第一个问题

此错误报告描述了提交64b875f7ac8a引入的两个问题（“ptrace：捕获ptracer的凭证而不是PT\_PTRACE\_CAP”，在v4.10中引入，但也稳定向后移植到旧版本）。

我会在一分钟内发送建议的补丁（“ptrace：Fix - > PTRACE\_TRACEME的ptracer\_cred处理”）。

当调用PTRACE\_TRACEME时，ptrace\_link（）将获得对父目标凭据的RCU引用，然后将该指针指向get\_cred（）。然而，像struct cred这样的对象生存期规则不允许无条件地将RCU引用转换为稳定引用。

PTRACE\_TRACEME记录父进程的凭证，就像父进程作为主体一样，但事实并非如此。

如果恶意无特权的子进程使用PTRACE\_TRACEME并且父进程享有特权，稍后，父进程将受到攻击者控制（因为它会丢弃权限并调用execve（）），攻击者最终控制了具有特权ptrace关系的两个进程，这可能被滥用来ptrace suid二进制文件并获得root权限。

当我试图重构cred\_guard\_mutex逻辑时，我偶然发现了以下问题：

ptrace关系可以通过两种方式设置：跟踪器连接到另一个进程（PTRACE\_ATTACH / PTRACE\_SEIZE），或者tracee强制其父级连接到它（PTRACE\_TRACEME）。当tracee通过特权获取execve（）时，内核会检查ptrace关系是否具有特权。如果不是，则抑制execve的特权获得效果。

这里的想法是一个特权跟踪器（例如，如果root运行“strace”

允许某些进程跟踪setuid / setcap执行，但是不允许使用非特权跟踪器，因为否则可能会将任意代码注入特权进程。

在PTRACE\_ATTACH / PTRACE\_SEIZE情况下，跟踪器的凭证在它调用PTRACE\_ATTACH / PTRACE\_SEIZE时被记录;之后，当tracee通过execve（）时，将检查记录的凭据是否能够通过tracee的用户命名空间。

但是在PTRACE\_TRACEME情况下，即使跟踪器没有请求操作，内核也会记录\_ tracer's\_credentials。这有两个问题。

首先，存在对象生存期问题：

ptrace\_traceme（） - > ptrace\_link（）在RCU读取端临界区中抓取\_\_task\_cred（new\_parent），然后将cred传递给\_\_ptrace\_link（），它会调用get\_cred（）。

如果父同时切换其信用（例如，通过setresuid（）），则信用'的引用可能已经为零，在这种情况下，put\_cred\_rcu（）已经被调度。

内核通常使用put\_cred\_rcu（）中的以下代码在此处发生内存损坏之前设法panic（）;

但是，如果这段代码完全以正确的方式进行竞争，我认为内存损坏也是可能的。

一、前置知识

Ptrace功能。

Ptrace是linux系统中为了方便控制和调试程序提供的系统调用。Ptrace()系统调用的第一个参数ptrace\_request指定不同的跟踪请求，当调用PTRACE\_TRACEME时，表示子进程发出请主动跟踪我的请求，如果该进程ptrace成功，父进程就可以对该进程进行调试操作。固定调用模式如下：

Pid = fork();

If(pid == 0 )//child

{

Ptrace(PTRACE\_TRACEME,0,NULL,NULL);

Execv();//启动要跟踪的程序

}

Else //parent

{

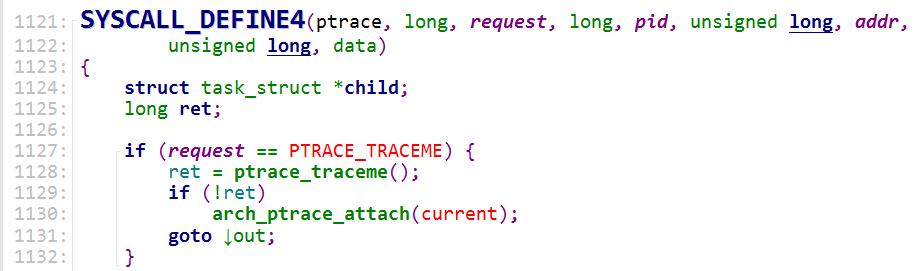
Wait(&status);

