



北京大学
PEKING UNIVERSITY

《When Top-down Meets Bottom-up: Detecting and Exploiting Use-After-Cleanup Bugs in Linux Kernel》

2025.01.16 江昊





UAC (Use-After-Cleanup)

Use-After-Cleanup: UAC基本原理类似UAF，和系统中特定的**设备 (device)** 的**卸载**（例如一个USB设备被用户拔出）相关，当一个特定的**设备释放**后，原来和这个**设备相关的内存对象应该就不再有效**。如下图

(1) 攻击者在**设备释放前**就开始**启动**相关内存对象访问①，**通过一些关键性的标志检查**

(2) 在**设备释放后**②（相关内存对象也不再有效）**再执行内存访问**③，就会产生和UAF漏洞攻击类似的效果。

漏洞根本原因：内核没有正确实现同步机制，所以syscall路径没有意识到对象已经被释放

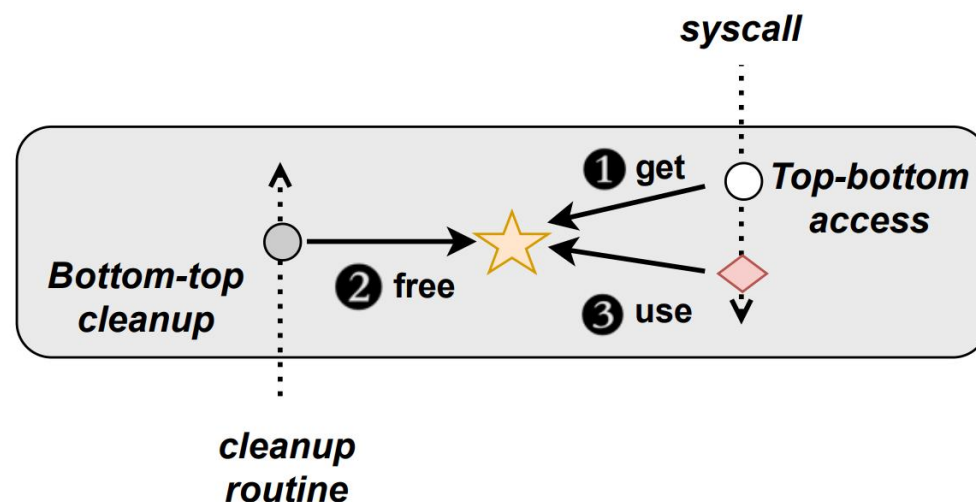


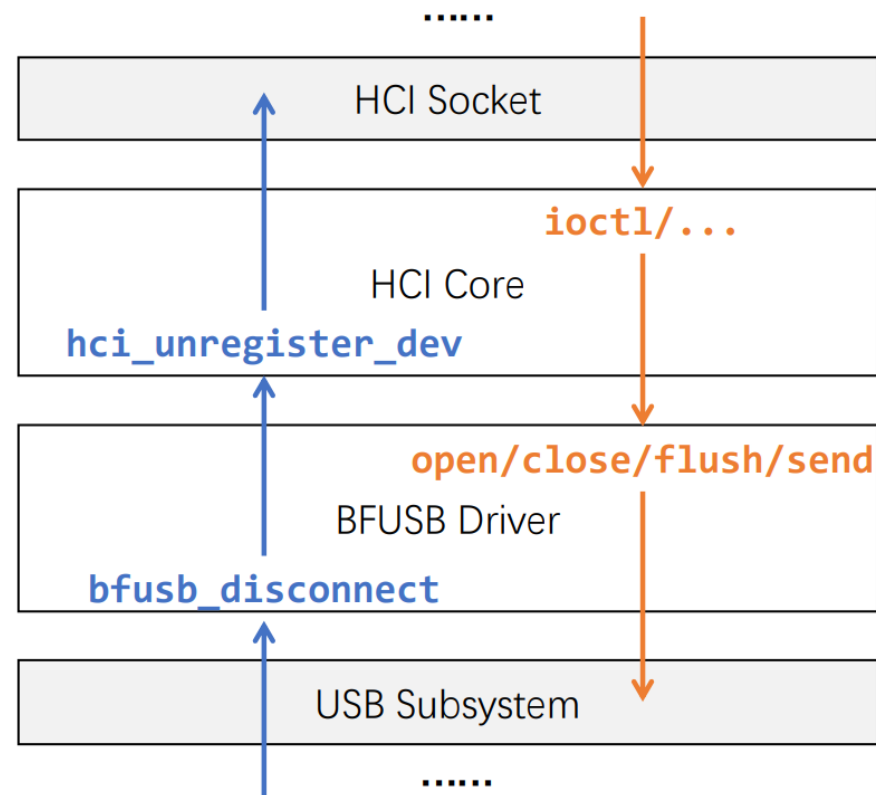
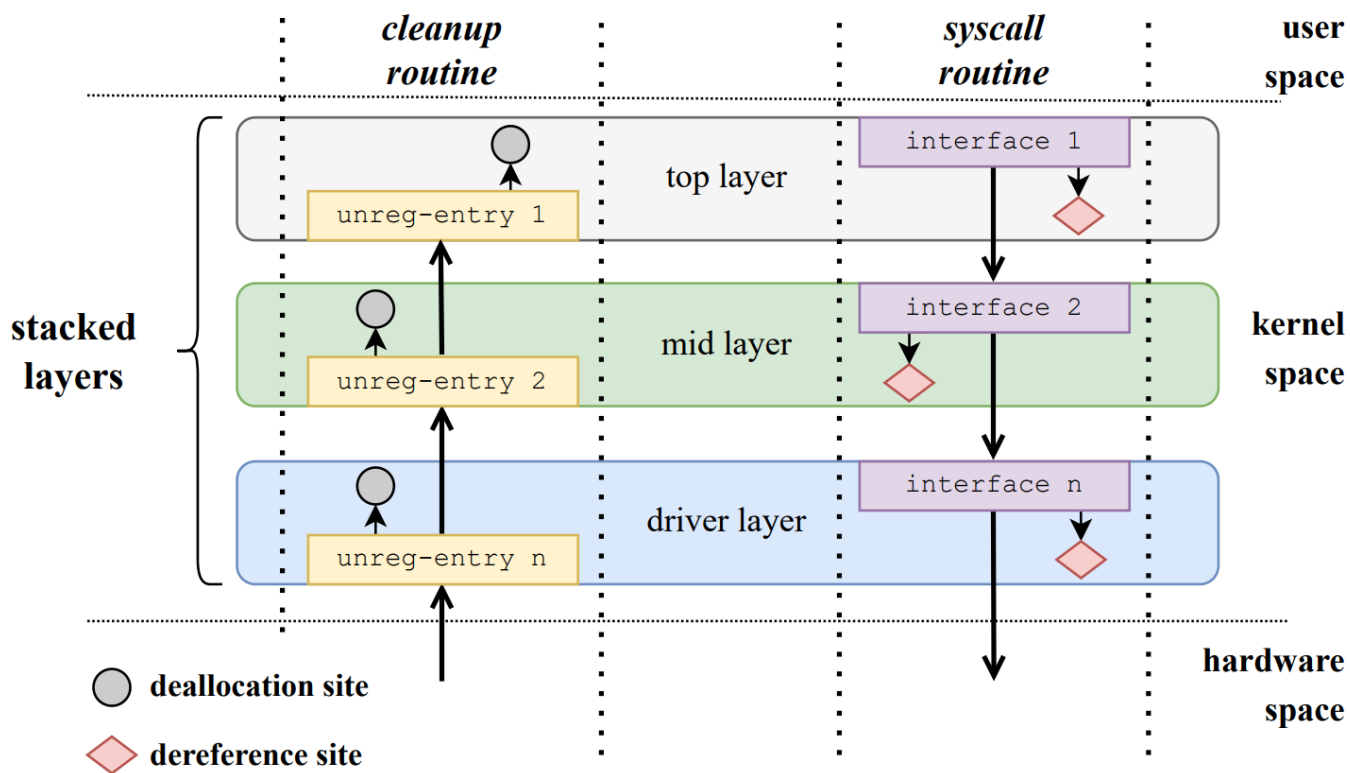
Fig. 1: The root cause of the UAC bug. The kernel object is freed by the bottom-top cleanup thread (②) and used by the top-bottom access thread (③).



