漫谈兼容内核之六:
二进制映像的类型识别

毛德操

除了某些嵌入式系统之外，一般而言操作系统都有个在创建(或转化成)新进程时如何装入目标程序的二进制映像并启动其运行的问题。由于在计算机技术的发展历史中并没有形成某种单一的、为所有操作系统和编译/连接工具所共同遵循的标准，这个装入/启动的过程就不可避免地呈现出多样性。而且，即使是同一种操作系统，也会在其发展的过程中采用多种不同的目标映像格式和装入机理。而动态连接库技术的出现，则又使这个过程进一步地复杂化了，因为此时需要装入的不仅是目标程序的映像，还有动态连接库的映像，并且还要解决目标程序与具体库函数的动态连接问题。至于这个过程的重要性，那是不言而喻的，要不然操作系统就要么实际上不能做“有用功”，要么失去了通用性和灵活性。

以 Linux 应用软件为例，就既有 a.out 格式，又有 ELF 格式，又支持动态连接库。我在“情景分析”一书中只讲了 a.out 映像的装入和启动，是因为 a.out 相对比较简单，否则篇幅太大。读者也许会问，既然有了更复杂、功能更强的 ELF 格式，为什么还要保留 a.out 格式呢？这当然是为了向后兼容，一种技术一旦被广泛采用以后就不会很快消失。与 Linux 相比，Windows 采用的格式就更多了，因为它还需要支持 DOS 时代的应用软件。

兼容内核既要支持 Linux 和 Windows 两种操作系统的应用软件，这个问题当然就更复杂、难度更大了。幸而 Wine 已经在我们之前以它的方式解决了这个问题，使我们至少有了可以借鉴的榜样。

在讲述 Wine 的软件映像的装入/启动过程之前，我们先考察一下，为了兼容 Windows 软件，Wine 需要支持那一些映像格式，以及如何识别一个映像所属的格式和类型。为此，我们看一下 Wine 的一段代码，这段代码在 dlls/kernel/module.c 中，是在用户空间执行的。

这是一个名为 MODULE_GetBinaryType()的函数，其作用是辨认一个已打开文件所属的映像格式并进而判定其类型，已定义的类型有：

```
enum binary_type
{
    BINARY_UNKNOWN,
    BINARY_PE_EXE,
    BINARY_PE_DLL,
    BINARY_WIN16,
    BINARY_OS216,
    BINARY_DOS,
    BINARY_UNIX_EXE,
    BINARY_UNIX_LIB
};
```

除 BINARY_UNKNOWN 表示无法辨认/判定以外，这里定义了 7 种映像类型。其中 BINARY_PE_EXE 和 BINARY_PE_DLL 是 Windows 的 32 位“PE 格式”映像，前者为目标应用程序，后者为动态连接库 DLL。注意前者是“有源”的主体，可以成为一个进程；而后者是“无源”的库程序，不能独立成为一个进程。BINARY_WIN16 和 BINARY_OS216

则为 16 位 Windows 应用；后者实际上是 OS/2 操作系统的应用程序，但是因为微软和 IBM 曾经紧密合作，所以 Windows 也支持 OS/2 的应用程序。再往下 BINARY_DOS 显然是 DOS 的应用软件，但是 DOS 上的可执行程序有.exe 和.com 两种，这里并未加以区分，其原因留待以后再说。最后是 BINARY_UNIX_EXE 和 BINARY_UNIX_LIB，Linux 是 Unix 的继承者，所以也适用于 Linux 的应用程序和动态连接库。

