




GDB动态调试攻防世界Simple-Check-100

原创

Tr0e  于 2021-10-05 14:20:54 发布  227  收藏 2

分类专栏: [CTF之路](#) 文章标签: [GDB 逆向分析](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: https://blog.csdn.net/weixin_39190897/article/details/120595138

版权



[CTF之路 专栏收录该内容](#)

17 篇文章 27 订阅

订阅专栏

文章目录

题目

IDA静态分析

gdb动态调试

gdb 基本使用

gdb peda插件

函数校验绕过

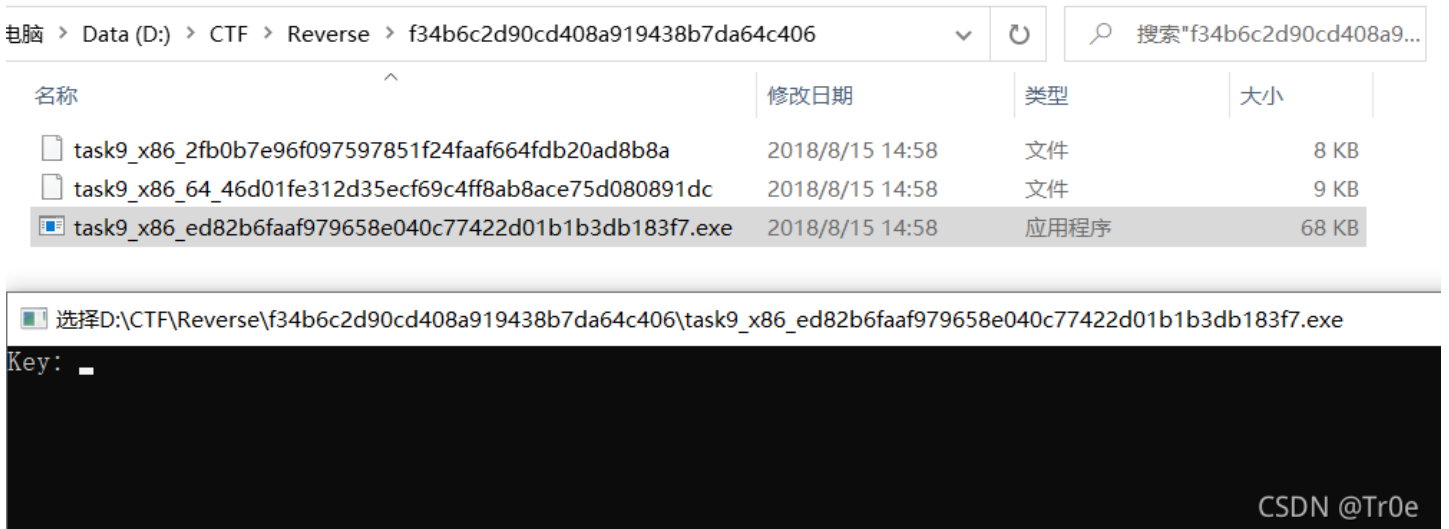
总结

题目

攻防世界 Reverse 高手区题目链接 [simple-check-100](#)，如下：



解压缩得到三个文件：

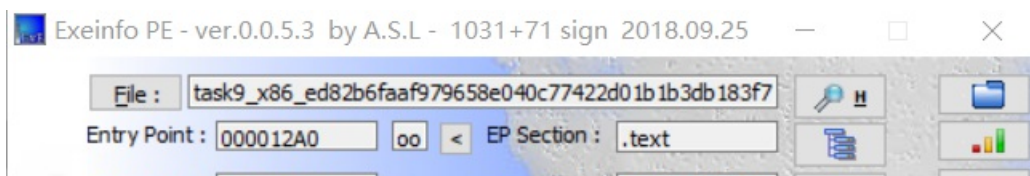


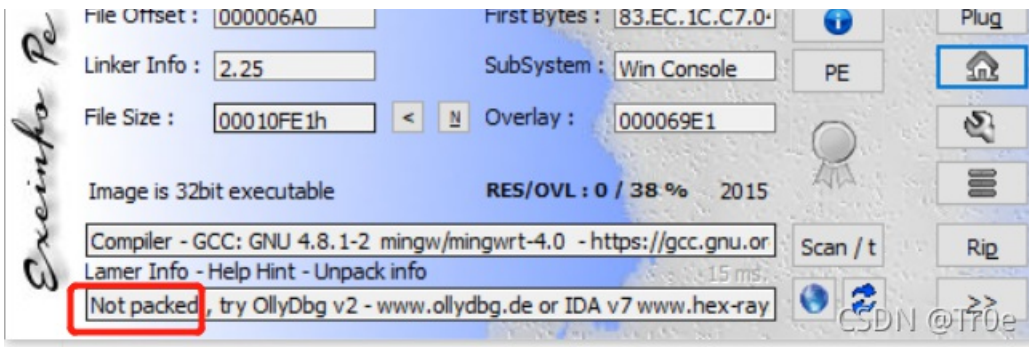
三个文件依次是一个 32 位 elf，一个 64 位 elf 和一个 32 位 exe。

ELF 文件 (Executable Linkable Format) 是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。Linux 下的目标文件和可执行文件都按照该格式进行存储，它是 Linux 的主要可执行文件格式。

