linux内存执行ELF之fexecve

作者: 腾讯安全平台部数据安全团队 七夜

文章来源: 腾讯安平小密圈

简介:

> 腾讯安全平台部成立于2005年，自身安全能力涵盖账号安全、反欺诈、撞库防护、业务防刷、DDoS防御、移动端安全、数据保护、入侵防护、AI攻防研究、安全大数据等，服务于腾讯全线业务，守护十亿用户的安全，更依托腾讯云和"互联网+"将安全能力输出到各个行业。

> 数据安全团队隶属于腾讯安全平台部，团队主要工作是负责腾讯业务的数据安全、反入侵、安全评估等工作；主要研究方向是 主机安全、网络安全、大数据分析、软件供应链攻击与检测、可信计算、安全标准等

内存执行ELF

最近会讲到了内存执行ELF以及混淆进程参数的方法，主要是涉及两个方法：memfd_create 和 fexecve，今天以fexecve原理讲解为主，会简单提一下memfd_create，毕竟是Linux的新特性。

> 文章中涉及的代码，都放到了git仓库：https://github.com/qiyeboy/TX-Knowledge-planet ，喜欢的大家自取。

memfd_create 和 fexecve

1. memfd_create：第一个允许我们在内存中创建一个文件，但是它在内存中的存储并不会被映射到文件系统中，至少如果映射了，我是没找到，因此不能简单的通过ls命令进行查看，现在看来这的确是相当隐蔽的。事实上，如果一个文件存在，那么我们还是可以去发现它的，谁会去调用这个文件呢？使用如下的命令：

```
lsof | grep memfd
```

2. 第二个函数，fexecve同样功能很强大，它可以帮助我们执行一个程序（同execve），但是传递给这个函数的是文件描述符，而不是文件的绝对路径，和memfd_create搭配使用非常完美！

但是这里有一个需要注意的地方，因为这两个函数相对的比较新，memfd_create 是在kernel3.17才被引进来，fexecve是glibc的一个函数，是在版本2.3.2之后才有的。 没有fexecve的时候， 只要我们弄清楚原理是可以使用其它方式取代它，而memfd_create只能用在相对较新的linux内核系统上。

fexecve的实现

今天不谈memfd_create，这是linux的新特性，与linux版本硬绑定，本人对fexecve 的实现更感兴趣，因为fexecve是glibc中的函数，我们可以看一下他的实现原理。首先介绍 fexecve的具体用法, 下面的fexecve_test.c 代码是 实现 `ls -l /dev/shm` 功能。

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>


static char *args[] = { //设置待执行程序的命令行参数
```

```
    "hic et nunc",

    "-l",
    "/dev/shm",
    NULL
};

extern char **environ;
int main(void)
{
    struct stat st;
    void *p;
    int fd, shm_fd, rc;

    shm_fd = shm_open("wurstverschwendung", O_RDWR | O_CREAT, 0777);  //创建了wurstversc
hwendung文件
    if (shm_fd == -1) {
 perror("shm_open");
 exit(1);
    }
    rc = stat("/bin/ls", &st);
    if (rc == -1) {
 perror("stat");
 exit(1);
    }
    rc = ftruncate(shm_fd, st.st_size);
    if (rc == -1) {
 perror("ftruncate");
 exit(1);
    }
    p = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE, MAP_SHARED,
      shm_fd, 0);
    if (p == MAP_FAILED) {
 perror("mmap");
 exit(1);
    }
    fd = open("/bin/ls", O_RDONLY, 0); //将ls 命令文件写入到wurstverschwendung文件
    if (fd == -1) {
 perror("openls");
 exit(1);
    }
    rc = read(fd, p, st.st_size);
    if (rc == -1) {
 perror("read");
 exit(1);
    }
    if (rc != st.st_size) {
 fputs("Strange situation!\n", stderr);
 exit(1);
    }
    munmap(p, st.st_size);
    close(shm_fd);

```

```
    shm_fd = shm_open("wurstverschwendung", O_RDONLY, 0);
    fexecve(shm_fd, args, environ); //执行 wurstverschwendung
    perror("fexecve");
    return 0;
}
```

代码中主要是分为了三步：

1. 首先通过shm_open函数创建了wurstverschwendung文件，文件其实是在/dev/shm目录中
2. 将ls 命令文件写入到wurstverschwendung文件
3. 通过fexecve执行wurstverschwendung文件，因为/dev/shm在内存中，因此fexecve实际上是在内存中执行文件。

对fexecve_test.c 进行编译并执行，可以看到/dev/shm下面确实生成了wurstverschwendung文件。

