OCCAM'S RAZOR

ISSN: 1998-0537

Número 2 Segunda Época
Abril 2017

EN ESTE NÚMERO

INGENIERÍA INVERSA

ESTEGANORAFÍA

RECONOCIMIENTO DE VOZ

SUMARIO

Número 2 Segunda Época
Abril 2017

Mensajes Clandestinos
por Yvil Yenius 4

CI2C: Curso de Ingeniería Inversa para Chainiks. Parte I
por Wh1t3 D3m0N 10

Introducción al Hackeo de Routers. Puerto Serie
por Mike Anic 16

CI2C: Curso de Ingeniería Inversa para Chainiks. Parte II
por Wh1t3 D3m0N 23

Conversaciones Cibernéticas
por Signul Floid 26

Nuestro Primer Polyglot
por Occam's Razor 31

Compilación, Compilación Cruzada y ToolChains
por Andrés "Andy" Pajuaker 32

Los Truquis de Occam's
por Tamariz el de la Perdiz 38

ESTA PUBLICACIÓN HA SIDO CREADA UTILIZANDO EXCLUSIVAMENTE SOFTWARE LIBRE:

Revista Online
Occam's Razor
Número 2 Nueva Época
Abril 2017

Dirección y Edición
David Martínez Oliveira

Colaboradores
Yvil Yenius
Wh1t3 D3m0N
Mike Anic
Signul Floid
Andrés "Andy" Pajuaker
Tamariz el de la Perdiz

Maquetación
DeMO

EDITORIAL

Un nuevo número con Sorpresilla!

por Occam's Razor

El número 2 de la nueva época de Occam's Razor ya está listo. Aquí mismo lo tienes. Esta vez hemos tardado un poco menos de 1 año en conseguir sacar un nuevo número de esta vuestra humilde publicación. Este número viene con algunas sorpresas... alguna cosa tenemos que dejar para aquellos de vosotros que seguís el blog y leéis la revista en *Tiempo Real* :). Dejad que os lo contemos en detalle.

Recientemente hemos descubierto el, totalmente alucinante, diario PoC |GTFO. Se trata de una publicación en inglés sobre ingeniería inversa. No podemos más que recomendarla como una lectura obligatoria para to buen geek. Además de unos contenidos muy recomendables, cada número de la revista es lo que sus autores llaman un *poligot*. En lenguaje plano, el PDF que os descargáis es un PDF pero también es otra cosa. Estos tipos son unos genios y os podéis encontrar desde una imagen ISO con un tetris, a una canción bastante geek.

En cualquier caso, la idea nos a parecido genial y nos la hemos apropiado en este número, con una implementación mucho más humilde dentro de nuestras posibilidades. Todos los detalles sobre el proceso los podéis encontrar en la revista.
A parte de eso, en este número *Wh1t3 D3M0n*, un nuevo colaborador, comienza un cursillo sobre ingeniería inversa. Empieza fuerte con 2 entregas en este mismo número. Yvil continua con sus planes para dominar el mundo, y Mike nos cuenta los primeros pasos para poder hackear routers y otros cacharros.

Además Signul nos cuenta como configurar PocketSphinx para que funcione en Español y Andy nos introduce en los detalles del proceso de configuración. Como siempre Tamariz... si, si, el de la Perdiz nos cuenta como hacer brujerías con la línea de comandos en su sección de trucos.

Para terminar, como podéis ver hemos editado de nuevo este número en formato revista... En esta ocasión hemos utilizado Scribus. La experiencia ha sido bastante satisfactorio, pero todavía no estamos totalmente convencidos... editar la revista directamente en LaTeX es mucho más güay. Como siempre vuestra opinión es importante, así que no dudéis en escribirnos para contarnos vuestras preferencias.

Esto es todo. Esperamos que os guste este número!

El Equipo
de Occam's Razor

Mensajes Clandestinos

Una breve introducción a la Esteganografía

por Yvil Yenius

Ahora que ya tenemos nuestro servidor clandestino en funcionamiento y podemos ponernos en contacto con nuestros secuaces de una forma fácil y super-secreta, es hora de idear un método para que nuestros mensajes de dominacón del mundo sea también clandestino... es más, que esos mensajes diabólicos

Para conseguir esto, vamos a utilizar una antigua técnica ideada por los griegos, conocida como esteganografía [1]. Así, a lo burro, la esteganografía nos permite ocultar una cosa, dentro de otra, de forma que la cosa oculta no sea perceptible a los ojos de la gente. Sí, la técnica se aplica en general a cualquier cosa, pero nosotros la vamos a utilizar para poder intercambiar mensajes secretos entre nuestros secuaces... sin que nadie se de cuenta.

Gatitos!

Puede que no lo sepas, pero los gatos son animales diabólicos, tradicionalmente relacionados con el mal y con el ocultismo y lo sobrenatural. Esas criaturas ya se han adueñado de Internet. Están por todas partes.... y todo el mundo los adora.

Así que, lo que vamos a hacer es utilizar fotos de gatitos para ocultar nuestros mensajes secretos y así poder dirigir nuestras hordas de secuaces en nuestra épica empresa para conquistar el mundo.

Como? ...

... os preguntaréis. Muy fácil, vamos a modificar sutilmente los bits bajos de cada pixel de una adorable imagen de un gatito y utilizarlos para almacenar nuestros mensajes secretos.

Como sabéis una imagen se puede ver como una matriz en la que cada entrada representa el color del punto correspondiente. Lo que llamamos pixels en la pantalla. Bueno, hay varias formas de codificar el color de cada pixel, y la que nos viene mejor para nuestro cometido es la conocida como **True Color**.

Una imagen en format **True Color**, utiliza 24 o 32 bits (3 o 4 bytes) por cada pixel. Los tres primeros bytes representan la intensidad de cada una de las componentes de color del pixel: Rojo, Verde y Azul, o como dirían los anglosajones Red, Green and Blue (RGB). El cuarto byte, si existe, se utiliza para indicar la transparencia del pixel.

Así que lo que vamos a hacer es:

- Coger una imagen en formato True Color

• Cada carácter de nuestro mensaje la representaremos con un byte, de hecho, solo necesitamos 7 bits para poder escribir mensajes sin acentos. Después de todo, a quién le interesa la ortografía?.

• Esos 7 bits los vamos a "esparramar" por las componentes del pixel. Almacenaremos 3 bits en la componente azul, 2 bits en la verde y 2 bits en la roja. De esta forma podemos almacenar una letra por pixel

```
1234567
LLLLLLL

123456_78    123456_78    12345_678
RRRRRR_RR    GGGGGG_GG    BBBBB_BBB
RRRRRR_LL    GGGGGG_LL    GGGGG_LLL
123456_12    123456_34    12345_567
```

Un programita

Bueno, ahora que ya sabemos lo que queremos hacer, escribamos un programa para que nuestra brillante idea se haga realidad. Lo llamaremos (al programa) el_codificador_clandestino... bueno, mejor ecc que así tenemos que escribir menos.

Nuestro pequeño programa usa una librería llamada libgd. Esta librería, es un clásico para generar imágenes y fue muy utilizada en páginas web dinámicas al principio de Internet (todavía se usa, pero ya no tanto). En cualquier caso... usamos libgd... y usamos goto.... sí, somos muy malos!. Podríamos evitar el goto fácilmente... pero después de todo no se puede dominar el mundo sin hacer cosas diabólicas... no?

El programa lee una imagen en memoria y luego un fichero de texto, el cual procesa carácter a carácter "esparramando" los bits de cada letra en los píxeles de la imagen. Una vez que se han procesado todos los ficheros volcamos la imagen en el disco y listo.

> Modificando los bits bajos de cada pixel podemos almacenar mensajes en una imagen

Como habréis comprobado, si especificamos como fichero de texto el carácter -, el programa leerá la entrada estándar. Pulsad, CONTROL + D cuando hayáis terminado de escribir.

Otro Programita

Bueno, para poder probar si nuestro *Codificador Clandestino* funciona, tendremos que programar.... Si, en un despilfarro de originalidad, lo hemos llamado edc. Ya sabéis, para escribir menos.

Este es el código, básicamente deshace el "esparramao" del ecc.

Bueno, no hay mucho que rascar no?. Extraemos las componentes de cada pixel en la imagen y juntamos los bits que nos hacen falta para recomponer nuestro carácter original. Cuando el

LISTADO 1. El Codificador Clandestino

```
#include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>
#include  <gd.h>

int
main(int argc, char *argv[])
{
  FILE        *f;
  gdImagePtr  image;
  int         i, l, x, y, w, h,k;
  int         size;
  int         rgb, color, r, g, b, cr, cg, cb, c, cnt;

  if (argc < 3)  {
      fprintf (stderr,
              "usage:%s input_image.png file.txt\n", argv[0]);
      exit (1);
    }
  f = fopen (argv[1], "rb");

  image = gdImageCreateFromPng (f);
  w = image->sx;
  h = image->sy;
  size = w * h;
  fprintf (stderr, "Tamaño: %dx%d (Capacidad: %d bytes)\n",
             w , h, size);
  fclose (f);

  if (argv[2][0] == '-')
    f = stdin;
  else
    f = fopen (argv[2], "rt");

  l = 0;
  cnt = 0;
  for (y = 0; y < h; y++) {
      for (x = 0; x < w; x++) {
            if (!l)   {
                fread (&c, 1, 1, f);
                if (feof (f))
                    { l = 1 ; c=0;}
                cnt++;
              }
            else
              goto listo;

            /* Get pixel value */
            rgb = gdImageGetPixel (image, x, y);

            r = ((gdImageRed(image, rgb)) & 0xfc )
              + (c & 0x03);
            g = ((gdImageGreen(image, rgb)) &0xfc)
              + ((c >> 2) & 0x03);
            b = (gdImageBlue(image, rgb) & 0xf0)
              + ((c >> 4) & 0x0f);
            color = gdTrueColor (r, g, b);
            gdImageSetPixel (image, x, y, color);
            i++;
          }
    }
 listo:
  gdImagePng (image, stdout);
  fprintf (stderr, "Encoded %d bytes out of %d bytes "
           "(%6.2f %% used)\n",
             cnt, size, (float)cnt * 100.0 / (float)(size));
}
```

LISTADO 2. El Decodificador Clandestino

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <gd.h>

char *s = "This is a secret message in a image!!!\n\0";

int
main(int argc, char *argv[])
{
  FILE        *f;
  gdImagePtr  image;
  int         i, l, x, y, w, h;
  int         rgb, color, r, g, b, cr, cg, cb, c;

  if (argc != 2)
    {
      fprintf (stderr, "usage:%s input_image.png\n", argv[0]);
      exit (1);
    }

  f = fopen (argv[1], "rb");

  image = gdImageCreateFromPng (f);
  w = image->sx;
  h = image->sy;
  fprintf (stderr, "+ Tamaño: %dx%d\n", w, h);
  fclose (f);

  i = 0;
  l = strlen (s);
  for (y = 0; y < h; y++)
    {
      for (x = 0; x < w; x++)
          {
            rgb = gdImageGetPixel (image, x, y);

            r = gdImageRed(image, rgb);
            g = gdImageGreen(image, rgb);
            b = gdImageBlue(image, rgb);
            c = (r & 0x03) + ((g & 0x03) << 2) + ((b &0x0f )<< 4);
            if (c == 0) exit (1);
            putchar (c);
          }
    }

}
```

FIGURA 1. Nuestra imagen contenedora.... Que kuki!
Fuente WikiMedia [2] Licencia: CC-BY-SA

resultado sea 0, paramos.

Vamos a probarlo.

Gatitos Clandestinos

Como adelantamos al principio, la mejor forma de enviar mensajes esteganográficos clandestinos es utilizando fotos de gatitos... Están hasta en la sopa, así que nadie va a desconfiar de esas fotos con adorables y pizperetas criaturas... Nosotros vamos a utilizar la que os mostramos en la Figura 1... que mono... ooohh:

Ahora escondamos un mensaje misterioso en la foto de nuestro gatito:

```
$ ecc el_gatito.png -  >  el_gatito_clandestino.png
+ Tamaño: 640x480 (Capacidad: 307200 bytes)
Este es un mensaje super clandestino para todos
nuestros secuaces
Esta tarde a las 5 birras en el Bar Gas Iosa
^D
+ Codificados 112 bytes de 307200 bytes disponibles
(  0.04 % usado)
```

Por cierto, nos vemos en el Bar Gas Iosa a eso de las 5. Vale?

Por si alguien tiene alguna duda, la forma de extraer el mensaje clandestino sería algo tal que así:

```
$ edc el_gatito_clandestino.png
+ Tamaño: 640x480
Este es un mensaje super clandestino para todos
nuestros secuaces
Esta tarde a las 5 birras en el Bar Gas Iosa
```

Se ve que funciona que te pasas... ahora junto con lo que hemos aprendido sobre Servidores Clandestinos [4] ya podríamos escribir un programa super secreto de chat clandestino... Lo llamaríamos: ***Le Chat Cachᵉ***.

Detectando Mensajes Clandestinos

Ahora que podemos esconder nuestros mensajes dentro de imágenes, no estaría mal saber como poder averiguar si una imagen contiene algún mensaje oculto. Esto es algo bastante sencillo, cuando el criptoanalista dispone de la imagen original...

En estos casos, la cosa es muy fácil, resta ambas imágenes, y si el resultado no es cero, es que algo hay ahí. Para nuestro ejemplo del gatito de más arriba, podéis probarlo siguiendo los pasos que enumeramos a continuación:

- Cargad ambas imágenes en GIMP
- Pegad una sobre la otra como una nueva capa

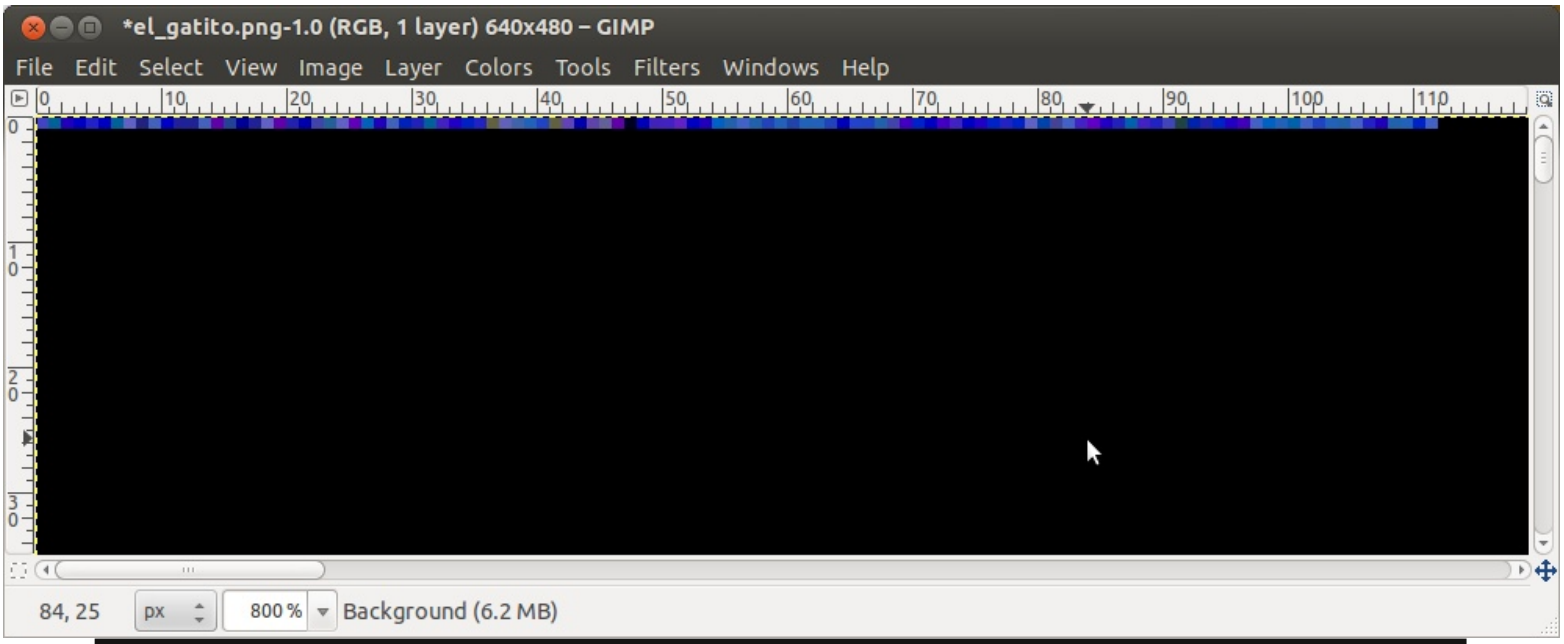


FIGURA 2. Detectando mensajes clandestinos con GIMP. Zoom al 800%

- En la lista de modos seleccionar **Diferencia**
- Mezclad las capas visibles
- Seleccionar **Ajuste de Curvas de color** en el menu **Colores**
- Moved la gráfica hacia arriba y... Voilá. Deberíais ver algo como esto

La Figura 2 muestra el resultado de este proceso.

Como podéis apreciar, toda la imagen es negra, excepto una parte de la primera línea, que, vaya por dios, es justo donde nuestro mensaje clandestino está almacenado.

Pero... y si no tenemos la imagen original?

Detectando y II

Bien, si no tenemos la imagen original las cosas son un poco más complicadas, especialmente si los mensajes son muy cortos en relación con el tamaño de la imagen... lo cual, por otra parte, es un desperdicio de ancho de banda... pero bueno.... somo malos no?

Así que lo que vamos a hacer es generar un fichero con un mensaje de mayor tamaño. Vamos a ocultar a nuestro gatito dentro de otro gatito. El segundo gatito tiene que ser una imagen mucho mayor que la primera. Obviamente.

Detectar mensajes esteganográficos es posible analizando la entropía del mensaje!

Debido a que nuestro codificador clandestino es muy cutre y solo funciona con cadenas de texto, convertiremos nuestra imagen en texto para poder codificarla. Base64 al rescate!

