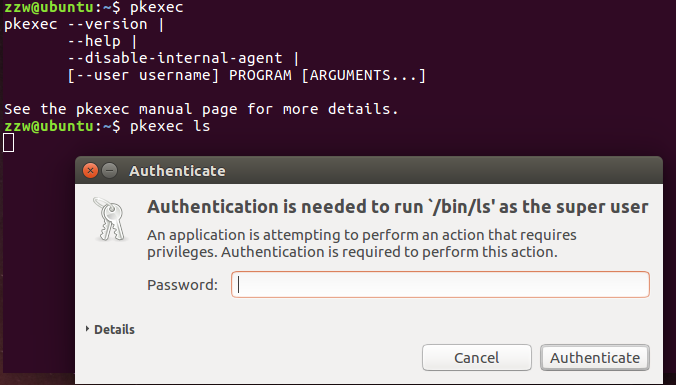
## 漏洞信息

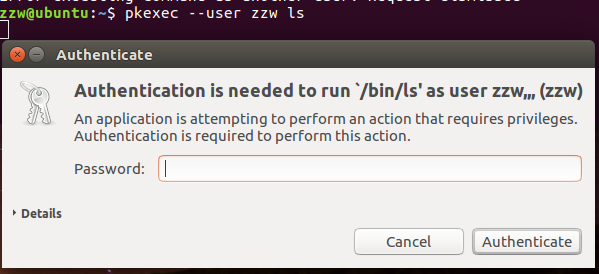
CVE-2021-4034 是在pokit套件中pkexec程序的一个本地权限提升漏洞。pkexec用来授权用户其他身份执行程序，其具备suid属性。常规功能下不直接支持执行命令，无法利用suid提权。但特定版本的pkexec无法正确处理参数传入，并尝试将环境变量作为命令执行。攻击者可以构造恶意环境变量，从而导致pkexec任意代码执行，提权。

### pkexec

pkexec是polkit中的一个程序。用以进行授权polkit是一个授权管理器，其系统架构由授权和身份验证代理组成，pkexec是其中polkit的其中一个工具，他的作用有点类似于sudo，允许用户以另一个用户身份执行命令。



当没有指定—user 时，程序执行默认为root.



### 漏洞复现

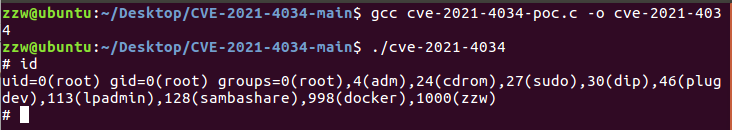
POC链接：<https://github.com/berdav/CVE-2021-4034.git>

<https://github.com/arthepsy/CVE-2021-4034>



主要过程有：

* 创建一个目录。目录名字为GCONV\_PATH=.（有点）
* 在GCONV\_PATH=.目录下，创建一个文件pwnkit。
* 给创建的pwnkit文件创建执行权限。
* 打开pwnkit,将shell指向的字符串写入pwnkit中。
* 将pwnkit文件编译为动态链接库。
* 设置相关的环境变量，执行pkexec。



## 漏洞分析

### 前置知识

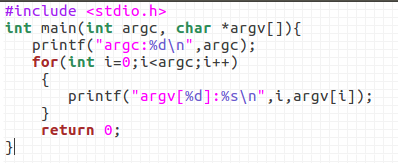
#### 2.1.1 argc与argv

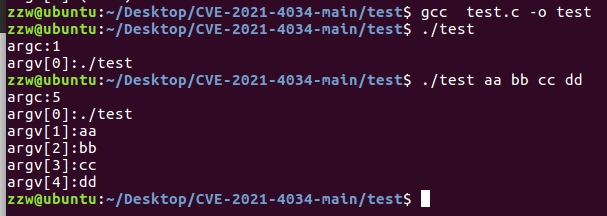
在日常编程时，main函数主要都是void main()或者int main(),但在C89/C99中main的主要形式是int main(int argc,char \* argv)

