|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **分析ELF的加载过程** |  | |
|  |

ELF的可执行文件与共享库在结构上非常类似,它们具有一张**程序段表**,用来描述这些段如何映射到内存。

**对于可执行文件来说,段的加载位置是固定的**,程序段表中如实反映了段的加载地址.**对于共享库来说,段的加载位置是浮动的,位置无关的**,程序段表反映的是以0作为基准地址的相对加载地址.尽管共享库的连接是不充分的,为了便于测试动态链接器,Linux允许直接加载共享库运行.**如果应用程序具有动态链接器的描述段,内核在完成程序段加载后,紧接着加载动态链接器,并且启动动态链接器的�肟�.**

typedef struct elf32\_hdr{

unsigned char e\_ident[EI\_NIDENT];

Elf32\_Half e\_type; /\* ET\_EXEC ET\_DYN 等 \*/

Elf32\_Half e\_machine;

Elf32\_Word e\_version;

Elf32\_Addr e\_entry; /\* Entry point \*/

Elf32\_Off e\_phoff; 程序段描述表的位置

Elf32\_Off e\_shoff; 一般段描述表的位置

Elf32\_Word e\_flags;

Elf32\_Half e\_ehsize; ELF头的大小

Elf32\_Half e\_phentsize; 程序段描述表单元的大小

Elf32\_Half e\_phnum; 程序段描述表单元的个数

Elf32\_Half e\_shentsize; 一般段描述表的单元大小

Elf32\_Half e\_shnum; 一般段描述表单元的个数

Elf32\_Half e\_shstrndx;

} Elf32\_Ehdr;

typedef struct elf32\_phdr{

Elf32\_Word p\_type; PT\_INTERP,PT\_LOAD,PT\_DYNAMIC等

Elf32\_Off p\_offset; 该程序段在ELF文件中的位置

Elf32\_Addr p\_vaddr; 该程序段被映射到进程的虚拟地址

Elf32\_Addr p\_paddr;

Elf32\_Word p\_filesz; 该程序段的文件尺寸

Elf32\_Word p\_memsz; 该程序段的内存尺寸

Elf32\_Word p\_flags; 该程序段的映射属性

Elf32\_Word p\_align;

} Elf32\_Phdr;

struct linux\_binprm

{

char buf[BINPRM\_BUF\_SIZE]; 预先读入的ELF文件头

struct page \*page[MAX\_ARG\_PAGES];

unsigned long p; /\* current top of mem \*/

int sh\_bang;

struct file \* file;

int e\_uid, e\_gid;

kernel\_cap\_t cap\_inheritable, cap\_permitted, cap\_effective;

int argc, envc;

char \* filename; /\* Name of binary \*/

unsigned long loader, exec;

};

static int load\_elf\_binary(struct linux\_binprm \* bprm, struct pt\_regs \* regs)

{

struct file \*interpreter = NULL; /\* to shut gcc up \*/

unsigned long load\_addr = 0, load\_bias;

int load\_addr\_set = 0;

char \* elf\_interpreter = NULL;

unsigned int interpreter\_type = INTERPRETER\_NONE;

unsigned char ibcs2\_interpreter = 0;

mm\_segment\_t old\_fs;

unsigned long error;

struct elf\_phdr \* elf\_ppnt, \*elf\_phdata;

unsigned long elf\_bss, k, elf\_brk;

int elf\_exec\_fileno;

int retval, size, i;

unsigned long elf\_entry, interp\_load\_addr = 0;

unsigned long start\_code, end\_code, start\_data, end\_data;

struct elfhdr elf\_ex;

struct elfhdr interp\_elf\_ex;

struct exec interp\_ex;

char passed\_fileno[6];

/\* Get the exec-header \*/

elf\_ex = \*((struct elfhdr \*) bprm->buf);

retval = -ENOEXEC;

