任务一

步骤一

```
pore@localhost:~$ cd lab8/task
pore@localhost:~/lab8/task$ LD_LIBRARY_PATH=./ ~/gdbserver 127.0.0.1:1234 ./Wann
aMedia.naive
Process ./WannaMedia.naive created; pid = 321
Listening on port 1234
Remote debugging from host 10.0.2.2
```

启动 gdbserver

```
peda-arm > target remote localhost:5555
Remote debugging using localhost:5555
Reading /lib/ld-linux-aarch64.so.1 from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to ac
cess files locally instead.
Reading /lib/ld-linux-aarch64.so.1 from remote target...
Reading symbols from target:/lib/ld-linux-aarch64.so.1...
Reading /usr/lib/debug/.build-id/6f/3b3489a317658fc47876d1d5fa707059844368.debug
 from remote target...
Reading /usr/lib/debug/.build-id/6f/3b3489a317658fc47876d1d5fa707059844368.debug
 from remote target...
Reading symbols from target:/usr/lib/debug/.build-id/6f/3b3489a317658fc47876d1d5
fa707059844368.debug...
Reading /usr/lib/debug/.build-id/b6/23e64d015dc5559de45c0023608651a61503f5.debug
 from remote target...
[----------------------------------REGISTERS-----------------------------------]
X0 : 0x0
X1 : 0x0
X2 : 0x0
X3 : 0x0
X4 : 0x0
X5 : 0x0
X6 : 0x0
```

gdb-multiarch WannaMedia.naive
#等待调试器启动完成
peda-arm> target remote localhost:5555 #localhost:5555 即 QEMU 中 1234, 详见 QEMU 虚拟机启动文件--start_vm.sh

步骤三:

```
Breakpoint 1, 0x0000aaaaaaab7018 in table_retrieve_val ()
peda-arm > x /s $x0
0xaaaaaaacd900: "/tmp/media"
```

任务二

通过 info function 来定位 0xaaaaaaab7018 是哪个函数的位置，找到函数是

```
0x0000aaaaaaab6f9c  table_retrieve_val
```

所以函数的开头地址是 0xaaaaaaab6f9c，在末尾加上 FuncStart('*0xaaaaaaab6f9c')

```
FuncStart('*0xaaaaaaab6f9c')
FuncEnd('*0xaaaaaaab7018') #
```

最后打印出来的开头参数:

```
HID: 0x36
```

结尾参数:

```
peda-arm > c
Continuing.
Decoded String: /tmp/media
```

gdb.execute("run")不需要添加，原因有二:

1.target remote 不支持 run 指令只支持 c 指令

2.加上 c 后会直接自动执行程序，不如手动输入 c 灵活

任务三:

在 main 函数之前就会被杀死，所以在 premain 打断点。然后发现 premain 调用了 magi_init 。 在 libmagi.so 中 找 到 magi_init 函 数 反 编 译 。

```
void __cdecl magi_init()
{
  unsigned int v0; // eax
  time_t t; // [rsp+8h] [rbp-18h] BYREF
  pthread_t id_0[2]; // [rsp+10h] [rbp-10h] BYREF

  id_0[1] = __readfsqword(0x28u);
  v0 = time(&t);
  srand(v0);
  id_0[0] = pthread_self();
  pipe(pipefd);
  pthread_create(id_0, 0LL, (void *(*)(void *))killer, 0LL);
  monitor();
  if ( !firstCheck )
    sleep(1u);
  anti_dbg_running = 1;
}
```

关键在于 monitor 函数。在 monitor 函数中，创建子进程，并且子进程会尝试通过 ptrace 来检测自身是否被调试。

```
  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  pid = getpid();
  sprintf(filename,"/proc/%d/status",(ulong)(uint)pid);
  p = fork();
  _Var2 = p;
  if (p == 0) {
    close(pipefd[0]);
    alive = 0;
    lVar3 = ptrace(PTRACE_TRACEME,0,0,0);
    pt = (int)lVar3;
    while( true ) {
      fd = (FILE *)fopen(filename,"r");
      if ((FILE *)fd == (FILE *)0x0) {
        puts("Parent have already been killed\n");
                    /* WARNING: Subroutine does not return */
        exit(0x7f);
      }
      do {
        pcVar4 = fgets(line,0x100,(FILE *)fd);
        if (pcVar4 == (char *)0x0) goto LAB_0010167d;
        lVar3 = FUN_00101330(line,"TracerPid");
      } while (lVar3 == 0);
      statue = atoi(line + 10);
      write(pipefd[1],&statue,4);
      fclose((FILE *)fd);
      firstCheck = true;
      _Var2 = childpid;
      if (statue != 0) break;
LAB_0010167d:
      sleep(1);
    }
```