```
$ base64 el_gatito.png | ./ecc el_gatazo.png -  >
el_gatazo_clandestino.png
$  ./edc el_gatazo_clandestino.png | base64 -d >
el_gatito_secreto.png
```

Ahora, lo que haremos será escribir un sencillo programa que lo que hará será modificar la imagen de forma que los bits más bajos de cada componente de cada pixel (donde almacenamos nuestro mensaje) se copien en los más altos. Vamos estamos amplificando los bits de menor peso de la imagen.

Podéis ver el código en el Listado 3.

Hemos escogido solo la componente verde por que es la que mejor se ve, y en este caso es suficiente para mostrar lo que queremos.

El Mensaje Clandestino al Descubierto

Vamos allá. La imagen orginal del gatazo la podéis encontrar en la Figura 3. Suficientemente grande para poder desparramar a nuestro gatito. Bien!

Cuando pasamos esta imagen por nuestro programa detector obtenemos como resultado la Figura 4.

Pero, si generamos nuestra imagen del gatazo clandestino como os hemos contado más arriba y ejecutamos nuestro programa detector... veríamos algo como la Figura 5.

LISTADO 3. Amplificador LSB

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <gd.h>

int
main(int argc, char *argv[])
{
  FILE        *f;
  gdImagePtr  image;
  int         i, l, x, y, w, h,k;
  int         size;
  int         rgb, color, r, g, b, cr, cg, cb, c, cnt;

  if (argc < 2)
    {
      fprintf (stderr, "usage:%s input_image.png\n",
argv[0]);
      exit (1);
    }

  f = fopen (argv[1], "rb");

  image = gdImageCreateFromPng (f);
  w = image->sx;
  h = image->sy;
  fclose (f);

  for (y = 0; y < h; y++)
    {
      for (x = 0; x < w; x++)
       {
         /* Get pixel value */
         rgb = gdImageGetPixel (image, x, y);

         r = g =b =0;

         g = ((gdImageGreen(image, rgb)) &0x03) << 6;

         color = gdTrueColor (0, g, 0);
         gdImageSetPixel (image, x, y, color);
       }
    }

  gdImagePng (image, stdout);
}
```

Como podéis ver, hay algo al principio del fichero que parece distinto al resto de la imagen, y que ocupa unos 2/3 de la imagen original. Ahora ya es cosa del pobre hombre que lo esté analizando, el encontrar como sacar lo que sea que está escondido ahí.

Detalles, detalles

Antes de concluir algunos detalles. La técnica que hemos utilizado se conoce como modificación LSB (least significat bit) y que consiste precisamente en modificar los bits más bajos de los pixels (eso no lo esperabais eh?). La idea es que un cambio en esos bits produce una variación de color tan pequeña que el ojo no percibe ninguna diferencia.

Precisamente por que la técnica modifica los bits bajos, amplificarlos (copiarlos en la parte alta), va a poner de manifiesto las diferencias entre la pixels que han sido modificados y los que no. En realidad dependerá de las propiedades estadísticas de ambos grupos de datos, pero en general alguna pista nos dará. Hay otros test más sofisticados, pero no vamos a hablar de ellos en este artículo.

Amplificando el LSB de cada pixel de la imagen, podemos "ver" el mensaje oculto

Como os podéis imaginar, esta técnica funciona mejor con imágenes realistas. Imágenes con areas de color plano, como dibujos animados, logotipos, o similares van a hacer más fácil detectar esas variaciones de color, incluso para el ojo desnudo.

Por último, esta técnica, para funcionar, necesita que la imagen se almacene con un formato sin compresión, o con compresión pero sin pérdidas. Formatos de compresión con pérdidas como JPG van a cambiar los colores de los píxeles y por tanto destruir nuestro mensaje oculto.

COLOFON

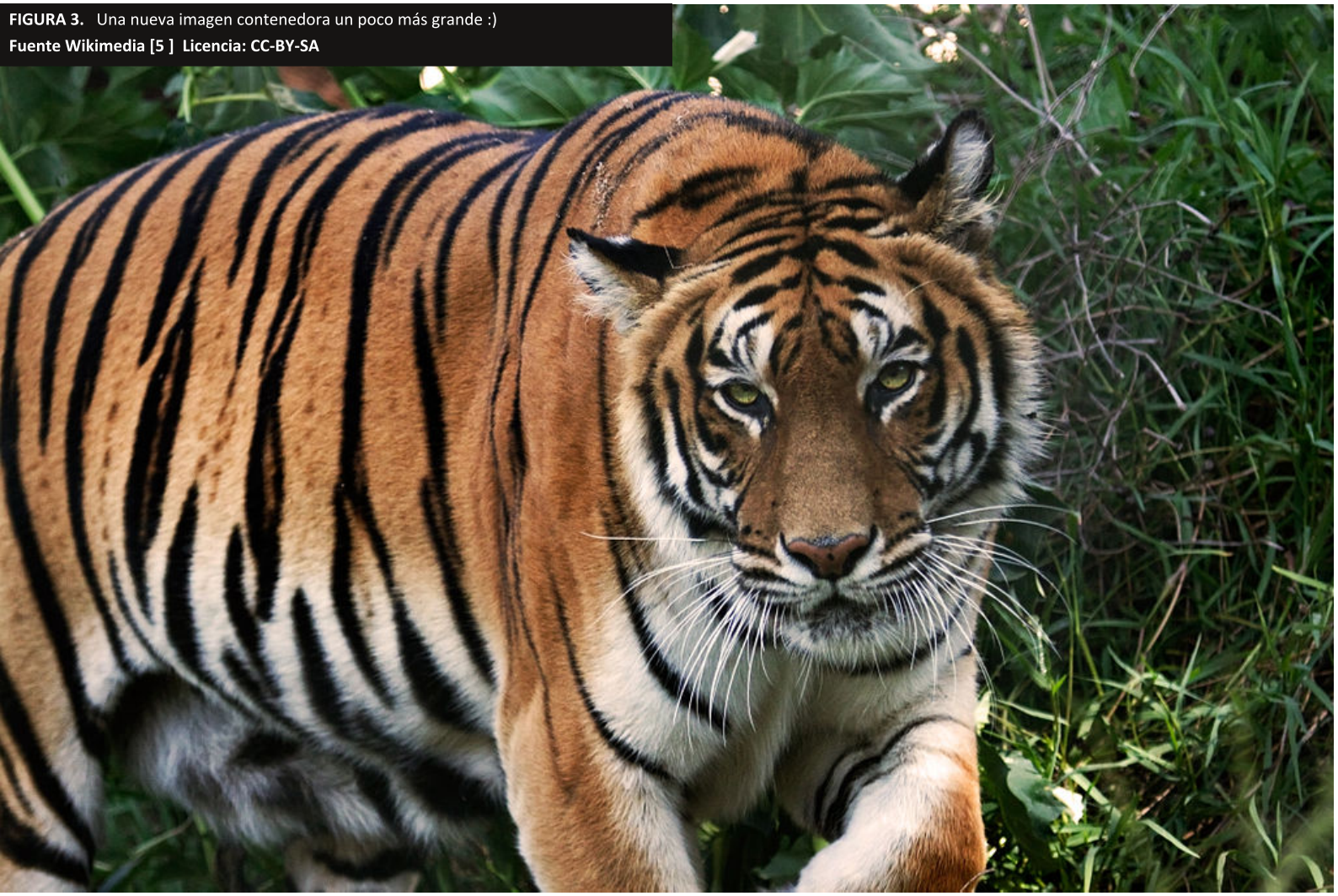
Un pequeño paso para el genio diabólico, pero un gran paso para la dominación del mundo. Ahora

Sobre el Autor

Yvil Yenius

Cuando se trata de dominar el mundo, Yvil es la referencia. Su privilegiada mente para el mal y su carisma para ganarse la lealtad de secuaces a diestro y siniestro, la han convertido en una de las más grandes mentes malvadas de la historia. En su tiempo libre, hornea deliciosas tartas que lanzar a sus enemigos.

FIGURA 3. Una nueva imagen contenedora un poco más grande :)
Fuente Wikimedia [5] Licencia: CC-BY-SA



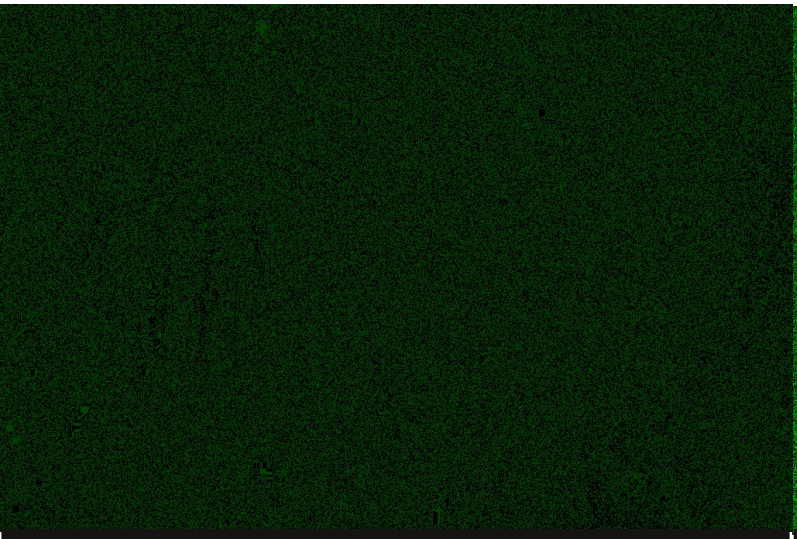


FIGURA 4. Nuestro detector sobre la imagen orignal

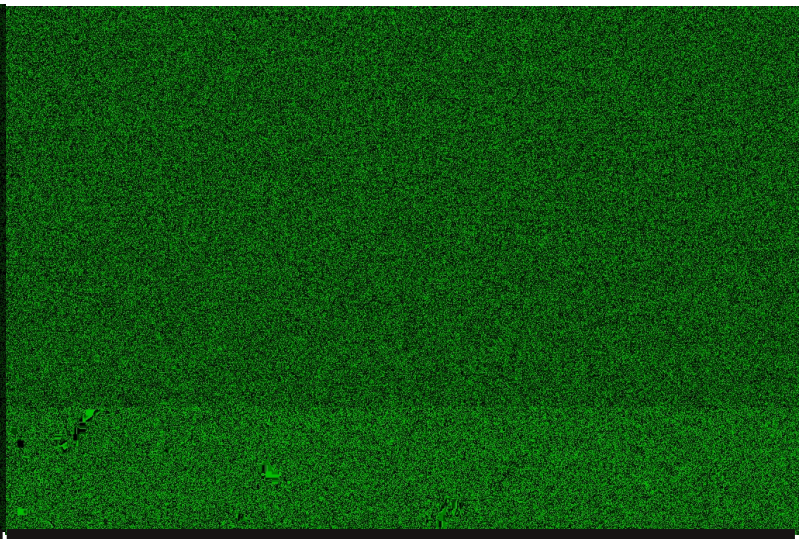


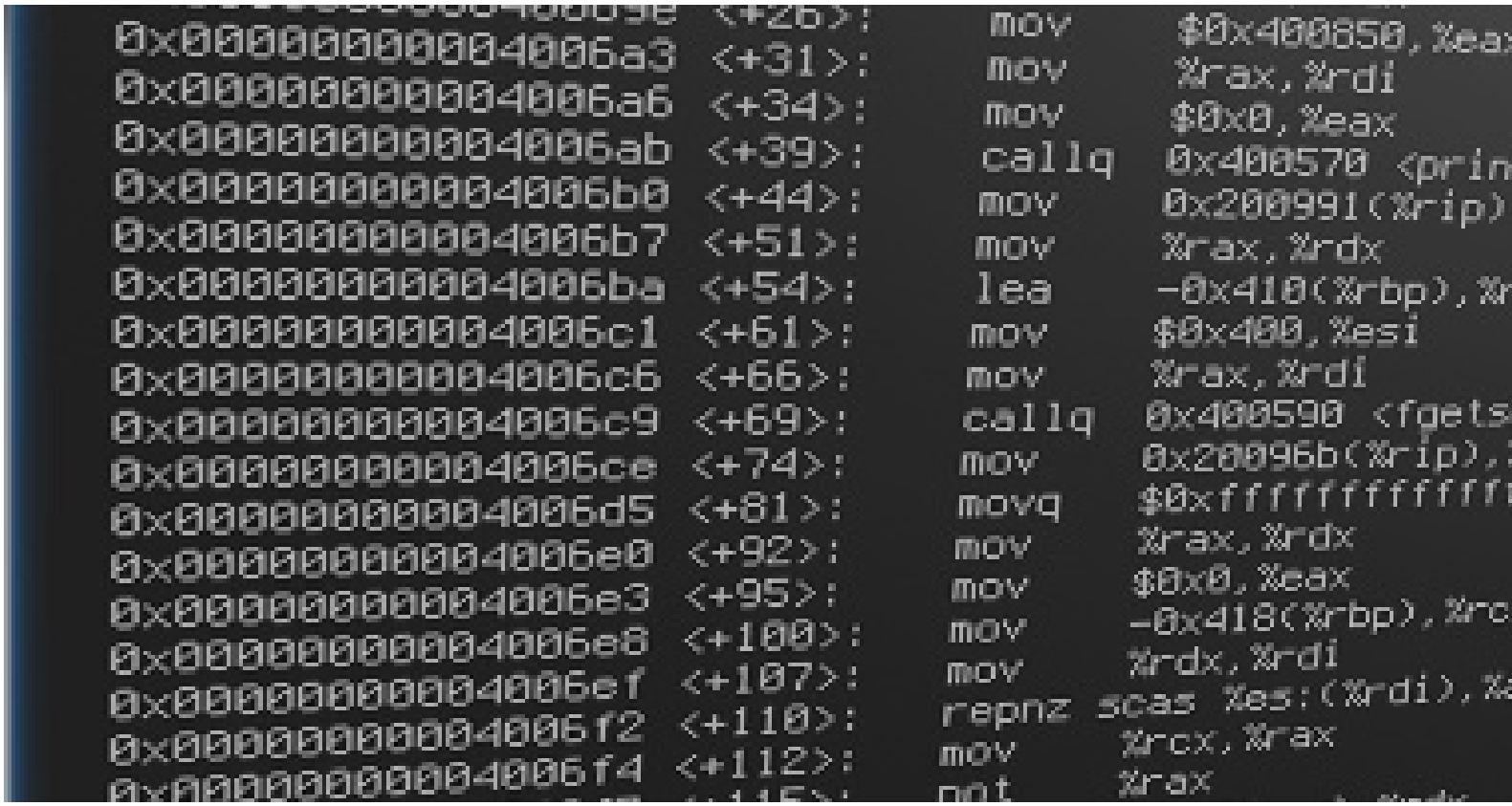
FIGURA 5. Nuestro detector sobre la imagen con le mensaje oculto

Enlaces Externos

[1] https://es.wikipedia.org/wiki/Esteganograf%C3%ADa
[2] https://commons.wikimedia.org/wiki/File:Gatito-1024x768.jpg
[3] https://creativecommons.org/licenses/by-sa/4.0/deed.en
[4] http://www.papermint-designs.com/roor/2016-01-servidor-clandestino
[5] https://commons.wikimedia.org/wiki/File:Tigre_de_Bengala-Panthera_tigris_tigris.jpg

Código Fuente :

http://papermint-designs.com/roor/download/mensajes_clandestinos.tgz



CI2C:Curso de Ingeniería Inversa para Chainiks. Parte I

Adéntrate en el misterioso mundo de la Ingeniería Inversa

por Wh1t3 D3m0N

Bienvenidos a la primera parte de el curso acelerado de ingeniería inversa para chainiks. En este curso aprenderéis todo lo que se necesita saber para convertirse en un experto ingeniero inverso... y eso no tiene nada que ver con vuestras tendencias sexuales.

