[**加载可执行二进制文件之1.copy\_strings**](https://www.cnblogs.com/joey-hua/p/5638306.html)

从现在开始就是分析最后的核心模块exec.c了，分析完这个文件后，就会和之前的所有分析形成一个环路，从创建进程、加载进程程序到进程调度、内存管理。

exec.c的核心do\_execve函数很长，而且用到了很多其他的函数，copy\_strings就是其中一个，我们这里就先来分析这个函数。

首先看调用处，在main.c中：

|  |
| --- |
| static char \*argv\_rc[] =  {  "/bin/sh", NULL};       // 调用执行程序时参数的字符串数组。  static char \*envp\_rc[] =  {  "HOME=/", NULL};        // 调用执行程序时的环境字符串数组。    void init(void){  ...      execve ("/bin/sh", argv\_rc, envp\_rc);   // 替换成/bin/sh 程序并执行。  ...  } |

再看exec.c中：

|  |
| --- |
| /\*  \* MAX\_ARG\_PAGES 定义了新程序分配给参数和环境变量使用的内存最大页数。  \* 32 页内存应该足够了，这使得环境和参数(env+arg)空间的总合达到128kB!  \*/  #define MAX\_ARG\_PAGES 32    do\_execve (unsigned long \*eip, long tmp, char \*filename,         char \*\*argv, char \*\*envp)  {      unsigned long page[MAX\_ARG\_PAGES];  // 参数和环境字符串空间的页面指针数组。      int i, argc, envc;      // 参数和环境字符串空间中的偏移指针，初始化为指向该空间的最后一个长字处。    unsigned long p = PAGE\_SIZE \* MAX\_ARG\_PAGES - 4;  ...      // 计算参数个数和环境变量个数。    argc = count (argv);    envc = count (envp);        // 若sh\_bang 标志没有设置，则设置它，并复制指定个数的环境变量串和参数串到参数和环境空间中。        if (sh\_bang++ == 0)      {        p = copy\_strings (envc, envp, page, p, 0);        p = copy\_strings (--argc, argv + 1, page, p, 0);      }  ...  } |

mm.h:

|  |
| --- |
| #define PAGE\_SIZE 4096      // 定义内存页面的大小(字节数)。 |

exec.c和segment.h放在一起：

|  |
| --- |
| /\*  \* count()函数计算命令行参数/环境变量的个数。  \*/  //// 计算参数个数。  // 参数：argv - 参数指针数组，最后一个指针项是NULL。  // 返回：参数个数。  static int  count (char \*\*argv)  {    int i = 0;    char \*\*tmp;      if (tmp = argv)      while (get\_fs\_long ((unsigned long \*) (tmp++)))        i++;      return i;  }    //// 读取fs 段中指定地址处的长字(4 字节)。  // 参数：addr - 指定的内存地址。  // %0 - (返回的长字\_v)；%1 - (内存地址addr)。  // 返回：返回内存fs:[addr]处的长字。  extern inline unsigned long  get\_fs\_long (const unsigned long \*addr)  {    unsigned long \_v;    \_\_asm\_\_ ("movl %%fs:%1,%0": "=r" (\_v):"m" (\*addr));    return \_v;  } |

先分析获取参数/环境变量的个数，首先声明了两个指针数组argv\_rc和envp\_rc并传入execve。

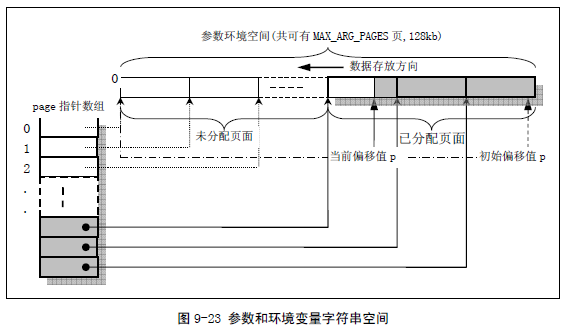
int\* a[4]     指针数组

                 表示：数组a中的元素都为int型指针

注意do\_execve的形参为char \*\*argv, char \*\*envp，指针的指针。所以也就是说在count函数中，tmp++是指针数组argv\_rc的其中的元素的地址，那么在get\_fs\_long中\*addr指的是argv\_rc的元素的值（也就是"/bin/sh"这个char类型指针），因为使用的是fs:%1而不是fs:[%1]，因此最终\_v得到的是char类型的完整地址。所以count就是根据是不是有地址值来判断数量。

