

**LABORATORIO 2 ORGANIZACIÓN DE COMPUTADORES  
DECOMPILACIÓN DE PROGRAMAS**

**DANY RUBIANO JIMENEZ**

Profesores: Felipe Garay  
Erika Rosas  
Nicolás Hidalgo  
Ayudante: Ian Mejias



# TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS.....	iv
ÍNDICE DE CUADROS .....	v
<b>CAPÍTULO 1. INTRODUCCIÓN.....</b>	<b>7</b>
1.1    MOTIVACIÓN Y ANTECEDENTES . . . . .	7
1.2    OBJETIVOS . . . . .	7
1.3    ORGANIZACIÓN DEL DOCUMENTO . . . . .	7
<b>CAPÍTULO 2. MARCO TEÓRICO.....</b>	<b>9</b>
2.1    LENGUAJE EMSAMBLADOR . . . . .	9
2.2    MÍPS . . . . .	9
2.3    OPCODE. . . . .	10
2.4    DECOMPILACIÓN . . . . .	10
<b>CAPÍTULO 3. DESARROLLO.....</b>	<b>13</b>
3.1    PROGRAMA 1 . . . . .	13
3.2    PROGRAMA 2 . . . . .	21
<b>CAPÍTULO 4. CONCLUSIÓN.....</b>	<b>13</b>
<b>CAPÍTULO 4. BIBLIOGRAFÍA.....</b>	<b>13</b>

# ÍNDICE DE FIGURAS

# ÍNDICE DE CUADROS



# **CAPÍTULO 1. INTRODUCCIÓN**

## **1.1 MOTIVACIÓN Y ANTECEDENTES**

Dentro del campo de software existen una variedad de vulnerabilidades o fallos que permiten violar la seguridad de un software determinado. En este caso se abocará en explotar la vulnerabilidad de la modificación de un programa, de la forma que se pueda manipular el comportamiento de este mismo, así, poder entender el modo de operación de susodicho método de vulneración. Para esto caso, se tienen dos programas hechos en lenguaje C y compilado para MIPS a los cuales se les debe modificar su comportamiento. Las herramientas que se utilizan son, Qemu, que es un emulador de procesadores basado en la traducción dinámica de binarios, y el paquete "gcc-5-mipsel-linux-gnu", que permite trabajar con programas en MIPS.

## **1.2 OBJETIVOS**

El objetivo del presente documento es exponer al lector el desarrollo y resultado de la decompilación de los programas propuestos para la modificación de su comportamiento, aplicando el conocimiento de MIPS adquirido a lo largo de la asignatura y las herramientas anteriormente mencionadas.

## **1.3 ORGANIZACIÓN DEL DOCUMENTO**

Para cumplir con los objetivos respectivos el presente documento distribuye su información de la manera siguiente: Primero se encuentra un marco teórico en el cual se expone al lector los conceptos teóricos que son utilizados a lo largo del informe con el fin de facilitar la comprensión del mismo.

Posteriormente se presentan los resultados obtenidos en este trabajo, donde se muestra como se desarrolla la decompilación y la modificación de los programas pedidos. Luego se presentan las conclusiones con las reflexiones que conllevo el trabajo realizado lo cual también incluye menciones a los métodos utilizados.





## CAPÍTULO 2. MARCO TEÓRICO

### 2.0.1 Lenguaje Ensamblador

El lenguaje ensamblador, es un lenguaje de programación de bajo nivel para los computadores, microprocesadores, microcontroladores y otros circuitos integrados programables. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura dada de CPU y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador. Esta representación es usualmente definida por el fabricante de hardware, y está basada en los mnemónicos que simbolizan los pasos de procesamiento (las instrucciones), los registros del procesador, las posiciones de memoria y otras características del lenguaje. Un lenguaje ensamblador es por lo tanto específico de cierta arquitectura de computador física (o virtual). Esto está en contraste con la mayoría de los lenguajes de programación de alto nivel, que idealmente son portátiles.

Un programa utilitario llamado ensamblador es usado para traducir sentencias del lenguaje ensamblador al código de máquina del computador objetivo. El ensamblador realiza una traducción más o menos isomorfa (un mapeo de uno a uno) desde las sentencias mnemónicas a las instrucciones y datos de máquina. Esto está en contraste con los lenguajes de alto nivel, en los cuales una sola declaración generalmente da lugar a muchas instrucciones de máquina.

Muchos sofisticados ensambladores ofrecen mecanismos adicionales para facilitar el desarrollo del programa, controlar el proceso de ensamblaje, y la ayuda de depuración. Particularmente, la mayoría de los ensambladores modernos incluyen una facilidad de macro (descrita más abajo), y se llaman macro ensambladores.

Fue usado principalmente en los inicios del desarrollo de software, cuando aún no se contaba con potentes lenguajes de alto nivel y los recursos eran limitados. Actualmente se utiliza con frecuencia en ambientes académicos y de investigación, especialmente cuando se requiere la manipulación directa de hardware, alto rendimiento, o un uso de recursos controlado y reducido. También es utilizado en el desarrollo de controladores de dispositivo (en inglés, device drivers) y en el desarrollo de sistemas operativos, debido a la necesidad del acceso directo a las instrucciones de la máquina. Muchos dispositivos programables (como los microcontroladores) aún cuentan con el ensamblador como la única manera de ser manipulados. (A1).

