漫谈兼容内核之八：
ELF 映像的装入(一)

毛德操

上一篇漫谈中介绍了 Wine 的二进制映像装入和启动，现在我们来看看 ELF 映像的装入和启动。

一般而言，应用软件的编程不可能是“一竿子到底”、所有的代码都自己写的，程序员不可避免地、也许是不自觉地、都会使用一些现成的程序库。对于 C 语言的编程，至少 C 程序库是一定会用到的。从编译/连接和运行的角度看，应用程序和库程序的连接有两种方法。一种是固定的、静态的连接，就是把需要用到的库函数的目标(二进制)代码从程序库中抽取出来，连接进应用软件的目标映像中，或者甚至干脆把整个程序库都连接进应用软件的映像中。这里所谓的连接包括两方面的操作，一是把库函数的目标代码“定位”在应用软件目标映像中的某个位置上。由于不同应用软件本身的大小和结构都可能不同，库函数在目标映像中的位置是无法预先确定的。为此，程序库中的代码必须是可以浮动的，即“与位置无关”的，在编译时必须加上-fPIC 选项，这里 PIC 是“Position-Independent Code”的缩写。一旦一个库函数在映像中的位置确定以后，就要使应用软件中所有对此函数的调用都指向这个函数。早期的软件都采用这种静态的连接方法，好处是连接的过程只发生在编译/连接阶段，而且用到的技术也比较简单。但是也有缺点，那就是具体库函数的代码往往重复出现在许多应用软件的目标映像中，从而造成运行时的资源浪费。另一方面，这也不利于软件的发展，因为即使某个程序库有了更新更好的版本，已经与老版本静态连接的应用软件也享受不到好处，而重新连接往往又不现实。再说，这也不利于将程序库作为商品独立发展的前景。于是就发展起了第二种连接方法，那就是动态连接。所谓动态连接，是指库函数的代码并不进入应用软件的目标映像，应用软件在编译/连接阶段并不完成跟库函数的连接；而是把函数库的映像也交给用户，到启动应用软件目标映像运行时才把程序库的映像也装入用户空间(并加以定位)、再完成应用软件与库函数的连接。说到程序库，最基本、最重要的当然是 C 语言库、即 libc 或 glibc。

这样，就有了两种不同的 ELF 格式映像。一种是静态连接的，在装入/启动其运行时无需装入函数库映像、也无需进行动态连接。另一种是动态连接的，需要在装入/启动其运行时同时装入函数库映像并进行动态连接。显然，Linux 内核应该既支持静态连接的 ELF 映像、也支持动态连接的 ELF 映像。进一步的分析表明：装入/启动 ELF 映像必需由内核完成，而动态连接的实现则既可以在内核中完成，也可在用户空间完成。因此，GNU 把对于动态连接 ELF 映像的支持作了分工：把 ELF 映像的装入/启动放在 Linux 内核中；而把动态连接的实现放在用户空间，并为此提供一个称为“解释器”的工具软件，而解释器的装入/启动也由内核负责。

大家知道，在 Linux 系统中，目标映像的装入/启动是由系统调用 execve()完成的，但是可以在 Linux 内核上运行的二进制映像有 a.out 和 ELF 两种。由于篇幅的关系，在“情景分析”一书中对于二进制映像只讲了 a.out 格式映像的装入/启动，而没有讲 ELF 格式映像的装入/启动。这是因为如果讲了 ELF 映像就不可避免地要讲到动态连接、讲到“解释器”，那样一来篇幅就大了。从对于装入/启动可执行映像的过程的一般了解而言，光讲 a.out 也许就够了；可是考虑到 ELF 映像(以及 Windows 软件的 PE 映像)对于兼容内核开发的重要意义，还是有必要补上这一课。

本文先介绍装入/启动一个 ELF 映像时发生于 Linux 内核中的操作，下一篇漫谈则介绍

发生于用户空间的操作、即“解释器”对于共享库的操作。

1．系统空间的操作

内核中实际执行 execv()或 execve()系统调用的程序是 do_execve()，这个函数先打开目标映像文件，并从目标文件的头部(从第一个字节开始)读入若干(128)字节，然后调用另一个函数 search_binary_handler()，在那里面让各种可执行程序的处理程序前来认领和处理。内核所支持的每种可执行程序都有个 struct linux_binfmt 数据结构，通过向内核登记挂入一个队列。而 search_binary_handler()，则扫描这个队列，让各个数据结构所提供的处理程序、即各种映像格式、逐一前来认领。如果某个格式的处理程序发现特征相符而，便执行该格式映像的装入和启动。

我们从 ELF 格式映像的 linux_binfmt 数据结构开始：

```
#define load_elf_binary load_elf32_binary

static struct linux_binfmt elf_format = {
        .module         = THIS_MODULE,
        .load_binary    = load_elf_binary,
        .load_shlib     = load_elf_library,
        .core_dump      = elf_core_dump,
        .min_coredump       = ELF_EXEC_PAGESIZE
};
```

