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

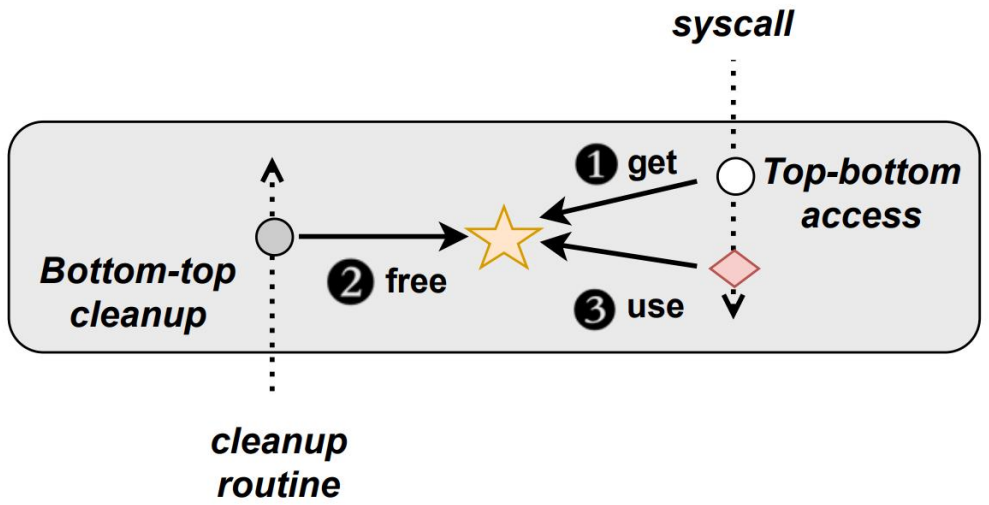


Fig. 1: The root cause of the UAC bug. The kernel object is freed by the bottom-top cleanup thread (❷) and used by the top-bottom access thread (❸).

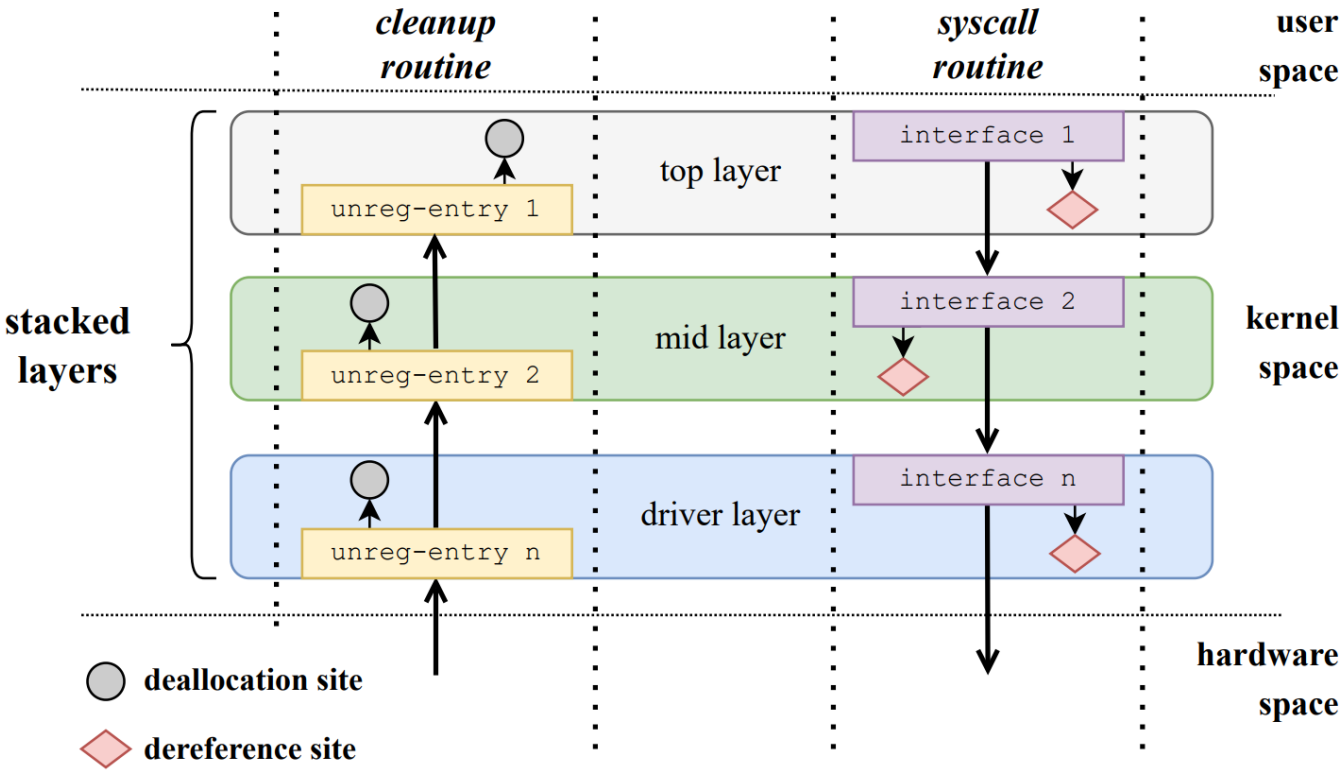
UAC (Use-After-Cleanup)