下面可以看代码了，我们分段阅读。

```
enum binary_type
MODULE_GetBinaryType( HANDLE hfile,   void **res_start,   void **res_end )
{
    union
    {
        struct
        {
            unsigned char magic[4];
            unsigned char ignored[12];
            unsigned short type;
        } elf;
        struct
        {
            unsigned long magic;
            unsigned long cputype;
            unsigned long cpusubtype;
            unsigned long filetype;
        } macho;
        IMAGE_DOS_HEADER mz;
    } header;

    DWORD len;

    /* Seek to the start of the file and read the header information. */
    if (SetFilePointer( hfile, 0, NULL, SEEK_SET ) == -1)
        return BINARY_UNKNOWN;
    if (!ReadFile( hfile, &header, sizeof(header), &len, NULL ) || len != sizeof(header))
        return BINARY_UNKNOWN;
```

无论是 Linux 还是 Windows，在文件系统的目录项中都没有关于文件格式/类型的说明，所以只能采取在文件的实际内容前面加上头部的方法来表明。但是，不同可执行映像的头部结构和大小又各不相同。而且，头部还可能是级连或嵌套的，即先由一级的头部进行大的分类，然后再由二级的头部作进一步的细分。所以这里定义了一个包含几种一级头部结构的 Union。其中的 elf 当然是 Linux 的 ELF 格式映像的头部(但是 a.out 格式不在内，所以也并不完整)；macho 大约是针对 MACH 操作系统的，我们并不关心；而 mz 是 DOS 以及 Windows 格式的一级头部，这是个相对较大的数据结构：

```
typedef struct _IMAGE_DOS_HEADER {
    WORD   e_magic;         /* 00: MZ Header signature */
    WORD   e_cblp;          /* 02: Bytes on last page of file */
    WORD   e_cp;            /* 04: Pages in file */
    WORD   e_crlc;          /* 06: Relocations */
    WORD   e_cparhdr;       /* 08: Size of header in paragraphs */
    WORD   e_minalloc;      /* 0a: Minimum extra paragraphs needed */
    WORD   e_maxalloc;      /* 0c: Maximum extra paragraphs needed */
    WORD   e_ss;            /* 0e: Initial (relative) SS value */
    WORD   e_sp;            /* 10: Initial SP value */
    WORD   e_csum;          /* 12: Checksum */
    WORD   e_ip;            /* 14: Initial IP value */
    WORD   e_cs;            /* 16: Initial (relative) CS value */
    WORD   e_lfarlc;        /* 18: File address of relocation table */
    WORD   e_ovno;          /* 1a: Overlay number */
    WORD   e_res[4];        /* 1c: Reserved words */
    WORD   e_oemid;         /* 24: OEM identifier (for e_oeminfo) */
    WORD   e_oeminfo;       /* 26: OEM information; e_oemid specific */
    WORD   e_res2[10];      /* 28: Reserved words */
    DWORD e_lfanew;         /* 3c: Offset to extended header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

这个数据结构提供了不少信息，都是与 DOS 环境下的目标映像装入/启动密切相关的。例如 e_ss 和 e_sp 就说明了堆栈的位置是预定的(而不是动态分配的)，而 e_ss 的使用又表明目标程序是在“实模式”下运行。如此等等，这里就不详说了。不过，当微软从 DOS 发展到 Windows 和 WinNT 时，仍旧套用了这个数据结构作为其应用程序目标映像的一级头部，而 WinNT 显然是在“保护模式”下运行。所以，这里的许多字段对于 Windows 目标映像实际上已经不再使用。

代码中首先通过类似于 lseek()的 SetFilePointer()把目标文件的读/写指针移到文件的开头，再按上述 Union 的大小读出，这就可以把几种头部、特别是 Linux 和 Windows 目标映像的头部都包含在内了。

下面就来辨认识别：