这个数据结构表明：ELF 格式的二进制映像的认领、装入和启动是由 load_elf_binary()完成的。而“共享库”、即动态连接库映像的装入则由 load_elf_library()完成。实际上共享库的映像也是二进制的，但是一般说“二进制”映像是指带有 main()函数的、可以独立运行并构成一个进程主体的可执行程序的二进制映像。另一方面，尽管装入/启动二进制映像的过程中蕴含了共享库的装入(否则无法运行)，但是在此过程中却并没有调用 load_elf_library()，而是通过别的函数进行，这个函数只是在 sys_uselib()、即系统调用 uselib()中通过函数指针 load_shlib 受到调用。所以，load_elf_library()所处理的是应用软件在运行时对于共享库的动态装入，而不是启动进程时的静态装入。

下面我们就来看 load_elf_binary()代码，这个函数在 fs/binfmt_elf.c 中。由于篇幅的关系，本文只能以近似于伪代码的形式列出经过简化整理的代码(下同)，有需要或兴趣的读者不妨结合源文件中的原始代码阅读。由于 load_elf_binary()是个比较大的函数，我们分段阅读。

```
[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

static int load_elf_binary(struct linux_binprm * bprm, struct pt_regs * regs)
{
    . . . . . .
    struct {
        struct elfhdr elf_ex;
        struct elfhdr interp_elf_ex;
        struct exec interp_ex;
```

```
    } *loc;

    loc = kmalloc(sizeof(*loc), GFP_KERNEL);
    . . . . . .

    /* Get the exec-header */
    loc->elf_ex = *((struct elfhdr *) bprm->buf);

    . . . . . .
    /* First of all, some simple consistency checks */
    if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
        goto out;                   //比对四个字符，必须是 0x7f、‘E’、‘L’、和‘F’。

    if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
        goto out;                   //映像类型必须是 ET_EXEC 或 ET_DYN。
    if (!elf_check_arch(&loc->elf_ex))
        goto out;                   //机器(CPU)类型必须相符。
    . . . . . .
```

首先是认领。ELF 映像文件的头部应该是个 struct elfhdr 数据结构，对于 32 位映像这实际上是 struct elf32_hdr 数据结构、即 Elf32_Ehdr，其定义如下所示：

```
#define elfhdr        elf32_hdr

typedef struct elf32_hdr{
  unsigned char     e_ident[EI_NIDENT];         // EI_NIDENT = 16
  Elf32_Half    e_type;                         //  即 unsigned shout
  Elf32_Half    e_machine;                      //  即  unsigned int
  Elf32_Word  e_version;
  Elf32_Addr  e_entry;    /* Entry point */
  Elf32_Off     e_phoff;
  Elf32_Off     e_shoff;
  Elf32_Word  e_flags;
  Elf32_Half    e_ehsize;
  Elf32_Half    e_phentsize;
  Elf32_Half    e_phnum;
  Elf32_Half    e_shentsize;
  Elf32_Half    e_shnum;
  Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

这个数据结构的前 16 个字节是 ELF 映像的标志 e_ident[ ]，其中开头的 4 个字节就是所谓“Magic Number”，应该是“\177ELF”。除这 4 个字符比对相符以外，还要看映像的类型是否 ET_EXEC 和 ET_DYN 之一；前者表示可执行映像，后者表示共享库(此外还有 ET_REL

和 ET_CORE，分别表示浮动地址模块和 dump 映像)。同时，映像所适用的 CPU 类型(如 x86 或 PPC)也须相符。如果这些条件都满足，就算认领成功，下面就是进一步的处理了。进一步的处理当然需要更多的信息，在 Elf32_Ehdr 中提供了两个指针，或者说两个(文件内的)位移量，即 e_phoff 和 e_shoff。如果非 0 的话，前者指向“程序头(Program Header)”数组的起点；后者指向“区段头(Section Header)”数组的起点。两个数组的大小(元素的个数)分别由 e_phnum 和 e_shnum 提供，而每个数组元素(表项)的大小由 e_phentsize 和 e_shentsize 提供。至于 e_ehsize，则是映像头部本身的大小。还有个值得特别说明的成分是 e_entry，那就是该映像的程序入口，一般是_start()的起点。

人们常常提到二进制代码映像中有所谓“程序段”“数据段”等等，那都属于映像中的“区段”即“Section”。但是区段的种类远远不止这些而有很多，例如“符号表”就是一个区段，再如用于动态连接的信息、用于 Debug 的信息等等，都属于不同的区段。而区段头数组、或曰区段头表，则为映像中的每一个区段都提供一个描述性的数据结构。

而程序头数组或曰程序头表中的每一个表项，则是对一个“部(Segment)”的描述。一个部可以包含若干个区段，也可以只是一个简单的数据结构。整个 ELF 映像就是由文件头、区段头表、程序头表、一定数量的区段、以及一定数量的部构成。而 ELF 映像的装入/启动过程，则就是在各种头部信息的指引下将某些部或区段装入一个进程的用户空间，并为其运行做好准备(例如装入所需的共享库)，最后(在目标进程首次受调度运行时)让 CPU 进入其程序入口的过程。读者将会看到，这个过程很可能是嵌套的，因为在装入一个映像的过程中很可能需要装入另一个或另几个别的映像。

我们继续往下看：

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
/* Now read in all of the header information */
. . . . . .
size = loc->elf_ex.e_phnum * sizeof(struct elf_phdr);
retval = -ENOMEM;
elf_phdata = (struct elf_phdr *) kmalloc(size, GFP_KERNEL);
if (!elf_phdata)
        goto out;

retval = kernel_read(bprm->file, loc->elf_ex.e_phoff, (char *) elf_phdata, size);
. . . . . .
files = current->files;              /* Refcounted so ok */
. . . . . .
retval = get_unused_fd();
. . . . . .
get_file(bprm->file);
fd_install(elf_exec_fileno = retval, bprm->file);

elf_ppnt = elf_phdata;
elf_bss = 0;
elf_brk = 0;
```

```
start_code = ~0UL;
end_code = 0;
start_data = 0;
end_data = 0;
```