Empecemos por el principio. La ingeniería inversa es la disciplina que estudia las técnicas necesarias para averiguar como funciona un determinado sistema. En general, veréis el término aplicado al SW, pero en realidad, podéis hacer ingeniería inversa sobre un dispositivo mecánico, un dispositivo electrónico o un algoritmo. En cierto modo, el cryptoanálisis se puede ver como una forma de ingeniería inversa.

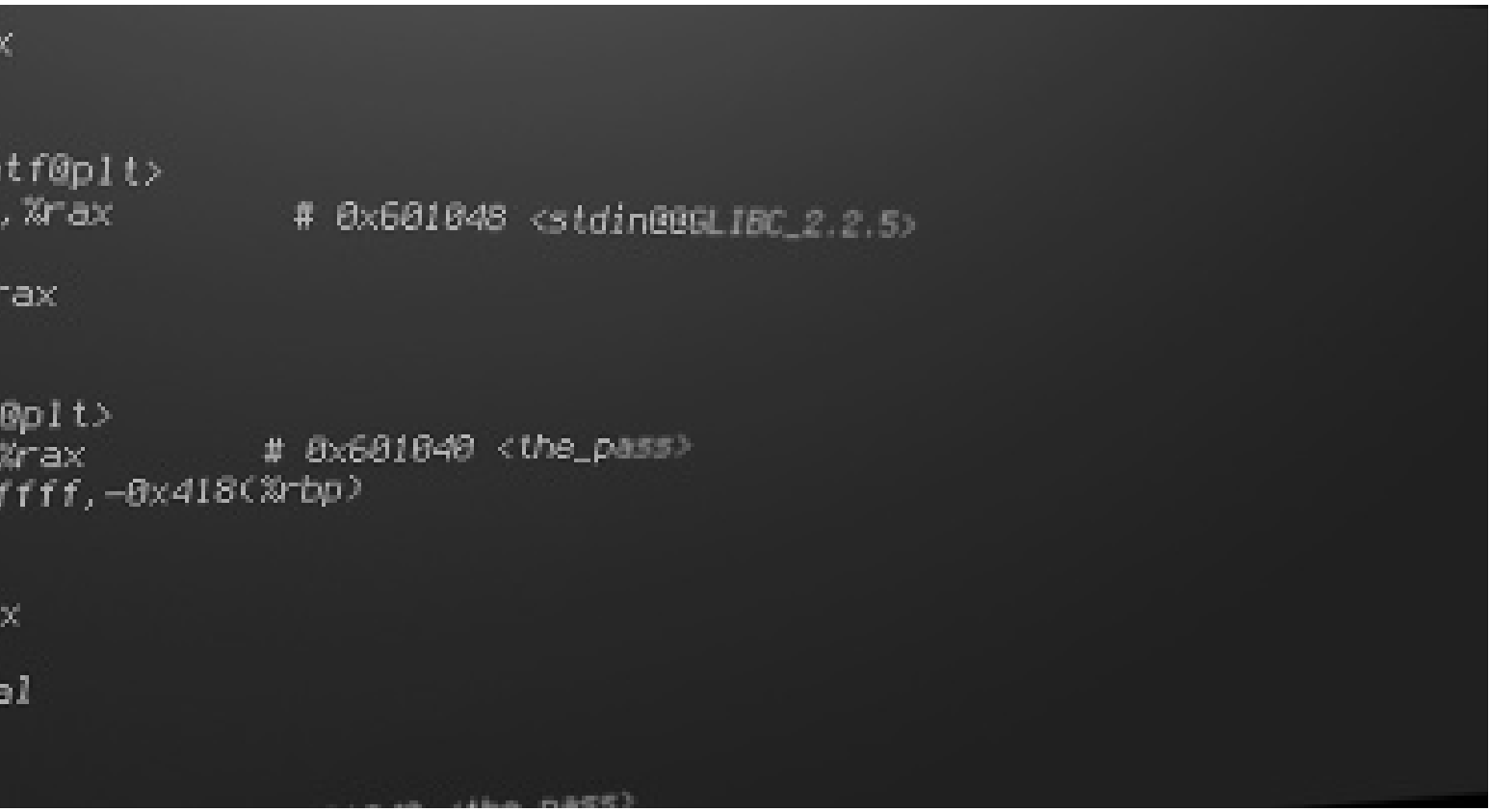
Podéis verlo, si queréis como el proceso contrario a la producción de algo. Obtener la información de diseño a partir de una implementación.

Nuestra intención es que este curso sea eminentemente práctico. Con poca teoría. La suficiente para que no seáis unos Script Kiddie [2], pero no demasiada para aburriros.

Para este curso necesitaréis tener una base de algunas cosas, de lo contrario, seamos sinceros, no os vais a enterar de mucho:

- Conocimientos básicos de programación en C
- Conocimientos básicos de programación en Ensamblador
- Conocimientos básicos sobre procesos y formatos ejecutables en GNU/Linux
- ... otras cosas que ahora no se me ocurren, pero que seguro que os harán falta.

Bueno, suficiente introducción. Vamos al tajo a ver que tal se nos da. Si esto no funciona, probaremos con el Duero o el Ebro :P



El típico Crackme 0

Todos los desafíos de ingeniería inversa siempre comienzan con un desafío muy sencillo con el que ponerse manos a la obra. Este desafío se puede superar sin hacer mucha ingeniería inversa, pero es una especie de "Hola Mundo" de esta disciplina.

El desafío consiste en acceder a una cierta parte de un programa protegida por una contraseña. Este sería el típico programa en C al que nos tendremos que enfrentar:

```
#include  <stdio.h>
#include  <string.h>
char *the_pass = "MyVoiceIsMyPassport";

int
main (void)
{
  char   user_pass[1024];

  printf ("Introduce la contraseña: ");
  fgets (user_pass, 1024, stdin);
  if (!strncmp (user_pass, the_pass,
strlen(the_pass)))
    {
      printf ("Bien Hecho!\n");
    }
  else
    printf ("Sigue Intentándolo\n");
}
```

> Un Crackme es un programa diseñado para ser crackeado a modo de práctica

Nuestro objetivo es que el programa genere como salida el mensaje "Bien Hecho!". Esto lo podemos hacer de tres formas:

1. Averiguando la contraseña
2. Crackeando el programa para que siempre muestre el mensaje incluso cuando la contraseña no es correcta
3. Cambiando la contraseña almacenada en el programa por una de nuestra elección

Veamos como conseguir acceder al mensaje que nos interesa en cada unos de estos tres casos:

Averiguando la contraseña

Hay dos formas fundamentales de averiguar la contraseña en este caso. Esto es así porque la contraseña está almacenada como texto plano en el fichero compilado, y por lo tanto es visible. Para obtener las cadenas de caracteres *imprimibles* almacenadas en un cierto fichero podemos utilizar el comando strings. De hecho, esto es algo que deberéis probar siempre ya que suele desvelar información interesante.

Copiad el código de nuestro programa ejemplo más arriba, y copiadlo en un fichero llamado c1.c. Ahora compilemos el programa:

```
$ make c1
```

Y veamos que tiene que decirnos strings sobre este ejecutable:

```
$ strings c1
(... mogollón de líneas aquí)
|$0H
MyVoiceIsMyPassport
Introduce la contrase
Bien Hecho!
Sigue Intent
ndolo
;*3$"
GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
.symtab
(... más líneas aquí ....)
```

Como podéis observar en medio de la salida de strings podemos ver todas las cadenas de caracteres que utiliza el programa, incluyendo la clave que buscamos. Llegados a este punto probaríamos las claves candidatas a ver si alguna funciona.

Para filtrar un poco la salida de strings podemos usar grep con el flag -C y una de las cadenas que sabemos se encuentran en el fichero. Por ejemplo "Sigue". El flag -C debe ir seguido de un nímero que indica cuantas líneas antes y después de la cadena buscada deben mostrarse. Por ejemplo con un valor de 5 obtendríamos la siguiente salida:

```
$  strings c1 | grep -C 5 Sigue
t$(L
|$0H
MyVoiceIsMyPassport
Introduce la contrase
Bien Hecho!
Sigue Intent
ndolo
;*3$"
GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
.symtab
.strtab
```

Averiguando la contraseña II

Si habéis prestado atención, os estaréis preguntando... cuál es la otra forma de averiguar la contraseña de este programa?... Bueno, pues haciendo ingeniería inversa. Sí esta vez si.

Para ello, deberemos desensamblar el programa. Destriparlo. Hurgar en sus tripas para encontrar lo que estamos buscando.... Sin piedad... ARRGGG!!. Bueno ya estoy más relajado.

Para desensamblar nuestro programa podemos utilizar diferentes herramientas. Para este primer ejemplo vamos a ir con algo intermedio... gdb

Primero abrimos el programa con gdb

```
$ gdb ./c1
```

Ahora vamos a poner un breakpoint en la función main. Un breakpoint nos permite parar la ejecución del programa en el punto que nosotros queramos, y utilizar el desensamblador para examinar el proceso.

```
(gdb) b main
Breakpoint 1 at 0x400688
```

La dirección que aparece en la respuesta de gdb puede ser diferente en vuestro sistema. Ahora podemos ejecutar el programa. El cual se parará al principio de la función main.

```
(gdb) r
Starting program: /home/wd/ci2c/c1

Breakpoint 1, 0x0000000000400688 in main ()
(gdb)
```

Ahora echemos un ojo al ensamblador de la función main.

El resultado en la Figura 1.

El depurador se convertirá en tu fiél amigo muy pronto. gdb es el depurador GNU... Será mejor que empecéis a conoceros

---

Wow, wow, woow... para el carro tío. Si, impresiona no?. Pero es mucho menos de lo que parece. Echadle un ojo al código, y veréis que en la dirección 0x0400712 encontramos la llamada a strncmp. Esa es la función que compara la contraseña que introducimos desde el teclado, con la que tenemos almacenada en nuestro programa.

Además, como no hemos eliminado los símbolos de este binario, gdb nos muestra incluso la dirección de memoria en la que se encuentra la variable the_pass (0x601040)... Echemos un ojo:

```
(gdb) x/s *0x601040
0x40083c:      "MyVoiceIsMyPassport"
```

Bien, el comando x nos permite hacer un volcado de memoria. Después de la barra podemos indicarle cuantos elementos queremos que vuelque y en que formato. En este caso hemos indicado 's' que significa que queremos ver el volcado como una cadena de caracteres. Podríamos haber usado x/20x *0x601040 para volcar veinte bytes en hexadecimal. Escribid help x en gdb para obtener más información sobre lo que es posible.

Bueno, hemos encontrado la clave esta vez usando el ensamblador y analizando los parámetros con los que llamamos a una función. Bueno, esto no lo hemos explicado todavía, pero en realidad es lo que hemos hecho... Por el momento deberéis hacer un pequeño acto de fé.

Crackeando el programa

Ya hemos visto dos formas con las que averiguar la clave que el programa espera que introduzcamos. Ahora es el momento de crackearlo para que de igual la clave que introduzcamos. Para ello tendremos que fijarnos más atentamente en el ensamblador. Especialmente en estas lineas:

```
0x0000000000400712 <+142>: callq  0x400540 <strncmp@plt>
0x0000000000400717 <+147>: test   %eax,%eax
0x0000000000400719 <+149>: jne    0x400727 <main+163>
0x000000000040071b <+151>: mov    $0x40086b,%edi
0x0000000000400720 <+156>: callq  0x400550 <puts@plt>
0x0000000000400725 <+161>: jmp    0x400731 <main+173>
0x0000000000400727 <+163>: mov    $0x400877,%edi
0x000000000040072c <+168>: callq  0x400550 <puts@plt>
```

FIGURA 1. DESENSAMBLADO DE LA FUNCION main

```
(gdb) disassem
Dump of assembler code for function main:
   0x0000000000400684 <+0>:   push   %rbp
   0x0000000000400685 <+1>:   mov    %rsp,%rbp
=> 0x0000000000400688 <+4>:   sub    $0x420,%rsp
   0x000000000040068f <+11>:  mov    %fs:0x28,%rax
   0x0000000000400698 <+20>:  mov    %rax,-0x8(%rbp)
   0x000000000040069c <+24>:  xor    %eax,%eax
   0x000000000040069e <+26>:  mov    $0x400850,%eax
   0x00000000004006a3 <+31>:  mov    %rax,%rdi
   0x00000000004006a6 <+34>:  mov    $0x0,%eax
   0x00000000004006ab <+39>:  callq  0x400570 <printf@plt>
   0x00000000004006b0 <+44>:  mov    0x200991(%rip),%rax        # 0x601048 <stdin@@GLIBC_2.2.5>
   0x00000000004006b7 <+51>:  mov    %rax,%rdx
   0x00000000004006ba <+54>:  lea    -0x410(%rbp),%rax
   0x00000000004006c1 <+61>:  mov    $0x400,%esi
   0x00000000004006c6 <+66>:  mov    %rax,%rdi
   0x00000000004006c9 <+69>:  callq  0x400590 <fgets@plt>
   0x00000000004006ce <+74>:  mov    0x20096b(%rip),%rax        # 0x601040 <the_pass>
   0x00000000004006d5 <+81>:  movq   $0xffffffffffffffff,-0x418(%rbp)
   0x00000000004006e0 <+92>:  mov    %rax,%rdx
   0x00000000004006e3 <+95>:  mov    $0x0,%eax
   0x00000000004006e8 <+100>: mov    -0x418(%rbp),%rcx
   0x00000000004006ef <+107>: mov    %rdx,%rdi
   0x00000000004006f2 <+110>: repnz scas %es:(%rdi),%al
   0x00000000004006f4 <+112>: mov    %rcx,%rax
   0x00000000004006f7 <+115>: not    %rax
   0x00000000004006fa <+118>: lea    -0x1(%rax),%rdx
   0x00000000004006fe <+122>: mov    0x20093b(%rip),%rcx        # 0x601040 <the_pass>
   0x0000000000400705 <+129>: lea    -0x410(%rbp),%rax
   0x000000000040070c <+136>: mov    %rcx,%rsi
   0x000000000040070f <+139>: mov    %rax,%rdi
   0x0000000000400712 <+142>: callq  0x400540 <strncmp@plt>
   0x0000000000400717 <+147>: test   %eax,%eax
   0x0000000000400719 <+149>: jne    0x400727 <main+163>
   0x000000000040071b <+151>: mov    $0x40086b,%edi
   0x0000000000400720 <+156>: callq  0x400550 <puts@plt>
   0x0000000000400725 <+161>: jmp    0x400731 <main+173>
   0x0000000000400727 <+163>: mov    $0x400877,%edi
   0x000000000040072c <+168>: callq  0x400550 <puts@plt>
   0x0000000000400731 <+173>: mov    -0x8(%rbp),%rdx
   0x0000000000400735 <+177>: xor    %fs:0x28,%rdx
   0x000000000040073e <+186>: je     0x400745 <main+193>
   0x0000000000400740 <+188>: callq  0x400560 <__stack_chk_fail@plt>
   0x0000000000400745 <+193>: leaveq
```

Vemos que tras ejecutar strncmp, el programa comprueba el valor del registro eax. Si el resulta de ese test es distinto de cero (jne *Jump Not Equal*) saltaremos al offset +163 y llamaremos a puts (que imprime una cadena de caracteres en la pantalla) para que imprima el contenido de 0x400877, sino, imprimiremos el contenido de 0x49986b. Veamos que tenemos en esas dos direcciones:

```
(gdb) x/s 0x40086b
0x40086b:     "Bien Hecho!"
(gdb) x/s 0x400877
0x400877:     "Sigue Intentándolo"
```

Así que lo que queremos es que el código en el offset +151 sea el que se ejecute siempre. Para ello podemos hacer varias cosas. Cambiar el salto condicional por un salto absoluto, cambiar la operación (test %eax, %eax) de forma que el salto se comporte siempre como queramos. O simplemente eliminar el salto. La solución dependerá del programa, de cuantos bytes de memoria podamos escribir y de cuantos bytes de memoria requiera la instrucción que queremos insertar.