```
if (!memcmp( header.elf.magic, "\177ELF", 4 ))
{
    /* FIXME: we don't bother to check byte order, architecture, etc. */
    switch(header.elf.type)
    {
    case 2: return BINARY_UNIX_EXE;
    case 3: return BINARY_UNIX_LIB;
    }
    return BINARY_UNKNOWN;
}
```

. . . . . .

先看是否 Linux 的 ELF 格式。ELF 头部的数据结构定义见上，其第一个字段是 4 字节的标识码 magic，也称为“签名”。ELF 格式签名的第一个字节是八进制的‘\177’，即十六进制的‘0x7f’，然后是‘E’、‘L’、‘F’三个字符。ELF 头部的 type 字段则进一步表明映像的性质，目前只定义了两种类型，即 BINARY_UNIX_EXE 和 BINARY_UNIX_LIB。

我们跳过对 macho 头部的辨认，往下看 DOS/Windows 头部的辨认。DOS 头部的签名定义于 include/winnt.h：

```
#define IMAGE_DOS_SIGNATURE         0x5A4D      /* MZ     */
#define IMAGE_OS2_SIGNATURE         0x454E      /* NE     */
#define IMAGE_OS2_SIGNATURE_LE      0x454C      /* LE     */
#define IMAGE_OS2_SIGNATURE_LX      0x584C      /* LX */
#define IMAGE_VXD_SIGNATURE         0x454C      /* LE     */
#define IMAGE_NT_SIGNATURE      .   0x00004550  /* PE00 */
```

数值 0x5A4D 实际上是‘M’、‘Z’两个字符的代码，因为 Intel 的 CPU 芯片采用“Little Ending”，所以次序是反的。注意这里只有 MZ 用于一级头部，其余都用于二级头部。

继续往下看代码：

```
/* Not ELF, try DOS */

if (header.mz.e_magic == IMAGE_DOS_SIGNATURE)
{
    union
    {
        IMAGE_OS2_HEADER os2;
        IMAGE_NT_HEADERS nt;
    } ext_header;

    /* We do have a DOS image so we will now try to seek into
     * the file by the amount indicated by the field
     * "Offset to extended header" and read in the
     * "magic" field information at that location.
     * This will tell us if there is more header information
     * to read or not.
     */
    if (SetFilePointer( hfile, header.mz.e_lfanew, NULL, SEEK_SET ) == -1)
        return BINARY_DOS;
    if (!ReadFile( hfile, &ext_header, sizeof(ext_header), &len, NULL ) || len < 4)
        return BINARY_DOS;

    /* Reading the magic field succeeded so we will try to determine what type it is.*/
    if (!memcmp( &ext_header.nt.Signature, "PE\0\0", 4 ))
```

```
{
    if (len >= sizeof(ext_header.nt.FileHeader))
    {
      if (len < sizeof(ext_header.nt))    /* clear remaining part of header if missing */
            memset( (char *)&ext_header.nt + len, 0, sizeof(ext_header.nt) - len );
      if (res_start) *res_start = (void *)ext_header.nt.OptionalHeader.ImageBase;
      if (res_end) *res_end = (void *)(ext_header.nt.OptionalHeader.ImageBase +
             ext_header.nt.OptionalHeader.SizeOfImage);
      if (ext_header.nt.FileHeader.Characteristics & IMAGE_FILE_DLL)
            return BINARY_PE_DLL;
      return BINARY_PE_EXE;
    }
    return BINARY_DOS;
}
```

如果一级头部的签名是“MZ”，那就是 DOS 一族的目标映像了，Windows 是从 DOS 发展过来的，所以目标映像同属 DOS 一族。进一步的细分要根据二级头部、或曰“扩充”头部才能辨认，所以这里又定义了一个 Union，即 ext_header。这一次的目的是要区分 Windows 和 OS/2 映像。我们在这里只关心 Windows 的目标映像，所以只看 IMAGE_NT_HEADERS 数据结构的定义：

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;          /* "PE"\0\0 */ /* 0x00 */
  IMAGE_FILE_HEADER       FileHeader;         /* 0x04 */
  IMAGE_OPTIONAL_HEADER   OptionalHeader;     /* 0x18 */
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

这个“头部”里面又嵌套着两个头部，它们的数据结构定义都在 winnt.h 中，这里就不一一列举了。不过需要说明，对于可执行映像而言，IMAGE_OPTIONAL_HEADER 可不是“可选”的，反倒是十分重要的，例如里面有个字段是 AddressOfEntryPoint，还有 BaseOfCode 和 BaseOfData；此外还有个可变大小的数组 DataDirectory[ ]；其重要性由此可见一斑。

还要说明，IMAGE_OS2_HEADER 并不如其名称所示那样仅仅是用于 OS/2 软件映像的，实际上也用于一些 16 位 Windows 软件的映像和 DOS 软件的映像。

IMAGE_DOS_HEADER 结构中的最后一个字段 e_lfanew 说明了扩充头部在文件中的位移，所以这一次把读/写指针移到这个位置上。

读入扩充头部以后，首先就检查是否有“PE”格式的签名。如果是，并且头部是完整的，就根据头部中 FileHeader.Characteristics 字段的 IMAGE_FILE_DLL 标志位判定其为.exe 映像还是 DLL 映像。此外，可能还要根据所读入头部提供的信息修正两个全局量 res_start 和 res_end 的数值，不过那与映像类型的识别是无关的。

如果头部签名不是“PE”，那就可能是 OS/2 或其它 Windows/DOS 的可执行映像了，此类映像在扩充头部中的签名是“NE”。我们再往下看。