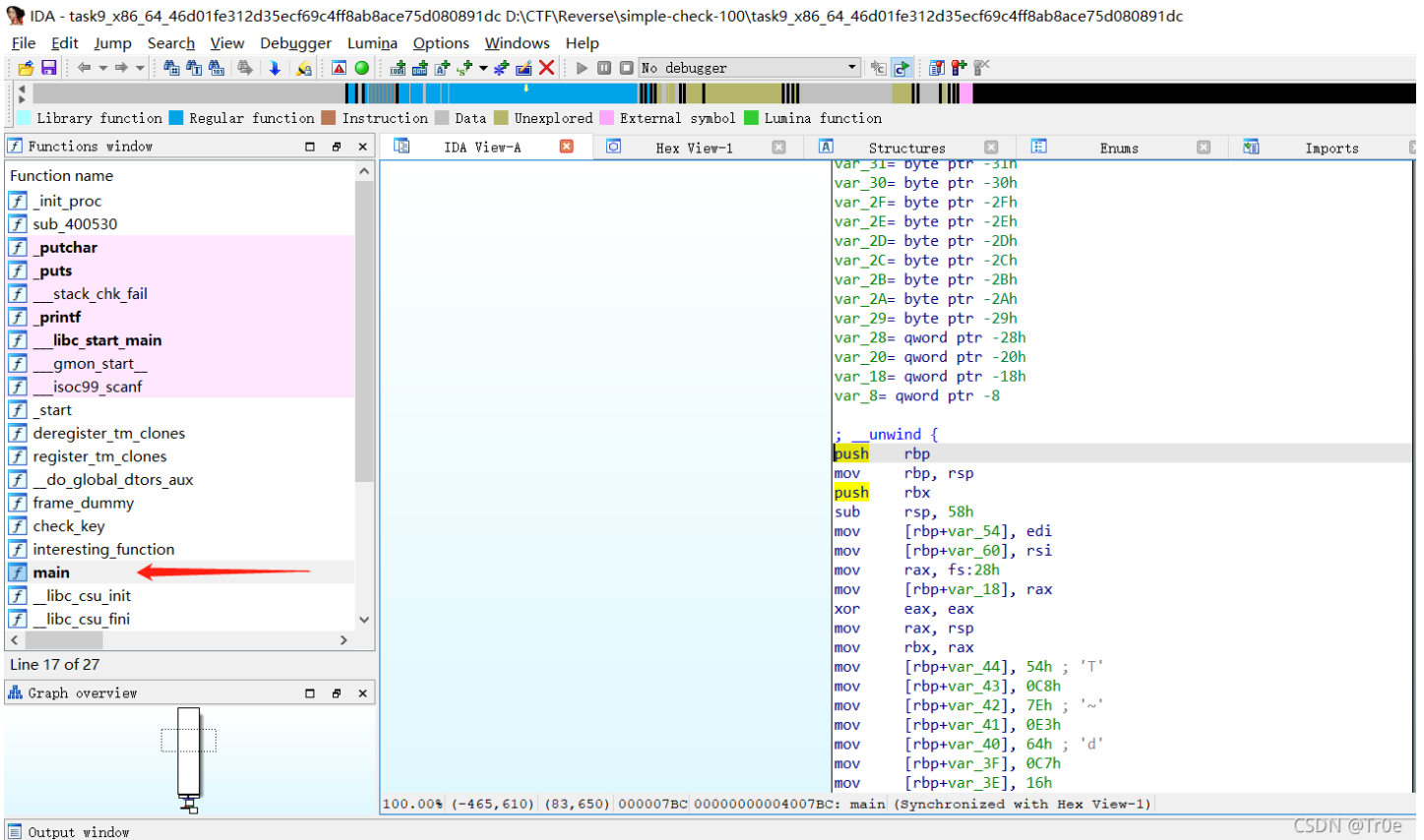
IDA静态分析

1、查壳发现未加壳：

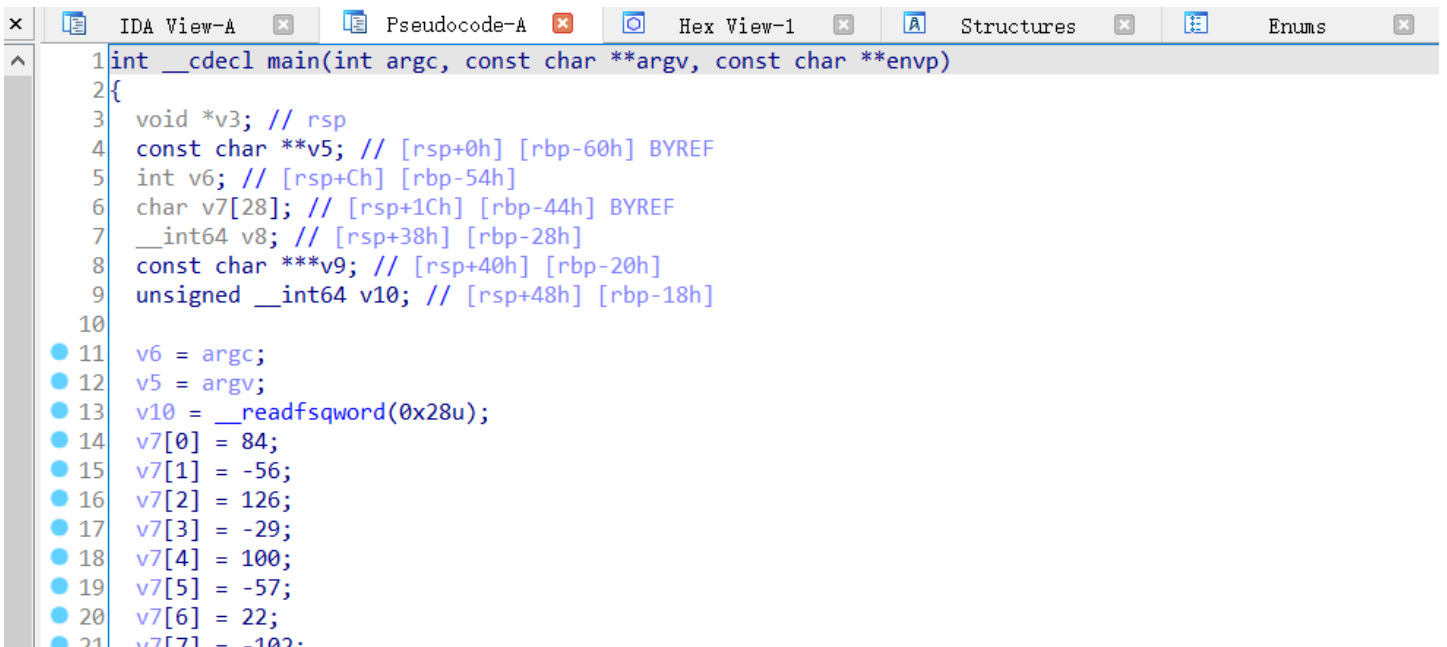




2、这 3 个文件拖进 IDA 后的反汇编结果是大体一致的，以 64 位 elf 文件为例进行分析，定位到 main 函数：



3、按 F5 查看反汇编结果的 C 语言伪代码：



```
21 v7[7] = -102;  
22 v7[8] = -51;  
23 v7[9] = 17;  
24 v7[10] = 101;  
25 v7[11] = 50;  
26 v7[12] = 45;  
27 v7[13] = -29;  
28 v7[14] = -45;  
29 v7[15] = 67;  
30 v7[16] = -110;  
31 v7[17] = -87;  
32 v7[18] = -99;  
33 v7[19] = -46;  
34 v7[20] = -26;  
35 v7[21] = 109;  
36 v7[22] = 44;  
37 v7[23] = -45;
```

000007BC main:1 (4007BC) CSDN@Tr0e

完整代码如下：

```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    void *v3; // rsp
    const char **v5; // [rsp+0h] [rbp-60h] BYREF
    int v6; // [rsp+Ch] [rbp-54h]
    char v7[28]; // [rsp+1Ch] [rbp-44h] BYREF
    __int64 v8; // [rsp+38h] [rbp-28h]
    const char ***v9; // [rsp+40h] [rbp-20h]
    unsigned __int64 v10; // [rsp+48h] [rbp-18h]

    v6 = argc;
    v5 = argv;
    v10 = __readfsqword(0x28u);
    v7[0] = 84;
    v7[1] = -56;
    v7[2] = 126;
    v7[3] = -29;
    v7[4] = 100;
    v7[5] = -57;
    v7[6] = 22;
    v7[7] = -102;
    v7[8] = -51;
    v7[9] = 17;
    v7[10] = 101;
    v7[11] = 50;
    v7[12] = 45;
    v7[13] = -29;
    v7[14] = -45;
    v7[15] = 67;
    v7[16] = -110;
    v7[17] = -87;
    v7[18] = -99;
    v7[19] = -46;
    v7[20] = -26;
    v7[21] = 109;
    v7[22] = 44;
    v7[23] = -45;
    v7[24] = -74;
    v7[25] = -67;
    v7[26] = -2;
    v7[27] = 106;
    v8 = 19LL;
    v3 = alloca(32LL);
    v9 = &v5;
    printf("Key: ");
    __isoc99_scanf("%s", v9);
    if ( (unsigned int)check_key(v9) )
        interesting_fuction(v7);
    else
        puts("Wrong");
    return 0;
}

```

可以看到，程序的核心是让输入一个字符串，然后使用 check_key 函数进行判断，如果返回非 0（为真），则执行 interesting_fuction 函数。

