基于 LINUX 的 PE 可执行文件加载软件

PELoader V1.0

设 计 说 明 书

目录

第1章 引言

1.1 编写目的

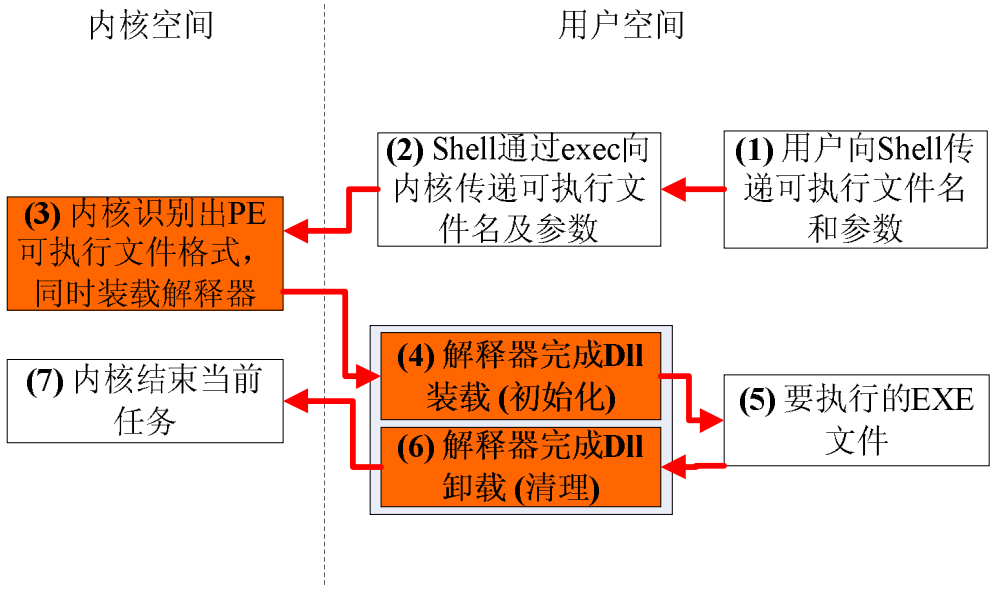
本软件基于 linux 2.6.13 内核开发。其目的是为了在 linux 平台上能够兼容运行 Windows 应用程序。

目前，国内用户大多习惯于使用 Windows 操作系统。为了更好推广 linux 操作系统，打破微软在操作系统领域的垄断地位，不仅具有重大的经济意义，而且具有深远的战略意义。为此，浙大网新毛德操老师提出了“linux 兼容内核”的想法：即改造 linux 内核使之兼容 Windows 的应用程序和驱动程序。本软件作为兼容内核项目的必要组成，使得 linux 内核可以识别 Windows 平台下的 PE 可执行文件格式，并且能够正确执行之。本软件又是兼容内核中相对独立的部分，也可以运行于 linux 其他版本的内核中。

众所周知，国外著名项目 WINE 是 linux 平台上运行 Windows 程序的模拟器。但由于其坚持在用户空间做模拟，因此效率不高。本软件的部分代码需要在 linux 内核空间运行，其执行流程与装载 linux 平台下 elf 文件相似，因此可以保证较快的启动速度。

1.2 系统说明

本软件实现了系统软件中装载器（loader）的完整功能。它由两部份组成：第一部分是内核模块，其功能主要包括 PE 可执行文件的识别，代码段、数据段到内存的映射，解释器的载入等；第二部分为解释器模块，其功能主要包括动态链接库（Dll）的装载与卸载，函数地址的链接，地址的重定位等。当解释器将初始化工作完成以后，将执行流程转交的真正的 EXE 文件入口，EXE 文件执行完成后重新返回解释器，解释器完成清理工作后，向系统发出 exit 系统调用。整个执行流程如下图所示：



本软件实现的总体功能主要包括：

- PE 执行模块的识别、载入和执行
- Dll 的载入和函数动态链接
- 支持 Windows 下三种类型的 Dll
  - 常规 Dll
  - 绑定 Dll
  - 转交 Dll
- 支持 Dll 初始化、清理以及重定位

1.3 术 语

链接器 (linker)

链接各个目标模块为一个整体，使之成为一个可运行文件。

载入器 (loader)