## 2.1 MIPS

MIPS (originalmente un acrónimo de microprocesador sin Etapas Interlocked Pipeline ) es un reducido equipo conjunto de instrucciones (RISC) conjunto de instrucciones de la arquitectura (ISA) desarrollada por MIPS Technologies (anteriormente MIPS Computer Systems, Inc.). Las primeras arquitecturas MIPS fueron de 32 bits, con versiones de 64 bits añadidos más tarde. Existen múltiples revisiones del MIPS, incluyendo MIPS I, II MIPS, MIPS III, IV de MIPS, MIPS V, MIPS32, y MIPS64. Las revisiones actuales son MIPS32 (para las implementaciones de 32 bits) y MIPS64 (para las implementaciones de 64 bits). MIPS32 y MIPS64 definen un control de registro establecido, así como el conjunto de instrucciones.

MIPS es una arquitectura basada en registro, es decir, la CPU utiliza registros para realizar operaciones

en. Hay otros tipos de procesadores por ahí, así, como los procesadores basados en la pila y los procesadores basados en acumuladores. (AA3).

## 2.2 OPCODE

Un opcode (operation code) o código de operación, es la porción de una instrucción de lenguaje de máquina que especifica la operación a ser realizada. Su especificación y formato serán determinados por la arquitectura del conjunto de instrucciones (ISA) del componente de hardware de computador - normalmente un CPU, pero posiblemente una unidad más especializada. Una instrucción completa de lenguaje de máquina contiene un opcode y, opcionalmente, la especificación de uno o más operandos - sobre los que la operación debe actuar. Algunas operaciones tienen operandos implícitos, o de hecho ninguno. Algunas ISAs tiene instrucciones con campos definidos para los opcodes y operandos, mientras que otras (ej. la arquitectura Intel x86) tienen una estructura más complicada y de propósito específico. Los operandos sobre los cuales los opcodes aplican pueden, dependiendo de la arquitectura del CPU, consistir de: registros, valores en memoria, valores almacenados en la pila, puertos de I/O, bus, etc. Las operaciones que un opcode puede especificar pueden incluir aritmética, copia de datos, operaciones lógicas, y control del programa.

Los opcodes también pueden ser encontrados en los bytecodes interpretados por un interpretador de código de byte (o la máquina virtual, en un sentido de ese término). En éstos, una arquitectura de conjunto de instrucciones es creada para ser interpretada por software en vez de un dispositivo de hardware. A menudo, los interpretadores de código de byte trabajan con tipos de datos y operaciones de más alto nivel, que el de un conjunto de instrucciones por hardware, pero son construidas a lo largo de líneas similares. (AA1).

## 2.3 DECOMPILACIÓN

El término "decompilar" se aplica comúnmente a programas cuya función es la de traducir un código ejecutable a código fuente, donde:

- el programa ejecutable está en código máquina, que es un lenguaje de bajo nivel (de hecho, el nivel de abstracción más bajo que existe), de la salida de un compilador)
- el código fuente está en un lenguaje de alto nivel, más inteligible y fácil de modificar por las personas, con la ventaja de poderse portar a alguna otra máquina (aunque tal vez necesite algunos cambios). Tras este proceso, el fuente puede volver a ser compilado para producir nuevamente un ejecutable que se comportará como el original.

En comparación, un desensamblador traduce un ejecutable exclusivamente a lenguaje ensamblador que, como diferencia, aún depende del soporte hardware y sigue teniendo un nivel de abstracción mínimo (sólo superior al código máquina), pero resulta legible por humanos (y este código puede volver a ser ensamblado en un programa ejecutable).

Decompilar es el acto de utilizar un decompilador, aunque si es usado como nombre, puede referirse a la salida de un decompilador. Puede ser usado para recuperar código fuente, y es muy útil en casos de

seguridad del ordenador, interoperatividad y corrección de errores.<sup>1</sup> El éxito de la decompilación depende de la cantidad de información presente en el código que está siendo decompilado y en la sofisticación del análisis realizado sobre él. Los formatos de bytecode utilizados por muchas máquinas virtuales en ocasiones incluyen metadatos en el alto nivel que hacen que la decompilación sea más flexible. Los lenguajes máquina normalmente tienen mucho menos metadatos, y son por lo tanto mucho más difíciles de compilar.

Algunos compiladores y herramientas de post-compilación producen código ofuscado (esto quiere decir que intentan producir una salida que es muy difícil de decompilar). Esto hace que sea más difícil revertir el código del ejecutable. (AA2).



## CAPÍTULO 3. DESARROLLO

### 3.1 PROGRAMA 1

En el caso de este programa, existen tres operaciones matemáticas a las cuales se pueden acceder mediante el ingreso de la letra de las distintas opciones que se presentan, pero el programador se equivocó al momento de realizar la selección correcta de cada una de ellas. Entonces, se pide modificar el programa para que utilice las letras correctas.