4、双击跟进查看 interesting_fuction 函数伪代码：

```
IDA View-A Pseudocode-A Hex View-1 Structures Enums
1 int __fastcall interesting_fuction(__int64 a1)
2 {
3     int *v1; // rax
4     unsigned int v3; // [rsp+1Ch] [rbp-24h] BYREF
5     int i; // [rsp+20h] [rbp-20h]
6     int j; // [rsp+24h] [rbp-1Ch]
7     __int64 v6; // [rsp+28h] [rbp-18h]
8     int *v7; // [rsp+30h] [rbp-10h]
9     unsigned __int64 v8; // [rsp+38h] [rbp-8h]
10
11     v8 = __readfsqword(0x28u);
12     LODWORD(v1) = a1;
13     v6 = a1;
14     for ( i = 0; i <= 6; ++i )
15     {
16         v3 = *(_DWORD *) (4LL * i + v6) ^ 0xDEADBEEF;
17         v1 = (int *)&v3;
18         v7 = (int *)&v3;
19         for ( j = 3; j >= 0; --j )
20             LODWORD(v1) = putchar((char)((_BYTE *)v7 + j) ^ flag_data[4 * i + j]);
21     }
22     return (int)v1;
23 }
```

CSDN @Tr0e

该函数对输入的 v7（未知）进行加密处理，最后 putchar 输出。

5、check_key 函数是一个计算校验和与正确的校验和对比的一个函数：

check_key 函数是一个计算校验和与正确的校验和对比的一个函数，ida 自动分析出来是这样：

```
1 |B00L8 __fastcall check_key(__int64 a1)
2 {
3     int checksum; // [rsp+8h] [rbp-10h]
4     signed int i; // [rsp+Ch] [rbp-Ch]
5
6     checksum = 0;
7     for ( i = 0; i <= 4; ++i )
8         checksum += *(_DWORD *) (4LL * i + a1);
9     return checksum == 0xDEADBEEF;
10 }
```

<https://blog.csdn.net/plp9822>

经过简单的修改参数类型之后如下，看起来更为清晰，input_key 作为一个 DWORD 数组，有 5 个元素，也就是 20 个字节的长度：

```
1 |B00L8 __fastcall check_key(_DWORD *input_key)
2 {
3     int checksum; // [rsp+8h] [rbp-10h]
4     signed int i; // [rsp+Ch] [rbp-Ch]
5
6     checksum = 0;
7     for ( i = 0; i <= 4; ++i ) // input_key
8         checksum += input_key[i];
9     return checksum == 0xDEADBEEF;
10 }
```

<https://blog.csdn.net/plp9822>

算法就是将 input_key 看做一个 DWORD 数组将其每一项相加得到校验和，而这个值要为 0xDEADBEEF 才是正确的，根据这个本来想爆破得到 key 的，可惜耗时太大不行。

