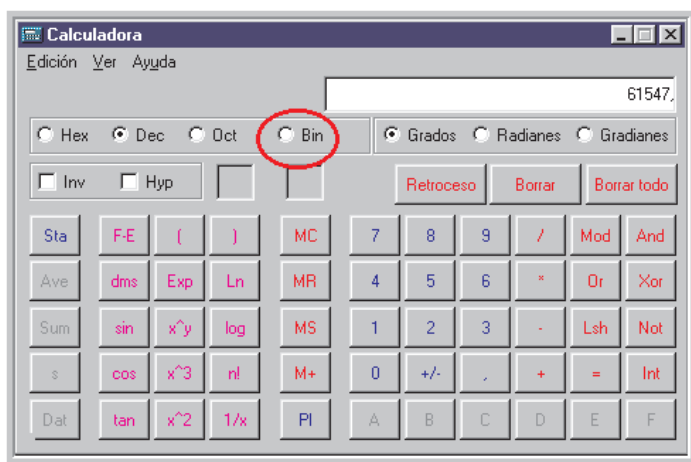
Por si queréis practicar, os dejo como ejercicio algunos números más, con su traducción, para que lo comprobéis vosotros mismos:

10110101 = 181
 00111100 = 60
 111010001010101010100101 = 15248037

2.2. Pasando de decimal a binario.

Aquí la cosa ya se pone más chunga. Aun así, juraría que esto ya lo expliqué en alguno de mis artículos.

Hay varios trucos para convertir de decimal a binario. El más sencillo, por supuesto, es meter el número en la calculadora de windows, y luego pinchar en donde pone BIN para que lo pase automáticamente, jeje. 😊



Para que...

Para que a nadie se le ocurra enviar un mail diciendo que la calculadora de Windows no puede hacer eso, venga, lo explicamos muy rápido. Abre la calculadora, Menu Ver y pulsa sobre Científica. Ya está 😊
 Ahora introduce cualquier número y pulsa sobre Bin 😊

Pero nosotros no nos conformamos con hacer las cosas, si no que nuestro auténtico interés es el saber cómo se hacen. Así que os explico rápidamente un algoritmo para convertir cualquier número decimal a binario.

Usemos para el ejemplo el número 137.

El proceso a seguir será ir dividiendo **el número** (en este caso 137) **por 2** y, en cada división, quedarnos con el **resto** de la división (que sólo podrá ser cero o uno). Ahora te quedará claro.

$$\begin{array}{r} 137 \quad | \quad 2 \\ 17 \quad | \quad 68 \\ \hline 1 \end{array}$$

El resultado de la primera división (llamado cociente, en verde) es 68, y el resto (en rojo) es 1. Este 1 será el bit menos significativo, es decir, la cifra binaria que está a la derecha del todo.

Continuamos el proceso con el nuevo cociente que hemos obtenido:

$$\begin{array}{r} 68 \quad | \quad 2 \\ 0 \quad | \quad 34 \end{array}$$

Ahora hemos obtenido un 0, que será la siguiente cifra binaria. Continuamos el proceso con el nuevo cociente, 34:

$34 / 2 = 17$, con resto 0.
 $17 / 2 = 8$, con resto 1.
 $8 / 2 = 4$, con resto 0.
 $4 / 2 = 2$, con resto 0.
 $2 / 2 = 1$, con resto 0.

Esta última operación (en azul) es fundamental, ya que el bit más significativo, es decir, el que hay más a la izquierda, será el último cociente, es decir $2/2 = 1$.

Por tanto, el número 137 en binario nos quedará: **1 0 0 0 1 0 0 1**.

Si tienes...

Si tienes interés en avanzar por tu cuenta en cálculo binario, hay miles de páginas en Internet que te lo explican perfectamente y por supuesto de forma gratuita. Busca en www.google.com y avanza tanto como quieras 😊

Construyendo un paquete TCP desde cero.

Ya podemos ponernos manos a la obra con el tema que nos ocupa, que son los paquetes TCP. Vamos a recordar la cabecera TCP (volved atrás un poco para ver la imagen), y a ir campo por campo construyendo el paquete.

Vamos a poner como ejemplo, el primer paquete que se envía cuando queremos establecer una conexión con un servidor de **FTP**.

En primer lugar, tenemos que conocer los **puertos de origen y de destino** (los dos primeros campos de la cabecera TCP). El puerto de **destino** será el **21**, que es el asignado por el estándar al servicio de **FTP** (aunque bien sabréis muchos de vosotros que no siempre se usa este puerto, como en el caso de los dumps, donde se suelen usar otros puertos menos "sospechosos").

El **puerto de origen** será un puerto aleatorio asignado por el sistema operativo a la hora de abrir el socket, es decir, la estructura utilizada por el sistema para establecer la nueva conexión TCP/IP. Supongamos que el sistema nos ha asignado el puerto **1345** como puerto de origen.

Ya tenemos los datos necesarios para rellenar la primera fila de la cabecera TCP. En primer lugar, tenemos que convertir el número 1345 en su equivalente binario, y lo mismo con el número 21.

Aplicando el algoritmo explicado en el punto anterior, obtenemos:

```
1345 = 10101000001
21   = 10101
```

Para construir la primera fila de la cabecera TCP tenemos que concatenar el puerto origen con el puerto destino, por lo que quedaría: 10101000001 10101.

Tenemos, por tanto, que nuestra primera fila consta de 16 bits... pero... no puede ser, si habíamos quedado en que cada fila de la cabecera TCP eran 32 bits.

El problema es que no hemos ajustado cada campo a su **tamaño de palabra**. El tamaño de palabra de cada uno de estos dos campos es de **16 bits**, por lo que cada uno de los dos números obtenidos tiene que ser representado con 16 cifras binarias. Para ello, habrá que poner el suficiente número de **ceros a la izquierda**, para rellenar las 16 cifras. Así, nos quedará:

```
1345 = 0000010101000001
21   = 0000000000010101
```

Ahora ya si que podemos concatenar ambos para conseguir la **primera fila** de la cabecera:

```
00000101010000010000000000010101
```

Como experimento, probad a convertir este número en su equivalente decimal. El resultado es 88145941. ¿Y qué nos dice este número? Pues absolutamente nada, ya que lo importante a la hora de traducir un número de una base a otra no es sólo la secuencia de cifras, si no también el cómo se agrupan estas. Si agrupamos esta secuencia en grupos de 16, entonces si que tendrá un sentido, pero si la agrupamos en una única secuencia de 32 bits, el número resultante no tiene ningún interés para nosotros. Por tanto, **es absolutamente imprescindible conocer el tamaño de las palabras**.

Vamos ahora con la segunda fila de la cabecera. Como vemos, ahora nos toca el campo **número de secuencia**. Este número también será asignado por el sistema operativo (recordemos del artículo anterior que no conviene que sea 0 cada vez que se establece una nueva conexión). En nuestro ejemplo el sistema nos asignará el número 21423994. Lo convertimos a binario, y nos da el número: 1010001101110011101111010.

Este número es de 25 bits, por lo que habrá que poner 7 ceros a su izquierda, para completar los 32 bits de la palabra que corresponde a este campo. Por tanto, la **segunda fila** de nuestra cabecera TCP será:

```
00000001010001101110011101111010
```

En este caso, el número completo de 32 cifras sí que tiene significado para nosotros, ya que el tamaño de la palabra es precisamente de 32 bits.

