Protecciones de GNU/Linux para exploits
de stack y heap buffer overflow

Tomás Touceda (LU:84024)
Director: Mg. Javier Echaiz
15 de Julio de 2011

Índice general

Capítulo 1

Introducción

1.1. Contexto actual

En la actualidad los ataques de los que más se ven noticias son los que actúan principalmente sobre aplicaciones web, a través de técnicas como SQL Injection, Cross Site Scripting (XSS), Distributed Denial of Service (DDoS), etc.

Más alla de la técnica usada en los ataques, los objetivo siempre suelen ser los mismos: obtener información, robar identidad, robo de dinero (también relacionado con el robo de identidad), y actualmente se ven también ataques con el mero objetivo de protesta.

Por otro lado, y no de forma tan pública, existe un grave problema: las botnets. Las botnets son redes de computadoras controladas por un software autónomo. Si bien el término se puede aplicar a muchos tipos de ámbitos computacionales, en general está ligado a redes de computadoras que han sido tomadas por algún tipo de malware sin que el dueño de la computadora lo note y son usadas para enviar spam o realizar ataques DDoS. La ganancia monetaria de las botnets viene dada a través de las empresas o personas que rentan tiempo de la misma para publicitar sus productos o servicios a través del spam.

La pregunta que puede surgir es: ¿Cómo nacen las botnets?. Como en todo lo referido a seguridad, un sistema es tan fuerte como su eslabon más

debil, y esto siempre ocurre cuando existe el factor humano. El software llamado "malware" puede ingresar en una computadora víctima de muchas maneras, una forma muy común es engañando al usuario a instalar la aplicación maliciosa, y otra es a través del explotado de vulnerabilidades en el sistema operativo, o en aplicaciones que corren en él y no son propiamente protegidas por el mismo. Si bien pueden existir problemas de seguridad tanto en la capa de aplicación como en la capa del sistema operativo, existe una relación implícita, en cierto sentido, en las herramientas que tiene el sistema operativo para defender una aplicación mal programada en caso de un ataque.

Dado que el problema que proviene del fator humano resulta muy complejo de tratar, en general se tratan soluciones o mejoras para el otro aspecto del problema: el sistema operativo. Un malware puede acceder y tomar control de la computadora a través del explotado de buffer overflow en el stack, o en el heap, y si el sistema operativo residente no posee formas de contingencia de ese problema entonces no existen límites para el atacante.

Desde 1996, año en el cual Aleph1 publicó su artículo "Smashing the stack for fun and profit" en la revista electrónica Phrack, las técnicas de explotado de vulnerabilidades han evolucionado rápidamente. Contrario a lo que se puede esperar, el hecho de que las vulnerabilidades se hagan cada vez más públicas no ha llevado a que los desarrolladores de software sean realmente conscientes de estos problemas, y programen para evitarlos.

Dada esta realidad, sistemas operativos como GNU/Linux han desarrollado a lo largo del tiempo distintas formas de protección "implícita" que opere sobre las aplicaciones vulnerables. En este trabajo se tratarán de explicar de qué tratan las técnicas más efectivas de protección y cómo operan para lograr su objetivo.

1.2. Estructura del trabajo

- **Capítulo 1**: Introducción

  Explicación básica de la idea detrás del presente trabajo.

- **Capítulo 2**: Problemas básicos de seguridad

  En este capítulo se introducirán las ideas básicas detrás de los problemas de seguridad del software más comunes: buffer overflows en el stack y el heap.

- **Capítulo 3**: Protecciones estáticas

  Dentro de las protecciones, se puede realizar una clasificación en "estáticas" y "dinámicas". Las estáticas involucran aquellas protecciones que son compiladas dentro del ejecutable final de la aplicación, y son las tratadas en este capítulo.

- **Capítulo 4**: Protecciones dinámicas

  Las protecciones dinámicas son aquellas que, si bien han sido compiladas y son estáticas en cierta forma, existen más alla de cómo se haya compilado la aplicación vulnerable, ya sea porque residen en el kernel del sistema, o porque forman parte de una biblioteca enlazada dinámicamente.

- **Capítulo 5**: Conclusiones

  Delineadas las protecciones más comunes y efectivas de GNU/Linux, en este capítulo se presenta una conclusión acerca del panorama actual.

1.3. Aclaraciones

Todos los ejemplos presentados en este documento son compilados y ejecutados en un entorno Gentoo Linux i686 con gcc-4.4.4.

Capítulo 2

Problemas básicos de seguridad

2.1. Stack based overflow

Los stack based overflow son históricamente los más populares y mejor entendidos métodos de explotación de software. Fueron documentados por primera vez por Aleph One en el artículo "Smashing the stack for fun and profit" de la revista electrónica Phrack. Este tipo de problemas existe desde que existe el lenguaje de programación C, y a pesar de ser tan comunes y conocidos son los bugs que más existen en el software actualmente. El problema de los stack overflows viene dado por mezclar datos con información de control en la aplicación en cuestión que conlleva a poder tomar el control de la ejecución a través de la manipulación de los datos.

Un buffer es una sucesión de locaciones de memoria finita, comúnmente representada como un arreglo en el lenguaje C. El acceso a dicha estructura de memoria no posee ningún tipo de checkeo de límites y por ello es que puede ocurrir un overflow o desbordamiento de la misma. En el caso de los stack overflows, esto ocurre cuando la aplicación almacena la estructura de datos en el stack, ya sea por ser un parámetro pasado por valor a una función o por ser una variable local a la misma, y no se corroboran los límites sobre los cuales se copian datos hacia la estructura. Por ejemplo:

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

```
int main(int argc,char **argv) {
  int array[5] = {1, 2, 3, 4, 5};

  printf("%d\n", array[5]);

  return 0;
}
```

Aqui el error reside en no haber recordado que si bien el array posee 5 elementos, la numeración comienza en cero, por lo que se están referenciando los 4 bytes (u 8 en arquitecturas de 64 bits) siguientes al de la última locación del arreglo definido. Al momento de compilación, este error no es detectado, y el resultado es el siguiente:

```
chiiph@delloise ~/src/tesis/tmp () $ gcc reference.c -o
    reference -Wall
chiiph@delloise ~/src/tesis/tmp () $ ./reference
134513712
```

Aun habilitando todos los warnings de compilación, el error no es detectado por el compilador, y se puede observar la locación lógica en decimal del byte siguiente al final del arreglo de enteros.

Este tipo de flexibilidad provista por el lenguaje C es la fuente de este tipo de problemas.

2.1.1. Llamada a funciones y el Stack

En arquitecturas en las cuales la cantidad de registros es limitada como en x86, el stack es utilizado para pasaje de parámetros a la hora de llamar a funciones, y para mantener la sucesión de llamadas.

Veamos un ejemplo:

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

```
void function(int a) {
  printf("Inside function: a=%d\n", a);
}

int main(int argc, char **argv) {
  function(4);

  return 0;
}
```

Analizando el código ensamblador que resulta de compilar el ejemplo anterior:

```
chiiph@delloise ~/src/tesis/tmp () $ gdb --quiet call
Reading symbols from /home/chiiph/src/tesis/tmp/call...(no
    debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x080483ec <+0>: push   %ebp
   0x080483ed <+1>: mov    %esp,%ebp
   0x080483ef <+3>: and    $0xfffffff0,%esp
   0x080483f2 <+6>: sub    $0x10,%esp
   0x080483f5 <+9>: movl   $0x4,(%esp)
   0x080483fc <+16>:  call   0x80483d0 <function>
   0x08048401 <+21>:  mov    $0x0,%eax
   0x08048406 <+26>:  leave
   0x08048407 <+27>:  ret
End of assembler dump.
(gdb)
```

La primer instrucción guarda en la pila el registro EBP (Extended Base Pointer), que referencia la base del stack. Luego se establece como nuevo Base Pointer el actual ESP (Extended Stack Pointer), de esta forma la función que luego será llamada operará con el nuevo stack "acotado". Luego se le resta al stack pointer 0x10 (16) para reservar la memoria necesaria para los valores necesarios almacenados para la llamada a *function*: 4 bytes para el parámetro

a de *function*, 4 bytes para la dirección de retorno, y los 4 bytes del EBP que se realizó el push anteriormente.