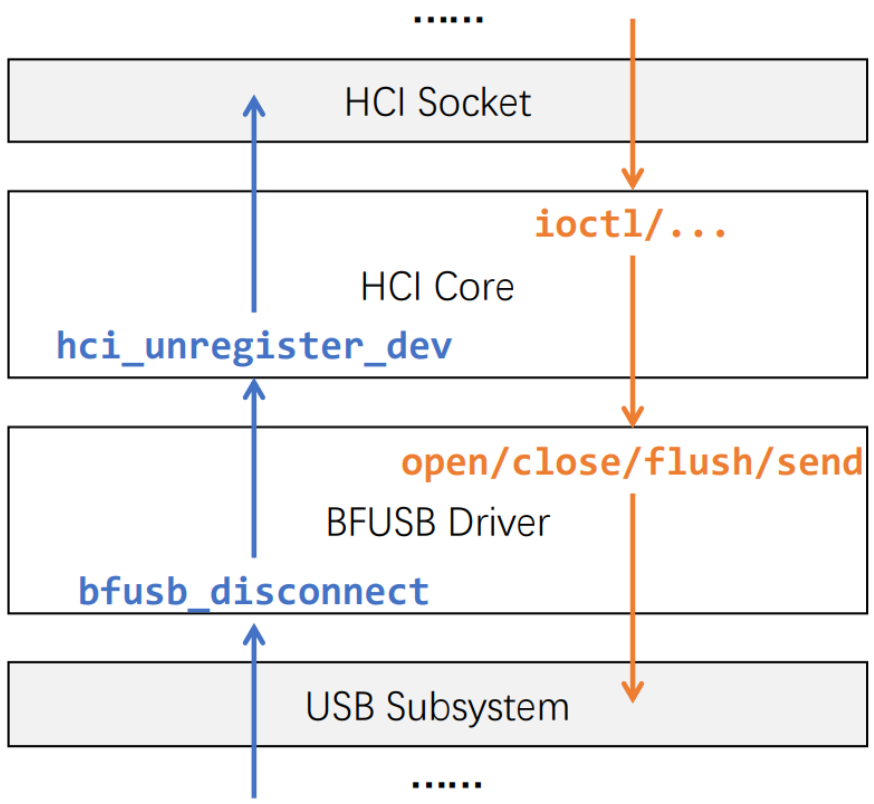
碰撞层次模型：

（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





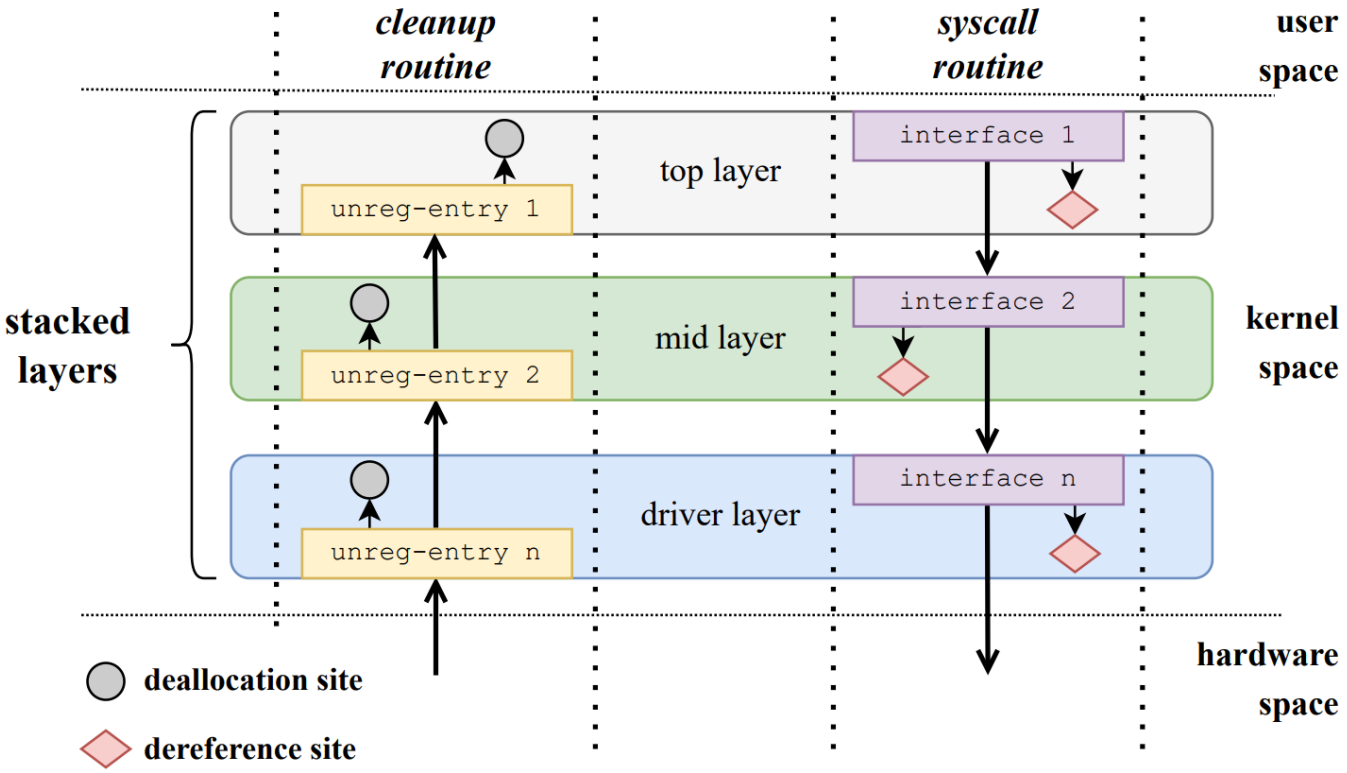