UAC (Use-After-Cleanup)

碰撞层次模型：

- (1) 设备的卸载是从硬件层自底向上通知直至用户层
- (2) 用户态代码对设备资源的访问则需要透过syscall从上往下访问
- (3) 这两类（并发）事件如果撞到一起，就很容易产生并发bug，从而导致UAC相关问题的发生





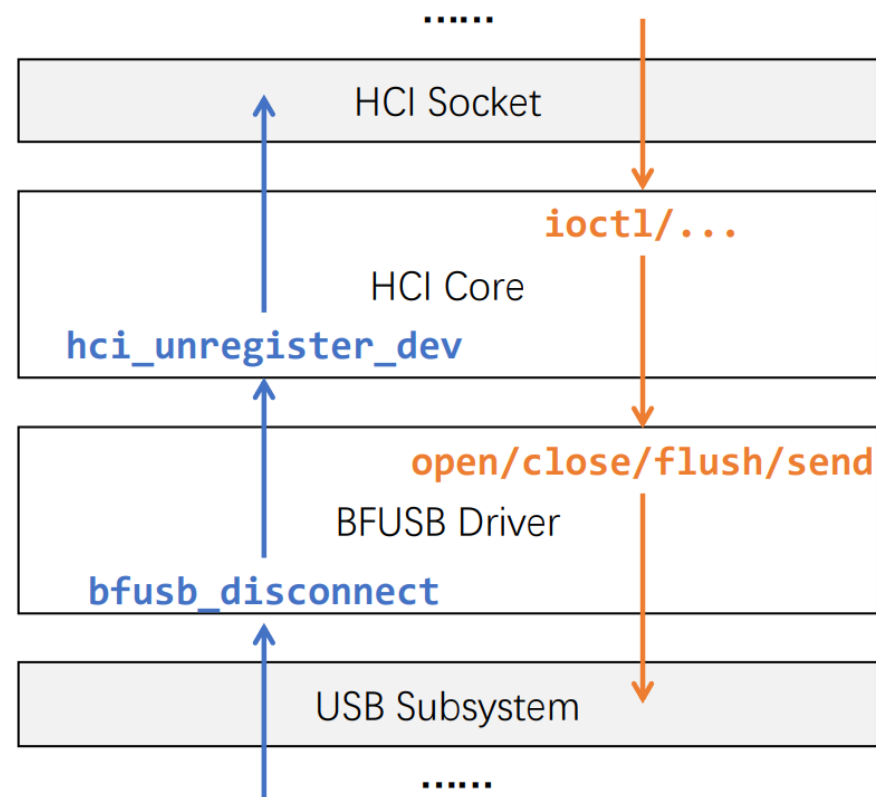
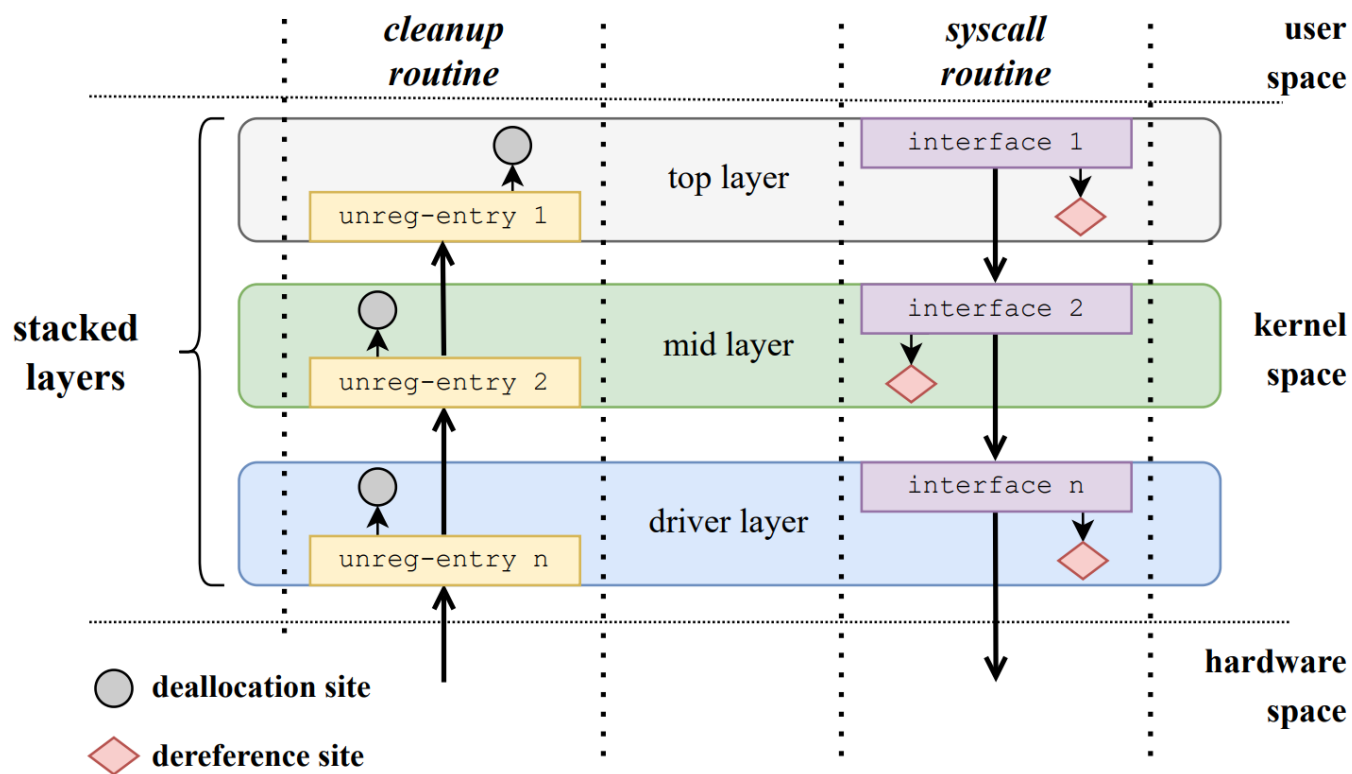
UAC (Use-After-Cleanup)

deallocation site: bottom-up cleanup routine 释放内核对象的地方

dereference site: top-down syscall routine 使用该内核对象的地方

dPair: 针对同一内核对象的一对 deallocation site 和 dereference site

layer-boundary functions: 包含 unreg-entry 函数（cleanup例程的入口函数）和 interface 函数（syscall例程的入口函数）