La **tercera fila** corresponde al campo **número de confirmación**. En nuestro caso tiene que ser cero, ya que es una conexión que aún no se ha establecido, por lo que el primer paquete no llevará confirmación de otro paquete anterior. Para representar el 0 con 32 bits, basta con meter 32 ceros. 😊

00000000000000000000000000000000

Ahora nos toca el campo **comienzo de datos**. Como ya vimos, el valor más habitual para este campo es 5, en el caso de que no haya ninguna opción. Pero nosotros vamos a incluir una opción, que ocupará 32 bits, como veremos más adelante. Como el campo Comienzo de datos indica el número de palabras de 32 bits que ocupa la cabecera TCP, al tener una palabra más para la opción, tendrá que ser 6, es decir: 110. Como el campo tiene una palabra de 4 bits, añadimos un cero a la izquierda: **0110**.

A continuación, la cabecera TCP tiene un campo vacío de 6 bits, que hay que rellenar con ceros: **000000**. Por tanto, de momento esta fila de la cabecera nos va quedando: **0110000000**.

Ahora le toca el turno a los **flags**. Como vimos en el artículo anterior, siempre que se desee establecer una nueva conexión el paquete ha de tener activado su flag **SYN**. El resto de flags estarán desactivados. Es decir, éste será el valor que tomarán todos los flags:

URG = 0
ACK = 0
PSH = 0
RST = 0
SYN = 1
FIN = 0

Si los colocamos todos juntitos en su orden nos quedará: **000010**.

Esto habrá que concatenarlo a lo que llevábamos ya construido de esta fila, por lo que nos quedaría: **01100000000000010**

Vamos ahora con el campo **tamaño de la ventana**. Un valor típico es, por ejemplo, 8192. Este número es una potencia de 2, concretamente 2^{13} . Por tanto, la traducción a binario es instantánea. Basta con poner 13 ceros a la derecha, y poner un único 1 a la izquierda del todo: 10000000000000. Esto nos da un número de 14 cifras, por lo que tenemos que ajustarlo al tamaño de la palabra de 16 bits con dos ceros a la izquierda: **0010000000000000**.

Por tanto, finalmente, la **cuarta fila** de la cabecera TCP nos quedará:

01100000000000100010000000000000

El próximo campo es el campo **suma de comprobación**, y es el que más quebraderos de cabeza nos va a dar. Si habéis seguido el resto del curso, habréis visto que hasta ahora he "eludido" un poco el tema, dándoos sólo una URL donde teníais un código en C ya hecho para calcular automáticamente los **checksums** (sumas de comprobación). Si lo hice así hasta ahora era porque sabía que más adelante llegaría el momento de enfrentarse cara a cara con los checksums, **y ese momento ya ha llegado.** 😊

Quizá os estaréis preguntando, ¿y por qué hay que enfrentarse al checksum si tenemos ya un código que nos lo calcula? ¿Para qué sirven todas estas vueltas y revueltas que estoy dando a los paquetes TCP cuando bastaría con conocer lo necesario para poder manejarlos?

Creo que es importante que hablemos aquí acerca del significado original de la palabra **HACK**, que forma parte del nombre de esta revista, y que justifica el hecho de que profundicemos hasta el más mínimo detalle en lo que explicamos.

El término *Hacker* ha sido muy desvirtuado con el paso del tiempo. Originalmente, un hacker era una persona cuya pasión era conocer el funcionamiento de las cosas. Para las personas "normales" una máquina es sólo una herramienta que se utiliza para algún fin concreto. En cambio, un hacker no se conforma sólo con usar las máquinas, si no que además ansía conocer su funcionamiento interno.

Hace años, se llamaba hackers a los grandes programadores, a los gurús de cualquier campo de la informática, y a toda esa clase de chiflados (a los cuales aspiro orgullosamente a pertenecer). Posteriormente, los medios de comunicación tergiversaron todo, y dieron a la palabra hacker el significado que antiguamente tenía la palabra cracker, y después estos términos han seguido evolucionando hasta el punto actual, en el cual hay mil y una definiciones para cada uno de los términos.

La cuestión es que si realmente queréis ser *hackers*, de los de toda la vida, vuestra pasión debe ser conocer hasta el mínimo detalle de cómo funcionan las cosas, y no sólo saber "manejarlas" sin más. Un tío que dedique a entrar en sitios donde teóricamente le estaba prohibido el paso, será un hacker sólo si su motivación para hacerlo sea explorar el funcionamiento de los sistemas, en caso contrario, su calificativo más apropiado será lamer, script kiddie, o el que más os guste. 😊

Pero bueno, ya he vuelto a salirme del tema... estábamos con el checksum. Pues me temo que de momento tenemos que dejar este punto en blanco, porque para calcular el checksum tenemos que tener terminada el resto de la cabecera, así que vamos a ver antes el resto de campos, y luego volvemos atrás sobre este punto.

El siguiente campo es el **puntero de urgencia**. Como el flag **URG** no está activo, este campo puede ser 0. Como son 16 bits, tendremos aquí: **0000000000000000**.

Por último, tenemos el campo **DATOS**. Como el paquete es sólo para establecer una conexión, no habrá ningún dato, por lo que este campo estará en blanco (ya no con ceros, si no que directamente no habrá **nada**).

Pero... ¡un momento! ¡Si habíamos dicho que íbamos a meter una opción! Entonces el campo **DATOS** no será el último, si no que tendremos antes el campo de **opciones TCP**. Para el caso nos va a dar igual, porque al fin y al cabo no hay campo de **DATOS**, así que en cualquier caso el campo **opciones** irá inmediatamente después del campo puntero de urgencia.

La opción que vamos a incluir es la única definida en el RFC de TCP, aunque ya vimos que existen muchas más: **Maximum Segment Size (MSS)**.

Todas las opciones empiezan con un byte que indica el **tipo de opción**. En nuestro caso, el código para la opción **MSS** es el **2**, es decir: **00000010**.

En el caso de la opción **MSS**, el siguiente byte contendrá la **longitud en bytes de la opción** que, contando con los dos primeros bytes que ya hemos mencionado (el que indica el código, y el que indica la longitud) será siempre **4**. Por tanto, el segundo byte de la opción **MSS** será siempre fijo: **00000100**.

Por último, los otros dos bytes que completarían la fila de 32 bits serán los que contengan el dato que queremos transmitir: el **tamaño máximo de segmento**. Si, por ejemplo, queremos un tamaño máximo de segmento de 1460 bytes, codificaremos este valor en 16 bits: **0000010110110100**.

Por tanto, toda la **fila de 32 bits para la opción MSS** nos quedaría: **000000100000001000000010110110100**.

Calculando la suma de comprobación (checksum).

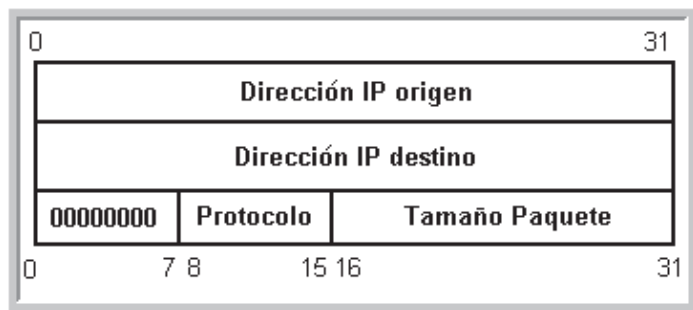
Ya podemos volver atrás un poco y calcular el último campo que nos falta para completar

la cabecera TCP de nuestro paquete. El primer paso a seguir es coger todo lo que tenemos hasta ahora, y agruparlo en palabras de 16 bits. Es decir, partimos de todos estos chorizos binarios:

0000 0101 0100 0001 = puerto de origen
 0000 0000 0001 0101 = puerto de destino
 0000 0001 0100 0110 = primeros 16 bits del número de secuencia
 1110 0111 0111 1010 = últimos 16 bits del número de secuencia
 0000 0000 0000 0000 = primeros 16 bits del número de confirmación
 0000 0000 0000 0000 = últimos 16 bits del número de confirmación
 0110 0000 0000 0010 = comienzo de datos, y flags
 0010 0000 0000 0000 = tamaño de la ventana
 0000 0000 0000 0000 = puntero de urgencia
 0000 0010 0000 0100 = código y longitud de la opción MSS
 0000 0101 1011 0100 = opción MSS

Por si todas estas ristras de ceros y unos os parecen pocas, todavía tenemos que añadir unas cuantas más, y es aquí cuando entra en juego esa pequeña cabecera de la que os hablé que se utilizaba a la hora de calcular el checksum.