UAC (Use-After-Cleanup)

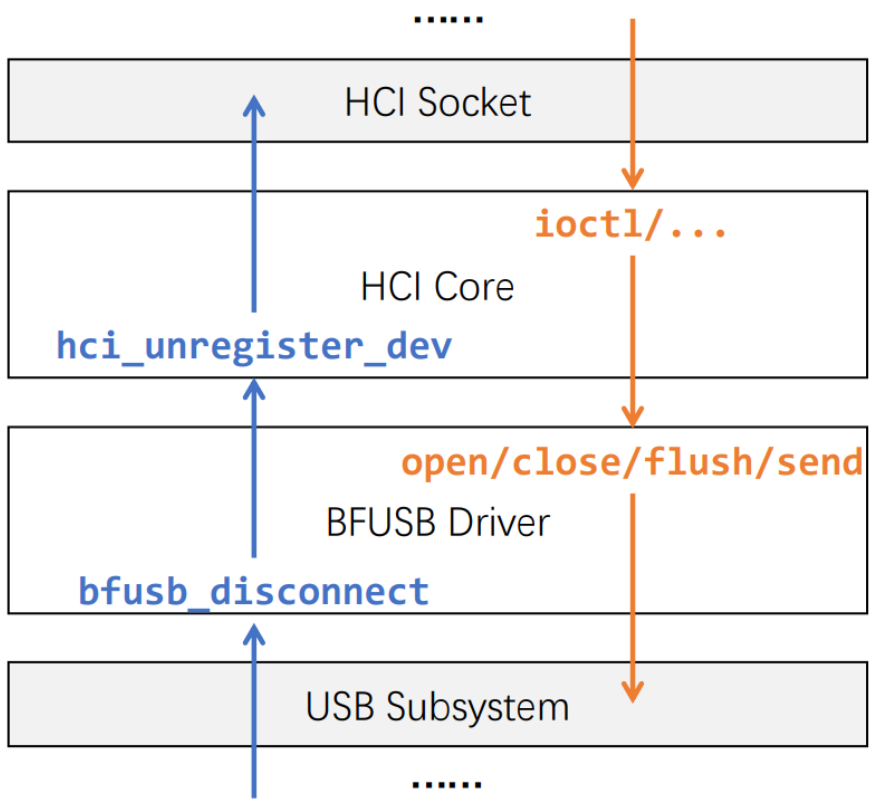
deallocation site：bottom-up cleanup routine 释放内核对象的地方