|  |
| --- |
| /\*  \* 'copy\_string()'函数从用户内存空间拷贝参数和环境字符串到内核空闲页面内存中。  \* 这些已具有直接放到新用户内存中的格式。  \*  \* 由TYT(Tytso)于1991.12.24 日修改，增加了from\_kmem 参数，该参数指明了字符串或  \* 字符串数组是来自用户段还是内核段。  \*  \* from\_kmem        argv \*      argv \*\*  \*          0                用户空间      用户空间  \*          1                 内核空间      用户空间  \*          2                 内核空间      内核空间  \*  \* 我们是通过巧妙处理fs 段寄存器来操作的。由于加载一个段寄存器代价太大，所以  \* 我们尽量避免调用set\_fs()，除非实在必要。  \*/  //// 复制指定个数的参数字符串到参数和环境空间。  // 参数：argc - 欲添加的参数个数；argv - 参数指针数组；page - 参数和环境空间页面指针数组。  // p -在参数表空间中的偏移指针，始终指向已复制串的头部；from\_kmem - 字符串来源标志。  // 在do\_execve()函数中，p 初始化为指向参数表(128kB)空间的最后一个长字处，参数字符串  // 是以堆栈操作方式逆向往其中复制存放的，因此p 指针会始终指向参数字符串的头部。  // 返回：参数和环境空间当前头部指针。  static unsigned long  copy\_strings (int argc, char \*\*argv, unsigned long \*page,            unsigned long p, int from\_kmem)  {    char \*tmp, \*pag;    int len, offset = 0;    unsigned long old\_fs, new\_fs;      if (!p)      return 0;           /\* bullet-proofing \*//\* 偏移指针验证 \*/  // 取ds 寄存器值到new\_fs，并保存原fs 寄存器值到old\_fs。    new\_fs = get\_ds ();    old\_fs = get\_fs ();  // 如果字符串和字符串数组来自内核空间，则设置fs 段寄存器指向内核数据段（ds）。    if (from\_kmem == 2)      set\_fs (new\_fs);  // 循环处理各个参数，从最后一个参数逆向开始复制，复制到指定偏移地址处。    while (argc-- > 0)      {  // 如果字符串在用户空间而字符串数组在内核空间，则设置fs 段寄存器指向内核数据段（ds）。        if (from\_kmem == 1)      set\_fs (new\_fs);  // 从最后一个参数开始逆向操作，取fs 段中最后一参数指针到tmp，如果为空，则出错死机。        if (!(tmp = (char \*) get\_fs\_long (((unsigned long \*) argv) + argc)))      panic ("argc is wrong");  // 如果字符串在用户空间而字符串数组在内核空间，则恢复fs 段寄存器原值。        if (from\_kmem == 1)      set\_fs (old\_fs);  // 计算该参数字符串长度len，并使tmp 指向该参数字符串末端。        len = 0;          /\* remember zero-padding \*/        do      {           /\* 我们知道串是以NULL 字节结尾的 \*/        len++;      }        while (get\_fs\_byte (tmp++));  // 如果该字符串长度超过此时参数和环境空间中还剩余的空闲长度，则恢复fs 段寄存器并返回0。        if (p - len < 0)      {           /\* this shouldn't happen - 128kB \*/        set\_fs (old\_fs);  /\* 不会发生-因为有128kB 的空间 \*/        return 0;      }  // 复制fs 段中当前指定的参数字符串，是从该字符串尾逆向开始复制。        while (len)      {        --p;        --tmp;        --len;  // 函数刚开始执行时，偏移变量offset 被初始化为0，因此若offset-1<0，说明是首次复制字符串，  // 则令其等于p 指针在页面内的偏移值，并申请空闲页面。        if (--offset < 0)          {            offset = p % PAGE\_SIZE;  // 如果字符串和字符串数组在内核空间，则恢复fs 段寄存器原值。            if (from\_kmem == 2)          set\_fs (old\_fs);  // 如果当前偏移值p 所在的串空间页面指针数组项page[p/PAGE\_SIZE]==0，表示相应页面还不存在，  // 则需申请新的内存空闲页面，将该页面指针填入指针数组，并且也使pag 指向该新页面，若申请不  // 到空闲页面则返回0。            if (!(pag = (char \*) page[p / PAGE\_SIZE]) &&            !(pag = (char \*) page[p / PAGE\_SIZE] =              (unsigned long \*) get\_free\_page ()))          return 0;  // 如果字符串和字符串数组来自内核空间，则设置fs 段寄存器指向内核数据段（ds）。            if (from\_kmem == 2)          set\_fs (new\_fs);            }  // 从fs 段中复制参数字符串中一字节到pag+offset 处。        \*(pag + offset) = get\_fs\_byte (tmp);      }      }  // 如果字符串和字符串数组在内核空间，则恢复fs 段寄存器原值。    if (from\_kmem == 2)      set\_fs (old\_fs);  // 最后，返回参数和环境空间中已复制参数信息的头部偏移值。    return p;  } |

首先p是指向参数和环境空间的最后一个长字处，逻辑地址，如下图所示



首先从最后一个参数开始逆向操作，取fs段中最后一个参数指针到tmp。

然后取字符串长度，注意get\_fs\_byte的\*addr为字符指针指向的值，也就是\_v得到的是字符值一个字节。

最后是从字符串尾部开始逆向复制，注意page数组不是用来映射的，而是保存内存页的地址。而offset是每次循环都会变化。

最终返回p。