Recordemos esta cabecera:



En primer lugar, necesitamos la **IP de origen**. Supongamos que tenemos un router ADSL que nos conecta con Internet, por lo que nuestra IP será una IP de red local, como por ejemplo: **192.168.1.1**.

Si, por ejemplo, el FTP al que conectamos es el de **Rediris** (<ftp.rediris.es>) sabemos que su **IP** es **130.206.1.5**.

El número de **protocolo** asignado a **TCP** es el **6**.

Por último, el tamaño del paquete TCP lo calculamos contando todos los bytes que

hay en la **cabecera TCP**, y todos los bytes de **DATOS**. Como en nuestro paquete no hay datos, bastará con contar los bytes (grupos de 8 bits) que ocupa la cabecera. Cada fila de la cabecera son 4 bytes (32 / 8 = 4), y tenemos un total de 6 filas, por lo que el tamaño de paquete será de 6 * 4 = **24**.

Ahora tenemos que pasar todo esto a binario:

IP de origen = 1100 0000 . 1010 1000 . 0000 0001 . 0000 0001
 IP de destino = 1000 0010 . 1100 1110 . 0000 0001 . 0000 0101
 Protocolo = 00000000000000110
 Tamaño de paquete = 0000000000011000

Lo que hay que hacer ahora con todos estos chorizos binarios es simplemente **sumarlos** (de ahí el nombre de "suma" de comprobación). El problema es que, si no tenéis práctica, sumar en binario os puede resultar complicado. Sería ya demasiado explicaros ahora toda la aritmética binaria, así que eso os lo dejo como ejercicio para que lo estudiéis por vuestra cuenta (www.google.com o utiliza la calculadora de Windows).

Lo que voy a hacer yo es pasar todo esto a hexadecimal para manejar menos cifras engorrosas. Lo que me queda al final es todo esto:

$$C0A8 + 0101 + 82CE + 0105 + 0006 + 0018 = 1459A$$

Este primer resultado es la suma de toda la pseudocabecera de checksum que acabamos de calcular. Ahora hay que hacer otra suma, pero con todos los chorizos que sacamos antes, de la propia cabecera TCP:

$$0541 + 0015 + 0146 + E77A + 0000 + 0000 + 6002 + 2000 + 0000 + 0204 + 05B4 = 175D0$$

Ahora sólo tenemos dos números, que tendremos que sumar a su vez:

$$1459A + 175D0 = 2BB6A$$

Este no es todavía el resultado, entre otros motivos porque el checksum ha de ocupar

sólo **16 bits**, y un número hexadecimal de 5 cifras, como el 2BB6A, ocupa 20 bits. Por tanto, lo que hacemos es coger la primera cifra (el 2) y sumarla al resto:

BB6A + 2 = BB6C

Ya tenemos completada la operación conocida como **suma en complemento a uno**.

Ahora sólo nos falta sacar a su vez el **complemento a uno** de este número, es decir, **invertir todos los bits** (donde haya un uno, poner un cero, y viceversa). Si pasamos este número (BB6C) a binario tenemos: **1011 1011 0110 1100**.

Si invertimos cada bit nos queda:

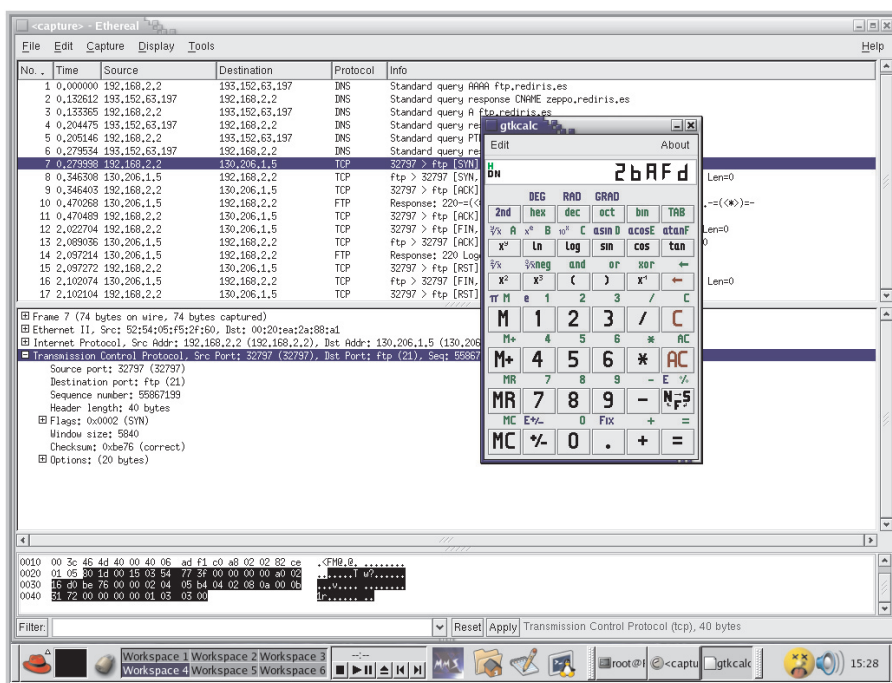
0100 0100 1001 0011

Este número de **16 bits** es al fin lo que tenemos que poner en el campo que nos quedaba por rellenar: la **suma de comprobación**.

Repasamos por tanto todo lo visto acerca de la suma de comprobación:

- 1- Agrupamos** toda la cabecera TCP en grupos de **16 bits**, y **sumamos** todos esos grupos entre sí.
- 2 -** A continuación, **construimos la pseudocabecera** con las ips de origen y de destino, el protocolo, y el tamaño de paquete, y **agrupamos** todo esto también en grupos **de 16 bits**.
- 3 - Sumamos** estos nuevos grupos al resultado obtenido en el primer paso.
- 4-** Del número obtenido en el paso 3, nos quedamos sólo con los **16 últimos bits**, y **los sobrantes los sumamos** a estos 16, quedándonos así un resultado de **16 bits**.
- 5-** El resultado del cuarto paso, lo **invertimos**, cambiando cada cero por un uno, y viceversa.

¿Que cómo he hecho todas esas sumas en hexadecimal? Pues, por supuesto, no de cabeza, si no usando una calculadora científica que admita hexadecimal. 😊



En esta captura podemos verme en plena faena, calculando el checksum de un paquete capturado con un sniffer. El sniffer me da directamente el paquete en hexadecimal, por lo que me facilita los cálculos.

¿Y cuál es la utilidad de todas estas operaciones? Pues para comprenderlo necesitáis saber que el complemento a uno de un número es el **opuesto** de ese número en complemento a uno, es decir, si sumas en complemento a uno, es decir, si sumas en complemento a uno, el resultado tiene que ser siempre **cero**. Vamos, en resumen, que es como decir que -5 es el complemento a uno de 5, ya que 5 + (-5) = 0.

Esto hace que el procesamiento de los paquetes sea bastante rápido a la hora de verificar los checksum, ya que basta con que el software sume todos los datos del paquete y, si el paquete es correcto, el resultado tiene que ser 0.