将程序的给定部分从外存加载到内存中，并将它的控制权交给计算机。

解释器

加载 PE 文件依赖的动态链接库，完成外部函数的动态链接。

PE 格式

Win32 平台的可执行文件格式。

Dll

动态链接库。

ELF 格式

Executable and Linking Format，是目前广泛用于类 Unix 系统的可执行文件格式。

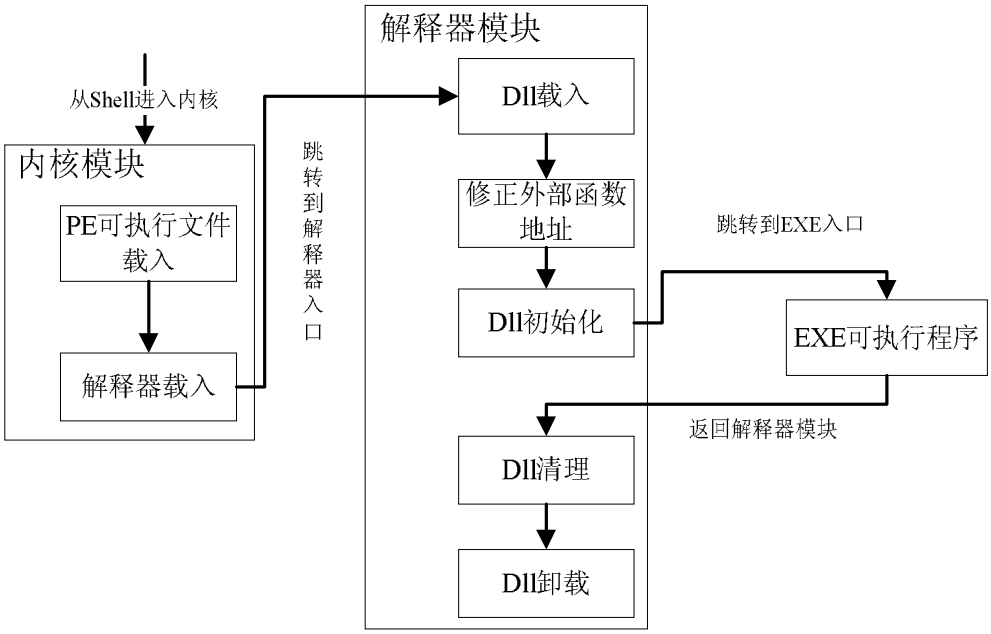
RVA

相对虚拟地址。

第2章 软件结构

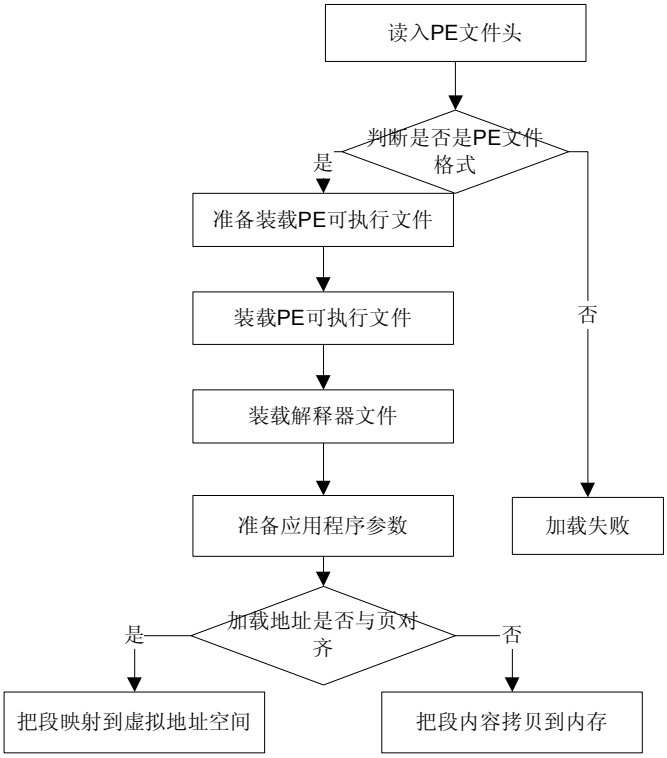
2.1 软件结构图

软件总体流程图如下：

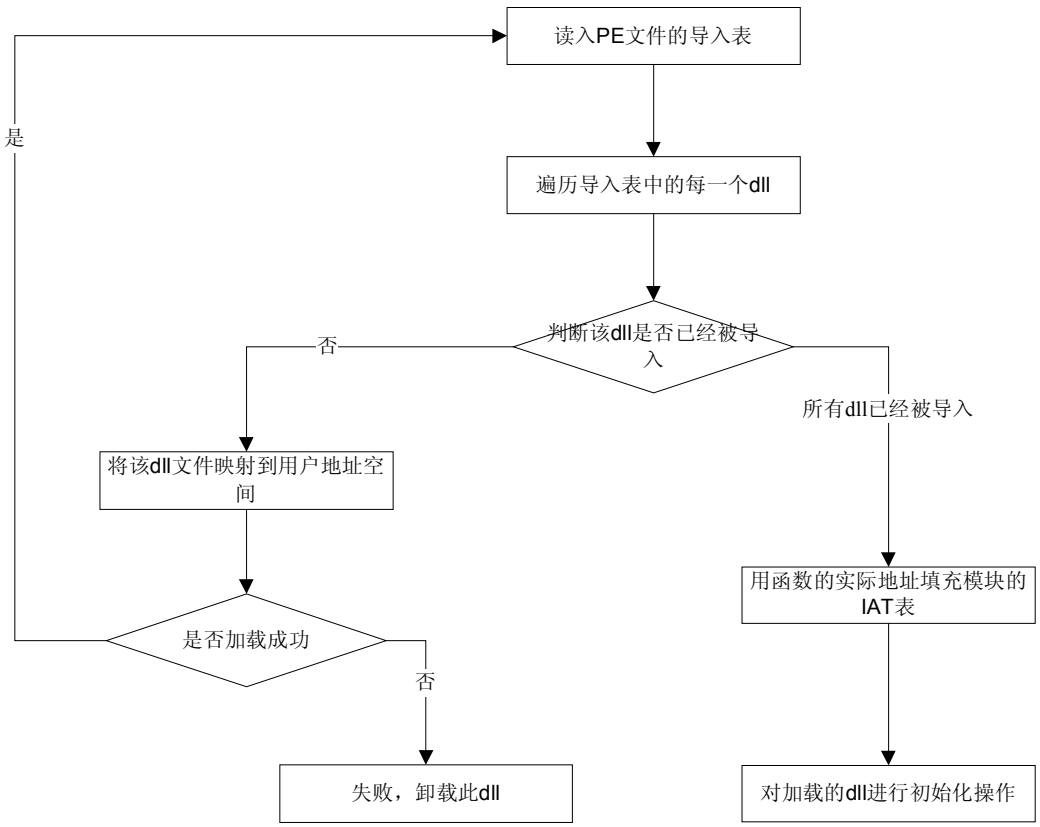


2.2 模块子结构图

2.2.1 内核模块



2.2.2 解释器模块



第3章 模块设计

3.1 内核模块

3.1.1 内核模块总体功能

内核模块主要完成 PE 可执行文件的认领和加载，完成对应的进程的初始化。此外完成解释器模块的加载，最后为应用程序的执行准备参数。把 CPU 执行权交给解释器模块，由解释器模块完成动态链接库的加载，最终跳到可执行文件。