Para realizar esta tarea, primero se debe decompilar el programa, para ello se ejecuta el siguiente comando, guardando el resultado en un archivo de texto, para hacer más fácil su análisis:

```
mipsel-linux-gnu-objdump -d p1 >> p1.txt
```

Luego se decompila la sección .data del ejecutable p1, mediante el comando:

```
mipsel-linux-gnu-objdump -s -j .rodata -d p1 >> p1_rodata.txt
```

Una vez obtenidos estos datos, se procede a analizar el código obtenido en la decompilación. Dado las características que se presentan para este programa y evaluando el programa, se puede encontrar en el main los llamados que se hace para que el usuario ingrese las opciones y los parámetros requeridos. Es en la función principal entonces, que se pueden encontrar aquellas instrucciones que imprimen los mensajes, y captan lo ingresado por el usuario para el desarrollo del programa. Es de esperarse, que como existen tres operaciones matemáticas, por lo tanto, existen tres opciones para acceder a estas.

```

00400ff8 <main>:
400ff8: 27bdfdd0      addiu    sp,sp,-48
400ffc: afbf002c      sw      ra,44(sp)
401000: afbe0028      sw      s8,40(sp)
401004: 03a0f025      move    s8,sp
401008: 3c1c004b      lui     gp,0x4b
40100c: 279cbf50      addiu    gp,gp,-16560
401010: afbc0010      sw      gp,16(sp)
401014: afc40030      sw      a0,48(s8)
401018: afc50034      sw      a1,52(s8)
40101c: 3c020047      lui     v0,0x47
401020: 24447460      addiu    a0,v0,29792
401024: 8f828168      lw      v0,-32408(gp)
401028: 0040c825      move    t9,v0
40102c: 04111ed0      bal     408b70 <_IO_printf>
401030: 00200825      move    at,at
401034: 8fdc0010      lw      gp,16(s8)
401038: 27c20020      addiu    v0,s8,32
40103c: 00402825      move    a1,v0
401040: 3c020047      lui     v0,0x47
401044: 24447484      addiu    a0,v0,29828
401048: 8f82816c      lw      v0,-32404(gp)
40104c: 0040c825      move    t9,v0
401050: 04111eeb      bal     408c00 <__isoc99_scanf>
401054: 00200825      move    at,at
401058: 8fdc0010      lw      gp,16(s8)
40105c: 3c020047      lui     v0,0x47
401060: 24447488      addiu    a0,v0,29832
401064: 8f828168      lw      v0,-32408(gp)
401068: 0040c825      move    t9,v0
40106c: 04111ec0      bal     408b70 <_IO_printf>
401070: 00200825      move    at,at
401074: 8fdc0010      lw      gp,16(s8)
401078: 27c20024      addiu    v0,s8,36
40107c: 00402825      move    a1,v0
401080: 3c020047      lui     v0,0x47
401084: 244474a8      addiu    a0,v0,29864
401088: 8f82816c      lw      v0,-32404(gp)
40108c: 0040c825      move    t9,v0
401090: 04111edb      bal     408c00 <__isoc99_scanf>
401094: 00200825      move    at,at
401098: 8fdc0010      lw      gp,16(s8)
40109c: afc00018      sw      zero,24(s8)
4010a0: afc0001c      sw      zero,28(s8)

```

Figura 3-1: Análisis 1 Main Programa 1

Se puede detallar en la imagen anterior, que la función *main* comienza en la dirección de memoria 00400ff8. Así mismo, se observa que en la dirección de memoria 40102c, se hace un llamado a la función *< IO\_printf >* mediante la instrucción *bal*, la cual es equivalente a un *jal*. Al buscar en las instrucciones anteriores a esta, se puede encontrar presente un *lui*, el cual carga en *v0* una constante de 0X470000 y un *addiu*, que añade un offset de 29792 a *a0* sumado con *v0*. Al calcular esta suma a través de Python, el resultado es de 0x477488, si se busca esa dirección de memoria en la sección de rodata, se puede encontrar el string que pide la opción para continuar con la ejecución del programa, esto da un indicio importante para encontrar el lugar en que se compara lo ingresado con las opciones, para hacer los respectivos saltos.

