==Ph4nt0m Security Team==

Issue 0x02, Phile #0x0A of 0x0A

|=--=|
|=----------------------=[pe/elf 文件加壳时的处理]=----------------------=|
|=--=|
|=--=|
|=-------------------------=[By dummy   ]=-------------------------=|
|=----------------------=[<dummy_at_ph4nt0m.org>]=----------------------=|
|=--=|

前言：

最初的壳是在感染型的病毒技术上发展出来的，加壳目的一般是压缩或加密。本文主要就x86平台下win32 pe和linux elf 加壳程序的实现做简单介绍和总结，以自己以前写相关程序做线索叙述，其中程序源码是开源的，有兴趣的朋友可以继续进行改进。

ps: 其中有些地方很久没碰，可能有地方描述有误，还请见谅:)

正文：

```
-----------------------------------------------------
slm         x86 win32 r3 pe packer
mimisys     x86 win32 r0 pe packer
elfp        x86 linux r3 elf packer
-----------------------------------------------------
```

一、一个壳的组成

一个完整的壳程序主要由 2 个部分组成 packer 和 loader。它们具体的作用分别是：

(1) packer

负责将待加壳程序压缩和加密处理、把loader写到待加壳程序上。以slm的pakcer为例具体操作包括，pe有效性判断、优化可压缩数据、压缩和加密、添加loader、存放加壳参数和待加壳程序原数据（oep等等）、改写入口点等等。

(2) loader

主要工作是解压或解密被加壳的程序，以slm的loader为例具体的操作包括：获取自身位置、获取加壳参数、进行解压或解密、填充导入表、重定位、tls 初始化等等。

二、slm (x86 win32 r3 pe packer)

资料:
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

工具：
```
lordpe    pe 文件格式查看编辑工具
dumpbin    vc 自带coff文件格式查看工具
ollydbg    r3 调试工具
```

源码结构：
```
./slm/cm 公共头文件和模块
./slm/pk    packer 实现
./slm/sc    loader 实现
```

在做这个时候对 pe 也是刚刚了解，所以 slm 很多地方现在看来有些问题:)。在第一节已经简单描述 slm 的工作流程，下面主要就我当初做的时候遇到的问题做一些描述：

(1) 资源的处理

slm 的资源处理做的比较烦琐，当初目的是为了把可压缩资源数据归并到一起，进行一次压缩，不可压缩单独存放。下面简单介绍一下资源的目录数据格式，详细还是看看微软的文档和相关源码:)

从IMAGE_NT_HEADERS.IMAGE_DATA_DIRECTORY[IMAGE_DIRECTORY_ENTRY_RESOURCE]取出资源数据的地址res_rva，经过转换后第一个结构体是IMAGE_RESOURCE_DIRECTORY

IMAGE_RESOURCE_DIRECTORY:

NumberOfIdEntries	目录下 id 名称入口项个数
NumberOfNamedEntries	目录下 name 名称入口项个数

紧跟着IMAGE_RESOURCE_DIRECTORY后面是IMAGE_RESOURCE_DIRECTORY_ENTRY结构数组，这个数组的元素个数是 NumberOfIdEntries + NumberOfNamedEntries。

IMAGE_RESOURCE_DIRECTORY_ENTRY:

Id	目录id，只有NameIsString非真才有效
NameIsString	目录名称是否是字符串，如果为真NameOffset有效
NameOffset	目录名称串的偏移，偏移是相对与res_rva*的。
DataIsDirectory	如果为真 OffsetToData 有效，否则OffsetToDirectory有效
OffsetToData	指向资源数据，偏移类型rva
OffsetToDirectory	指向子目录，偏移类型rva

如果目录入口名称是字符串，通过NameOffset获取PIMAGE_RESOURCE_DIR_STRING_U的结构指针，目录名是unicode格式，并且不是以零结尾的字符串。如果目录名不是字符串而是id，那么其值在winnt.h 定义。常见id有RT_ICON、RT_VERSION等等。

结构大致简单描述完了，有点要注意OffsetToDirectory、OffsetToData修改时要进行 DWORD 对齐，否则会出现奇怪的现象。

(2) 导入表处理

从IMAGE_NT_HEADERS.IMAGE_DATA_DIRECTORY[IMAGE_DIRECTORY_ENTRY_IMPORT]取出导入表的地址imp_rva，经过转换后第一个结构体是IMAGE_IMPORT_DESCRIPTOR。

IMAGE_IMPORT_DESCRIPTOR:

Name	指向导入 dll 的名称，偏移类型 rva
FirstThunk	指向 IMAGE_THUNK_DATA 结构体，偏移类型 rva
OriginalFirstThunk	指向FirstThunk 的副本，可以为空。偏移类型 rva