通过 puts 的内容“Parent have already been killed”可以推测 p==0 那么就会杀死父进程，所以不能跳进去。

具体来说， fork() 函数创建一个子进程，并且返回子进程的 PID，通过改变 p 能够改变了父进程在控制子进程上的逻辑，从而避免了其中的子进程退出逻辑。

类 似 于 ppt9

回顾：自Trace

代码实现

- fork一个子进程（行02-03）
- 用子进程调试父进程（行07）
- 此时外部调试器便无法调试父进程

- 补充知识
  - 此类反调试机制，往往会在子进程中加一个状态报告过程（行16-行19）
  - 父进程收不到状态报告时，会进行退出等操作

```
01 {
02  pid = getpid();
03  p = fork();//fork出子进程
04  if(p==0) //当前进程为子进程
05  {
06    printf("Child");
07    pt = ptrace(PTRACE_TRACEME, 0, 0, 0); //子进程
08    while(true)
09    {
10      fd = fopen(filename,"r");
11      while(fgets(line,MAX,fd))
12      {
13        if(strstr(line,"TracerPid") != NULL)
14        {
15          printf("line %s",line);
16          int statue = atoi(&line[10]);
17          printf("########## tracer pid:%d", statue);
18          //子进程向父进程写statue值以检查是否仍处于安全状态
19          write(pipefd[1],&statue,4);
20          break;
21        }
22      }
23    }
24  }
25  else{//若当前为父进程，则只打印一条信息出来，而不做其它操作
26    printf("Parent");
27    childpid = p;
28  }
29 }
```

所以不能跳进去子进程

```
0x0000fffff7fa0fc0 <+60>:    bl      0xfffff7fa0c20 <sprintf@plt>
0x0000fffff7fa0fc4 <+64>:    bl      0xfffff7fa0c50 <fork@plt>
0x0000fffff7fa0fc8 <+68>:    str     w0, [sp, #36]
0x0000fffff7fa0fcc <+72>:    ldr     w0, [sp, #36]
0x0000fffff7fa0fd0 <+76>:    cmp     w0, #0x0
0x0000fffff7fa0fd4 <+80>:    b.ne    0xfffff7fa10d4 <monitor+336>  // b.any
```

在这里可以看到在 0xfffff7fa0fd0 打上断点，把 w0（即 x0）改为非 0 的值，直接 set $w0=1 就可以绕过反调试机制。

```
pore@localhost:~/lab8/task$ LD_LIBRARY_PATH=./ ~/gdbserver 127.0.0.1:1234 ./Wann
aMedia
Process ./WannaMedia created; pid = 484
Listening on port 1234
Remote debugging from host 10.0.2.2
Detaching from process 484
opendir failed: No such file or directory
opendir failed: No such file or directory
```

```
[----------------------Legend: code, 

Thread 2.1 "WannaMedia" hit Breakpoint
gi/magi.c:77
77      in magi/magi.c
peda-arm > set $x0=1
peda-arm > c
Continuing.
```

绕过后不会再触发 kill 机制
如果运气够好可以看到循环跳出来的字符/tmp/media，和上一个任务一样

未绕过反调试机制的情况：

```
peda-arm > c
Continuing.

Program terminated with signal SIGKILL, Killed.
The program no longer exists.
```

显然触发了 kill 机制

这个反调试机制的关键在于子进程调用 ptrace(PTRACE_TRACEME, 0, 0, 0)允许自身被调试，然后通过检查 TracerPid 字段来确认是否有调试器正在跟踪。
如果发现调试器存在，会发送信号给父进程，父进程可以根据这个信息采取相应的行动。