0.编写关键是获取API地址并使用。

**导出表中的API：**



**Name**

DLL名字字符串的相对虚拟地址

**NumberOfNames**

通过名字导出的API的个数

**AddressOfFunctions**

指向所有函数的VA数组的相对虚拟机地址。每个VA加上模块基址，能得到一个导出函数的地址。

**AddressOfNames**

指向所有函数名的VA数组的相对虚拟机地址。每个VA加上模块基址，能得到表示API的非0结尾的字符串的地址。

**AddressOfNameOrdinals**

序号数组的相对虚拟地址。每个序号表示一个AddressOfFunctions数组的索引。

**Base是dll的基址，hash是2个CRC-32C的哈希值=** crc32c(API字符串)+ crc32c(DLL字符串)**。**

**下面这个函数能得到api address，但是不能解析：**

LPVOID search\_exp(LPVOID base, DWORD hash)

{

  PIMAGE\_DOS\_HEADER       dos;

  PIMAGE\_NT\_HEADERS       nt;

  DWORD                   cnt, rva, dll\_h;

  PIMAGE\_DATA\_DIRECTORY   dir;

  PIMAGE\_EXPORT\_DIRECTORY exp;

  PDWORD                  adr;

  PDWORD                  sym;

  PWORD                   ord;

  PCHAR                   api, dll;

  LPVOID                  api\_adr=NULL;

  dos = (PIMAGE\_DOS\_HEADER)base;

  nt  = RVA2VA(PIMAGE\_NT\_HEADERS, base, dos->e\_lfanew);

  dir = (PIMAGE\_DATA\_DIRECTORY)nt->OptionalHeader.DataDirectory;

  rva = dir[IMAGE\_DIRECTORY\_ENTRY\_EXPORT].VirtualAddress;

  // if no export table, return NULL

  if (rva==0) return NULL;

  exp = (PIMAGE\_EXPORT\_DIRECTORY) RVA2VA(ULONG\_PTR, base, rva);

  cnt = exp->NumberOfNames;

  // if no api, return NULL

  if (cnt==0) return NULL;

  adr = RVA2VA(PDWORD,base, exp->AddressOfFunctions);

  sym = RVA2VA(PDWORD,base, exp->AddressOfNames);

  ord = RVA2VA(PWORD, base, exp->AddressOfNameOrdinals);

  dll = RVA2VA(PCHAR, base, exp->Name);

  // calculate hash of DLL string

  dll\_h = crc32c(dll);

  do {

    // calculate hash of api string

    api = RVA2VA(PCHAR, base, sym[cnt-1]);

    // add to DLL hash and compare

    if (crc32c(api) + dll\_h == hash) {

      // return address of function

      api\_adr = RVA2VA(LPVOID, base, adr[ord[cnt-1]]);

      return api\_adr;

    }

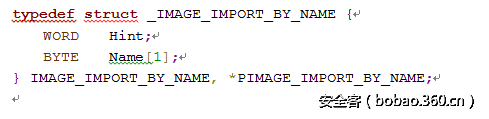
  } while (--cnt && api\_adr==0);

  return api\_adr;

}

**导入表中的API：**

如果PE文件从其他模块导入API，这个导入目录将包含导入描述符的数组，每个代表一个模块。每个描述符包含了指向Image Thunk Data结构数组的指针。每个入口表示了导入的API的信息。



**Hint**

包含索引到DLL函数导出表的位置。这个字段被PE加载器使用，因此它能够在DLL导出表中快速的查找函数。这个值不是必须的，有些链接器将这个字段设置为0。

**Name**

包含导入函数的名字。是一个ASCIIZ字符串。注意Name字段的大小是可变的。提供的结构方便使您可以使用描述性名称引用数据结构。

因为我们不处理从序号导入的情况，所以我们不关心hint字段，只需要非0结尾字符串表示的API名。

1. PIC 位置独立代码， 指代码在任意地址都可以执行，对地址没有依赖。
2. shellcode编译的时候不能借助系统包含或者库。

gcc参数 -nostdinc 和 -ffreestanding 和 -nostdlib（不会链接静态库与动态库） 和 -fPIC 和 -Wl（设置链接器选项）

-march 指定目标平台

从汇编语言建立shellcode：gcc -nostdlib -o output.out inputfile.asm -Wl,--oformat,binary,-T,linker\_script.ld

如果汇编代码没有对全局偏移表的使用，那么就不需要使用链接脚本： gcc -nostdlib -o output.out input.asm -Wl,--ofomat,biinary

从C语言建立shellcode：gcc -O3 -fPIC -nostdlib --ffreestanding -nostdinc -Wl,--oformat,binary,-T,link\_script.ld -o output.out input.c

1. 编码器

msf /usr/share/metasploit-framework/modules/encoders提供x64平台下 xor、zutto\_dekiru两种编码器，均可以绕过360

alpha2 将命令序列转换为纯字母序列，但是在x86上使用纯字母shellcode可能会出现问题。在位解码技术需要知道shellcode被注入到哪一个地址上（在此地址上回复的代码成为getPC），windows下可以使用SEH。其余方法，我见过遍历内存，还见过提前监测（在装载之前系统就需要确定下位置，这时候只需要监测对应寄存器，比如call eax就应该监测eax寄存器）。