CSDN @Tr0e

【解题思路】

只要让程序绕过 check_key 函数的检查，强行执行 interesting_fuction 函数，就能获得目标 Flag。故可以对程序进行动态调试，将 check_key 的返回结果改为 1，就能调用 interesting_fuction 函数并得到正确的 key。

gdb 动态调试

GDB（GNU Debugger）是一个由 GNU 开源组织发布的、UNIX/LINUX 操作系统下的、基于命令行的、功能强大的程序调试工具。像所有的调试器一样，GDB 可以让你调试一个程序，包括让程序在你希望的地方停下，此时你可以查看变量、寄存器、内存及堆栈，更进一步你可以修改变量及内存值。对于一名 Linux 下工作的 C/C++ 程序员或逆向工作者，gdb 是必不可少的工具。

gdb 基本使用

gdb 的常用命令如下：

表 3.9 gdb 的常用命令

命令	简写	功能
list	l	列出源码
break	b	设置断点
run	r	从头开始运行程序
continue	c	从停止处继续运行程序
next	n	向前执行一句（不进入被调用函数中）
step	s	向前执行一句（可进入被调用函数中）
return	ret	从当前函数返回
print	p	显示变量或表达式的值
x	x	显示内存值
backtrace	bt	显示调用栈
quit	q	退出 gdb
symbol-file	sy	从可执行文件中加载符号表

CSDN @Tr0e

本人在自己的 VPS 服务器（Centos8）上安装 gdb，步骤如下：

```
#安装C++编译器
yum install gcc gcc-c++
#下载源码包
wget http://ftp.gnu.org/gnu/gdb/gdb-10.2.tar.gz
#编译安装
tar -zxvf gdb-10.2.tar.gz
cd gdb-10.2
./configure --with-python=/usr/bin/python --enable-targets=all
make
sudo make install
#查看是否安装成功
gdb -v
```

【注意防坑】编译 gdb 时必须加上 `--with-python` 参数指定本地的 Python 路径，否则后续使用 gdb 插件时将各种报错（为此我浪费了整整一天……最终参照 [博文](#) 找到了该解决的办法）。

如下已成功安装 gdb 工具:

```
1 MyVPS x +
[root@hwc-hwp-587401-751218 ~]# gdb -v
GNU gdb (GDB) 10.2
Copyright (C) 2021 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.
[root@hwc-hwp-587401-751218 ~]#
[root@hwc-hwp-587401-751218 ~]#
```

CSDN @Tr0e

GDB基础用法演示

下面给出一个具有 test.c 的程序:

```
#include <stdio.h>
int getSum(int n) {
    int sum=0,i;
    for (i=1;i<=n;i++)
        sum+=i;
    return sum;
}
int main(){
    int res=getSum(100);
    printf("1+2+...+100=%d\n",res);
}
```

1、使用 gcc 编译 C 语言程序:

```
[root@hwc-hwp-587401-751218 Test]# ls
test.c
[root@hwc-hwp-587401-751218 Test]# gcc -g test.c -o test
[root@hwc-hwp-587401-751218 Test]# ls
test test.c
[root@hwc-hwp-587401-751218 Test]#
```


【注意】如果要调试程序，则在进行 gcc 编译的时候要加上 -g 参数（表示 debug 模式），即 `gcc -g test.c -o test`（如果是 C++ 程序的话则是 `g++ -g test.cpp -o test`）。如果没有 -g 参数，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。

2、执行命令 `gdb test`，开始使用 GDB 对生成的 test 可执行程序进行调试（GDB 会显示自己的提示符 gdb，提示并等待你输入调试命令）:

```
[root@hwc-hwp-587401-751218 Test]# ls
test test.c
[root@hwc-hwp-587401-751218 Test]# gdb test
GNU gdb (GDB) 10.2
Copyright (C) 2021 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
```




```
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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 test...
(gdb) 
```


CSDN @Tr0e

3、输入 `l` 命令（相当于 list），gdb 将从第一行开始列出源码（默认前 10 行）：

```
Reading symbols from test...
(gdb) l
1      #include <stdio.h>
2      int getSum(int n) {
3          int sum=0,i;
4          for (i=1;i<=n;i++)
5              sum+=i;
6          return sum;
7      }
8      int main(){
9          int res=getSum(100);
10         printf("1+2+...+100=%d\n",res);
(gdb) 
```

CSDN @Tr0e

直接回车表示，重复上一次命令（继续输出第 10-20 行的源码）：

```
Reading symbols from test...
(gdb) l
1      #include <stdio.h>
2      int getSum(int n) {
3          int sum=0,i;
4          for (i=1;i<=n;i++)
5              sum+=i;
6          return sum;
7      }
8      int main(){
9          int res=getSum(100);
10         printf("1+2+...+100=%d\n",res);
(gdb)
11     }
(gdb) 
```

CSDN @Tr0e

4、执行命令 `break 9`、`break getSum`，表示在第 9 行处、getSum 函数处设置断点，同时执行命令 `info break` 可查看设置的断点信息：

```
(gdb) break 9
Breakpoint 1 at 0x4005cc: file test.c, line 9.
(gdb) break getSum
```

```
Breakpoint 2 at 0x40059d: file test.c, line 3.
```

```
(gdb) info break
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000000004005cc	in main at test.c:9
2	breakpoint	keep	y	0x000000000040059d	in getSum at test.c:3

```
(gdb) █
```

CSDN @Tr0e

5、执行命令 `run` (简写 `r`)，开始从头运行程序，直到程序结束或者遇到断点并等待下一个命令：

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000000004005cc	in main at test.c:9
2	breakpoint	keep	y	0x000000000040059d	in getSum at test.c:3

