2017
绿盟科技
恶意样本分析手册

绿盟科技安全能力中心(SAC)

[理论篇]

(第二版)

关于绿盟科技

北京神州绿盟信息安全科技股份有限公司（简称绿盟科技）成立于 2000 年 4 月，总部位于北京。在国内外设有 30 多个分支机构，为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。

基于多年的安全攻防研究，绿盟科技在网络及终端安全、互联网基础安全、合规及安全管理等领域，为客户提供入侵检测 / 防护、抗拒绝服务攻击、远程安全评估以及 Web 安全防护等产品以及专业安全服务。

北京神州绿盟信息安全科技股份有限公司于 2014 年 1 月 29 日起在深圳证券交易所创业板上市交易。

股票简称：绿盟科技　股票代码：300369

目录

目录

目录

一．大小端模式

在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在 C 语言中除了 8bit 的 char 之外，还有 16bit 的 short 型，32bit 的 long 型（要看具体的编译器），另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。例如一个 16bit 的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么 0x11 为高字节，0x22 为低字节。对于 大端模式，就将 0x11 放在低地址中，即 0x0010 中，0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。我们常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多的 ARM，DSP 都为小端模式。有些 ARM 处理器还可以随时在程序中 (在 ARM Cortex 系列使用 REV、REV16、REVSH 指令) 进行大小端的切换。

1.1 大端模式

所谓的大端模式（Big-endian），是指数据的高字节，保存在内存的低地址中，而数据的低字节，保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

1.2 小端模式

所谓的小端模式（Little-endian），是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致。

例如：对于 0x11223344 的存储如下

大端模式

11	22	33	44		

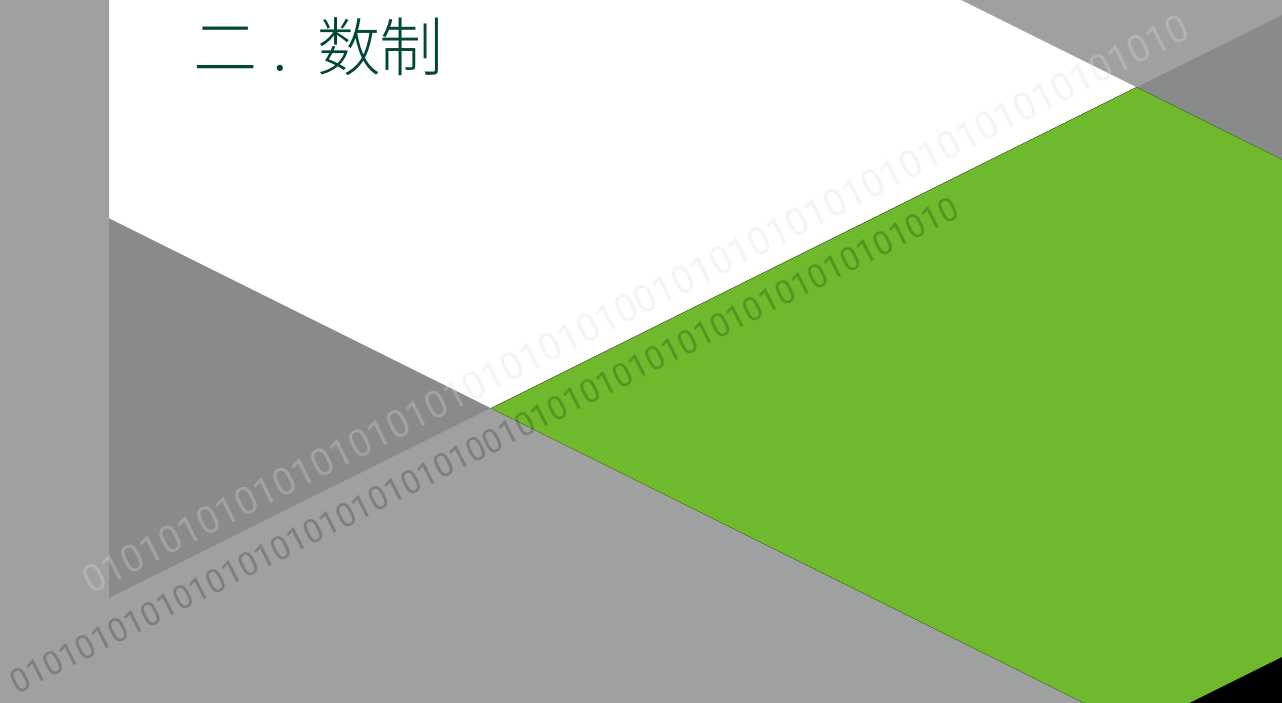
低地址 —> 高地址

11	22	33	44		

小端模式

高地址 —> 低地址

二．数制



进制也就是进位制，是人们规定的一种进位方法。 对于任何一种进制 ---X 进制，就表示某一位置上的数运算时是逢 X 进一位。 十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推，x 进制就是逢 x 进位。

2.1 二进制

二进制数有两个特点：它由两个基本数字 0，1 组成，二进制数运算规律是逢二进一。

为区别于其它进制数，二进制数的书写通常在数的右下方注上基数 2，或加后面加 B 表示。

例如：二进制数 10110011 可以写成（10110011)2，或写成 10110011B

2.2 八进制

八进制的基 R=8=2^3，有数码 0、1、2、3、4、5、6、7，并且每个数码正好对应三位二进制数，所以八进制能很好地反映二进制。八进制用下标 8 或数据后面加 O 表示 例如：二进制数据 （ 11 101 010 . 010 110 100)2 对应 八进制数据 (3 5 2 . 2 6 4)8 或 352.264O.

2.3 十六进制

十六进制数有两个基本特点：它由十六个字符 0 ～ 9 以及 A，B，C，D，E，F 组成（它们分别表示十进制数 10 ～ 15），十六进制数运算规律是逢十六进一，即基 R=16=2^4，通常在表示时用尾部标志 H 或下标 16 以示区别。

例如：十六进制数 4AC8 可写成（4AC8)16，或写成 4AC8H。

2.4 进制转换

说到进制转换，不得不提一下位权。对于形式化的进制表示，我们可以从 0 开始，对数字的各个数位进行编号，即个位起往左依次为编号 0，1，2，……；对称的，从小数点后的数位则是 -1，-2，……

进行进制转换时，我们不妨设源进制（转换前所用进制）的基为 R1，目标进制（转换后所用进制）的基为 R2，原数值的表示按数位为 $A_nA_{(n-1)}$ ……$A_2A_1A_0.A_{-1}A_{-2}$……，R1 在 R2 中的表示为 R，则有（$A_nA_{(n-1)}$ ……$A_2A_1A_0.A_{-1}A_{-2}$……） R1=(A_n*R^n+$A_{(n-1)}$*R^(n-1)+……+A_2*R^2+A_1*R^1+A_0*R^0+A_{-1}*R^(-1)+A_{-2}*R^(-2))R2

二进制数、十六进制数转换为十进制数的规律是相同的。把二进制数（或十六进制数）按位权形式展开多项式和的形式，求其最后的和，就是其对应的十进制数——简称“按权求和”.

例如：

把（1001.01)2 二进制计算。

解：（1001.010)2

=8*1+4*0+2*0+1*1+0*(1/2)+1*(1/4)

=8+0+0+1+0+0.25

=9.25

把（38A.11)16 转换为十进制数

解：（38A.11)16

=3×16 的 2 次方 +8×16 的 1 次方 +10×16 的 0 次方 +1×16 的 -1 次方 +1×16 的 -2 次方

=768+128+10+0.0625+0.0039

=906.0664

十进制数转换为二进制数，十六进制数（除 2/16 取余法）

整数转换 . 一个十进制整数转换为二进制整数通常采用除二取余法，即用 2 连续除十进制数，直到商为 0，逆序排列余数即可得到——简称除二取余法.

例：将 25 转换为二进制数

解：25÷2=12 余数 1

12÷2=6 余数 0

6÷2=3 余数 0

3÷2=1 余数 1

1÷2=0 余数 1

所以 25=(11001)2

同理，把十进制数转换为十六进制数时，将基数 2 转换成 16 就可以了 .

例：将 25 转换为十六进制数

解：25÷16=1 余数 9

1÷16=0 余数 1

所以 25=(19)16

二进制数与十六进制数之间的转换

由于 4 位二进制数恰好有 16 个组合状态，即 1 位十六进制数与 4 位二进制数是一一对应的 . 所以，十六进制数与二进制数的转换是十分简单的 .

(1) 十六进制数转换成二进制数，只要将每一位十六进制数用对应的 4 位二进制数替代即可——简称位分四位 .

例：将（4AF8B)16 转换为二进制数 .

解： 4 A F 8 B

0100 1010 1111 1000 1011

所以（4AF8B)16=(10010101111110001011)2

(2) 二进制数转换为十六进制数，分别向左，向右每四位一组，依次写出每组 4 位二进制数所对应的十六进制数——简称四位合一位 .

例：将二进制数 (000111010110)2 转换为十六进制数 .

解： 0001 1101 0110

1 D 6

所以 (111010110)2=（1D6)16

转换时注意最后一组不足 4 位时必须加 0 补齐 4 位

数制转换的一般化

1）R 进制转换成十进制

任意 R 进制数据按权展开、相加即可得十进制数据。

例如：N = 1101.0101B = 1*2^3+1*2^2+0*2^1+1*2^0+0*2^-1+1*2^-2+0*2^-3+1*2^-4 = 8+4+0+1+0+0.25+0+0.0625 = 13.3125

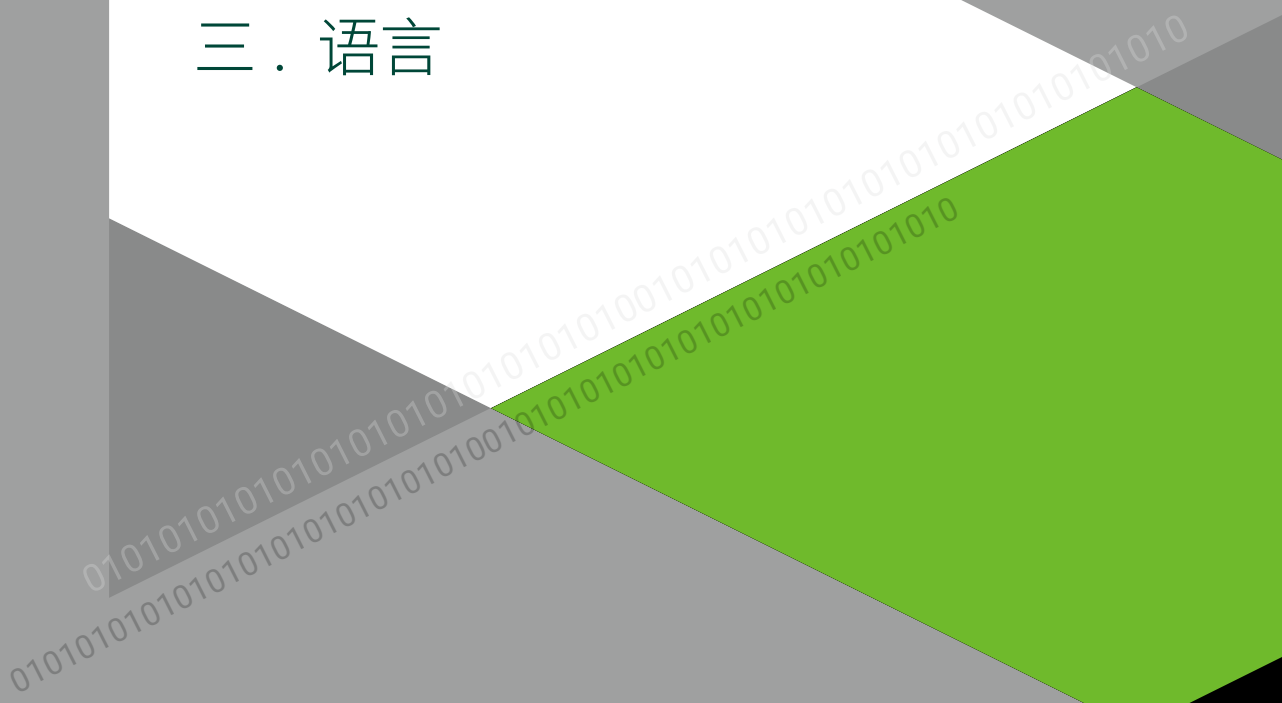
N = 5A.8H = 5*16^1+A*16^0+8*16^-1 = 80+10+0.5 = 90.5

2）十进制转换 R 进制

十进制数转换成 R 进制数，须将整数部分和小数部分分别转换 .