En este caso, lo más sencillo es eliminar el salto (jne). Para ello sustituiremos todos sus bytes por NOPs. El comando NOP no hace nada, es decir, estamos borrando el jne de memoria.

Así que lo que tenemos que hacer es encontrar ese salto en el fichero y cambiar el código máquina asociado por una secuencia de instrucciones NOP.

Parcheando un binario

Vamos a dejar gdb por un momento y volver a nuestra línea de comandos para parchear el programa. Lo primero que tenemos que hacer es averiguar cual es el código máquina asociado al salto que queremos eliminar. Para ello vamos a utilizar la utilizad objdump. Veamos como:

```
$  objdump -d c1 | grep -C 5 "jne.*400727"
  400705:       48 8d 85 f0 fb ff ff    lea    -0x410(%rbp),%rax
  40070c:       48 89 ce                mov    %rcx,%rsi
  40070f:       48 89 c7                mov    %rax,%rdi
  400712:       e8 29 fe ff ff          callq  400540 <strncmp@plt>
  400717:       85 c0                   test   %eax,%eax
  400719:       75 0c                   jne    400727 <main+0xa3>
  40071b:       bf 6b 08 40 00          mov    $0x40086b,%edi
  400720:       e8 2b fe ff ff          callq  400550 <puts@plt>
  400725:       eb 0a                   jmp    400731 <main+0xad>
  400727:       bf 77 08 40 00          mov    $0x400877,%edi
  40072c:       e8 1f fe ff ff          callq  400550 <puts@plt>
```

La utilizad objdump nos permite, entre otras cosas, volcar tanto el ensamblador como el código máquina de un programa. Podríamos haberlo hecho desde gdb, pero así vemos otra herramienta. El flag -d nos permite desensamblar el programa.

Luego, utilizamos grep para localizar el salto en el que estamos interesados. Una vez más utilizamos el flag -C para mostrar las 10 líneas (5 arriba y 5 abajo) que se encuentran en torno a la instrucción en la que estamos interesados.

En el volcado proporcionado por objdump, podemos ver que el código máquina que tenemos que buscar y sustituir por NOPs (0x90) es 0x75 0x0c. Vamos allá.

Si disponéis de un editor hexadecimal... pues adelante, solo tenéis que buscar la cadena hexadecimal del salto y cambiar por NOPs. Si no disponéis de un editor hexadecimal, podéis utilizar la utilidad xxd. Esta utilidad permite hacer volcados hexadecimales de ficheros binarios y convertir a formato binarios volcados hexadecimales. Veamos como utilizarla:

xxd es también una herramienta muy útil

Primero generamos nuestro volcado hexadecimal:

```
$ cat c1 | xxd -g 1 > c1.txt
```

Ahora podemos abrir el fichero c1.txt con cualquier editor de texto y buscar la secuencia de bytes. Observad. El flag -g 1 le dice a xxd que vuelque bytes, ya que, por defecto, volcaría palabras. Aseguraros de que habéis encontrado el salto correcto comprobando las secuencia hexadecimal de las instrucciones anteriores y posteriores... Para eso le hemos dicho a grep que nos muestre lo que hay alrededor del salto un poco más arriba.

Ahora cambia "75 0c" por "90 90" y graba el fichero. Una vez modificado nuestro volcado hexadecimal podremos utilizar de nuevo la utilizad xxd para general el fichero binario de nuevo.

```
$ cat c1 | xxd -r > c1-mod
```

Como os podéis imaginar, el flag -r es el encargado de hacer el proceso inverso (*Reverse*) para convertir el volcado hexadecimal en un fichero binario de nuevo.

Ahora podéis probar a ejecutar c1-mod y introducir cualquier clave que queráis :)

Cambiando la contraseña

Si recordáis, al principio del artículo comentamos tres posibles soluciones al desafío. La ultima opción para solucionar este desafío era modificar la contraseña de forma que sepamos cual es.

Ahora que sabéis como hacer volcados hexadecimales de ficheros y como parchear los ficheros... esta opción la dejamos como ejercicio para el lector... No deberíais tener ningún problema. De lo contrario no dudéis en preguntar!

Hasta la siguiente entrega!... Que será mucho más emocionante!

Enlaces Externos

[1] http://www.catb.org/jargon/html/C/chainik.html

[2] https://en.wikipedia.org/wiki/Script_kiddie

Sobre el Autor

Wh1t3 D3M0n

Me conocen como Wh1th3 D3M0n y soy un hacker. Me oculto entre las sombras de la red, paseándome entre máquinas y datos como un espectro... invisible a los ojos de los usuarios...

Que va es coña!

Twitter: @ChainkMaster | Blog: https://thehackerkid.tumblr.com/

Introducción al Hackeo de Routers. Puerto Serie

Usa tu router como el ordenador que es

por Mike Anic

Muchos de los routers Wifi de bajo coste esconden misterios en su interior... Bueno, lo que queremos decir es que muchos de ellos ofrecen un puerto serie con el que acceder a la consola del sistema GNU/Linux que utilizan. En este artículo os vamos a contar localizarlos y acceder a ellos. Antes de comenzar, aquí va el mensajillo obligatorio cada vez que os contamos como, potencialmente, destruir vuestros dispositivos o incluso haceros daño... Básicamente esto quiere decir que si la liáis es vuestro problema.

DECLINACIÓN DE RESPONSABILIDADES

Ni el autor de este artículo, ni la Revista On-line Occam's Razor se hacen responsable de cualquier daño que se pueda causar derivado del contenido de este artículo Se responsable, aprende y presta atención para no hacerte daño y para no destruir tu equipo.

Normalmente, una vez que desmontes el dispositivo habrás perdido la garantía. Utiliza dispositivos viejos fuera de garantía o atente a las consecuencias.

Finalmente comentaros que el contenido de este artículo es una versión en castellano extendida del siguiente artículo en inglés.

http://papermint-designs.com/dmo-blog/2016-12-hardware-hacking-box--serial-access-to-routers [1]

Sin más preámbulos... amo allá

La Víctima

Nuestra víctima va a ser un **TP-Link WR841N**. Se trata de un router un poco viejo (uno que utilizábamos para pruebas) y de bajo coste (unos 20 euros).

Lo bueno de este router es que hay cantidad de información sobre él, de tal forma que si nos quedamos atascados en algo, será fácil encontrar la solución en Internet. Además sabemos seguro que este ofrece un puerto serie con el que aplicar nuestros conocimientos (además de un puerto JTAG... pero eso es para otro artículo).

Nuestro **WR841N** lo hemos reflasheado con el firmware OpenWRT (https://openwrt.org/) [2]. Esto lo hicimos hace algunos años, así que no podemos daros muchos detalles del firmware original del router. Sin embargo, el proceso de reflasheo, tanto de OpenWRT como de DD-WRT es muy sencillo y altamente recomendable.

En caso de que no encontréis o dispongáis de uno de estos, y realmente queráis juguetear con un

router, nuestra recomendación es que consultéis la base de datos on-line de OpenWRT antes de comprar uno y os aseguréis que, esos interfaces en los que estáis interesados están disponibles en el modelo que queréis adquirir. En el caso de que sea necesario hacer alguna modificación a la placa, ahí también os lo contarán y podréis decidir si os veis capaces o mejor lo intentáis con otro dispositivo.

Preparación

Antes de ponernos a hurgar en las tripas de los cacharros, vamos a necesitar algún material. Además de algunas herramientas, las cuales variarán dependiendo del dispositivo que queráis abrir, vamos a necesitar un adaptar Serie-TTL.

La mayoría de los puertos serie que ofrecen los routers no utilizan los +12/-12 V típicos de los interfaces serie RS-232C.. Normalmente utilizan niveles TTL (5V o 3.3V) para transmitir la información. De tal forma que tenemos un par de opciones para poder hablar con nuestro router:

La primera es construir nuestro propio adaptador. Hay mucha información en Internet sobre el tema y no vamos a discutir esta opción aquí. Podéis consultar esta página para conseguir más detalles: https://wiki.openwrt.org/doc/hardware/port.serial.cables [3]

La siguiente opción es usar uno de esos adaptadores USB-Serie baratos. Nosotros solemos utilizar uno basado en el PL2303, como el que aparece en la página del enlace anterior. Cuesta unos poco euros y la verdad que es bastante útil. En general, cualquiera de estos adaptadores suelen funcionar bien.

La última opción es utilizar un dispositivo intermedio capaz de hablar con nuestro ordenador (**Arduino**) o que sea un ordenador en si mismo (**Rpi, BBB, Olinuxino, BananaPi**) y que además sea capaz de conectarse a un puerto Serie-TTL. Cualquiera de las opciones anteriores es perfectamente válida. Es una cuestión de que tenéis a mano y de cuanto dinero os queréis gastar.

En concreto nosotros hemos utilizado un adaptador USB-Serie PL2303 y una Raspberry Pi para este artículo. Ambos funcionan perfectamente, si bien, en el caso de la Rpi es necesaria alguna configuración. Volveremos sobre esto en un momento cuando hablemos del SW.

Abriendo el TP-Link WR841N

Vamos a comenzar a destripar nuestra plataforma de referencia, el TP-Link WR841N.

FIGURA 1. TP-WR841N Nuestra plataforma de referencia
Fuente: ROOR Licencia: CC0

Si tenéis uno de estos, podéis encontrar todos los detalles sobre el cacharro, en los siguientes enlaces:

https://wiki.openwrt.org/toh/tp-link/tl-wr841nd [4]

http://www.dd-wrt.com/phpBB2/viewtopic.php?p=627915 [5]

https://giannoug.gr/unbricking-a-tp-link-tl-wr841nd/ [6]

Este router es bastante fácil de abrir... sólo hay que quitar unos tornillos y tirar. Los tornillos se encuentra bajo las almohadillas en la parte inferior del dispositivo. Al abrirlo intentad tener cuidado, nosotros rompimos una de las pestañas de plástico que mantienen unidas las dos partes de la carcasa.

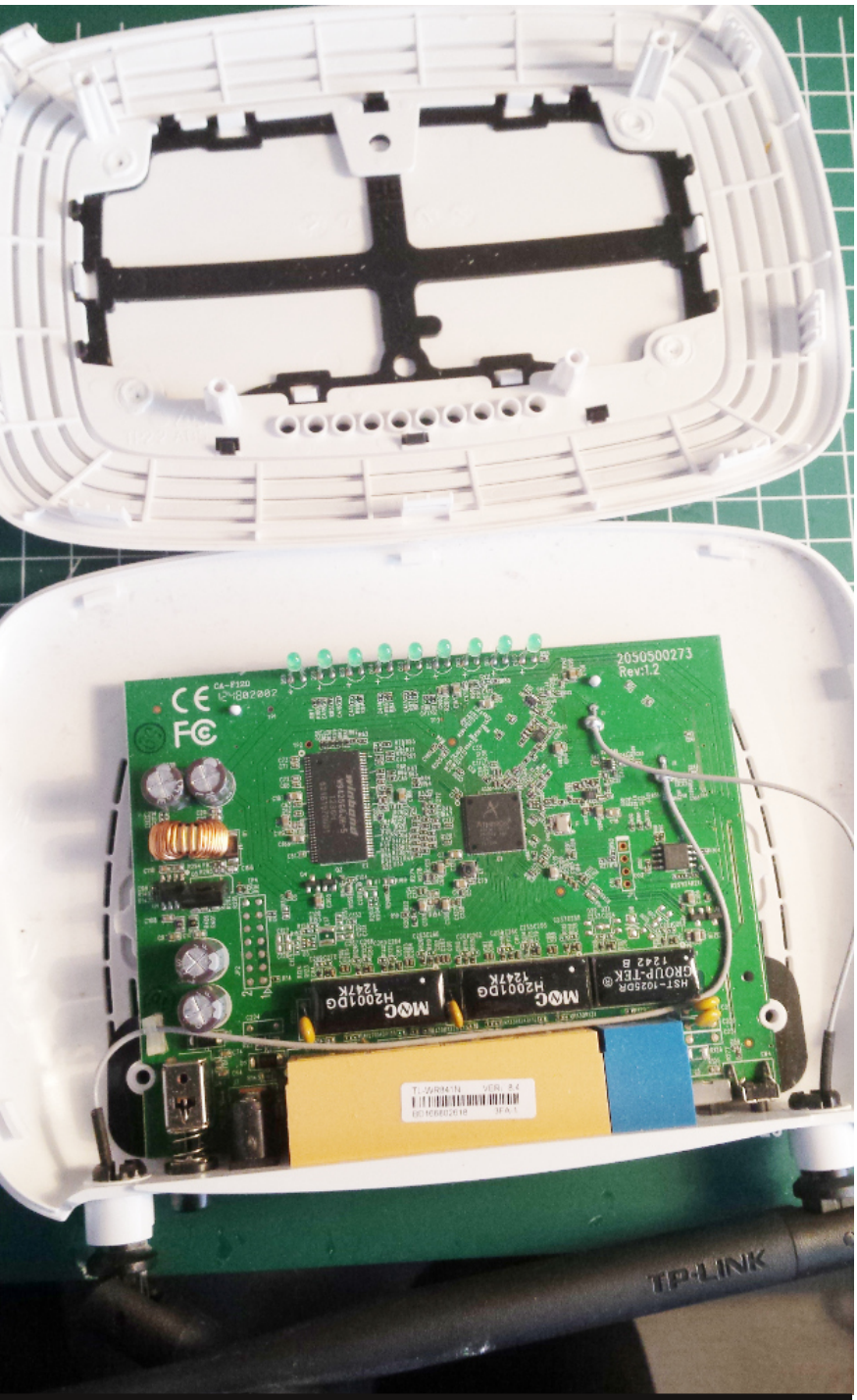


FIGURA 2. Las entrañas del TP-Link WR841N (Fuente: ROOR Licencia: CC0)

realzados en la Figura 4.

Como lo sabemos todo sobre este router, no necesitamos hacer nada especial para localizar el conector con el puerto serie y la función de cada una de las conexiones. Pero bueno, en general, cuando veáis un conector de 4 pines en la placa... bueno, ese suele ser el que nos interesa.

Ahora, dependiendo de las herramientas que tengáis, podréis conectar cables a las conexiones de la placa de distintas formas. Podéis soldar los cables directamente, usar algunos PogoClips, o soldar unos PINs para conectar esos prácticos jumpers que utilizamos con las placas de prototipos. Nosotros nos decantamos por esta última solución.

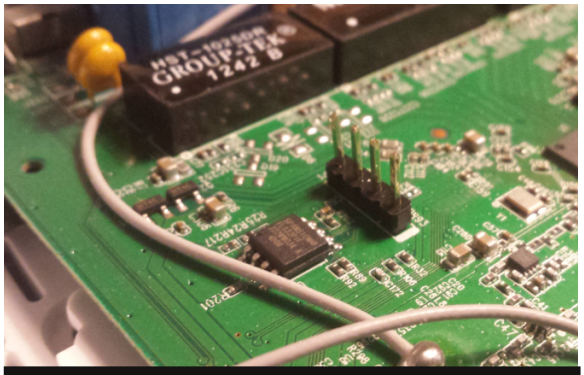


FIGURA 3. Pin Header para el puerto serie del WR841N (Fuente: ROOR Licencia: CC0)

Puertos Serie. Una breve Introducción

Si ya sabes como funciona un puerto serie, te puedes saltar esta sección. Si eso de 8N1 te suena a chino, entonces sigue leyendo.

Cuando quieres conectar dos dispositivos digitales para que intercambien información, tienes básicamente dos opciones. Puedes enviar los datos en serie o en paralelo. En principio, la unidad mínima de información que podemos intercambiar es un byte (8 bits en general, desde el punto de vista de la CPU), así que tenemos dos opciones.