```
if (!memcmp( &ext_header.os2.ne_magic, "NE", 2 ))
```

```
        {
            /* This is a Windows executable (NE) header.    This can
             * mean either a 16-bit OS/2 or a 16-bit Windows or even a
             * DOS program (running under a DOS extender).    To decide
             * which, we'll have to read the NE header.
             */
            if (len >= sizeof(ext_header.os2))
            {
               switch ( ext_header.os2.ne_exetyp )
               {
               case 1:    return BINARY_OS216;    /* OS/2 */
               case 2:    return BINARY_WIN16;    /* Windows */
               case 3:    return BINARY_DOS;    /* European MS-DOS 4.x */
               case 4:    return BINARY_WIN16; /* Windows 386; FIXME: is this 32bit??? */
               case 5:    return BINARY_DOS;
                                                  /* BOSS, Borland Operating System Services */
               /* other types, e.g. 0 is: "unknown" */
               default:
               return MODULE_Decide_OS2_OldWin(hfile, &header.mz, &ext_header.os2);
               }
            }
            /* Couldn't read header, so abort. */
            return BINARY_DOS;
        }

        /* Unknown extended header, but this file is nonetheless DOS-executable. */
        return BINARY_DOS;
    }

    return BINARY_UNKNOWN;
}
```

显然，IMAGE_OS2_HEADER 头部中的字段 ne_exetyp 进一步说明了具体的映像类型。从这里也可以看出，DOS/Windows 与 OS/2 真的是你中有我、我中有你。除 1-5 以外，还有些类型码和头部特征是用于某些特别老的 OS/2 和 Windows（版本 3.0 以前）目标映像的，那要进一步通过 MODULE_Decide_OS2_OldWin()加以识别，这里就不赘述了，有兴趣的读者可以自己阅读和研究。

最后，MODULE_GetBinaryType()的返回值就是目标映像的类型代码。

有了这个函数，加上有关数据结构的定义，读者完全可以自己写个程序，打印出给定二进制映像文件的映像类型，并进一步打印出该映像的许多特性和参数。在 Wine 代码的 tools/winedump 目录下那些文件中有一些函数，包括 dump_pe_header()、dump_le_header()、dump_ne_header()等等，就是用来打印出各种格式的映像头部。下面不加说明地列出其中 dump_pe_header()的代码，供读者自己阅读，好处是可以从这些代码中看出各个头部(例如 IMAGE_FILE_HEADER 和 IMAGE_OPTIONAL_HEADER)中许多字段的作用和意义：