401090:	04111edb	bal	408c00 <__isoc99_scanf>
401094:	00200825	move	at,at
401098:	8fdc0010	lw	gp,16(s8)
40109c:	afc00018	sw	zero,24(s8)
4010a0:	afc0001c	sw	zero,28(s8)
4010a4:	83c20020	lb	v0,32(s8)
4010a8:	24030063	li	v1,99
4010ac:	10430009	beq	v0,v1,4010d4 <main+0xdc>
4010b0:	00200825	move	at,at
4010b4:	24030070	li	v1,112
4010b8:	1043000e	beq	v0,v1,4010f4 <main+0xfc>
4010bc:	00200825	move	at,at
4010c0:	24030061	li	v1,97
4010c4:	10430013	beq	v0,v1,401114 <main+0x11c>
4010c8:	00200825	move	at,at
4010cc:	10000019	b	401134 <main+0x13c>
4010d0:	00200825	move	at,at
4010d4:	8fc20024	lw	v0,36(s8)
4010d8:	00402025	move	a0,v0
4010dc:	0c1003c4	jal	400f10 <f>
4010e0:	00200825	move	at,at
4010e4:	8fdc0010	lw	gp,16(s8)
4010e8:	f7c00018	sdcl	\$f0,24(s8)
4010ec:	10000018	b	401150 <main+0x158>
4010f0:	00200825	move	at,at
4010f4:	8fc20024	lw	v0,36(s8)
4010f8:	00402025	move	a0,v0
4010fc:	0c1003d9	jal	400f64 <g>
401100:	00200825	move	at,at
401104:	8fdc0010	lw	gp,16(s8)
401108:	f7c00018	sdcl	\$f0,24(s8)
40110c:	10000010	b	401150 <main+0x158>
401110:	00200825	move	at,at
401114:	8fc20024	lw	v0,36(s8)
401118:	00402025	move	a0,v0
40111c:	0c1003e9	jal	400fa4 <h>
401120:	00200825	move	at,at
401124:	8fdc0010	lw	gp,16(s8)
401128:	f7c00018	sdcl	\$f0,24(s8)
40112c:	10000008	b	401150 <main+0x158>
401130:	00200825	move	at,at
401134:	3c020047	lui	v0,0x47
401138:	244474ac	addiu	a0,v0,29868
40113c:	8f828170	lw	v0,-32400(gp)
401140:	0040c825	move	t9,v0
401144:	04112112	bal	409590 <_IO_puts>

Figura 3-2: Análisis 2 Main Programa 1

Del análisis de la porción de código anterior, se puede ver que en la dirección de memoria 4010a8 esta presente la instrucción *li*, que carga en *v1* un 99 en ASCII, valor que representa a la letra *c*. Luego se puede ver presente un *beq*, que compara *v0*, que es el valor ingresado con *v1*, si es igual salta a la dirección de memoria 4010d4, en la que hay próximo un *jal* que llama a la función de una operación determinada, de lo contrario sigue a la próxima instrucción. Así mismo, nuevamente se encuentra un *li* en la dirección 4010b4, que carga en *v1* un 112 en ASCII, valor que representa a la letra *p*. Luego al igual que antes, se puede ver presente un *beq*, que compara *v0*, que es el valor ingresado con *v1*, si es igual salta a un *jal*, que llama a la función de una operación determinada, de lo contrario sigue a la próxima instrucción. Y por último, encontramos la tercera opción, en la dirección 4010c0, en donde se carga con un *li* el valor 97 en *v1*, y se hace la comparación con el mismo proceso que lo anteriormente descrito.

Debido a que se pide arreglar la selección de las opciones, se debe entonces ver su comportamiento y lo que cada una de estas representa, por lo tanto, se ejecuta el programa en reiteradas ocasiones tal como se

muestra en las siguientes imágenes:

```

dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 78
Val: 78
Resultado: 474552.000000
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 1
Val: 1
Resultado: 1.000000
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 2
Val: 2
Resultado: 8.000000
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 3
Val: 3
Resultado: 27.000000
dany@Colombi-Pc ~/orga $

```

ASCII	Hex	Símbolo
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d

Figura 3-3: Resultados opción a

En esta opción, se puede inferir que para cada uno de los números ingresados, el resultado dado es el cubo de dicho número.(debería ser c)

$$a = \text{numero}^3$$

```

dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 1
Val: 1
Resultado: 3.141593
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 2
Val: 2
Resultado: 12.566371
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 3
Val: 3
Resultado: 28.274334
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 4
Val: 4
Resultado: 50.265482
dany@Colombi-Pc ~/orga $

```

Figura 3-4: Resultados opción c

El resultado de esta opción tiene cierta complicación, pero que se logra dilucidar ingresando como parámetro el número 1, ya que da como resultado el numero  $\pi$ . Luego con otros números y con la ayuda de la calculadora, se puede encontrar que el resultado está dado por:

$$c = \pi * \text{numero}^2$$



```

Terminal
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): p
Escriba el valor (número): 1
Val: 1
Resultado: 6.283185
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): p
Escriba el valor (número): 2
Val: 2
Resultado: 12.566371
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): p
Escriba el valor (número): 3
Val: 3
Resultado: 18.849556
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): p
Escriba el valor (número): 4
Val: 4
Resultado: 25.132741
dany@Colombi-Pc ~/orga $

```

Figura 3-5: Resultados opción *p*

Y por último, para esta opción se logra inferir que el resultado está dado por la operación:

$$p = 2 * \pi * \text{numero}$$

Si se observa las características de las operaciones encontradas, para la opción *a* se obtiene el cubo del número, para *c* podría ser que este represente el área de una circunferencia, y por último para la opción *p*, se puede encontrar el perímetro de una circunferencia. Por lo tanto, de lo anterior se puede deducir que si las letras representan la inicial del nombre de dichas operaciones matemáticas, se encuentra que hay un error debido a que *a* es el cubo y *c* es el área, entonces esto se debe intercambiar para que *a* sea el área y *c* el cubo; para el caso de la opción *p*, esta según este supuesto estaría de forma correcta.