Como se ve, el stack crece hacia las direcciones más bajas. Cuando un item es agregado a la misma, el ESP es decrementado, pero el item en sí mismo se referencia hacia las direcciones mayores. Es decir, si el item almacenado es un arreglo, la dirección del elemento 0 del arreglo será menor que la del elemento 1. Esta propiedad del manejo de memoria hace que si el arreglo del que se habló se escribiera más alla de los límites de tamaño del mismo, se sobreescribirían los elementos que se encuentran "por debajo de él", es decir, la dirección de retorno y el EBP guardado. Si se puede controlar la dirección de retorno de una función, se puede controlar la ejecución de la aplicación a partir de ese punto. Hacia donde se dirige la ejecución y cómo se lo aprovecha depende de muchos factores, y de cómo está protegido el entorno donde se ejecuta la aplicación.

2.1.2. Ejemplo básico

A modo de ejemplo, se muestra un stack based overflow muy simple:

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

int main(int argc,char **argv) {
  char buf[256];

  strcpy(buf,argv[1]);

  return 0;
}
```

En este ejemplo la estructura utilizada para sobrecargar el stack es el array *buf*, cuyo tamaño es fijo y establecido en la definición del mismo, por ello es almacenado en el stack. Sobre esta estructura se realiza la copia de la cadena de carácteres pasada por parámetro a la aplicación. El problema reside en no corroborar el tamaño de la cadena copiada en *buf*. Si el primer parámetro

de la llamada a la aplicación excede los 256 bytes, se sobreescribirá cualquier dato que esté almacenado contiguamente en el stack.

```
chiiph@delloise ~/sec/abos/abo1 () $ ./ejemplo1 $(python -c "
    print 'A'*500")
Segmentation fault

chiiph@delloise ~/sec/abos/abo1 () $ gdb ejemplo1
Reading symbols from /home/chiiph/src/tesis/tmp/ejemplo1...(no
    debugging symbols found)...done.
(gdb) r $(python -c "print 'A'*500")
Starting program: /home/chiiph/src/tesis/tmp/ejemplo1 $(python -
    c "print 'A'*500")

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Se puede ver claramente como pasarle más de 256 bytes como entrada causa un *Segmentation Fault*, y corriéndola dentro de gdb podemos ver por qué sucede lo propio. La dirección 0x41414141 no es válida, y en particular, 0x41 es el valor en hexadecimal de la letra A.

En general, se busca sobreescribir el valor de retorno de la llamada a función para redirigir la ejecución al contenido del stack, o a la locación de una función de la *glibc* utilizando el stack para pasar parámetros a ella, y de esta forma tomar control de la aplicación.

2.1.3. Algunos tipos de ataques

Los overflows basados en stack son muy simples de entender, pero no siempre son tan simples de explotar. En esta sección, se enumerarán alguna de las técnicas usadas para aprovechar esta vulnerabilidad, y cómo técnicas de protección que a veces se utilizan no son realmente útiles.

Filtros en la entrada

En muchas aplicaciones la entrada a partir de la cual ocurre el overflow puede que esté filtrada, ya sea porque los carácteres que espera tienen que ser en minúscula o mayúscula, o tal vez utiliza caráteres unicode. Por lo que el shellcode que se utilice para explotar la aplicación puede que sea alterado, o que el mismo no funcione por contener carácteres inválidos.

Esto puede llevar a pensar que funciona en algún modo como protección para ciertos shellcodes. Pero es muy facil de saltear. Los filtros, ya sea una aplicación open source o de código cerrado, son muy fáciles de predecir, por lo que se puede producir una sucesión de carácteres que luego del filtrado se transforme en el shellcode deseado. O se puede realizar uno libre de carácteres arbitrarios inválidos para la aplicación.

Setuid

A la hora de explotar una vulnerabilidad de este tipo en una aplicación, se busca explotar procesos de binarios con SUID, es decir, aplicaciones que se llaman en modo usuario pero cambian el usuario efectivo al administrador del sistema para ejecutar instrucciones privilegiadas. Entre este tipo de aplicaciones se encuentran ping, sudo, etc.

Lo que muchas aplicaciones hacen para evitar correr completamente como administrador es setear el usuario efectivo como el que está ejecutando la aplicación hasta que sea necesario hacer uso de codigo que no se puede correr en modo de usuario, como crear un socket en la aplicación ping.

```
setuid(getuid());

// Codigo no privilegiado
...

setuid(0);

// Codigo privilegiado
...
```

Esto da una falsa sensación de seguridad, ya que la aplicación sigue teniendo el SUID bit habilitado. De esta forma, si se logra ejecutar código arbitrario a través de alguna vulnerabilidad, es solo cuestión de ejecutar *setuid*(0) y no importará qué usuario efectivo la aplicación había establecido.

Existen sistemas GNU/Linux que saltean este grave problema haciendo uso de las *filesystem capabilities*. Openwall GNU/*/Linux, un proyecto con más de 10 años de desarrollo, utiliza esta estrategia, entre muchas otras, para asegurar el sistema.

chroot

Los servidores HTTP, y de otros servicios remotos, suelen utilizar los *chroots* para correr cada servicio en un entorno separado del resto del sistema, simulando un encapsulamiento del mismo.

La idea de diseño puede parecer muy efectiva, aun cuando un usuario logre tomar control del sistema a través de alguna vulnerabilidad, solo verá el *subsistema* acotado por el *chroot*. Pero de nuevo, esto no representa una seguridad real. Una vulnerabilidad en el sistema de *chroot* hace que ejecutando el siguiente comando se "rompa" el chroot y se tenga acceso al sistema completo:

```
void main() {
  mkdir("sh",0755);
  chroot("sh");
  chroot("../../../../../../../../../../../../../../../../");
}
```

Existe una alternativa a los *chroots* presente en sistemas BSD, llamados *jails*, que no poseen este grave problema.

Sistemas de detección de intrusos

Los sistemas de detección de intrusos (IDS) funcionan de forma similar a los firewalls, analizando el tráfico de entrada al sistema a través de la red buscando patrones conocidos que pueden representar una amenaza.

Los shellcodes utilizados, para ejecutar una shell, o abrir puertos en el sistema vulnerable, etc, suelen ser siempre los mismos, o muy similares. Estos patrones son los que el IDS busca, y bloquea.

El problema reside en la premisa de que los shellcodes suelen ser similares. Existe una técnica, llamada *shellcode polimórfico* que funciona como contraejemplo a la premisa.

La idea inicial de polimorfismo consistía en cifrar el shellcode, y generar una secuencia de descifrado que procese la secuencia de bytes cifrado y lo "convierta" en una secuencia de código ejecutable. Diferentes formas de cifrado, hacen que el mismo shellcode se vea diferente a los ojos del IDS, y de esta forma no detectar que es una secuencia de instrucciones que comprometerán al sistema cuando la aplicación vulnerable reciba este paquete de datos de la red.

Veamos la idea con el siguiente shellcode:

```
push byte 0x68
push dword 0x7361622f
push dword 0x6e69622f
mov ebx,esp

xor edx,edx
push edx
push ebx
mov ecx,esp
push byte 11
pop eax
int 80h
```

La sección que ejecuta $execve("/bin/bash", ["/bin/bash", NULL], NULL)$ se codifica en el shellcode como la siguiente secuencia de bytes:

```
"\x6A\x68\x68\x2F\x62\x61\x73\x68\x2F\x62\x69\x6E\x89\xE3\x31\
    xD2"
"\x52\x53\x89\xE1\x6A\x0B\x58\xCD\x80"
```

Si el IDS analiza un paquete que contiene esta sucesión de carácteres, sabrá que es parte de un shellcode para ejecutar una shell y lo bloqueará.

Si el mismo shellcode es codificado, tal vez con técnicas tan simples como realizar un XOR de cada byte, se generará una sucesión de carácteres diferente. A la vez, el código de descifrado puede contener código de relleno que haga que tampoco pueda ser facilmente detectado por su *signature*.