1. 整数转换——除 R 取余法 规则：（1）用 R 去除给出的十进制数的整数部分，取其余数作为转换后的 R 进制数据的整数部分最低位数字；（2）再用 R 去除所得的商，取其余数作为转换后的 R 进制数据的高一位数字；（3）重复执行（2）操作，一直到商为 0 结束。例如：115 转换成 Binary 数据和 Hexadecimal 数据 （图 2-4） 所以 115 = 1110011 B = 73 H
2. 小数转换——乘 R 取整法规则：（1）用 R 去乘给出的十进制数的小数部分，取乘积的整数部分作为转换后 R 进制小数点后第一位数字；（2）再用 R 去乘上一步乘积的小数部分，然后取新乘积的整数部分作为转换后 R 进制小数的低一位数字；（3）重复（2）操作，一直到乘积为 0，或已得到要求精度数位为止

三．语言



3.1 逻辑运算

逻辑运算又称为布尔运算。通常用来测试真假值，最常见到的逻辑运算就是循环的处理，用来判断是否该离开循环或继续执行循环内的指令。

逻辑常量只有两个，0 和 1，用来表示两个对立的逻辑状态。在逻辑代数中，有与、或、非三种基本的逻辑运算。

与运算（AND）

“与”运算是一种二元运算，它定义了两个变量 A 和 B 的一种函数关系。用语句来描述它，这就是：当且仅当变量 A 和 B 都为 1 时，函数 F 为 1。

下表是与运算的真值表：

AND	1	0
1	1	0
0	0	0

或运算（OR）

“或”运算是另一种二元运算，它定义了变量 A、B 与函数 F 的另一种关系。用语句来描述它，这就是：只要变量 A 和 B 中任何一个为 1，则函数 F 为 1。

下表是或运算的真值表：

OR	1	0
1	1	1
0	1	0

非运算（NOT）

逻辑“非”运算是一元运算，它定义了一个变量（记为 A）的函数关系。用语句来描述之，这就是：当 A=1 时，则函数 F=0；反之，当 A=0 时，则函数 F=1

下表是非运算的真值表：

NOT	1	0
	0	1

算术左移、逻辑左移

左移用来将一个数的各位二进制位全部左移若干位。例如：

将 a 的二进制数左移 2 位，右补 0。若 a=15，即二进制数 00001111，左移 2 位得 00111100，即十进制数 60，高位左移后溢出，舍弃。左移一位相当于该数乘以 2，左移 2 位相当于该数乘以 2^2=4。上面举的例子 15<<2=60，即乘了 4。但此结论只适用于该数左移时被溢出舍弃的高位中不包含 1 的情况。

循环左移

循环左移类似于逻辑左移，不同的是，循环左移会将左边移出的位添补到左边。例如原数 $x_3x_2x_1x_0$, 循环左移一位后，变为 $x_2x_1x_0\ x_3$。

可见，所有的位顺序向左移 1 位，最低位由最高位循环移入。

带进位的循环左移

此方法和循环左移类似，只是多了一个符号位，举例来说：

原数：$CX_3X_2X_1X_0$，循环左移一位后变为 $X_3X_2X_1X_0C$。

算术右移

算术右移用来将一个数的各位二进制位全部右移若干位，然后在左侧用原符号位补齐。在汇编语言中，如果最高位为 1，则补 1，否则补 0。如将 10000000 算术右移 7 位，应该变成 11111111。

逻辑右移

逻辑右移是将各位依次右移指定位数，然后在左侧补 0. 不考虑符号位。例如将 10000000 逻辑右移 7 位，变为 00000001。

循环右移

循环右移类似于逻辑右移，不同的是，循环右移会将右边移出的位添补到左边。例如原数 $x_3x_2x_1x_0$, 循环右移一位后，变为 $x_0x_3x_2x_1$。

可见，所有的位顺序向右移 1 位，最高位由最低位循环移入。

带进位的循环右移

此方法和循环右移类似，只是多了一个符号位，举例来说：

原数：$CX_3X_2X_1X_0$，循环右移一位后变为 $X_0CX_3X_2X_1$。

3.2 流程控制语句的识别

流程控制语句的识别时进行逆向分析和还原高级代码的基础，详细的理解此基础可以更好的理解高级语言中流程控制的内部实现机制，对开发和调试大有益处。

If 语句

If 语句是分支结构的重要组成部分。If 语句的功能是现对运算条件进行比较，然后根据比较结果选择对应的语句块来执行。If 语句只能判断两种情况：0 为假值，非 0 为真值。如果为真值，则进入语句块内执行语句；如果为假值，则跳过 if 语句块，继续执行程序的其他语句。要注意的是，if 语句转换的条件跳转指令与 if 语句的判断结果是相反的。

If 语句的一般流程如下：

```
// 先执行各类影响标志位的指令
// 其后是各种条件跳转指令
jxx   xxxx
```

if…else…语句

if 语句是一个单分支结构，if…else…组合后是一个双分支结构。两者完成的功能有所不同。从语法上看，

if…else…只比 if 语句多出了一个 else。else 有两个功能，如果 if 判断成功，则跳过 else 分支语句块；如果 if 判断失败，则进入 else 分支语句块中。有了 else 语句的存在，程序在进行流程选择时，必会经过两个分支中的一个。

if…else…的大致流程如下：

先执行影响标志位的相关指令

```
    jxx else_begin    // 该地址为 else 语句块的首地址
if_begin
    .......            //if 语句块内的执行代码
if_end
    jmp else_end  // 跳转到 else 语句块的结束地址
else_begin
    ......            //else 语句块内的执行代码
else_end
```

如果遇到以上指令序列，先考察其中的两个跳转指令，当第一个条件跳转指令跳转到地址 else_begin 处之前有个 jmp 指令，则可将其视为由 if...else... 组合而成的双分支结构。根据这两个跳转指令可以得到 if 和 else 语句块的代码边界。通过 cmp 和 jxx 可还原出 if 的比较信息，jmp 指令之后即为 else 块的开始。

if 构成的多分支结构

多分支结构类似于 if…else…的组合方式，在 if…else…的 else 之后再添加一个 else if 进行二次比较，这样就可以进行多次比较，再次选择程序流程，形成多分支流程。它的 c++ 语法格式为：if…else if…else if…，可重复后缀为 else if。当最后为 else 时，便到了多分支结构的末尾处。

一般流程如下：

jxx 指出了下一个 else if 的起始点，而 jmp 指出了整个多分支结构的末尾地址以及当前 if 或者 else if 语句块的末尾。最后的 else 块的边界也很容易识别，如果发现多分支块内的某一段代码在执行前没有判定，即可定义为 else 块。

```
// 会影响标志位的指令
    jxx else_if_begin // 跳到下一条 else if 语句块的首地址
if_begin
.....                  //if 语句块内的执行代码
if_end
jmp end            // 跳转到多分枝结构的结尾地址
else_if_begin      //else if 语句块的起始地址
// 可影响标志位的指令
```

```
    jxx else_begin
.....
else_if_end:
    jmp end
else_begin:
.....
end
.....
```

当每个条件跳转指令的跳转地址之前都紧跟 jmp 指令，并且他们的跳转地址值都一样时，可视为一个多分支结构。

switch

switch 是比较常用的多分支结构，使用起来也非常方便，并且效率也高于 if…else if 多分枝结构。switch 语句将所有的条件跳转都放置在了一起，并没有发现 case 语句块的踪影，通过条件跳转指令，跳转到相应的 case 语句块中。因此每个 case 的执行是由 switch 比较结果引导跳过来的。

一般流程如下：

```
mov reg,mem // 取出 switch 中考察的变量
// 影响标志位的指令
jxx  xxxx
// 影响标志位的指令
jxx  xxxx
// 影响标志位的指令
jxx  xxxx
jmp end  // 跳到 switch 语句块的结尾地址出
.....  //case 语句块首地址
jmp end  // 跳到 switch 语句块的结尾地址出
.....  //case 语句块首地址
jmp end  // 跳到 switch 语句块的结尾地址出
.....  //case 语句块首地址
jmp end  // 跳到 switch 语句块的结尾地址出
end:
```

.....

当分支数小于 4 的情况下，VC 6.0 会采取模拟 if else 的方法

当分支数大于 3，并且 case 的判定值存在明显线性关系组合时，它会制作一份 case 地址数组（case 地址表），这个数组保存了每个 case 语句块的首地址，并且数组下标以 0 起始。如果每两个 case 值之间的差值小于等于 6，并且 case 语句数大于等于 4，编译器就会形成这种线性结构。

对于非线性的 switch 结构，会进行索引表优化，需要两张表：一张为 case 语句块地址表，另一张为 case 语句块索引表

地址表中的每一项保存了一个 case 语句块的首地址，有几个 case 语句就有几项。此情况适用于差值小于等于 255 的情况，大于 255 的话可以通过树方式优化。

do 循环

do 循环的工作流程清晰，识别起来也相对简单。根据其特性，先执行语句块，再进行比较判断，当条件成立时，会继续执行语句块。

if 语句的比较是相反的，并且跳转地址大于当前代码的地址，是一个向下跳转的过程；而 do 中的跳转地址小于当前代码的地址，是一个向上跳转的过程，所以条件跳转的逻辑与源码中的逻辑相同

do 循环的一般流程：

do_begin

..... // 循环语句块

; 影响标记位的指令

jxx do_begin

while 循环

while 循环和 do 循环正好相反，在执行循环语句块之前，必须要进行条件判断，根据比较结果再选择是否执行循环语句块。识别 while 循环，查看条件跳转地址，如果这个地址上面有一个 jmp 指令，并且此指令跳转到的地址小于当前代码地址，那么明显是一个向上跳转的地址。要完成语句循环，就需要修改程序流程，回到循环语句处，因此向上跳转就成了循环语句的明显特征。在条件跳转的地址附近会有 jmp 指令修改程序流程。

while 循环用了两次跳转，因此比 do 循环效率低一些。

while 循环的一般流程：

while_begin

; 影响标记位的指令

jxx while_end

.....

jmp while_begin

while_end:

for 循环

for 循环是三种循环结构中最复杂的一种。for 循环由赋初值，设置循环条件，设置循环步长这三条语句组成。由于 for 循环更符合人类的思维方式，在循环结构中被使用的频率也很高。

for 循环的一般流程：

```
mov mem/reg ,xxx    // 赋初值
jmp for_cmp          // 跳到循环条件判定部分
for_step:
// 修改循环变量 step
mov reg,step
add reg,xxxx       // 修改循环变量的计算过程，在实际分析中，视算法不同而不同
mov step,eax
for_cmp:           // 循环条件判定部分
mov ecx,dword ptr step
// 判定循环变量和循环终止条件 stepend 的关系，满足条件则退出 for 循环
cmp ecx,stepend
jxx for_end          // 条件成立则结束循环
.....
jmp for_step         // 向上跳转，修改流程回到步长计算部分
for_end:
```

在计数器变量被赋初值后，利用 jmp 跳过第一次步长计算，然后，可以通过三个跳转指令还原 for 循环的各个组成部分：第一个 jmp 指令之前的代码为初始化部分；从第一个 jmp 指令到循环条件比较处（也就是上面代码中 for_cmp 标号的位置）之间的代码为步长计算部分；在条件跳转指令 jxx 之后寻找一个 jmp 指令，这 jmp 指令必须是向上跳转的，且其目标是到步长计算的位置，在 jxx 和这个 jmp（也就是上面代码的省略号所在的位置）之间的代码是循环语句块。

3.3 栈

从数据结构角度看，栈是一种用来存储数据的容器。放入数据的操作称为压入（push），从栈中取出数据的操作被称为弹出（pop）。存取数据的一条基本规则是后进先出。

X86 架构有对栈的内建支持。用于这种支持的寄存器包括 esp 和 ebp。其中 esp 是栈指针，包含了指向栈顶的内存地址。当数据被压入或弹出栈时，这个寄存器的值相应的改变。Ebp 是栈基址寄存器，在一个函数中保持

不变，因此程序把它当成定位器，用来确定局部变量和参数的位置。

与栈有关的指令包括 push，pop，call，leave，enter 和 ret。在内存中，栈被分配成自顶向下的，最高的地址最先被使用。当一个值被压入栈时，使用低一点的地址。

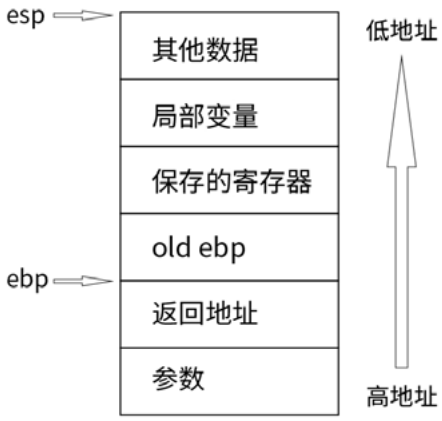
栈只能用于短期存储。他经常用于保存局部变量、参数和返回地址。主要用途是管理函数调用之间的数据交换。不同的编译器对这种管理方法的具体实现有所不同，但大部分常见约定都使用 ebp 的地址来引用局部变量与参数。

栈的布局：

在经典的操作系统中，栈总是向下（低地址）增长的。

栈保存一个函数调用所需要的维护信息，常被称为堆栈帧或者是活动记录，堆栈帧一般包括：

1. **函数的返回地址和参数**
2. **临时变量：**包括函数的非静态局部变量以及编译器生成的其他局部变量
3. **保存的上下文：**包括在函数调用前后保持不变的寄存器



3.4 堆

堆是组织内存的另一种重要方法，是程序运行期动态申请内存空间的主要途径。与栈空间是由编译器产生的代码自动分配和释放不同，堆上的空间需要程序员自己写代码来申请（HeapAlloc）和释放（HeapFree），而且分配和释放操作应该严格匹配，忘记释放或多次释放都是不正确的。

3.5 异常

异常通常是 CPU 在执行指令是因为检测到预先定义的某个（或多个）条件而产生的同步事件，异常的来源有 3 种，第一种是程序错误，即当 CPU 在执行程序指令时遇到操作数有错误（执行除法指令时遇到除数是 0）或检测到指令规范中定义的非法情况（用户模式下执行特权指令等）。第二种来源是某些特殊指令，这些指令的预期行为就是产生相应的异常，比如 INT3 指令，该指令的目的就是产生一个断点异常，让 CPU 中断进调试器。第三种来源是奔腾 CPU 引入的机器检查异常，即当 CPU 执行指令期间检测到 CPU 内部或外部的硬件错误。

异常分为 3 类，错误，陷阱和中止

错误类异常

导致错误类异常的情况通常可以被纠正，而且一旦纠正后，程序可以无损失的恢复执行。此类异常的一个最常见的例子就是内存页错误。页错误异常的发生是因为它是虚拟内存的基础。因为物理内存的空间有限，所以操作系统会把某些在那时不用的内存以页为单位交换到外部存储器上。当有程序访问到这些不在物理内存种的页所对应的内存地址时，CPU 便会产生一个页错误异常（缺页错误、缺页异常），并转去执行该异常的处理程序，后者会调用内存管理器的函数把对应的内存页交换回物理内存，然后再让 CPU 返回到导致该异常的那条指令处恢复执行。当第二次执行刚才导致异常的指令时，对应的内存页已经在物理内存中（错误情况被纠正），因此就不会再产生页错误异常了。

当 CPU 报告错误类异常时，CPU 将其状态恢复成导致该异常的指令被执行之前的状态。而且在 CPU 转去执行异常处理程序前，在栈中保存的 CS 和 EIP 指针是指向导致异常的这条指令的（而不是下一条指令）。因此，当异常处理程序返回继续执行时，CPU 接下来执行的第一条指令仍然是刚才导致异常的那条指令。所以，如果导致异常的情况还没有被消除，那么 CPU 会再次产生异常。

陷阱类异常

当 CPU 报告陷阱类异常时，导致该异常的指令已经执行完毕，压入栈的 CS 和 EIP 值是导致该异常的指令执行后紧接着要执行的下一条指令。值得说明的是，下一条指令并不一定是与导致异常的指令相邻的下一条。如果导致异常的指令是跳转指令或函数调用指令，那么下一条指令可能是内存地址不相邻的另一条指令。