Siguiendo el supuesto anterior, ya que no se indica cual debe ser la nueva representación que debe tener para arreglar este programa, se debe cambiar la forma en que se piden las opciones. Para ello, se debe buscar el valor hexadecimal de la instrucción que carga en *v1* el valor de ASCII.

Tabla 3.1: Valores hexadecimales de las instrucciones *li*

Letra	ASCII	Hexadecimal	Inst. Hex
a	97	61	24030061
c	99	63	24030063
p	112	70	24030070

Una vez encontrados los valores hexadecimales de las instrucciones, con la ayuda de *bless*, se procede a modificar las instrucciones necesarias, de tal modo que se haga el intercambio mencionado. Para ello, se modifica la dirección de memoria 4010b4 y la dirección 4010c0. El resultado se muestra en la siguiente imagen:

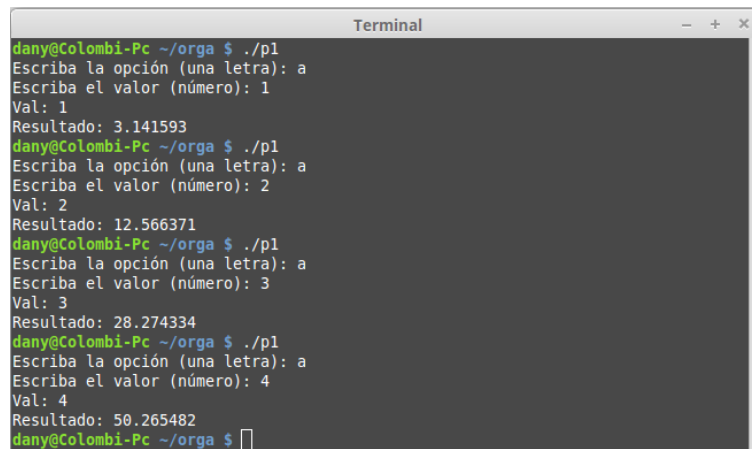
```

401090:    04111edb    bal     408c00 <__isoc99_scanf>
401094:    00200825    move   at,at
401098:    8fdc0010    lw     gp,16(s8)
40109c:    afc00018    sw     zero,24(s8)
4010a0:    afc0001c    sw     zero,28(s8)
4010a4:    83c20020    lb     v0,32(s8)
4010a8:    24030061    li     v1,97
4010ac:    10430009    beq    v0,v1,4010d4 <main+0xdc>
4010b0:    00200825    move   at,at
4010b4:    24030070    li     v1,112
4010b8:    1043000e    beq    v0,v1,4010f4 <main+0xfc>
4010bc:    00200825    move   at,at
4010c0:    24030063    li     v1,99
4010c4:    10430013    beq    v0,v1,401114 <main+0x11c>
4010c8:    00200825    move   at,at
4010cc:    10000019    b      401134 <main+0x13c>
4010d0:    00200825    move   at,at
4010d4:    8fc20024    lw     v0,36(s8)
4010d8:    00402025    move   a0,v0
4010dc:    0c1003c4    jal    400f10 <f>
4010e0:    00200825    move   at,at
4010e4:    8fdc0010    lw     gp,16(s8)
4010e8:    f7c00018    sdcl   $f0,24(s8)
4010ec:    10000018    b      401150 <main+0x158>
4010f0:    00200825    move   at,at
4010f4:    8fc20024    lw     v0,36(s8)
4010f8:    00402025    move   a0,v0
4010fc:    0c1003d9    jal    400f64 <g>
401100:    00200825    move   at,at
401104:    8fdc0010    lw     gp,16(s8)
401108:    f7c00018    sdcl   $f0,24(s8)
40110c:    10000010    b      401150 <main+0x158>
401110:    00200825    move   at,at
401114:    8fc20024    lw     v0,36(s8)
401118:    00402025    move   a0,v0
40111c:    0c1003e9    jal    400fa4 <h>
401120:    00200825    move   at,at
401124:    8fdc0010    lw     gp,16(s8)
401128:    f7c00018    sdcl   $f0,24(s8)
40112c:    10000008    b      401150 <main+0x158>

```

Figura 3-6: Manipulación Programa 1

De esta forma, si se ejecuta de nuevo el programa con diferentes pruebas para las distintas opciones, se puede observar que ahora se tiene el comportamiento esperado, tal como se muestra en las siguientes imágenes. No se incluye los resultado para la opción *p*, ya que esta porción no se vio alterada.