UAC (Use-After-Cleanup)

(1) hci_unregister_dev()函数就是HCI层的 **unreg-entry**

(2) hci_sock_sendmsg()函数是HCI层的 **interface**

(3) 当蓝牙设备卸载时，会利用hci_sock_dev_event()通知所有socket来回收目标对象hdev->workqueue，在3899行调用destroy_workqueue()释放目标对象；

(4) hci_sock_sendmsg()（通过调用 sendmsg 触发）在1829行会用到该对象。

(5) 由于没有控制对hdev的并发访问，因此构成了UAC漏洞

FILE: linux-5.11/net/Bluetooth/hci_core.c

```
3852. void hci_unregister_dev(...) {
.....
3886.  hci_sock_dev_event(hdev, ...); // ①
.....
3899.  destroy_workqueue(hdev->workqueue); // ② deallocation
3900.  destroy_workqueue(hdev->req_workqueue);
.....
3922. }
```

FILE: linux-5.11/net/Bluetooth/hci_sock.c

```
734. void hci_sock_dev_event(...) {
.....
767.  hci_pi(sk)->hdev = NULL;
.....
778. }

-----

1701. static int hci_sock_sendmsg(...) {
.....
1746.  hdev = hci_pi(sk)->hdev; // ③
.....
1829.  queue_work(hdev->workqueue, ...); // ④ dereference
.....
1841. }
```

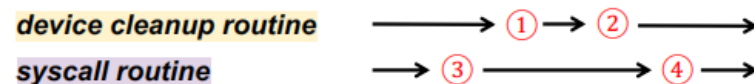


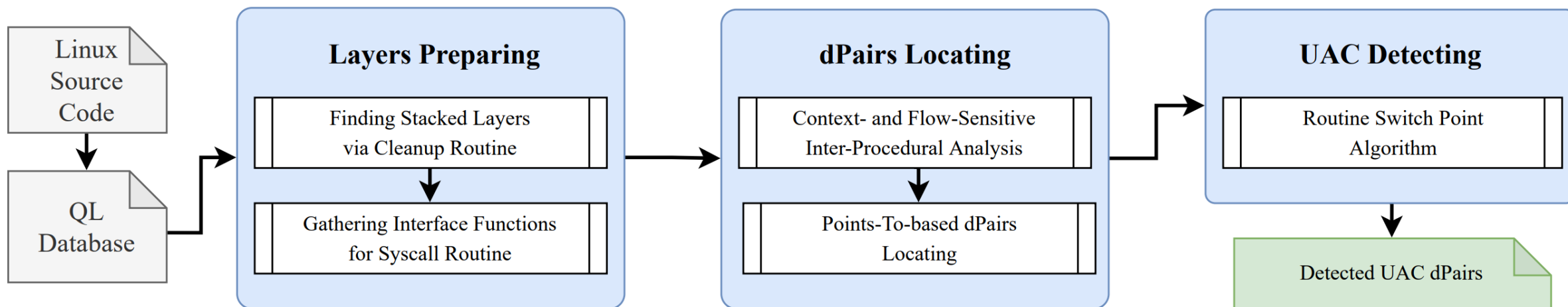
Fig. 4: A reported UAC bug in the Bluetooth stack and the routine interleaving sequence for triggering it.



UACatcher

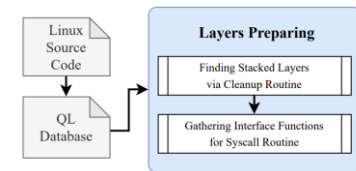
UACatcher工作机制:

- (1) 首先, UACatcher要借助Linux内核驱动的一些知识帮忙, 分析和收集那些与设备相关的代码 (下图中的Layer Preparing), 提取出所有的unreg-entry函数和Interface函数
- (2) 然后, UACatcher会寻找那些和特定设备相关的内存对象, 并确定一个对象的 deallocation site 和 dereference site (dPair Locating)
- (3) 最后就是利用上下文切换算法来确认某个dPair是否会导致UAC行为的发生。





Finding Stacked Layers via Cleanup Routine



UACatcher查找从底层到顶层堆叠在一起的设备层。设备层中的层通过unreg-entry函数和interface函数相互连接。

(1) 为了在内核中找到设备层和相应的unreg-entry，首先找到底层驱动层，然后找到上层。

①UACatcher通过检查驱动结构是否嵌入device_driver作为成员，来扫描整个内核，以找到所有派生的驱动类型

②手动标记所有找到的驱动结构中负责设备移除的指针字段（例如usb_driver结构中的disconnect字段）作为该层的unreg-entry。

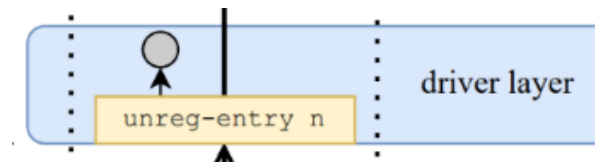
③基于预定义的规则扫描unreg-entry会调用的上层潜在的unreg-entry（规则见后）

(2) 当定位了该层的unreg-entry后，UACatcher通过解析KBuild和Makefile构建该层的层次指纹（层次指纹结构见后）

```
struct device_driver {
    ....
    // Called when the device is removed
    // from the system to unbind a device
    // from its driver
    int (*remove) (struct device *dev);
    ....
}

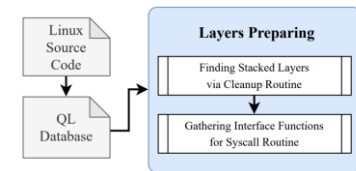
struct usbdrv_wrap {
    struct device_driver driver;
    ....
}

struct usb_driver {
    ....
    // Called when the interface is no
    // longer accessible, usually because
    // its device has been (or is being)
    // disconnected or the driver module
    // is being unloaded.
    void (*disconnect) (...);
    ....
    // embed device_driver
    struct usbdrv_wrap drvwrap;
    ....
}
```