3.1.2 内核模块工作流程

3.1.2.1 PE 可执行文件格式认领

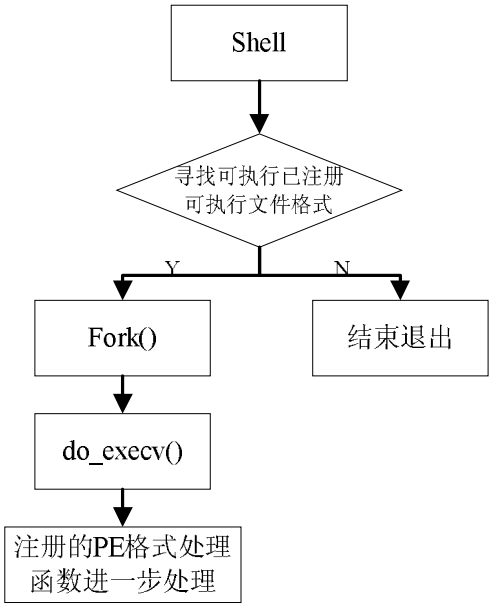
实现如何将 PE 可执行文件交给对应的注册函数以便进行下一步处理。

首先用 init_pe_binfmt 函数把 pe_format 结构注册到内核中。最后用 exit_pe_binfmt 函数

在内核中注销 pe_format 结构。pe_format 结构代码如下：

```
static struct linux_binfmt pe_format = {
    .module         = THIS_MODULE,
    .load_binary    = load_pe_binary,
    ……
};
```

在 linux 内核中有个 format 队列，挂在此队列中的是代表各种可执行文件格式的“代理人”。每个成员只认识并只处理一种特定格式的可执行文件的运行。在实际运行中，在 shell 中执行一个命令后，shell 进程会去寻找相应的可执行文件，若找到了，便 fork 产生一个子进程。随后在子进程代码中，会执行 do_execve 系统调用。在这个系统调用中，系统会安排好子进程执行的环境，随后让 format 队列的各个成员去认领相应的可执行文件。谁要是辨认到了相应的可执行文件格式，便交给它去执行。linux_binfmt 就是 format 队列成员的数据结构。我们要实现 pe 文件格式的加载机制，其实就是要实现 pe 文件格式相应的 linux_binfmt pe_format 对象。在数据结构 linux_binfmt 中，.module = THIS_MODULE 标明本模块，.load_binary=load_pe_binary 表明本模块用来装入可执行程序的函数是 load_pe_binary。.load_shlib = load_pe_library 表明 load_pe_library 用来装入动态链接库。无疑，这里最关键的问题是如何实现 load_pe_binary 函数。



3.1.2.2 PE 文件加载过程

要把 PE 文件加载到内核，就要根据 PE 文件自身的结构特点，对 PE 文件的各个结构进行分析，对各种信息进行提取。借鉴 Linux 系统加载其他文件格式的过程，加载主要由以下几步依次完成。

判断文件是否符合 PE 格式

步骤如下：

A. 首先检验文件头部第一个字的值是否等于 IMAGE_DOS_SIGNATURE，是则 DOS

MZ header 有效。

B. 一旦证明文件的 DOS header 有效后，就可用 e_lfanew 来定位 PE header。

C. 比较 PE header 的第一个字的值是否等于 IMAGE_NT_HEADER。如果前后两个值都匹配，那我们就认为该文件是一个有效的 PE 文件。

内核中为可执行文件的装入定义了一个数据结构 linux_binprm，以便运行一个可执行文件时所需的信息组织在一起。在执行 load_pe_binary 之前，do_execve 会将这个数据结构相应值填好，并通过 bprm 传递给 load_pe_binary 函数。

do_execve 中事先会把可执行文件的头 BINPRM_BUF_SIZ 个字节拷入 char buf[BINPRM_BUF_SIZE]中，这些字节可以充分而且必要的包含了文件的属性。如果 IMAGE_NT_HEADERS 的 signature 域值等于"PE\0\0"，那么就是有效的 PE 文件。

小结一下 PE 文件的识别过程：

在 load_pe_binary 中，首先根据保存在 buf 中的 128 个字节判断是不是 pe 文件格式。具体依据为：buf 头为 IMAGE_DOS_HEADER 数据结构，其成员 e_magic 是否等于 IMAGE_DOS_SIGNATURE，若不是，则返回错误；否则，再取成员 e_lfanew 值。它指向文件中 IMAGE_NT_HEADERS32 数据结构的文件偏移量，这个结构包含了 PE 文件的大部分属性。用 kernel_read 从 IMAGE_NT_HEADERS32 数据结构从文件读入内存，判断其成 Signature 是否等于“PE\0\0"，若不是，也错误返回。接下来，就要根据 IMAGE_NT_HEADERS32 的 FileHeader 的 Characteristics 判断是 PE 可执行文件还是 PE dll。

准备装载 PE 可执行文件

这是 PE 格式文件装载中的核心环节。步骤如下：

A.判断内存能否加载。

首先判断可执行文件加载后会不会超过系统为当前任务所设置的各项资源的限制，如数据段大小，一般不会超过。然后，调用内核提供的 flush_old_exec 函数，来与父进程彻底脱离。包括页表的共享分离，继承的打开文件表和信号量的抛弃等。接着，设置本任务的代码段、数据段的开始、结束地址，扩展段的起始地址等进程内存信息。接下来，就要实际装载文件中的各个段了。