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);// ②deallocation
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. }
```

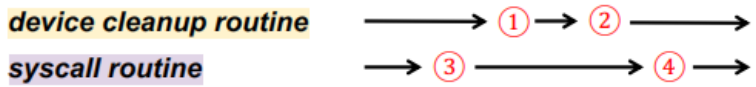


Fig. 4: A reported UAC bug in the Bluetooth stack and the routine interleaving sequence for triggering it.

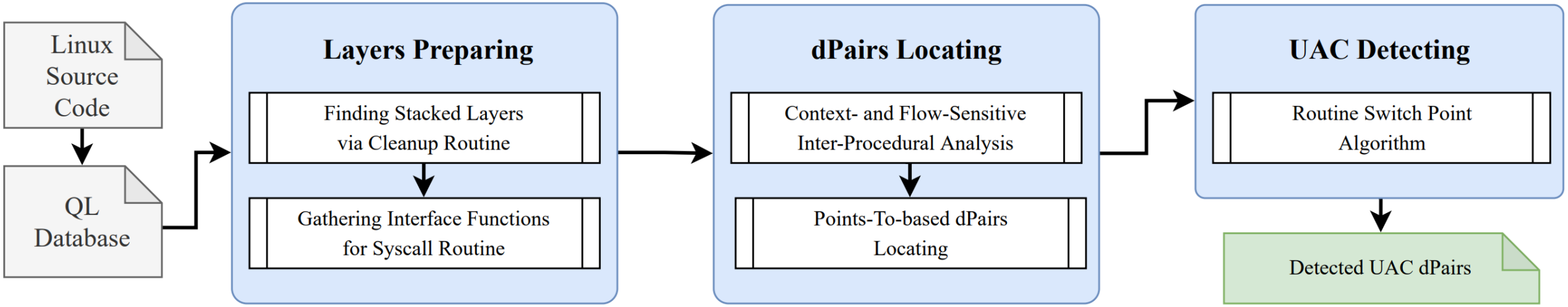
UACatcher

UACatcher工作机制：

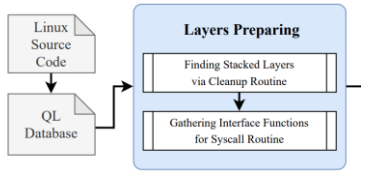
（1）首先，UACatcher要借助Linux内核驱动的一些知识帮忙，分析和收集那些与设备相关的代码（下图中的Layer Preparing），提取出所有的unreg-entry函数和Interface函数

（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。

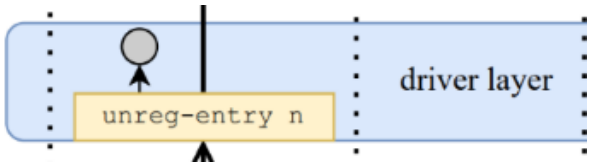
③基于预定义的规则扫描unreg-entry会调用的上层潜在的unreg-entry（规则见后）

（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）

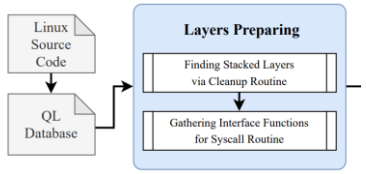
```
struct device_driver {
    ....
    // Called when the device is removed
    // from the system to unbind a device
    // from its driver
    int (*remove)  (struct device *dev);
    ....
}