En conclución, los IDSs funcionan para ciertos tipos de ataques, pero no son muy complejos de saltear.

Egg hunting

Existe el caso en el cual la cantidad de bytes de entrada es realmente acotada, por lo que de existir un buffer overflow, tal vez no haya ningún shellcode válido lo suficientemente pequeño para ser inyectado. Esto puede verse como una protección, pero obviamente no presenta una real barrera. Existe una técnica llamada *egg hunting*, la cual consiste en inyectar un shellcode de muy pocas instrucciones cuyo objetivo sea buscar el espacio de memoria de la aplicación hasta encontrar un cierto *signature* que representará el shellcode real.

Un ejemplo de este tipo de ataques se vio en una vulnerabilidad en el navegador web Konqueror, cuyo parser HTML poseía buffer overflows con colores cuyo valor era demasiado largo, y similares con otros tag. De esta forma, puede que sea posible inyectar el *egghunter* en el tag de color, y el shellcode real en alguna otra sección del código HTML.

2.1.4. Return to library

Uno de los primeros pasos para la proteccion contra buffer overflows basados en el stack fue establecer que la sección de memoria que representa el stack sea ejecutable o escribible, pero no ambas. En un principio, esta protección fue realizada por software, como parte del kernel Linux, para luego pasar a ser implementada por hardware en los microprocesadores. Esto solucionó ataques clásicos de inyección de código a través del buffer que desataba la vulnerabilidad, pero al no resolver la raíz del problema, no pasó mucho

tiempo hasta que técnicas alternativas de explotado surgieron. Estas técnicas son las llamadas "return to library" en general, ya que sobreescriben la dirección de retorno con la dirección de alguna operación perteneciente a una biblioteca del sistema, y utilizan el stack para pasarle parámetros a la misma.

ret2libc

Esta técnica hace uso de la biblioteca *libc*, presente en todo sistema GNU/Linux, y particularmente de la operación *system*, que sobreescribe el espacio de memoria actual con la del archivo binario pasado como párametro y lo ejecuta.

De esta forma, que el stack no sea ejecutable no impide la ejecución de una shell, por ejemplo.

ret2strcpy

Otra posibilidad es utilizar la biblioteca de manejo de strings, específicamente la operación strcpy. De esta forma, se puede copiar código inyectado en el stack a una sección que sí sea ejecutable en el heap, y retornar hacia la dirección donde comienza este espacio de memoria.

Return oriented code

Ambas técnicas anteriores pueden ser detenidas simplemente evitando que ciertas operaciones de las bibliotecas sean cargadas en memoria, pero eso puede ser salteado con un método un poco más complejo: return oriented code.

Los ataques que utilizan esta técnica no realizan ningún tipo de llamado a funciones. De hecho, utiliza secuencias de intrucciones de la libc que no necesariamente fueron compiladas de forma explícita en ella.

Esta técnica combina un gran número de secuencias de instrucciones cortas utilizadas para contruir *gadgets* que permiten computación arbitraria. Veamos esta idea con un ejemplo:

El siguiente es un fragmento de la función *ecb_crypt* de la GNU LibC, a la izquierda se pueden ver los carácteres en hexadecimal que resultan de

compilar la instrucción de la derecha.

```
f7 c0 07 00 00 00    test $0x00000007, %edi
0f 95 45 c3          setnzb -61(%ebp)
```

Si analizamos la misma secuencia comenzando en el byte despues del $f7$

```
c0 07 00 00 00 0f    movl $0x0f000000, (%edi)
95                   xchg %ebp, %eax
45                   inc %ebp
c3                   ret
```

La frecuencia con que estas situaciones ocurren depende de las características del lenguaje en cuestión. Particularmente, el set de instrucciones de la arquitectura x86 es muy denso, lo que implica que una sucesión aleatoria de bytes puede ser interpretada como distintas intrucciones con una muy alta probabilidad. Por lo que, en código x86 es muy facil encontrar no solo palabras de memoria que tengan significados no intencionados, sino sucesiones enteras de palabras. Para que una secuencia sea util en este tipo de ataques, tiene que terminar en la instrucción de retorno (byte c3).

A la hora de construir un ataque con esta técnica, la forma de interacción con la libc es muy distinta a las otras técnicas en los siguientes puntos:

- Las secuencias de código que se ejecutan son cortas (dos o tres instrucciónes) y, cuando son ejecutadas por el procesador, realizan muy poco trabajo neto. En ataques típicos de ret2libc, la base sobre la cual se construye el ataque son funciones enteras que representan mucha más funcionalidad que meras instrucciones.

- Las secuencias de código que se utilizan no poseen ni el típico comienzo de una llamada a función, ni las intrucciones típicas de fin de la llamada, como se explicó en la sección 2.1.1.

- Las secuencias de código que se utilizan no son llamados de una forma estandar y uniforme, a diferencia del llamado a funciones.

Las primeras formas de estos ataques utilizaban una única secuencia corta código de la libc. Especialmente secuencias de bytes que representaban instrucciones como *pop%reg* para setear registros cuando estos son usados para pasaje de parámetros en arquitecturas como SPARC o x86_64. Otras secuencias utilizadas son las que encadenan instrucciones de intruccion *pop%reg* y *ret*. Estas secuencias de código son usadas en estos ataques para redireccionar la ejecución a código en una dada locación o para realizar llamadas a funciones.

2.2. Heap based overflow

Además de los buffers almacenados en el stack, otro tipo de buffers muy importantes son los que residen en el heap.

Un proceso en Linux posee diferentes segmentos de memoria, muchos de los cuales es imposible saber cuanta memoria es necesaria en tiempo de cargado, por lo que estos segmentos son reservados generalmente en tiempo de ejecución. Estas secciones se pueden ver con gdb:

```
(gdb) maintenance info sections
Exec file:
    `/home/chiiph/src/tesis/tmp/heap', file type elf32-i386.
    0x8048154->0x8048167 at 0x00000154: .interp ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x8048168->0x8048188 at 0x00000168: .note.ABI-tag ALLOC LOAD
         READONLY DATA HAS_CONTENTS
    0x8048188->0x80481bc at 0x00000188: .hash ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x80481bc->0x80481dc at 0x000001bc: .gnu.hash ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x80481dc->0x804825c at 0x000001dc: .dynsym ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x804825c->0x80482bd at 0x0000025c: .dynstr ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x80482be->0x80482ce at 0x000002be: .gnu.version ALLOC LOAD
        READONLY DATA HAS_CONTENTS
```

```
    0x80482d0->0x80482f0 at 0x000002d0: .gnu.version_r ALLOC
        LOAD READONLY DATA HAS_CONTENTS
    0x80482f0->0x80482f8 at 0x000002f0: .rel.dyn ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x80482f8->0x8048328 at 0x000002f8: .rel.plt ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x8048328->0x804833f at 0x00000328: .init ALLOC LOAD
        READONLY CODE HAS_CONTENTS
    0x8048340->0x80483b0 at 0x00000340: .plt ALLOC LOAD READONLY
         CODE HAS_CONTENTS
    0x80483b0->0x80485c8 at 0x000003b0: .text ALLOC LOAD
        READONLY CODE HAS_CONTENTS
    0x80485c8->0x80485e4 at 0x000005c8: .fini ALLOC LOAD
        READONLY CODE HAS_CONTENTS
    0x80485e4->0x8048620 at 0x000005e4: .rodata ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x8048620->0x8048624 at 0x00000620: .eh_frame ALLOC LOAD
        READONLY DATA HAS_CONTENTS
    0x8049f0c->0x8049f14 at 0x00000f0c: .ctors ALLOC LOAD DATA
        HAS_CONTENTS
    0x8049f14->0x8049f1c at 0x00000f14: .dtors ALLOC LOAD DATA
        HAS_CONTENTS
    0x8049f1c->0x8049f20 at 0x00000f1c: .jcr ALLOC LOAD DATA
        HAS_CONTENTS
    0x8049f20->0x8049ff0 at 0x00000f20: .dynamic ALLOC LOAD DATA
         HAS_CONTENTS
    0x8049ff0->0x8049ff4 at 0x00000ff0: .got ALLOC LOAD DATA
        HAS_CONTENTS
    0x8049ff4->0x804a018 at 0x00000ff4: .got.plt ALLOC LOAD DATA
         HAS_CONTENTS
    0x804a018->0x804a020 at 0x00001018: .data ALLOC LOAD DATA
        HAS_CONTENTS
    0x804a020->0x804a028 at 0x00001020: .bss ALLOC
    0x0000->0x005a at 0x00001020: .comment READONLY HAS_CONTENTS
(gdb)
```

El heap es el área de memoria utilizada por las aplicaciones para reservar memoria dinámicamente en tiempo de ejecución. Este tipo de buffer overflow

son comunes como los de stack, pero muy distintos a la hora de explotarlos. A diferencia de los basados en el stack, los heap overflow puede ser muy inconsistentes y pueden ser explotados de diversas formas, con distintas consecuencias.