Finding Stacked Layers via Cleanup Routine



UACatcher——Layers Preparing——Finding Stacked Layers via Cleanup Routine

为了扫描底层unreg-entry的上层unreg-entry，UACatcher基于以下原则进行匹配

- (1) 上层的unreg-entry函数必须从下层调用一次，因为清理例程从底层到顶层遍历
- (2) unreg-entry函数具有以下特征：

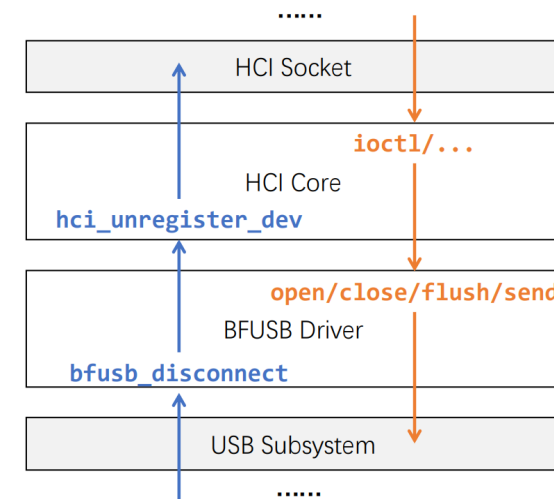
- ①类型特征：unreg-entry函数是void（不返回值）函数。
- ②参数特征：unreg-entry函数只有一个指针参数。
- ③名称特征：unreg-entry函数的名称包含隐含的关键字，例如unregister。

```
3  Function directCallParent(Function dst) {
4      result = dst.getACallToThisFunction().getEnclosingFunction()
5  }
6
7  from Function unreg, FunctionCall upper_unreg_call, Function upper_unreg
8  where
9      unreg.getName().matches("{funcname}") and
10     unreg.getFile().getRelativePath().matches("{funcfile}") and
11     not upper_unreg.isDefined() and
12     upper_unreg.getNumberOfParameters() = 1 and
13     upper_unreg.getType().toString().matches("void") and
14     upper_unreg.getName().matches(["%unregister%", "%deregister%"]) and
15     upper_unreg_call.getTarget() = upper_unreg and
16     unreg = directCallParent*(upper_unreg)
17  select upper_unreg.getName()
```

```
struct device_driver {
    ....
    // Called when the device is removed
    // from the system to unbind a device
    // from its driver
    int (*remove) (struct device *dev);
    ....
}

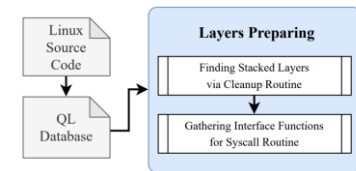
struct usbdrv_wrap {
    struct device_driver driver;
    ....
}

struct usb_driver {
    ....
    // Called when the interface is no
    // longer accessible, usually because
    // its device has been (or is being)
    // disconnected or the driver module
    // is being unloaded.
    void (*disconnect) (...);
    ....
    // embed device_driver
    struct usbdrv_wrap drvwrap;
    ....
}
```



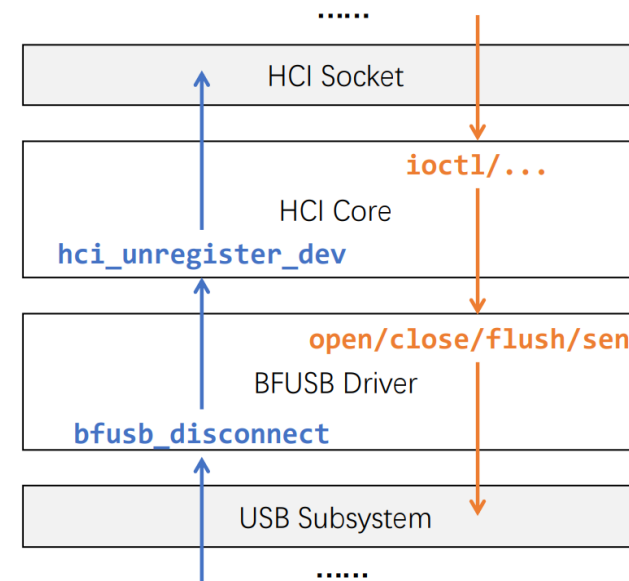


Finding Stacked Layers via Cleanup Routine



为了建立设备层的层次关系，UACatcher会扫描KBuild和MakeFile，提取出keyword， enclosedir， relative_path以生成一个唯一的层次描述符

- ① keyword: 编译选项中的CONFIG宏
- ② enclosedir: MakeFile所在目录的目录名
- ③ relative_path: 相对路径

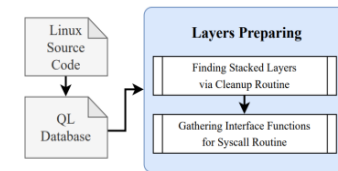


```
437 def _get_layer_name(enclosedir: str, keyword: str, relative_path: str) -> str:
438     return "layer_{}_{_}".format(
439         *args: enclosedir,
440         hashlib.md5(keyword.encode()).hexdigest()[:8],
441         hashlib.md5(relative_path.encode()).hexdigest()[:8])
```

```
obj-$(CONFIG_BT_HCIVHCI) += hci_vhci.o
obj-$(CONFIG_BT_HCIUART) += hci_uart.o
obj-$(CONFIG_BT_HCIBCM203X) += bcm203x.o
obj-$(CONFIG_BT_HCIBPA10X) += bpa10x.o
obj-$(CONFIG_BT_HCIBFUSB) += bfusb.o
obj-$(CONFIG_BT_HCIDTL1) += dtl1_cs.o
obj-$(CONFIG_BT_HCIBT3C) += bt3c_cs.o
obj-$(CONFIG_BT_HCIBLUECARD) += bluecard_cs.o
```



Gathering Interface Functions for Syscall Routine



根据分层模型，接口函数由下层暴露给上层以承担顶层系统调用例程。UACatcher通过两个步骤收集这些函数

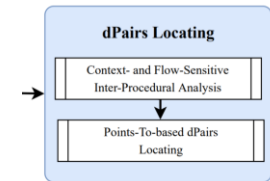
(1) Interface函数总是通过**函数指针调用**，并总是**打包在特定的（静态、全局）结构**中。例如，网络层的所有接口函数都打包在proto_ops结构中。然而，由于这些结构太多且没有统一的模式，因此，UACatcher查找所有可能的结构并提取其初始化的函数指针成员，以避免漏报。

(2) 为了减少误报，UACatcher通过**确认上层是否访问这些函数来过滤找到的层的Interface函数**。具体来说，对于从成员A中提取的接口函数，其结构体类型为B，UACatcher遍历上层的所有调用点，查找是否有间接调用解引用结构指针类型为B并访问成员A。

```
2827 const struct file_operations usbdev_file_operations = {
2828     .owner = THIS_MODULE,
2829     .llseek = no_seek_end_llseek,
2830     .read = usbdev_read,
2831     .poll = usbdev_poll,
2832     .unlocked_ioctl = usbdev_ioctl,
2833     .compat_ioctl = compat_ptr_ioctl,
2834     .mmap = usbdev_mmap,
2835     .open = usbdev_open,
2836     .release = usbdev_release,
2837 };
```