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。

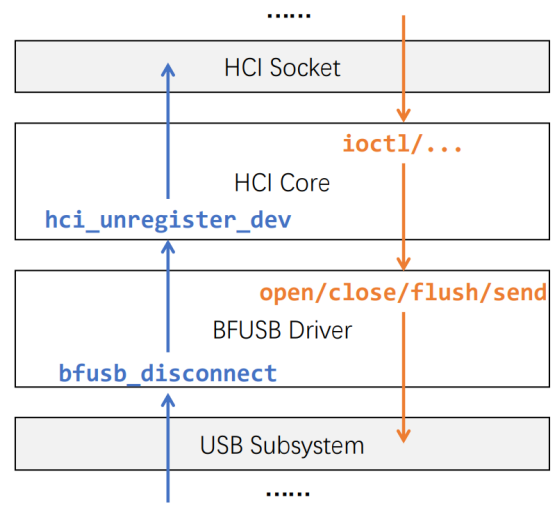
③名称特征：unreg-entry函数的名称包含隐含的关键字，例如unregister。

```
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()
```

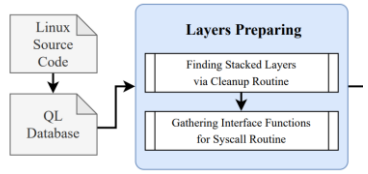
```
struct device_driver {
    ....
    // Called when the device is removed
    // from the system to unbind a device
    // from its driver
    int (*remove)  (struct device *dev);
    ....
}

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

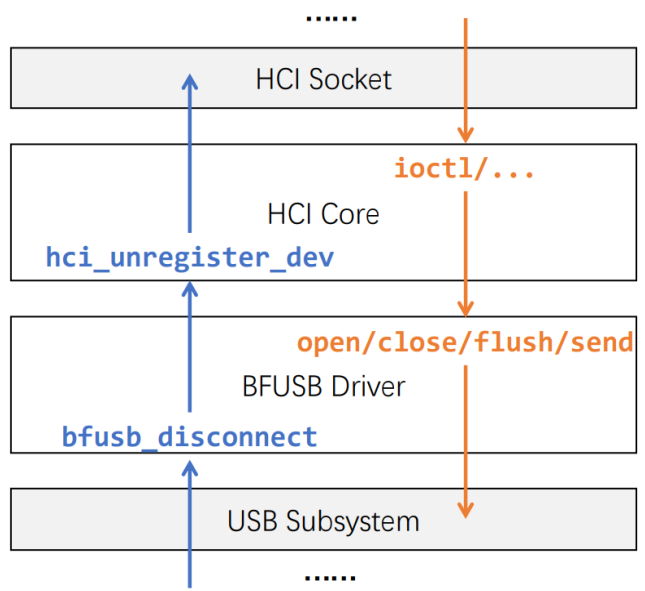


为了建立设备层的层次关系，UACatcher会扫描KBuild和MakeFile，提取出keyword，enclosedir，relative_path以生成一个唯一的层次描述符

① keyword：编译选项中的CONFIG宏

② enclosedir：MakeFile所在目录的目录名

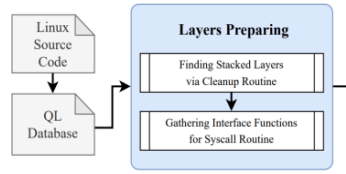
③ relative_path：相对路径




```
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



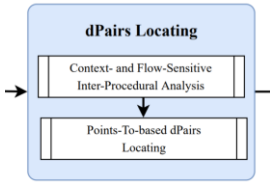
根据分层模型，接口函数由下层暴露给上层以承担顶层系统调用例程。UACatcher通过两个步骤收集这些函数

（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。

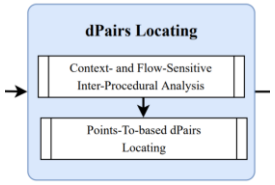
②每个待分析函数会被分解为基本块并构建CFG图，进而以流敏感的方式确定被分析函数中的所有函数调用