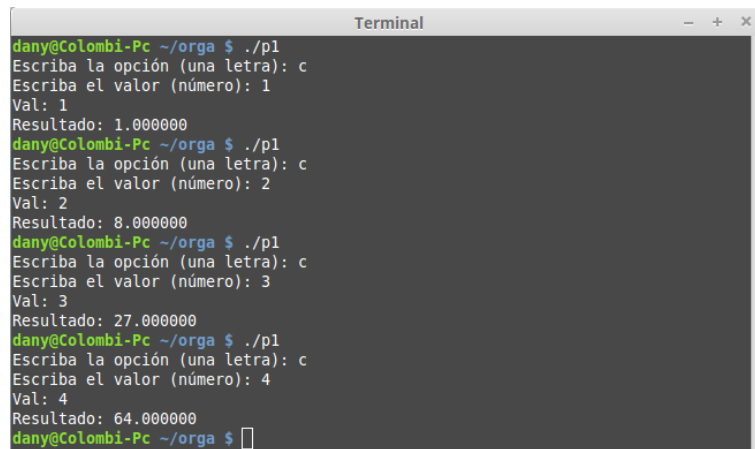


```

Terminal
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 1
Val: 1
Resultado: 3.141593
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 2
Val: 2
Resultado: 12.566371
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 3
Val: 3
Resultado: 28.274334
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): a
Escriba el valor (número): 4
Val: 4
Resultado: 50.265482
dany@Colombi-Pc ~/orga $

```

Figura 3-7: Resultados opción a arreglado



```
Terminal
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 1
Val: 1
Resultado: 1.000000
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 2
Val: 2
Resultado: 8.000000
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 3
Val: 3
Resultado: 27.000000
dany@Colombi-Pc ~/orga $ ./p1
Escriba la opción (una letra): c
Escriba el valor (número): 4
Val: 4
Resultado: 64.000000
dany@Colombi-Pc ~/orga $
```

*Figura 3-8: Resultados opción c arreglado*

## 3.2 PROGRAMA 2

Para el caso de este programa, aquí se solicita una clave y un número, si la clave es correcta entonces devuelve el cuadrado del número, en caso contrario indica un error. Se pide entonces, modificar el programa para que permita calcular el cuadrado del número sin importar la clave que se ingrese.

Para realizar esta tarea, primero se debe decompilar el programa, para ello se ejecuta el siguiente comando, guardando el resultado en un archivo de texto, para hacer más facil su análisis:

```
mipsel-linux-gnu-objdump -d p2 >> p2.txt
```

Luego se decompila la sección .data del ejecutable p2, mediante el comando:

```
mipsel-linux-gnu-objdump -s -j .rodata -d p1 >> p1_rodata.txt
```

Una vez obtenidos estos datos, se procede a analizar el código obtenido en la decompilación. Dado las características que se presentan para este programa y evaluando el programa, se puede encontrar en el main los llamados que se hace para que el usuario ingrese los parámetros requeridos.

```

00400f5c <main>:
400f5c: 27bdfc8      addiu    sp,sp,-56
400f60: afbf0034     sw      ra,52(sp)
400f64: afbe0030     sw      s8,48(sp)
400f68: 03a0f025     move    s8,sp
400f6c: 3c1c004b     lui     gp,0x4b
400f70: 279cbbd0     addiu   gp,gp,-17456
400f74: afbc0010     sw      gp,16(sp)
400f78: afc40038     sw      a0,56(s8)
400f7c: afc5003c     sw      a1,60(s8)
400f80: afc00018     sw      zero,24(s8)
400f84: afc00028     sw      zero,40(s8)
400f88: afc0001c     sw      zero,28(s8)
400f8c: 3c020047     lui     v0,0x47
400f90: 24447150     addiu   a0,v0,29008
400f94: 8f828168     lw      v0,-32408(gp)
400f98: 0040c825     move    t9,v0
400f9c: 04111ed0     bal     408ae0 <_IO_printf>
400fa0: 00200825     move    at,at
400fa4: 8fdc0010     lw      gp,16(s8)
400fa8: 27c20024     addiu   v0,s8,36
400fac: 00402825     move    a1,v0
400fb0: 3c020047     lui     v0,0x47
400fb4: 24447164     addiu   a0,v0,29028
400fb8: 8f82816c     lw      v0,-32404(gp)
400fbc: 0040c825     move    t9,v0
400fc0: 04111eeb     bal     408b70 <__isoc99_scanf>
400fc4: 00200825     move    at,at
400fc8: 8fdc0010     lw      gp,16(s8)
400fcc: 3c020047     lui     v0,0x47
400fd0: 24447168     addiu   a0,v0,29032
400fd4: 8f828168     lw      v0,-32408(gp)
400fd8: 0040c825     move    t9,v0
400fdc: 04111ec0     bal     408ae0 <_IO_printf>
400fe0: 00200825     move    at,at
400fe4: 8fdc0010     lw      gp,16(s8)
400fe8: 27c20028     addiu   v0,s8,40
400fec: 00402825     move    a1,v0
400ff0: 3c020047     lui     v0,0x47
400ff4: 24447164     addiu   a0,v0,29028
400ff8: 8f82816c     lw      v0,-32404(gp)
400ffc: 0040c825     move    t9,v0
401000: 04111edb     bal     408b70 <__isoc99_scanf>
401004: 00200825     move    at,at
401008: 8fdc0010     lw      gp,16(s8)
40100c: 8fc30024     lw      v1,36(s8)
401010: 24020001     li      v0,1
401014: 14620003     bne     v1,v0,401024 <main+0xc8>
401018: 00200825     move    at,at
40101c: 24020001     li      v0,1
401020: afc2001c     sw      v0,28(s8)
401024: 8fc30024     lw      v1,36(s8)