Context and Flow Sensitive Analysis



UACatcher采用上下文和流敏感的跨过程分析来实现准确的静态分析。

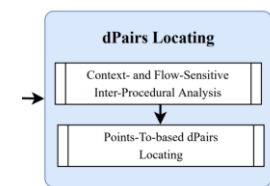
(1) 首先从输入层的边界函数开始遍历调用关系生成其（有向无环）调用图。

- ①用一个入口函数初始化分析栈，并循环分析直到栈为空，栈每次弹出一个待分析函数。
- ②每个待分析函数会被分解为基本块并构建CFG图，进而以流敏感的方式确定被分析函数中的所有函数调用
- ③如果被调用的函数定义在其他层中，它将被视为外部符号，循环继续。
- ④如果被调用的函数在本层，则调用点信息和传递的参数将被推入分析栈以供后续处理。

(2) UACatcher在整个遍历后输出一个调用图。该图的节点实际上是相应函数的控制流图，边由详细的调用点和参数组成。一旦构建了调用图，它将用于Point-to分析和UAC漏洞检测。



Points-To-based dPairs Locating



UACatcher采用Points-To analyse定位释放点和解引用点。

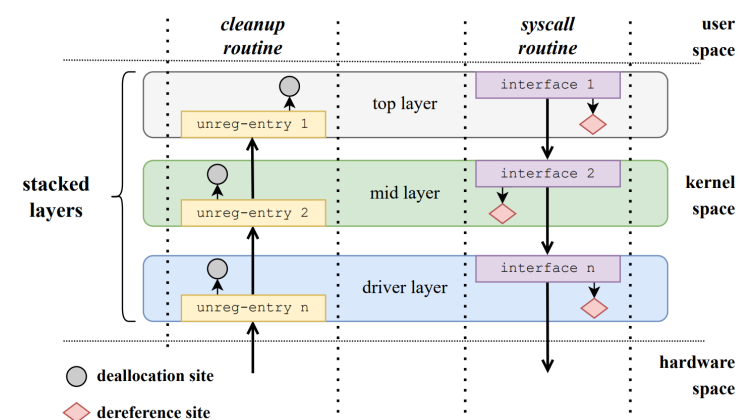
(1) 释放点：从unreg-entry函数生成调用图时，UACatcher通过跟踪到释放函数的调用点来定位释放点。为此，UACatcher收集了先前工作中披露的最常用的释放函数，并且总结了一些专用对象释放函数的特征。

① 专用对象释放函数通常只接收一个特定指针作为参数，因此可使用基于签名的分析找到可能候选者

② 大多数释放函数包含关键词，例如destroy、free、release等，便于通过正则表达式筛选

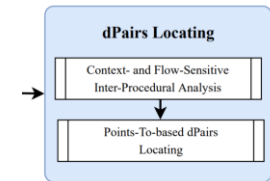
(2) 解引用点：UACatcher利用字段敏感的指向分析来定位相关的解引用点。除了所有指针解引用表达式，如果指针作为参数传递给外部函数，我们也将其视为解引用操作。在分析过程中，UACatcher遍历层中的所有解引用操作，并检查该操作（无论是表达式还是参数）是否指向与先前定位的释放点相同的位置。如果是，UACatcher将此操作标记为相应释放点的解引用点。

```
10 class KStandardDeallocationFunction extends DeallocationFunction {
11     int freedArg;
12
13     Quick Evaluation: KStandardDeallocationFunction
14     KStandardDeallocationFunction() {
15         hasName([
16             "kfree", // kfree(const void *);
17             "kfree_sensitive", // kfree_sensitive(const void *)
18             "kfree_const", // void kfree_const(const void *x)
19             "kvfree" // void kvfree(const void *addr)
20             // "kfree_rcu" macro here
21         ]) and
22         freedArg = 0
23         or
24         hasName([
25             "kmem_cache_free" // kmem_cache_free(struct kmem_cache *, void *)
26         ]) and
27         freedArg = 1
28         or
29         // export function need to be modeled
30         hasName([
31             "kfree_skb",
32             "destroy_workqueue",
33             "rkill_destroy",
34             "crypto_free_shash"
35             // TODO: more?
36         ]) and
37         freedArg = 0
38     }
```





Points-To-based dPairs Locating



UACatcher采用Points-To analyse定位释放点和解引用点。

(3) 路径：对于所有定位的释放点及其相应的解引用点，UACatcher构建从边界函数开始到达这些点的所有简单路径。路径的生成还考虑了控制流图。如果有多个调用点允许父函数调用子函数，UACatcher将记录所有可能的基本块路径，以保证分析结果的完备性。

(4) 为了减少错误结果，UACatcher添加了几个过滤器

①指向分析被设置了阈值。如果指向分析未能准确定位指向集（在内核分析中很常见）并获得低置信度，该站点会被丢弃

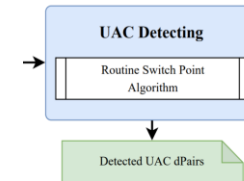
②过滤器检查定位的释放点是否与分配点在同一函数内。如果是，则此释放点会被丢弃，因为它很可能充当错误处理，与UAC漏洞关系不大。

③对于每对释放路径和解引用路径，过滤器收集并检查它们的公共段集。如果两条路径有公共段，我们将丢弃这对路径，因为这些路径可能实际上不会并发执行。

④如果经过上述操作后，没有剩余路径能到达dPair，该dPair将被丢弃



Routine Switch Point Algorithm



真假UAC

(1) 左侧的路径导致真实的UAC漏洞，其目标对象是req_workqueue

(2) 右侧路径没漏洞，因为test_bit在锁里面

上下文切换点：对于真UAC，只要巧妙的控制上下文切换，就能出发漏洞

① syscall routine先过了test_bit，停在hci_req_sync_lock（上下文切换点）

② device cleanup routine进入hci_req_sync_lock，clear bit，放锁，然后释放req_workqueue（上下文切换点）

③ syscall routine拿锁，解引用req_workqueue

结论：所以检测锁和约束对验证UAC真实性起到决定性作用

device cleanup routine (BT top)
..... 1. hci_req_sync_lock(hdev) 2. test_and_clear_bit(HCI_UP, &hdev->flags) 3. hci_req_sync_unlock(hdev) 4. destroy_workqueue(hdev->req_workqueue)
syscall routine (BT top)
..... 5. test_bit(HCI_UP, &hdev->flags) 6. hci_req_sync_lock(hdev) 7. queue_delayed_work(hdev->req_workqueue, ...)

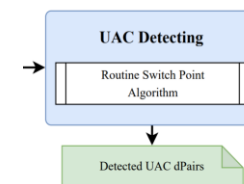
✓ true UAC bug

device cleanup routine (NFC nci)
..... 1. mutex_lock(&ndev->req_lock) 2. test_and_clear_bit(NCI_UP, &ndev->flags) 3. mutex_unlock(&ndev->req_lock) 4. destroy_workqueue(ndev->cmd_wq)
syscall routine (NFC nci)
..... 5. mutex_lock(&ndev->req_lock) 6. test_bit(NCI_UP, &ndev->flags) 7. queue_work(ndev->cmd_wq, ...)