这里通过 kernel_read()读入的是目标映像的整个程序头表，这是一个 struct elf_phdr、实际上是 struct elf32_phdr 结构数组。这种数据结构的定义为：

```
typedef struct elf32_phdr{
  Elf32_Word      p_type;
  Elf32_Off       p_offset;
  Elf32_Addr      p_vaddr;
  Elf32_Addr      p_paddr;
  Elf32_Word      p_filesz;
  Elf32_Word      p_memsz;
  Elf32_Word      p_flags;
  Elf32_Word      p_align;
} Elf32_Phdr;
```

这里的 p_type 表示部的类型。

同时，这里还为已打开的目标映像文件在当前进程的打开文件表中另外分配一个表项，类似于执行了一次 dup()，目的在于为目标文件维持两个不同的上下文，以便从不同的位置上读出。

接着是对 elf_bss 、elf_brk、start_code、end_code 等等变量的初始化。这些变量分别纪录着当前(到此刻为止)目标映像的 bss 段、代码段、数据段、以及动态分配“堆” 在用户空间的位置。除 start_code 的初始值为 0xffffffff 外，其余均为 0。随着映像内容的装入，这些变量也会逐步得到调整，读者不妨自己留意这些变量在整个过程中的变化。

读入了程序头表，并对 start_code 等变量进行初始化以后，下面的第一步就是在程序头表中寻找“解释器”部、并加以处理的过程。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
for (i = 0; i < loc->elf_ex.e_phnum; i++) {
    if (elf_ppnt->p_type == PT_INTERP) {
        . . . . . .
        retval = -ENOMEM;
        elf_interpreter = (char *) kmalloc(elf_ppnt->p_filesz, GFP_KERNEL);
        . . . . . .
        retval = kernel_read(bprm->file, elf_ppnt->p_offset,
                                        elf_interpreter, elf_ppnt->p_filesz);
        . . . . . .
        interpreter = open_exec(elf_interpreter);
        retval = PTR_ERR(interpreter);
        if (IS_ERR(interpreter))
```

```
                goto out_free_interp;
            retval = kernel_read(interpreter, 0, bprm->buf, BINPRM_BUF_SIZE);
            . . . . . .
            /* Get the exec headers */
            loc->interp_ex = *((struct exec *) bprm->buf);
            loc->interp_elf_ex = *((struct elfhdr *) bprm->buf);
            break;
        }
        elf_ppnt++;
    }
```

显然，这个 for 循环的目的仅在于寻找和处理目标映像的“解释器”部。ELF 格式的二进制映像在装入和启动的过程中需要得到一个工具软件的协助，其主要的目的在于为目标映像建立起跟共享库的动态连接。这个工具称为“解释器”。一个 ELF 映像在装入时需要用什么解释器是在编译/连接是就决定好了的，这信息就保存在映像的“解释器”部中。“解释器”部的类型为 PT_INTERP，找到后就根据其位置 p_offset 和大小 p_filesz 把整个“解释器”部读入缓冲区。整个“解释器”部实际上只是一个字符串，即解释器的文件名，例如“/lib/ld-linux.so.2”。有了解释器的文件名以后，就通过 open_exec()打开这个文件，再通过 kernel_read()读入其开头 128 个字节，这就是映像的头部。早期的解释器映像是 a.out 格式的，现在已经都是 ELF 格式的了，/lib/ld-linux.so.2 就是个 ELF 映像。

下面是对解释器映像头部的处理，首先要确认其为 ELF 格式还是 a.out 格式。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
    . . . . . .
    /* Some simple consistency checks for the interpreter */
    if (elf_interpreter) {
        interpreter_type = INTERPRETER_ELF | INTERPRETER_AOUT;

        /* Now figure out which format our binary is */
        if ((N_MAGIC(loc->interp_ex) != OMAGIC) &&
            (N_MAGIC(loc->interp_ex) != ZMAGIC) &&
            (N_MAGIC(loc->interp_ex) != QMAGIC))
            interpreter_type = INTERPRETER_ELF;

        if (memcmp(loc->interp_elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
            interpreter_type &= ~INTERPRETER_ELF;
        . . . . . .
    } else {
        . . . . . .
    }

    /* OK, we are done with that, now set up the arg stuff,
       and then start this sucker up */
```