导入由IMAGE_IMPORT_DESCRIPTOR结构数组组成，数组长度由一个Name域为空的结构表明。

FirstThunk和OriginalFirstThunk都是指向以IMAGE_THUNK_DATA数组组成的数据结构，系统的加载器在进行导入表填充时，会把FirstThunk指向的结构修改掉。

(3) TLS 处理

这里说的tls是所谓的静态tls(在pe文件结构上进行实现)，关于什么是tls可以看看《windows 核心编程》线程那章。

1、tls 是怎样的

比如要在vc中声明一个tls变量需要这样__declspec(thread) int x = 0;在链接时这个变量会被链接器放入.tls的节中。这个节从外边看和其他的节没有什么不同，唯一的区别在IMAGE_DATA_DIRECTORY[IMAGE_DIRECTORY_ENTRY_TLS]指向的一个结构会对此节进行描述，这个结构是IMAGE_TLS_DIRECTORY。

IMAGE_TLS_DIRECTORY:

StartAddressOfRawData	tls数据开始地址类型va
EndAddressOfRawData	tls数据结束地址类型va
AddressOfIndex;	tls slot的地址，默认tls slot为0
AddressOfCallBacks	指向一个PIMAGE_TLS_CALLBACK的数组，这个数组以0结尾，每个PIMAGE_TLS_CALLBACK都是va类型
SizeOfZeroFill	数据区需要进行清 0 数据的大小
Characteristics	

2、系统加载器怎样处理exe的tls

系统加载器在完成重定位和输入表填充后，就开始处理tls。如果存在tls_dir，求出tls数据的大小EndAddressOfRawData - StartAddressOfRawData + SizeOfZeroFill，按照大小分配一块内存，地址存入(PDWORD)fs:[0x2c] + tls_slot，接着拷贝StartAddressOfRawData -> EndAddressOfRawData之间的数据到新分配的内存中，然后使用SizeOfZeroFill 清零剩下的数据，然后进行循环回调AddressOfCallBacks中的函数，PIMAGE_TLS_CALLBACK函数和DllMain原型很像，只是没有返回值。

3、系统加载器怎样处理dll的tls

首先明确dll是可以使用tls的，唯一的不同是AddressOfCallBacks调用方式会有些区别。如果目标dll是被静听链接到其他文件上，在进程创建完成时即被加载，那么他的tls callback会触发，而LoadLibrary方式加载不会触发。

(4) rva & raw 转换

pe 文件中许多结构域的指针类型是rva，rva是pe文件由系统加载后，方法相关数据的相对偏移量。而我们进行加壳处理时，是直接map的文件数据访问都是使用文件指针，这就需要rva进行转换。(每次做pe相关工具时，我都会习惯写一个这样的函数，现在不下10中版本，竟然没有一个可以保证是正确的 - -)

下面这个是最新的rva2raw版本，不保证正确性。

三、mimisys (x86 win32 r0 pe packer)

资料：

Windows Research Kernel

wrk/base/ntos/mm/sysload.c:MmLoadSystemImage

工具:

syser	内核调试器，你也可以选择其他的r0调试器
vmware	如果不想频繁重启，需要一个虚拟机

文件格式的一些处理参考slm，这里主要就r0 pe和r3 pe区别做介绍：

(1) 节和页

r0空间的内存常常很紧张，就导致sys section属性有几个特殊地方

1、可换出和禁止换出

在内存不足时，系统内存管理器，会枚举已加载的section object，如果存在pageout属性，那么系统内存管理器就会换出这个节对应的页（这个节经过系统页对齐后换出内存）节对齐原则VirtualAddress向上、VirtualSize向下。禁止换出如名字所名这个节将永驻内存。

2、节对齐小于一页

大多数sys的节对齐指数都是小于一页，系统加载器在处理这类文件时相当很简单。加载后文件和磁盘上的文件布局基本一致。当节对齐小于一页时，SizeOfRawData必须大于等于VirtualSize，即不支持未初始化节。mimisys通过增加SizeOfImage在文件加载后分配一个未初始化的缓冲区，保证解压过程。

(2) checksum校验

一句话: 只有正确的checksum sys文件才允许被加载。

(3) win2k相关问题

win2k的系统加载器和其他nt系统有几处不同，r3和r0都会有一些区别，比如r3 pe的必须要有导入表，否则拒绝加载，r0 pe必须要有重定位信息，否则也会拒绝加载。这种情况需要构造一个空的重定位目录即可。

mimisys采取的是合并节，导致加壳后只剩下两个节，第一个节是loader, 存放loader和各种加壳参数，第二个节是原程序优化压缩后的数据（移动重定位，移动资源等等）。两个节的属性都是不允许换出的。

四、elfp (x86 linux r3 elf packer)