B.如果内存有足够的资源，则为把 PE 文件的代码段、数据段等加载到内存中作准备

C.读入 PE 解释器(是一个 elf 格式的文件)的 elf 头。

装载 PE 可执行文件

装载方式优先采用内存映射的方式，将各个段映射到相对的虚拟地址上。如果不能进行内存映射，则立即预留相应的虚拟地址空间，并将段复制到这块内存上。

首先调用内核函数 kmalloc 在内核中预留好 PE 文件头 image_nt_header 和段表 section table 的大小，并调用 kernel_read 将 PE 文件头和段表拷到内存中。

接着，调用 LdrpMapSections 函数，把所有段映射到虚拟内存中.这又分两步实现:

第一步:先把所有的文件头(包括 MZ 头，DOS Stub，PE 头，section table)映射到内存.由于 PE 可执行文件是第一个映射到虚拟内存的文件，所以不会产生冲突。所以就映射到默认的加载地址，即 ImageBase 字段表示的地址。

第二步:根据每个段头的信息，把每一段映射到对应的虚拟地址。由于段头的 VirtualAddress 字段表示该段相对于 ImageBase 的相对虚拟地址，所以只要把 VirtualAddress 加上 ImageBase 就得到了该段的要加载的虚拟地址。由于 Linux 中地址映射必须以页为单位，那么如果遇到段的大小不是页的整数倍该怎么办呢？我们的解决办法是先用 do_brk()预留地址空间然后调用 kernel_read 把段复制到内存中来。

3.1.2.3 装载解释器文件

装载解释器的任务由 load_pe_interp()来完成。解释器是一个 elf 格式的动态链接文件。elf 文件有以下几部分组成：elf header，program header，section header，section。section 载入内存后叫 segment。elf header 描述了 elf 文件的基本信息，包括 program header 和 section header 的个数，大小和偏移量。program header 描述了 segment 的信息，包括载入的地址和权限等。section header 描述了 section 在链接器链接时用到的信息。

载入解释器文件的过程如下：

先读入 elf header，判断是否是有效的 elf 文件。

然后根据 elf header 中的 program header 的信息（个数，大小，文件中的偏移量），在内核空间中 kmalloc 一块内存，然后读入所有 program header 的信息。(由于这里只是装载 elf 文件，所以不需要用到 section header 的信息)。

接下去，对每一个 program header，根据 segment 的虚拟地址 p_vaddr 把对应的 segment 映射到内存中。

最后，把 bss 段映射到内存中。

3.1.2.4 为 PE 可执行文件准备参数

将可执行文件的各个段装载完后，再将 pe_format 结构挂到当前任务描述符 task_struct 中。最后，调用内核提供的 setup_arg_pages 函数。这个函数会在用户空间的堆栈区顶部为进程建立起一个虚存区间，并将执行参数以及环境变量所占的物理页面与此虚存区间建立起映射。需要注意的是，由于我们的 pe 应用程序是在 windows 环境下编译，而在 windows 环境下，应用程序占用的虚拟内存为 0-2G,之上的 2G-4G 空间为内核使用，这一点与 linux 不同。在 linux 中，用户进程占用的虚拟内存为 0-3G，之上的 3G-4G 空间为内核使用。所以，在调用 setup_arg_pages 函数时，需将传给它的第二个参数指明 0x80000000，表示用户进程的虚拟地址不能超过 0x80000000 这个地址。

前面 setup_arg_pages 为用户空间的堆栈建立起一个虚存空间，并将执行参数以及环境变量所占的物理页面与此虚存区间建立起映射。

在我们的 PE 文件中，用户空间的内存从 0-2G,所以从 2G 往下就是这个虚存区间。这个虚存区间为一个数组，数组中的每个元素都是一个页面，数组的大小为 MAX_ARG_PAGES,而实际映射的页面数量则取决于这些执行参数和环境变量的数量(因为一个参数或一个环境变量的最大可达一个页面)。现在返回到 load_pe_binary 后，就要调用 create_pe_tables 在这部分虚存区间下分配堆栈空间。我们都知道用户程序的 main 函数有两个参数 argc 和 argv[]。其中，argv[]是字符指针数组，argc 为数组的大小。但是实际上还有个隐藏着的字符指针数

组 envp[]用来传递环境变量，只是不在用户的视野之内而已。所以，用户空间堆栈中一开始就要设置好三项数据，即 envp[]、argv[]以及 argc。此外，还要将保存着的（字符串形式的）参数和环境变量复制到用户空间的顶端。这些都是 create_pe_tables 完成的。参数 p 实际上来自于 bprm->p,代表要执行的用户程序的参数和环境变量所占的虚存区间。

3.1.3 内核模块主要函数

3.2 解释器模块

3.2.1 解释器模块总体功能

解释器模块是 elf 格式的动态链接文件，主要用来加载、映射 PE 文件所引入的 dll 文件，解析 PE 文件中引入函数的正确地址，对加载的 dll 文件进行初始化，卸载 dll 文件。首先镜像文件所引入的 dll 文件映射到用户空间，如果被引入的 dll 又存在导入表，会递归的进行这个过程直到所有关联的被映射到用户空间，如果映射失败，则会卸载 dll 文件，返回错误。PE 文件中，有一个专门的数据结构来保存其引入的函数的相关信息，这个数据结构就是 IAT(Imported Address Table)。IAT 是一个 IMAGE_THUNK_DATA 类型的数组。有多少个函数被导入，这个数组就有多少个成员，在 IMAGE_THUNK_DATA 保存每个函数信息，也包括该函数的入口地址。解释器模块根据被引入 dll 模块所加载的地址和 dll 的导出表，可以获得被导入函数的实际入口地址，用这个地址来填充 IAT 表里面的函数地址，等完成 IAT 表中的每一项，这时候 IAT 存放的就是被引入函数的实际地址。接下来对每个引入的 dll 进行初始化工作，完成所有 dll 的初始化后，准备工作就执行好了，接下来跳转到 exe 文件的入口函数地址。