….//进行一些调试命令的操作

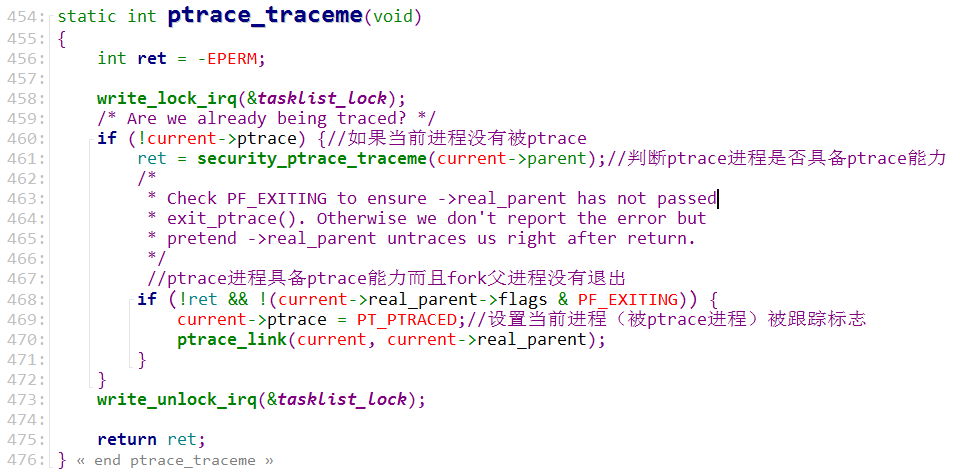
}

二、漏洞原理

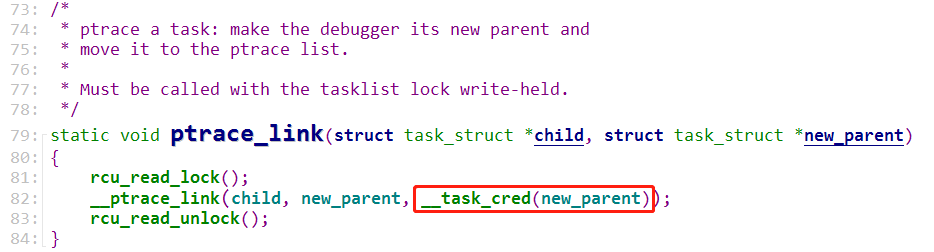
该漏洞是多核条件竞争类型的，发生在ptrace\_link中。在ptrace系统调用时，代码kernel\ptrace.c中实现如下：



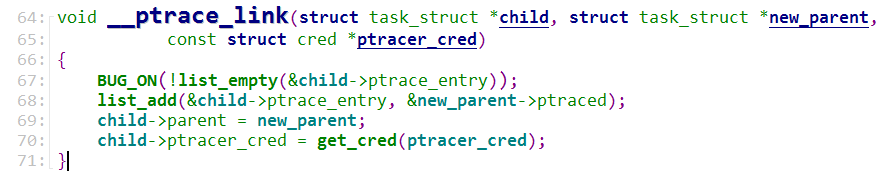
当request等于PTRACE\_TRACEME时，子进程主动发出被跟踪请求。接着调用ptrace\_traceme()函数，具体实现如下：



Current全局变量是当前子进程的task\_struct，通过current->ptrace标志判断当前进程是否已经被ptrace，如果没有，进入if语句。461行，判断跟踪当前进程的进程是否具备ptrace能力，468行，判断当前进程的父进程（调用fork的进程）是否退出。如果都满足条件，则先设置current->ptrace为PT\_PTRACED，表示该进程处于被跟踪状态。然后调用ptrace\_link，ptrace\_link函数就是把当前被跟着进程放到父进程（调用fork函数的进程）的调试列表中，具体实现代码如下：



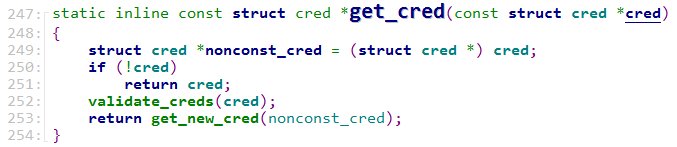
\_\_ptrace\_link()函数实现如下：



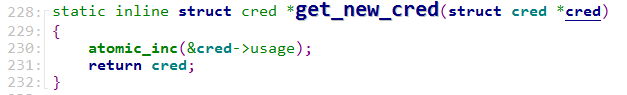
\_\_ptrace\_link()函数第三个参数是cred类型，传入的是\_\_task\_cred(current->real\_parent)，\_\_task\_cred()实现如下：