/\* First of all, some simple consistency checks \*/

if (memcmp(elf\_ex.e\_ident, ELFMAG, SELFMAG) != 0)

goto out; 文件头标记是否匹配

if (elf\_ex.e\_type != ET\_EXEC && elf\_ex.e\_type != ET\_DYN)

goto out; 文件类型是否为可执行文件或共享库

if (!elf\_check\_arch(&elf\_ex))

goto out;

if (!bprm->file->f\_op||!bprm->file->f\_op->mmap)

goto out; 所在的文件系统是否具有文件映射功能

/\* Now read in all of the header information \*/

retval = -ENOMEM;

size = elf\_ex.e\_phentsize \* elf\_ex.e\_phnum; 求程序段表总长度

if (size > 65536)

goto out;

elf\_phdata = (struct elf\_phdr \*) kmalloc(size, GFP\_KERNEL); 分配程序段表空间

if (!elf\_phdata)

goto out;

; 读入程序段表

retval = kernel\_read(bprm->file, elf\_ex.e\_phoff, (char \*) elf\_phdata, size);

if (retval < 0)

goto out\_free\_ph;

retval = get\_unused\_fd(); 取可用进程文件表的自由槽位

if (retval < 0)

goto out\_free\_ph;

get\_file(bprm->file);

fd\_install(elf\_exec\_fileno = retval, bprm->file); 将打开的文件安装到进程文件表

elf\_ppnt = elf\_phdata; 指向程序段表

elf\_bss = 0; bss段的起始地址

elf\_brk = 0; bss段的终止地址

start\_code = ~0UL; 代码段的开始

end\_code = 0; 代码段的终止

start\_data = 0; 数据段的开始

end\_data = 0; 数据段的终止

; 扫描ELF程序段表,搜寻动态链接器定义

for (i = 0; i < elf\_ex.e\_phnum; i++) {

if (elf\_ppnt->p\_type == PT\_INTERP) {

retval = -EINVAL;

if (elf\_interpreter)

goto out\_free\_dentry; 如果包含多个动态链接器描述项

/\* This is the program interpreter used for

\* shared libraries - for now assume that this

\* is an a.out format binary

\*/

retval = -ENOMEM; 为动态链接器名称字符串分配空间

elf\_interpreter = (char \*) kmalloc(elf\_ppnt->p\_filesz,

GFP\_KERNEL);

if (!elf\_interpreter)

goto out\_free\_file;

; 将动态链接器的文件名读入内存

retval = kernel\_read(bprm->file, elf\_ppnt->p\_offset,

elf\_interpreter,

elf\_ppnt->p\_filesz);

if (retval < 0)

goto out\_free\_interp;

/\* If the program interpreter is one of these two,

\* then assume an iBCS2 image. Otherwise assume

\* a native linux image.

\*/

if (strcmp(elf\_interpreter,"/usr/lib/libc.so.1") == 0 ||

strcmp(elf\_interpreter,"/usr/lib/ld.so.1") == 0)

ibcs2\_interpreter = 1; 说明应用程序是IBCS2仿真代码

interpreter = open\_exec(elf\_interpreter); 打开动态链接器文件

retval = PTR\_ERR(interpreter);

if (IS\_ERR(interpreter))

goto out\_free\_interp;

retval = kernel\_read(interpreter, 0, bprm->buf, BINPRM\_BUF\_SIZE);

; 读入动态链接器文件头

if (retval < 0)

goto out\_free\_dentry;

/\* Get the exec headers \*/

interp\_ex = \*((struct exec \*) bprm->buf); 假定为a.out格式的文件头结构

interp\_elf\_ex = \*((struct elfhdr \*) bprm->buf); 假定为ELF文件头结构

}

elf\_ppnt++; 下一片段目录项

}

/\* Some simple consistency checks for the interpreter \*/

if (elf\_interpreter) {

; 如果定义了动态链接器,分析其格式类型

interpreter\_type = INTERPRETER\_ELF | INTERPRETER\_AOUT;

/\* Now figure out which format our binary is \*/

if ((N\_MAGIC(interp\_ex) != OMAGIC) &&

(N\_MAGIC(interp\_ex) != ZMAGIC) &&

(N\_MAGIC(interp\_ex) != QMAGIC))

interpreter\_type = INTERPRETER\_ELF; 如果不是AOUT标识

if (memcmp(interp\_elf\_ex.e\_ident, ELFMAG, SELFMAG) != 0)

interpreter\_type &= ~INTERPRETER\_ELF; 如果没有ELF标识

retval = -ELIBBAD;

if (!interpreter\_type) 不能识别动态链接器类型

goto out\_free\_dentry;