Esto es lógico, ya que la suma que nosotros hemos hecho hace un momento contenía todos los datos del paquete excepto uno: la propia suma de comprobación. El paquete que llegue al receptor, en cambio, si que contendrá además ese dato, y como ese dato es precisamente el opuesto de la suma de todos los demás, al sumar todos los datos más la suma de comprobación, el resultado será:

$$\text{checksum} + \text{resto de cabecera} = 0$$

Ya que, insisto:

$$\text{checksum} = - (\text{resto de cabecera}).$$

Os propongo como ejercicio que comprobéis todo esto con un sniffer. Capturad un paquete TCP, sumad todos los datos de la cabecera TCP, del campo DATOS, y de la pseudocabecera utilizada en el checksum, y comprobaréis que, si el paquete no contiene errores, el resultado es siempre cero.

Resumiendo

Al final, este es el paquete que nos queda, y que será enviado tal cual desde nuestro PC hasta el receptor (en este caso, el servidor FTP de Rediris):

0000 0101 0100 0001	0000 0000 0001 0101
0000 0001 0100 0110 1110 0111 0111 1010	
0000 0000 0000 0000 0000 0000 0000 0000	
0110 000000	0010 0000 0000 0000
0100 0100 1001 0011	0000 0000 0000 0000
0000 0010	0000 0100 0000 0100

4. La respuesta a nuestro paquete

Al recibir este paquete el servidor FTP de Rediris, nos responderá con el siguiente paquete, que os muestro como ejercicio para que lo analicemos:

0000 0000 0001 0101	0000 0101 0100 0001
0010 0111 1001 0010 1000 0000 0101 0011	
0000 0001 0100 0110 1110 0111 0111 1011	
0110 000000	0011 0000 0010 0110
0111 1101 0000 0010	0000 0000 0000 0000
0000 0010	0000 0100 0000 0100

Vamos a analizarlo:

En primer lugar, los **puertos de destino y de origen** están **intercambiados**, como es lógico.

En segundo lugar, vemos que el **número de secuencia** no tiene nada que ver con el nuestro, ya que cada extremo de la comunicación usará sus propios números de secuencia.

En cambio, el que sí que tiene que ver con nuestro número de secuencia es **su número de confirmación**. Al no contener datos nuestro paquete, el próximo byte que enviaríamos sería el inmediatamente posterior al número de secuencia de nuestro paquete anterior. Por tanto, el servidor de Rediris estará esperando recibir en el próximo paquete un número de secuencia que sea el que enviamos antes, + 1.

Vemos que su campo **comienzo de datos** es el mismo que el nuestro, ya que el paquete también contendrá una única fila de opciones (de 32 bits).

Donde vemos que sí que hay un cambio es en los **flags**, ya que aquí no sólo está activado el flag **SYN**, si no también el flag **ACK**. En el próximo punto veremos en detalle a qué se debe esto.

Vemos también que el **tamaño de la ventana** es diferente, ya que cada extremo de la comunicación puede tener su propio tamaño de ventana.

La **suma de comprobación**, por supuesto, es diferente para cada paquete. Os propongo como ejercicio que comprobéis la validez de la suma de comprobación de este paquete y, en caso de que sea incorrecta, que calculéis cuál sería el checksum correcto.

El **puntero de urgencia** también está a cero, ya que tampoco tiene el flag **URG** activado.

Por último, vemos que también añade la **opción MSS**, con el mismo tamaño máximo

de segmento: 1460 bytes.

El paquete tampoco contiene nada en el campo **DATOS**, ya que es otro de los paquetes utilizado únicamente para establecer la conexión.

5. Los estados de conexión TCP

No podemos completar un curso sobre TCP sin hablar de los estados de conexión. Para ello, empezaremos viendo una serie de procedimientos que se llevan a cabo en las conexiones TCP, para luego enumerar los estados que hemos ido descubriendo.

5.1. Establecimiento de conexión TCP.

En el ejemplo anterior, los paquetes involucrados correspondían a un establecimiento de conexión. Como vimos, había por lo menos dos paquetes encargados de establecer la conexión: uno por nuestra parte, que le decía al servidor de Rediris que queríamos establecer la conexión, y otro por parte del servidor de Rediris que nos decía que estaba de acuerdo y él también quería establecer la conexión.

En realidad, el establecimiento de conexión TCP requiere aún otro paquete más, y es lo que hace que a este sistema de establecimiento de conexión se le llame **3-way handshake** o, traducido así a lo bruto: saludo de 3 pasos.

El sistema de establecimiento de conexión TCP consiste en lo siguiente:

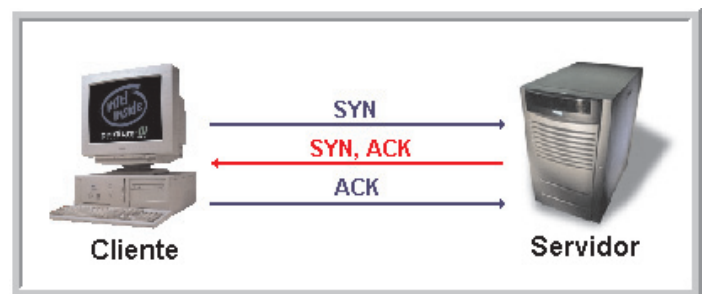
- 1- El cliente solicita al servidor una conexión.
- 2- El servidor responde aceptando la conexión.
- 3- El cliente responde aceptando la conexión.

¿A qué se debe la necesidad de este tercer paso? Este tercer paso permite una sincronización exacta entre cliente y servidor, y además permite al cliente "echarse atrás" si no le gusta algo en la respuesta del servidor.

Por ejemplo, en el caso del servidor de Rediris, si habéis probado a conectaros habréis visto que nada más conectar envía un texto de presentación. Si nosotros no respondiésemos al servidor diciéndole que aceptamos la conexión, aún habiéndola solicitado nosotros, el servidor se pondría en vano a enviar todo ese texto sin saber si al otro lado hay alguien escuchando.

Estos tres paquetes especiales utilizados en el 3-way handshake se caracterizan por sus flags:

- ▶ El cliente solicita conexión al servidor: flag **SYN**.
- ▶ El servidor acepta la conexión: flags **SYN** y **ACK**.
- ▶ El cliente acepta la conexión: flag **ACK**.



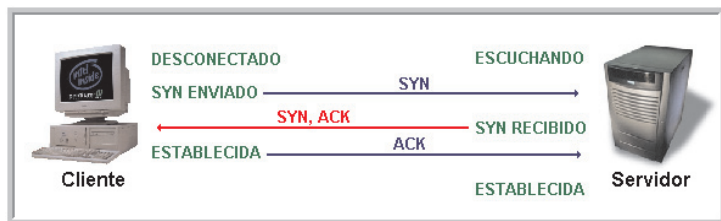
Esto ha de ser siempre así, y si alguno de esos flags no es enviado en el orden correcto, todo el establecimiento de conexión será anulado.

Intuitivamente, nos damos cuenta de que todo este mecanismo nos da lugar a diferentes estados en la conexión:

- ▶ En primer lugar, el estado primordial es el estado **DESCONECTADO**, que es cuando aún ni siquiera hemos enviado el SYN.
- ▶ En segundo lugar, una vez enviado el SYN, estamos en un estado diferente que espera al próximo paso. A este paso lo podemos llamar **SYN ENVIADO**.
- ▶ Una vez que el servidor recibe el primer SYN, nos enviará su respuesta con el SYN y el ACK en el mismo

paquete. En ese caso, pasaremos a un estado que podemos llamar **SYN RECIBIDO**. Ahora nos falta esperar al último paso del establecimiento, que es el último ACK.