通过rcu\_dereference对父进程的real\_cred进行解引用，其实就是取real\_cred地址指针。然后再调用get\_cred()处理，get\_cred实现如下：



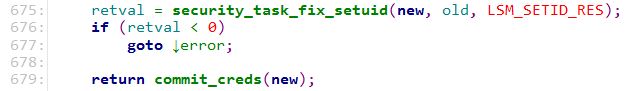
然后返回get\_new\_cred()处理的结果，get\_new\_cred()函数实现如下：



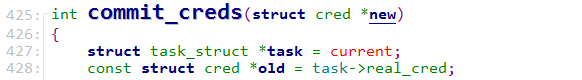
其实就是将current->real\_parent->real\_cred的值赋值给子进程使用，并增加一次引用计数。通俗的讲，就是子进程引用了父进程的cred资源。注意，这里是引用，不是复制。那么问题就出现了，如果在父进程中释放了cred，子进程中不停的引用该cred，那就有可能发生条件竞争，引用已经释放的内存，导致系统崩溃。

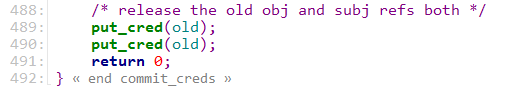
1. poc分析。

从作者提供的poc可知，通过在父进程中不断调用setresuid()函数便有机会释放cred，代码实现如下：

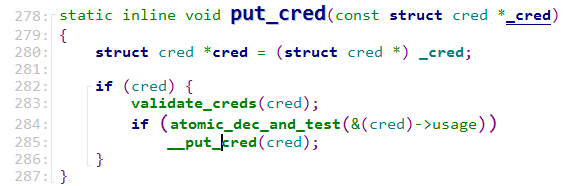


调用commit\_creds函数。

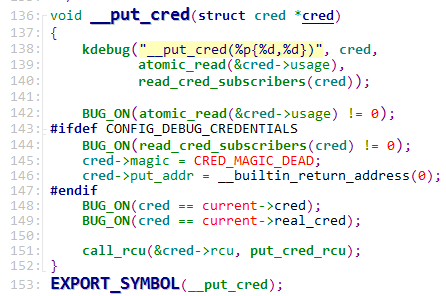




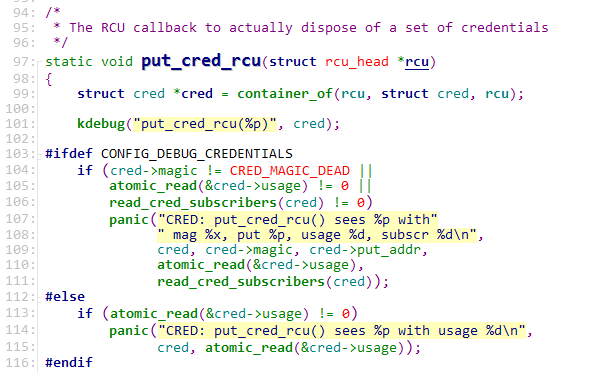
调用put\_cred()释放当前进程的cred。继续看put\_cred()函数实现：



Atomic\_dec\_and\_test判断cred->usage是否为1，即引用计数是否就只为1，如果为1，就减一，此时引用计数为0，表明该资源不在被引用，可以被释放。然后调用\_\_put\_cred()。该函数实现如下：



这里会调用call\_rcu设置rcu释放资源的回调函数put\_cred\_rcu()。实现如下：

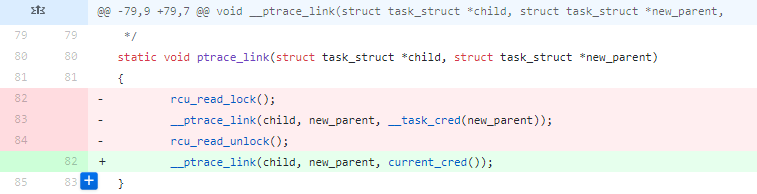


这里会再次判断cred->usage是否等于0，如果不等于0，那就说明该资源还处在被引用状态，不能被释放，故触发panic。

四、关于如何正确运行poc并触发漏洞。

通俗的讲，对于单个CPU核心，由于call\_rcu()函数本身是不可被中断的，即会一直占用cpu资源。所以如果只有单个CPU的话，就无法保证条件竞争，因此要采用多核。

五、补丁：



将第三个参数换成了current\_cred()，引用当前子进程自己的cred。