La mayor diferencia entre el stack y el heap es que es persistente entre llamadas a funciones, la memoria permanece reservada hasta que es explícitamente liberada. Esto implica que un heap overflow puede ocurrir en un dado momento, pero no ser notado hasta tiempo después cuando ese espacio del heap es utilizado nuevamente. En este tipo de sobrecargado de memoria no existe la idea de sobreescribir el registro EIP salvado, pero otros datos son almacenados en el heap que pueden ser manipulados por sobrecargas de estos buffers dinámicos.

El heap se encuentra en el mismo segmento de memoria que el stack, pero a diferencia de este último, crece hacia las direcciones altas de memoria. La memoria es reservada en este espacio a través de las funciones del estilo del *malloc* (*HeapAlloc*(), *malloc*(), *new*(), etc). De forma análoga se liberan los espacios utilizados con funciones del estilo del $free$ (*HeapFree*(), $free$(), *delete*(), etc). El manejo del heap es realizado por un $HeapManager$, que maneja la reserva y liberación de memoria de forma dinámica.

A diferencia del manejo de memoria con el stack, el manejo dinámico de memoria debe ser manejado explícitamente. En este texto nos concentraremos en el lenguaje C, y las implementaciónes del $HeapManager$ de la GNU LibC, ya que no son uniformes a lo largo de las distintas plataformas existentes.

De forma general, se puede considerar al heap como un conjunto de bloques, algunos en uso y otros libres. Estos bloques estan reservados en el heap de forma contigua, con información de control entre cada bloque de datos puro.

2.2.1. Ejemplo Básico

Veamos un ejemplo de overflow simple en el heap:

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

int main(int argc, char **argv) {
  char *input = malloc(20);
  char *output = malloc(20);

  strcpy(output, "normal output");
  strcpy(input, argv[1]);

  printf("input at %p:%s\n", input, input);
  printf("output at %p:%s\n", output, output);

  printf("\n%s\n", output);

  return 0;
}
```

Compilando y ejecutando el ejemplo:

```
chiiph@delloise ~/src/tesis/tmp () $ gcc -o heap heap.c -Wall
chiiph@delloise ~/src/tesis/tmp () $ ./heap asajdioasj doisad
input at 0x804b008:asajdioasj
output at 0x804b020:normal output

normal output
chiiph@delloise ~/src/tesis/tmp () $ ./heap $(python -c 'print "
    A"*40')
input at 0x804b008:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
output at 0x804b020:AAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAA
```

Por lo que sobreescribir variables en el heap es bastante simple, y no necesariamente las aplicaciones abortan como ocurre con los stack overflow.

2.2.2. Primeros ataques

Un gran problema a la hora de explotar un heap overflow es que el manejo de los bloques de memoria reservados y liberados es dependiente de la implementación. Por ello, en una aplicación multiplataforma, puede ocurrir un heap overflow en una plataforma y no en otra.

Muchas implementaciones guardan información de control entre cada bloque de datos. Esto permite a un atacante potencialmente sobreescribir secciones específicas de memoria de forma tal que al ser usada por *malloc* sobreescriba virtualmente cualquier locación en la memoria que se desee.

Para llevar a cabo un ataque exitoso, hay que estudiar la implementación específica del *malloc* en la plataforma en la cual se quiera llevar a cabo el ataque. En este caso, por cuestiones de analizar partiendo de lo más básico, se analizará la implementacion de Doug Lea que pertence a la GNU LibC 2.3. Actualmente, la GNU LibC más utilizada es la 2.11, cuya implementación resuelve muchos problemas, pero aun es explotable, como veremos más adelante.

Si bien estos ataques que se mostrarán a continuación son mayoritariamente obsoletos, sirven para graficar las particularidades de explotar un heap overflow.

dlmalloc

Las secciones de memoria reservada por *malloc* poseen *boundarytags*, que son campos que contienen información acerca del tamaño de las secciones de memoria reservadas antes y después de la actual, asi como punteros a la sección anterior y a la siguiente. Además en el tamaño se encuentran codificadas flags, el tamaño siempre es múltiplo de 8, por lo que los últimos 3 bits se encuentran libres.

En otras palabras, el heap se ve como una sucesion de bloques de tamaño variable, y listas enlazada (*bins*) de bloques libres clasificados según el tamaño del bloque.

El heap es manejado a traves de una sección de memoria llamada bloque *desierto*, cuando se reserva un espacio nuevo de memoria, *malloc* divide el

bloque desierto. Cuando un bloque reservado es liberado por una llamada a $free()$, puede ser unido al bloque anterior a él (*backward consolidation*) o el siguiente (*forward consolidation*) si están libres. De esta forma se asegura que no exitirán dos bloques contiguos libres en memoria. El bloque resultante es finalmente enlazado a la lista doblemente enlazada de bloques libres de tamaño similar al liberado.

El manejo de los bloques libres es similar al manejo clásico de listas doblemente enlazadas. Si P es el bloque a librerar, se reemplaza el puntero del bloque siguiente a P (BK) con el del bloque anterior a P. El puntero del bloque anterior a P (FD) se reemplaza con el del bloque siguiente a P. Es decir,

```
*(P->fd+12) = P->bk;
*(P->bk+8) = P->fd;
```

En la primer sentencia se le suma 12 al puntero por ser 4 bytes para el tamaño del bloque, 4 bytes para el tamaño del bloque anterior, y 4 bytes para fd. En la segunda sentencia, son 4 bytes para el tamaño y 4 bytes para el tamaño del bloque anterior.

En otras palabras, la dirección (o cualquier dato) contenida en el puntero al bloque anterior de un bloque es escrita en la locación referenciada por el puntero al bloque siguiente más 12. Si un atacante puede sobreescribir estos dos punteros y forzar la llamada a la función $free()$, puede sobreescribir cualquier locación de memoria.

Analicemos ahora el manejo de los *bins* de bloques libres. Los bloques dentro de un *bin* son organizados en orden de tamaño decreciente. Bloques del mismo tamaño son enlazados con los que fueron liberados más recientemente en el frente de la lista y son tomados a la hora de reservar memoria del final de la lista. Es decir, presenta un manejo de tipo FIFO.

Cuando se libera un bloque se busca el índice de un *bin* correspondiente al tamaño del bloque liberado, luego se marca el *bin* elegido para establecer que no esta vacío. Luego se determina la dirección de memoria donde está almacenado el *bin*, y por último se guarda el bloque liberado en el lugar

apropiado dentro del *bin*.

Teniendo en cuenta esto, se puede observar lo siguiete:

- Si no existen bloques adyacentes al liberado, este será puesto en el *bin* que corresponde.
- Si el bloque contiguo en memoria al liberado está libre, y si este este último es la frontera del bloque *desierto*, entonces ambos son cosolidados con el bloque *desierto*.
- Si el bloque siguiente o anterior al bloque liberado está libre, estos son sacados de los *bins* a los que pertenecen con el procedimiento ya descripto, son consolidados y puestos en el nuevo *bin* que les corresponde con el nuevo tamaño.

Ahora supongamos que la aplicación vulnerable reserva dos bloques adyacentes en memoria (bloque A y B). El bloque A presenta un buffer overflow que permite sobrecargarlo hasta sobreescribir el bloque B. Se contruye los datos de la sobrecarga de forma tal que cuando $free(A)$ es llamado, el algoritmo decide que el bloque siguiente a A está libre (no necesariamente es el bloque B). De esta forma, $free(A)$ trata de realizar *forward consolidation* entre A y un bloque C. En la sobreescritura, también se le da al bloque C los punteros fd y bk tal que cuando se lo saque del *bin* al que teóricamente está enlazado, sobreescriba una locación de memoria a elección.

2.2.3. Formas en las que se presentan los heap overflow

- Samba:

```
memcpy(array[user_supplied_int], user_supplied_buffer,
    user_supplied_int2);
```

- Microsoft IIS:

```
buf=malloc(user_supplied_int+1);
memcpy(buf,user_buf,user_supplied_int);
```

```
buf=malloc(strlen(user_buf+5));
strcpy(buf,user_buf);
```

- Solaris Login:

```
buf=(char **)malloc(BUF_SIZE);
while (user_buf[i]!=0) {
  buf[i]=malloc(strlen(user_buf[i])+1);
  i++;
}
```

- Solaris Xsun:

```
buf=malloc(1024);
strcpy(buf,user_supplied);
```

2.2.4. Ataques avanzados

Si bien la idea de "avanzado" es relativa con heap overflows, ya que no es una vulnerabilidad simple de explotar, se enuncian en esta sección vectores de ataque aun más dependientes de las bibliotecas utilizadas y especialmente del sistema operativo subyacente.

En general, se tratan de realizar tres tipos de ataques:

- Sobreescribir un puntero a función.
- Sobreescribir código que se encuentra en un segmento con permisos de escritura.
- Sobreescribir alguna variable lógica para controlar el flujo del programa.

Entradas GOT

GOT significa Global Offset Table, es la tabla donde se almacenan los offsets para llamadas a funciones con link dinámico. Si se puede sobreescribir la dirección que es usada para realizar la llamada a una dada función se puede ejecutar una función arbitraria.

En general se utilizan funciones muy usadas como $printf()$, ya que en general es simple saber cuando esta función es ejecutada, es posible tener mayor control de la situación de explotado.

A continuación se muestra la GOT del programa de ejemplo utilizado anteriormente:

```
chiiph@delloise ~/src/tesis/tmp () $ objdump -R heap

heap:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   __gmon_start__
0804a004 R_386_JUMP_SLOT   __libc_start_main
0804a008 R_386_JUMP_SLOT   memcpy
0804a00c R_386_JUMP_SLOT   strcpy
0804a010 R_386_JUMP_SLOT   printf
0804a014 R_386_JUMP_SLOT   malloc
```

.DTORS

.DTORS son destructores que gcc utiliza cuando se termina una aplicación. Sobreescribir algun puntero a función en los destructores, nos dará control cuando la aplicación finalice.

Capítulo 3

Protecciones estáticas

3.1. Introducción a GCC

GCC es un compilador que en sus comienzos solo compilaba codigo en lenguaje C y GCC significaba GNU C Compiler. Con el tiempo, el compilador se fue reestructurando para pasar a soportar más lenguajes, y GCC pasó a significar GNU Compiler Collection.

3.1.1. Estructura

GCC en la actualidad posee una estructura modular que permite agregar todo tipo de features de forma completamente encapsulada, y que repercute en muchos entornos al mismo tiempo.

El compilador de GNU está formado principalmente por tres módulos:

- Front-end: encargado de la interfaz entre el código fuente del lenguaje en cuestión y el lenguaje intermedio (GENERIC o GIMPLE, dependiendo del front-end).
- Middle-end: encargado de las optimizaciones del código y el pasaje de GIMPLE al lenguaje RTL.
- Back-end: encargado de procesar RTL para producir código ejecutable en alguna dada arquitectura.

De esta forma, existen tres frentes a la hora de desarrollar para GCC. Por un lado, se puede agregar un lenguaje nuevo a la colección de compiladores, en cuyo caso se desarrollaría un front-end. Otra posibilidad es soportar una nueva arquitectura, y para ello se deberá implementar un back-end. Y por último si se desea generar una nueva forma de optimización o de checkeos en tiempo de compilación que no tengan que ver con particularidades del lenguaje a compilar, se implementa una nueva parte en el middle-end.

Con todo esto es a lo que se referia en el primer párrafo en cuanto a la repercución. Si se desarrolla un nuevo front-end para un lenguaje, el mismo va a estar soportado en todas las arquitecturas en las cuales existe back-end, sin que el desarrollador del front-end explícitamente lo desarrolle. Y lo mismo sucede con el back-end y módulos del middle-end.

3.1.2. Optimizaciones

Las protecciones de las aplicaciones en tiempo de compilación proveídas por GCC funcionan con mayor efectividad cuando se utiliza compilación optimizada, por ello analizaremos en más detalle como opera este módulo del compilador.

GCC utiliza tres lenguajes intermedios para representar el programa durante la compilación: GENERIC, GIMPLE y RTL. GENERIC es una representación independiente del lenguaje generado por cada front-end. Es usado como una interfaz entre el modulo de parsing y el de optimización.

GIMPLE y RTL es usado para optimizar el programa. GIMPLE es usado para optimizaciones independientes de la arquitectura final y del lenguaje utilizado para programar la aplicación, por ejemplo, para optimizar llamadas con inlines, eliminación de tail recursion, eliminación de redundancias, etc. A diferencia de GENERIC, la representación GIMPLE es más estricta: las expresiones no deben contener más de tres operandos exceptuando llamadas a funciones. No posee sentencias de control y expresiones con efectos secundarios son únicamente permitidas en el operando derecho de asignaciones.

3.2. FORTIFY_SOURCE

Los overflows en un buffer del stack pueden ser detectados en tiempo de compilación o en tiempo de ejecución. FORTIFY_SOURCE clasifica los tipos más simples de overflow, y basándose en esto realiza trés acciones distintas en tiempo de compilación:

- Si es sabido que no es posible que exista un overflow, se compila normalmente el código.
- Si es posible que ocurra un overflow en una dada secuencia de instrucciones, se agregan checkeos adicionales para prevenirlos.
- Si es asegurado que un overflow sucederá, se informa de esto y no se continúa con la compilación.

El segundo caso corresponde a un checkeo en tiempo de ejecución, y el tercero a uno en tiempo de compilación, por lo que esta técnica puede ser considerada "mixta" en la división que se llevó a cabo de los temas en este trabajo.

Para realizar los checkeos, se computan el número de bytes que restan hasta el final de un dado objeto (estructura de datos, etc) en cada llamada a funciones de manejo de memoria (`memcpy()`) o de manejo de strings (`strcpy()`).

De los overflows más simples, se pueden identificar los siguientes casos:

Dado el siguiente buffer:

```
char buf[5];
```

El primer caso abarca operaciones del estilo de:

```
memcpy(buf, foo, 5);
strcpy(buf, "abcd");
```

Las cuales son correctas en cuanto a buffer overflows, y se compilarán sin necesidad de checkeos adicionales.

El segundo caso abarca los siguientes casos:

```
memcpy(buf, foo, n);
strcpy(buf, bar);
```

En este caso, no es posible saber en tiempo de compilación si un overflow ocurrirá ya que, por ejemplo, la variable `n` puede ser proveída por el usuario. Lo que sí se puede efectuar es un checkeo en tiempo de ejecución. El compilador conoce los bytes que restan en el objeto pero no conoce el tamaño real de la copia que se va a realizar. Funciones alternativas de la libc, `__memcpy_chk()` y `__strcpy_chk()` son usadas en este caso para corroborar si un overflow ocurrió. En caso de que uno sea detectado, la función `__chk_fail()` es llamada, cuyo comportamiento es el de abortar la ejecución y mostrar un mensaje informando lo ocurrido.

El tercer caso:

```
memcpy(buf, foo, 6);
strcpy(buf, "abcde");
```

Es incorrecto siempre, ya que el primer `memcpy()` copia un byte de más, y el `strcpy()` copia cinco bytes, anulando el lugar para el caracter de finalización `\0`.

En este caso el compilador puede detectar el error en tiempo de compilación. En versiones anteriores a la 4.5.0 GCC simplemente daba un warning en estos casos. A partir de la versión 4.5.0, los warnings pasaron a ser errores de compilación.

Y el último caso:

```
memcpy(p, q, n);
strcpy(p, q);
```

El cual no es posible saber si es correcto, y no es posible checkearlo en tiempo de ejecución. Este tipo de overflows son pasados por alto por FORTIFY_SOURCE.

Todo este proceso se realiza para las siguientes funciones:

```
void *memcpy(void *dest, const void *src, size_t n);
void *mempcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
void *memset(void *s, int c, size_t n);

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format,
    va_list ap);
```

FORTIFY_SOURCE se puede aplicar en dos niveles. El nivel elegido determina la rigurosidad de los checkeos de tamaños del objeto en cuestión. Veamos un ejemplo:

```
struct S {
  struct T {
    char buf[5];
    int x;
  } t;
  char buf[20];
} var;
```

y el código posee la siguiente linea:

```
strcpy(&var.t.buf[1], "abcdefg");
```

Si se compila el ejecutable con `-D_FORTIFY_SOURCE=1`, el overflow pasará desapercibido, ya que solo se tomará en cuenta el tamaño total de la variable var, es decir, involucra también el arreglo de 20 carácteres que continúa a la variable de tipo `struct T`.

Por el contrario, si se compila con `-D_FORTIFY_SOURCE=2`, el objeto se analiza con mayor profundidad, y se detectará el overflow.

Por otro lado, el segundo nivel de FORTIFY_SOURCE tiene en cuenta las funciones de formato de strings, como `printf()`, en los casos en los que se usa el parámetro `%n`, el cual es muy usado en ataques de format string. Por ello, solo se permite el uso de dicho parámetro cuando el mismo está contenido en una sección estática de memoria. En los ataques de format string, la cadena de formato estará contenida en alguna sección escribible por el atacante.

3.3. Stack Protector

El Stack Protector de GCC es una herramienta que modifica el código generado de forma tal que un buffer overflow en el stack pueda ser detectado y la aplicación aborte para no permitir que la vulnerabilidad sea explotada.

Una variable guardia o canario es introducida en el código. Esta técnica fue originalmente presentada en el proyecto StackGuard, y estaba diseñada para posicionar la variable guardia inmediatamente despues de la dirección de retorno. La diferencia con el método utilizado en el Stack Protector es la locación de la variable y la protección para punteros a funciones. La variable es situada luego del puntero al frame anterior, y antes de un arreglo.

Para ilustrar el funcionamiento del Stack Protector se mostrará un ejemplo en código C. La protección es llevada a cabo en el middle-end de GCC, por lo que se trabaja sobre el lenguaje GIMPLE.

Dado el código de una función, un preprocesado insertará código en las locaciones correspondientes a la declaración de las variables locales, el punto de entrada donde se definirá el valor del canario, y el punto de salida donde

se corroborará que el valor del canario sea el original, si no lo es, el programa abortará.

```
...
// variables locales
volatile int guard;

...
// punto de entrada
guard = guard_value;

...
// punto de salida
if(guard != guard_value) {
  printf("Stack overflow detected\n");
  abort();
}
```

La función de ejemplo:

```
void bar( void (*func1)() ) {
  void (*func2)();
  char buf[128];
  ...
  strcpy(buf, getenv("HOME"));
  (*func1)();
  (*func2)();
}
```

En esta función, la declaración del canario se realizaria justo antes de la declaración del arreglo de carácteres, y la corroboración del canario será realizada justo despues de la llamada a $(*func2)()$. Esta forma de analizar el funcionamiento tiene algunos problemas, particularmente el hecho de que una función típica escrita en lenguaje C posee varias sentencias de retorno a lo largo de toda la función, por lo que habría que introducir el código del punto de salida justo antes de cada *return*. Como ya se explicó, la implementación

real de la protección fue programada para operar sobre el lenguaje intermedio que utiliza GCC, en el cual existe únicamente un punto de salida de la misma, y el orden final de las variables ya ha sido establecido.

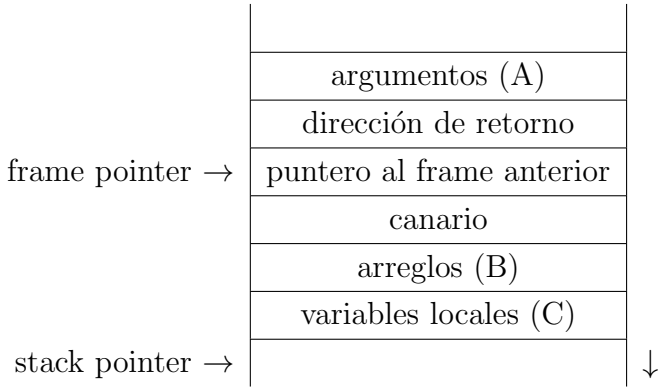
El requerimieto principal de la variable guardia es que posea un valor que el atacante no sepa. Si lo supiera, el atacante puede llenar el stack entre el buffer y el puntero al frame anterior con el valor del canario. Con lo cual saltearía el test en el punto de salida y podria potencialmente tomar control de la aplicación.

Stack Protector elije el valor del canario de forma aleatoria. Este número es calculado en el momento de inicialización de la aplicación, el cual no puede ser divisado por un usuario no privilegiado. Para que el número sea lo más impredecible posible, se utiliza el generador de números aleatorios proveído por Linux, cuya interfaz se presenta a través del dispositivo */dev/urandom* o */dev/random*. Este utiliza la actividad del entorno actual del sistema operativo para generar el número.

Otra posibilidad es definir el canario con un byte nulo (0x00), ya que todos los overflows en el stack son a través de cadenas de carácteres y estas poseen como carácter terminal el 0x00. Aun cuando el valor del canario sea conocido, cuando el atacante trate de escribir en el buffer el valor del canario para que el overflow no sea detectado, la cadena será terminada justo al escribir el byte nulo y no se podrá proceder con el posterior explotado de la aplicación.

En un principio, los ataques de overflow en el stack se realizaban únicamente para sobreescribir la dirección de retorno. Esta parte fue el objetivo del proyecto StackGuard, que luego se demostro obsoleto ya que no solo la dirección de retorno era un dato importante dentro de la función, sino que también lo eran otros datos internos, como punteros a funciones. Stack Protector, para solucionar este tipo de casos definió el concepto de “modelo de función segura”. No es un modelo sin posibilidad de errores, pero protege muchos casos que otros métodos no abarcan.

La idea de este modelo es limitar el uso del stack de manera que la locación A no contenga ningun arreglo ni variables de tipo puntero, la locación B tiene arreglos o estructuras que contienen arreglos, y la locación C no tiene arreglos.



De esta forma, la locación B es el único punto vulnerable donde un ataque puede comenzar a modificar el stack.

Este modelo hace que las locaciones fuera del frame de una función no puedan ser dañadas cuando la función retorna, ataques a variables de tipo puntero fuera del frame de la función no serán exitosos, y por último, un ataque a las variables de tipo puntero dentro del frame de la función tampoco será exitoso. Todo gracias a que la distribución de los datos harán que la variable canario detecte el ataque cuando se llegue al punto de salida.

3.3.1. Problemas

Si bien Stack protector previene la ejecución de código arbitrario, el overflow ocurre de todas formas, y existe un bug de "information leak". Veamoslo con un ejemplo, tomemos la siguiente aplicación:

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

char *cadena = "Esto es un secreto";

int main(int argc, char **argv) {
  char buf[32];
  strcpy(buf, argv[1]);

  return 0;
}
```

Lo compilamos con `-fstack-protector` para habilitar esta feature. Y utilizando la herramienta `objdump` encontramos la ubicación de `cadena` en `0x08048568`.