```

Figura 3-9: Análisis 1 Main Programa 2

De la imagen anterior se puede observar que la función *main* comienza en la dirección de memoria 00400f5c. Así mismo, se observa que en la dirección de memoria 400f9c, se hace un llamado a la función `< IO_printf >` mediante la instrucción *bal*. Al buscar en las instrucciones anteriores a esta, se puede encontrar presente un *lui*, el cual carga en *v0* una constante de 0X470000 y un *addiu*, que añade un offset de 29008 a *a0* sumado con *v0*. Al calcular esta suma a través de Python, el resultado es de 0x477150, si se busca esa dirección de memoria en la sección de rodata, se puede encontrar el string que pide la clave numérica para continuar con la ejecución del programa, esto da un indicio importante para encontrar el lugar en que se debe modificar este.

Se puede deducir que si el programa pide una clave, se necesita de una instrucción que compare lo

ingresado con un valor que este guardado en una variable, y esto se debería encontrar después de una instrucción que lea lo ingresado, entonces, se procede a buscar estas características en el código.

400fe8:	27c20028	addiu	v0,s8,40
400fec:	00402825	move	a1,v0
400ff0:	3c020047	lui	v0,0x47
400ff4:	24447164	addiu	a0,v0,29028
400ff8:	8f82816c	lw	v0,-32404(gp)
400ffc:	0040c825	move	t9,v0
401000:	04111edb	bal	408b70 <__isoc99_scanf>
401004:	00200825	move	at,at
401008:	8fdc0010	lw	gp,16(s8)
40100c:	8fc30024	lw	v1,36(s8)
401010:	24020001	li	v0,1
401014:	14620003	bne	v1,v0,401024 <main+0xc8>
401018:	00200825	move	at,at
40101c:	24020001	li	v0,1
401020:	afc2001c	sw	v0,28(s8)
401024:	8fc30024	lw	v1,36(s8)
401028:	24020003	li	v0,3
40102c:	14620003	bne	v1,v0,40103c <main+0xe0>
401030:	00200825	move	at,at
401034:	24020001	li	v0,1
401038:	afc2001c	sw	v0,28(s8)
40103c:	8fc30024	lw	v1,36(s8)
401040:	24020005	li	v0,5
401044:	14620003	bne	v1,v0,401054 <main+0xf8>
401048:	00200825	move	at,at
40104c:	24020001	li	v0,1
401050:	afc20018	sw	v0,24(s8)
401054:	8fc30024	lw	v1,36(s8)
401058:	24020008	li	v0,8
40105c:	14620003	bne	v1,v0,40106c <main+0x110>
401060:	00200825	move	at,at
401064:	24020001	li	v0,1
401068:	afc2001c	sw	v0,28(s8)
40106c:	8fc30024	lw	v1,36(s8)
401070:	2402000a	li	v0,10
401074:	14620003	bne	v1,v0,401084 <main+0x128>
401078:	00200825	move	at,at
40107c:	24020001	li	v0,1
401080:	afc2001c	sw	v0,28(s8)
401084:	8fc30024	lw	v1,36(s8)
401088:	2402000d	li	v0,13
40108c:	14620003	bne	v1,v0,40109c <main+0x140>
401090:	00200825	move	at,at
401094:	24020001	li	v0,1
401098:	afc2001c	sw	v0,28(s8)
40109c:	8fc20028	lw	v0,40(s8)
4010a0:	00402825	move	a1,v0
4010a4:	8fc40018	lw	a0,24(s8)
4010a8:	0c1003c4	jal	400f10 <ejecutar>
4010ac:	00200825	move	at,at
4010b0:	8fdc0010	lw	gp,16(s8)
4010b4:	afc20020	sw	v0,32(s8)
4010b8:	8fc20020	lw	v0,32(s8)

Figura 3-10: Análisis 2 Main Programa 2

Continuando con el análisis del código, se puede observar que hay presente un *bal* que llama a la función *scanf*, y que en unas instrucciones después hay presente un *bne*, el cual compara el valor ingresado con otro valor. Este ultimo puede ser la contraseña y se evidencia que está en el registro s8, en la posición 36.

*v1* es comparado con *v0* en el *bne*, si son iguales se sigue con la ejecución, en caso contrario, salta a 401024 donde se carga nuevamente la clave inserta en el registro y se realiza de nuevo una comparación. De forma recurrente se presenta estas comparaciones hasta la dirección de memoria 40108c, en donde si son iguales sigue hasta un *jal* que llama a *<ejecucion>*.

Por lo tanto, es necesario evitar las comparaciones para que en cualquiera de los casos que se ingrese una contraseña se siga con la ejecución del programa. Para esto se utiliza la herramienta *bleess*, en donde para cada caso en que se encuentre la instrucción 14620003 en su transformación (03006214), siempre y cuando siga la estructura del código y este en el debido lugar, se transforme en una instrucción *nop*, y así manipular la ejecución del programa.