3.2.2 解释器模块工作流程

3.2.2.1 Dll 载入

解释器模块内部用三个链表 InitOrderList，LoadOrderList，MemOrderList 来维护 PE 文件所引入的 DLL 模块，首先对三个链表进行初始化，然后将进程模块作为第一个结点，加入到链表中。接下来就通过函数 LdrpWalkImportDescriptorA 对进程模块的导入表进行遍历，依次加载模块所引入的 DLL 模块，这是一个递归的过程，对于每个模块引入的 DLL 模块解释器通过 LdrpWalkImportDescriptorA 将依次把它们加载进来，直到没有新的 DLL 引入为止，因为所有的 DLL 最终引入的模块为 NTDLL.DLL，所以这个过程是有穷的。对于每个要加载的 DLL，首先通过 LdrpCheckForLoadedDllA 函数，来判断 DLL 是否已经被加载到内存中，对于已经加载的直接跳过，否则通过 LdrpMapDllA 函数来将 DLL 映射到用户空间中，最后将已经映射成功的 DLL 模块插入到 InitOrderList 的尾部。

3.2.2.2 函数地址修正

完成所有的 DLL 映射工作之后，通过 LdrpSnapIAT 函数来根据所有的映射的 DLL 模块，

用所引入函数的实际地址来覆盖 IAT（Import Address Table）每一项的地址。首先得到引入 dll 的 Export Directory 指针，然后修改 IAT 表所在内存的属性，使其可以被读写。然后对于每个转发链和普通的引入表的每项，通过 LdrpSnapThunk 函数来修正引入函数的地址。如果函数是通过序数引入的，则直接得到其序数，否则根据其引入名字来通过函数 LdrpNameToOrdinal 来得到其引入序数。根据引入序数在引入 dll 的 Export Address Table (EAT) 中找到对应的函数的偏移地址，在加上 dll 的基地址则是函数在内存中的实际地址，通过它来修正 IAT。如果引入函数是个转发函数，则解析出转引的 dll 名字，通过 LdrpLoadDllA 函数来加载此 dll，然后通过 LdrpGetProcedureAddress 函数来得到被转引函数的实际地址。

3.2.2.3 Dll 初始化

完成 IAT 表的修正之后，所有引入的函数地址已经是正确的，然后通过调用 LdrpRunInitializeRoutines 函数来初始化每个 dll。首先调用函数 LdrpClearLoadInProgress 清除所有 MODULEITEM 数据结构的 LOAD_IN_PROGRESS 标志，并且得到需要初始化 dll 模块的个数，进程堆上创建一个数组，其成员是将要调用初始化函数的模块所对应的 PLDR_DATA_TABLE_ENTRY 指针，然后遍历 InitOrderList，对于链表中需要初始化的项，调用 dll 的入口点函数。

3.2.2.4 Dll 卸载

在进程退出和 dll 加载发生异常的时候，解释器会调用 LdrpUnloadDll 函数来对 dll 进行卸载。首先确认 dll 已经是被加载进来的，如果模块的引用计数为-1，说明模块为进程模块或 NTDLL.DLL 因为这两个是不可被解释器卸载的，所以直接返回，否则对模块的引用计数递减，如果模块的引用计数为 0 的话，则对三个链表进行遍历，删除这个模块结点。

第4章 主要函数列表

4.1 内核模块

4.1.1 load_pe_binary

函数原型：

int load_pe_binary(struct linux_binprm *bprm, struct pt_regs *regs);

功能描述：

该函数载入 bprm 对应的 PE 可执行文件，如果成功，返回值>=0，否则，返回出错信息。

参数说明：

bprm

[in] 记录了可执行文件的部分信息

regs

[in] 发生系统调用时保存的寄存器的值

返回说明：

如果成功，返回值>=0，否则，返回出错信息。

4.1.2 LdrpMapSections

函数原型：

static int LdrpMapSections(
struct linux_binprm *bprm,
PIMAGE_NT_HEADERS NtHeaders);

功能描述：

把 bprm 所指的 PE 可执行文件映射到虚拟地址空间(不包括 PE 解释器和用到的 dll 文件)。若成功则返回值为 0，失败则返回值大于 0。

参数说明：

bprm

[in] 记录了 PE 可执行文件的部分信息

NtHeaders

[in] PE 可执行文件的 Nt 头

返回说明：

若成功则返回值为 0，失败则返回值大于 0。

4.1.3 load_pe_interp

函数原型：

static unsigned long load_pe_interp(
struct elfhdr *interp_elf_ex,
struct file *interpreter,
unsigned long *interp_load_addr);

功能描述：

把 interpreter 所指的 elf 文件映射到虚拟地址空间。若成功则返回解释器的入口地址，失败则返回出错信息。

参数说明：

interp_elf_ex

[in] 解释器的 elf 头

interpreter

[in] 解释器文件对应的 struct file

interp_load_addr

[out] 解释器文件在虚拟地址空间中的基地址。

返回说明：

若成功则返回解释器的入口地址，失败则返回出错信息。

4.1.4 set_brk

函数原型：

static int set_brk(unsigned long start, unsigned long end);

功能描述：

在虚拟地址空间分配 start 到 end 这一段地址，如果 start 和 end 不是页对齐，就缩小 start，扩大 end，使它们页对齐。若成功则返回 0，失败则返回出错信息。