- Utilizamos 8 cables y enviamos un bit por cada cable, de forma que podemos transmitir nuestro byte de una sola tacada. Esto es lo que hace un puerto paralelo. El antiguo interfaz centronics de las impresoras, o el bus GPIB utilizado en instrumentación, funcionan de esta manera.

- La otra opción es utilizar un solo cable y enviar los bits de nuestro byte, uno detrás del otro en bloques. El dispositivo en el otro lado, lee los bits y los junta de nuevo en un byte que se envía al programa principal que recibe/transmite los datos. Los puertos serie clásicos (RS-232C, RS-485,... etc) funcionan de esta forma. En la actualidad, los puertos USB,

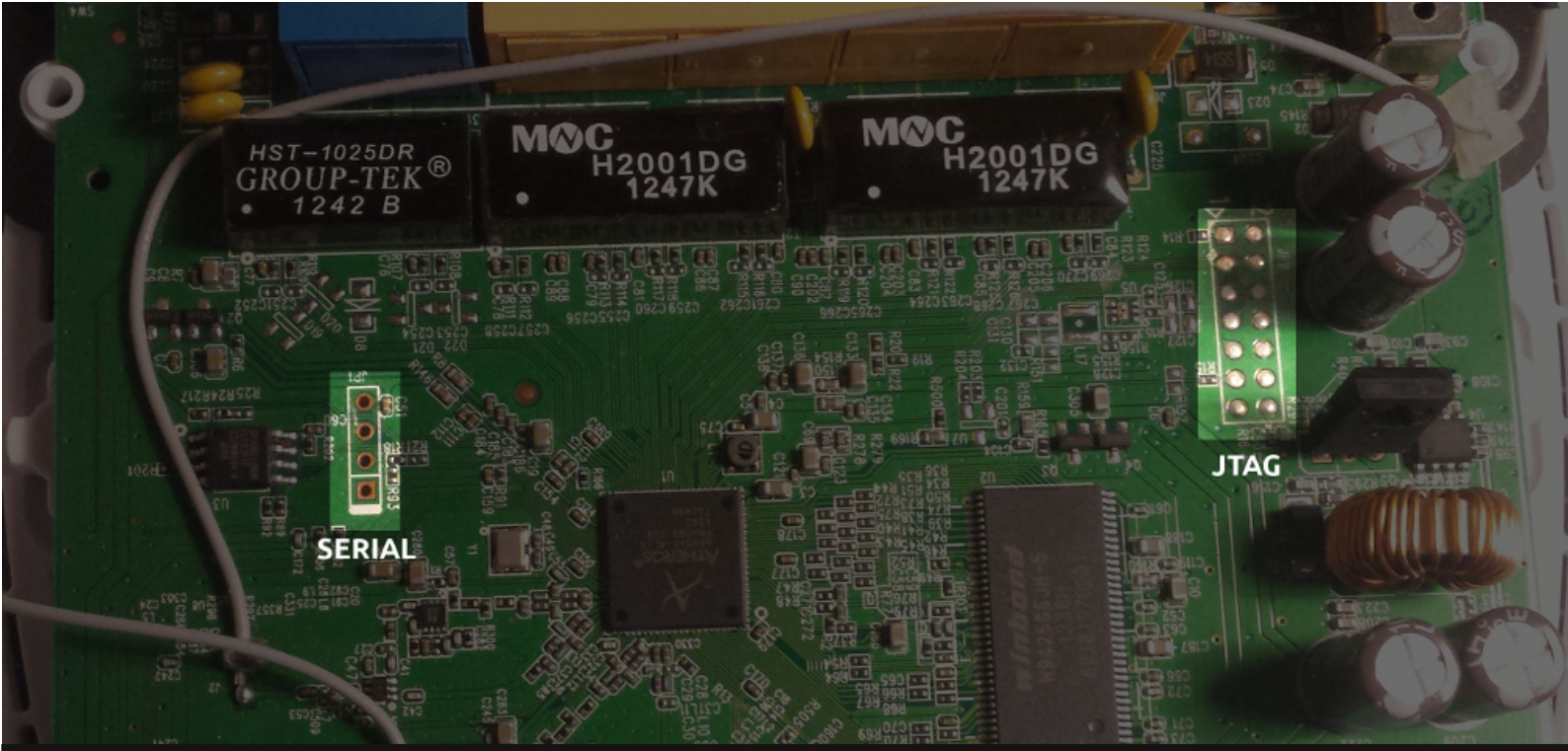


FIGURA 4. Ubicación de las conexiones para puerto serie e JTAG del TP-Link WR841N (Fuente: ROOR Licencia: CC0)

I2C o SPI también funcionan así.

Cada método, tiene sus ventajas e inconvenientes. En la actualidad la transmisión en paralelo no se utiliza para dispositivos externos. Los buses internos de los ordenadores siguen transmitiendo datos en paralelo, ya que es mucho más rápido, pero a un coste.

Un puerto serie envía y recibe datos bit a bit a través de un solo cable.

El principal problema de la transmisión en paralelo es que cuesta más. En lugar de un cable, necesitamos 8, 16 o 32 (dependiendo del tipo de interfaz). Además, cuando más rápido queramos enviar los datos, nuestro enlace será más sensible a errores de fase. En otras palabras, cuando más rápido sea nuestro enlaces, pequeñas diferencias en la longitud de los cables puede hacer que unas señales lleguen antes que otras y que se produzcan errores (eso puede producir un error de fase). Hay otros problemas, pero esos son quizás los más evidentes.

Cuando utilizamos un sistema de transmisión serie, también nos vamos a encontrar problemas similares (normalmente de sincronización de reloj, jitter y cosas de esas), así que no basta con simplemente enviar los 8 bits uno tras otro y ya está. En su forma más sencilla, los puertos serie de toda la vida, envían unos cuantos bits extra para, entre otras cosas detectar errores en la transmisión y saber donde empieza y donde termina un byte...

En general, un enlace serie va a enviar:

- Un bit de comienzo conocido como START BIT. Este bit indica el comienzo de un dato
- Los bits de datos que pueden ser 5, 6, 7 u 8. Normalmente utilizamos 8 bits, pero, recordemos que el código ASCII original utilizaba solo 7 bits
- A continuación se suele incluir un bit extra llamado de PARIDAD. En su forma más sencilla el bit de paridad indica si el número de "UNOS" en los datos es par o impar. Así, la paridad puede ser PAR o IMPAR. Este bit es una rudimentaria forma de detectar errores en los datos. En conexiones directas de corta distancia, este bit no se suele utilizar.
- Uno o dos bits indicando el fin del dato. Estos bits se conocen STOP bits o bits de parada ;).

Así, los parámetros de comunicación de un determinado puerto serie se suelen expresar como la velocidad de transmisión seguido de una cadena del tipo DPS donde:

- D indica los bits de datos que vamos a utilizar
- P indica el tipo de parida. Los valores posibles son N (None, sin parida), E (Even, paridad par) u O (Odd, paridad impar). Existen otros tipos de paridad pero son menos comunes.
- S indica el número de bits de parada (STOP).

Este valor puede ser 1 o 2.

Con todo esto, la inmensa mayoría de dispositivos serie en la actualidad utilizan una configuración **8N1**. Es decir 8 bits de datos, son paridad y un bit de parada.

Cada vez que queremos comunicarnos a través de un puerto serie con otro dispositivo, necesitamos ponernos de acuerdo en el formato de las secuencias de bits que vamos a intercambiar. Para el caso concreto de nuestro router la configuración es:

```
115200 8N1
```

Accediendo Al WR-841N

Ahora ya podemos acceder al puerto serie. Solo tenemos que conectar los cables a nuestro adaptador (ya sea un USB-Serie o un pequeño ordenador -**Arudio, Rpi**,...-). Si consideramos que el pin 1 es el que se encuentra más cerca de los conectores Ethernet del router, los pines en el router tienen las siguientes funciones:

```
PIN 1 Vcc
PIN 2 GND
PIN 3  Rx
PIN 4  Tx
```

En principio, el PIN 1 no se suele conectar. Sin embargo para este modelo, y la versión específica que nosotros tenemos es necesario. Antes de conectar nada, comprobad la documentación. Hay varias versiones de este router y cada una tiene sus peculiaridades con respecto al puerto serie.

> En general NUCA connectéis el pin VCC del primeras. Podéis dañar el dispositivo

En general, **NUNCA** conectéis el pin Vcc de primeras. Si no conseguís acceder al dispositivo, intentad buscar información en Internet primero, y si eso no funciona, y realmente queréis acceder a ese puerto cueste lo que cueste, pues probad, pero bajo vuestra única y exclusiva responsabilidad. Tenéis muchas papeletas para romper el puerto serie... el router quizás siga funcionando... depende.

La conexión específica al adaptador que hayáis elegido será diferente en cada caso. Sin embargo, existen unas reglas generales: -

- Conectad siempre todas las tierras (Pines GND) juntas y conectadlas antes que ningún otro pin. Si no lo hacéis... bueno, un montón de cosas raras pueden pasar. O dicho de otra forma: cuando pasen cosas muy raras que no tengan ningún sentido... Comprobad que todos los pines GND están conectados juntos
- Los pines Tx/Rx deben conectarse en lo que se conoce como una configuración de modem nulo. Esto significa que el pin Tx del router debe conectarse al pin Rx de nuestro adaptador y el pin Rx del router debe conectarse al pin Tx de nuestro adaptador. Esto suele requerir un poco de prueba y error, ya que ciertos adaptador marcan los pines desde su propio punto de vista y otros desde el punto de vista del dispositivo que se conecta al adaptador...

> En una configuración de modem Nulo, los pines TX/RX de los dos terminales están cruzados

Una breve pincelada sobre identificar pins

Nuestra recomendación fue empezar con un router del que dispongáis la información, pero si eso no es posible, tendréis que obtener esa información vosotros mismos. La siguiente página ofrece una descripción detallada del proceso para localizar los PINes en la placa del router (la información interesante está al final de la página).

https://wiki.openwrt.org/doc/hardware/port.serial [7]

Para los que no queráis leer la página en inglés (que deberíais ya que contiene muchísima información muy útil), este es un resumen de como proceder si solo disponemos de un multí-metro. - Activar el test de continuidad (o medidor de resistencia).

- Para localizar el PIN de tierra (GND), poned una de las pruebas del multímetro en un punto que sepáis que es tierra y comprobad que pin está conectado con tierra. Un buen punto de partida es utilizar uno de los pines en el conector de alimentación.... Haced este test con el cacharro apagado
- Para localizar el pin de alimentación, colocad una de las pruebas en el pin de tierra que habéis encontrado y cambiad el multímetro para medir voltaje, en una escala de al menos 5V. Luego, con la otra prueba del multímetro,

comprobad los pines, y buscad uno con unos 3.3V o 5V estables. Esto hay que hacerlo con el router encendido... así que mucho cuidadín.

- En esta misma configuración, comprobaremos los otros pines. Una vez que tengáis las pruebas del multímetro conectadas a GND y al pin a comprobar, encenderemos el router. Esto provoca que el router envíe todos esos mensajes de arranque y por tanto, el pin Tx será aquel en el que el la lectura varíe más.
- El pin que queda es Rx

En la realidad las cosas no son tan fáciles, pero el proceso que acabamos de describir funciona en la mayoría de los casos. Si disponéis de un analizador lógico u osciloscopio, probablemente el proceso sea todavía más sencillo. Software

Ahora que ya estamos conectados necesitamos algún programa para poder recibir y enviar datos por el puerto serie. Tenemos varias opciones:

- **Minicom**. Minicom es la mejor opción. Es un programa diseñado para trabajar con puertos serie y lo hace de forma correcta. Además nos permitirá, de forma sencilla, probar distintas configuraciones del puerto serie, para encontrar la que le guste a nuestro router. Para iniciar minicom contra un determinado puerto utilizad el comando:

```
minicom -D /dev/TuDispositivo
```

  Luego podéis utilizar los menús para cambiar la configuración cuando lo necesitéis.

- **screen**. Este programa se utiliza normalmente para ejecutar sesiones en background, de forma que nuestros programas sigan ejecutándose cuando nosotros ya no estemos conectados. Además de eso, screen nos permite conectarnos a puertos serie con un comando como este:

```
screen /dev/ttyS0 115200
```

  Donde el número final es la velocidad del puerto serie al que nos queremos conectar.

---

minicom, screen o NetKitty son algunas de las opciones para utilizar el puerto serie

---

- **NetKitty**. Esta es mi versión reducida de Netcat a la que le he añadido soporte para acceder a puertos serie estándar (8N1). La ventaja de Netkitty is que podéis descargar una versión estática compilada para las principales plataformas (podrías utilizar bustro teléfono android para acceder al router por ejemplo) y que nos permite acceder al puerto serie de forma remota.

```
nk -c S,/dev/ttyS0,115200
```

  Si además de acceder al puerto serie localmente, queremos acceder a él a través de una conexión TCP al puerto 5000 de la máquina conectada al router, solo tenemos que escribir algo como esto.

```
nk -hub -s T,5000 -c
S,/dev/ttyS0,115200
```

  Netkitty ofrece muchas más opciones. Echadle un ojo a este tutorial [9].

Para poder utilizar el puerto serie de la Raspberry Pi, tendremos que hacer algunas cosillas...

Una vez que hayáis elegido vuestro software y os hayáis conectado al router, lo más probable es que el programa solo muestra caracteres extraños. Necesitamos determinar la velocidad a la que se ha configurado el puerto serie. Bien, la única forma de hacer esto es probando... cambiando la velocidad hasta que lo que recibamos tenga sentido. Hay algunos programas para automatizar el proceso, pero en la práctica son solo 4 o 5 valores que debemos probar. Empezaremos por el máximo 115200 he iremos bajando... En general, la consola se configura a la máxima velocidad posible, por eso sugerimos empezar por el valor más alto... pero cualquier otra estrategia funcionará igual

Liberando el puerto serie de la RPI

A diferencia de otras tarjetas de desarrollo, la RPI tiene un solo puerto serie accesible (en principio) que, por defecto se utiliza como consola del sistema. Es decir, durante el arranque es adquirido por el programa (mgetty o equivalente) para proporcionar un login al sistema. Normalmente esto se configuraba en el fichero /etc/inittab pero en versiones recientes, la versión de DEbian en la que se basa Raspbian a comenzado a utilizar systemd.

En cualquier caso, si vais a utilizar una Rpi con una

versión reciente de Raspbian, seguid los siguientes pasos en caso de que no consigáis acceder al puerto serie de la tarjeta.

- Utilizando la herramienta de configuración de la tarjeta (raspi-config) acceded al menu de las opciones avanzadas y luego seleccionad Serial. El programa os preguntará si queréis acceso shell a través del puerto serie. Responded que no
- Luego comprobad que el ficherto /boot/config.txt contiene la linea enable_uart=1
- Reiniciar

Esto debería liberar el puerto serie para poder conectar nuestro router, o cualquier otro cacharro. Recuerda que el puerto serie en la Rpi es de 3.3V. Conectar algo con mayor voltaje puede dañar tu tarjeta.

Esto esto. Esperamos que hayáis encontrado interesante este artículo y no dudéis en comentar sobre vuestros proyectos. Hasta la próxima

Sobre el Autor

Mike Anic

Mike es nuestro experto en las cosas reales. Todo eso que puedes tocar... Microcontroladores... chupao. El internet de las cosillas.... menuda tontería...Condensadores de Fluzo... estamos trabajando en ello.

Enlaces Externos

[1] http://papermint-designs.com/dmo-blog/2016-12-hardware-hacking-box--serial-access-to-routers
[2] https://openwrt.org/
[3] https://wiki.openwrt.org/doc/hardware/port.serial.cables
[4] https://wiki.openwrt.org/toh/tp-link/tl-wr841nd
[5] http://www.dd-wrt.com/phpBB2/viewtopic.php?p=627915
[6] https://giannoug.gr/unbricking-a-tp-link-tl-wr841nd/
[7] https://wiki.openwrt.org/doc/hardware/port.serial
[8] http://savannah.nongnu.org/projects/netkitty/

CI2C. Curso de Ingeniería Inversa para Chainiks. Parte II

Inyección de librería dinámicas...

por Wh1t3 D3M0n

Aha!. No habéis tenido suficiente con la parte I. Algunos valientes habéis vuelto a por más. No hay problema tenemos de sobra para los bravos de corazón que desean continuar el camino hacia la iluminación ingenieril.