导致陷阱类异常的情况通常也是可以无损失的恢复执行的。比如 INT 3 指令导致的断点异常就属于陷阱类异常，该异常会使 CPU 中断到调试器，从调试器返回后，被调试程序可以继续执行。

中止类异常

中止类异常主要用来报告严重的错误，比如硬件错误和系统表中包含非法值或不一致的状态等。这类异常不允许恢复继续执行。首先，当这类异常发生时，CPU 并不总能保证报告的异常的指令地址是精确地。另外，出于安全性的考虑，这类异常可能是由于导致该异常的程序执行非法操作导致的，因此就应该强迫其中止退出。

3.6 中断 / 异常处理

中断和异常从产生的根源来看有着本质的区别，但是系统（CPU 和操作系统）是用统一的方式来响应和管理他们的。中断和异常处理的核心数据结构是中断描述符表（IDT）。当中断和异常发生时，CPU 通过查找 IDT 表来定位处理例程的地址，然后转去执行该处理例程。这个查找的过程是在 CPU 内部执行的。通常，系统软件（操作系统和 BIOS 固件）在系统初始化阶段就准备好中断处理例程和 IDT 表，然后把 IDT 表的位置通过 IDTR 寄存器告诉 CPU。

实模式下 IVT（中断向量表）位于物理地址 0 开始的 1KB 内存区中，每个 IVT 表项的长度是 4 个字节，共有 256 个表项，与 x86CPU 的 256 个中断向量一一对应。实模式下，每个 IVT 表项的资格字节分为两部分，高两个字节为中断例程的段地址，低两个字节为中断例程的偏移地址。因为是在实模式下，所以段地址左移 4 位再加上偏移地址便可以得到 20 位的中断例程地址。

下面是 IA-32 CPU 相应中断和异常的全过程：

1. 将代码段寄存器 CS 和指令指针寄存器（EIP）的低 16 位压入堆栈
2. 将标志寄存器 EFLAGS 的低 16 位压入堆栈
3. 清除标志寄存器的 IF 标志，以禁止其他中断
4. 清除标志寄存器的 TF，RF，AC 标志
5. 使用向量号 n 作为索引，在 IVT 中找到对应的表项（n*4+IVT 表基地址）
6. 将表项中的段地址和偏移地址分别装入 CS 和 EIP 寄存器中，并开始执行对应的代码
7. 中断例程总是以 IRET 指令结束。IRET 指令会从堆栈中弹出前面保存的 CS，IP 和标志寄存器值，然后返回执行被中断的程序。

3.7 中断

中断通常是由 CPU 外部的输入输出设备（硬件）所触发的，供外部设备通知 CPU“有事情要处理”，因此又叫中断请求。中断请求的目的是希望 CPU 暂时停止执行 当前正在执行的程序，转去执行中断请求所对应的中断处理例程。

中断机制为 CPU 和外部设备间的通信提供了一种高效的方法，有了中断机制，CPU 就可以不用去频繁的查询外部设备的状态了，因为外部设备有事需要处理时，他可以发出中断请求通知 CPU。

在硬件级，中断是由一块专门芯片来管理的，通常称为中断控制器。他负责分配中断资源和管理各个中断源发出的中断请求。为了便于标识各个中断请求，中断管理器通常用 IRQ 后面加上数字来表示不同路的中断请求信号。

3.8 函数调用约定

cdecl

cdecl 调用约定又称为 C 调用约定，是 c/c++ 语言缺省的调用约定。参数按照从右至左的方式入栈，函数本身不清理栈，此工作有调用者负责，返回值在 eax 中。由于由调用者清理栈，所以允许可变参数函数存在。

stdcall

stdcall 很多时候被称为 pascal 调用约定。pascal 语言是早期很常见的一种教学用计算机程序设计语言，其语法严谨，参数按照从右至左的方式入栈，函数自身清理堆栈，返回值在 eax 中。

fastcall

fastcall 的调用方式运行相对快，因为它通过寄存器来传递参数。它使用 ecx 和 edx 传送两个双字或更小的参数，剩下的参数按照从右至左的方式入栈，函数自身清理堆栈，返回值在 eax 中。

naked

naked 是一个很少见的调用约定，一般不建议使用。编译器不会给这种函数增加初始化的清理代码，更特殊的是，你不能用 return 返回返回值，只能用插入汇编返回结果，此调用约定必须跟 declspec 同时使用，例如声明一个函数，如 _declspec(naked) int add(int a,int b);

pascal

这是 pascal 语言的调用约定，跟 stdcall 一样，参数按照从右至左的方式入栈，函数自身清理堆栈，返回值在 eax 中，vc 已经废弃了这种调用方式，因此在写 vc 程序时，建议使用 stdcall。

thiscall

这是 c++ 语言特有的一种调用方式，用于类成员函数的调用约定。如果参数确定，this 指针存放于 ecx 寄存器，函数自身清理堆栈；如果参数不确定，this 指针在所有参数入栈后再入栈，调用者清理栈。Thiscall 不是关键字，程序员不能使用。参数按照从右至左的方式 r 入栈。

四．文件格式

4.1 ELF 文件格式

目标文件有三种类型：

可重定位文件（ Relocatable File） 包含适合于与其他目标文件链接来创建可执行文件或者共享目标文件的代码和数据。

可执行文件（ Executable File） 包含适合于执行的一个程序，此文件规定了 exec() 如何创建一个程序的进程映像。

共享目标文件（ Shared Object File） 包含可在两种上下文中链接的代码和数据。首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理，生成另外一个目标文件。其次，动态链接器（Dynamic Linker）可能将它与某个可执行文件以及其它共享目标一起组合，创建进程映像。

目标文件全部是程序的二进制表示，目的是直接在某种处理器上直接执行。

ELF 文件是运行在 unix 平台下的可执行文件，在学习其文件格式前，首先了解一下文件中使用的数据表示方式。如下表所示：

名称	大小	对齐	目的
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等正数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
Unsigned char			无符号小正数

目标文件格式

目标文件既要参与程序链接，又要参与程序执行。出于方便性和效率考虑，目标文件格式提供了两种并行视图，分别反应了这些活动的不同需求。

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表（可选）	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表（可选）

如上图所示，在文件开始处是一个 ELF 头部，用来描述整个文件的组织。节区部分包含链接视图的大量信息：指令，数据，符号表，重定位信息等等。

程序头部表，如果存在的话，告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区头部表包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称，节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可有可无。

注意：尽管头部显示的各个组成部分是有顺序的，实际上除了 ELF 头部表以外，其他节区和段都没有规定的顺寻。

ELF Header 部分

文件的最开始几个字节给出如何解释文件的提示信息。这些信息独立于处理器，也独立于文件中的其余内容。ELF Header 部分可以用下面的数据结构表示：

```
#define EI_NIDENT 16
typedef struct{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
}Elf32_Ehdr;
```

e_ident：目标文件标识，标志此文件是一个 ELF 文件。

e_type：目标文件的类型，取值取下

名称	取值	含义
ET_NONE	0	未知目标文件格式
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	共享目标文件
ET_CORE	4	Core 文件（转储格式）
ET_LOPROC	0xff00	特定处理器文件
ET_HIPROC	0xffff	特定处理器文件

ET_LOPROC 和 ET_HIPROC 之间的取值用来标识与处理器相关的文件格式。

e_machine：给出文件的目标体系结构类型，取值如下：

名称	取值	含义
EM_NONE	0	未指定
EM_M32	1	AT&T WE 32100

EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

特定处理器的 ELF 名称会使用机器名来进行区分。

e_version: 目标文件版本。

e_entry：程序入口的虚拟地址。如果目标文件没有程序入口，可以为 0.

e_phoff：程序头部表格（ Program Header Table）的偏移量（按字节计算）。如果文件没有程序头部表格，可以为 0。

e_shoff：节区头部表格（ Section Header Table） 的偏移量（按字节计算）。 如果文件没有节区头部表格，可以为 0。

e_flags：保存与文件相关的， 特定于处理器的标志。 标志名称采用 EF_machine_flag 的格式。

e_ehsize ：ELF 头部的大小（以字节计算）。

e_phentsize 程序头部表格的表项大小（按字节计算）。

e_phnum 程序头部表格的表项数目。可以为 0。

e_shentsize 节区头部表格的表项大小（按字节计算）。

e_shnum 节区头部表格的表项数目。可以为 0。

e_shstrndx：节区头部表格中与节区名称字符串表相关的表项的索引。 如果文件没有节区名称字符串表，此参数可以为 SHN_UNDEF。

节区

节区中包含目标文件中的所有信息，除了：ELF 头部，程序头部表格，节区头部表格。节区满足以下条件：

1. 目标文件中的每个节区都有对应的节区头部描述它，反过来，有节区头部不意味着有节区
2. 每个节区占用文件中一个连续字节区域（这个区域可能长度为 0）
3. 文件中的节区不能重叠，不允许一个字节存在于两个节区中的情况发生
4. 目标文件中可能包含非活动空间（ INACTIVE SPACE）。这些区域不属于任何头部和节区，其内容未指定

节区头部表格

ELF 头部中， e_shoff 成员给出从文件头到节区头部表格的偏移字节数； e_shnum 给出表格中条目数目；e_shentsize 给出每个项目的字节数。 从这些信息中可以确切地定位节区的具体位置、长度。

每个节区头部可以用如下数据结构描述：

typedef struct{

```
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
}Elf32_Shdr;
```

各个成员含义如下：

成员	说明
sh_name	给出节区名称。是节区头部字符串表节区（Section Header StringTable Section）的索引。名字是一个 NULL 结尾的字符串。
sh_type	为节区的内容和语义进行分类。
sh_flags	节区支持 1 位形式的标志，这些标志描述了多种属性。此字段定义了一个节区中包含的内容是否是可以修改、是否可以执行等信息。如果一个标志位被设置，则该值取值为 1. 未定义的各位都设置为 0.
sh_addr	如果节区将出现在进程的内存映像中， 此成员给出节区的第一个字节应处的位置。否则，此字段为 0。
sh_offset	此成员的取值给出节区的第一个字节与文件头之间的偏移。不过，SHT_NOBITS 类型的节区不占用文件的空间，因此其 sh_offset 成员给出的是其概念性的偏移。
sh_size	此成员给出节区的长度（字节数）。除非节区的型是 SHT_NOBITS，否则节区占用文件中的 sh_size 字节。类型为 SHT_NOBITS 的节区长度可能非零， 不过却不占用文件中的空间。
sh_link	此成员给出节区头部表索引链接。其具体的解释依赖于节区类型。
sh_info	此成员给出附加信息，其解释依赖于节区类型。
sh_addralign	某些节区带有地址对齐约束。例如，如果一个节区保存一个 doubleword， 那么系统必须保证整个节区能够按双字对齐。 sh_addr 对 sh_addralign 取模，结果必须为 0。目前仅允许取值为 0 和 2 的幂次数。数值 0 和 1 表示节区没有对齐约束。
sh_entsize	某些节区中包含固定大小的项目， 如符号表。 对于这类节区， 此成员给出每个表项的长度字节数。 如果节区中并不包含固定长度表项的表格，此成员取值为 0。

特殊节区

很多节区中包含了程序和控制信息。下面的表格中给出了系统使用的节区以及他们的类型和属性。

名称	大小	对齐	目的
.bss	SHT_NOBITS	SHF_ALLOC +SHF_WRITE	包含将出现在程序的内存映像中的为初始化数据。根据定义，当程序开始执行， 系统将把这些数据初始化为 0。 此节区不占用文件空间。
.comment	SHT_PROGBITS	无	包含版本控制信息。
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE	这些节区包含初始化了的数据，将出现在程序的内存映像中。
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE	这些节区包含初始化了的数据，将出现在程序的内存映像中。
.debug	SHT_PROGBITS	无	此节区包含用于符号调试的信息。
.dynamic	SHT_DYNAMIC		此节区包含动态链接信息。 节区的属性将包含 SHF_ALLOC 位。是否 SHF_WRITE 位被设置取决于处理器。

.dynstr	SHT_STRTAB	SHF_ALLOC	此节区包含用于动态链接的字符串， 大多数情况下这些字符串代表了与符号表项相关的名称。
.dynsym	SHT_DYNSYM	SHF_ALLOC	此节区包含了动态链接符号表。
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	此节区包含了可执行的指令， 是进程终止代码的一部分。 程序正常退出时， 系统将安排执行这里的代码。
.got	SHT_PROGBITS		此节区包含全局偏移表。
.hash	SHT_HASH	SHF_ALLOC	此节区包含了一个符号哈希表。
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	此节区包含了可执行指令， 是进程初始化代码的一部分。 当程序开始执行时， 系统要在开始调用主程序入口之前（通常指 C 语言的 main 函数）执行这些代码。
.interp	SHT_PROGBITS		此节区包含程序解释器的路径名。 如果程序包含一个可加载的段， 段中包含此节区， 那么节区的属性将包含 SHF_ALLOC 位，否则该位为 0。
.line	SHT_PROGBITS	无	此节区包含符号调试的行号信息， 其中描述了源程序与机器指令之间的对应关系。 其内容是未定义的。
.note	SHT_NOTE	无	此节区中包含注释信息，有独立的格式。
.plt	SHT_PROGBITS		此节区包含过程链接表（ procedure linkagetable） 。
.relname	SHT_REL		这些节区中包含了重定位信息。如果文件中包含可加载的段，段中有重定位内容， 节区的属性将包含 SHF_ALLOC 位，否则该位置 0。 传统上 name 根据重定位所适用的节区给定。例如 .text 节区的重定位节区名字将是： .rel.text 或者 .rela.text。
.relaname	SHT_RELA		同上
.rodata	SHT_PROGBITS	SHF_ALLOC	这些节区包含只读数据， 这些数据通常参与进程映像的不可写段。
.rodata1	SHT_PROGBITS	SHF_ALLOC	同上
.shstrtab	SHT_STRTAB		此节区包含节区名称。
.strtab	SHT_STRTAB		此节区包含字符串， 通常是代表与符号表项相关的名称。如果文件拥有一个可加载的段， 段中包含符号串表， 节区的属性将包含 SHF_ALLOC 位，否则该位为 0。
.symtab	SHT_SYMTAB		此节区包含一个符号表。 如果文件中包含一个可加载的段， 并且该段中包含符号表， 那么节区的属性中包含 SHF_ALLOC 位， 否则该位置为 0。
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	此节区包含程序的可执行指令。

字符串表：

字符串表节区包含以 NULL 结尾的字符序列，通常称为字符串。ELF 目标问件通常使用字符串来表示符号和节区名称。对字符串的引用通常以字符串在字符串表中的下标给出。