Argc是argument count的缩写，表示传入main函数传入的参数个数。

Argv是argument value的缩写，表示传入main函数的参数序列或指针。且argv[0]一定是程序的名称（包含程序所在的完整路径）。即参数个数为argc-1。

示例如下：

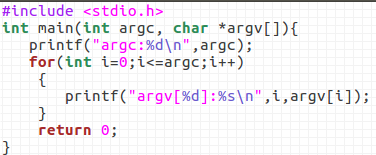


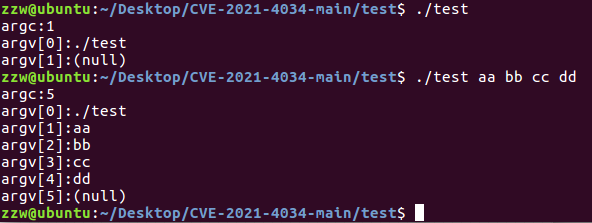


要注意第一个参数：argv[0]指的是程序的真实路径名。

#### 2.1.2 argv的越界数据读取

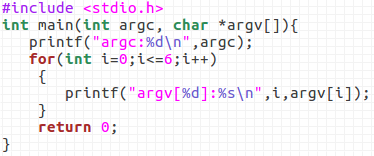
在上述代码中，打印的数量是argc的个数。没有越界，如果打印的i<=argc呢？

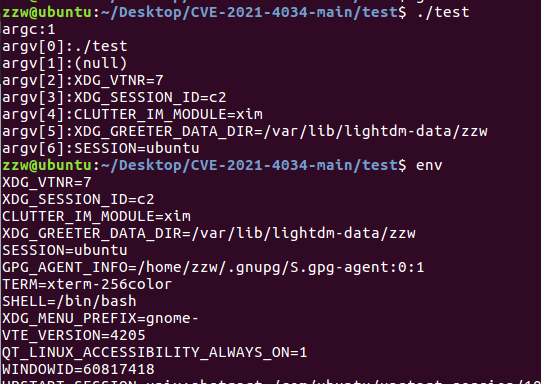




可以发现argv[argc]=null，是个空。

如果再越界多打印一些呢？比如将i=6,打印





可以看到当argv[]对参数访问时，如果越界，argv[argc]=null,之后就是系统的环境变量。

#### 2.1.3 有意思的execve()

定义：int execve(const char \*filename, char \*const argv[ ], char \*const envp[ ]);

功能：在子进程中执行一个程序。

参数1:二进制可执行文件路径

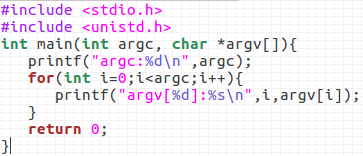
参数2 :要调用程序执行的参数序列，即传入的参数。包括argv[0],一般是程序名。并以null结束。

参数3: 环境变量参数序列。一般传递NULL,表示可变参数的结尾。

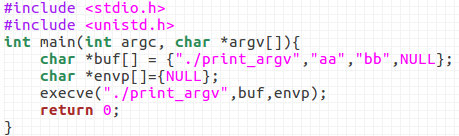
注意：argc和envp 都必须要以null指针结束

现在用代码观察其特性：

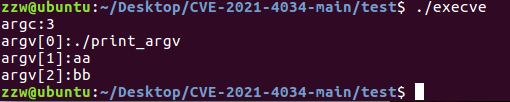
Print\_argv.c 编译为print\_argv，功能只是单纯的打印参数。



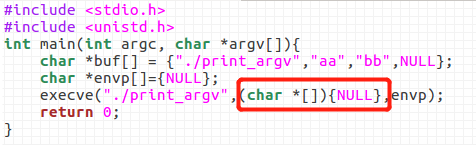
execve.c 编译为execve,代码中调用execve执行上面编译的print\_argv程序，参数为aa,bb，并以Null结尾，环境变量为空。