Primero veamos que sucede cuando sobrecargamos el buffer y se sobreescribe el canario:

```
chiiph@adell ~/src/tesis/tmp $ ./ssp $(python -c "print 'A'*200"
    )
*** stack smashing detected ***: ./ssp terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x50)[0xb767f6c0]
/lib/libc.so.6(+0xe566a)[0xb767f66a]
./ssp[0x8048495]
[0x41414141]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:06 649532     /home/chiiph/
    src/tesis/tmp/ssp
08049000-0804a000 r--p 00000000 08:06 649532     /home/chiiph/
    src/tesis/tmp/ssp
0804a000-0804b000 rw-p 00001000 08:06 649532     /home/chiiph/
    src/tesis/tmp/ssp
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b757e000-b7597000 r-xp 00000000 08:06 370407     /usr/lib/gcc/
    i686-pc-linux-gnu/4.5.2/libgcc_s.so.1
b7597000-b7598000 r--p 00018000 08:06 370407     /usr/lib/gcc/
    i686-pc-linux-gnu/4.5.2/libgcc_s.so.1
b7598000-b7599000 rw-p 00019000 08:06 370407     /usr/lib/gcc/
    i686-pc-linux-gnu/4.5.2/libgcc_s.so.1
b7599000-b759a000 rw-p 00000000 00:00 0
b759a000-b76f2000 r-xp 00000000 08:06 447557     /lib/libc
    -2.12.2.so
b76f2000-b76f4000 r--p 00158000 08:06 447557     /lib/libc
    -2.12.2.so
b76f4000-b76f5000 rw-p 0015a000 08:06 447557     /lib/libc
    -2.12.2.so
b76f5000-b76f8000 rw-p 00000000 00:00 0
b7708000-b7709000 rw-p 00000000 00:00 0
b7709000-b770a000 r-xp 00000000 00:00 0          [vdso]
```

```
b770a000-b7726000 r-xp 00000000 08:06 448778     /lib/ld-2.12.2.
    so
b7726000-b7727000 r--p 0001b000 08:06 448778     /lib/ld-2.12.2.
    so
b7727000-b7728000 rw-p 0001c000 08:06 448778     /lib/ld-2.12.2.
    so
bfbf3000-bfc14000 rw-p 00000000 00:00 0          [stack]
Aborted
```

La aplicación aborta y nos muestra un backtrace y un mapa de la memoria. Ahora veamos que sucede si cambiamos el contenido del buffer a la dirección de la cadena, y lo sobreescribimos un poco más allá:

```
chiiph@adell ~/src/tesis/tmp $ ./ssp $(python -c "print '\x68\
    x85\x04\x08'*55")
*** stack smashing detected ***: Esto es un secreto terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x50)[0xb77bb6c0]
/lib/libc.so.6(+0xe566a)[0xb77bb66a]
Esto es un secreto[0x8048495]
Esto es un secreto[0x8048568]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:06 649532     /home/chiiph/
    src/tesis/tmp/ssp
08049000-0804a000 r--p 00000000 08:06 649532     /home/chiiph/
    src/tesis/tmp/ssp
0804a000-0804b000 rw-p 00001000 08:06 649532     /home/chiiph/
    src/tesis/tmp/ssp
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b76ba000-b76d3000 r-xp 00000000 08:06 370407     /usr/lib/gcc/
    i686-pc-linux-gnu/4.5.2/libgcc_s.so.1
b76d3000-b76d4000 r--p 00018000 08:06 370407     /usr/lib/gcc/
    i686-pc-linux-gnu/4.5.2/libgcc_s.so.1
b76d4000-b76d5000 rw-p 00019000 08:06 370407     /usr/lib/gcc/
    i686-pc-linux-gnu/4.5.2/libgcc_s.so.1
b76d5000-b76d6000 rw-p 00000000 00:00 0
b76d6000-b782e000 r-xp 00000000 08:06 447557     /lib/libc
    -2.12.2.so
b782e000-b7830000 r--p 00158000 08:06 447557     /lib/libc
    -2.12.2.so
```

```
b7830000-b7831000 rw-p 0015a000 08:06 447557     /lib/libc
    -2.12.2.so
b7831000-b7834000 rw-p 00000000 00:00 0
b7844000-b7845000 rw-p 00000000 00:00 0
b7845000-b7846000 r-xp 00000000 00:00 0          [vdso]
b7846000-b7862000 r-xp 00000000 08:06 448778     /lib/ld-2.12.2.
    so
b7862000-b7863000 r--p 0001b000 08:06 448778     /lib/ld-2.12.2.
    so
b7863000-b7864000 rw-p 0001c000 08:06 448778     /lib/ld-2.12.2.
    so
bf88c000-bf8ad000 rw-p 00000000 00:00 0          [stack]
Aborted
```

Podemos ver como lo que antes era `./ssp` ahora se convirtió en el contenido del string. Esto sucedió porque para armar el mensaje de salida cuando la aplicación aborta utiliza `argv[0]` para saber el nombre de la aplicación en cuestión. Este dato se almacena en el stack, junto con el resto de los parámetros. Con nuestro overflow sobreescribimos el puntero con el de la cadena "secreta" dentro de la aplicación vulnerable.

3.4. Mudflap

Mudflap es una pasada del compilador GCC, que es aplicada luego de la pasada de parseo del código y antes de las optimizaciones y los backends, es decir, trabaja sobre los árboles SSA internos de GCC. Busca patrones de anidamiento en el árbol que correspondan a operaciones que involucran punteros potencialmente peligrosas. Estas estructuras son reemplazadas con expresiones que evaluan al mismo valor que la estructura anterior, pero que agrega partes que refieren a la *libmudflap*. El compilador, a su vez, agrega código asociado con variables locales.

Este código agregado tiene como propósito evaluar la validez de la utilización de un puntero derreferenciado. Esta evaluación implica simplemente determinar si la memoria referencia por dicho puntero corresponde a un valor válido en el espacio de memoria del proceso. En caso de no ser asi, se ha

detectado un potencial ataque a la aplicación.

Para evaluar si un acceso a memoria es válido, la biblioteca necesita mantener una "base de datos" de objetos válidos en memoria. Esta base de datos almacena muchos datos acerca de cada objeto de memoria, y puede que los guarde aun luego de haber liberado el mismo. Entre ellos se guardan el rango de direcciones del objeto, el nombre, el archivo de código donde es declarado y la linea, lugar donde se almacena (stack, heap, etc), estadísticas de acceso, y estampillas de tiempo. Con el objetivo de actualizar y buscar objetos en la base de datos, libmudflap exporta funciones que serán utilizadas por el código insertado en la pasada de compilación. Entre estas funciones, las más básicas son para evaluar accesos a memoria, para agregar o eliminar objetos de la base de datos. La base de datos es almacenada como un árbol binario, ordenado según la dirección de los objetos actualmente referenciados por la aplicación. Para mantener objetos "populares" lo más cerca de la raíz posible, periódicamente el árbol es reordenado.

Durante el comienzo de la ejecución de una aplicación, unos objetos son insertados a la base de datos como objetos especiales de "no acceso". Estos representan rangos de direcciones que son certeramente direcciones fuera del rango de la aplicación.

Para optimizar las búsquedas en la base de datos, libmudflap mantiene una cache de búsqueda. Esta cache es un arreglo compacto de mapeo directo, indexado por una función hash que recibe el valor del puntero al objeto buscado.

Expresiones potencialmente inseguras que involucren punteros son fáciles de divisar en código C/C++. Expresiones del estilo de `*p,p->f,p[a],`deben ser evaluadas. Como mudflap opera en medio de la compilación, no puede buscar estos patrones. En cambio, utiliza la representación interna similar a la de un árbol de sintaxis, codificada a través de la estructura tree de GCC.

Mudflap recorre el árbol que representa cada función en busca de ciertas estructuras relacionadas con uso de punteros o arreglos. Estos nodos del árbol son modificados, reemplazando la expresión simple del puntero con una más compleja que realiza las verificaciónes ya descriptas. A modo de ejemplo, la expresión p-¿f se transforma, a grandes razgos, en (check (p, ...); p;)-¿f.

Esta modificación, al realizarse "in-place", soporta derreferenciamiento anidado del estilo de `p->array[i]->f`, analizando primero la referencia `p->array[i]` y luego `array[i] ->f`.

Mudflap, como ya se explicó, opera sobre la etapa de compilación de una aplicación. Sin embargo, en general no es posible recompilar una aplicación completamente junto con todas las bibliotecas del sistema que necesita, lo que implica que muchas partes de la aplicación pueden ser vulnerables a pesar de la utilización de esta herramienta.