✗ false UAC bug



Routine Switch Point Algorithm



(1) 算法相关宏

- ①HHoldLock: 获取输入位置的历史持有锁
- ②CHoldLock: 获取输入位置的当前持有锁, 或锁集
- ③Pred获取输入路径中输入位置的前一个位置
- ④Succ获取输入路径中输入位置的下一个位置

(2) 算法输入

- ①dPair的释放路径Path1
- ②dPair解引用路径Path2

(3) 算法输出

- ①如果此函数返回True, 则认为dPair导致UAC漏洞
- ②输出上下文切换点P

Algorithm 1: Routine Switch Point Algorithm

Input : deallocation path $Path_1$, dereference path $Path_2$

Output: context switch points P

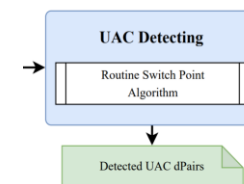
```

1 Function UACDetect ()
2    $S_1 = \text{End}(Path_1), S_2 = \text{End}(Path_2)$ 
3    $r = S_2, PC1 = \emptyset, PC2 = \emptyset$ 
4    $L_{inter} = \text{InterLockA}(S_1, r)$ 
5   while  $L_{inter} \neq \emptyset$  do
6      $r = \text{Pred}(r, Path_2)$ 
7     if  $\text{Action}(r) \in \text{ConstraintCheck}$  then
8        $PC1 = PC1 \cup \{\text{Action}(r)\}$ 
9      $L_{inter} = \text{InterLockA}(S_1, r)$ 
10   $P = P \cup \{r\}$ 
11   $r = \text{Start}(Path_1)$ 
12   $L_{inter} = \text{InterLockB}(S_2, r)$ 
13  while  $L_{inter} \neq \emptyset$  do
14     $r = \text{Succ}(r, Path_1)$ 
15    if  $\text{Action}(r) \in \text{ConstraintChange}$  then
16       $PC2 = PC2 \cup \{\text{Action}(r)\}$ 
17     $L_{inter} = \text{InterLockB}(S_2, r)$ 
18   $P = P \cup \{r\}$ 
19  if  $\text{Satisfy}(PC1, PC2)$  then
20    return True
21  return False
22 Function InterLockA( $s1, s2$ )
23   return  $\text{HHoldLock}(s1) \cap \text{CHoldLock}(s2)$ 
24 Function InterLockB( $s1, s2$ )
25   return  $\text{CHoldLock}(s1) \cap \text{CHoldLock}(s2)$ 
  
```

cleanup routine:	syscall routine:
⑥ 1. lock(l1);	9. CONS1-CHECK
2. kfree(p); ①	10. lock(l1); ← ⑤
3. CONS1-CHANGE	11. *p = 1; ③
4. unlock(l1);	12. lock(l2);
⑦ 5. lock(l2);	13. CONS2-CHECK
6. CONS2-CHANGE	14. *q = 2; ④
7. kfree(q); ②	15. unlock(l2);
⑧ 8. unlock(l2);	16. unlock(l1);



Routine Switch Point Algorithm



算法流程(伪代码感觉有问题):

(1) 释放点①, 解引用点③

1. 从③往前, 走到⑤, 所持锁释放l1, 锁集空。这是switch-1。同时记录一下 joint-lock为l1

2. 从调用路径最开始的位置开始扫描, 即为⑥。

3. 一直往后走到⑦, 中途记录约束CONS1-CHANGE, l1释放。这里是 switch-2

4. cleanup约束为CONS1-CHANGE, syscall约束为空。两者不对应, 所以为 UAC

Algorithm 1: Routine Switch Point Algorithm

Input : deallocation path $Path_1$, dereference path $Path_2$

Output: context switch points P

```

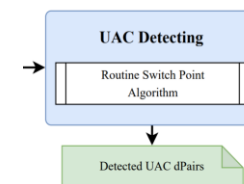
1 Function UACDetect ()
2    $S_1 = \text{End}(Path_1), S_2 = \text{End}(Path_2)$ 
3    $r = S_2, PC1 = \emptyset, PC2 = \emptyset$ 
4    $L_{inter} = \text{InterLockA}(S_1, r)$ 
5   while  $L_{inter} \neq \emptyset$  do
6      $r = \text{Pred}(r, Path_2)$ 
7     if  $\text{Action}(r) \in \text{ConstraintCheck}$  then
8        $PC1 = PC1 \cup \{\text{Action}(r)\}$ 
9      $L_{inter} = \text{InterLockA}(S_1, r)$ 
10   $P = P \cup \{r\}$ 
11   $r = \text{Start}(Path_1)$ 
12   $L_{inter} = \text{InterLockB}(S_2, r)$ 
13  while  $L_{inter} \neq \emptyset$  do
14     $r = \text{Succ}(r, Path_1)$ 
15    if  $\text{Action}(r) \in \text{ConstraintChange}$  then
16       $PC2 = PC2 \cup \{\text{Action}(r)\}$ 
17     $L_{inter} = \text{InterLockB}(S_2, r)$ 
18   $P = P \cup \{r\}$ 
19  if  $\text{Satisfy}(PC1, PC2)$  then
20    return True
21  return False
22 Function InterLockA( $s1, s2$ )
23   return  $\text{HHoldLock}(s1) \cap \text{CHoldLock}(s2)$ 
24 Function InterLockB( $s1, s2$ )
25   return  $\text{CHoldLock}(s1) \cap \text{CHoldLock}(s2)$ 

```

cleanup routine:	syscall routine:
⑥ → 1. lock(l1);	9. CONS1-CHECK
2. kfree(p); ①	10. lock(l1); ← ⑤
3. CONS1-CHANGE	11. *p = 1; ③
⑦ → 4. unlock(l1);	12. lock(l2);
5. lock(l2);	13. CONS2-CHECK
6. CONS2-CHANGE	14. *q = 2; ④
7. kfree(q); ②	15. unlock(l2);
⑧ → 8. unlock(l2);	16. unlock(l1);



Routine Switch Point Algorithm



算法流程(伪代码感觉有问题):

(2) 释放点②, 解引用点④ (感觉这里算法说不通)

1. 从④往前, 走到⑤, 所持锁释放l1, l2, 锁集空, 这是switch-1。同时记录一下joint-lock为l1, l2, 路过一个约束CONS2-CHECK
2. 从调用路径最开始的位置开始扫描, 即为⑥。
3. 一直往后走到⑧, 中途记录约束CONS1-CHANGE, CONS2-CHANGE, l1和l2释放。这里是switch-2
4. cleanup约束为CONS1-CHANGE, CONS2-CHANGE, syscall约束为CONS2-CHECK。两者存在对应, 所以不为UAC