一般，第一个字节（索引为 0）定义为一个空字符串。类似的，字符串表的最后一个字节也定义为 NULL，以确保所有字符串都以 NULL 结尾。

例如：对于各个节区而言，节区头部的 sh_name 成员包含其对应的节区头部字符串表节区的索引，此节区由 ELF 头的 e_shstrndx 成员给出。

符号表

目标文件的符号表中包含用来定位、重定位程序中符号定义和引用的信息。符号表索引是对此数组的索引。索引 0 表示表中的第一表项，同时也作为未定义符号的索引。

符号表项的格式如下：

```
typedef struct {
    Elf32_Word st_name;
```

```
    Elf32_Addr st_value;
    Elf32_Word st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half st_shndx;
} Elf32_sym;
```

各个字段的含义如下：

字段	说明
st_name	包含目标文件符号字符串表的索引， 其中包含符号名的字符串表示。 如果该值非 0， 则它表示了给出符号名的字符串表索引， 否则符号表项没有名称。 注：外部 C 符号在 C 语言和目标文件的符号表中具有相同的名称。
st_value	此成员给出相关联的符号的取值。 依赖于具体的上下文， 它可能是一个绝对值、一个地址等等。
st_size	很多符号具有相关的尺寸大小。 例如一个数据对象的大小是对象中包含的字节数。如果符号没有大小或者大小未知，则此成员为 0。
st_info	此成员给出符号的类型和绑定属性。
st_other	该成员当前包含 0，其含义没有定义。
st_shndx	每个符号表项都以和其他节区间的关系的方式给出定义。 此成员给出相关的节区头部表索引。某些索引具有特殊含义。

程序头部

可执行文件或者共享目标文件的程序头部是一个结构数组，没有结构描述了一个段或者系统准备程序执行所必须的其他信息。目标文件的“段”包含了一个或者多个“节区”，也就是“段内容”。程序头部仅对于可执行文件和共享目标文件有意义。

可执行目标文件在 ELF 头部的 e_phentsize 和 e_phnum 成员中给出其自身程序头部的大小，程序头部的数据结构如下图：

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_phdr;
```

各个字段说明如下：

p_type：此数组元素描述的段的类型，或者如何解释此数组元素的信息。

p_offset：此成员给出从文件头到该段第一个字节的偏移。

p_vaddr：此成员给出段的第一个字节将被放到内存中的虚拟地址。

p_paddr：此成员仅用于与物理地址相关的系统中。因为 System V 忽略所有应用程序的物理地址信息，此字段对与可执行文件和共享目标文件而言具体内容是未指定的。

p_filesz：此成员给出段在文件映像中所占的字节数。可以为 0。

p_memsz：此成员给出段在内存映像中占用的字节数。可以为 0。

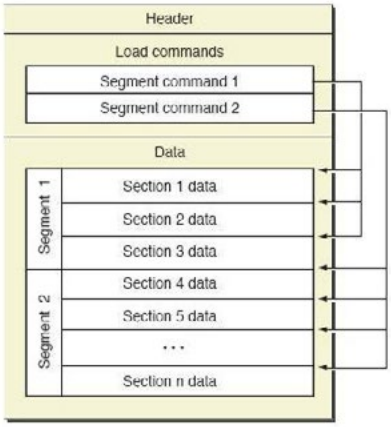
p_flags：此成员给出与段相关的标志。

p_align：可加载的进程段的 p_vaddr 和 p_offset 取值必须合适， 相对于对页面大小的取模而言。 此成员给出段在文件中和内存中如何对齐。数值 0 和 1 表示不需要对齐。否则 p_align 应该是个正整数， 并且是 2 的幂次数， p_vaddr 和 p_offset 对 p_align 取模后应该相等

4.2 MACH-O 文件格式

Mach-o 格式是 OS X 系统上的可执行文件格式，类似于 Windows 的 PE 与 linux 的 ELF。每个 mach-o 文件头包含一个 mach-o 头，然后载入命令（Load Commands），最后是数据块（Data）。下面来对整个 mach-o 的格式进行详细分析。

Mach-o 文件的格式如下图所示：



由如下几部分组成：

Header：保存了 mach-o 的一些基本信息，包括了平台，文件类型，LoadCommands 的个数等等。

LoadCommands：这一段紧跟 Header。加载 mach-o 文件时会使用这里的数据来确定内存的分布。

Data：每一个 segment 的具体数据都保存在这里，这里包含了具体的代码，数据等等。

Header

Headers 的定义可以在开源的内核代码中找到。

```
/*
 * The 32-bit mach header appears at the very beginning of the object file for
 * 32-bit architectures.
 */
struct mach_header {
    uint32_t        magic;          /* mach magic number identifier */
    cpu_type_t      cputype;        /* cpu specifier */
    cpu_subtype_t   cpusubtype;     /* machine specifier */
    uint32_t        filetype;       /* type of file */
    uint32_t        ncmds;          /* number of load commands */
    uint32_t        sizeofcmds;     /* the size of all the load commands */
    uint32_t        flags;          /* flags */
};

/* Constant for the magic field of the mach_header (32-bit architectures) */
#define    MH_MAGIC        0xfeedface      /* the mach magic number */
#define MH_CIGAM        0xcefaedfe      /* NXSwapInt(MH_MAGIC) */

/*
 * The 64-bit mach header appears at the very beginning of object files for
 * 64-bit architectures.
 */
struct mach_header_64 {
    uint32_t        magic;          /* mach magic number identifier */
    cpu_type_t      cputype;        /* cpu specifier */
    cpu_subtype_t   cpusubtype;     /* machine specifier */
    uint32_t        filetype;       /* type of file */
    uint32_t        ncmds;          /* number of load commands */
    uint32_t        sizeofcmds;     /* the size of all the load commands */
    uint32_t        flags;          /* flags */
    uint32_t        reserved;       /* reserved */
};

/* Constant for the magic field of the mach_header_64 (64-bit architectures) */
#define MH_MAGIC_64 0xfeedfacf /* the 64-bit mach magic number */
#define MH_CIGAM_64 0xcffaedfe /* NXSwapInt(MH_MAGIC_64) */
```

根据 mach_header 和 mach_header_64 的定义，很明显可以看出，Headers 的主要作用是帮助系统迅速的定位 mach-o 文件的运行环境，文件类型。

Magic：0xfeedface 是 32 位，0xfeedfacf 是 64 位。

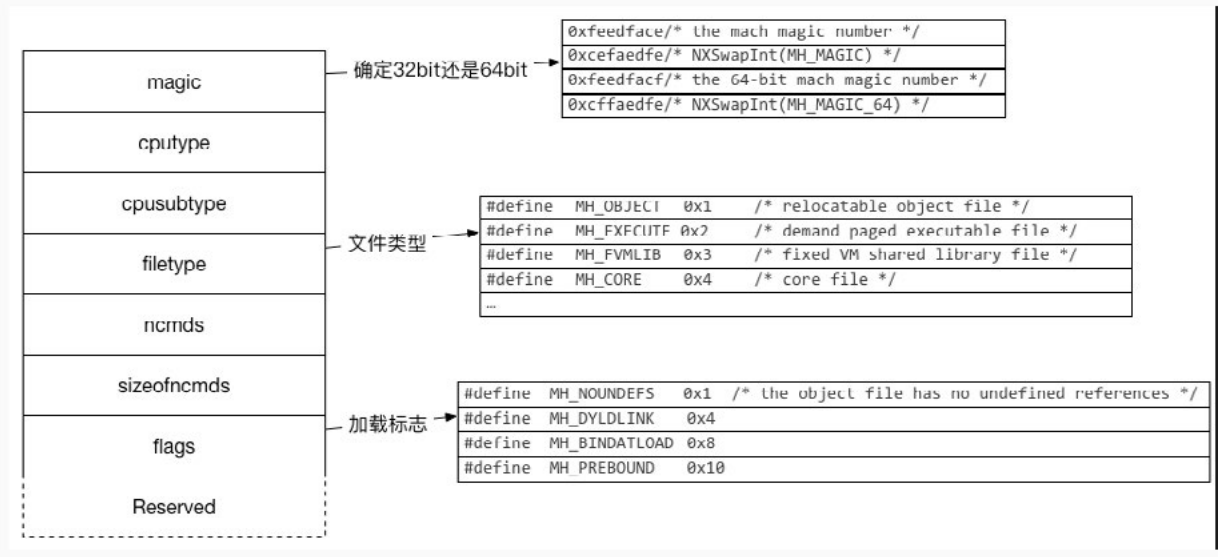
Cputype，cpusubtype：确定 CPU 的平台与版本（ARM-V7）

Filetype：文件类型（执行文件，库文件，core，内核扩展…）

Ncmds，sizeofncmds：Load Commands 的个数和长度

Flags：dyld 加载时需要的标志位

Reserved：只有 64 位的时候才存在的字段，暂时没有用



Load Commands

Load_Command 的数据结构如下：

```
struct load_command {
    uint32_t cmd;           /* type of load command */
    uint32_t cmdsize;       /* total size of command in bytes */
};
```

Load Commands 直接就跟在 Header 后面，所有 command 占用内存的总和在 Mach-O Header 里面已经给出了。在加载过 Header 之后就是通过解析 LoadCommand 来加载接下来的数据了。

Segment&Section

加载数据时，主要加载的就是 LC_SEGMET 或者 LC_SEGMENT_64。LC_SEGMET 和 LC_SEGMENT_64 的数据结构如下所示：

```
struct segment_command { /* for 32-bit architectures */
    uint32_t    cmd;        /* LC_SEGMENT */
    uint32_t    cmdsize;    /* includes sizeof section structs */
    char        segname[16]; /* segment name */
    uint32_t    vmaddr;     /* memory address of this segment */
    uint32_t    vmsize;     /* memory size of this segment */
    uint32_t    fileoff;    /* file offset of this segment */
    uint32_t    filesize;   /* amount to map from the file */
```

```
    vm_prot_t       maxprot;        /* maximum VM protection */
    vm_prot_t       initprot;       /* initial VM protection */
    uint32_t        nsects;         /* number of sections in segment */
    uint32_t        flags;          /* flags */
};
struct segment_command_64 { /* for 64-bit architectures */
    uint32_t        cmd;            /* LC_SEGMENT_64 */
    uint32_t        cmdsize;        /* includes sizeof section_64 structs */
    char            segname[16];    /* segment name */
    uint64_t        vmaddr;         /* memory address of this segment */
    uint64_t        vmsize;         /* memory size of this segment */
    uint64_t        fileoff;        /* file offset of this segment */
    uint64_t        filesize;       /* amount to map from the file */
    vm_prot_t       maxprot;        /* maximum VM protection */
    vm_prot_t       initprot;       /* initial VM protection */
    uint32_t        nsects;         /* number of sections in segment */
    uint32_t        flags;          /* flags */
};
```

可以看出，这里大部分的数据是用来帮助内核将 Segment 映射到虚拟内存的。主要要关注的是 nsects 字段，标示了 Segment 中有多少 section。section 是具体有用的数据存放的地方。

Section 的数据结构如下：

```
struct section { /* for 32-bit architectures */
    char            sectname[16];   /* name of this section */
    char            segname[16];    /* segment this section goes in */
    uint32_t        addr;           /* memory address of this section */
    uint32_t        size;           /* size in bytes of this section */
    uint32_t        offset;         /* file offset of this section */
    uint32_t        align;          /* section alignment (power of 2) */
    uint32_t        reloff;         /* file offset of relocation entries */
```

```
    uint32_t        nreloc;         /* number of relocation entries */
    uint32_t        flags;          /* flags (section type and attributes)*/
    uint32_t        reserved1;      /* reserved (for offset or index) */
    uint32_t        reserved2;      /* reserved (for count or sizeof) */
};
struct section_64 { /* for 64-bit architectures */
    char            sectname[16];   /* name of this section */
    char            segname[16];    /* segment this section goes in */
    uint64_t        addr;           /* memory address of this section */
    uint64_t        size;           /* size in bytes of this section */
    uint32_t        offset;         /* file offset of this section */
    uint32_t        align;          /* section alignment (power of 2) */
    uint32_t        reloff;         /* file offset of relocation entries */
    uint32_t        nreloc;         /* number of relocation entries */
    uint32_t        flags;          /* flags (section type and attributes)*/
    uint32_t        reserved1;      /* reserved (for offset or index) */
    uint32_t        reserved2;      /* reserved (for count or sizeof) */
    uint32_t        reserved3;      /* reserved */
};
```

除了同样有帮助内存映射的变量外，在了解 mach-o 格式的时候，只需要知道不同的 Section 有着不同的作用就可以了。

Section	作用
_text	代码
_cstring	硬编码的字符串
_const	Const 关键词修饰过的变量
_DATA._bss	Bss 段

4.3 PE 文件格式

PE 是 Portable Executable Format（可移植的执行体）简写，它是目前 Windows 平台的主流可执行文件格式。

MS-DOS 头部

每个 PE 文件是以一个 DOS 程序开始的，有了它，一旦程序运行在 DOS 下执行，DOS 就能识别出这是有效的执行体，然后运行紧随 MZ header 之后的 DOS stub（DOS 块）。DOS stub 实际上是一个有效的 EXE，平

常把 DOS MZ 头与 DOS stub 合称为 DOS 文件头。

PE 文件的第一个字节起始于一个传统的 MS-DOS 头部，被称作 IMAGE_DOS_HEADER。其结构如下：

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

其中有两个值比较重要，分别是 e_magic 合 e_lfanew。e_magic 字段需要被设置为值 5A4D，再 ASCII 表示法里，它的 ASCII 值为“MZ”。e_lfanew 字段是真正的 PE 文件头的相对偏移，其指出真正 PE 头的文件偏移位置，它占用 4 字节，位于文件开始偏移 3CH 字节中。

PE 文件头

紧跟着 Dos stub 的是 PE 文件头（PE Header），PE Header 是 PE 相关结构 NT 映像头（IMAGE_NT_HEADERS）的简称，其中包含很多 PE 装载器用到的重要字段。知性体再支持 PE 文件结构的操作系统中执行时，PE 装载器从 IMAGE_DOS_HEADER 结构中的 e_lfanew 字段里找到 PE Header 的起始偏移量，加上基址得到 PE

文件头的指针。

PNTHeader=ImageBase+dosHeader->e_lfanew

实际上又两个版本的 IMAGE_NT_HEADER 结构，一个是为 32 位的可执行文件准备的，另一个是 64 位版本，在后面的讨论中不做考虑，他们几乎没有区别。

IMAGE_NT_HEADER 由三个字段组成：