至此，我们已为目标映像和解释器映像的装入作好了准备。可以让当前进程(线程)与其父进程分道扬镳，转化成真正意义上的进程，走自己的路了。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
/* Flush all traces of the currently running executable */
retval = flush_old_exec(bprm);
      . . . . . .
/* OK, This is the point of no return */
current->mm->start_data = 0;
current->mm->end_data = 0;
current->mm->end_code = 0;
current->mm->mmap = NULL;
current->flags &= ~PF_FORKNOEXEC;
current->mm->def_flags = def_flags;
. . . . . .
/* Do this so that we can load the interpreter, if need be.  We will
   change some of these later */
retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP), executable_stack);
. . . . . .
```

可想而知，flush_old_exec()把当前进程用户空间的页面都释放了。这么一来，当前进程的用户空间是“一片白茫茫大地真干净”，什么也没有了，原有的物理页面映射都已释放。

现在要来重建用户空间的映射了。一个新的映像要能运行，用户空间堆栈是必须的，所以首先要把用户空间的一个虚拟地址区间划出来用于堆栈。进一步，当 CPU 进入新映像的程序入口时，堆栈上应该有 argc、argv[]、envc、envp[]等参数。这些参数来自老的程序，需要通过堆栈把它们传递给新的映像。实际上，argv[]和 envp[]中是一些字符串指针，光把指针传给新映像，而不把相应的字符串传递给新映像，那是毫无意义的。为此，在进入 search_binary_handler()、从而进入 load_elf_binary()之前，do_execve()已经为这些字符串分配了若干页面，并通过 copy_strings()从用户空间把这些字符串拷贝到了这些页面中。现在则要把这些页面再映射回用户空间(当然是在不同的地址上)，这就是这里 setup_arg_pages()要做的事。这些页面映射的地址是在用户空间堆栈的最顶部。对于 x86 处理器，用户空间堆栈是从 3GB 边界开始向下伸展的，首先就是存放着这些字符串的页面，再往下才是真正意义上的用户空间堆栈。而 argc、argv[]这些参数，则就在这真正意义上的用户空间堆栈上。

下面就可以装入新映像了。所谓“装入”，实际上就是将映像的(部分)内容映射到用户(虚拟地址)空间的某些区间中去。在 MMU 的 swap 机制的作用下，这个过程甚至并不需要真的把映像的内容读入物理页面，而把实际的读入留待将来的缺页中断。

首先装入的是目标映像本身。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
/* Now we do a little grungy work by mmaping the ELF image into
   the correct location in memory.  At this point, we assume that
```

```
        the image should be loaded at fixed address, not at a variable address. */

for(i = 0, elf_ppnt = elf_phdata; i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
    int elf_prot = 0, elf_flags;
    unsigned long k, vaddr;

    if (elf_ppnt->p_type != PT_LOAD)
        continue;
    . . . . . .
    vaddr = elf_ppnt->p_vaddr;
    if (loc->elf_ex.e_type == ET_EXEC || load_addr_set) {
        elf_flags |= MAP_FIXED;
    } else if (loc->elf_ex.e_type == ET_DYN) {
        /* Try and get dynamic programs out of the way of the default mmap
           base, as well as whatever program they might try to exec.  This
           is because the brk will follow the loader, and is not movable.  */
        load_bias = ELF_PAGESTART(ELF_ET_DYN_BASE - vaddr);
    }

    error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt, elf_prot, elf_flags);
    . . . . . .

    if (!load_addr_set) {
        load_addr_set = 1;
        load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
        if (loc->elf_ex.e_type == ET_DYN) {
            load_bias += error -
                        ELF_PAGESTART(load_bias + vaddr);
            load_addr += load_bias;
            reloc_func_desc = load_bias;
        }
    }
    k = elf_ppnt->p_vaddr;
    if (k < start_code) start_code = k;
    if (start_data < k) start_data = k;
     . . . . . .
    k = elf_ppnt->p_vaddr + elf_ppnt->p_filesz;

    if (k > elf_bss)
        elf_bss = k;
    if ((elf_ppnt->p_flags & PF_X) && end_code < k)
        end_code = k;
    if (end_data < k)
        end_data = k;
```

```
        k = elf_ppnt->p_vaddr + elf_ppnt->p_memsz;
        if (k > elf_brk)
                elf_brk = k;
    }                                           //end for() loop

    loc->elf_ex.e_entry += load_bias;
    elf_bss += load_bias;
    elf_brk += load_bias;
    start_code += load_bias;
    end_code += load_bias;
    start_data += load_bias;
    end_data += load_bias;

    /* Calling set_brk effectively mmaps the pages that we need
     * for the bss and break sections.    We must do this before
     * mapping in the interpreter, to make sure it doesn't wind
     * up getting placed where the bss needs to go.
     */
    retval = set_brk(elf_bss, elf_brk);
    . . . . . .
```