Algorithm 1: Routine Switch Point Algorithm

Input : deallocation path $Path_1$, dereference path $Path_2$

Output: context switch points P

```

1 Function UACDetect ()
2    $S_1 = \text{End}(Path_1), S_2 = \text{End}(Path_2)$ 
3    $r = S_2, PC1 = \emptyset, PC2 = \emptyset$ 
4    $L_{inter} = \text{InterLockA}(S_1, r)$ 
5   while  $L_{inter} \neq \emptyset$  do
6      $r = \text{Pred}(r, Path_2)$ 
7     if  $\text{Action}(r) \in \text{ConstraintCheck}$  then
8        $PC1 = PC1 \cup \{\text{Action}(r)\}$ 
9      $L_{inter} = \text{InterLockA}(S_1, r)$ 
10   $P = P \cup \{r\}$ 
11   $r = \text{Start}(Path_1)$ 
12   $L_{inter} = \text{InterLockB}(S_2, r)$ 
13  while  $L_{inter} \neq \emptyset$  do
14     $r = \text{Succ}(r, Path_1)$ 
15    if  $\text{Action}(r) \in \text{ConstraintChange}$  then
16       $PC2 = PC2 \cup \{\text{Action}(r)\}$ 
17     $L_{inter} = \text{InterLockB}(S_2, r)$ 
18   $P = P \cup \{r\}$ 
19  if Satisfy  $(PC1, PC2)$  then
20    return True
21  return False
22 Function InterLockA( $s1, s2$ )
23   return  $\text{HHoldLock}(s1) \cap \text{CHoldLock}(s2)$ 
24 Function InterLockB( $s1, s2$ )
25   return  $\text{CHoldLock}(s1) \cap \text{CHoldLock}(s2)$ 
  
```

问题: 伪代码中认为只要 L_{inter} 为空就停止循环。但这样的话第2步一开始应该就停了
根据实现代码, 这里应该是如果joint-lock全部被获取并释放了的才停

cleanup routine:	syscall routine:
⑥ → 1. lock(l1);	9. CONS1-CHECK
2. kfree(p); ①	10. lock(l1); ← ⑤
3. CONS1-CHANGE	11. *p = 1; ③
⑦ → 4. unlock(l1);	12. lock(l2);
5. lock(l2);	13. CONS2-CHECK
6. CONS2-CHANGE	14. *q = 2; ④
7. kfree(q); ②	15. unlock(l2);
⑧ → 8. unlock(l2);	16. unlock(l1);