```
[root@VM_0_13_centos netelf]# gcc -lrt fexecve_test.c  -o  fexecve_test
[root@VM_0_13_centos netelf]# ./fexecve_test
total 116
-rwxrwxr-x 1 root root 117616 Feb 23 23:16 wurstverschwendung
[root@VM_0_13_centos netelf]#
```

调试角度

fexecve是如何执行内存中的文件的呢？一般可以从调试和源码的角度来探究其中的原理。首先使用strace调试fexecve_test：

```
strace -f -tt -T ./fexecve_test
```

从打印的日志中，找到open系统调用，从创建文件开始关联：

```
23:19:26.848971 futex(0x7f73776cd310, FUTEX_WAKE_PRIVATE, 2147483647) = 0 <0.000010>
23:19:26.849018 open("/dev/shm/wurstverschwendung", O_RDWR|O_CREAT|O_NOFOLLOW|O_CLOEXEC, 0777) = 3 <0.000014>
23:19:26.849060 stat("/bin/ls", {st_mode=S_IFREG|0755, st_size=117616, ...}) = 0 <0.000012>
23:19:26.849103 ftruncate(3, 117616)    = 0 <0.000013>
23:19:26.849138 mmap(NULL, 117616, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0) = 0x7f73778bf000 <0.000011>
23:19:26.849171 open("/bin/ls", O_RDONLY) = 4 <0.000012>
23:19:26.849204 read(4, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\2\0>\0\1\0\0\0HK@\0\0\0\0\0"..., 117616) = 117616 <0.0001
23:19:26.849331 munmap(0x7f73778bf000, 117616) = 0 <0.000016>
23:19:26.849371 close(3)                = 0 <0.000008>
23:19:26.849399 open("/dev/shm/wurstverschwendung", O_RDONLY|O_NOFOLLOW|O_CLOEXEC) = 3 <0.000010>
23:19:26.849447 execve("/proc/self/fd/3", ["hic et nunc", "-l", "/dev/shm"], [/* 33 vars */]) = 0 <0.000178>
23:19:26.849689 brk(0)                  = 0x150c000 <0.000008>
```

大家可以看到shm_open 其实是在/dev/shm创建文件，而execve的执行文件为/proc/self/fd/3，为进程中打开的文件符号链接，这个指向的就是shm_open创建的文件，但是从监控execve的角度来说， execve无法获取执行文件的路径，从而实现了混淆。

源码角度

从上文中，我们大致知道了原理。具体细节还是要看源码：glibc中的代码库中（https://github.com/jeremie-koenig/glibc/blob/master-beware-rebase/sysdeps/unix/sysv/linux/fexecve.c）。

```
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>


/* Execute the file FD refers to, overlaying the running program image.
   ARGV and ENVP are passed to the new program, as for `execve'.  */
int
fexecve (fd, argv, envp)
     int fd;
     char *const argv[];
     char *const envp[];
{
  if (fd < 0 || argv == NULL || envp == NULL)
    {
      __set_errno (EINVAL);
      return -1;
    }

  /* We use the /proc filesystem to get the information.  If it is not
     mounted we fail.  */
  char buf[sizeof "/proc/self/fd/" + sizeof (int) * 3];
  __snprintf (buf, sizeof (buf), "/proc/self/fd/%d", fd);

  /* We do not need the return value.  */
  __execve (buf, argv, envp);

  int save = errno;

  /* We come here only if the 'execve' call fails.  Determine whether
     /proc is mounted.  If not we return ENOSYS.  */
  struct stat st;
  if (stat ("/proc/self/fd", &st) != 0 && errno == ENOENT)
    save = ENOSYS;

  __set_errno (save);

  return -1;
}

```

关键部位代码：

```
 char buf[sizeof "/proc/self/fd/" + sizeof (int) * 3];
 __snprintf (buf, sizeof (buf), "/proc/self/fd/%d", fd);

 /* We do not need the return value.  */
 __execve (buf, argv, envp);
```

fexecve本质上还是调用execve，只不过文件路径是通过/proc 目录映射的。fexecve_test中实现的功能，可以用bash来简单描述，作用是等同的：

```
[root@VM_0_13_centos ~]# ls -l /dev/shm/wurstverschwendung
-rwxrwxr-x 1 root root 117616 Feb 23 23:19 /dev/shm/wurstverschwendung
[root@VM_0_13_centos ~]# bash -c "exec /dev/shm/wurstverschwendung -l /dev/shm"
total 116
-rwxrwxr-x 1 root root 117616 Feb 23 23:19 wurstverschwendung
[root@VM_0_13_centos ~]#
```

参考文献

https://0x00sec.org/t/super-stealthy-droppers/3715
https://github.com/jeremie-koenig/glibc/blob/master-beware-rebase/sysdeps/unix/sysv/linux/fexecve.c