En esta entrega vamos a explorar otra forma de crackear el programa que utilizamos en la parte I. Para los que no los despistados, el programa era este:

```
#include  <stdio.h>
#include  <string.h>
char *the_pass = "MyVoiceIsMyPassport";

int
main (void)
{
  char   user_pass[1024];

  printf ("Introduce la contraseña: ");
  fgets (user_pass, 1024, stdin);
  if (!strncmp (user_pass, the_pass,
strlen(the_pass)))
    {
      printf ("Bien Hecho!\n");
    }
  else
    printf ("Sigue Intentándolo\n");
}
```

A estas alturas ya sabéis como encontrar la clave usando strings, como modificar la clave por una que nosotros queramos (este era el ejercicio propuesto, espero que hayáis hecho los deberes) y como *crackear* el programa para saltarnos la comprobación de la clave.

En esta ocasión vamos a ver una técnica alternativa para crackear la clave en un caso sencillo como este. La técnica se conoce como inyección de librerías dinámicas... El nombrecito mola o qué?

Funciones externas

Vamos a echar un ojo al desensamblado de nuestro programa víctima. En concreto a la parte en la que comprobamos la clave utilizando la función strncmp (Listado 1).

Os resulta familiar verdad?. Bien, fijémonos en la llamada a la función.

```
callq  0x400540  <strncmp@plt>
```

Como podéis ver, gdb identifica la función como strncmp@plt. Sin entrar en muchos detalles por ahora, esto significa que la función strncmp está definida en una librería dinámica externa. En concreto, esta función forma parte de la librería C estándar, conocida como libc.so.

```
0x00000000004006fa  <+118>:   lea    -0x1(%rax),%rdx
0x00000000004006fe  <+122>:   mov    0x20093b(%rip),%rcx        # 0x601040  <the_pass>
0x0000000000400705  <+129>:   lea    -0x410(%rbp),%rax
0x000000000040070c  <+136>:   mov    %rcx,%rsi
0x000000000040070f  <+139>:   mov    %rax,%rdi
0x0000000000400712  <+142>:   callq  0x400540  <strncmp@plt>
0x0000000000400717  <+147>:   test   %eax,%eax
0x0000000000400719  <+149>:   jne    0x400727  <main+163>
```

LISTADO 1. Ensamblador de la parte del program que comprueba la clave

Veámoslo:

```
$ ldd c1
	linux-vdso.so.1 =>  (0x00007ffefeffd000)
	libc.so.6 =>  /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3d5be0d000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f3d5c1fd000)
```

Si, la segunda línea es la librería estándar. Vamos a olvidarnos por el momento de las otras dos que no nos interesan para el tema que nos ocupa.

Para poder ejecutar el programa, el sistema tiene que cargar estas librerías dinámicas, de forma que podamos llamar a funciones como fgets, printf o strncmp. La librería se va a cargar en una determinada posición de memoria que, en general, va a ser diferente para cada programa. Dependiendo del tamaño del programa, que otras librerías tenemos que cargar y otros factores, nuestra función strncmp se puede encontrar en cualquier lugar de la memoria. Bueno, no en cualquier, pero en principio no podemos saberlo a priori.

> Una de las funciones del enlazador dinámico es la relocalización de símbolos

Para solucionar este problema el sistema utiliza unos cuantos elementos. Por una parte tenemos el Enlazador Dinámico (*Dynamic linker*). Este es, de hecho, la última librería que nos muestra el comando ldd. El Enlazador dinámico es el encargado de cargar la librería y hacer que los programas puedan encontrar las funciones que ofrece, independientemente de en que parte de la memoria acabe.

Una de las cosas que hace el Enlazador Dinámico, es lo que se llama relocalización (*relocation*). Sin entrar en muchos detalles por el momento, la relocalización es el proceso por el que las direcciones de las funciones de la librería se ajustan dependiendo de la posición en memoria en la que la librería acabe.

Finalmente, el sistema ofrece una optimización. En lugar de relocalizar todas las funciones (todos los símbolos en general... cualquier cosa del programa que tenga nombre y sea accesible desde fuera) al cargar el programa, lo que lleva tiempo y haría que el programa tardase mucho en cargar, espera a que el programa llame a la función para llevar a cabo este proceso. Esto se conoce como Enlazado Holgazan (*Lazy Binding*)... solo se resuelven las funciones que se utilizan y solo cuando se utilizan.

Para poder llevar a cabo este proceso, el programa no puede llamar directamente a las funciones... básicamente no tiene ni idea de en que posición de memoria están. Así que el compilador general una tabla de punteros que apuntan a la función que localiza el símbolo asociado. Una vez que se conoce en que dirección de memoria está la función a la que queremos acceder, esa tabla se modifica para que apunte a la función real, de tal forma que la próxima vez que llamemos a la función saltaremos directamente a su código y no a la rutina de relocalización.

PLT. La Tabla de Enlazado de Procedimientos... Flipa

Bueno, puede que no os hayáis enterado del rollo que acabamos de soltar. No os preocupéis volveremos sobre este tema más adelante. Por el momento lo que tenéis que saber es que, las funciones externas, las que ofrecen librerías dinámicas no se llaman directamente.

El programa tiene que acceder a la denominada PLT Tabla de Enlazado de procedimientos (*Procedure Linkage Table*). El enlazador dinámico... o un ente externo muy listo, si lo preferís, se va a encargar de que en esa tabla aparezcan las direcciones reales de las funciones que queremos ejecutar.

En nuestro caso, strncmp es una función externa, y nuestro programa víctima ejecuta esta función con un salto (callq) a la entrada PLT asociada a strncmp. Si echáis un ojo al resto de la función main, veréis instrucciones similares para acceder a fgets o puts.

Que tiene de interesante todo este rollo. Bien, la respuesta es: la pre-carga de librerías.

Linux nos permite proporcionar una librería alternativa que se cargará antes que nada en la memoria. El efecto que esto tiene es que, al final, el puntero en la tabla PLT va a apuntar al código de esa librería que hemos pre-cargado, en lugar de a la librería estándar. En otras palabras, disponemos de una herramienta con la que modificar el comportamiento de cualquier función de una forma muy sencilla.

> La PLT es una tabla que contine los punteros a todas las funciones ofrecidas por las librerías dinámicas utilizadas

Crackeando el programa y 2

Con esta nueva herramienta, lo que vamos a hacer es una versión de strncmp que, por una parte nos muestre los parámetros que el programa está pasando a la función, y que por otra parte devuelva siempre el valor 0. La función strncmp original devolverá el valor 0 sólo cuando las dos cadenas que recibe como parámetro sean iguales. Haciendo que la función devuelva siempre 0 estaremos haciendo creer al programa principal que ambas cadenas son iguales, aunque no lo sean.

> Escribiendo nuestra propia versión de strncmp podemos cracker el programa fácilmente e incluso obtener la clave

Manos a la obra. Nuestra función strncmp sería algo como esto:

```
#include  <stdio.h> int
strncmp (char *s, char *d, int n)
{
  pritnf ("[Ci2C] Cadena 1: %s\n", s);
  pritnf ("[Ci2C] Cadena 2: %s\n", d);

  return 0;
}
```

Grabamos este código en un fichero llamado c1-so.c. Ahora tendremos que generar una librería dinámica a partir de este fichero:

```
$ gcc -shared -fPIC -o libci2c.so.1.0.0 c1-so.c
c1-so.c:4:1: warning: conflicting types for built-in function 'strncmp'! !  [enabled by default]
```

Veamos que estamos haciendo. El flag -shared le dice al compilador que queremos generar un objeto dinámico. El flag -fPIC le dice al compilador que tiene que generar código independiente de la posición (*Position Independent Code*). Esto nos permite cargar la librería dinámica en cualquier posición de memoria dejar al enlazador dinámica hacer su magia. El resto ya deberíais saber que significa.

Bueno, bueno, es hora de probar nuestra librería:

```
$  LD_PRELOAD=./libci2c.so.1.0.0 ./c1
Introduce la contraseña: aa
[Ci2C] Cadena 1: aa

[Ci2C] Cadena 2: MyVoiceIsMyPassport
Bien Hecho!
```

Genial!... No solo hemos crackeado el programa (da igual que contraseña introduzcamos), también estamos mostrando en la pantalla la contraseña real. Dos pájaros de un tiro!

Mú Fácil

Hasta aquí las cosas han sido, mú, mú fáciles, pero espero que hayáis pillado la idea general. Crackear un programa nunca es tan fácil, pero gracias a este sencillo ejemplo, hemos introducido unas cuantas herramientas que vamos a usar extensivamente en las próximas entregas.

No os preocupéis si no habéis entendido completamente todo el proceso del enlazado dinámico. Volveremos sobre él varias veces muy pronto y al final, sin daros cuenta lo habréis pillado.

No os perdáis la siguiente entrega... diversión asegurada!

Sobre el Autor

Wh1t3 D3M0n

Me conocen como Wh1th3 D3M0n y soy un hacker. Me oculto entre las sombras de la red, paseándome entre máquinas y datos como un espectro... invisible a los ojos de los usuarios...

Que va es coña!

Twitter: @ChainkMaster | Blog: https://thehackerkid.tumblr.com/

Conversaciones Cibernéticas

Habla con tu ordenador en Español

por Signul Floid

Todos le hablamos al ordenador. Eso es un hecho. Normalmente no para decirle nada bueno. Más bien todo lo contrario... Pero existe otro mundo. Un mundo en el que charlar con tu ordenador de una forma agradable y amena. Un mundo sin insultos y golpes a inertes pantallas e indefensos teclados...En este artículo os vamos a descubrir como utilizar pocketsphinx en Español y así, abriros la puerta a un nuevo mundo lleno de bonitas sensaciones... o no?

Por supuesto, Google Voice, Siri, Alexa... todos esos sistemas funcionan de vicio. Así que muchos de vosotros os estaréis preguntando... Pá qué?. Bueno, ahí van algunas razones:

- Porque es super-güay hacer las cosas uno mismo :P
- Algunos paranoicos no estamos muy cómodos compartiendo huellas biométricas y localizaciones físicas con corporaciones que siguen reglas cuestionables sobre como manejar información privada
- A veces no tienes una conexión de red disponible para acceder a los servidores de esas compañías
- He dicho ya que es güay?

Así que, los que estéis cómodos con esos servicios de reconocimiento de voz, podéis saltaros este artículo... eso sí, cuando vuestra privacidad se vea malamente comprometida no nos vengáis a llorar... Ajo y agua :P

Sin más preámbulos, vamos al tajo.... luego pasaremos por el Duero y, sin ninguna duda visitaremos el Ebro.

La Plataforma

Podéis utilizar cualquier tipo de ordenador para jugar con lo que estamos a punto de contaros, pero nos gustaría decir un par de palabras sobre el tema, basadas en nuestra experiencia durante la preparación del artículos.

La única restricción obligatoria es que el ordernador que elijáis tenga una entrada de audio... Vamos una tarjeta de audio y que funcione. Como os decíamos, hemos probado unas cuantas opciones con diferentes resultados.

- PC. La primera plataforma que probamos fue nuestro querido PC con GNU/Linux, el cual, como era de esperar, funcionó perfectamente.
- Raspberry Pi. La Rpi funciona perfectamente, incluso la primera generación. Sin embargo, es necesario utilizar una tarjeta de audio USB externa (o alguno de esos Hats) para poder capturar el audio
- Banana Pi.La Banana Pi es un clon de la Rpi con algunas mejoras. Además de un interesante interfaz SATA para conectar discos duros externos, la tarjeta trae un micrófono

incorporado de forma que no es necesario ningún HW adicional.

- Android. También hemos probado pocketsphinx en Android, si bien, es un poco más complicado y no vamos a tratar ese tema en este artículo

Bueno, nosotros nos quedamos con la BananaPi para las pruebas, pero los pasos para utilizar pocketsphinx son los mismos para cualquier ordenador que funcione con GNU/Linux... Así que vamos a instalar.

INSTALACIÓN

La instalación es bastante sencilla. Las instrucciones son para sistemas basados en Debian, los más comunes hoy en día. Para otras distribuciones la forma de instalar los paquetes será diferente... Lo primero que necesitamos es el compilador. Probablemente eso ya lo tengas instalado, pero sino, el paquete build-essential suele ser todo lo que necesitamos:

```
sudo apt-get install build-essential
```

Además del compilador, vamos a necesitar bison y el paquete de desarrollo de ALSA:

```
sudo apt-get install bison libasound2-dev
```

Esas son todas las dependencias que deberíamos necesitar... No dudéis en dejar un comentario si descubrís que algún paquete extra necesita ser instalado.

Ahora debemos descargar y compilar pocketsphinx. Durante unos cuantos años, pocketsphinx no sacó nuevos *releases*, siendo la version 0.8 la última disponible. Parece ser que, recientemente, se a publicado una nueva versión. Nosotros la probamos rápidamente, pero no funcionó inmediatamente y por ello hemos decidido escribir este artículo sobre la versión 0.8.

> PocketSphinx es un paquete libre para el reconocimiento de voz.

Las instrucciones para instalar la nueva versión son prácticamente las mismas, sin embargo, esta nueva versión respondía de forma muy lenta, probablemente debido a alguna configuración nueva que no hemos visto. En cualquier caso, si la probáis y conseguís que funcione correctamente, enviadnos una nota... plis

Compilando PocketSphinx

Lo primero que debemos hacer es descargar los paquetes sphinxbase y pocketsphinx de la web de proyecto:
https://sourceforge.net/projects/cmusphinx/files/

Crearemos un directorio y descomprimiremos ambos paquetes en el:

```
~$ mkdir compilar
~$ cd compilar
~/compilar $ tar xzvf
~/Downloads/sphinxbase-0.8.tgz
~/compilar $ tar xzvf
~/Downloads/pocketsphinx-0.8.tgz
```

Es importante que los dos paquetes se encuentren en el mismo directorio. PocketSphinx utiliza algunos ficheros de sphinxbase y por ello necesita saber donde encontrarlos.

Ahora solo tenemos que compilar sphinxbase y luego pocketsphinx

```
~/compilar $ cd sphinxbase-0.8
~/compilar/sphinxbase-0.8 $ ./configure  &&  make
~/compilar/sphinxbase-0.8 $ sudo make install
~/compilar/sphinxbase-0.8 $ cd ../../pocketsphinx-0.8
~/compilar/pocketsphinx-0.8 $ ./configure  &&  make
~/compilar/pocketsphinx-0.8 $ sudo make install
~/compilar/pocketsphinx-0.8 $ sudo ldconfig -v
```

Ya estaría listo. Intentad ejecutar pocketsphinx_continuous. Por defecto utiliza el lenguaje inglés... y no es que funcione muy bien con los modelos distribuidos por defecto, pero si se ejecuta y reconoce algo, aunque sea mal, significa que nuestra instalación ha funcionado.

Ahora hablemos español.

Hablando en Español

Pocketsphinx viene con soporte para inglés (y creo que chino, pero ese no lo he probado :P). Lo que nos interesa a nosotros, es poder hablar en Español, o incluso gallego (el modelo acústico y de lenguaje es prácticamente el mismo), Así que tenemos que instalar algunos ficheros extra.

Los ficheros los podéis encontrar en esta página web:

http://www.speech.cs.cmu.edu/sphinx/models/hub4spanish_itesm/ [2]

Descargad los fichero y guardadlos en un lugar conveniente. Ahora solo tenemos escribir una línea de comandos pasándolo todos estos ficheros que nos acabamos de descargar.

```
pocketsphinx_continuous -mdef ./es/H4.2500.mdef -lm
./es/H4.arpa.Z.DMP -hmm ./es/H4.cd_continuous_8gau -dict
./es/h4.dict -feat s3_1x39
```

Ya os habréis dado cuenta de que el parámetro que pasamos con el flag hmm es uno de los ficheros tar que hemos descargado. Tendréis que descromprimirlo para poder usarlo. Estos ficheros son modelos acústicos... básicamente, probad el que mejor funciona para vosotros.

Si habéis seguido los pasos hasta aquí, y todo ha ido bien, dispondréis de un gracioso reconocedor de voz que parece un viejo medio sordo. Jamás hemos conseguido que el reconocimiento libre funcionara bien, si bien, también es cierto, que nunca hemos intentado entrenar modelos acústicos y de lenguaje específicos.