```
IMAGE_NT_HEADERS STRUCT
{
    DWORD Signature;                        //PE 文件头标志，为 ASCII 的“PE”，+0h
    IMAGE_FILE_HEADER FileHeader;           //+4h
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; //+18h
}
```

Signature 字段被设置为 00004550h，ASCII 码字符是“PE00”.“PE\0\0”是 PE 文件头的开始，DOS 头部的 e_lfanew 字段正是执行“PE\0\0”。

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;                        //+04h
    WORD    NumberOfSections;               //+06h  文件的区块数目
    DWORD   TimeDateStamp;                  //+08h
    DWORD   PointerToSymbolTable;           //+0Ch
    DWORD   NumberOfSymbols;                //+10h
    WORD    SizeOfOptionalHeader;           //+14h  IMAGE_OPTIONAL_HEADER32 结构大小
    WORD    Characteristics;                //+16h  文件属性
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

IMAGE_FILE_HEADER（映像文件头）结构包含了 PE 文件的一些基本信息，最重要的一个域指出了 IMAGE_OPTIONAL_HEADER 的大小。

Machine：可执行文件的目标 CPU 类型。PE 文件可以在多种机器上使用，不同平台指令机器码是不同的，下表所示是几种典型的机器类型标志。

机器	标志
Intel i386	14Ch
MIPS R3000	162h
MIPS R4000	166h
Alpha AXP	184h
Power PC	1F0H

NumberOfSections：区块数目，块表紧跟在 IMAGE_NT_HEADERS 后面

TimeDateStamp：表明文件是何时被创建的。

PointerToSymbolTable：COFF 符号表的问啊金偏移位置

NumberOfSymbols：如果有 COFF 符号表，它代表其中的符号数目。

SizeOfOptionalHeader：紧跟着 IMAGE_FILE_HEADER 后面的数据的大小。在 PE 文件中，这个数据结构叫 IMAGE_OPTIONAL_HEADER，其大小依赖于 32 位还是 64 位文件，对于 32 位文件，这个域通常是 00E0h；对于 64 位文件，这个域是 00F0h。这些值要求的最小值，较大的值也可能出现。

IMAGE_OPTIONAL_HEADER 结构

可选映像头（IMAGE_OPTIONAL_HEADER）是一个可选的结构，但实际上 IMAGE_FILE_HEADER 结构不足以定义 PE 文件属性，因此可选映像头中定义了更多的数据，完全不必考虑两个结构区别在哪里，两者连起来就是一个完整的 PE 文件头结构。IMAGE_OPTIONAL_HEADER32 结构如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER
{
    //
    // Standard fields.
    //
    +18h WORD Magic; // 标志字 , ROM 映像（0107h）, 普通可执行文件（010Bh）
    +1Ah BYTE MajorLinkerVersion; // 链接程序的主版本号
    +1Bh BYTE MinorLinkerVersion; // 链接程序的次版本号
    +1Ch DWORD SizeOfCode; // 所有含代码的节的总大小
    +20h DWORD SizeOfInitializedData; // 所有含已初始化数据的节的总大小
    +24h DWORD SizeOfUninitializedData; // 所有含未初始化数据的节的大小
    +28h DWORD AddressOfEntryPoint; // 程序执行入口 RVA
    +2Ch DWORD BaseOfCode; // 代码的区块的起始 RVA
    +30h DWORD BaseOfData; // 数据的区块的起始 RVA
    //
    // NT additional fields. 以下是属于 NT 结构增加的领域。
    //
    +34h DWORD ImageBase; // 程序的首选装载地址
    +38h DWORD SectionAlignment; // 内存中的区块的对齐大小
    +3Ch DWORD FileAlignment; // 文件中的区块的对齐大小
```

```
    +40h WORD MajorOperatingSystemVersion; // 要求操作系统最低版本号的主版本号
    +42h WORD MinorOperatingSystemVersion; // 要求操作系统最低版本号的副版本号
    +44h WORD MajorImageVersion; // 可运行于操作系统的主版本号
    +46h WORD MinorImageVersion; // 可运行于操作系统的次版本号
    +48h WORD MajorSubsystemVersion; // 要求最低子系统版本的主版本号
    +4Ah WORD MinorSubsystemVersion; // 要求最低子系统版本的次版本号
    +4Ch DWORD Win32VersionValue; // 莫须有字段，不被病毒利用的话一般为 0
    +50h DWORD SizeOfImage; // 映像装入内存后的总尺寸
    +54h DWORD SizeOfHeaders; // 所有头 + 区块表的尺寸大小
    +58h DWORD CheckSum; // 映像的校检和
    +5Ch WORD Subsystem; // 可执行文件期望的子系统
    +5Eh WORD DllCharacteristics; // DllMain() 函数何时被调用，默认为 0
    +60h DWORD SizeOfStackReserve; // 初始化时的栈大小
    +64h DWORD SizeOfStackCommit; // 初始化时实际提交的栈大小
    +68h DWORD SizeOfHeapReserve; // 初始化时保留的堆大小
    +6Ch DWORD SizeOfHeapCommit; // 初始化时实际提交的堆大小
    +70h DWORD LoaderFlags; // 与调试有关，默认为 0
    +74h DWORD NumberOfRvaAndSizes; // 下边数据目录的项数，这个字段自 Windows NT 发布以来
    // 一直是 16
    +78h IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
    // 数据目录表
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

下面讲解几个比较重要的字段：

AddressOfEntryPoint：程序执行入口 RVA，对于 DLL，这个入口点是在进程初始化和关闭时以及线程创建 / 毁灭时被调用。在大多数可执行文件中，这个地址并不直接指向 Main，WinMain 或 DllMain，而是指向运行时库代码并由他来调用上述函数。在 DLL 中这个域能被设为 0，前面提到的通知消息都不能收到。链接器 / NOENTRY 开关可以设置这个域为 0.

ImageBase：文件在内存中首选装入地址。如果有可能（也就是说，目前如果没有其他占据这个块地址，它是正确对齐的并且是一个合法的地址等），加载器试图在这个地址装入 PE 文件，如果可执行文件是在这个地址转入的，那么加载器将跳过应用基址重定位的步骤。

SectionAlignment：当被装入内存时的区块对齐大小。每个块被装入的地址必定是本字段指定数值的整数倍。默认的对齐尺寸是目标 CPU 的页尺寸。

FileAlignment：磁盘上 PE 文件内的区块对齐大小。组成块的原始数据必须保证从本字段的倍数地址开始。对于 x86 可执行文件，这个值通常是 200h 或 1000h，这是为了保证块总是从磁盘的扇区开始。这个值必须是 2 的幂，其最小值为 200h，并且，如果 SectionAlignment 小于 CPU 的页尺寸，这个域必须与 SectionAlignment 匹配。

Subsystem：一个标明可执行文件所期望的子系统的枚举值，取值如下：

取 值	Windows.inc 中的预定义值	含 义
0	IMAGE_SUBSYSTEM_UNKNOWN	未知的子系统
1	IMAGE_SUBSYSTEM_NATIVE	不需要子系统（如驱动程序）
2	IMAGE_SUBSYSTEM_WINDOWS_GUI	Windows 图形界面
3	IMAGE_SUBSYSTEM_WINDOWS_CUI	Windows 控制台界面
5	IMAGE_SUBSYSTEM_OS2_CUI	OS2 控制台界面
7	IMAGE_SUBSYSTEM_POSIX_CUI	POSIX 控制台界面
8	IMAGE_SUBSYSTEM_NATIVE_WINDOWS	不需要子系统
9	IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	Windows CE 图形界面

Data Directory[16]：数据目录表，由数个相同 IMAGE_DATA_DIRECTORY 结构组成，指向输出表，输入表，资源块等数据。IMAGE_DATA_DIRECTORY 结构的定义如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;                 // 数据块的起始 RVA
    DWORD   Size;                           // 数据块长度
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

数据目录列表如下：

索 引	索引值在 Windows.inc 中的预定义值	对应的数据块
0	IMAGE_DIRECTORY_ENTRY_EXPORT	导出表
1	IMAGE_DIRECTORY_ENTRY_IMPORT	导入表
2	IMAGE_DIRECTORY_ENTRY_RESOURCE	资源
3	IMAGE_DIRECTORY_ENTRY_EXCEPTION	异常（具体资料不详）
4	IMAGE_DIRECTORY_ENTRY_SECURITY	安全（具体资料不详）
5	IMAGE_DIRECTORY_ENTRY_BASERELOC	重定位表
6	IMAGE_DIRECTORY_ENTRY_DEBUG	调试信息
7	IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	版权信息
8	IMAGE_DIRECTORY_ENTRY_GLOBALPTR	具体资料不详
9	IMAGE_DIRECTORY_ENTRY_TLS	Thread Local Storage
10	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	具体资料不详
11	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	具体资料不详
12	IMAGE_DIRECTORY_ENTRY_IAT	导入函数地址表
13	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	具体资料不详
14	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	具体资料不详
15	未使用	

区块表

紧跟着 IMAGE_NT_HEADER 后的是区块表，他是一个 IMAGE_SECTION_HEADER 结构数组。每个 IMAGE_SECTION_HEADER 结构包含了它所关联区块的信息，如位置，长度，属性，该数组的数目由 IMAGE_NT_HEADERS.FileHeader.NumberOfSections 指出。

IMAGE_SECTION_HEADER 结构定义如下：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];      //8 个字节的区块名
    union {                                     // 区块尺寸
        DWORD   PhysicalAddress;
        DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;                     // 区块的 RVA 地址
    DWORD   SizeOfRawData;                      // 文件对齐后的尺寸
    DWORD   PointerToRawData;                   // 文件偏移
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;                    // 区块的属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

VirtualSize：指出实际的，被使用的区块大小，是区块在没对齐处理前的实际大小。如果 VirtualSize 大于 SizeOfRawData，那么 SizeOfRawData 是来自可执行文件初始化数据的大小，与 VirtualSize 相差的字节用零填充。

VirtualAddress：该块装载到内存中的 RVA。这个地址是按照内存页对齐的，它的数值总是 SectionAligment 的整数倍。

SizeOfRawData：该块在磁盘文件中所占的大小。在可执行文件中，该字段包含经过 File Alignment 调整后的块的长度。例如：指定 FileAlignment 的大小为 200h，如果 VirtualSize 中的块的长度为 19Ah 个字节，这一块应保存的长度为 200h 个字节。

PointerToRawData：该块在磁盘文件中所占的偏移。程序进编译或汇编后生成原始数据，这个字段用于给出原始数据在文件中的偏移。

输入表

PE 文件头的可选映像头中数据目录表的第二成员指向输入表。输入表以一个 IMAGE_IMPORT_DESCRIPTOR（IID）数组开始。每个被 PE 文件隐式的连接进来的 DLL 都有一个 IID。在这个数组中，没有字段指出该结构数组的项数，但他的最后一个单元是 NULL，可以由此计算出该数组的项数。例如：某个 PE 文件从两个 DLL 文件中引入函数，就存在两个 IID 结构来描述这些 DLL 文件，并在两个 IID 结构的最后由一个内容全为 0 的 IID 结构作为结束。

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {                                 // +00h
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;         // 指向输入名称表 INT
    };
    DWORD   TimeDateStamp;                  //+04h
    DWORD   ForwarderChain;                 //+08h
    DWORD   Name;                           //+0Ch DLL 名称的指针
    DWORD   FirstThunk;                     //10h 指向输入地址表 IAT。
} IMAGE_IMPORT_DESCRIPTOR;
```

OriginalFirstThunk（Characteristics）：包含指向输入名称表（INT）的 RVA，INT 是一个 IMAGE_THUNK_DATA 结构的数据，数组中的每个 IMAGE_THUNK_DATA 结构指向 IMAGE_IMPORT_BY_NAME 结构，数组最后以一个内容为 0 的 IMAGE_THUNK_DATA 的结构结束。

Name：DLL 名字的指针。一个以 00 结尾的 ASCII 字符的 RVA 地址，该字符串包含输入的 DLL 名。

FirstThunk：包含指向输入地址表（IAT）的 RVA，IAT 是一个 IMAGE_THUNK_DATA 结构的数组。

OriginalFirstThunk 和 FirstThunk 很相似。他们指向两个本质上相同的数组 IMAGE_THUNK_DATA，名字叫做输入名称表（Import Name Table,INT）和输入地址表（Import Address Table，IAT）。

两个数组都有 IMAGE_THUNK_DATA 结构类型的元素，他是一个指针大小的联合。每个 IMAGE_THUNK_DATA 元素对应一个从可执行文件输入的函数。两个数组的结束是通过一个值为 0 的 IMAGE_THUNK_DATA 元素来表示的。IMAGE_THUNK_DATA 实际上是一个双字，该结构在不同时刻由不同的含义，定义如下：

```
typedef struct _IMAGE_THUNK_DATA {
    union {
        PBYTE  ForwarderString;             // 指向一个转向者字符串的 RVA
        PDWORD Function;                    // 被输入的函数的内存地址
```

```
        DWORD Ordinal;                          // 被输入的 API 的序数值
        PIMAGE_IMPORT_BY_NAME  AddressOfData;   // 指向 IMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA;
```

当 IMAGE_THUNK_DATA 值的最高位为 1 时，表示函数以序号的方式输入，这是低 31 位（或者一个 64 位可执行文件的低 63 位）被看作是一个函数序号。当双字的最高位为 0 时，表示函数以字符串类型的函数名方式输入，这时双字的值是一个 RVA，指向一个 IMAGE_IMPORT_BY_NAME 结构。

IMAGE_IMPORT_BY_NAME 结构仅一个字大小，村有一个 输入函数的相关信息结构，其结构如下：

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Hint：指示本函数在其所驻留 DLL 的输出表中的序号。该域被 PE 装载器用来在 DLL 的输出表里快速查询函数。

Name：函数输入函数的函数名，函数名是一个 ASCII 码字符串，以 NULL 结尾。虽然这里将 Name 的大小定义为字节，起始它是可变尺寸域。

输入地址表 (IAT)