参数说明：

start

[in] 要分配的起始虚拟地址

end

[in] 要分配的结束虚拟地址

返回说明：

若成功则返回 0，失败则返回出错信息。

4.1.5 padzero

函数原型：

static int padzero(unsigned long elf_bss);

功能描述：

把从 elf_bss 对应的页的起始地址到 elf_bss 这一段内存初始化为 0。

参数说明：

elf_bss

[in] 要初始化的这段地址空间的结束地址

返回说明：

若成功则返回 0，失败则返回出错信息。

4.1.6 elf_map

函数原型：

```
static unsigned long elf_map(
                                struct file *filep,
                                unsigned long addr,
                                struct elf_phdr *eppnt,
                                int prot,
                                int type);
```

功能描述：

把 filep 对应的 elf 文件中的 eppnt 这个 program header 对应的 segment 映射到起始地址为 addr 的虚拟地址空间中。其中该 segment 的权限为 prot,类型为 type。若成功，返回映射的地址，否则，返回出错信息。

参数说明：

filep

[in] 要映射的 elf 文件对应的 struct file 结构体。

addr

[in] 要映射的 segment 的虚拟地址

eppnt

[in] 要映射的 segement 对应的 program header

prot

[in] 要映射的 segement 的权限

type

[in] 要映射的 segement 的类型

返回说明：

实际映射的起始地址。

4.1.7 create_pe_tables

函数原型：

static unsigned long __user *create_pe_tables(struct linux_binprm *bprm);

功能描述：

在 load_pe_exe 中的 setup_arg_pages 为用户空间的堆栈建立起了一个虚拟空间，create_pe_tables 就是在这部分虚拟空间中分配堆栈空间。

参数说明：

bprm

[in] 代表要执行的用户程序的参数和环境变量所占的虚拟空间。

返回说明：

成功返回指向堆栈空间的指针。

4.1.8 init_pe_binfmt

函数原型：

static int init_pe_binfmt(void);

功能描述：

向系统注册 PE 文件格式

参数说明：

无

返回说明：

-EINVAL:

错误的文件格式。

- EBUSY:

系统设备或系统资源忙。

0:

注册文件格式成功。

4.1.9 exit_pe_binfmt

函数原型：

static void exit_pe_binfmt(void);

功能描述：

将 PE 文件格式从系统中删除。

参数说明：

无。

返回说明：

无。

4.2 解释器模块

4.2.1 RtlImageRvaToSection

函数原型：

```
PIMAGE_SECTION_HEADER RtlImageRvaToSection(
                                    IN PIMAGE_NT_HEADERS NtHeaders,
                                    IN PVOID Base,
                                    IN ULONG Rva);
```

功能描述：

该函数查看给定的 Rva 位于哪个 section 当中，并返回对应的 section header。

参数说明：

NtHeaders

[in] 指向 Nt 头的指针

Base

[in] 模块的基底地址

Rva

[in] Rva 值

返回说明：

返回相应的 section header 的地址

4.2.2 RtlImageNtHeader

函数原型：

PIMAGE_NT_HEADERS RtlImageNtHeader(IN PVOID ModuleAddress);

功能描述：

该函数根据模块基地址返回 NT 头的地址。

参数说明：

ModuleAddress

[in] 模块基地址

返回说明：

该函数返回 NT 头的地址

4.2.3 RtlImageRvaToVa

函数原型：

```
PVOID RtlImageRvaToVa(
        IN PIMAGE_NT_HEADERS      NtHeaders,
        IN PVOID                  ModuleBase,
        IN ULONG                  Rva,
        IN OUT PIMAGE_SECTION_HEADER *pLastSection OPTIONAL);
```

功能描述：

将 PE 作为内存映射文件映射到起始地址 ModuleBase，找出给出的 Rva 对应的虚拟地址；如果给出了正确的 Section，将直接使用它，否则也会将将之设为与 Rva 对应的 section header。

参数说明：

NtHeaders

[in] 指向 Nt 头的指针

ModuleBase

[in] 模块基底地址

Rva

[in] Rva 值

pLastSection

[in][out] 指向与 Rva 对应的 section 头的地址

返回说明：

该函数返回与 Rva 对应的虚拟地址

4.2.4 RtlImageDirectoryEntryToData

函数原型：

```
PVOID RtlImageDirectoryEntryToData(
            IN PVOID Base,
            IN BOOLEAN MappedAsImage,
            IN USHORT DirectoryEntry,
            OUT PULONG Size);
```

功能描述：

根据 OptionalHeader 中由 DirectoryEntry 指定的 Directory 给出的 Rva 找到真正的结构数组（例如 Import Descriptor Array）。如果是 Image，则返回 Base+Rva；如果是内存映射文件，则返回 RtlImageRvaToVa(NtHeaders, Base, Rva, NULL)。

参数说明：

Base

[in] Image 或内存映射文件的基底地址

MappedAsImage

[in] 如果为内存映射文件则为 FALSE，如果是 Image 则为 TRUE

DirectoryEntry

[in] PE 文件中的 DirectoryEntry，用来定位 Directory

Size

[out] 返回 Directory 的大小

返回说明：

返回对应 Directory 的起始地址

4.2.5 LdrpLoadDllA

函数原型：

```
NTSTATUS LdrpLoadDllA(IN LPCSTR SearchPath,
                      IN PULONG ResolveDllReferences,
                      IN PSTR DllName,
                      OUT PULONG ImageBase,
                      IN ULONG NormalProcessing);
```