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，UACachter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。

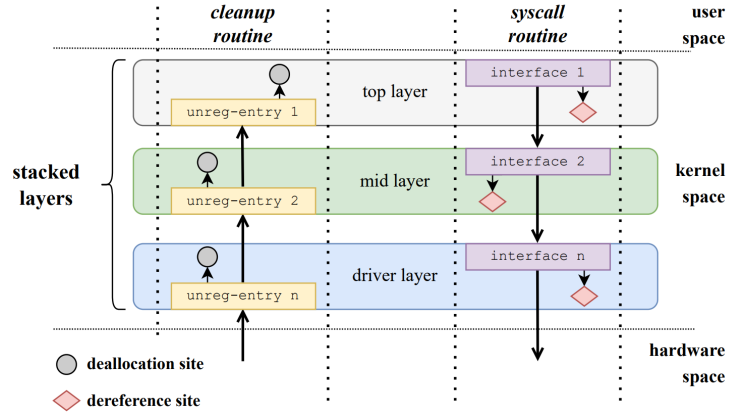
① 专用对象释放函数通常只接收一个特定指针作为参数，因此可使用基于签名的分析找到可能候选者

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

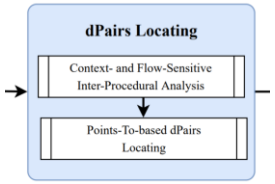
```
class KStandardDeallocationFunction extends DeallocationFunction {
  int freedArg;

  Quick Evaluation: KStandardDeallocationFunction
  KStandardDeallocationFunction() {
      hasName([
          "kfree", // kfree(const void *);
          "kfree_sensitive",  // kfree_sensitive(const void *)
          "kfree_const", // void kfree_const(const void *x)
          "kvfree"    // void kvfree(const void *addr)
          // "kfree_rcu" macro here
      ]) and
      freedArg = 0
      or
      hasName([
          "kmem_cache_free"  // kmem_cache_free(struct kmem_cache *, void *)
      ]) and
      freedArg = 1
      or
      // export function need to be modeled
      hasName([
          "kfree_skb",
          "destroy_workqueue",
          "rfkill_destroy",
          "crypto_free_shash"
          // TODO: more?
      ]) and
      freedArg = 0
  }
```

（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。



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

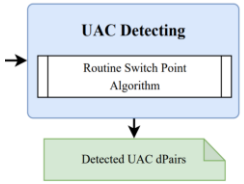
①指向分析被设置了阈值。如果指向分析未能准确定位指向集（在内核分析中很常见）并获得低置信度，该站点会被丢弃

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)

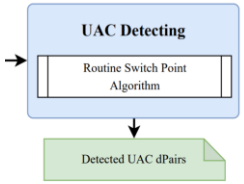
```
.....
1. mutex_lock(&ndev->req_lock)
2. test_and_clear_bit(NCI_UP, &ndev->flags)
.....
3. mutex_unlock(&ndev->req_lock)
4. destroy_workqueue(ndev->cmd_wq)
.....
```

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

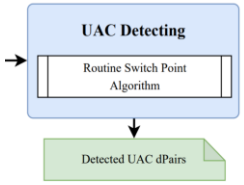
Input : deallocation path $Path_1$, dereference path $Path_2$
Output: context switch points P

Function UACDetect()
  $S_1 = \text{End}(Path_1), S_2 = \text{End}(Path_2)$
  $r = S_2, PC1 = \emptyset, PC2 = \emptyset$
  $L_{inter} = \text{InterLockA}(S_1, r)$
  while $L_{inter} \neq \emptyset$ **do**
    $r = \text{Pred}(r, Path_2)$
    if $\text{Action}(r) \in$ ConstraintCheck **then**
      $PC1 = PC1 \cup \{\text{Action}(r)\}$
    $L_{inter} = \text{InterLockA}(S_1, r)$
  $P = P \cup \{r\}$
  $r = \text{Start}(Path_1)$
  $L_{inter} = \text{InterLockB}(S_2, r)$
  while $L_{inter} \neq \emptyset$ **do**
    $r = \text{Succ}(r, Path_1)$
    if $\text{Action}(r) \in$ ConstraintChange **then**
      $PC2 = PC2 \cup \{\text{Action}(r)\}$
    $L_{inter} = \text{InterLockB}(S_2, r)$
  $P = P \cup \{r\}$
  if Satisfy $(PC1, PC2)$ **then**
    return $True$
  return $False$
Function InterLockA$(s1, s2)$
  return HHoldLock$(s1) \cap$ CHoldLock$(s2)$
Function InterLockB$(s1, s2)$
  return CHoldLock$(s1) \cap$ CHoldLock$(s2)$

```
cleanup routine:          syscall routine:
1. lock(l1);              9.  CONS1-CHECK
2. kfree(p);①             10. lock(l1);
3. CONS1-CHANGE           11. *p = 1; ③
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. unlock(l2);            16. unlock(l1);
```