还是从目标映像的程序头表中搜索，这一次是寻找类型为 PT_LOAD 的部(Segment)。在二进制映像中，只有类型为 PT_LOAD 的部才是需要装入的。

找到一个 PT_LOAD 片以后，先要确定其装入地址。正如代码前面的注释所述，这里先假定装入地址是固定的，然后再根据映像是否允许浮动而作出调整。具体片头数据结构中的 p_vaddr 提供了映像在连接时确定的装入地址 vaddr。如果映像的类型为 ET_EXEC，(或者 load_addr_set 已经被设置成 1，见下)那么装入地址就是固定的。而若类型为 ET_DYN、即共享库，那么即使装入地址固定也要加上一个偏移量，代码中给出了计算方法，其中 ELF_ET_DYN_BASE 对于 x86 定义为(TASK_SIZE / 3 * 2)，所以这是 2GB 边界，而 ELF_PAGESTART 表示按页面边界对齐。

确定了装入地址以后，就通过 elf_map()、实际上是 elf32_map()、建立用户空间虚存区间与目标映像文件中某个连续区间之间的映射。这个函数基本上就是 do_mmap()，其返回值就是实际映射的(起始)地址。对于类型为 ET_EXEC 的可执行程序映像而言，代码中的 load_bias 是 0，所以装入的起点就是映像自己提供的地址 vaddr。另一方面，对于 ET_EXEC，由于参数中的 elf_flags 中的 MAP_FIXED 标志位为 1，所以给定的映射地址是刚性的而不容许变通，如果与已经映射的区间有冲突就以失败告终。不过，目标映像的映射是从一片空白开始的，所以实际上不可能失败。顺便提一下，现在又多了一种 ELF 格式的目标映像，称为 FDPIC，其装入地址就是可浮动的。

即使总的装入地址是浮动的，一旦装入了第一个 Segment 以后，下一个 Segment 的装入地址就应该是固定的了，所以这里一方面把 load_addr_set 设置成 1，

我们不妨以程序 wine 为例看一下映像的装入。GNU 提供了一个很有用的工具 readelf，可以用来观察各种 ELF 映像的内部结构。我们就用它来看/usr/local/bin/wine 的各种头部。首先是它的 ELF 头部：

ELF Header:

Magic:	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF32
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Intel 80386
Version:	0x1
Entry point address:	0x8048750
Start of program headers:	52 (bytes into file)
Start of section headers:	114904 (bytes into file)
Flags:	0x0
Size of this header:	52 (bytes)
Size of program headers:	32 (bytes)
Number of program headers:	6
Size of section headers:	40 (bytes)
Number of section headers:	36
Section header string table index:	33

可见，这是 EXEC 型的映像，其装入地址是固定的、不可浮动的。这个映像有 6 个程序头、36 个 section 头。我们先看程序头表：

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x011cc 0x011cc R E 0x1000
  LOAD           0x0011cc 0x0804a1cc 0x0804a1cc 0x00158 0x00160 RW  0x1000
  DYNAMIC        0x0011d8 0x0804a1d8 0x0804a1d8 0x000d8 0x000d8 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4
```

所以需要装入的是两个 Segment，从它们在映像中的起始地址和大小可以看出，它们在映像中是连续的。但是，从它们的装入地址却可以看出，装入到用户空间之后它们就分开了。第一个 Segment 的装入地址是 0x08048000，装入以后应该占据 0x08048000-0x080491cc，而第二个 Segment 的装入地址却是 0x0804a1cc。再看区段头表：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
  . . . . . .
```

```
[10] .init             PROGBITS        080485e8 0005e8 000017 00  AX  0   0  4
[11] .plt              PROGBITS        08048600 000600 000150 04  AX  0   0  4
[12] .text             PROGBITS        08048750 000750 0008d8 00  AX  0   0  4
[13] .fini             PROGBITS        08049028 001028 00001b 00  AX  0   0  4
[14] .rodata           PROGBITS        08049060 001060 000166 00   A  0   0 32
[15] .eh_frame         PROGBITS        080491c8 0011c8 000004 00   A  0   0  4
[16] .data             PROGBITS        0804a1cc 0011cc 00000c 00  WA  0   0  4
. . . . . .
[21] .got              PROGBITS        0804a2c4 0012c4 000060 04  WA  0   0  4
[22] .bss              NOBITS          0804a324 001324 000008 00  WA  0   0  4
. . . . . .
[34] .symtab           SYMTAB          00000000 01c678 000890 10     35  5c  4
. . . . . .
```

前面说装入的第一个 Segment 在映像中的位置是 0x0，长度是 0x0011cc。跟区段头表中的信息一对照，就可以知道在第 16 项.data 以前的所有区段都是要装入用户空间的。这里面包括了大家所熟知的.text 即“代码段”。此外，.init、.fini 两个区段也有着特殊的重要性，因为映像的程序入口就在.init 段中，实际上在进入 main()之前的代码都在这里。而从 main()返回之后的代码，包括对 exit()的调用，则在.fini 中。还有一个区段.plt 也十分重要，plt 是“Procedure Linkage Table”的缩写，这就是用来为目标映像跟共享库建立动态连接的。再看第二个 Segment，这是从.data、即“数据段”开始的。第二个 Segment 的长度是 0x00160，所以应该包括.got 和.bss。这里的.got 又是个重要的区段，got 是“Global Offset Table”的缩写，里面纪录着供动态连接的函数在映像中的位置。显然，这对于共享库是必不可少的。所以，除大家所熟知的.text、.data、.bss 等区段以外，映像中还有许多信息都是要装入到用户空间的。这么多的信息给谁用呢？这主要是给“解释器”用的，下一片漫谈我将为读者介绍解释器 ld-linux.so.2。另一方面，映像中还有包括符号表.symtab 在内的许多别的信息，但是因为不在类型为 LOAD 的 Segment 中而不会被装入用户空间。