```
(gdb) run
```

```
Starting program: /root/Test/test
```

```
Breakpoint 1, main () at test.c:9
```

```
9          int res=getSum(100);
```

```
(gdb)
```

CSDN @Tr0e

6、执行命令 `continue` (简写 `c`)，表示从暂停处继续运行程序：

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 2, getSum (n=100) at test.c:3
```

```
3          int sum=0,i;
```

```
(gdb) █
```

7、执行 `step` 命令，表示向前执行一步（可进入被调用函数中），进一步可利用 `print i`（变量名）来查看变量的值：

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 2, getSum (n=100) at test.c:3
```

```
3          int sum=0,i;
```

```
(gdb) step
```

```
4          for (i=1;i<=n;i++)
```

```
(gdb) print i
```

```
$1 = 0
```

```
(gdb) step
```

```
5          sum+=i;
```

```
(gdb) print i
```

```
$2 = 1
```

```
(gdb)
```

CSDN @Tr0e

8、执行命令 `backtrace` (简写 `bt`)，可查询当前函数调用栈，执行命令 `finish`，可退出当前函数并返回到上层函数中（本例为 `main` 主函数）：

```
(gdb) bt
```

```
#0 getSum (n=100) at test.c:5
```

```
#1 0x00000000004005d6 in main () at test.c:9
```

```
71 0x0000000000000000 in main () at test.c:9
(gdb) finish
Run till exit from #0  getSum (n=100) at test.c:5
0x0000000000004005d6 in main () at test.c:9
9          int res=getSum(100);
Value returned is $2 = 5050
(gdb) bt
#0  0x0000000000004005d6 in main () at test.c:9
(gdb) 
```

CSDN @Tr0e

9、执行命令 `set res = 6666`，可改变程序中指定变量的值：

```
(gdb) finish
Run till exit from #0  getSum (n=100) at test.c:4
0x0000000000004005d6 in main () at test.c:9
9          int res=getSum(100);
Value returned is $4 = 5050
(gdb) next
10         printf("1+2+...+100=%d\n",res);
(gdb) print res
$5 = 5050
(gdb) set res=6666
(gdb) print res
$6 = 6666
(gdb) 
```

CSDN @Tr0e

10、`info reg` 命令可查看寄存器使用情况，`info stack` 命令可查看堆栈使用情况：

```
(gdb) info reg
rax          0x13ba          5050
rbx          0x0             0
rcx          0x7ffff7dca758 140737351821144
rdx          0x7ffff7ffe538 140737488348472
rsi          0x7ffff7ffe528 140737488348456
rdi          0x64            100
rbp          0x7ffff7ffe440 0x7ffff7ffe440
rsp          0x7ffff7ffe430 0x7ffff7ffe430
r8           0x7ffff7dcbd20 140737351826720
r9           0x7ffff7dcbd20 140737351826720
r10          0x1             1
r11          0x202           514
r12          0x4004b0        4195504
r13          0x7ffff7ffe520 140737488348448
r14          0x0             0
r15          0x0             0
rip          0x4005d9        0x4005d9 <main+21>
eflags      0x202           [ IF ]
cs           0x33            51
ss           0x2b            43
ds           0x0             0
es           0x0             0
fs           0x0             0
gs           0x0             0
(gdb) info stack
```

```
(gdb) info stack
#0  main () at test.c:10
(gdb)
```

CSDN @Tr0e

最后执行命令 `quit`（简写 `q`），即可退出 `gdb`：

```
(gdb) step
1+2+...+100=6666
11    }
(gdb) step
0x00007ffff7a2e493 in __libc_start_main () from /lib64/libc.so.6
(gdb)
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 44527) exited normally]
(gdb) quit
[root@hwc-hwp-587401-751218 Test]#
```

CSDN @Tr0e

而未被篡改变量值的程序正常的运行输出应当如下：

```
(gdb) c
Continuing.
1+2+...+100=5050
[Inferior 1 (process 20069) exited normally]
(gdb) quit
[root@hwc-hwp-587401-751218 Test]#
```

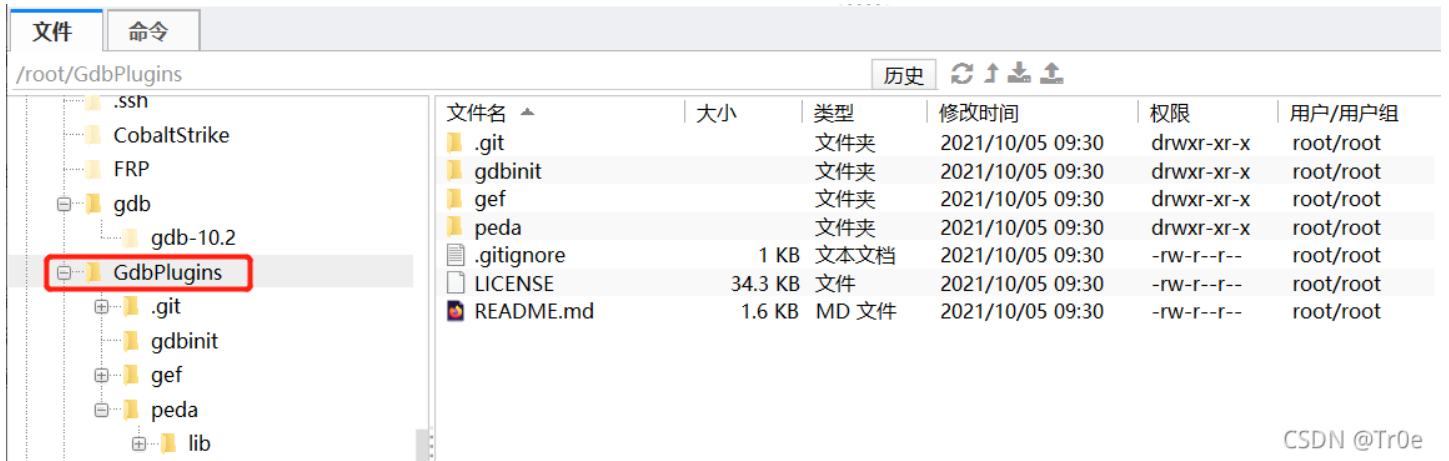
gdb peda插件

从上面的演示实例中也可以看出，`gdb` 非常强大，但是在调试过程中对于需要查看的辅助信息（如寄存器信息）都要手动输入命令，未免有点麻烦，所以就出现了插件，把某一些经常要查看的信息每一步都自动帮你显示出来，方便调试。一般来说有常用的 3 个 GDB 插件：`peda`、`gef`、`gdbinit`。

，完整介绍可参见：[GDB的三个插件（gef gdbinit peda）](#)。Github 上已经有人把这 3 个插件的项目合成了一个，使得插件的安装使用很方便：