运行execve



但是，如果将上述的buf字符串数组置空，并作为execve的第二个函数，有什么现象呢？





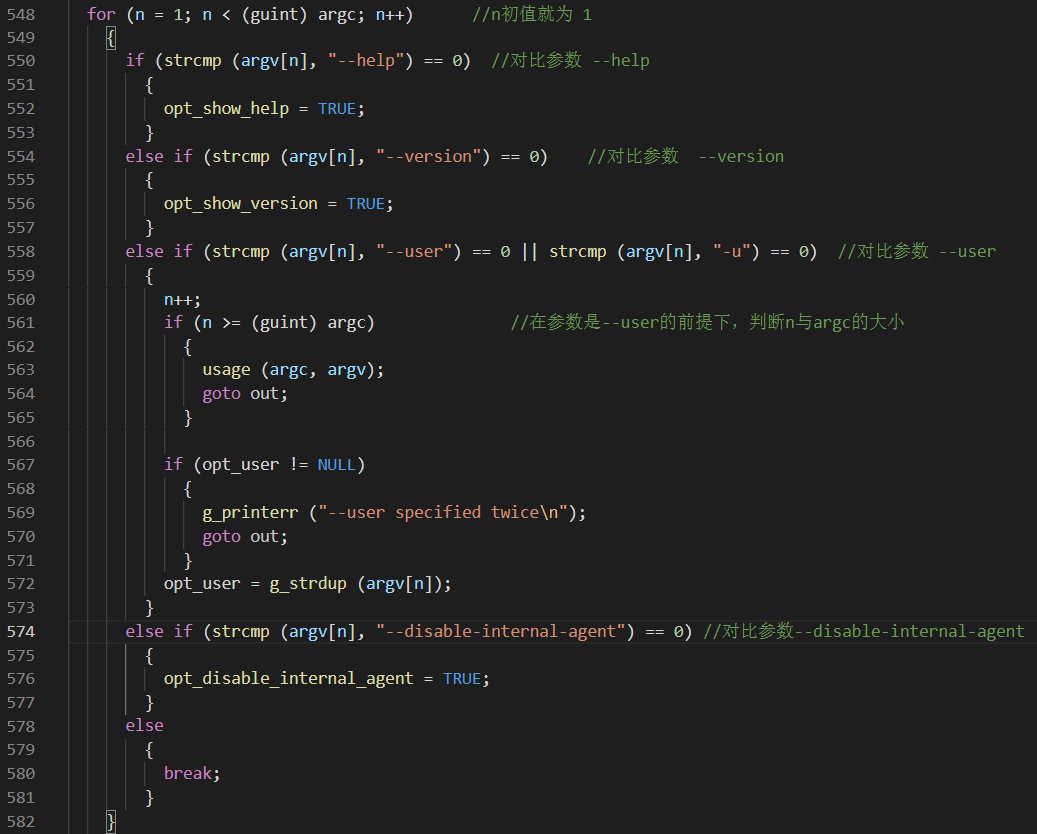
execve这个程序调用了execve函数，执行了print\_argv，但是参数为空，所以打印了0。

而pkexec的这个漏洞正是利用了execve的这个特性造成数组访问越界。

### 漏洞原理

从正常使用pkexec的视角，查看下pkexec的源代码，从main函数入手：

https://github.com/wingo/polkit/blob/master/src/programs/pkexec.c



第548行直接把n赋值为1。

第550行，554行，558行，574行都是对比参数，如果不符合参数直接打印用法或者退出。



例如场景：$:pkexec ls

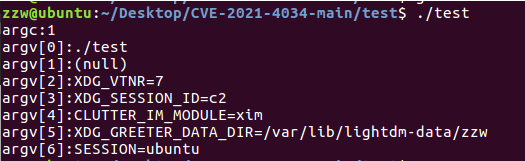
最后argv[1]=path= /bin/ls 得到了其绝对路径

如果正常使用，按照解析流程是没有问题的。如果argc为0，则会导致越界访问。

548行，n为1

624行，argv[1]越界访问。（argc为0）

因为没有输入参数，而根据栈帧的布局，argv和envp是挨在一起的。如下图



在上图中argc和argv紧邻在一起。参数和环境变量之间有个null。环境变量的第一个也就是参数变量(argv）的第二个。Argv[2] == envp[0] arg[3] == envp[1]。