► Una vez recibido el último ACK, nuestra conexión pasará finalmente al estado de conexión **ESTABLECIDA**.



Si alguna vez habéis utilizado la herramienta **netstat** posiblemente os suenen estos nombres. Si no, probad ahora mismo, desde una shell de Linux/Unix, o una ventana Ms-DOS de Windows, a escribir:

netstat

Veréis la lista de conexiones de vuestra máquina, con el estado de cada una. Las conexiones que estén en estado ESTABLECIDA, que son las más habituales, aparecerán como **ESTABLISHED**.

Podréis ver otros estados como **CLOSE_WAIT**, **FIN_WAIT**, **LISTEN**, etc. En breve explicaremos todos ellos.

Escaneo de puertos con SYN

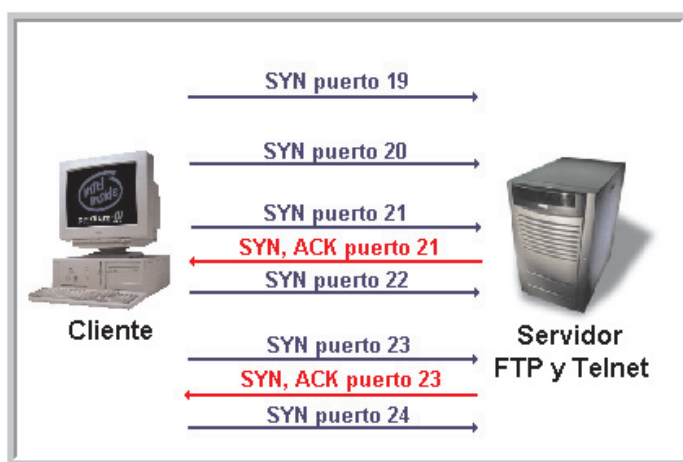
Os propongo como experimento que pongáis en marcha alguna aplicación de escaneo de puertos, y a continuación hagáis un **netstat**.

Probablemente (dependiendo del tipo de escaneo que utilice la aplicación), veréis que hay montones de conexiones en estado **SYN_SENT**, es decir, lo que nosotros hemos llamado SYN ENVIADO.

Esto se debe a que un sistema clásico de escaneo consiste en hacer solicitudes al servidor para establecer conexiones en cada uno de los puertos, es decir, enviamos un paquete **SYN** a cada puerto. Los paquetes

que tengan como respuesta un **SYN, ACK** corresponderán a los puertos abiertos de la máquina. Los paquetes que no tengan respuesta, o bien que sean respondidos con un flag **RST**, estarán cerrados.

Lo interesante aquí es que nosotros no responderemos a ninguno de los **SYN, ACK**, por lo que ninguna conexión quedará establecida, ya que sería necesario que respondiésemos con un nuevo paquete **ACK** por cada paquete **SYN** que enviásemos.



Ataque SYN Flood

Ya que hemos empezado hablando de cosas divertidas, como el escaneo de puertos, vamos a rematar la faena hablando de una técnica de hacking realmente interesante, aunque más por el interés de su funcionamiento que por su utilidad práctica, ya que es un ataque de tipo **DoS (Denial of Service)**, es decir, que sólo sirve para fastidiar y tirar abajo un servidor.

¿Por qué hablamos de diferentes estados en una conexión? Pues porque es necesario tener en cuenta los diferentes estados a la hora de llevar a cabo todos los pasos necesarios para conseguir llevar a cabo la comunicación.

Por ejemplo, en el momento en que pasamos al estado **SYN_SENT**, tenemos que crear una **estructura de datos** que necesitaremos para mantener controlado el estado de la conexión en todo momento. Sería absurdo tener estos

datos creados de antemano, ya que ocuparían una gran cantidad de memoria innecesaria y, además, tampoco podríamos saber cuántas estructuras de este tipo necesitaríamos, pues depende en todo momento de cuántas conexiones tengamos establecidas. Por tanto, al cambiar un sistema al estado SYN_SENT, creará una estructura de datos para mantener la conexión inminente. Esta estructura de datos se llama **TCB (Transmission Control Block)**.

Por tanto, cada vez que intentamos conectar con un servidor, estamos haciéndole crear una estructura que ocupa lugar en su memoria. En el momento en que se cierre esa conexión, el servidor podrá borrar el TCB correspondiente, recuperando así el espacio que había ocupado en su memoria.

¿Qué pasaría entonces si saturásemos al servidor a base de conexiones? Si esta saturación es suficiente, conseguiremos dejar sin memoria al servidor para establecer nuevas conexiones.

Para que esta saturación sea efectiva, es conveniente que utilicemos direcciones IP falsas, es decir, que hagamos un **IP Spoofing**.

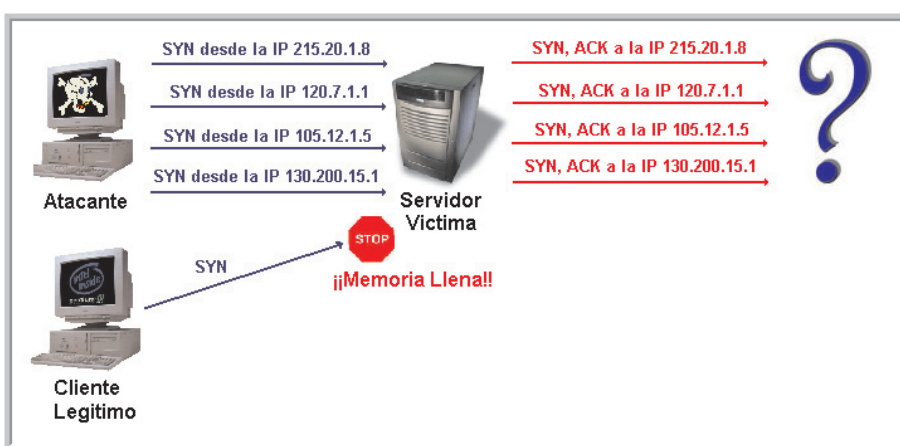
Si conseguimos enviar al servidor una gran cantidad de paquetes, cada uno de ellos conteniendo un flag **SYN** solicitando una conexión, y cada uno con una dirección IP falsa, el servidor creará un TCB para cada una de esas conexiones falsas inminentes y, al mismo tiempo, enviará el **SYN, ACK** a cada una de las direcciones falsas.

A continuación, se quedará esperando a recibir el **ACK** para completar el establecimiento de conexión de cada una de las conexiones falsas.

Por supuesto, este **ACK** jamás llegará, y el servidor se quedará esperando hasta que se canse. Lo malo es que los servidores son

bastante pacientes, y suelen esperar en torno a unos 3 minutos antes de dar por perdida una conexión. Por tanto, si saturamos al servidor a base de **SYNs**, en unos 3 minutos nadie podrá conectarse legítimamente al servidor, ya que su memoria para nuevas conexiones estará llena en espera de completar las que tiene pendientes.

Si este bombardeo de **SYNs** se repite constantemente, el servidor quedará inutilizado mientras dure el bombardeo.



Si leísteis mi artículo de la serie RAW sobre el protocolo **DNS**, o mi artículo sobre **UDP** del curso de TCP/IP, conoceréis la técnica de envenenamiento de caché DNS. Cuando expliqué esta técnica mencioné que una ayuda para hacer más efectivo el ataque era conseguir hacer una denegación de servicio (DoS) al servidor DNS legítimo.

En cambio, la técnica de **SYN Flood** no puede funcionar en UDP, ya que sencillamente UDP no tiene ningún flag, y menos aún el flag SYN, por lo que en este caso el utilizar UDP frente a TCP es una ventaja para el protocolo DNS.