回到 load_elf_binary()的代码。当程序中的 for 循环结束时，目标映像本身需要装入的内容都已经映射到了用户空间合适的位置上。如果是类型为 ET_DYN 的映像，则 elf_bss 等等变量以及映像的程序入口地址都还需要加上偏移量 load_bias。

现在该装入解释器的映像了，我们再往下看。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
    if (elf_interpreter) {
        if (interpreter_type == INTERPRETER_AOUT)
            elf_entry = load_aout_interp(&loc->interp_ex, interpreter);
        else
            elf_entry = load_elf_interp(&loc->interp_elf_ex, interpreter, &interp_load_addr);
        . . . . . .
        reloc_func_desc = interp_load_addr;

        allow_write_access(interpreter);
        fput(interpreter);
```

```
        kfree(elf_interpreter);
    } else {
        elf_entry = loc->elf_ex.e_entry;
    }
```

.这段程序的逻辑很简单：如果需要装入解释器，并且解释器的映像是 ELF 格式的，就通过 load_elf_interp()装入其映像，并把将来进入用户空间时的入口地址设置成 load_elf_interp()的返回值，那显然是解释器的程序入口。而若不装入解释器，那么这个地址就是目标映像本身的程序入口。

显然，关键的操作是由 load_elf_interp()完成的，所以我们追下去看 load_elf_interp()的代码。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary() > load_elf_interp()]

```
static unsigned long load_elf_interp(struct elfhdr * interp_elf_ex,
                                     struct file * interpreter,    unsigned long *interp_load_addr)
{
    struct elf_phdr *elf_phdata;
    struct elf_phdr *eppnt;
    unsigned long load_addr = 0;
    int load_addr_set = 0;
    unsigned long last_bss = 0, elf_bss = 0;
    unsigned long error = ~0UL;
    int retval, i, size;

    /* First of all, some simple consistency checks */
    if (interp_elf_ex->e_type != ET_EXEC && interp_elf_ex->e_type != ET_DYN)
        goto out;
    . . . . . .

    size = sizeof(struct elf_phdr) * interp_elf_ex->e_phnum;
    . . . . . .
    elf_phdata = (struct elf_phdr *) kmalloc(size, GFP_KERNEL);
    . . . . . .
    retval = kernel_read(interpreter,interp_elf_ex->e_phoff,(char *)elf_phdata,size);
    . . . . . .
    eppnt = elf_phdata;
    for (i=0; i<interp_elf_ex->e_phnum; i++, eppnt++) {
      if (eppnt->p_type == PT_LOAD) {
        . . . . . .
        vaddr = eppnt->p_vaddr;
        if (interp_elf_ex->e_type == ET_EXEC || load_addr_set)
            elf_type |= MAP_FIXED;
```

```
        map_addr = elf_map(interpreter, load_addr + vaddr, eppnt, elf_prot, elf_type);
        error = map_addr;
        if (BAD_ADDR(map_addr))
                goto out_close;

        if (!load_addr_set && interp_elf_ex->e_type == ET_DYN) {
          load_addr = map_addr - ELF_PAGESTART(vaddr);
          load_addr_set = 1;
        }

        /*
          * Check to see if the section's size will overflow the
          * allowed task size. Note that p_filesz must always be
          * <= p_memsize so it is only necessary to check p_memsz.
          */
        k = load_addr + eppnt->p_vaddr;
        if (k > TASK_SIZE || eppnt->p_filesz > eppnt->p_memsz ||
        eppnt->p_memsz > TASK_SIZE || TASK_SIZE - eppnt->p_memsz < k) {
                error = -ENOMEM;
                goto out_close;
        }

        /*
          * Find the end of the file mapping for this phdr, and keep
          * track of the largest address we see for this.
          */
        k = load_addr + eppnt->p_vaddr + eppnt->p_filesz;
        if (k > elf_bss)
        elf_bss = k;

        /*
          * Do the same thing for the memory mapping - between
          * elf_bss and last_bss is the bss section.
          */
        k = load_addr + eppnt->p_memsz + eppnt->p_vaddr;
        if (k > last_bss)
        last_bss = k;
    }           //end if
}           //end for

/*
  * Now fill out the bss section.    First pad the last page up
  * to the page boundary, and then perform a mmap to make sure
  * that there are zero-mapped pages up to and including the
```