资料：

Tool Interface Standard (TIS) Executable and Linking Format
http://www.x86.org/ftp/manuals/tools/elf.pdf
毛德操 《漫谈内核兼容》8,9 ELF映像的装入
http://linux.insigma.com.cn/jszl.asp?docid=132762762
http://linux.insigma.com.cn/jszl.asp?docid=133617926
linux 内核源码
linux/fs/binfmt_elf.c:load_elf_binary

工具:

objdump 进行elf文件格式的结构查看
http://www.gnu.org/software/binutils/binutils.html

ald 汇编级调试器，gdb无法调试没有调试信息文件的。
http://ald.sourceforge.net/

elfp是在magiclinux完成的linux elf文件压缩壳。

elf的格式是linux下主要的可执行文件格式，它也是在coff上基础上设计的，所以它和pe文件的格式很相似，下面的叙述过程中会和 pe 文件以对比形式进行描述。

elf文件的第一个数据结构是以Elf32_Ehdr开始

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

e_ident 在 elf.h 中对应的 ELFMAG 宏，长度是四个字节
e_entry 入口点映像偏移(映像偏移即 pe 中所说的 rva)
e_phoff Elf32_Phdr 数组的文件偏移
e_shoff Elf32_Shdr 数组的文件偏移
e_ehsize Elf32_Ehdr 结构的大小
e_phentsize Elf32_Phdr 结构大小
e_phnum Elf32_Phdr 数组成员个数

```
e_shentsize    Elf32_Shdr 结构大小
e_shnum        Elf32_Shdr 数组成员个数
```

在Elf32_Ehdr之后即Elf32_Phdr数组，Elf32_Phdr的地址通过Elf32_Ehdr.e_ehsize来确定。Elf32_Ehdr数组（或叫段表），你可以把phdr看做pe的节表。

```
typedef struct
{
  Elf32_Word    p_type;           /* Segment type */
  Elf32_Off     p_offset;        /* Segment file offset */
  Elf32_Addr    p_vaddr;        /* Segment virtual address */
  Elf32_Addr    p_paddr;        /* Segment physical address */
  Elf32_Word    p_filesz;        /* Segment size in file */
  Elf32_Word    p_memsz;        /* Segment size in memory */
  Elf32_Word    p_flags;        /* Segment flags */
  Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;
```

```
p_type    描述这个段的加载行为属性
p_offset  段数据在文件中偏移，类似pe节中的PointerToRawData
p_vaddr   段数据加载后在映像中偏移，类似pe节中的VirtualAddress
p_filesz  段数据在文件中大小，类型pe节中的SizeOfRawData
p_memsz   段数据加载后在映像中大小，类型pe节中的VirtualSize
p_flags   描述这个段的内存属性，类似pe节中的节属性
p_align   段对齐粒度
```

p_type主要的类型有

```
PT_LOAD      这个段需要装载到内存中
PT_PHDR      这个段存放的是Elf32_Phdr数组
PT_INTERP    这个段存放一个解释器名，请求系统加载器把映像装载需求转给这
               个解释器，关于elf的解释器问题，可以理解为windows下ntdll装载
                 pe文件，elf文件的解释器主要负责重定位、导入表填充等操作
```

p_flags 主要类型有

```
PF_X         这个段可执行
PF_W         这个段可写
PF_R         这个段可读
```

在Elf32_Ehdr(段表)之后便是Elf32_Shdr数组(节表)，你可能到这里很奇怪了，怎么这个叫节表？如果你熟悉pe应该知道节表对pe文件的重要性，但这个可不是pe中的那个节表，你应该把它看做nt header中data_dir[]结构，加载器或调试器等工具会通过节名确定节具体用途，比如存储调试信息、版本信息、字符串表等等、elfp在加壳过程会丢弃节表。

下面简单讲讲elfp的loader处理过程（加壳过程很简单），在一个elf文件被加载后，它的入口点在执行之前，堆栈中会由系统加载器push的一些参数。

```
//  堆栈结构：
//  +-------------------+
//  |   return address  |        返回地址
//  +-------------------+
//  |   argc            |        参数个数
//  +-------------------+
//  |   argv[?], NULL   |        参数表，以 NULL 结尾
//  +-------------------+
//  |   envp[?], NULL   |        环境表，以 NULL 结尾
//  +-------------------+
//  |   auxv[?]         |        中文不知道叫什么它，这个主要是给解释器使用,
//  +-------------------+        存放这个elf的相关信息如果被加壳程序需要解
                                 释器，你需要重写正确填写这个参数，让解释器可
                                       以正确的找到相关数据的地址。
```

elfp壳loader的执行流程大致如下:

申请内存-->把每个段解压到指定的地址上-->获取被加壳程序原始信息-->检查原始段表、重写 auxv-->加载解释器-->调用解释器

关于 elf 的解释器，可以参考资料链接上的文字，那里比我描述的完整。

五、附录

[1] 本文代码
./pstzine_0A_01.zip

-EOF-