Podéis probar a entrenar vuestros modelos. La web de CMU Sphinx contiene todos los detalles. Nosotros vamos a dejar de lado el reconocimiento libre, vamos lo que sería el dictado, y vamos a concentrarnos en el uso de gramáticas.

UNA GRAMÁTICA

Cuando reducimos las palabras de nuestro lenguaje a una pequeña lista con reglas más estrictas, la cosa mejora un montón. Pocketsphinx nos permite definir gramáticas con las que restringir el número de palabras a reconocer.

Las gramáticas en pocketsphinx se definen utilizando el formato JSGF (Java Speech Grammar Format [3]). El formato es muy sencillo así que vamos a ir directamente con un ejemplo que podáis probar:

```
#JSGF V1.0;
grammar test;
public  <computer>  = [PEPA]  <command>;
 <command>  =  <saludo>  |  <action>  ;
 <saludo>  = (HOLA | BUENAS | ADIOS);
 <action>  = (MUSICA | INFORME);
```

El formato es muy sencillo. La definición de la gramática realmente empieza con la palabra clave public. Cualquier palabra entre los caracteres < y > es un identificador. Cualquier cadena entre corchetes representa un palabra opcional. La barra vertical (|) es un OR y más o menos esa es la cosa. La gramática que hemos definido más arriba, se podría explicar de la siguiente forma:

- Cualquier comando esta formado por la palabra PEPA, que puede omitirse seguida de uno de los siguientes comandos ()
- Los posibles comandos están agrupados en dos categorias: <saludo> representa posibles saludos y <action> representa acciones que queremos que nuestro sistema lleve a cabo
- Los posibles <saludos> que queremos reconocer son las palabras: "HOLA", "BUENAS" y "ADIOS"
- Las posibles acciones son "MUSICA" e "INFORME"

> El formato JSGF nos permite definir fácilmente gramáticas para nuestro reconocedor de voz

Con esta gramática el reconocedor de voz solo va a intentar reconocer las palabras que hemos indicado en el fichero, y además con las condiciones indicadas. Por ejemplo, los saludos son

excluyentes. Podemos decir: "HOLA", "BUENAS" y "ADIOS", pero no podemos decir "HOLA BUENAS" ya que esa combinación no está recogida en la gramática.

Si probáis esta gramática comprobaréis que no funciona. La razón es que, el diccionario que viene con el paquete de Español que hemos instalado es bastante limitado. Así que tendremos que añadir las palabras que no aparezcan en el diccionario.

> Utilizando unos cuantos ficheros podemos hacer, fácilmente, que PocketSphinx entienda Español!

Comenzaremos haciendo una copia del fichero original (h4.dict) a la que llamaremos mi.dict. Ahora vamos a añadir la palabra PEPA, que no se encuentra en el fichero original. Abrimos el fichero con un editor de texto y buscamos la parte del fichero donde se encuentran las palabras que empiezan por P (está ordenado alfabéticamente). Realmente no es necesario añadir la palabra en el orden alfabético correcto, pero es conveniente para poder encontrarla fácilmente si tenemos que hacer alguna modificación.

El diccionario no es más que la transcripción fonética de una lista de palabras. En Español, la transcripción fonética es bastante directa. En caso de duda, buscad una palabra que suene igual en el diccionario y comprobad su transcripción.

Las líneas que hemos añadido al diccionario para soportar nuestra gramática son:

```
(....)
ADIOS     A D I O S
(...)
MUSICA     M U S I K A
(....)
PEPA     P E P A
(...)
```

Como podéis ver, para el caso de la palabra música, utilizamos la K en lugar de la C. La Ka suena distinto que la Ce. Bueno, ahora podéis probar el reconicimiento usando nuestra nueva gramática y diccionario:

```
pocketsphinx_continuous -mdef
./es/H4.2500.mdef -lm ./es/H4.arpa.Z.DMP
-hmm ./es/H4.cd_continuous_8gau -dict
./es/mi.dict -feat s3_1x39
```

Ahora debería funcionar perfecto.

MODIFICANDO CONTINUOUS

Hasta ahora hemos estado utilizando el programa pocketsphinx_continuous que viene con PocketSphinx y mola mucho. Pero lo que realmente queremos es que el ordenador haga cosas cuando le hablemos.

Una forma de conseguirlo es filtrando la salida de pocketsphinx_continuous con algún tipo de script que localice el texto reconocido. Aún así, el programa imprime un montón de mensajes que son un poco difíciles de filtar, así que mejor, vamos a cambiar el programa un poco para que podamos obtener el resultado del reconocimiento de una forma mucho más sencilla. Realmente no necesitamos hacer esto para filtrar la salida, pero así os enseñamos la parte del programa que os va a interesar modificar en vuestras aplicaciones.

Ya estáis yendo al directorio con el código fuente de PocketSphinx y abriendo el código fuente. La función que nos interesa se llama recognize_from_microphone.

> Un cambio mínimo en el programa de ejemplo y nos ponemos en marcha

Esta función inicializa el reconocedor y lee las muestras de audio del micrófono, utilizando unas funciones que proporciona SphinxBase. Bueno, lee audio hasta que detecta un silencio y en ese momento intenta reconocer el texto.

Busquemos la llamada a la función ps_get_hyp dentro de recognize_from_microphone. Hay un printf:

```
printf("%s: %s\n", uttid, hyp);
```

Esta es la línea que imprime el resultado del reconocimiento tal que así:

```
000000002: PEPA ADIOS
```

Lo que vamos a hacer es cambiar esa línea por

```
printf("##RESULT:%s\n", hyp);
```

De tal forma que un simple "grep "##" nos muestre solo las líneas que nos interesan.

Así que comentemos el printf original.

Cambiémoslo por el nuevo y a recompilar se ha dicho:

```
gcc -o continuous continuous.c -lsphinxbase -lsphinxad
-lpocketsphinx -I /usr/local/include/sphinxbase/ -I
/usr/local/include/pocketsphinx/
```

Probemos que tal funciona ahora:

```
 ./continuous -mdef ./es/H4.2500.mdef -lm
./es/H4.arpa.Z.DMP -hmm ./es/H4.cd_continuous_8gau -dict
./es/mi.dict -feat s3_1x39 -jsgf simple.jsgf 2> /dev/null
| grep "##"
```

Estupendo!. Ahora solo tenemos que enviar esos comando a algún programa para que haga algo útiles con ellos. Alternativamente, podemos procesar los mensajes directamente en el programa C...

El Cielo es el límite

A partir de aquí solo vuestra imaginación puede poner límites a lo que hacer con esta tecnología. Ahí van algunos ejemplos:

Alexa?... que Alexa ni que porras. Pepa, Manuela, Gumersinda... usad el nombre que queráis. Una pequeña modificación de la gramática y un poco de código para modificar el programa y que la salida de comandos solo se active si pronunciamos el nombre correcto. Un flag, un temporizador, un if y ya estamos listos

Y si hacemos el servicio accesible sobre la red?. Nada más fácil. Solo hay que empipar la salida de grep a NetKitty en modo hub servidor (nk -hub -s T,5000). Nos conectamos al puerto y obtenemos el resultado del reconocimiento en cualquier máquina de nuestra red

Enlaces Externos

[1] https://sourceforge.net/projects/cmusphinx/files/

[2] http://www.speech.cs.cmu.edu/sphinx/models/hub4spanish_itesm/

[3] https://en.wikipedia.org/wiki/JSGF

Sobre el Autor

Signul Floid

Tras abandonar sus estudios de psiquiatría, Signul se volcó en el apasionante mundo del procesado de señales convirtíendose en una de las eminencias mundiales en el tema. Cuando no está transformando o aplicando complejos algorithmos a cualquier señal que se le ponga por delante, Signul disfruta psicoanalizando a cualquier persona que se le cruce.

Nuestro Primer Polyglot

PDF , ZIP o programa Java?... Porque no todos?

por Occam's Razor

Como ya sabréis, este número de Occam's Razor, en formato PDF, se puede interpretar como un fichero ZIP o como una aplicación Java que implementa el algorithm esteganográfico descrito en el artículo Mensajes Clandestinos. El fichero ZIP contiene el código fuente de los artículos de la revista, y la revista en si misma... Moooola... que no?

Como funciona?

Dejando a un lado algunas limitaciones de las que hablaremos en un momento, el truco es bastante sencillo. Los lectores de PDF dejan de interpretar el fichero una vez que encuentra la marca de final de fichero %%EOF. De tal forma que, cualquier cosa que añadamos al final del fichero no va a tener ningún efecto en la visualización del fichero PDF... solo lo hace más grande.

Así que solo tenemos que crear un fichero ZIP con los contenidos que deseemos añadir a nuestro PDF y listo.

Y el Programa Java?

Pues hacer que nuestro PDF sea una aplicación Java es un efecto secundario de como Java procesa los ficheros JAR. Un fichero .jar no es más que un fichero ZIP con un fichero de manifiesto. Java, va a buscar la cabecera ZIP en el fichero que le pasemos como parámetro aunque no se encuentre al principio del fichero. Así que lo único que tenemos que hacer es generar un fichero jar con nuestro programa y concatenarlo con el fichero.

Generando el Polyglot

Con todo lo que hemos comentado, la forma de generar el poliglot es muy sencilla. Crearemos un directorio en el que almacenar nuestro código Java, y los ficheros que queramos incluir en el ZIP (pueden estar en subdirectorios si queremos). Luego compilaremos nuestro programa Java y generamos un .jar, incluyendo el resto de fichero que queremos incluir. Algo tal que así:

```
$ javac roor/Stega.java

$ jar cvfe roor-n2.jar roor.Stego roor/Stego,class ./files/*
```

Una vez que tengamos nuestro ZIP/JAR, solo tenemos que concatenarlo al final del fichero PDF, y arreglar el ZIP para que Winzip esté contento

```
$ cat or-n2.pdf roor-n2.jar > or-n2-cool.pdf
$ zip -F or-n2-cool.pdf --out or-n2-final.pdf
```

Como utilizar este PDF

Bueno, damos por hecho que ya sabéis como visualizar el PDF... de lo contrario no estaríais leyendo esto :P. Para extraer los contenidos de las fuentes, inclido el código Java, solo tenéis que ejecutar:

```
$ unzip x or-n2-final.pdf
```

Si lo que queréis es ejecutar el código Java, este es el comando que os interesa:

```
$ java -jar or-n2-final.pdf
```

Como os hemos dicho, el programa nos permite almacenar mensajes secretos en una imágen. Es un poco más limitado que la versión C del artículo "Mensajes Clandestinos"... Sorry, mi Java está un poco oxidado :)

Un fuerte saludo al equipo de PoC | GTFO por inspirarnos para hacer esto y la excelente calidad de su publicación.
https://unpack.debug.su/pocorgtfo/

Compilación, Compilación Cruzada y ToolCh

Investigamos como funciona el compilador en detalle..

por Andrés "Andy" Pajuaker

Estamos seguros de que nuestros ávidos lectores han compilado cientos de programas. En general, no necesitamos tener ni idea de como funciona el proceso. Nuestro querido ./configure && make && make install funciona perfectamente. Pero a veces, necesitamos ir un poquito más allá... por ejemplo cuando necesitamos compilar algo para otra plataforma.

Si sois lectores de esta nuestra humilde publicación, estamos seguros de que en algún momento os vais a ver en la tesitura de compilar algo para alguna plataforma rara. Las cosas han mejorado mogollón en los últimos años, pero aún hoy en día (en pleno 2017 :), de vez en cuando, nos encontramos con alguna plataforma con un soporte limitado. Muchas veces esa plataforma no es muy popular, otras veces es muy nueva y las herramientas habituales no están todavía niqueladas...

En esos casos, creedme, el saber como funciona todo esto del proceso de compilación os puede ahorra muchos quebraderos de cabeza.

Todo el artículo está desarrollado en torno a ejemplos prácticos de forma que podáis ir probando cada una de las cosas que os contamos según vayáis leyendo. Así que sin más... vamos al temita!

Compilación

Vamos a empezar viendo lo que pasa cuando compilamos un programa. Tomemos, por ejemplo, el infame "Hola Mundo". Aquí tenéis nuestra versión:

```
#include  <stdio.h>
int
main ()
{
        printf ("Que pasaaa güorl!\n");
        return 0;
}
```

El infame Hola Mundo... con un twist

Como todos sabéis podemos compilarlo utilizando gcc con un comando tal que asín:

```
$ gcc -o hola hola.c
```

Hasta aquí todo bien. Tenemos un programa llamado `gcc` que toma código fuente en C y genera un ejecutable.... Mal!

Veamos que es lo que realmente está pasando. Para ello utilizaremos la utilidad `strace`. `strace` nos permite ver que llamadas al sistema realiza un determinado programa. Os adelantamos que `gcc` no es el único responsable de generar el ejecutable... como todos nosotros, él también necesita ayuda de vez en cuando y nosotros vamos a enterarnos de quienes son sus compinches.

```
strace -f -e trace=execve gcc -o hola hola.c
2> traza-execve.txt
```

Este comando va a volcar en un fichero todas las llamadas al sistema `execve` que se producen cuando compilamos nuestro programa.

En el hipotético caso de que alguno no lo sepa, la forma de ejecutar programas en un sistema UNIX requiere dos llamadas al sistema. Lo primero que hacemos es ejecutar `fork`, que nos permite crear un nuevo proceso (un nuevo programa) que será una copia exacta del proceso que ejecuta la llamada `fork`. Luego utilizamos la llamada al sistema `execve` para cargar el código que queremos que ejecute ese nuevo proceso.

La utilidad `strace`, por defecto, solo mira en el proceso actual. Si el proceso ejecuta `fork` y crea un nuevo proceso, éste no aparecerá en la salida de `strace`, a no ser que utilicemos el flag `-f`. Con este flag le indicamos a `strace` que también mire que es lo que hace cualquier nuevo proceso creado durante la monitorización del programa inicial. Si el flag `-f` no veríamos gran cosa... simplemente probadlo.

Compinches

Es hora de echa un ojo a la salida de nuestro amigo `strace`. Aún cuando le hemos dicho que solo estamos interesados en las llamadas a `execve`, `strace` muestra alguna información adicional (se podría eliminar pero añadiendo unos cuantos flags más). Para facilitar nuestro análisis vamos a filtrar el fichero usando `awk` de forma que solo el primer parámetro de `execve` sea mostrado. Por qué awk?... Porque es super alucinante!

```
tulchein@dev $ grep execve traza-execve.txt
$  awk 'BEGIN{FS="("} /execve/{print $1}' trace-
execve.txt
execve("/usr/bin/gcc"
[pid  8726] execve("/usr/lib/gcc/x86_64-linux-
gnu/4.6/cc1"
[pid  8727] execve("/usr/lib/lightdm/lightdm/as"
[pid  8727] execve("/usr/local/sbin/as"
[pid  8727] execve("/usr/local/bin/as"
[pid  8727] execve("/usr/sbin/as"
[pid  8727] execve("/usr/bin/as"
[pid  8728] execve("/usr/lib/gcc/x86_64-linux-
gnu/4.6/collect2"
[pid  8729] execve("/usr/bin/ld"
```

Bueno, dejando a un lado los distintos paths que intenta `gcc` para localizar herramientas como `as`, en la salida del comando podemos ver las siguientes herramientas:

- cc1. Este es el compilador C. En realidad es un traductor de lenguaje C a lenguaje ensamblador
- as. Este es el ensamblador de GNU (ASsembler). Este programa convierte programas en lenguaje ensamblador en código objeto
- collect2: Este es un programa que gestiona constructores en el código en la fase de enlazado.
- ld: Este es el enlazador (Linker), el encargado de producir el ejecutable final.