```
static void	dump_pe_header(void)
{
	const char			*str;
	IMAGE_FILE_HEADER		*fileHeader;
	IMAGE_OPTIONAL_HEADER	*optionalHeader;
	unsigned			i;

	printf("File Header\n");
	fileHeader = &PE_nt_headers->FileHeader;

	printf("	Machine:				%04X (%s)\n",
	    fileHeader->Machine, get_machine_str(fileHeader->Machine));
	printf("	Number of Sections:			%d\n", fileHeader->NumberOfSections);
	printf("	TimeDateStamp:				%08lX (%s) offset %lu\n",
	    fileHeader->TimeDateStamp, get_time_str(fileHeader->TimeDateStamp),
	    Offset(&(fileHeader->TimeDateStamp)));
	printf("	PointerToSymbolTable:			%08lX\n", fileHeader->PointerToSymbolTable);
	printf("	NumberOfSymbols:			%08lX\n", fileHeader->NumberOfSymbols);
	printf("	SizeOfOptionalHeader:			%04X\n", fileHeader->SizeOfOptionalHeader);
	printf("	Characteristics:			%04X\n", fileHeader->Characteristics);
#define	X(f,s)	if (fileHeader->Characteristics & f) printf("		%s\n", s)
	X(IMAGE_FILE_RELOCS_STRIPPED,	"RELOCS_STRIPPED");
	X(IMAGE_FILE_EXECUTABLE_IMAGE,	"EXECUTABLE_IMAGE");
	X(IMAGE_FILE_LINE_NUMS_STRIPPED,	"LINE_NUMS_STRIPPED");
	X(IMAGE_FILE_LOCAL_SYMS_STRIPPED,	"LOCAL_SYMS_STRIPPED");
	X(IMAGE_FILE_16BIT_MACHINE,	"16BIT_MACHINE");
	X(IMAGE_FILE_BYTES_REVERSED_LO,	"BYTES_REVERSED_LO");
	X(IMAGE_FILE_32BIT_MACHINE,	"32BIT_MACHINE");
	X(IMAGE_FILE_DEBUG_STRIPPED,	"DEBUG_STRIPPED");
	X(IMAGE_FILE_SYSTEM,		"SYSTEM");
	X(IMAGE_FILE_DLL,		"DLL");
	X(IMAGE_FILE_BYTES_REVERSED_HI,	"BYTES_REVERSED_HI");
#undef X
	printf("\n");

	/* hope we have the right size */
	printf("Optional Header\n");
	optionalHeader = &PE_nt_headers->OptionalHeader;
	printf("	Magic					0x%-4X		%u\n",
	    optionalHeader->Magic, optionalHeader->Magic);
	printf("	linker version				%u.%02u\n",
	    optionalHeader->MajorLinkerVersion, optionalHeader->MinorLinkerVersion);
	printf("	size of code				0x%-8lx	%lu\n",
```