```
# 从 Github 将汇聚了 gdb 的 3 个插件的项目拷贝到本地
git clone https://github.com/gatieme/GdbPlugins.git ~/GdbPlugin
# GdbPlugins 文件夹包含的 3 个插件对应启动命令：
echo "source ~/GdbPlugins/peda/peda.py" > ~/.gdbinit
echo "source ~/GdbPlugins/gef/gef.py" > ~/.gdbinit
echo "source ~/GdbPlugins/gdbinit/gdbinit" > ~/.gdbinit
```

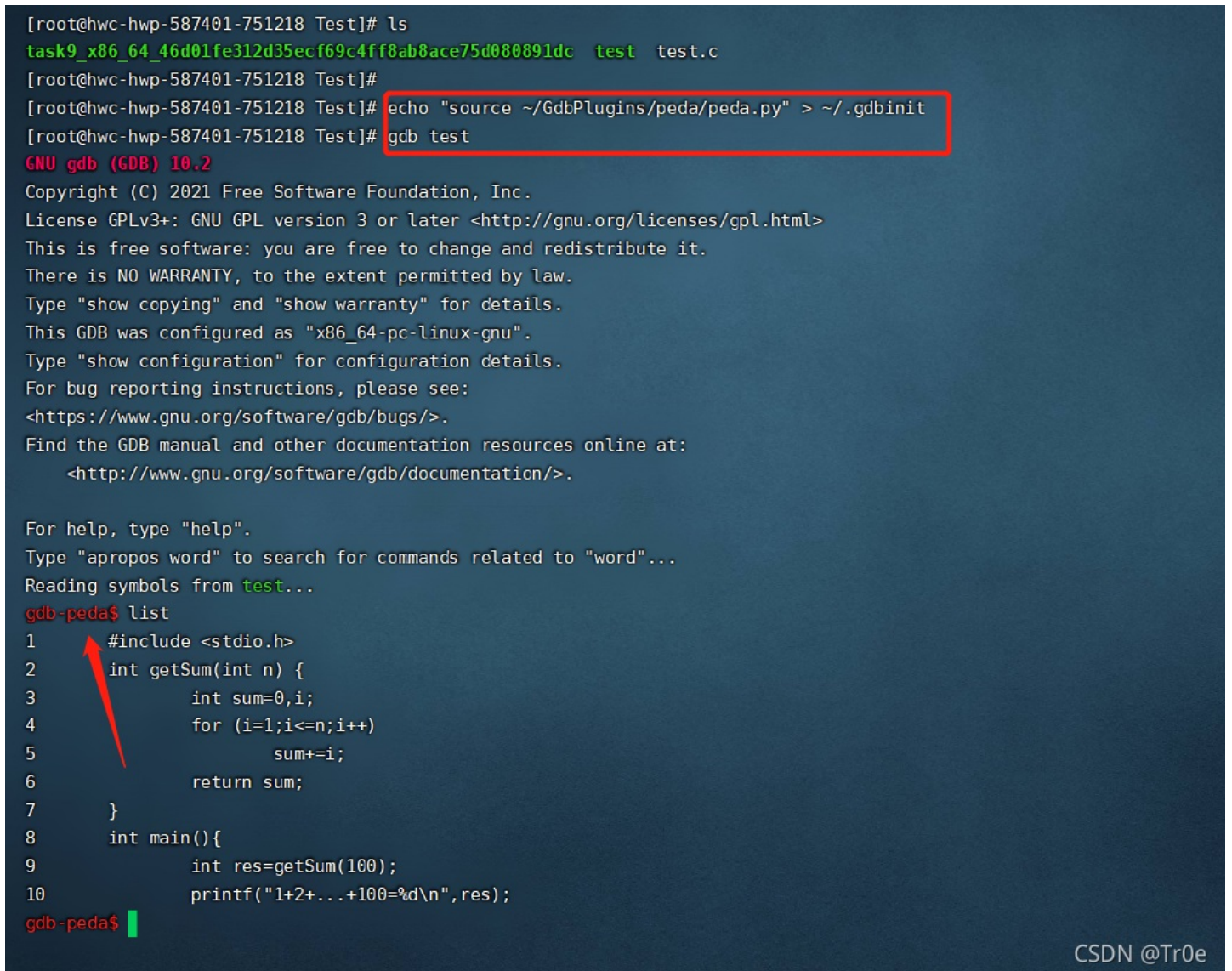
成功克隆到本地后可以看到包含 3 个插件的文件夹：



CSDN @Tr0e

下文将介绍其中的 peda 插件——peda 是 gdb 调试工具的插件，用于增强 gdb 的调试能力，同时增强 gdb 的显示：在调试过程中着色并显示反汇编代码，寄存器和内存信息（单纯的 gdb 在内存信息、汇编信息的显示和查看上的不方便）。

1、设置 gdb 以 peda 插件的执行形式启动，并调试上面编译好的 test 程序：



CSDN @Tr0e

2、输入 start 命令开始调试程序，从下图中可以看到寄存器 (registers)，汇编代码 (code)，栈空间数据 (stack) 等信息：