```
         * last bss page.
         */
        if (padzero(elf_bss)) {
                error = -EFAULT;
                goto out_close;
        }

        elf_bss = ELF_PAGESTART(elf_bss + ELF_MIN_ALIGN - 1);
                                                * What we have mapped so far */

        /* Map the last of the bss segment */
        if (last_bss > elf_bss) {
                down_write(&current->mm->mmap_sem);
                error = do_brk(elf_bss,    last_bss - elf_bss);
                up_write(&current->mm->mmap_sem);
                if (BAD_ADDR(error))
                        goto out_close;
        }

        *interp_load_addr = load_addr;
        error = ((unsigned long) interp_elf_ex->e_entry) + load_addr;

out_close:
        kfree(elf_phdata);
out:
        return    error;
}
```

代码中的 do_brk()从用户空间分配一段空间。这段代码总体上与前面映射目标映像的那一段相似，就把它留给读者细细研究吧。注意解释器映像的类型一般都是 ET_DYN，所以 load_addr 可能不等于 0。

回到 load_elf_binary()的代码中。

```
[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

        compute_creds(bprm);
        current->flags &= ~PF_FORKNOEXEC;
        create_elf_tables(bprm, &loc->elf_ex, (interpreter_type == INTERPRETER_AOUT),
                        load_addr, interp_load_addr);
        /* N.B. passed_fileno might not be initialized? */
        if (interpreter_type == INTERPRETER_AOUT)
                current->mm->arg_start += strlen(passed_fileno) + 1;
        current->mm->end_code = end_code;
        current->mm->start_code = start_code;
```

```
    current->mm->start_data = start_data;
    current->mm->end_data = end_data;
    current->mm->start_stack = bprm->p;
```

在完成装入，启动用户空间的映像运行之前，还需要为目标映像和解释器准备好一些有关的信息，这些信息包括常规的 argc、argv[]、envc、envp[]、还有一些所谓的“辅助向量(Auxiliary Vector)”。这些信息已经存在于内核中，但是需要把它们复制到用户空间，使它们在 CPU 进入解释器或目标映像的程序入口时出现在用户空间堆栈上。这里的create_elf_tables()就起着这个作用。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary() > create_elf_tables()]

```
static int create_elf_tables(struct linux_binprm *bprm,    struct elfhdr * exec,    int interp_aout,
                             unsigned long load_addr,    unsigned long interp_load_addr)
{
    unsigned long p = bprm->p;
    int argc = bprm->argc;
    int envc = bprm->envc;
    elf_addr_t __user *argv;
    elf_addr_t __user *envp;
    elf_addr_t __user *sp;
    elf_addr_t __user *u_platform;
    const char *k_platform = ELF_PLATFORM;
    int items;
    elf_addr_t *elf_info;
    int ei_index = 0;
    struct task_struct *tsk = current;

    . . . . . .

    /* Create the ELF interpreter info */
    elf_info = (elf_addr_t *) current->mm->saved_auxv;
#define NEW_AUX_ENT(id, val) \
    do { elf_info[ei_index++] = id; elf_info[ei_index++] = val; } while (0)

    NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
    NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
    NEW_AUX_ENT(AT_CLKTCK, CLOCKS_PER_SEC);
    NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
    NEW_AUX_ENT(AT_PHENT, sizeof (struct elf_phdr));
    NEW_AUX_ENT(AT_PHNUM, exec->e_phnum);
    NEW_AUX_ENT(AT_BASE, interp_load_addr);
    NEW_AUX_ENT(AT_FLAGS, 0);
    NEW_AUX_ENT(AT_ENTRY, exec->e_entry);
```

```
        NEW_AUX_ENT(AT_UID, (elf_addr_t) tsk->uid);
        NEW_AUX_ENT(AT_EUID, (elf_addr_t) tsk->euid);
        NEW_AUX_ENT(AT_GID, (elf_addr_t) tsk->gid);
        NEW_AUX_ENT(AT_EGID, (elf_addr_t) tsk->egid);
        NEW_AUX_ENT(AT_SECURE, (elf_addr_t) security_bprm_secureexec(bprm));
        . . . . . .
        if (bprm->interp_flags & BINPRM_FLAGS_EXECFD) {
            NEW_AUX_ENT(AT_EXECFD, (elf_addr_t) bprm->interp_data);
        }
#undef NEW_AUX_ENT
        /* AT_NULL is zero; clear the rest too */
        memset(&elf_info[ei_index], 0,
               sizeof current->mm->saved_auxv - ei_index * sizeof elf_info[0]);

        /* And advance past the AT_NULL entry.    */
        ei_index += 2;

        sp = STACK_ADD(p, ei_index);          //实际上是(p - ei_index)，因为堆栈向下伸展。

        items = (argc + 1) + (envc + 1);
        if (interp_aout) {
            items += 3; /* a.out interpreters require argv & envp too */
        } else {
            items += 1; /* ELF interpreters only put argc on the stack */
        }
        bprm->p = STACK_ROUND(sp, items);      //计算(sp - items)并与 16 字节边界对齐。

        /* Point sp at the lowest address on the stack */
#ifdef CONFIG_STACK_GROWSUP
        . . . . . .
#else
        sp = (elf_addr_t __user *)bprm->p;
#endif