UACatcher——效果

```
linux-5.15.161 > drivers > infiniband > hw > hfi1 > C sdmac > _sdma_txclean(hfi1_devdata *s, sdma_breq *)
1631 /*
1632 */
1633 void __sdma_txclean(
1634     struct hfi1_devdata *dd,
1635     struct sdma_txreq *tx)
1636 {
1637     u16 i;
1638
1639     if (tx->num_desc > 0) {
1640         u8 skip = 0, mode = ahg_mode(tx);
1641
1642         /* unmap first */
1643         sdma_unmap_desc(dd, &tx->desc[0]);
1644         /* determine number of AHG descriptors to skip */
1645         if (mode > SDMA_AHG_APPLY_UPDATE1)
1646             skip = mode >> 1;
1647         for (i = 1 + skip; i < tx->num_desc; i++)
1648             sdma_unmap_desc(dd, &tx->desc[i]);
1649         tx->num_desc = 0;
1650     }
1651     kfree(tx->coalesce_buf);
1652     tx->coalesce_buf = NULL;
1653     /* kalloc'ed desc */
1654     if (unlikely(tx->desc_limit > ARRAY_SIZE(tx->descs))) {
1655         tx->desc_limit = ARRAY_SIZE(tx->descs);
1656         kfree(tx->descs); // dealloc location
1657     }
1658 }
1659
1660 static inline u16 sdma_gethead(struct sdma_engine *sde)
1661 {
1662     struct hfi1_devdata *dd = sde->dd;
1663 }

linux-5.15.161 > drivers > infiniband > hw > hfi1 > C sdmac > submit_tx(sdma_engine *s, sdma_breq *)
2264 /*
2265 static inline u16 submit_tx(struct sdma_engine *sde, struct sdma_txreq *tx)
2266 {
2267     int i;
2268     u16 tail;
2269     struct sdma_desc *descp = tx->descp;
2270     u8 skip = 0, mode = ahg_mode(tx);
2271
2272     tail = sde->descq_tail & sde->sdma_mask;
2273     sde->descq[tail].qw[0] = cpu_to_le64(descp->qw[0]); // deref location
2274     sde->descq[tail].qw[1] = cpu_to_le64(add_gen(sde, descp->qw[1]));
2275     trace_hfi1_sdma_descriptor(sde, descp->qw[0], descp->qw[1],
2276                               tail, &sde->descq[tail]);
2277     tail = ++sde->descq_tail & sde->sdma_mask;
2278     descp++;
2279     if (mode > SDMA_AHG_APPLY_UPDATE1)
2280         skip = mode >> 1;
2281     for (i = 1 + skip; i < tx->num_desc; i++, descp++) {
2282         u64 qw1;
2283
2284         sde->descq[tail].qw[0] = cpu_to_le64(descp->qw[0]);
2285         if (skip) {
2286             /* edits don't have generation */
2287             qw1 = descp->qw[1];
2288             skip--;
2289         } else {
2290             /* replace generation with real one for non-edits */
2291             qw1 = add_gen(sde, descp->qw[1]);
2292         }
2293         sde->descq[tail].qw[1] = cpu_to_le64(qw1);
2294         trace_hfi1_sdma_descriptor(sde, descp->qw[0], qw1,
2295                                   tail, &sde->descq[tail]);
2296     }
2297 }

linux-5.15.161 > drivers > infiniband > hw > hfi1 > C sdmac > sdma_process_event(sdma_engine *s, sdma_events event)
2498 int sdma_send_txlist(struct sdma_engine *sde, struct iowait_work *wait,
2499                    sde->descq_full_count++;
2500                    goto update_tail;
2501 }
2502 static void sdma_process_event(struct sdma_engine *sde, enum sdma_events event)
2503 {
2504     unsigned long flags;
2505     spin_lock_irqsave(&sde->tail_lock, flags);
2506     write_seqlock(&sde->head_lock);
2507     __sdma_process_event(sde, event);
2508     if (sde->state.current_state == sdma_state_s9g_running)
2509         sdma_desc_avail(sde, sdma_descq_freecnt(sde));
2510     write_sequnlock(&sde->head_lock);
2511     spin_unlock_irqrestore(&sde->tail_lock, flags); // switchB
2512 }
2513 static void __sdma_process_event(struct sdma_engine *sde,
2514                                enum sdma_events event)
2515 {
2516     struct sdma_state *ss = &sde->state;
2517     int need_progress = 0;
2518
2519     /* CONFIG SDMA temporary */
2520     #ifdef CONFIG_SDMA_VERBOSEITY
2521     dd_dev_err(sde->dd, "CONFIG SDMA(%u) [%s] %s\n", sde->this_id,
2522               sdma_state_names[ss->current_state],
2523               sdma_event_names[event]);
2524 }

linux-5.15.161 > drivers > infiniband > hw > hfi1 > C sdmac > sdma_send_txreq(sdma_engine *s, iowait_work *wait, sdma_txreq *)
2353 int sdma_send_txreq(struct sdma_engine *sde,
2354 {
2355     int ret = 0;
2356     u16 tail;
2357     unsigned long flags;
2358
2359     /* user should have supplied entire packet */
2360     if (unlikely(tx->tlen))
2361         return -EINVAL;
2362     tx->wait = iowait_iowm_to_iow(wait);
2363     spin_lock_irqsave(&sde->tail_lock, flags); // switchA
2364     retry:
2365     if (unlikely(!sdma_running(sde)))
2366         goto unlock_nocount;
2367     if (unlikely(tx->num_desc > sde->descq_avail))
2368         goto nodesc;
2369     tail = submit_tx(sde, tx);
2370     if (wait)
2371         iowait_sdma_inc(iowait_iowm_to_iow(wait));
2372     sdma_update_tail(sde, tail);
2373     unlock:
2374     spin_unlock_irqrestore(&sde->tail_lock, flags);
2375     return ret;
2376     unlock_nocount:
2377     if (wait)
2378         iowait_sdma_inc(iowait_iowm_to_iow(wait));
2379     tx->next_descq_idx = 0;
2380     #ifdef CONFIG_HFI1_DEBUG_SDMA_ORDER
2381     tx->sn = sde->tail_sn++;
2382     trace_hfi1_sdma_in_sn(sde, tx->sn);
2383     #endif
2384     spin_lock(&sde->flush_list_lock);
2385 }

report_routine-switch_18_sdma_txclean-kfree_18_555fa5c63bb6.json
8B87d7d0deaff > report_routine-switch_18_sdma_txclean-kfree_18_555fa5c63bb6.json > [ ] uac details > { } 9 > [ ] detected combinations > { } 0 > [ ] dealloc
6 "uac details": {
331 {
332     "deref location": "drivers/infiniband/hw/hfi1/sdma.c:2273:39",
333     "detected combination count": 1,
334     "detected combinations": [
335     {
336         "dealloc chain": [
337             "remove_one%drivers/infiniband/hw/hfi1/init.c",
338             "shutdown_device%drivers/infiniband/hw/hfi1/init.c",
339             "hfi1_quiet_serdes%drivers/infiniband/hw/hfi1/chip.c",
340             "set_link_state%drivers/infiniband/hw/hfi1/chip.c",
341             "handle_linkup_change%drivers/infiniband/hw/hfi1/intr.c",
342             "start_freeze_handling%drivers/infiniband/hw/hfi1/chip.c",
343             "sdma_freeze_notify%drivers/infiniband/hw/hfi1/sdma.c",
344             "sdma_process_event%drivers/infiniband/hw/hfi1/sdma.c",
345             "__sdma_process_event%drivers/infiniband/hw/hfi1/sdma.c",
346             "sdma_set_state%drivers/infiniband/hw/hfi1/sdma.c",
347             "sdma_flush%drivers/infiniband/hw/hfi1/sdma.c",
348             "complete_tx%drivers/infiniband/hw/hfi1/sdma.c",
349             "__sdma_txclean%drivers/infiniband/hw/hfi1/sdma.c"
350         ],
351         "deref chain": [
352             "tid_rdma_trigger_resume%drivers/infiniband/hw/hfi1/tid_rdma.c",
353             "hfi1_do_send%drivers/infiniband/hw/hfi1/ruc.c",
354             "hfi1_verbs_send%drivers/infiniband/hw/hfi1/verbs.c",
355             "hfi1_verbs_send_dma%drivers/infiniband/hw/hfi1/verbs.c",
356             "sdma_send_txreq%drivers/infiniband/hw/hfi1/sdma.c",
357             "submit_tx%drivers/infiniband/hw/hfi1/sdma.c"
358         ],
359         "joint locks": [
360             "drivers/infiniband/hw/hfi1/sdma.c:1398:3"
361         ],
362         "switchA": "drivers/infiniband/hw/hfi1/sdma.c:2366:2",
363         "switchB": "drivers/infiniband/hw/hfi1/sdma.c:2515:2",
364         "condition checks": {
365             "sdma_send_txreq%drivers/infiniband/hw/hfi1/sdma.c": [],
366             "submit_tx%drivers/infiniband/hw/hfi1/sdma.c": []
367         },
368         "condition writes": {
369             "remove_one%drivers/infiniband/hw/hfi1/init.c": [],
370             "shutdown_device%drivers/infiniband/hw/hfi1/init.c": [],
371             "hfi1_quiet_serdes%drivers/infiniband/hw/hfi1/chip.c": [
372                 {
373                     "struct": "hfi1_pportdata",
374                     "field": "link_enabled"
375                 },
376                 {
377                     "struct": "hfi1_pportdata",
378                     "field": "driver_link_ready"
379                 }
380             ],
381             "set_link_state%drivers/infiniband/hw/hfi1/chip.c": [],
382             "handle_linkup_change%drivers/infiniband/hw/hfi1/intr.c": [
383                 {
384                     "struct": "hfi1_pportdata",
385                     "field": "actual_vls_operational"
386                 },
387                 {
388                     "struct": "hfi1_pportdata",
389                     "field": "linkup"
390                 }
391             ],
392             "start_freeze_handling%drivers/infiniband/hw/hfi1/chip.c": [],
393             "sdma_freeze_notify%drivers/infiniband/hw/hfi1/sdma.c": [],
394             "sdma_process_event%drivers/infiniband/hw/hfi1/sdma.c": [
395                 {
396                     "struct": "sdma state",
397                     "field": "current_state"
398                 }
399             ]
400         }
401     }
402 ]
403 }
```




UACatcher——效果

```
Total files:190
Total UAC report count:17304
Total useful UAC report count (without condition write exists check):2011
Total useful UAC report count (with condition write exists check):682
Number of useful files: 9
Useful files:
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_bluetooth_98d561a8c05573c0\report_routine-switch_863_mgmt_cmd_complete-kfree_skb_888_1ea9e2590e8b.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_bluetooth_98d561a8c05573c0\report_routine-switch_856_mgmt_cmd_status-kfree_skb_913_dfd87ef03018.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_qib_cf427505e1d115a3\report_routine-switch_29_cleanup_device_data-kfree_29_6b1005d64410.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_bluetooth_98d561a8c05573c0\report_routine-switch_894_hci_sock_dev_event-kfree_skb_897_78d13c695cf8.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_hfi1_9a8bf7d7db0eeaff\report_routine-switch_18___sdma_txclean-kfree_18_555fa5c63bb6.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_wireless_2c0996f3a9cd2874\report_routine-switch_11___cfg80211_connect_result-kfree_sensitive_25_58de810396d5.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_qib_cf427505e1d115a3\report_routine-switch_416_qib_free_ctxtdata-kfree_416_87f3d79e4149.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_mISDN_e440ac5c2e401ef2\report_routine-switch_338_release_card-kfree_338_c2e6e44dadd3.json
C:\Users\kairosjiang\Desktop\研究学习\uacatcher result\p3output\layer_mISDN_23a0ea2b2e401ef2\report_routine-switch_21_release_card-kfree_21_ab0281129ab0.json
```