/\* Make sure only one type was selected \*/

if ((interpreter\_type & INTERPRETER\_ELF) &&

interpreter\_type != INTERPRETER\_ELF) {

printk(KERN\_WARNING "ELF: Ambiguous type, using ELF\n");

interpreter\_type = INTERPRETER\_ELF;

}

}

/\* OK, we are done with that, now set up the arg stuff,

and then start this sucker up \*/

if (!bprm->sh\_bang) {

char \* passed\_p;

if (interpreter\_type == INTERPRETER\_AOUT) {

sprintf(passed\_fileno, "%d", elf\_exec\_fileno);

passed\_p = passed\_fileno;

if (elf\_interpreter) {

retval = copy\_strings\_kernel(1,&passed\_p,bprm);

; 将程序的文件描述符压入参数堆栈,准备传递给aout格式的动态链接器

if (retval)

goto out\_free\_dentry;

bprm->argc++; bprm->page[]中参数的数目

}

}

}

/\* Flush all traces of the currently running executable \*/

retval = flush\_old\_exec(bprm);

if (retval)

goto out\_free\_dentry;

/\* 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;

elf\_entry = (unsigned long) elf\_ex.e\_entry; 应用程序的入口地址

/\* Do this immediately, since STACK\_TOP as used in setup\_arg\_pages

may depend on the personality. \*/

SET\_PERSONALITY(elf\_ex, ibcs2\_interpreter);

; 如果是ibcs2\_interpreter非0,置PER\_SVR4代码个性,否则置PER\_LINUX代码个性

/\* Do this so that we can load the interpreter, if need be. We will

change some of these later \*/

current->mm->rss = 0;

setup\_arg\_pages(bprm); /\* XXX: check error \*/

; 建立描述堆栈参数页的初始虚存范围结构

current->mm->start\_stack = bprm->p;

/\* 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\_ex.e\_type==ET\_DYN ? ELF\_ET\_DYN\_BASE : 0);

; 如果需要加载的是共享库,则设置共享库的加载偏置为ELF\_ET\_DYN\_BASE

/\* 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. \*/

old\_fs = get\_fs();

set\_fs(get\_ds());

; 再次扫描程序段描述表,映射其中的程序段到进程空间

for(i = 0, elf\_ppnt = elf\_phdata; i < elf\_ex.e\_phnum; i++, elf\_ppnt++) {

int elf\_prot = 0, elf\_flags;

unsigned long vaddr;

if (elf\_ppnt->p\_type != PT\_LOAD)

continue; 如果程序段不可加载

if (elf\_ppnt->p\_flags & PF\_R) elf\_prot |= PROT\_READ;

if (elf\_ppnt->p\_flags & PF\_W) elf\_prot |= PROT\_WRITE;

if (elf\_ppnt->p\_flags & PF\_X) elf\_prot |= PROT\_EXEC;

elf\_flags = MAP\_PRIVATE|MAP\_DENYWRITE|MAP\_EXECUTABLE;

; 根据程序段描述设置相应的mmap参数

vaddr = elf\_ppnt->p\_vaddr; 段起始地址

if (elf\_ex.e\_type == ET\_EXEC || load\_addr\_set) {

; 对于可执行程序,使用固定映射

elf\_flags |= MAP\_FIXED;

}

error = elf\_map(bprm->file, load\_bias + vaddr, elf\_ppnt, elf\_prot, elf\_flags);

; 将该程序段[eppnt->p\_offset,eppnt->p\_filesz]映射到虚存(load\_bias+vaddr)开始的区域

if (!load\_addr\_set) {

load\_addr\_set = 1;

load\_addr = (elf\_ppnt->p\_vaddr - elf\_ppnt->p\_offset);

; 求出该ELF文件在用户虚存中的起始地址

if (elf\_ex.e\_type == ET\_DYN) {

load\_bias += error -

ELF\_PAGESTART(load\_bias + vaddr);

load\_addr += error;