Veamos como funciona todo esto. Primero vamos a convertir nuestro código C en ensamblador usando `cc1` (Figura 1)

Observad que hemos utilizado el mismo path que hemos obtenido de la salida de `strace`. El comando anterior produce un fichero llamado `hola.s`. La extensión `.s` es utilizada para los ficheros en ensamblador. El nuestro tiene esta pinta.

```
        .file   "hola.c"
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-
1ubuntu5) 4.6.3"
        .section        .note.GNU-
stack,"",@progbits
```

Ahora podemos convertir este fichero ensamblador en código objeto utilizando el ensamblador:

```
$ as -o hola.o hola.s
$ file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

Linkando

Ya solo nos quedaría linkarlo... Bueno, el linkado, es un poquitín complicado, incluso para un programa tan sencillo como este. Lo que pasa es

```
$ /usr/lib/gcc/x86_64-linux-gnu/4.6/cc1 hola.c
 main
Analyzing compilation unit
Performing interprocedural optimizations
      Assembling functions:
 main
Execution times (seconds)
 preprocessing         :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (50%) wall     130 kB ( 9%) ggc
 lexical analysis      :   0.00 ( 0%) usr   0.01 (100%) sys   0.00 ( 0%) wall       0 kB ( 0%) ggc
 parser                :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (50%) wall     299 kB (20%) ggc
 TOTAL                 :   0.00             0.01             0.02               1521 kB
```

FIGURA 1. Resultado de la ejecución directa de cc1

```
$  gcc -v hola.c
(... mogollón de cosas ...)
 /usr/lib/gcc/x86_64-linux-gnu/4.6/collect2 --sysroot=/ --build-id --no-add-needed --as-needed --eh-frame-hdr -m
elf_x86_64 --hash-style=gnu -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-
gnu/4.6/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/4.6/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.6 -L/usr/lib/gcc/x86_64-linux-
gnu/4.6/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../../lib -L/lib/x86_64-linux-gnu
-L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/4.6/../../..
/tmp/ccYlYezt.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/x86_64-linux-gnu/4.6/crtend.o /usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-gnu/crtn.o
$
```

FIGURA 2. Resultado de compilar nuestro programa utilizando el flag -v de gcc

que, cualquier programa que utilice la librería C, tiene que enlazarse con ella. Además, todo programa que utiliza la librería C estándar necesita un cierto código de inicialización que nosotros, los programadores de aplicaciones normales y corrientes no solemos ver.

El comando de linkado depende bastante de donde se hayan instalado todos esos fichero, lo cual puede variar de sistema a sistema, de plataforma a plataforma o de versión del compilador a versión del compilador.

Pero bueno, que no cunda el pánico. El propio `gcc` nos va a ayudar a entender que es lo que está pasando. Os presentamos el flag `-v`. La figura 2 muestra la información que obtenemos al compilar nuestro programa con este flag.

Como podéis ver el flag `-v` nos da un montón de información sobre donde se encuentran los distintos ficheros que usa `gcc` así como los flags que añade sin decir ni mú. Lo que nos interesa a nosotros en la última linea. Si amigos, ese es el comando de linkado.

Como podéis ver, el linkado se inicia desde el programa `collect2` que a su vez ejecuta `ld`, el enlazador. Para este programa tan sencillo, podemos ejecutar `ld` directamente en lugar de `collect2`. De cualquier forma, todos esos parámetros que veis en el comando son esos ficheros necesarios para utilizar la librería C estándar, además de la librería en si misma (puedes encontrar a Wally?. Digo `-lc`).

> gcc utiliza varios programas para generar un ejecutable a partir del código fuente C (cc1, as, ld,...)

Si miráis atentamente veréis que en el medio de todo ese galimatías está el fichero `/tmp/ccYlYezt.o`. Este es el nombre temporal que `gcc` asigna a la salida del ensamblador. Nosotros le llamamos `hola.o`, así que si queréis convertir el `hola.o` que generamos antes en un ejecutable debéis hacer lo siguiente:

- Copiad la última línea generado por gcc -v, la que ejecuta la utilizad collect2
- Cambiad collect2 por ld (esto es totalmente opcional)
- Buscad el fichero con el nombre raro y cambiadlo por hola.o
- Añadir -o hola al final si queréis que vuestro programa no se llame a.out :)

Para nuestro caso... el comando quedaría más o menos así:

```
$ ld  --sysroot=/ --build-id --no-add-needed --as-needed --
eh-frame-hdr -m elf_x86_64 --hash-style=gnu -dynamic-linker
/lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-
linux-gnu/4.6/../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-
gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/4.6/crtbegin.o
-L/usr/lib/gcc/x86_64-linux-gnu/4.6 -L/usr/lib/gcc/x86_64-
linux-gnu/4.6/../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../../lib
-L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-
linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-
gnu/4.6/../../.. -lgcc --as-needed -lgcc_s --no-as-needed
-lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/x86_64-linux-gnu/4.6/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-
gnu/crtn.o hola.o
```

Observad que hemos eliminado el fichero en `/tmp` con el nombre raro y hemos añadido `hola.o` al final. Tras ejecutar esto, un flamante binario con nombre a.out estará listo para ser ejecutado.

Toolchains

Lo que acabamos de describir es lo que se conoce como la **ToolChain** (Cadena de Hherramientas) del sistema. En realidad la toolchain contiene algunas herramientas más, pero digamos que las que hemos descrito hasta ahora son las mínimas necesarias para poder compilar un programa C (como acabamos de ver).

Bien, pues cuando queremos compilar un programa para otra plataforma, necesitamos un toolchain para esa plataforma. No solo versiones del compilador, ensamblador, enlazador... también los ficheros de inicialización y las librerías estándard que hemos visto cuando linkábamos

manualmente nuestro hola mundo.

Para ilustrar esto vamos a instalar un toolchain para máquinas ARM. Si tienes una Raspbery Pi, una BeagleBone, un Olinuxino entonces tienes una máquina ARM con la que probar. Es más, si tienes un teléfono Android, lo más seguro es que sea una máquina ARM. Veremos en un segundo como probar nuestro SW en esos dispositivos.

Para los afortunados usuarios de Debian y derivados, la distribución incluye desde hace ya algún tiempo un toolchain para máquinas ARM. Así que... apt-getealo:

```
sudo apt-get install gcc-arm-linux-gnueabi
sudo apt-get install gcc-arm-linux-gnueabihf
sudo apt-get install binutils-arm-linux-gnueabihf
```

El segundo paquete proporciona soporte para Hard Floating point... es decir, para procesadores ARM con soporte hardware para trabajar con números decimales... lo cual, a día de hoy es lo normal. En caso de duda instalad ambos paquetes. El último incluye algunas herramientas adicionales que, si bien no vamos a utilizar en este artículo, os resultarán útiles.

Para los usuarios de otras distribuciones, consultad vuestros repositorios. Si no encontráis un toolchain, descargadlo dela web de Linaro. Solo tenéis que descomprimir el `tar.gz` en algún sitio y añadir el directorio con los binarios al PATH.

Ahora disponéis del toolchain para ARM. Escribid en el terminal `arm-linux-gnueabihf-` y pulsad TAB.... Esto es lo que veréis más o menos:

```
$  arm-linux-gnueabihf-
arm-linux-gnueabihf-addr2line  arm-linux-gnueabihf-ld
arm-linux-gnueabihf-ar         arm-linux-gnueabihf-ld.bfd
arm-linux-gnueabihf-as         arm-linux-gnueabihf-ld.gold
arm-linux-gnueabihf-c++filt    arm-linux-gnueabihf-nm
arm-linux-gnueabihf-cpp        arm-linux-gnueabihf-objcopy
arm-linux-gnueabihf-cpp-4.6    arm-linux-gnueabihf-objdump
arm-linux-gnueabihf-elfedit    arm-linux-gnueabihf-ranlib
arm-linux-gnueabihf-gcc        arm-linux-gnueabihf-readelf
arm-linux-gnueabihf-gcc-4.6    arm-linux-gnueabihf-size
arm-linux-gnueabihf-gcov       arm-linux-gnueabihf-strings
arm-linux-gnueabihf-gcov-4.6   arm-linux-gnueabihf-strip
arm-linux-gnueabihf-gprof
```

Si habéis instalado otro toolchain, el prefijo de las herramientas puede ser diferente. Comprobad el contenido del directorio de binarios para estar seguros

Ese es el listado completo del toolchain en mi máquina. Podéis ver `gcc`, `as`, `ld` y otras muchas herramientas de las que no vamos a hablar todavía. Como podéis ver `cc1` y `collect2` no aparecen. Como ya sabemos esos programas son lanzados automáticamente durante el proceso de compilación o linkado y en principio nunca son utilizados por el usuario directamente.

Podéis repetir el proceso anterior con nuestro nuevo toolchain. Todas las herramientas funcionan igual, solo que empiezan por `arm-linux-gnueabihf-`. Por ejemplo, para producir una versión de hola mundo para ARM solo tenéis que:

```
$ arm-linux-gnueabihf-gcc hola.c -o hola-arm
```

Probando Nuestro Programa

Los que tengáis algún dispositivo ARM ejecutando alguna variante reciente de Debian, podéis copiar este binario (`hola-arm`) en él y ejecutarlo. Debería funcionar sin más. Nosotros lo hemos probado en los dispositivos que tenemos normalmente en línea: Olinuxino, BananaPi y RaspberryPi.

Si tenéis algún problema ejecutando el programa, probad con el otro toolchain (el no hf) en caso de que vuestro dispositivo sea un poco antiguo y no tenga el soporte hardware para los números decimales.

Si nada funciona, probad a compilarlo estáticamente:

```
$ arm-linux-gnueabihf-gcc -static hola.c -o hola-arm
```

El binario estático debe funcionar en cualquier ARM, incluyendo vuestro teléfono Android ARM, siempre y cuanto no estéis utilizando un kernel muy viejo en el dispositivo.

Los binarios dinámicos, como el primero que hemos generado, hacen uso del enlazador dinámico (un programa llamado `ld-ALGO.so`. El `algo` depende de vuestro sistema. Pues bien, este programa se va a encargar de encontrar la librería C estándar en vuestro dispositivo, cargarla, y modificar la imagen en memoria de nuestro programa (actualizar la tabla PLT) para que utilice el `printf` que proporciona la librería local.

Mientras el interfaz de la librería C y el del enlazador dinámico no cambien, el programa debería funcionar sin problemas. Esto puede pasar en sistemas muy antiguos, o en sistemas "distintos", como Android.

El caso de Android

Android no es más que otra distribución Linux, en la que la han liado bien. Todavía tenemos un kernel Linux, igualito que el de nuestro ordenador, pero todo el espacio de usuario se ha modificado...

```
~/android $ export PATH=${HOME}/android/ndk/build/tools:$PATH
~/android $ export PATH=${HOME}/android/sdk/platform-tools/:$PATH
```

FIGURA 2. Configuración del SDK de Android

```
~/android $ make-standalone-toolchain.sh --platform=android-19 --install-dir=${HOME}/android/android-19
--arch=arm
~/android $ PATH=${HOME}/android/android-19/bin:$PATH
```

FIGURA 3. Generando y configurando un Tollchain para Android usando el NDK de Android

desde los scripts de arranque a las aplicaciones de usuario (que normalmente son bytecodes java)... pasando por el enlazador dinámico y la librería estándar C.

En Android, la librería estándar se ha modificado bastante. Incluso su nombre ha cambiado (`bionic` en lugar de `libc` :), y su localización. La localización no debería ser un problema ya que es el trabajo del enlazador dinámico encontrarla, pero es que el enlazador dinámico también es diferente, así que nuestro binario no va a funcionar en un teléfono Android.

Si bien es posible "parchear" todo esto, el proceso es complicado y tedioso, así que, en estos casos, lo mejor es hacernos con el toolchain adecuado. Esto pasa también con algunos routers. Utilizar el toolchain correcto va a hacer vuestras vidas mucho más fáciles.

Para poder compilar nuestro programa para Android, debemos instalar el SDK (Software Development Kit) y el NDK (Native Development Kit) de Google, especialmente este último. La instalación es sencilla, solo hay que descomprimir y añadir al PATH los directorios con los programas que queremos utilizar. Algo como los comandos en la Figura 2.

Si tenéis problemas instalando el SDK/NDK, simplemente echadle un ojo a las páginas de Google... es muy fácil.

Bueno, ahora que tenemos las herramientas instaladas, podemos generar nuestro toolchain. Y esto se hace con un comando como el mostrado en la Figura 3.

En este caso hemos elegido el API 19 que se corresponde con Android 4.4 o superior. Podéis consultar los APIs disponibles en esta página:

http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels [1]

Si no os molesta leer en Inglés, todo esto está explicad: en más detalle en esta serie de artículos

http://papermint-designs.com/dmo-blog/2016-04-awesome-android-extreme-hacking--part-i [2]

Pues, ya está listo. Compilamos, copiamos al teléfono (o emulador) y ejecutamos:

```
$ arm-linux-androideabi-gcc -o hola-arm hola.c
$ adb push hola-arm /data/local/tmp
$ adb shell "chmod 777 /data/local/tmp/hola-arm;
/data/local/tmp/hola-arm"
```

Y así concluye esta primera parte de nuestro alucinante viaje hacia los entresijos más íntimos del antiguo arte de la compilación... Qué va! Es coña!. En el próximo número seguiremos explorando los detalles del proceso de compilación. Esperamos que os haya resultado interesante.

Imagen de Cabecera: MAROQUOTIDIEN PLUS [3]

Enlaces Externos

[1] http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels

[2] http://papermint-designs.com/dmo-blog/2016-04-awesome-android-extreme-hacking--part-i

[3] https://unsplash.com/@maroquotidien/

Sobre el Autor

Andres "Andy" Pajuaker

En lo que a seguridad se refiere Andy es un crack. Conocedor de los secretos más oscuros de tu sistema operativo, es capaz de extorsionarlo para conseguir de el lo que quiera. Andy es licenciado por la Universidad del Humor y recientemente a defendido su tesis: "Origen Epistemológico de los Chiste Paramétricos en Entornos en bebidos", en la que contó con la ayuda inestimable de Jim "Tonys".

LOS TRUQUIS DE OCCAM´S

Brujerías en la línea de Comandos. Los mejores trucos

por Tamariz el de la Perdíz

Imagen cortesía de
https://unsplash.com/@sebastian_unrau

Lista entradas repetidas en un fichero de texto

Una forma sencilla de conseguirlo es utilizando los comandos `sort` y `uniq`. Algo tal que asín:

```
cat fichero.txt | sort | uniq  -D
```

En ocasiones nos puede interesar los comparar un número determinado de caracteres. El flag `-w` nos permite hacerlo de forma sencilla:

```
cat fichero.txt | sort | uniq -w 32 -D
```

Códigos QR

Los códigos QR (esos cuadrados llenos de cuadraditos) resultan bastante útiles para intercambiar pequeñas cantidades de información. Podemos utilizarlos de forma muy sencilla desde la línea de comandos con las utilidades `qrencode` y `zbarimg`. Aquí tenéis un ejemplo:

```
$ qrencode -o image.png 'Tus datos aquí'
$ zbarimg --raw image.png
```

vim. Una caja de sorpresas

Nunca terminamos de maravillarnos con las posibilidades de `vim`. Aquí van un par te trucos interesantes.

Firmar un mensaje de texto con gpg dentro de vim

Utilizando el comando ! podemos ejecutar, fácilmente, gpg sobre el documento de texto que estamos editando con vim, y firmarlo digitalmente.

```
:%!gpg --clearsign`
:w file.asc
```

Codificación ROT13 del texto actual

No es que sea lo más útil del mundo, pero en más de una ocasión me ha venido muy bien.

```
ggg?G
```

Crear ficheros de texto encriptados

Sí, podemos editar ficheros encriptados con vim de forma muy sencilla...

```
vim +X file_name
```

o si ya tenemos el fichero abierto

```
:X
```

Encontrar que proceso está bloqueando un determinado fichero

A veces algún proceso al que le hemos perdido la pista mantiene algún fichero bloqueado. Podemos saber cual es el proceso problemático utilizando `fuser` o `lsof`.

```
fuser path_to_file
lsof | grep file
```

Eliminar información EXIF

Las imágenes almacenan un montón de información de la que muchas veces no somos conscientes. Fechas, lugares (si un GPS estaba activo en el momento de sacar la foto), typo de dispositivo, etc... En ocasiones nos puede interesar eliminar toda esa información de nuestros ficheros. La utilidad exiftools nos lo pone fácil

```
exiftool -all= image.jpg
```

Envíanos tus Trucos

Tienes trucos alucinantes que deseas compartir con el mundo?. Escríbemos a root AT papermint-designs.com y cuéntanos.

Incluye la información que deseas que aparezca junto con tu truqui. Nombre, dirección de correp, blog/website... Tu decides!

CONSEGUIDO

Lo hemos vuelto a hacer... Y esta vez hemos tardado menos de un año LoL

Imagen cortesía de https://unsplash.com/@nhinton4

BLOG GitHub YouTube

Entérate antes que nadie de las últimas novedades de Occam's Razor