Más adelante veremos cómo llevar a cabo un ataque SYN Flood en detalle de forma práctica. 😊

5.2. Cierre de conexión TCP

Continuando con el asunto de los estados de conexión en TCP, vamos a ver otro procedimiento que se puede llevar a cabo con cualquier conexión, y es el cierre de la misma.

Si pensamos un poco, nos daremos cuenta de que hay un pequeño problema inherente a cualquier **conexión full-duplex**, es decir, las conexiones en las que cualquiera de las dos partes puede tanto transmitir como recibir. El problema es que, si ambos quieren transmitir datos, ambos tendrán que ponerse de acuerdo para decidir en qué momento hay que cerrar la conexión.

Si no hubiese esta clase de acuerdos, ocurrirían cosas poco deseables, como por ejemplo que conectásemos con un FTP, solicitásemos un archivo, y tras enviárnoslo el servidor nos cerrase la conexión, asumiendo que ya no queremos bajar ni subir nada más. O, por ejemplo, conectarnos a un chat, decir "hola", y que el servidor decidiese que ya no queremos decir nada más y, por tanto, nos cerrase la conexión.

Por tanto, en una conexión full-duplex es necesario que ambas partes se pongan de acuerdo sobre el momento en el que hay que cerrar la conexión.

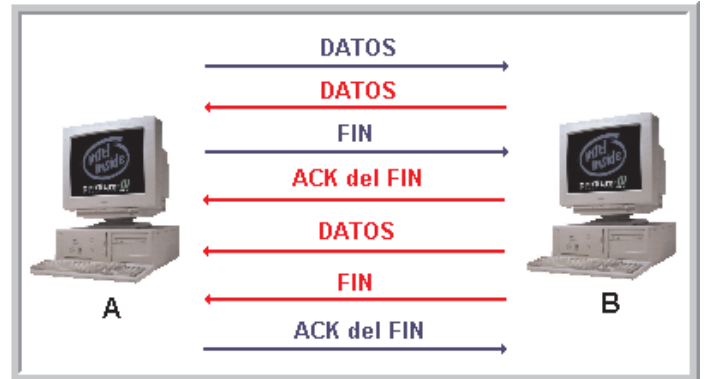
En el caso de TCP, el sistema que se utiliza para conseguir esto es sencillamente que cada uno, por su cuenta, indique al otro el momento en el que quiere cerrar la conexión.

Una vez que el otro se ha enterado, habrá que esperar a que él también desee cerrar la conexión. Es decir, si hemos terminado de enviar datos, decimos "Por mi ya está todo hecho. Avisame cuando termines tú". En el momento en que el otro termine, avisará diciendo: "Vale, yo también he terminado, así que hasta luego".

Para dar este tipo de avisos lo que se hace es enviar un paquete especial que tiene activado un flag que sirve precisamente para indicar que deseamos **FINALizar** la conexión. Este flag es el flag **FIN**.

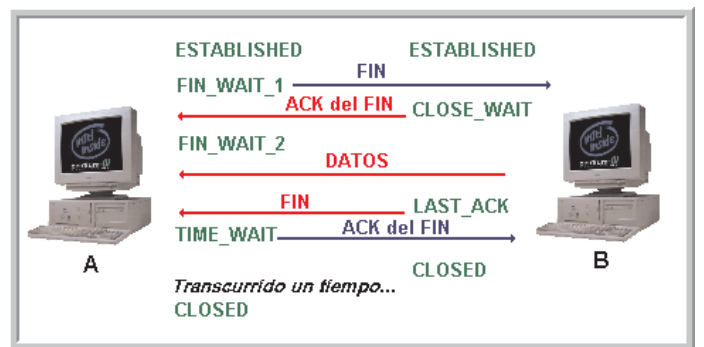
A partir del momento en que enviamos un paquete con el flag **FIN**, ya **no debemos enviar** ningún paquete más (excepto en el

caso de que tuviéramos que reenviar un paquete anterior porque no recibiésemos su confirmación), y lo único que debemos hacer es **seguir recibiendo** los datos de nuestro compañero, hasta que éste también nos envíe su paquete con el flag **FIN**.



Esto, una vez más, nos da lugar a nuevos estados de conexión. Una vez que enviamos nuestro **FIN** entramos en un estado en el que no podemos enviar, y sólo podemos recibir. A este estado lo podemos llamar **ESPERA DEL FIN**.

En realidad, el cierre de conexión da lugar a varios estados diferentes, que podemos ver en la siguiente imagen:



Aquí he puesto ya los nombres auténticos de los estados, en inglés, porque sería demasiado rebuscado tratar de traducirlos, jeje.

Como vemos, el estado que he llamado antes ESPERA DEL FIN es el que se llama **FIN_WAIT_1**.

En el momento en que nuestro compañero recibe nuestro **FIN**, él entra en otro estado diferente, que es el **CLOSE_WAIT**, es decir,

en espera de cerrar, ya que es ahora responsabilidad suya terminar de enviar cuando pueda, para enviar él también su **FIN**.

Por otra parte, cuando recibe nuestro **FIN**, tiene que confirmarnos su recepción, igual que con cualquier otro paquete, enviándonos un **ACK**.

En el momento en que recibimos ese **ACK**, entramos en otro estado, que es el **FIN_WAIT_2**.

En el momento en que nuestro compañero termina, éste envía su **FIN**, y queda en espera de recibir nuestra confirmación. Durante esta espera, entra en estado **LAST_ACK**.

Una vez que recibimos ese **FIN**, entramos en estado **TIME_WAIT**, y enviamos el **ACK** para el **FIN** de nuestro compañero.

Una vez completado todo esto, ambas partes pasan a estado **CERRADO**, y se acabó el tema. La máquina **A** esperará un tiempo prudencial para pasar a estado **CERRADO**, ya que no puede estar seguro de que **B** haya recibido correctamente su último **ACK**.

5.3. Lista de estados TCP

Ya podemos ver la lista completa de estados de conexión en TCP, que os servirá como guía de referencia, sobre todo para cuando utilicéis herramientas como **netstat**.

LISTEN – Es el estado en el que permanece cualquier servidor cuando está en espera a que un cliente se conecte. Todos los puertos que tengamos abiertos en nuestro PC nos generarán un socket TCP (o UDP, según el caso) en estado LISTEN. Cada vez que un cliente se conecte, si permitimos más de una conexión, se creará un socket en estado ESTABLISHED para ese cliente, y otro en estado LISTEN para esperar al resto de clientes.

SYN_SENT – Se entra en este estado cuando solicitamos una conexión a un servidor (enviamos un paquete SYN), y aún no hemos recibido su aceptación o su rechazo. (*Ver establecimiento de conexión*).

SYN_RECEIVED – Se entra en este estado cuando tanto cliente como servidor han enviado sus correspondientes SYN, y estamos en espera de que se complete el tercer y último paso del establecimiento de conexión. (*Ver establecimiento de conexión*).

ESTABLISHED – Conexión establecida. Es el estado habitual.

FIN_WAIT_1 – Hemos enviado un paquete FIN, y sólo podemos recibir, pero no enviar. Estamos esperando a que nuestro compañero nos confirme que ha recibido nuestro FIN. Queremos cerrar la conexión, pero aún no sabemos si nuestro compañero se ha enterado. (*Ver cierre de conexión*).

FIN_WAIT_2 – Hemos enviado un paquete FIN, y sólo podemos recibir, pero no enviar, pero además hemos recibido ya la confirmación (ACK) de nuestro FIN. Por lo tanto, sabemos ya que nuestro compañero conoce nuestras intenciones de cerrar la conexión. (*Ver cierre de conexión*).