Después de realizado este proceso, el resultado es el siguiente:

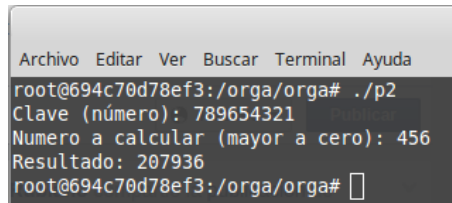
```

400fe0: 00200825  move at,at
400fe4: 8fdc0010  lw gp,16(s8)
400fe8: 27c20028  addiu v0,s8,40
400fec: 00402825  move a1,v0
400ff0: 3c020047  lui v0,0x47
400ff4: 24447164  addiu a0,v0,29028
400ff8: 8f82816c  lw v0,-32404(gp)
400ffc: 0040c825  move t9,v0
401000: 04111edb  bal 408b70 <_isoc99_scanf>
401004: 00200825  move at,at
401008: 8fdc0010  lw gp,16(s8)
40100c: 8fc30024  lw v1,36(s8)
401010: 24020001  li v0,1
401014: 00000000  nop
401018: 00200825  move at,at
40101c: 24020001  li v0,1
401020: afc2001c  sw v0,28(s8)
401024: 8fc30024  lw v1,36(s8)
401028: 24020003  li v0,3
40102c: 00000000  nop
401030: 00200825  move at,at
401034: 24020001  li v0,1
401038: afc2001c  sw v0,28(s8)
40103c: 8fc30024  lw v1,36(s8)
401040: 24020005  li v0,5
401044: 00000000  nop
401048: 00200825  move at,at
40104c: 24020001  li v0,1
401050: afc20018  sw v0,24(s8)
401054: 8fc30024  lw v1,36(s8)
401058: 24020008  li v0,8
40105c: 00000000  nop
401060: 00200825  move at,at
401064: 24020001  li v0,1
401068: afc2001c  sw v0,28(s8)
40106c: 8fc30024  lw v1,36(s8)
401070: 2402000a  li v0,10
401074: 00000000  nop
401078: 00200825  move at,at
40107c: 24020001  li v0,1
401080: afc2001c  sw v0,28(s8)
401084: 8fc30024  lw v1,36(s8)
401088: 2402000d  li v0,13
40108c: 00000000  nop
401090: 00200825  move at,at
401094: 24020001  li v0,1
401098: afc2001c  sw v0,28(s8)
40109c: 8fc20028  lw v0,40(s8)
4010a0: 00402825  move a1,v0
4010a4: 8fc40018  lw a0,24(s8)
4010a8: 0c1003c4  jal 400f10 <ejecutar>
4010ac: 00200825  move at,at
4010b0: 8fdc0010  lw gp,16(s8)

```

Figura 3-11: Manipulación Programa 2

Ahora, al momento de ejecutar el programa, no importa la clave que se ingrese, se muestra el cuadrado del número, comportándose de la manera esperada. Ejemplo de ello, se muestra en la siguiente figura:



```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
root@694c70d78ef3:/orga/orga# ./p2
Clave (número): 789654321
Numero a calcular (mayor a cero): 456
Resultado: 207936
root@694c70d78ef3:/orga/orga#
```

*Figura 3-12: Prueba Programa 2*





## **CAPÍTULO 4. CONCLUSIONES**

Una vez culminado el laboratorio, se puede decir que de alguna manera u otra, se han cumplido con los objetivos propuestos, ya que se manipuló el comportamiento de los programas pedidos, entregando los resultados esperados. De esta forma, se explotó la vulnerabilidad de los códigos, para modificar su comportamiento. Sin duda, esto representa una pequeña introducción a la exploración de la seguridad informática, estudiando cada una de las posibles vulnerabilidades que puede afectar al software, de manera que se pueda tener cuidado en la forma en que se desarrolla cualquier software, siguiendo todos los estándares, protocolos y leyes que se tienen para minimizar los posibles riesgos a la infraestructura de este.

Durante el desarrollo de este laboratorio, la principal dificultad se presentó al momento de analizar los códigos de cada uno de los programas, tratando de entender su comportamiento y el uso de cada una de las instrucciones allí presentes. Esta dificultad aumentó en gran manera al tratarse de una decompilación a lenguaje MIPS, debido a que es un lenguaje de bajo nivel, haciendo difícil su comprensión.

En cuanto a las herramientas que se utilizan, el tutorial de decompilación presente en el curso de Usach-Virtual fue de gran ayuda para comprender su funcionamiento, así se pudo obtener los elementos necesarios para realizar el trabajo. Así mismo, con el ejemplo allí descrito se pudo realizar de manera más rápida la modificación al Programa 2, en el caso del Programa 1, fue más difícil, pues su contexto es distinto y no se entendió en un principio lo que se tenía que hacer en esta ocasión.



## **CAPÍTULO 5. BIBLIOGRAFÍA**