由 OriginalFirstThunk 所指向的那个数组是单独的一项，而且不可改写，称为 INT。FirstThunk 所指向的数组是由 PE 装载器重写的。PE 装载器搜索 OriginalFirstThunk，如果找到，加载程序迭代搜索数组中的每个指针，找到每个 IMAGE_IMPORT_BY_NAME 结构所指向的输入函数的地址，然后加载器用函数真正入口地址来替代由 FirstThunk 指向的 IMAGE_THUNK_DATA 数组里的元素值。Jmp dword ptr[xxxxxxxx] 中的 [xxxxxxxx] 是指 FirstThunk 数组中的一个入口，因此它称为输入地址表（IAT）。因此，当 PE 文件装在内存后准备执行时，所有函数入口地址被排列在一起，此时，输入表中其他部分就不重要了，程序依靠 IAT 提供的函数地址就可以正常运行。

有些情况下，一些函数仅有序号引出，也就是说，不能用函数名来调用他们，只能用他们的位置来调用。

另一种情况是程序 OriginalFirstThunk 的值为 0。在初始化时，系统根据 FirstThunk 的值找到指向函数名的地址串，由地址串找到函数名，再根据函数名得到入口地址，然后用入口地址取代 FirstThunk 指向的地址串中的原值。

输出表

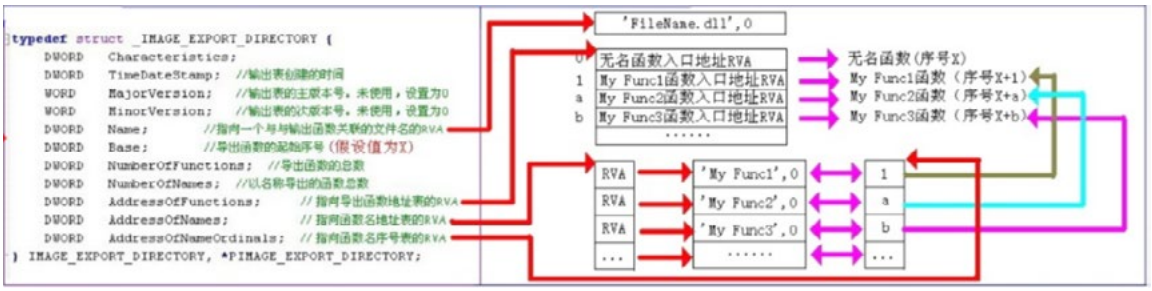
输出表中的主要成分是一个表格，内含函数名称，输出序数等。序数是指定 DLL 中某个函数的 16 位数字，在所指向的 DLL 里是独一无二的。

输出表是数据目录表的第一个成员，指向 IMAGE_EXPORT_DIRECTORY（IED）结构，定义如下：

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD    Characteristics;
```

```
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;                   // 模块的真实名称
    DWORD   Base;                   // 基数，加上序数就是函数地址数组的索引数
    DWORD   NumberOfFunctions;      // AddressOfFunctions 中元素个数
    DWORD   NumberOfNames;          // AddressOfNames 中元素个数
    DWORD   AddressOfFunctions;     // 指向函数地址数组
    DWORD   AddressOfNames;         // 函数名字的指针地址
    DWORD   AddressOfNameOrdinals;  // 指向输出序列号数组
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

输出表的设计是为了方便 PE 装载器工作。首先，模块必须保存所有输出函数的地址，供 PE 装载器查询。模块将这些信息保存在 AddressOfFunctions 域所指向的数组中，而数组元素数目存放在 NumberOfFunctions 域中。如果有些函数是通过名字引出的，那么模块必定也在文件中保留了这些信息。这些名字的 RVA 存放在一个数组中，供 PE 装载器查询。该数组由 AddressOfName 指向，NumberOfNames 包含名字数组。Pe 装载器知道函数名，并想以此获取这些函数的地址。至今为止已有两个模块：名字数组和地址数组，PE 参考指出使用到地址数组的索引为连接两者的纽带，因此 PE 装载器在名字数组中找到匹配名字的同时，它也获取指向地址表中对应元素的索引。这些索引保存在由 AddressOfNameOrdinals 域所指向的另一个数组中。由于该数组起到联系名字和地址的作用，所以起元素数目必定和名字数组相同。



五．Windows 内核加载器

在 Windows NT/XP/2003 系统中，Windows 内核加载器指的是 NTLDR 文件，而在 Vista/Windows 7 中，指的是 bootmgr 文件。这里主要说 ntldr 文件。它位于系统分区的根目录下，如 C:/ntldr，是一个隐藏的，只读的文件，去除“隐藏受保护的操作系统文件”就可以看见。Ntldr 的主要作用是引导和加载操作系统。

Ntldr 是可执行的 16 进制文件，有两部分组成：前半部分是 startup.com，称为 su 模块（一部分是 16 位程序，在实模式下运行，另一部分是 32 位程序，在保护模式下运行）。后半部分是 osloader.exe，称为 loader 模块（32 位程序，主要在保护模式下运行）。

引导驱动器读取第一个扇区到 0x7c00 后，控制权交给 MBR，MBR 代码再搜索系统活动分区表，加载活动分区第一扇区到特定的内存地址（如物理内存地址 0xd000）. 这个扇区称为操作系统分区引导记录（Partition Boot Record，PBR）。MBR 接着将控制权交给 PBR。PBR 代码解析 FAT 和 NTFS 格式找到引导内核的文件 NTLDR，并将 NTLDR 文件加载到指定物理内存地址（0x20000）, 最后将控制权转移交给 NTLDR。NTLDR 的 SU 模块首先获得控制权，前半部分主要是在实模式下工作，检测物理内存，开始 A20 地址线，重定位 GDT 和 IDT。开启保护模式后，SU 解析 osloader.exe 文件，将其加载到物理地址 0x00400000，最后控制权交给 Loader。

5.1 主引导记录 MBR 讲解

MBR 在 windows 启动之前已经被填充好，它并不属于任何一个操作系统。MBR 包含代码和数据两部分，前半部分是启动引导代码，后半部分是一张磁盘分区表，记录每个分区在磁盘上的位置，大小及分区类型。MBR 只有 512B，具有一下特征：

1. 446B 的代码区
2. 64B 的磁盘分区表，分为四个分区表项，每项用 16B 描述
3. 2B 的 MBR 签名，固定为 55AA

MBR 找到 NTLDR 文件在磁盘上的地址后，需要将其加载到物理地址 0x20000，加载的方式是调用一个 BIOS 中断 13h/AH=42h 进行读取。BIOS 中断 13h/AH=42h 一次最多只能读取 127 扇区。

5.2 SU 模块

SU 模块是 NTLDR 文件的前半部分，负责实模式下的初始化工作，为 NTLDR 文件的另一部分 Loader 模块提供支持。SU 模块的主要工作流程如下：

1. 检测物理地址
2. 开启 A20 地址线
3. 重定位 GDT 和 ID T
4. 开启保护模式
5. 加载 Loader 模块
6. 转移控制权

检测物理内存

检测物理内存，实际上借助 BIOS 中断 15H/E820 完成。获得的内存块存储在物理地址 0x70000，是以数组的形式存储。SU 模块并不对物理内存进行管理，它只管收集物理内存，内存管理要到 Loader 之后。

计算机中的物理内存分成若干区域，有些区域是可用的，还有少量区域则被保留。SU 将调用 ConstructMemoryDescriptors 函数来检测可用的内存区域，生成一个内存描述符链表结构，由 MemoryDescriptorList 链表指针所指。这条内存描述链将被 Loader 使用。内存描述链的每个表项都是 MEMORY_DESCRIPTOR 结构，用来描述一块内存区域的基址和长度。内存描述符结构如下：

```
typedef struct _MEMORY_DESCRIPTOR{
    ULONG BlockBase;// 物理块基址
    ULONG BlockSize;// 物理块大小
} MEMORY_DESCRIPTOR,*P MEMORY_DESCRIPTOR;
```

这个结构提供两个字段，Block Base 指明物理内存的起始地址，BlockSize 指明物理内存块的大小。还有一个结构是提供给 BIOS 中断 15H/E820 使用的结构，用来检测和返回物理内存块的信息，它描述一块物理内存的具体信息：

```
E820Frame struc
ErrorFlag dd ? // 错误标志
Key dd ?  // 是否还有物理内存块存在
DescSize dd ?  // 物理内存块描述信息的大小，即 Descriptor 结构大小
BaseAddrLow dd ? // 物理内存块起始地址低 32 位
BaseAddrHigh dd ? // 物理内存块起始地址高 32 位
SizeLow dd ? // 物理内存块大小的低 32 位
SizeHigh dd ? // 物理内存块大小的高 32 位
MemoryType dd ? // 物理内存块的类型，设为 1 时表示可用，为 0 表示不可用
E820Frame ends
ConstructMemoryDescriptors
;ConstructMemoryDescriptors 的 c 语言版本
BOOLEAN ConstructMemoryDescriptors(VOID)
{
    ULONG StartAddr,EndAddr;
    E820FRAME E820Frame;
    MemoryDescriptorList->BlockSize = 0; // 初始化首个链表为 0
```

```
MemoryDescriptorList->BlockBase = 0; // 初始化首个链表为 0
// 循环检测物理地址
do{
    E820Frame.Size = sizeof(E820Frame.Descriptor);
    Int15E820 (&E820Frame);
    if (E820Frame.ErrorFlag || E820Frame.Size(sizeof(E820Frame.Descriptor)) // 这里作者在书上的括号
    写错了
     break;
    // 获取开始地址和结束地址
    StartAddr = E820Frame.Descriptor.BaseAddrLow;
    EndAddr = E820Frame.Descriptor.BaseAddrLow + E820Frame.Descriptor.SizeLow - 1;
    // 高于 4G 的内存并不使用
    if(E820Frame.Descriptor.BaseAddrHigh == 0)
    {
        if(EndAddr < StartAddr)
        {
            //EndAddr 字长是 4B 及 32 位 最多表示 4G
            // 如果 EndAddr  <  StartAddr 表示 EndAddr 溢出了直接设置为 0xFFFFFFFF 及 4G
            // 终于晓得为啥子 32 位只支持 4G 寻址了
            EndAddr = 0xFFFFFFF;
        }
        // 仅需要内存类型为 BiosMemoryUsable (1) 的内存
        if(E820Frame.Descriptor.MemoryType==1)
        {
            // 插入内存描述符链表
            InsertDescriptor(StartAddr,EndAddr - StartAddr +1)
        }
    }
}while (E820Frame.Key);  // 如果 key 不等于 0 说明还有内存块继续循环
return TRUE;
}
```

内存描述链表指针 Memory DescriptorList 指向的第一个表项（MEMORY_DESCRIPTOR 结构）的 BlockBase 和 BlockSize 均初始化为 0，指示第一个内存块为空，也就表示当前没有检测到内存块。

下面进入循环检测物理内存阶段。调用 Int15E820 函数进行物理内存块的枚举（其内部使用 BIOS 中断 INT 15H/E820）。循环条件是 E820Frame.key，当该值不为 0 时，表示还有物理内存块存在，需要继续进行检测，这个 Key 值是调用 INT 15H 后由 ebx 寄存器返回的结果值。

Key 值为 0 时，表示已无物理内存块存在，可以停止检测。调用 Int15E820 函数返回一块物理内存的具体信息，若该物理内存信息无错误，计算该块物理内存的起始地址和结束地址，因为这里最多只支持 4GB 内存，所以内存地址的高字节只能为 0. 若参数值 MemroyType 为 BiosMemoryusable（1），表示物理内存可用，那么调用 InsertDescriptor 函数就可以将物理内存块插入到物理内存描述链表中。

开启 A20 地址线

开启 A20 地址线，可以寻址大于 1MB 的物理内存，如果不开启 A20，后面无法加载 osloader.exe 到物理内存空间 0x400000，并且地址线的第 20 位只能为 0，那么能被访问的内存只能是奇数 M 段，即 1M，3M，5M……。这样可以访问的内存不是连续的。只有开启 A20 地址线，才能访问到连续的内存。

重新定位 GDT 和 IDT

GDT 是一个由段描述符及其他描述符构成的表，而 IDT 是一个中断和异常描述符的表，这两个表的存放位置是紧连着的，在 SU 数据段中已经设置好。GDT 属于保护模式范畴，IDT 属于中断和异常。

Relocatex86Structures 函数用来把在 SU 数据段的 GDT 和 IDT 移到物理地址 0x1e000.

```
Relocatex86Structures:
    push bp
    mov bp,sp
    push si; 保存寄存器
    push di
    push es
; 把指针移动到 GDT 开始，并计算需要移动的大小
    mov si,GDTBase
    mov ax,SYSTEM_PAGE_BASE>>4
    mov es,ax
    xor di,di
    mov cx,IDTEnd-GDTBase
; 用一个循环移动数据到 0x1e000, 循环次数为 IDTEnd-GDTBase
.loop
```

```
        mov al,[si]
es      mov [di],al
        inc si
        inc di
        dec cx
        jnz .loop; 循环复制
; 返回寄存器
        pop es
        pop di
        pop si
        mov sp,bp
        pop bp
        retn
```

保护模式

保护模式是从 80286 系列开始出现的一种新的运行模式。在实模式中，采用的是 16 位地址模式，最多只能寻址 10FFEFh 的地址空间，同时，它的分段是针对所有的物理空间进行的，系统程序和用户程序都能访问素有的地址，如果某个存放了系统程序的内存空间呗用户程序修改了，将会造成无法预料的错误。引入保护模式就是为了解决上述两个缺陷。一方面，在保护模式中，采用的是 32 位地址模式，全部 32 条地址线都能使用，因此最多能寻址 232B=4GB 内存空间。另一方面，保护模式中还引入了段保护机制，使得物理内存不能再被直接访问，程序使用的都是虚拟地址，需要通过操作系统页表将这些虚拟地址转换为物理地址，才能被访问。

开启保护模式

GDT 表已经在 SU 数据段中建立，但是需要让处理器知道在哪里。GDTR 寄存器就是用来专门存放 GDT 表的入口，使用 lgdt 指令加载进 GDTR 寄存器。CPU 以后就根据 GDTR 寄存器来访问 GDT 表。GDTR 是 48 位寄存器，前 16 位指明 GDT 表的大小，后 32 位指明 GDT 的地址。