CLOSE_WAIT – Hemos recibido el FIN de nuestro compañero, pero a nosotros todavía nos quedan datos por enviar. (*Ver cierre de conexión*).

CLOSING – Esperamos a la última confirmación para cerrar definitivamente la conexión. Es un estado al que se llega cuando ambas partes desean cerrar la conexión simultáneamente, al contrario del caso que explicamos anteriormente.

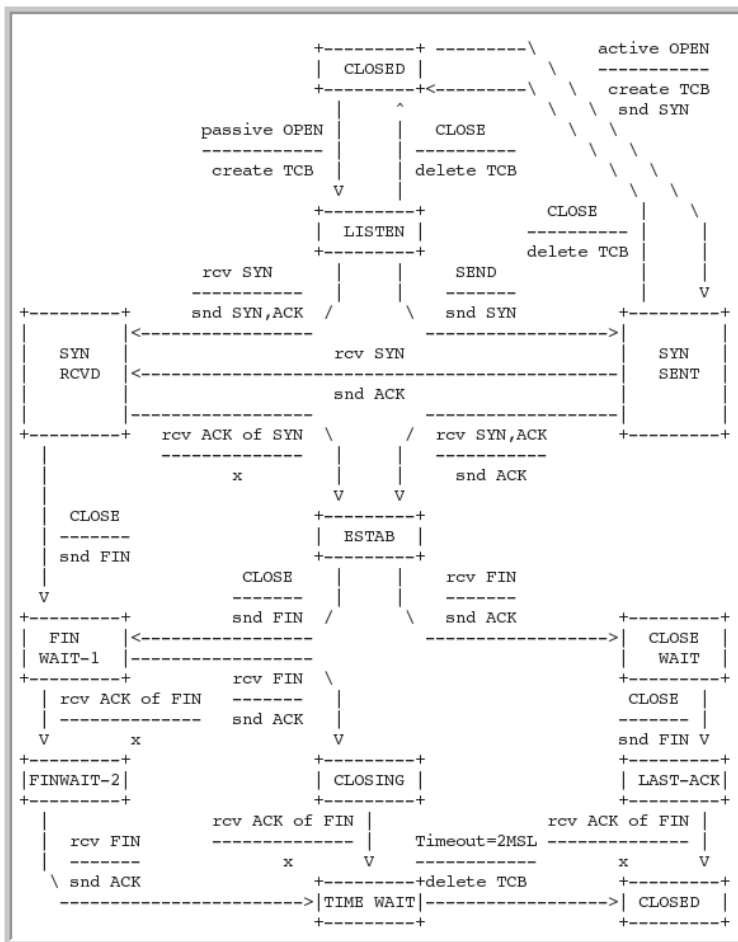
LAST_ACK – Esperamos a la confirmación (ACK) de nuestro FIN, cuando eramos nosotros los últimos que faltabamos por enviar el FIN. (*Ver cierre de conexión*).

TIME_WAIT – Hemos enviado la confirmación del FIN a nuestro compañero, cuando era él el que faltaba por enviar el FIN. Se llama así, porque lo que hacemos es esperar un tiempo prudencial para asumir que ha recibido nuestra confirmación. Siempre que nosotros cerremos

una conexión, durante un tiempo permaneceremos en estado TIME_WAIT, por lo que es bastante común encontrar este estado cuando hacemos netstat. (Ver cierre de conexión).

CLOSED - Es un estado ficticio, que simplemente dice que no hay ningún tipo de conexión.

A continuación, muestro el diagrama de estados de TCP, tal y como lo podéis encontrar en el RFC:



6. RAW Sockets TCP (Nemesis y Hping)

Vamos a recordar un poco las herramientas que expliqué para manejar RAW sockets en el artículo sobre UDP, pero en este caso aplicadas al caso de TCP. Esta vez empezaremos por Linux.

6.1. Hping2 para Linux

El caso de TCP es bastante más complicado que el de UDP, ya que TCP da muchísimo más juego. No hay más que ver el manual:

man hping2

Para ver la gran cantidad de opciones que hay para TCP que, por cierto, es el protocolo por defecto de Hping2. Nosotros vamos a ver sólo las opciones que directamente tienen que ver con lo explicado hasta ahora.

Empezamos con la prueba más sencilla: **hping2 130.206.1.5 --destport 21 --count 1**

Con esto enviamos un único (--count 1) paquete TCP a la dirección 130.206.1.5, al puerto 21.

Este paquete tendrá todos sus parámetros tal y como los tiene configurados hping2 por defecto, es decir, sin ningún flag, con un tamaño de ventana de 64 bytes, y sin opciones.

Este paquete, por tanto, sirve para bien poco, aunque en el manual de hping2 nos explican que, si ni siquiera utilizamos el parámetro --destport, por defecto envía el paquete al puerto 0, y esto puede ser útil para hacer un "ping" a una máquina que tenga un firewall que filtre los auténticos pings (que no son TCP, si no ICMP, que es otro protocolo), y además es probable que éste intento nuestro de ping ni siquiera quede reflejado en los logs de la máquina. Aún así, esto es algo que yo no he comprobado, así que no sé qué utilidad tendrá. Os propongo que lo probéis vosotros mismos como ejercicio. 😊

Vamos a ver las diversas opciones que nos da Hping2 para TCP:

--baseport: permite especificar nuestro puerto de origen.

--destport: permite especificar el puerto de destino.

--keep: si enviamos varios paquetes, evita

que el puerto de origen se vaya incrementando automáticamente, tal y como vimos con UDP.

- win:** fija el tamaño de la ventana TCP.
- tcpoff:** envía un valor falso para el campo Comienzo de Datos de la cabecera TCP.
- tcpseq:** especifica el Número de Secuencia.
- tcpack:** especifica el Número de Confirmación.
- badcksum:** igual que en UDP, envía un checksum erróneo.
- fin:** el paquete que enviamos tiene activo el flag FIN.
- syn:** el paquete que enviamos tiene activo el flag SYN.
- rst:** el paquete que enviamos tiene activo el flag RST.
- push:** el paquete que enviamos tiene activo el flag PUSH.
- ack:** el paquete que enviamos tiene activo el flag ACK.
- data:** especifica el tamaño del campo DATOS, sin contar con la cabecera.
- file:** igual que en UDP, permite rellenar el campo DATOS con los contenidos de un archivo que especifiquemos.
- safe:** nos permite asegurarnos de que los paquetes que enviamos llegan a su destino ya que, tal y como ha de hacerse en TCP, si no recibimos la confirmación de alguno de los paquetes, hping2 lo reenviará automáticamente.

Vamos a ver todo esto y mucho más con un ejemplo muy interesante, que es para poner en práctica la técnica de SYN Flood explicada anteriormente.

SYN Flood mediante Hping2.

Esta técnica sólo debéis utilizarla para hacer pruebas con vosotros mismos, para comprender el funcionamiento de la técnica, y también para poner a prueba la seguridad de vuestra red, por si queréis hacer una auditoría de seguridad y arreglar los agujeros que tengáis.

Cualquier otro uso que le deis, al margen de que pueda ser ilegal, éticamente será indeseable, ya que no estaréis más que

fastidiando por fastidiar. Además, es poco probable que funcione un IP spoofing a pelo como el que voy a explicar, ya que los routers que haya en el camino desde vosotros hasta vuestra "víctima" probablemente rechacen los paquetes si no provienen de una IP que forme parte de su red.

Recordemos que para explotar la técnica de SYN Flood "simplemente" hay que enviar gran cantidad de paquetes con flag **SYN**, cada uno con una **dirección IP de origen falsa** y, a ser posible, diferente. Hping2 "casualmente" tiene opciones para automatizar todo esto, por lo que nos basta con esta línea:

```
hping2 192.168.1.1 --rand-source --destport 21 --syn --count 100
```

Con esta línea enviaremos 100 paquetes (--count 100) al puerto 21 de la IP 192.168.1.1, utilizando como IP de origen una aleatoria en cada paquete (--rand-source), y con el flag SYN activado.