```
    optionalHeader->SizeOfCode, optionalHeader->SizeOfCode);
printf("    size of initialized data                0x%-8lx         %lu\n",
    optionalHeader->SizeOfInitializedData, optionalHeader->SizeOfInitializedData);
printf("    size of uninitialized data              0x%-8lx         %lu\n",
    optionalHeader->SizeOfUninitializedData, optionalHeader->SizeOfUninitializedData);
printf("    entrypoint RVA                          0x%-8lx         %lu\n",
    optionalHeader->AddressOfEntryPoint, optionalHeader->AddressOfEntryPoint);
printf("    base of code                            0x%-8lx         %lu\n",
    optionalHeader->BaseOfCode, optionalHeader->BaseOfCode);
printf("    base of data                            0x%-8lX         %lu\n",
    optionalHeader->BaseOfData, optionalHeader->BaseOfData);
printf("    image base                              0x%-8lX         %lu\n",
    optionalHeader->ImageBase, optionalHeader->ImageBase);
printf("    section align                           0x%-8lx         %lu\n",
    optionalHeader->SectionAlignment, optionalHeader->SectionAlignment);
printf("    file align                              0x%-8lx         %lu\n",
    optionalHeader->FileAlignment, optionalHeader->FileAlignment);
printf("    required OS version                     %u.%02u\n",
    optionalHeader->MajorOperatingSystemVersion,
    optionalHeader->MinorOperatingSystemVersion);
printf("    image version                           %u.%02u\n",
    optionalHeader->MajorImageVersion, optionalHeader->MinorImageVersion);
printf("    subsystem version                       %u.%02u\n",
    optionalHeader->MajorSubsystemVersion, optionalHeader->MinorSubsystemVersion);
printf("    Win32 Version           0x%lX\n", optionalHeader->Win32VersionValue);
printf("    size of image           0x%-8lx         %lu\n",
    optionalHeader->SizeOfImage, optionalHeader->SizeOfImage);
printf("    size of headers         0x%-8lx         %lu\n",
    optionalHeader->SizeOfHeaders, optionalHeader->SizeOfHeaders);
printf("    checksum                0x%lX\n", optionalHeader->CheckSum);
switch (optionalHeader->Subsystem)
{
default:
case IMAGE_SUBSYSTEM_UNKNOWN:               str = "Unknown";   break;
case IMAGE_SUBSYSTEM_NATIVE:                str = "Native";        break;
case IMAGE_SUBSYSTEM_WINDOWS_GUI:       str = "Windows GUI";    break;
case IMAGE_SUBSYSTEM_WINDOWS_CUI:       str = "Windows CUI";    break;
case IMAGE_SUBSYSTEM_OS2_CUI:               str = "OS/2 CUI";   break;
case IMAGE_SUBSYSTEM_POSIX_CUI:             str = "Posix CUI";  break;
}
printf("    Subsystem               0x%X (%s)\n", optionalHeader->Subsystem, str);
printf("    DLL flags               0x%X\n", optionalHeader->DllCharacteristics);
printf("    stack reserve size      0x%-8lx         %lu\n",
    optionalHeader->SizeOfStackReserve, optionalHeader->SizeOfStackReserve);
```

```
    printf("    stack commit size          0x%-8lx        %lu\n",
        optionalHeader->SizeOfStackCommit, optionalHeader->SizeOfStackCommit);
    printf("    heap reserve size          0x%-8lx        %lu\n",
        optionalHeader->SizeOfHeapReserve, optionalHeader->SizeOfHeapReserve);
    printf("    heap commit size           0x%-8lx        %lu\n",
        optionalHeader->SizeOfHeapCommit, optionalHeader->SizeOfHeapCommit);
    printf("    loader flags               0x%lX\n", optionalHeader->LoaderFlags);
    printf("    RVAs & sizes               0x%lX\n", optionalHeader->NumberOfRvaAndSizes);
    printf("\n");

    printf("Data Directory\n");
    printf("%ld\n",
         optionalHeader->NumberOfRvaAndSizes* sizeof(IMAGE_DATA_DIRECTORY));

    for (i = 0;    i < optionalHeader->NumberOfRvaAndSizes && i < 16;    i++)
    {
    printf("    %-12s    rva: 0x%-8lX    size: %8lu\n",
                DirectoryNames[i],
                optionalHeader->DataDirectory[i].VirtualAddress,
                optionalHeader->DataDirectory[i].Size);
    }
    printf("\n");
}
```

代码中的 PE_nt_headers 是个 PE 格式头部数据结构指针，头部的内容在此以前已经读入。Rva 是“Relative Virtual Address”的缩写，表示一个符号相对于其所在浮动代码块起点的虚拟地址。

现在，读者应该已经大体上明白了如何识别目标映像，以及如何解释映像头部许多字段和标志位的作用和意义，为进一步考察目标映像的装入和启动打下了基础。在以后的漫谈中，我将讲述 Wine 怎样装入和启动目标映像，到那时这些字段的作用和意义就更清楚了。

Wine 的上述代码都是在用户空间执行的，但是若要将这些代码移到内核中也很容易。