功能描述：

该函数加载一个指定名字的 Dll 模块

参数说明：

SearchPath

[in] 搜索路径

ResolveDllReferences

[in] 该函数被 LoadLibraryExW 调用时，该参数一定为 2

DllName

[in] 要加载的 Dll 的名字

ImageBase

[out] 返回 Dll 加载到的内存地址

NormalProcessing

[in] 如果是正常处理，该参数为 1；如果是转交处理（forwarded processing），该参数为 0，这时该函数将在 LdrpSnapThunk 中被调用

返回说明：

成功返回 STATUS_SUCCESS

4.2.6 LdrpMapDllA

函数原型：

```
NTSTATUS LdrpMapDllA(   IN LPCSTR SearchPath,
                        IN LPCSTR DllName,
                        IN PULONG ResolveDllReferences,
                        IN ULONG StaticLink,
                        IN NTSTATUS MapDllReturnCode,
                        OUT PMODULEITEM * pModuleItem);
```

功能描述：

该函数将 Dll 映射到用户地址空间中

参数说明：

SearchPath

[in] Dll 的搜索路径

DllName

[in] Dll 的名字

ResolveDllReferences

[in] 参见 LdrpLoadDllA 中的说明

StaticLink

[in] 如果该 Dll 被静态链接（即出现在导入表中，而不是通过 LoadLibrary 函数动态链接进来的），则该参数为 TRUE

MapDllReturnCode

[in] 目前没有用到

pModuleItem

[out] 返回对应 MODULEITEM 结构的地址

返回说明：

成功返回 STATUS_SUCCESS

4.2.7 LdrpAllocateDataTableEntry

函数原型：

PMODULEITEM LdrpAllocateDataTableEntry(IN PVOID hDll);

功能描述：

给定指定模块的基地址，分配对应的 MODULEITEM 数据结构。每个进程维护一个 MODULEITEM 结构链表，记录该模块的相关信息，e.g. 加载的基地址、入口点、模块名称等。并做简单的初始化。

参数说明：

hDll

[in] Dll 模块的基底地址

返回说明：

返回为模块分配的 MODULEITEM 结构的地址

4.2.8 LdrpInsertMemoryTableEntry

函数原型：

void LdrpInsertMemoryTableEntry(PMODULEITEM ModuleItem);

功能描述：

将指定的 MODULEITEM 数据结构加入 LoadOrder 链表和 MemOrder 链表。

参数说明：

ModuleItem

[in] 指向 MODULEITEM 数据结构的指针

返回说明：

该函数无返回值

4.2.9 LdrpUpdateLoadCount

函数原型：

void LdrpUpdateLoadCount(PMODULEITEM pModuleItem, ULONG Loading);

功能描述：

该函数 dereference 一个已加载的 Dll，并调整它的引用计数；然后再 dereference 每个被它引用的 Dll。

参数说明：

pModuleItem

[in] 指向 MODULEITEM 结构的指针

Loading

[in] 如果要增加对应 Dll 的引用计数则该参数为 1，否则为 0

返回说明：

该函数无返回值

4.2.10 LdrpCheckForLoadedDllA

函数原型：

```
BOOLEAN LdrpCheckForLoadedDllA(
                        IN LPCSTR SearchPath,
                        IN PSTR DllName,
                        IN ULONG UseLdrpHashTable,
                        IN ULONG ReturnCode,
                        OUT PMODULEITEM *pModuleItem);
```

功能描述：

该函数检查 LoadOrder 链表，查看指定的 Dll 是否已经加载，如果已经加载则返回对应的 MODULEITEM 结构。在 LoadOrder 链表中，具有最小依赖的 Dll 将会出现在最前面。

参数说明：

SearchPath

[in] Dll 的搜索路径

DllName

[in] Dll 的名称

UseLdrpHashTable

[in] 该函数被 LdrpLoadDll 调用时，该参数为 0；被 LdrpUpdateLoadCount 和 LdrpWalkImportDescriptor 调用时，该参数为 1

ReturnCode

[in] 目前没有用到

pModuleItem

[out] 指向 MODULEITEM 数据结构的指针

返回说明：

如果 Dll 已经加载则返回 TRUE，否则返回 FALSE

4.2.11 LdrpUnloadDll

函数原型：

NTSTATUS LdrpUnloadDll(IN PVOID DllHandle);

功能描述：

该函数卸载指定的 Dll

参数说明：

DllHandle

[in] 指定要卸载的 Dll 的句柄，其实就是该 Dll 加载的基底地址

返回说明：

成功返回 STATUS_SUCCESS

4.2.12 LdrpRunInitializeRoutines

函数原型：

NTSTATUS LdrpRunInitializeRoutines(IN DWORD bImplicitLoad OPTIONAL);

功能描述：

该函数在所有 Dll 加载到内存后，按照出现在 LoadOrder 链表中的次序，对每个 Dll 进行初始化

参数说明：

bImplicitLoad

[in] 如果 Dll 被静态链接/隐式链接，该参数为 TRUE，这时需要执行各个 Dll 中的初始化例程

返回说明：

成功返回 STATUS_SUCESS

4.2.13 LdrpClearLoadInProgress

函数原型：

ULONG LdrpClearLoadInProgress();

功能描述：

清除所有 MODULEITEM 数据结构的 LOAD_IN_PROGRESS 标志。该函数仅用于两次不同的计数值更新之间。因为计数值更新会应用到所有依赖的 Dll 上，为了防止循环和重复引用，会在第一次引用时增加 LOAD_IN_PROGRESS 标志，下次遇到已经引用过的 Dll 可以直接跳过，无需对其进行再次引用。

参数说明：

无

返回说明：

返回自上次调用依赖刚刚映射到内存（还没有调用初始化例程）的 Dll 的数目

4.2.14 LdrpCheckForLoadedDllHandle

函数原型：

BOOLEAN LdrpCheckForLoadedDllHandle(
IN HMODULE hDll,
OUT PMODULEITEM *pModuleItem);