执行下面代码就能进入保护模式：

```
EnableProtectPaging:
;BIOS 留下的标志寄存器 flags，在进入保护模式前，我们把它设为 0，同时 ES 和 GS 也设为 0
        push dword 0
        popfd
        mov bx,sp
        mov dx,[bx+2]
```

```
    xor ax,ax
    mov gs,ax
    mov es,ax
;FS 指向 PCR
    push KePcrSelector
    pop fs
; 加载 GDT 和 ID T，关中断，因为在进入保护模式前我们暂时不能处理中断
    cli
; 利用指令 lgdt，将 GDT 表加载进 GDTR 寄存器
    lgdt [GDTregister]
; 利用指令 lidt，将 IDT 表加载进 IDT 寄存器
    lidt [IDTregister]
    mov eax,cr0
; 若 dx=0，表示只开启保护模式，不开启分页模式
    or dx,dx
    jz .set_protect
; 设置保护模式和分页机制，将 CR0 寄存器的 bit 0 位置 1 开启保护模式，bit 31 位置 1 开启分页机制
    or eax,PROTECT_MODE|ENABLE_PAGING
    mov cr0,eax
    jmp .set_end
.set_protect:
    or eax,PROTECT_MODE; 开启保护模式
    mov cr0,eax
.set_end:
; 将 SU 代码选择子 58h 加载进 CS 段寄存器
    push SuCodeSelector
    push .restart
    retf
.restart:
```

```
; 将 SU 数据选择子 60h 加载进 DS，SS 段寄存器
    push ax,SuCodeSelector
    mov ds,ax
    mov ss,ax
    xor bx,bx
; 加载局部描述符，只能在保护模式下使用这条指令
    lldt bx
    or dx,dx
    jnz .return
    mov bx,KeTssSelector
;ltr 是一条特权指令，一般在操作系统初始化过程中执行，用来初始化任务寄存器，之后任务寄存器的内容通过每次任务切换来改变
    ltr bx
.return:
    retn
```

进入保护模式后，我们访问的地址一般叫线性地址，段寄存器变成了段描述符的选择子，不再参与地址转译。

加载 Loader 模块

oloader.exe 是标准的 PE 文件，在运行前，必须按照标准格式加载进内存。Osloader.exe 文件在编译链接时嵌套在 startup.com 末尾，所以解析时只需定位到 startup.com 的结尾，即 osloader.exe 文件的开始。首先需要解析 osloader.exe 文件头，然后按内存对齐值复制个个段，复制完成后返回 osloader.exe 映像代码入口。

将 Loader 加载进来之后，SU 把系统控制权移交给 Loader。所谓移交控制权，实际上就是跳转到 osloader.exe 的入口点，即交由 Loader 的 NtProcessstartup 函数去执行。

六. Hook、RootKit

6.1 使用注册表来注入 DLL

在注册表路径 HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersioin\Windows\ 下，AppInit_Dlls 键的值可能会包含一个 DLL 的文件名活一组 DLL 的文件名（通过空格或逗号分隔）。将自己写的 DLL 文件的路径值写入 AppInit_Dlls 中，再创建一个名为 LoadAppInit_Dlls，类型为 DWORD 的注册表项，并将其值设为 1. 当 User32.dll 被映射到一个新的进程时，会受到 DLL_PROCESS_ATTACH 通知。当 User32.dll 对它进行处理的时候，会获取上述注册表键的值，并调用 LoadLibrary 来载入这个字符串中指定的每个 DLL。

6.2 使用 Widows 挂钩来注入 DLL

调用函数 SetWindowsHookEx 来安装钩子，此函数的声明如下：

```
HHOOK WINAPI SetWindowsHookEx(
  _In_  int idHook,
  _In_  HOOKPROC lpfn,
  _In_  HINSTANCE hMod,
  _In_  DWORD dwThreadId
);
```

idHook 表示要安装的挂钩的类型，lpfn 是一个函数的地址，在窗口即将处理一条消息的时候，系统应该调用这个函数，hMod 标识一个 DLL，这个 DLL 包含了 lpfn 函数，dwThreadId 表示要给哪个线程安装挂钩。如果这个参数传 0，表示要给系统中所有 GUI 线程安装挂钩。

例如进程 A 使用 SetWindowsHookEx(WH_GETMESSAGE,GetMsgProc，hInstDll，0) 函数安装挂钩后：

1. 进程 B 中的一个线程准备向一个窗口派送一条消息
2. 系统检查该线程是否安装了 WH_GETMESSAGE 挂钩
3. 系统检查 GetMsgProc 所在的 DLL 是否已经被映射到进程 B 的地址空间中，如果 DLL 尚未被映射，那么系统会强制将该 DLL 映射到进程 B 的地址空间中，并将进程 B 中该 DLL 的锁计数器递增
4. 由于 DLL 的 hInstDll 是在进程 B 中映射的，因此系统会对他进行检查，看他在进程 A 中的位置是否相同，如果相同，那么在两个进程空间中，GetMsgProc 函数位于相同的位置，系统就可以直接在进程 A 的地址空间中调用 GetMsgProc。如果不相同，那么系统必须确定 GetMsgProc 函数在进程 B 的地址空间中的虚拟内存地址。使用公式 GetMsgProc B=hInstDll B+(GetMsgProc A-hInstDll A) 获得
5. 系统在进程 B 中递增该 DLL 的锁计数器
6. 系统在进程 B 的地址空间中调用 GetMsgProc 函数
7. 当 GetMsgProc 返回的时候，系统递减该 DLL 在进程 B 中的锁计数器

6.3 使用远程线程来注入 DLL

1. 使用函数 VirtualAllocEx 在远程进程的地址空间中分配一块内存
2. 使用函数 WriteProcessMemory 函数把 DLL 的路径名复制到第一步分配的内存中
3. 使用函数 GetProcAddress 得到 LoadLibrary 函数的实际地址
4. 使用函数 CreateRemoteThread 函数在远程进程中创建一个线程，让新线程调用正确的 LoadLibrary 函数并在参数中传入第一步分配的内存地址。现在远程进程中有一块内存，它是在第一步分配的，DLL 也还在远程进程的地址空间中。为了对它进行清理，需要在远程线程退出之后执行后续步骤
5. 使用函数 VirtualFreeEx 释放第一步分配的内存
6. 使用函数 GetProcAddress 来得到 FreeLibrary 函数的实际地址
7. 使用函数 CreateRemoteThread 在远程进程中创建一个线程，让该线程调用 FreeLibrary 函数并在参数中传入远程 DLL 的 HMODULE.

6.4 动态库劫持

简单来说就是 DLL 文件替换。通俗说法如下：

A.exe 想要调用 B.dll，并且使用里面的 FunC 函数，这样的话我们把 B.Dll 改名 BB.Dll(有的不用，直接根据路径劫持), 然后我们自己写一个 B.Dll（假的）里面有一个 FunC 这个函数，然后我们在这个函数里加载 BB.Dll（原 B.Dll），并且调用里面的 FunC 函数，之后我们在干一些自己的事，对于 A.exe 来说通常没什么异常感觉，这样我们的目的就达到了，记住此时的你，也就是 B.dll(假的) 的权限和内存归属都是 A 的，也即是你和 A 是一家的了，类似于代码注入之后直接修改内存一样。

WIndows 上的 Dll 加载有一个默认的规则，就是先在主程序目录下查找 B.dll, 如果没有就在系统路径下找，如果还没有，就去环境变量路径里找，就因为这个我们可以轻松的在相应的位置给做劫持，然后问题就是如果实现劫持，就要知道 B.Dll 里面的所有函数名字以及函数参数，这个地方比较不好搞，此地不考虑。

6.5 APC 注入

APC 注入的原理是利用当线程被唤醒时 APC 中的注册函数会被执行的机制，并以此去执行我们的 DLL 加载代码，进而完成 DLL 注入的目的，其具体流程如下：

1. 当 EXE 里某个线程执行到 SleepEx() 或者 WaitForSingleObjectEx() 时，系统就会产生一个软中断
2. 当线程再次被唤醒时，此线程会首先执行 APC 队列中的被注册的函数
3. 利用 QueueUserAPC() 这个 API 可以在软中断时向线程的 APC 队列插入一个函数指针，如果我们插入的是 Loadlibrary() 执行函数的话，就能达到注入 DLL 的目的

6.6 使用 CreateProcess 注入代码

1. 用 CreateProcess 以 CREATE_SUSPENDED 的方式启动目标进程

2. 找到目标进程的入口
3. 将目标进程入口的代码保存起来
4. 在目标进程的入口写 LoadLibrary（MyDll）实现 Dll 注入
5. 用 ResumeThread 运行目标进程
6. 目标进程就运行了 LoadLibrary（MyDll），实现 DLL 的注入
7. 目标进程运行完 LoadLibrary（MyDll）后，将原来的代码写回目标进程的入口
8. 目标进程 jmp 到原来的入口，继续运行程序

InlineHook

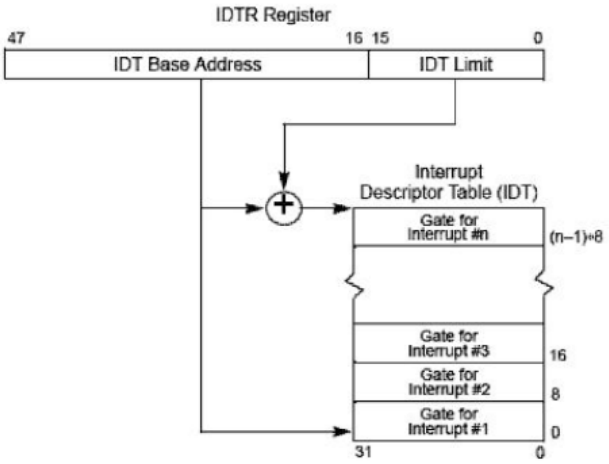
InlineHook 的工作方式如下所示：

1. 在内存中对要拦截的函数进行定位，从而得到它的内存地址
2. 把这个函数起始的几个字节保存到我们自己的内存中
3. 使用 jmp 指令来覆盖这个函数起始的几个字节，这条 jmp 指令用来跳转到我们的替代函数的内存地址。当然，我们的替代函数的函数签名必须与要拦截的函数的函数签名完全相同：所有的参数必须相同，返回值必须相同，调用约定也必须相同
4. 现在，当线程调用被拦截函数的时候，跳转指令实际上会跳转到我们的替代函数。这时，我们就可以执行自己想要执行的任何代码
5. 为了撤销对函数的拦截，需要把第二步保存下来的字节放回被拦截函数起始的几个字节中
6. 我们调用被拦截函数（现在已经不再对它进行拦截了），让函数执行它正常处理
7. 当原来的函数返回时，我们再次执行第二步和第三步，这样替代函数将来还会被调用到

6.7 IDT Hook

IDT=Interrupt Descriptor Table 中断描述表。IDT 是一个有 256 个入口的线形表，每个 IDT 的入口是 8 字节的描述符，所以整个 IDT 表的大小为 256*8=2048 bytes，每个中断向量关联了一个中断处理过程。所谓的中断向量就是把每个中断或者异常用一个 0-255 的数字识别。Intel 称这个数字为向量 (vector)。

对于中断描述表，操作系统使用 IDTR 寄存器来记录 idt 位置和大小。IDTR 寄存器是 48 位寄存器，用于保存 idt 信息。其中低 16 位代表 IDT 的大小，大小为 7FFH，高 32 位代表 IDT 的基地址。我们可以利用指令 sidt 读出 IDTR 寄存器中的信息，从而找到 IDT 在内存中的位置。



IDT 有三种不同的描述符或者说是入口，分别是：

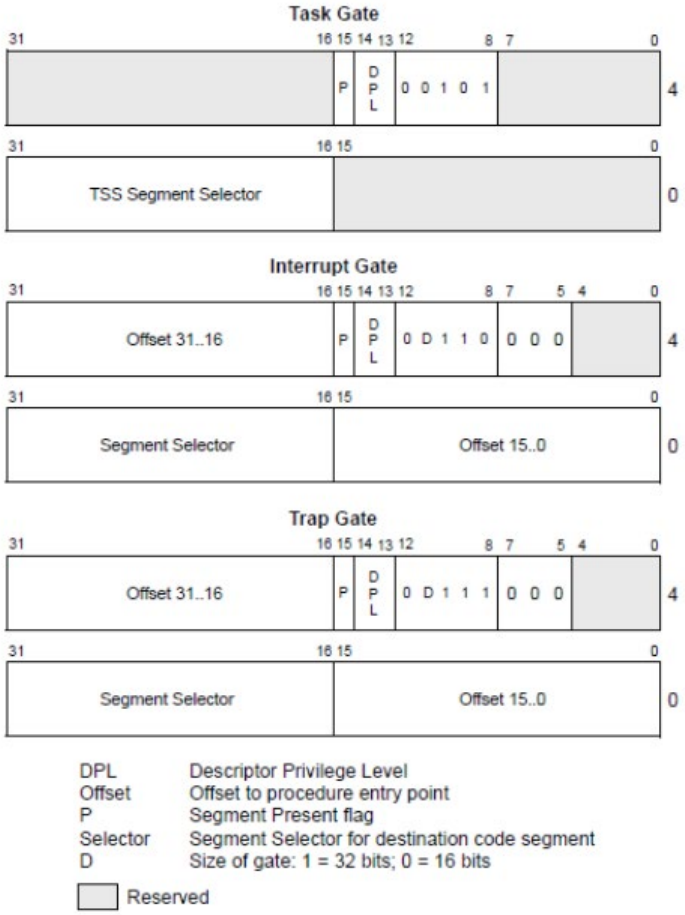
1. 任务门描述符
2. 中断门描述符
3. 陷阱门描述符

也就是说，在保护模式下，80386 只有通过中断门、陷阱门或任务门才能转移到对应的中断或异常处理程序。

中断分为两种类型：可屏蔽中断 -- 它在短时间片段里可被忽略；不可屏蔽中断 -- 它必须被立即处理。例如：硬件失败为不可屏蔽中断，IRQS（中断请求）失败为可屏蔽中断。