}

}

k = elf\_ppnt->p\_vaddr;

if (k < start\_code) start\_code = k; 取最小的段地址作为代码段起始

if (start\_data < k) start\_data = k; 取最大的段地址作为数据段起始

k = elf\_ppnt->p\_vaddr + elf\_ppnt->p\_filesz; 这时k指向文件段尾

if (k > elf\_bss)

elf\_bss = k; 取最大文件段尾作为BSS段起始

if ((elf\_ppnt->p\_flags & PF\_X) && end\_code < k)

end\_code = k; 取最大可执行的文件段尾作为可执行段的终止

if (end\_data < k)

end\_data = k; 取最大的文件段尾作为数据段的终止

k = elf\_ppnt->p\_vaddr + elf\_ppnt->p\_memsz; 这时k指向内存段尾

if (k > elf\_brk)

elf\_brk = k; 取最大的内存段尾作为BSS段的终止

}

set\_fs(old\_fs);

elf\_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;

if (elf\_interpreter) {

if (interpreter\_type == INTERPRETER\_AOUT)

elf\_entry = load\_aout\_interp(&interp\_ex,

interpreter);

else

elf\_entry = load\_elf\_interp(&interp\_elf\_ex, 动态链接器的文件头

interpreter, 动态链接器打开的文件结构

&interp\_load\_addr); 输出链接器的加载地址

; 可执行程序的入口变为动态链接器的入口

allow\_write\_access(interpreter);

fput(interpreter);

kfree(elf\_interpreter);

if (elf\_entry == ~0UL) {

printk(KERN\_ERR "Unable to load interpreter\n");

kfree(elf\_phdata);

send\_sig(SIGSEGV, current, 0);

return 0;

}

}

kfree(elf\_phdata); 释放程序段表

if (interpreter\_type != INTERPRETER\_AOUT)

sys\_close(elf\_exec\_fileno);

set\_binfmt(&elf\_format); 增加ELF内核模块的引用计数

compute\_creds(bprm);

current->flags &= ~PF\_FORKNOEXEC;

bprm->p = (unsigned long) 建立入口函数参数表

create\_elf\_tables((char \*)bprm->p,

bprm->argc,

bprm->envc,

(interpreter\_type == INTERPRETER\_ELF ? &elf\_ex : NULL),

load\_addr, load\_bias,

interp\_load\_addr,

(interpreter\_type == INTERPRETER\_AOUT ? 0 : 1));

/\* N.B. passed\_fileno might not be initialized? \*/

if (interpreter\_type == INTERPRETER\_AOUT)

current->mm->arg\_start += strlen(passed\_fileno) + 1;

current->mm->start\_brk = current->mm->brk = elf\_brk;

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;

/\* Calling set\_brk effectively mmaps the pages that we need

\* for the bss and break sections

\*/

set\_brk(elf\_bss, elf\_brk); 建立bss的虚存映射,elf\_bss是bss的开始,elf\_brk是bss的结束

padzero(elf\_bss); 如果bss不起始于页连界上,说明与data段有重叠,则将该页bss区域清0

#if 0

printk("(start\_brk) %lx\n" , (long) current->mm->start\_brk);

printk("(end\_code) %lx\n" , (long) current->mm->end\_code);

printk("(start\_code) %lx\n" , (long) current->mm->start\_code);

printk("(start\_data) %lx\n" , (long) current->mm->start\_data);

printk("(end\_data) %lx\n" , (long) current->mm->end\_data);

printk("(start\_stack) %lx\n" , (long) current->mm->start\_stack);

printk("(brk) %lx\n" , (long) current->mm->brk);

#endif

if ( current->personality == PER\_SVR4 )

{

/\* Why this, you ask??? Well SVr4 maps page 0 as read-only,

and some applications "depend" upon this behavior.

Since we do not have the power to recompile these, we

emulate the SVr4 behavior. Sigh. \*/

/\* N.B. Shouldn't the size here be PAGE\_SIZE?? \*/

down(&current->mm->mmap\_sem);

error = do\_mmap(NULL, 0, 4096, PROT\_READ | PROT\_EXEC,

MAP\_FIXED | MAP\_PRIVATE, 0);

up(&current->mm->mmap\_sem);

}

#ifdef ELF\_PLAT\_INIT

/\*

\* The ABI may specify that certain registers be set up in special

\* ways (on i386 %edx is the address of a T\_FINI function, for

\* example. This macro performs whatever initialization to

\* the regs structure is required.