对于上述所说的argc 为0的情况：

Argc[0] == null

Argc[1] == envp[0]

Argc[2] == envp[1]

这样应该就很好理解了。

当argc为0时，argv[1]实际访问的就是envp[0]的环境变量了。代入上文624和653行：

Path = envp[0]

S= g\_find\_program\_in\_path (path)

这个函数会在PATH环境变量的目录中搜素程序的绝对路径并返回。

Argv[1] = envp[0] = 绝对路径

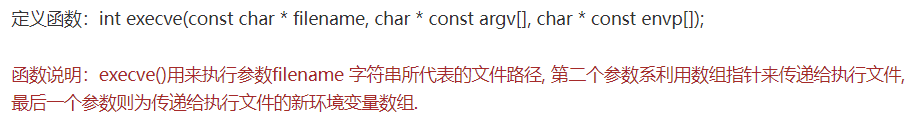
通过以上方式就可以注入一个环境变量。

* 如果PATH环境变量路径中有PATH=/home/user/aaa/目录。/home/user/aaa/目录存在，且在目录下存在bb这个可执行程序。
* 按照以上的逻辑s= g\_find\_program\_in\_path (bb)，则会返回/home/user/aaa/bb。即argv[1]=envp[0]= /home/user/aaa/bb。
* 进一步：让组合文件名中包含“=”，即PATH=/home/user/aaa/=. 。再创建一个/home/user/aaa/=.目录。并放入一个可执行文件bb。最终效果为envp[0]的值就为/home/user/aaa/=./bb

根据分析，如果指定了恶意的envp[0],那么可以写入一个环境变量到目标进程中。现在问题是需要找到一个可以利用的环境变量，然后让程序执行。

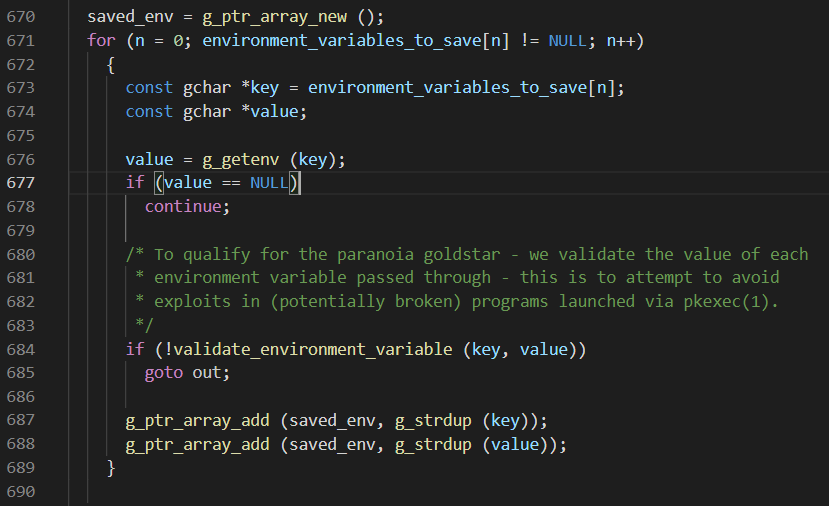
### 思考

注入了环境变量，一般都是引入一个危险的环境变量，进而导入恶意的so执行。但execve的第三个参数可以直接写入环境变量。



原因在于linux的动态连接器ld-linux-x86-64.so.2 会在特权程序执行的时候清除敏感环境变量。

此外在pkexec函数的670-689行也对环境变量进行了一定的判断。防止恶意利用。



现在所做的利用就是两个问题：

1. 找到一个合适的环境变量，进行注入。
2. 通过环境变量的设置加载执行恶意代码。

#### 2.3.1寻找不安全的环境变量

GCONV\_PATH是一个危险的环境变量，其常使用场景是在字符集切换时，可以强制iconv\_open读取另一个配置文件（默认配置文件/usr/lib/gconv/gconv-modules）。而在程序代码中正好有函数可以调用到iconv\_open。

g\_printerr中间接调用了linux的iconv\_open函数:



在代码中有很多的g\_printerr()函数。g\_printerr()通常打印UTF-8错误消息，但如果环境变量CHARSET不是utf-8,则可以打印另一个字符集中的消息。Charset是安全的环境变量，不会被清除。

当字符集切换时，会触发iconv\_open()执行共享库。

其iconv\_open执行过程:

Iconv\_open函数首先会找到系统提供的gconv-modules配置文件，这个文件中有各个字符集的相关存储路径，每个字符集的相关信息存储在一个so文件中，即gconv-modules提供了各个字符集so的位置，之后会调用so中的gconv和gonv\_init函数。

1.iconv\_open 函数依照 GCONV\_PATH 找到 gconv-modules 文件，这个文件中包含了各个字符集的相关信息存储的路径，每个字符集的相关信息存储在一个.so 文件中，即 gconv-modules 文件提供了各个字符集的.so 文件所在位置。

2. 根据 gconv-modules 文件的指示找到参数对应的.so 文件。

3. 调用.so 文件中的 gconv() 和 gonv\_init() 函数。

那么，如果修改了系统的GCONV\_PATH环境变量，就可以改变gconv-modules配置文件的位置，从而执行恶意so中的文件实现任意命令执行。

所以，可以重新引入GCONV\_PATH这个被清除的环境变量，让pkexec以特权身份执行恶意共享库。

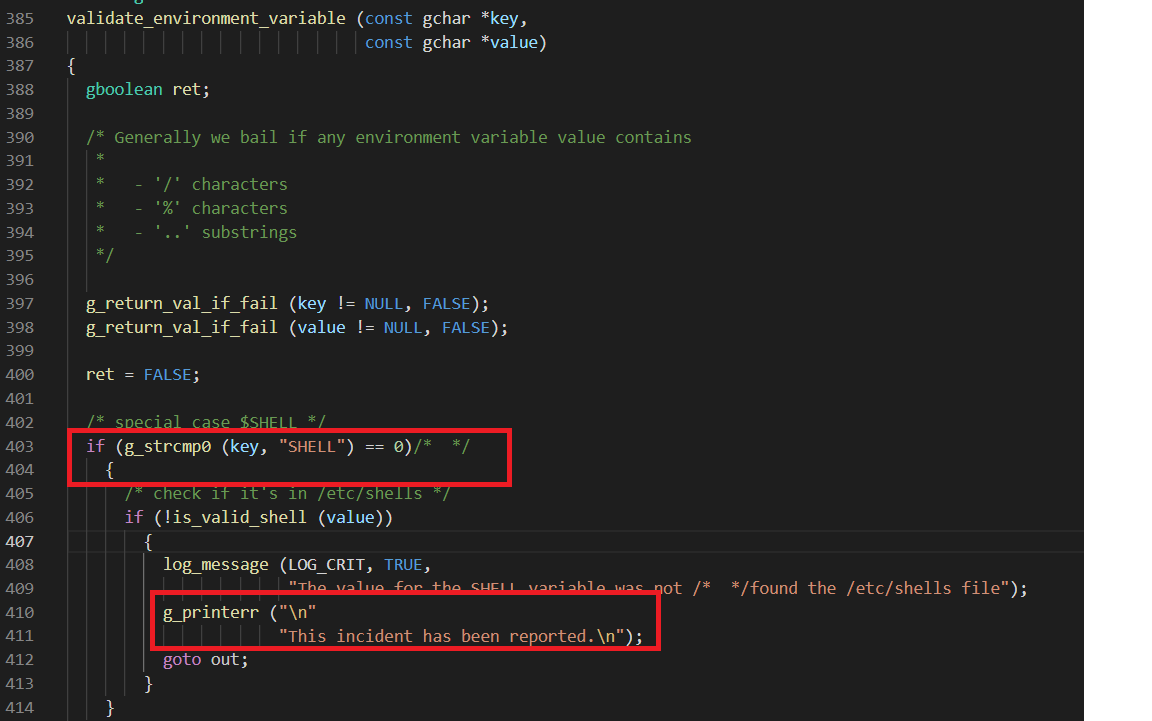
总结流程如下：

* 修改CHARSET环境变量
* g\_printerr打印触发iconv\_open
* iconv\_open从GCONV\_PATH环境变量中寻找modules目录
* GCONV\_PATH环境变量是可以伪造的，即伪造modules目录
* 在伪造的modules目录中，伪造恶意so的路径信息
* 加载so,执行提权函数。