Os puedo asegurar que esta línea funciona, ya que acabo de probarla ahora mismo con el puerto de telnet (--destport 23) de mi router ADSL, y ahora me es imposible conectar con el telnet del router.

¿Significa esto que yo, que precisamente estoy explicando estas cosas, tengo un grave problema de seguridad? Realmente no, por tres motivos. En primer lugar, porque el puerto de Telnet lo tengo abierto sólo hacia mi red local, por lo que sólo podría atacarme... yo mismo. En segundo lugar, porque no es un servicio de importancia crítica, es decir, me da igual tirarme el tiempo que sea sin poder acceder al telnet de mi router, ya que sólo lo uso muy rara vez, cuando tengo que modificar algo en la configuración. En tercer lugar, al tratarse de un router hardware y no de un simple programa de PC, tendría que esperar a que saliese una nueva actualización del firmware que solucionase este problema, así que en cualquier caso no está en mi mano la solución, si no en la del fabricante del router. Ya que la cosa se está calentando un poco,

vamos a probar alguna técnica más de hacking relacionada con TCP. 😊

Ataques por adivinación de número de secuencia con Hping2.

Vamos ahora con una técnica realmente interesante que, de hecho, utilizó incluso el propio Kevin Mitnick (uno de los hackers más famosos de la historia) como parte de las andanzas que le hicieron terminar en la cárcel (aplicaos el cuento, jeje).

En este caso, no se trata de un simple ataque DoS, como el SYN Flood, si no de un ataque mucho más versátil que nos permitirá **colarnos en conexiones ajenas**, con todo lo que ello implica.

Conseguir un ataque de este tipo con éxito es realmente complicado, así que lo que voy a contar, que en la teoría puede parecer tan "sencillo", en la práctica choca con mil y un inconvenientes, empezando por la dificultad que comenté antes de hacer un IP Spoofing sin que se enteren los routers que transportan el paquete.

Como lo importante es comprender la teoría de las cosas, y no meternos en líos, voy a explicar las bases de este tipo de ataques.

Para comprender el funcionamiento de estos ataques debemos recordar qué es lo que define exactamente una conexión, es decir, lo que identifica unívocamente a una conexión para diferenciarla de cualquier otra de Internet. Pues son estos los parámetros: **una IP de origen, una IP de destino, un puerto de origen, un puerto de destino, y los números de secuencia de cada una de las dos partes.**

Por tanto, si conociéramos todos estos datos, teniendo una herramienta como Hping2 que nos permite crear paquetes a medida, podríamos insertar cualquier dato en una conexión ajena.

Por ejemplo, si sabemos que la máquina A, con IP 192.168.1.2, tiene establecida una

conexión de FTP (puerto de destino 21) con la máquina B, con IP 192.168.1.5, utilizando como puerto de origen el 3560, nos bastaría con saber el número de secuencia que está utilizando la máquina A para poder inyectar paquetes en su conexión de FTP. Supongamos que sabemos que su número de secuencia es el 24560.

Bastará con hacer:

```
hping2 192.168.1.5 --spooft 192.168.1.2 --baseport 3560 --destport 21 --tcpseq 24560 --file comandos.txt --data 14 --count 1
```

Con esto enviamos un único paquete (--count 1) enviando como IP spoofeada la de la máquina A (--spooft 192.168.1.2), e inyectando como datos unos comandos de FTP que hemos metido previamente en el archivo COMANDOS.TXT.

Por supuesto, el gran problema de esto es que es realmente complicado conocer el número de secuencia de una conexión ya que, además de ser un número realmente grande (32 bits), va cambiando constantemente a lo largo de una conexión.

Hping2 nos ofrece una herramienta para ayudarnos a la hora de adivinar el número de secuencia, comprobando si un determinado sistema utiliza números de secuencia fáciles de predecir. Para ello nos da la opción --seqnum, que hace un análisis de los números de secuencia utilizados por un sistema:

```
hping2 192.168.1.5 --seqnum --destport 21 --syn
```

Con esto veríamos cómo varían los números de secuencia de la máquina B cada vez que intentamos conectar con su puerto de FTP. Hping2 nos mostrará el número de secuencia utilizado, y el incremento con respecto al utilizado anteriormente. Si este incremento es siempre el mismo, entonces estaremos ante una máquina con números de secuencia "fácilmente" predecibles.

Una vez que ya tenemos una idea del rango de números de secuencia que puede utilizar la máquina que queremos suplantar, podemos intentar lanzar miles de paquetes iguales, en los cuales sólo cambie el número de secuencia, y esperar que el azar nos recompense con la suerte de que alguno de ellos haya acertado, y el paquete se inyecte correctamente en la conexión ajena.

6.1. Nemesis para Windows

Para empezar, os recuerdo que en el directorio de Nemesis tenéis un archivo de ayuda para cada protocolo. En este caso el que nos interesa es el archivo **nemesis-tcp.txt**. Como ya me estoy quedando sin espacio, os resumo brevemente las opciones que nos da Nemesis para TCP, que son bastantes:

- x : permite especificar el **puerto de origen**.
- y : permite especificar el **puerto de destino**.
- s : permite especificar el **número de Secuencia**.
- a : permite especificar el **numero de confirmación**.
- fS : activa el flag **SYN**
- fA : activa el flag **ACK**
- fR : activa el flag **RST**
- fP : activa el flag **PSH**
- fF : activa el flag **FIN**
- fU : activa el flag **URG**
- w : permite especificar el **tamaño de la ventana**.
- u : permite especificar el campo **puntero**

de urgencia.

- o : permite incluir un fichero que contenga las **opciones TCP** que queramos.
- v : activa el modo **verbose** que nos da información más detallada de lo que estamos haciendo.

Os recuerdo también que necesitaremos un par de opciones referentes a IP:

- D : permite especificar la **IP de destino** (imprescindible).
- S : permite especificar la **IP de origen** (IP Spoofing).

Por ejemplo, si queremos enviar un paquete de solicitud de conexión al FTP de Rediris podemos hacer:

```
Nemesis tcp -v -S 192.168.1.1 -D 130.206.1.5 -x 1000 -y 21 -fS -a 0
```

En primer lugar especificamos nuestra IP (-S 192.168.1.1), que en este caso es una IP de red local porque nos encontramos detrás de un router ADSL. En segundo lugar indicamos la IP de destino, es decir, la del servidor FTP de Rediris (-D 130.206.1.5). A continuación especificamos los puertos de origen y de destino (-x 1000 -y 21). A continuación, activamos el flag SYN para este paquete (-fS). Por último, ponemos un 0 en el campo número de confirmación, ya que es para una conexión aún no establecida (-a 0).

¿QUIERES COLABORAR CON PC PASO A PASO?

PC PASO A PASO busca personas que posean conocimientos de informática y deseen publicar sus trabajos.

SABEMOS que muchas personas (quizás tu eres una de ellas) han creado textos y cursos para “consumo propio” o “de unos pocos”.

SABEMOS que muchas personas tienen inquietudes periodísticas pero nunca se han atrevido a presentar sus trabajos a una editorial.

SABEMOS que hay verdaderas “obras de arte” creadas por personas como tu o yo y que nunca verán la luz.

PC PASO A PASO desea contactar contigo!

NOSOTROS PODEMOS PUBLICAR TU OBRA!!!

SI DESEAS MÁS INFORMACIÓN, envíanos un mail a empleo@editotrans.com y te responderemos concretando nuestra oferta.