La mayoría de las aplicaciones C/C++ utilizan llamadas a funciones de bibliotecas externas. Tomemos por ejemplo la función *strcpy*. Generalmente estas bibliotecas no han sido compiladas con mudflap, y por ende no realizan ninguno de los checkeos ya discutidos. Una aplicación puede utilizar punteros inválidos como parámetros a strcpy y de esa forma saltear las protecciones que se cree que mudflap esta proporcionando. Para solventar esto, libmudflap provee wrappers a muchas funciones populares, a través de directivas del preprocesador para que puedan ser utilizadas de forma transparente, y a través de ellas, de forma segura.

Otra situación de conflicto involucra el uso de bibliotecas que devuelvan, a alguna función en la aplicación en cuestión, memoria reservada en un heap compartido. Estas regiones de memoria deben ser registradas por libmudflap para lograr evaluar con éxito las referencias de punteros a esas locaciones externas. Interceptar llamadas a *malloc* usando directivas del preprocesador no es posible en este caso, ya que la biblioteca ya ha sido compilada sin el soporte de mudflap. Estas llamadas deben ser interceptadas en tiempo de linkeo. Para ello, existen diversos mecanismos como por ejemplo *symbolwrapping* cuando se trata de enlazado estático, o *symbolinterposition* para enlazado dinámico.

En otros casos, puede suceder que una biblioteca no protegida por libmudflap le devuelve a la aplicación protegida un putero a una sección válida de memoria estática en la biblioteca. En este caso, ninguna de las técnicas de intercepción en tiempo de linkeo sirven, ya que estas secciones de memoria no son reservadas con ninguna función del sistema. Para establecer si dichos punteros son válidos, libmudflap usa un método heurístico. Esta

herística es utilizada cada vez que un acceso a memoria es considerado una violación, y utiliza información dependiente de la plataforma como límites de los segmentos del proceso, stack pointer, etc.

Capítulo 4

Conclusiones

El panorama actual se ve muy bien equipado para proteger a los usuarios de los problemas más comunes, pero ¿Realmente está todo protegido?. La respuesta, obviamente, es "No".

Lo que sucede es que todas las protecciones, en algún punto, aseguran propiamente la puerta principal pero siempre dejan una ventana no tan asegurada. De alguna de estas situaciones ya se habló, y existen otras que superan el alcanse de este trabajo, pero lo importante es que existen.

Tomemos un ejemplo de esta situación, analizando primero la condición actual de la mayoría de las computadoras con GNU/Linux: el mayor porcentaje no cuentan con la protección más poderosa de todas: PaX. Esta solo se encuentra presente en sistemas que han pasado por un proceso de "hardening", como pueden ser servidores públicos, o computadoras con datos sensibles. Esta situación nos deja únicamente con NX, ASLR básico, protecciones de la GLibC, y el Stack Protector de GCC. Dependiendo de la distribución utilizada, Stack Protector puede no estar habilitado.

Ahora al ejemplo: LibTiff, vulnerabilidad CVE-2010-2067. Esta vulnerabilidad se encuentra en el parseo de entradas de directorios propios del formato Tiff (TIFFDirEntry), y se trata de un llamado a la operación memcpy que utiliza un parámetro especificado en el archivo de imagen para saber cuantos bytes copiar de una dada locación al buffer "vulnerable". En una imagen bien formada, esta operación concluiría con éxito y sin problemas. Si

la imagen es propiamente alterada, se produce un overflow del buffer.

LibTiff está enlazada a una gran cantidad de aplicaciónes, cualquiera que utilice el formato de imágenes Tiff en GNU/Linux utiliza esta biblioteca. Esto muestra un grave problema: una vulnerabilidad en una biblioteca muy utilizada desemboca en una vulnerabilidad que abarca un gran porcentaje del sistema.

Este buffer overflow del stack, tiene una característica muy interesante: ocurre en un array con solo dos elementos, por lo que el Stack Protector configurado con los valores por defecto no pondrá un canario para este buffer, es decir, acabamos de saltear una protección.

Para explotar una vulnerabilidad es necesario lograr ejecutar código inyectado, para ello hay que cargar el shellcode en algún buffer al cual se pueda direccionar la ejecución. Aca nos encontramos con dos problemas: no se puede ejecutar en un buffer (NX), y no podemos saber la posición del buffer (ASLR). Pero estos en realidad no son problemas si utilizamos return oriented programming y un buffer que sea estático.

La idea es la siguiente: con ASLR básico, solo los mapeos anónimos son randomizados, es decir, los mapeos de archivos (no anónimos) siguen siendo mapeados a direcciones estáticas. Como LibTiff sirve para manipular imágenes, podemos utilizar la sección de datos de la misma para cargar nuestro payload en una sección fija de la memoria. Haciendo uso de return oriented programming, podremos convertir el mapeo del archivo a uno ejecutable para luego dirigir el registro EIP a la locación conocida de nuestro payload y asi tomar total control de la aplicación. Y de esta forma, salteamos todas las protecciones del sistema.

Cabe destacar que cuando se toma control de la ejecución, todo lo que se haga se ejecutará con los permisos de la aplicación explotada. Por lo que si explotamos un navegador de internet, puede ser bueno para cierto tipo de ataques, pero no estaríamos aprovechando al máximo las posibilidades que nos da esta vulnerabilidad. Para esto existe una aplicación llamada “tiffinfo”, que tiene como owner al usuario “root” pero es ejecutable por “world”, lo que significa que es el escenario perfecto.

De todo esto, podemos concluir que a pesar de todos los esfuerzos de

muchas personas por proteger las computadoras a pesar del factor humano, es imposible saltearlo ya que el mismo software esta programado por seres humanos, y como tales, cometemos errores. Asi, un buffer overflow del stack, que uno podría creer obsoleto con todos los métodos de protección actuales, se convierte en un atacante con privilegios de administrador.

Las protecciones del sistema operativo siguen siendo un porcentaje de la lucha contra este tipo de problemas, pero no comprenden la totalidad de la solución.

Bibliografía

[1] Shon Harris et al., *Gray Hat Hacking: The Ethical Hacker's handbook*. McGraw Hill, 2da Edición, 2008.

[2] Jon Erickson, *Hacking: The art of exploitation*. 1era Edición, 2003.

[3] Yu Ding et al., *Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks*. Annual Computer Security Applications Conference (ACSAC), 2010.

[4] Frank Ch. Eigler, *Mudflap: Pointer Use Checking for C/C++*. GCC Developers' Summit, 2003.

[5] Jason Merrill, *GENERIC and GIMPLE: A New Tree Representation for Entire Functions*. GCC Developers' Summit, 2003.

[6] Diego Novillo, *Tree SSA: A New Optimization Infrastructure for GCC*. GCC Developers' Summit, 2003.

[7] Perry Wagle, *StackGuard: Simple Stack Smash Protection for GCC*. GCC Developers' Summit, 2003.

[8] Piotr Bania, *JIT spraying and mitigations*. Kryptos Logic Research, 2010.

[9] Hovav Shacham, *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*. Proceedings of ACM CCS, 2010.

[10] Piotr Bania, *Security Mitigations for Return-Oriented Programming Attacks*. Kryptos Logic Research, 2010.

[11] Chris Anley et al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Willey Publishing Inc., 2da Edición, 2010.

[12] James C. Foster et al., *Writing Security Tools and Exploits*. Syngress, 1era Edición, 2006.

[13] Hiroaki Etoh et al., *Protecting from stack-smashing attacks*. IBM Research, 2000.

[14] Diego Bovet et al., *Understanding the Linux Kernel*. O'Reilly, 3era Edición, 2005.

[15] *OpenWall Project*. http://www.openwall.com/.

[16] *Doug Lea's malloc implementation*. http://g.oswego.edu/dl/html/malloc.html.

[17] *Pax Documentation: Design*. http://pax.grsecurity.net/docs/pax.txt.

[18] *Pax Documentation: ASLR*. http://pax.grsecurity.net/docs/aslr.txt.