⑥ ⑤ ⑦ ⑧

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

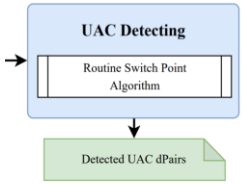
Algorithm 1: Routine Switch Point Algorithm

Input : deallocation path $Path_1$, dereference path $Path_2$
Output: context switch points P

Function UACDetect()
  $S_1 = \text{End}(Path_1), S_2 = \text{End}(Path_2)$
  $r = S_2, PC1 = \emptyset, PC2 = \emptyset$
  $L_{inter} = \text{InterLockA}(S_1, r)$
  while $L_{inter} \neq \emptyset$ **do**
    $r = \text{Pred}(r, Path_2)$
    if $\text{Action}(r) \in$ ConstraintCheck **then**
      $PC1 = PC1 \cup \{\text{Action}(r)\}$
    $L_{inter} = \text{InterLockA}(S_1, r)$
  $P = P \cup \{r\}$
  $r = \text{Start}(Path_1)$
  $L_{inter} = \text{InterLockB}(S_2, r)$
  while $L_{inter} \neq \emptyset$ **do**
    $r = \text{Succ}(r, Path_1)$
    if $\text{Action}(r) \in$ ConstraintChange **then**
      $PC2 = PC2 \cup \{\text{Action}(r)\}$
    $L_{inter} = \text{InterLockB}(S_2, r)$
  $P = P \cup \{r\}$
  if Satisfy $(PC1, PC2)$ **then**
    return *True*
  return *False*
Function InterLockA$(s1, s2)$
  return $\text{HHoldLock}(s1) \cap \text{CHoldLock}(s2)$
Function InterLockB$(s1, s2)$
  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

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

Algorithm 1: Routine Switch Point Algorithm

Input : deallocation path $Path_1$, dereference path $Path_2$
Output: context switch points P

Function UACDetect()
  $S_1 = \text{End}(Path_1), S_2 = \text{End}(Path_2)$
  $r = S_2, PC1 = \emptyset, PC2 = \emptyset$
  $L_{inter} = \text{InterLockA}(S_1, r)$
  while $L_{inter} \neq \emptyset$ **do**
    $r = \text{Pred}(r, Path_2)$
    if $\text{Action}(r) \in$ ConstraintCheck **then**
      $PC1 = PC1 \cup \{\text{Action}(r)\}$
    $L_{inter} = \text{InterLockA}(S_1, r)$
  $P = P \cup \{r\}$
  $r = \text{Start}(Path_1)$
  $L_{inter} = \text{InterLockB}(S_2, r)$
  while $L_{inter} \neq \emptyset$ **do**
    $r = \text{Succ}(r, Path_1)$
    if $\text{Action}(r) \in$ ConstraintChange **then**
      $PC2 = PC2 \cup \{\text{Action}(r)\}$
    $L_{inter} = \text{InterLockB}(S_2, r)$
  $P = P \cup \{r\}$
  if Satisfy $(PC1, PC2)$ **then**
    return *True*
  return *False*