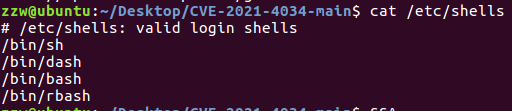
#### 2.3.2 如何劫持执行流

根据以上分析过程，g\_printerr函数是执行利用的入口。那么如何触发执行g\_printerr函数呢？

在pkexec的源码中寻找可以触发的g\_printerr函数执行地方，有很多处错误打印，但我们寻找的最好是环变量相关操作引起的打印执行，因为环境execve执行时环境变量可以注入。



在验证环境变量相关操作时，会有对比SHELL环境变量的一个操作，这样就可以对shell环境变量进行构造。从而触发g\_printerr打印。



### 利用过程

根据poc一步一步分析过程。



* **伪造环境变量所指的目录文件结构**

23行 创建GCONV\_PATH=.目录，并在目录中创建pwnkit文件

24行 创建pwnkit目录（用来存放恶意的so），并将恶意so的路径信息写入pwnkit/gconv-modules文件。内容是：module UTF-8// PWNKIT// pwnkit 1 这是modules配置文件的语法。具体含义参考 [reference](https://xy2401.com/local-docs/gnu/manual.zh/libc/glibc-iconv-Implementation.html) .含义主要是将utf-8字符集切换到PWNKIT字符集。转换所需要so的信息在pwnkit中。

* 在pwnkit目录下编译一个so。25——28行
* Execve调用execve

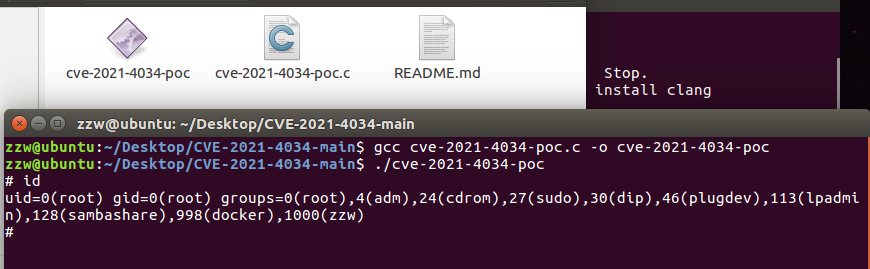
char \*env[] = { "pwnkit", #触发越界写漏洞，最终使得写入环境变量：GCONV\_PATH=./pwnkit

"PATH=GCONV\_PATH=.", #使得g\_find\_program\_in\_path查找pwnkit时会在GCONV\_PATH=.目录中找到pwnkit

"CHARSET=PWNKIT", #触发g\_printerr更换编码字符，从而调用so中的恶意代码

"SHELL=pwnkit", #触发调用g\_printerr函数

NULL };



## 小结

* 首先利用越界读写，伪造了GCONV\_PATH=./pwnkit环境变量。这样为后续字符集切换时提供了要查找的目录
* 使用SHELL环境变量触发g\_printerr 函数。触发切换字符集，加载恶意so函数，提权。

## 参考链接

<https://xz.aliyun.com/t/10870>

<https://www.anquanke.com/post/id/267774#h3-10>

<https://xz.aliyun.com/t/10905>

<https://www.anquanke.com/post/id/267774#h2-12>

<https://www.wangan.com/p/7fy7fg4103b2ee22#%E5%88%A9%E7%94%A8GCONV_PATH%E4%B8%8Eiconv>