```
[-----]
RAX: 0x4005c4 (<main>: push rbp)
RBX: 0x0
RCX: 0x7ffff7dca758 --> 0x7ffff7dcb20 --> 0x0
RDX: 0x7ffffffffff538 --> 0x7ffffffffff789 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=01;0
5;37;41:su=37;41:sg=30;43:ca=30;41:tw=30;42:cw=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01"...)
RSI: 0x7ffffffffff528 --> 0x7ffffffffff779 ("/root/Test/test")
RDI: 0x1
RBP: 0x7ffffffffff440 --> 0x400600 (<_libc_csu_init>: endbr64)
RSP: 0x7ffffffffff430 --> 0x7ffffffffff520 --> 0x1
RIP: 0x4005cc (<main+8>: mov edi,0x64)
R8 : 0x7ffff7dcb20 --> 0x0
R9 : 0x7ffff7dcb20 --> 0x0
R10: 0x1
R11: 0x202
R12: 0x4004b0 (<_start>: endbr64)
R13: 0x7ffffffffff520 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----]
0x4005c4 <main>: push rbp
0x4005c5 <main+1>: mov rbp,rsp
0x4005c8 <main+4>: sub rsp,0x10
=> 0x4005cc <main+8>: mov edi,0x64
0x4005d1 <main+13>: call 0x400596 <getSum>
0x4005d6 <main+18>: mov DWORD PTR [rbp-0x4],eax
0x4005d9 <main+21>: mov eax,DWORD PTR [rbp-0x4]
0x4005dc <main+24>: mov esi,eax
[-----]
0000| 0x7ffffffffff430 --> 0x7ffffffffff520 --> 0x1
0008| 0x7ffffffffff438 --> 0x0
0016| 0x7ffffffffff440 --> 0x400600 (<_libc_csu_init>: endbr64)
0024| 0x7ffffffffff448 --> 0x7ffff7a2e493 (<_libc_start_main+243>: mov edi,eax)
0032| 0x7ffffffffff450 --> 0x7ffffffffff528 --> 0x7ffffffffff779 ("/root/Test/test")
0040| 0x7ffffffffff458 --> 0x7ffffffffff528 --> 0x7ffffffffff779 ("/root/Test/test")
CSDN @Tr0e
```

在前面的文章 [浅析缓冲区溢出漏洞的利用与Shellcode编写](#) 中曾经介绍了函数调用过程中的内存堆栈变化，建议读者同步阅读博文 [基于GDB-peda汇编调试理解函数调用栈](#)，作者利用 peda 插件来汇编调试一段程序，帮助深入理解函数调用栈。

函数校验绕过

返回到 CTF 题目中，在 Linux 中正常运行程序如下：

```
[root@hwc-hwp-587401-751218 Test]# ls
task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc test test.c
[root@hwc-hwp-587401-751218 Test]#
[root@hwc-hwp-587401-751218 Test]# ./task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc
Key: 123
Wrong
[root@hwc-hwp-587401-751218 Test]#
CSDN @Tr0e
```

下面开始借助 gdb 对程序进行动态调试，将 check_key 函数的返回结果改为真即可正确的 key。

1、启动 gdb 调试程序，执行命令 `break main` 在 main 函数设置断点（Breakpoint 1 at 0x4007c0），然后执行命令 `run` 开始运行程序：

```
[root@hwc-hwp-587401-751218 Test]# gdb task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc
GNU gdb (GDB) 10.2
Copyright (C) 2021 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 "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
```

```
<https://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 task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc...
(No debugging symbols found in task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc)
gdb-peda$ break main
Breakpoint 1 at 0x4007c0
gdb-peda$ run
Starting program: /root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc
[-----Registers-----]
RAX: 0x4007bc (<main>: push rbp)
RBX: 0x0
RCX: 0x7ffff7dca758 --> 0x7ffff7dcb20 --> 0x0
RDX: 0x7fffffe4d8 --> 0x7fffffe758 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=01;0
5;37;41:su=37;41:sg=30;43:ca=30;41:tw=30;42:cw=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01"... )
RSI: 0x7fffffe4c8 --> 0x7fffffe717 ("/root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc")
RDI: 0x1
RBP: 0x7fffffe3e0 --> 0x400920 (<_libc_csu_init>: push r15)
RSP: 0x7fffffe3e0 --> 0x400920 (<_libc_csu_init>: push r15)
RIP: 0x4007c0 (<main+4>: push rbx)
R8 : 0x7ffff7dcb20 --> 0x0
R9 : 0x7ffff7dcb20 --> 0x0
R10: 0x3
R11: 0x7ffff7a2e3a0 (<_libc_start_main>: endbr64)
CSDN @Tr0e
```

程序暂停在断点处:

```
RSI: 0x7fffffe4c8 --> 0x7fffffe717 ("/root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc")
RDI: 0x1
RBP: 0x7fffffe3e0 --> 0x400920 (<_libc_csu_init>: push r15)
RSP: 0x7fffffe3e0 --> 0x400920 (<_libc_csu_init>: push r15)
RIP: 0x4007c0 (<main+4>: push rbx)
R8 : 0x7ffff7dcb20 --> 0x0
R9 : 0x7ffff7dcb20 --> 0x0
R10: 0x3
R11: 0x7ffff7a2e3a0 (<_libc_start_main>: endbr64)
R12: 0x4005b0 (<_start>: xor ebp,ebp)
R13: 0x7fffffe4c0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----Code-----]
0x4007bb <interesting_function+186>: ret
0x4007bc <main>: push rbp
0x4007bd <main+1>: mov rbp, rsp
=> 0x4007c0 <main+4>: push rbx
0x4007c1 <main+5>: sub rsp, 0x58
0x4007c5 <main+9>: mov DWORD PTR [rbp-0x54], edi
0x4007c8 <main+12>: mov QWORD PTR [rbp-0x60], rsi
0x4007cc <main+16>: mov rax, QWORD PTR fs:0x28
[-----Stack-----]
0000| 0x7fffffe3e0 --> 0x400920 (<_libc_csu_init>: push r15)
0008| 0x7fffffe3e8 --> 0x7ffff7a2e493 (<_libc_start_main+243>: mov edi, eax)
0016| 0x7fffffe3f0 --> 0x0
0024| 0x7fffffe3f8 --> 0x7fffffe4c8 --> 0x7fffffe717 ("/root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc")
0032| 0x7fffffe400 --> 0x100000000
0040| 0x7fffffe408 --> 0x4007bc (<main>: push rbp)
0048| 0x7fffffe410 --> 0x0
0056| 0x7fffffe418 --> 0xae0616905faa880
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000004007c0 in main ()
gdb-peda$
```

2、上述程序暂停在 main 函数断点处，执行命令 next 开始一步步单步执行程序（一直按回车键重复执行 next 命令即可），直到运行至 check_key 函数所在位置，随意输入 key 之后会有判断函数，所以注意看 check_key 所在位置：