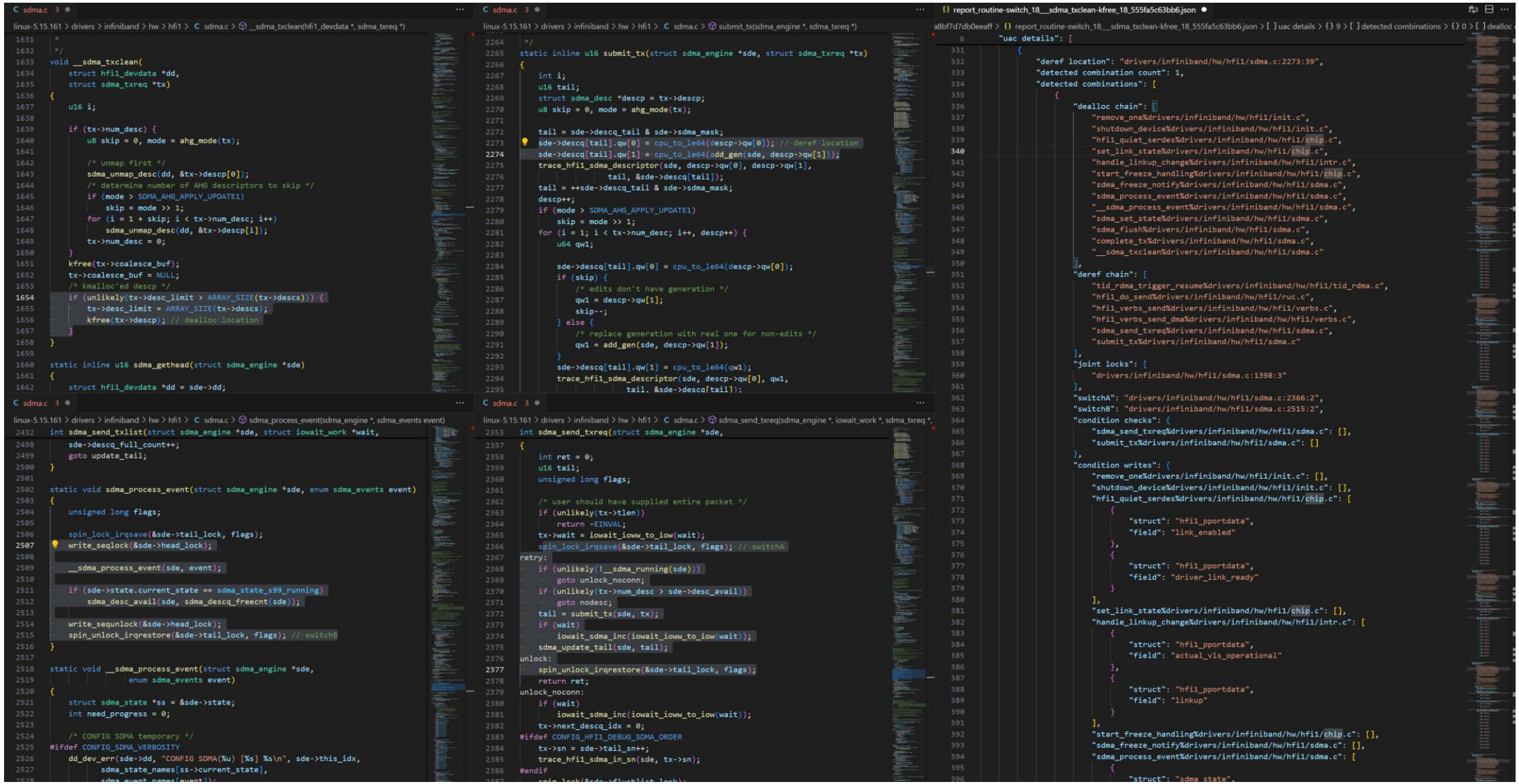
Function InterLockA$(s1, s2)$
  return HHoldLock$(s1) \cap$ CHoldLock$(s2)$

Function InterLockB$(s1, s2)$
  return CHoldLock$(s1) \cap$ CHoldLock$(s2)$

```
cleanup routine:          syscall routine:
1. lock(l1);              9.  CONS1-CHECK
2. kfree(p);①             10. lock(l1);
3. CONS1-CHANGE           11. *p = 1; ③
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. unlock(l2);            16. unlock(l1);
```

⑥ → 1. lock(l1);
⑤ ← 10. lock(l1);
⑦ → 4. unlock(l1);
⑧ → 8. unlock(l2);

UACatcher——效果



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
```