        /* Now, let's put argc (and argv, envp if appropriate) on the stack */
        if (__put_user(argc, sp++))
            return -EFAULT;
        if (interp_aout) {
            . . . . . .
        } else {
            argv = sp;                        //用户空间堆栈上的 argv[]从这里开始
            envp = argv + argc + 1;           //用户空间堆栈上的 envp[]从这里开始
        }
```

```
    /* Populate argv and envp */
    p = current->mm->arg_end = current->mm->arg_start;
    while (argc-- > 0) {
        size_t len;
        __put_user((elf_addr_t)p, argv++);
        len = strnlen_user((void __user *)p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
            return 0;
        p += len;
    }
    if (__put_user(0, argv))
        return -EFAULT;
    current->mm->arg_end = current->mm->env_start = p;
    while (envc-- > 0) {
        size_t len;
        __put_user((elf_addr_t)p, envp++);
        len = strnlen_user((void __user *)p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
            return 0;
        p += len;
    }
    if (__put_user(0, envp))
        return -EFAULT;
    current->mm->env_end = p;

    /* Put the elf_info on the stack in the right place.    */
    sp = (elf_addr_t __user *)envp + 1;      //用户空间堆栈上的 elf_info[]从这里开始
    if (copy_to_user(sp, elf_info, ei_index * sizeof(elf_addr_t)))
        return -EFAULT;
    return 0;
}
```

这个函数的代码大体上可以分成前后两半。

前一半是准备阶段，特别是对诸多辅助向量的准备。辅助向量是以编号加值的形式成对出现的。例如，AT_PHDR 是个编号，表示目标映像中程序头数组在用户空间的位置(可想而知这是解释器需要的信息)，而(load_addr + exec->e_phoff)是它的值，二者占据相继的两个 32 位长字。同样，AT_PHNUM 是个编号，而 exec->e_phnum 是它的值，余类推。当然，这些编号对于内核和解释器都有着相同的意义。这里先把这些向量准备好在一个数组 elf_info[]中，最后以编号 AT_NULL 即 0 作为数组的结尾。

后一半则是复制阶段，从代码中的注释行“/* Now, let's put argc (and argv, envp if appropriate) on the stack */”开始。这个阶段的目的是把这些信息复制到用户空间，把它们“种”在堆栈上，为解释器和目标映像的运行做好准备。代码中的变量 bprm->p 实质上是个指针，它代表着用户空间的堆栈指针。进入 create_elf_tables()以后，就把 bprm->p 的值赋给了这里的变量 p，所以在这里 p 也代表着用户空间的堆栈指针。注意这里通过__put_user()写入用户

空间堆栈上的 argv[]和 envp[]中的只是一些指针，而相应的字符串则已经由 do_execve()通过 copy_strings()从用户空间拷贝到内核空间的某些页面中，后来这些页面又被映射到了用户空间(新的地址上)，这里写入用户空间 argv[]和 envp[]中的那些指针就是指向各个字符串在用户空间的新的起点。函数 strnlen_user()的作用是获取用户空间字符串的长度。

再回到 load_elf_binary()的代码，剩下的只是“临门一脚”了。

[sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()]

```
        . . . . . .
        start_thread(regs, elf_entry, bprm->p);
        retval = 0;
        . . . . . .
}
```

最后的 start_thread()是个宏操作，其定义如下：

```
#define start_thread(regs, new_eip, new_esp) do {           \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));      \
        set_fs(USER_DS);                        \
        regs->xds = __USER_DS;                          \
        regs->xes = __USER_DS;                          \
        regs->xss = __USER_DS;                          \
        regs->xcs = __USER_CS;                          \
        regs->eip = new_eip;                        \
        regs->esp = new_esp;                        \
} while (0)
```

这几条指令把作为参数传下来的用户空间程序入口和堆栈指针设置到 regs 数据结构中，这个数据结构实际上在系统堆栈中，是在当前进程通过系统调用进入内核时由 SAVE_ALL 形成的，而指向所保存现场的指针 regs 则作为参数传给了 sys_execve()，并逐层传了下来。把所保存现场中的 eip 和 esp 改成了新的地址，就使得 CPU 在返回用户空间时进入新的程序入口。如果有解释器映像存在，那么这就是解释器映像的程序入口，否则就是目标映像的程序入口。那么什么情况下有解释器映像存在，什么情况下没有呢？如果目标映像与各种库的连接是静态连接，因而无需依靠共享库、即动态连接库，那就不需要解释器映像，启动目标映像运行的条件已经具备；否则就一定要有解释器映像存在。现代的二进制映像一般都使用共享库，所以一般都需要有解释器映像。

现在，对于需要动态连接的目标映像，目标映像和解释器映像都已映射到了当前进程的用户空间，并且“井水不犯河水”、同时并存。但是要启动目标映像的运行则条件还不具备。因为还需要装入(映射)某些共享库的映像，并使目标映像与这些共享库映像之间建立起动态连接，而这需要由解释器在用户空间完成，好在启动解释器运行的条件已经具备了。