```
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

```

[-----code-----]
0x4008c4 <main+264>: mov    rsi, rax
0x4008c7 <main+267>: mov    edi, 0x4009f2
0x4008cc <main+272>: mov    eax, 0x0
=> 0x4008d1 <main+277>: call  0x4005a0 <_isoc99_scanf@plt>
0x4008d6 <main+282>: mov    rax, QWORD PTR [rbp-0x20]
0x4008da <main+286>: mov    rdi, rax
0x4008dd <main+289>: call  0x4006a6 <check_key>
0x4008e2 <main+294>: test   eax, eax

Gussed arguments:
arg[0]: 0x4009f2 --> 0x676e6f7257007325 ('%s')
arg[1]: 0x7fffffff360 --> 0x0

[-----stack-----]
0000| 0x7fffffff360 --> 0x0
0008| 0x7fffffff368 --> 0x0
0016| 0x7fffffff370 --> 0x0
0024| 0x7fffffff378 --> 0x0
0032| 0x7fffffff380 --> 0x7fffffff4c8 --> 0x7fffffff717 ("/root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc")
0040| 0x7fffffff388 --> 0x100000000
0048| 0x7fffffff390 --> 0x0
0056| 0x7fffffff398 --> 0xe37ec85400000000

Legend: code, data, rodata, value
0x0000000004008d1 in main ()
gdb-peda$
Key: 123

```

3、当判断函数执行完之后，再次跳到 `test eax, eax` 时候，可以用 `printi $eax` 查看寄存器的值，发现是 0：

```

R12: 0x4005b0 (<_start>:      xor    ebp, ebp)
R13: 0x7fffffff4c0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x217 (CARRY PARITY ADJUST zero sign trap INTERRUPT direction overflow)

[-----code-----]
0x4008d6 <main+282>: mov    rax, QWORD PTR [rbp-0x20]
0x4008da <main+286>: mov    rdi, rax
0x4008dd <main+289>: call  0x4006a6 <check_key>
=> 0x4008e2 <main+294>: test   eax, eax
0x4008e4 <main+296>: je    0x4008f4 <main+312>
0x4008e6 <main+298>: lea   rax, [rbp-0x44]
0x4008ea <main+302>: mov    rdi, rax
0x4008ed <main+305>: call  0x400701 <interesting_function>

[-----stack-----]
0000| 0x7fffffff360 --> 0x333231 ('123')
0008| 0x7fffffff368 --> 0x0
0016| 0x7fffffff370 --> 0x0
0024| 0x7fffffff378 --> 0x0
0032| 0x7fffffff380 --> 0x7fffffff4c8 --> 0x7fffffff717 ("/root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc")
0040| 0x7fffffff388 --> 0x100000000
0048| 0x7fffffff390 --> 0x0
0056| 0x7fffffff398 --> 0xe37ec85400000000

Legend: code, data, rodata, value
0x0000000004008e2 in main ()
gdb-peda$ print $eax
$7 = 0x0

```

4、这样的话不会跳转藏有 flag 函数的位置，所以执行命令 `set $eax=1`，手动篡改寄存器 `eax` 的值，最后执行命令 `continue`，运行程序直至程序终止，可获得 flag 值如下：

```

=> 0x4008e2 <main+294>: test   eax, eax
0x4008e4 <main+296>: je    0x4008f4 <main+312>
0x4008e6 <main+298>: lea   rax, [rbp-0x44]
0x4008ea <main+302>: mov    rdi, rax
0x4008ed <main+305>: call  0x400701 <interesting_function>

[-----stack-----]
0000| 0x7fffffff360 --> 0x333231 ('123')
0008| 0x7fffffff368 --> 0x0
0016| 0x7fffffff370 --> 0x0

```



```
0024| 0x7fffffff378 --> 0x0
0032| 0x7fffffff380 --> 0x7fffffff4c8 --> 0x7fffffff717 ("/root/Test/task9_x86_64_46d01fe312d35ecf69c4ff8ab8ace75d080891dc")
0040| 0x7fffffff388 --> 0x100000000
0048| 0x7fffffff390 --> 0x0
0056| 0x7fffffff398 --> 0xe37ec85400000000
[-----]
Legend: code, data, rodata, value
0x0000000004008e2 in main ()
gdb-peda$ print $eax
$7 = 0x0
gdb-peda$ set $eax=1
gdb-peda$ print $eax
$8 = 0x1
gdb-peda$ continue
Continuing.
flag_is_you_know_cracking!!![Inferior 1 (process 1338605) exited normally]
Warning: not running
gdb-peda$
```

提交 flag, over!



总结

本文通过一道 CTF 题目，学习记录了 GDB 调试工具及其 peda 插件的使用，实现了对二进制程序的内存数值进行篡改并成功绕过程序的逻辑校验的目的，这有点类似于 Android 中使用 Frida 对 APP 的函数返回值进行 hook 拦截和篡改。