功能描述：

该函数扫描 LoadOrder 链表，检查指定的 Dll 是否已经加载内存空间中。如果是，则返回其对应的 MODULEITEM 数据结构。

参数说明：

hDll

[in] 用于指定 Dll

pModuleItem

[out] 返回 Dll 对应的 MODULEITEM 数据结构

返回说明：

如果指定的 Dll 已经加载则返回 TRUE，否则返回 FALSE

4.2.15 LdrpWalkImportDescriptorA

函数原型：

NTSTATUS LdrpWalkImportDescriptorA(
IN PSTR SearchPath OPTIONAL,
IN PMODULEITEM pModuleItem);

功能描述：

该函数是一个递规过程。它遍历指定模块中的导入表，加载指定的每个 Dll

参数说明：

SearchPath

[in] Dll 的搜索路径

pModuleItem

[in] 指定 MODULEITEM 数据结构

返回说明：

成功则返回 STATUS_SUCESS

4.2.16 LdrpNameToOrdinal

函数原型：

USHORT LdrpNameToOrdinal(
IN PSZ Name,
IN ULONG NumberOfNames,
IN PVOID DllBase,
IN PULONG NameTableBase,
IN PUSHORT NameOrdinalTableBase);

功能描述：

根据函数名称找到在其 AddressOfFunction 表中的索引，据此索引可以找到函数的输出地址。

参数说明：

Name

[in] 函数名称

NumberOfNames

[in] Dll 中以名字输出的函数个数

DllBase

[in] Dll 的基底地址

NameTableBase

[in] Dll 中 NameTable 的起始地址

NameOrdinalTableBase

[in] Dll 中 NameOrdinalTable 的起始地址

返回说明：

返回函数名称对应的序号值

4.2.17 LdrpGetProcedureAddress

函数原型：

```
NTSTATUS LdrpGetProcedureAddress(
                    ULONG ForwardedDLLImageBase,
                    LPCSTR APIName,
                    ULONG Ordinal,
                    PULONG pIATEntry,
                    BOOLEAN fRunInitRoutine);
```

功能描述：

该函数在指定的 Dll 中，为指定的函数找到其入口地址

参数说明：

ForwardedDLLImageBase

[in] 指明从哪个 Dll 加载函数地址

APIName

[in] 指定需要定位的函数名称

Ordinal

[in] 指定需要定位的函数序号，如果已指定函数名称，则忽略该参数

pIATEntry

[in] 返回函数的入口地址

fRunInitRoutine

[out] 是否需执行其初始化例程

返回说明：

成功则返回 STATUS_SUCCESS

4.2.18 LdrpSnapThunk

函数原型：

```
NTSTATUS LdrpSnapThunk(
                IN ULONG AdditionalItemImageBase,
                IN ULONG LoadingItemImageBase,
                IN PIMAGE_THUNK_DATA OriginalThunk,
```

IN OUT PIMAGE_THUNK_DATA Thunk,
IN PIMAGE_EXPORT_DIRECTORY pImageExportDirectory,
IN ULONG pExportDirectoryDataSize,
IN ULONG Unknown20,
IN LPSTR ImportedDLLName);

功能描述：

该函数用指定导出表中的数据填充到表需要引用相应函数地址的内存空间中

参数说明：

AdditionalItemImageBase

[in] 指定有输出的模块

LoadingItemImageBase

[in] 指定需要修正函数地址的模块

OriginalThunk

[in] 要修正函数地址模块中的 OriginalTrunk 结构的起始地址

Thunk

[in] 要修正函数地址模块中的 Thunk 结构起始地址，即指向需要修正函数地址的内存空间

pImageExportDirectory

[in] 提供输出表数据

pExportDirectoryDataSize

[in] 输出表的大小

Unknown20

[in] 目前没有用到

ImportedDLLName

[in] 被引入 dll 的名字

返回说明：

成功则返回 STATUS_SUCCESS，否则 STATUS_PROCEDURE_NOT_FOUND

4.2.19 LdrpSnapIAT

函数原型：

NTSTATUS LdrpSnapIAT(
IN PMODULEITEM pAdditionalItem,
IN PMODULEITEM pLoadingItem,
IN PIMAGE_IMPORT_DESCRIPTOR pImageImportDescriptor,
IN BOOLEAN Flag);

功能描述：

该函数填充指定的 Import Address Table (IAT) 表

参数说明：

pAddtionalItem

[in] 指定有函数输出的模块

pLoadingItem

[in] 指定需要 import 函数的模块

pImageImportDescriptor

[in] 指向 IAT 表

Flag

[in] 如果为真，则仅仅填充转发函数地址，如果为假，则填充全部函数地址。

返回说明：

成功则返回 STATUS_SUCCESS

4.2.20 LdrpLoadImportModuleA

函数原型：

```
NTSTATUS LdrpLoadImportModuleA(LPCSTR SearchPath,
                               LPSTR ImportModule,
                               ULONG ImageBase,
                               PMODULEITEM *pModuleItem,
                               BOOLEAN *AlreadLoaded);
```

功能描述：

引入（Import）一个 Dll。该过程大致等同与 LdrpLoadDllA，但又有所不同，该函数用于引入一个隐式链接的 Dll，而 LdrpLoadDllA 则动态加载一个显示的 Dll。

参数说明：

SearchPath

[in] Dll 搜索路径

ImportModule

[in] 需要引入的模块

ImageBase

[in] 模块加载的基地址

pModuleItem

[in] 用来保存模块信息的结构体

AlreadyLoaded

[in] 是否已经加载

返回说明：

成功则返回 STATUS_SUCCESS