\*/

ELF\_PLAT\_INIT(regs);

#endif

start\_thread(regs, elf\_entry, bprm->p); 将返回的eip设为elf\_entry,esp设为bprm->p

if (current->ptrace & PT\_PTRACED)

send\_sig(SIGTRAP, current, 0); 如果进程处于跟踪状态,则生成SIGTRAP信号

retval = 0;

out:

return retval;

/\* error cleanup \*/

out\_free\_dentry:

allow\_write\_access(interpreter);

fput(interpreter);

out\_free\_interp:

if (elf\_interpreter)

kfree(elf\_interpreter);

out\_free\_file:

sys\_close(elf\_exec\_fileno);

out\_free\_ph:

kfree(elf\_phdata);

goto out;

}

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;

if (!elf\_check\_arch(interp\_elf\_ex))

goto out;

if (!interpreter->f\_op || !interpreter->f\_op->mmap)

goto out;

/\*

\* If the size of this structure has changed, then punt, since

\* we will be doing the wrong thing.

\*/

if (interp\_elf\_ex->e\_phentsize != sizeof(struct elf\_phdr))

goto out;

/\* Now read in all of the header information \*/

size = sizeof(struct elf\_phdr) \* interp\_elf\_ex->e\_phnum;

if (size > ELF\_MIN\_ALIGN)

goto out; 如果动态链接器的程序段表的尺寸大于1个页面

elf\_phdata = (struct elf\_phdr \*) kmalloc(size, GFP\_KERNEL);

if (!elf\_phdata)

goto out;

retval = kernel\_read(interpreter,interp\_elf\_ex->e\_phoff,(char \*)elf\_phdata,size);

error = retval;

if (retval < 0)

goto out\_close;

eppnt = elf\_phdata;

for (i=0; ie\_phnum; i++, eppnt++) {

if (eppnt->p\_type == PT\_LOAD) {

int elf\_type = MAP\_PRIVATE | MAP\_DENYWRITE;

int elf\_prot = 0;

unsigned long vaddr = 0;

unsigned long k, map\_addr;

if (eppnt->p\_flags & PF\_R) elf\_prot = PROT\_READ;

if (eppnt->p\_flags & PF\_W) elf\_prot |= PROT\_WRITE;

if (eppnt->p\_flags & PF\_X) elf\_prot |= PROT\_EXEC;

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);

if (!load\_addr\_set && interp\_elf\_ex->e\_type == ET\_DYN) {

load\_addr = map\_addr - ELF\_PAGESTART(vaddr);

load\_addr\_set = 1;

}

/\*

\* 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;

}

}

/\* Now use mmap to map the library into memory. \*/

/\*

\* 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.

\*/

padzero(elf\_bss);

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)

do\_brk(elf\_bss, last\_bss - elf\_bss);

\*interp\_load\_addr = load\_addr;

error = ((unsigned long) interp\_elf\_ex->e\_entry) + load\_addr;

out\_close:

kfree(elf\_phdata);

out:

return error;

}

static inline unsigned long

elf\_map (struct file \*filep, unsigned long addr, struct elf\_phdr \*eppnt, int prot, int  
type)

{

unsigned long map\_addr;

down(&current->mm->mmap\_sem);

map\_addr = do\_mmap(filep, ELF\_PAGESTART(addr),

eppnt->p\_filesz + ELF\_PAGEOFFSET(eppnt->p\_vaddr), prot, type,

eppnt->p\_offset - ELF\_PAGEOFFSET(eppnt->p\_vaddr));

up(&current->mm->mmap\_sem);

return(map\_addr);

}

void set\_binfmt(struct linux\_binfmt \*new)

{

struct linux\_binfmt \*old = current->binfmt;

if (new && new->module)

\_\_MOD\_INC\_USE\_COUNT(new->module);

current->binfmt = new;

if (old && old->module)

\_\_MOD\_DEC\_USE\_COUNT(old->module);

}

static void set\_brk(unsigned long start, unsigned long end)

{

start = ELF\_PAGEALIGN(start);

end = ELF\_PAGEALIGN(end);

if (end <= start)

return;

do\_brk(start, end - start);

}

static void padzero(unsigned long elf\_bss)

{

unsigned long nbyte;

nbyte = ELF\_PAGEOFFSET(elf\_bss);

if (nbyte) {

nbyte = ELF\_MIN\_ALIGN - nbyte;

clear\_user((void \*) elf\_bss, nbyte);

}

}