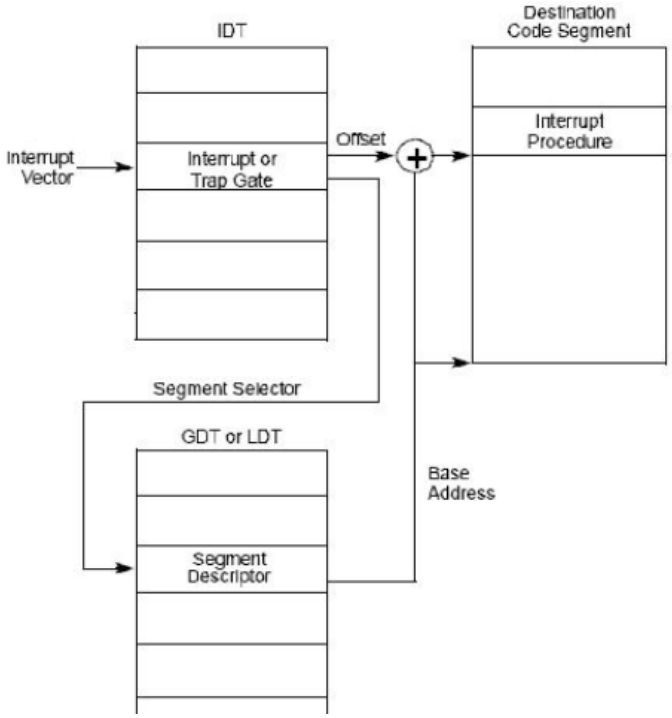
异常被分为不同的两类：处理器产生的异常 (Faults, Traps, Aborts) 和编程安排的异常（用汇编指令 int or int3 触发）。后一种就是我们经常说到的软中断。

下图是三种描述符的图示：

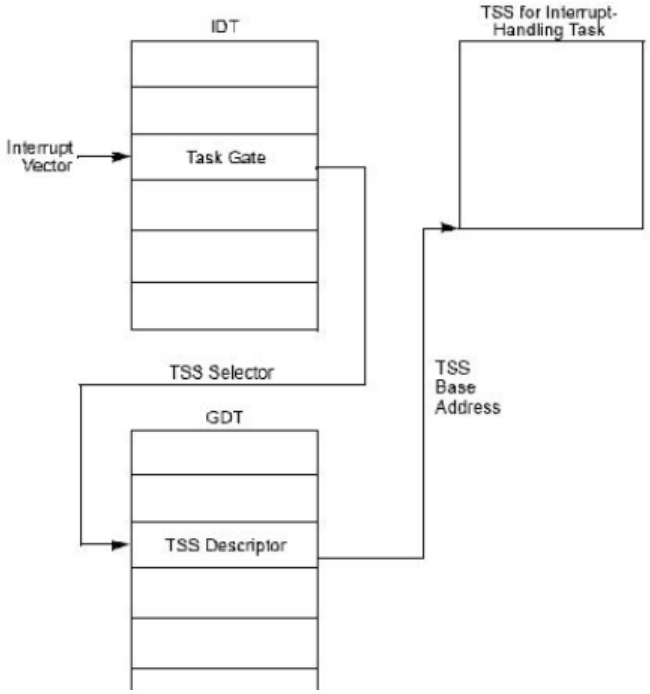


其中：后两种描述符，非常的相似，只有 1 个 bit 位的差别。在处理上，采用相同的处理方式。如图所示，在这后两类的描述符里面记录了一个中断服务程序（ISR) 的地址 offset. 在 IDT 的 256 个向量中，除 3 个任务门入口外，其他都是这两种门的入口。并且所有的 trap/interrupt gate 的入口，他们的 segment selector 都是一样的，即：08h. 我们察看 GDT 中 Selector = 8 的描述符，描述的是 00000000h ~ 0ffffffffh 的 4G 地址空间。 因此，在描述符中的中断服务程序（ISR) 的地址 offset 就代表了函数的入口地址。windows 在处理的时候，按照下图方式，来处理这两类的描述符入口。即：根据 segment selector 在 GDT 中找出段基地址等信息，然后跟描述符中的中断服务程序（ISR) 的地址 offset 相加得到代码段中的函数入口地址。然后调用该函数。

这个过程，我写得比较直接，在操作系统执行这过程时，还有很多的出错判断和异常保护，这里我们略过。

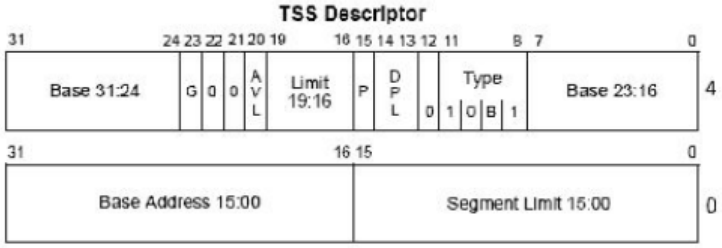


下图是任务门描述符的情况



首先，根据 IDT 中任务门描述符的 TSS Segment Selector , 我们在 GDT 中找出这个选择子。在这个选择子中，对应一个 tss 描述符，即：任务状态段描述符。这个描述符大小为 068h, 即 104 字节。

下面是这个任务状态段描述符的格式。



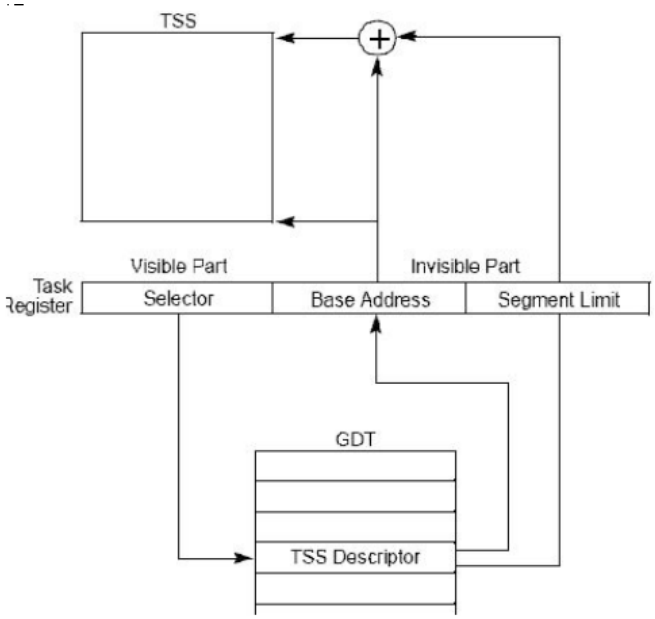
在这个描述符中记录了任务状态段的位置和大小。

根据任务状态段描述符中的 base Address, 找到 TSS 的内存位置。然后就可以进行任务切换。所谓任务切换是指，挂起当前正在执行的任务，恢复或启动另一任务的执行。在任务切换过程中，首先，处理器中各寄存器的当前值被自动保存到 TR 所指定的 TSS 中；然后，下一任务的 TSS 的选择子被装入 TR；最后，从 TR 所指定的 TSS 中取出各寄存器的值送到处理器的各寄存器中。由此可见，通过在 TSS 中保存任务现场各寄存器状态的完整映象，实现任务的切换。 TR 寄存器可见部分保存了 tss selector, 不可见部分，保存了任务状态段的位置和大小 .

任务状态段 TSS 的基本格式如下图所示。

	BIT31-BIT16	BIT15-BIT1	BIT0	Offset
任务状态段基本部分的格式	0000000000000000	链接字段		0
	ESP0			4
	0000000000000000	SS0		8
	ESP1			0CH
	0000000000000000	SS1		10H
	ESP2			14H
	0000000000000000	SS2		18H
	CR3			1CH
	EIP			20H
	EFLAGS			24H
	EAX			28H
	ECX			2CH
	EDX			30H
	EBX			34H
	ESP			38H
	EBP			3CH
	ESI			40H
	EDI			44H
	0000000000000000	ES		48H
	0000000000000000	CS		4CH
	0000000000000000	SS		50H
	0000000000000000	DS		54H
	0000000000000000	FS		58H
	0000000000000000	GS		5CH
	0000000000000000	LDTR		60H
	I/0 许可位图偏移	000000000000000		64H

从图中可见，TSS 的基本格式由 104 字节组成。这 104 字节的基本格式是不可改变的，但在此之外系统软件还可定义若干附加信息。



知道了 IDT 的基本知识后，再来理解 IDT Hook 的原理就比较简单了。就是将将系统原来的中断处理函数地址替换为我们自己的函数的地址。这样系统在处理相应的中断时，就会调用我们的处理函数。

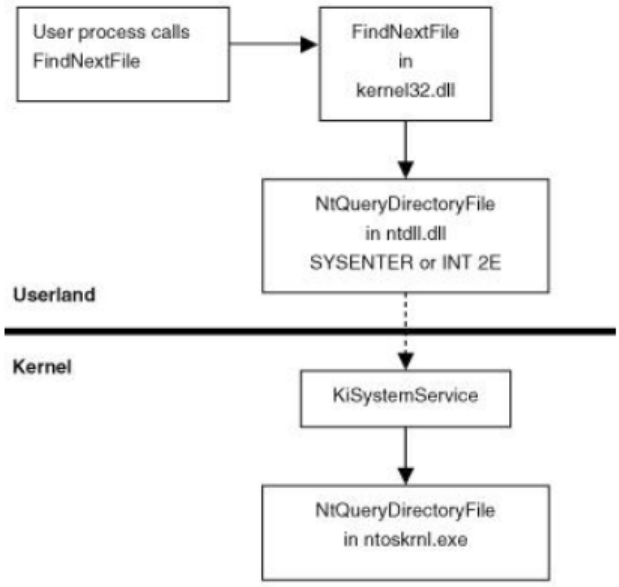
比如：出现页错误，调用 IDT 中的 0x0E。或用户进程请求系统服务 (SSDT) 时，调用 IDT 中的 0x2E。而系统服务的调用是经常的，这个中断就能触发。所以方法就是先在系统中找到 IDT，然后确定 0x2E 在 IDT 中的地址，最后用我们的函数地址去取代它，这样以来，用户的进程（可以特定设置）调用系统服务，我们的 hook 函数即被激发。

使用 sidt 指令可以在内存中找到 IDT 的地址，返回一个 IDTINFO 结构的地址。这个结构中国含有 IDT 的高半地址和低半地址。IDT 有最多 256 个入口。将 IDT 看作 是一排有 256 间房组成的线性结构，那么只要知道了整个入口结构，就相当于知道了每间房的长度，先获取所有的入口 idt_entrys，那么第 0x2E 个房间的地址就可以确定了。即 idt_entrys[0x2E]。找到目标入口后，将我们的函数与其原来的函数进行替换即可。

6.8 SSDT Hook、SSSDT Hook

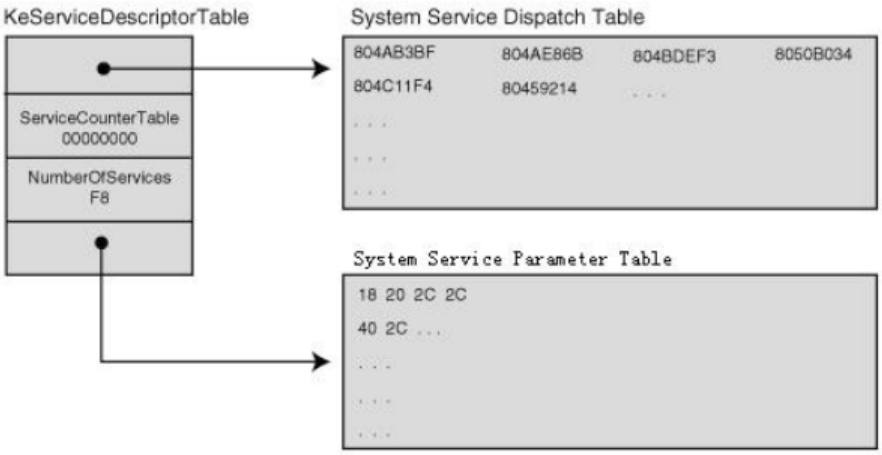
SSDT 既 System Service Dispath Table。在 Windows NT 下， NT 的 executive（ NTOSKRNL.EXE 的一部分）提供了核心系统服务。由于子系统不同， API 函数的函数名也不同。 例如，要用 Win32API 打开一个文件，应用程序会调用 CreateFile()，而要用 POSIXAPI，则应用程序调用 open() 函数。这两种应用程序最终都会调用 NT executive 中的 NtCreateFile() 系统服务。

用户模式（ User mode）的所有调用，如 Kernel32,User32.dll,Advapi32.dll 等提供的 API， 最终都封装在 Ntdll.dll 中 , 然后通过 Int 2E 或 SYSENTER 进入到内核模式， 通过服务 ID, 在 System Service DispatcherTable 中分派系统函数。例如下图：



SSDT 就是一个表，这个表中有内核调用的函数地址。从上图可见，当用户层调用 FindNextFile 函数时， 最终会调用内核层的 NtQueryDirectoryFile 函数， 而这个函数的地址就在 SSDT 表中， 如果我们事先把这个地址改成我们特定函数的地址，那么就实现了 SSDT Hook。

下面来介绍以下 SSDT 的结构：



KeServiceDescriptorTable 是由内核（ntoskrnl.exe）导出的一个表，这个表是访问 SSDT 的关键，结构形式如下:

```
typedef struct ServiceDescriptorTable {
PVOID ServiceTableBase;
PVOID ServiceCounterTable(0);
unsigned int NumberOfServices;
PVOID ParamTableBase;
```

}

ServiceTableBase： System Service Dispatch Table 的基地址。

NumberOfServices ：由 ServiceTableBase 描述的服务的数目。

ServiceCounterTable：此域用于操作系统的 checked builds，包含着 SSDT 中每个服务被调用次数的计数器。这个计数器由 INT 2Eh 处理程序 (KiSystemService) 更新。

ParamTableBase： 包含每个系统服务参数字节数表的基地址。

System Service Dispath Table（ SSDT）：系统服务分发表，给出了服务函数的地址，每个地址 4 子节长。

System Service Parameter Table(SSPT)：系统服务参数表，定义了对应函数的参数字节，每个函数对应一个字节。如在 0x804AB3BF 处的函数需 0x18 字节的参数。

要对 SSDT 进行 Hook，首先需要改变 SSDT 的内存保护，因为系统对 SSDT 都是只读的，不能写。如果视图去写，就会造成蓝屏。一般可以修改内存的方法有通过 cr0 寄存器和 Memory Descriptor List（MDL）。

通过 cr0 寄存器：

Windows 对内存的分配，是采用的分页管理，其中有个 cr0 寄存器，其中第一位叫做保护属性位，控制着页的读或写属性。如果为 1，则可以读 / 写执行；如果为 0，则只可以读执行。所以我们要将这一位设为 1.

通过 MDL

将原来的 SSDT 的区域映射到我们自己的 MDL 区域中，并把这个区域设置成可写就行了。

接下来获得 SSDT 中函数的地址。使用四个有用的宏。

SYSTEMSERVICE macro：可以获得由 ntoskrnl.exe 导出函数 , 以 Zw* 开头函数的地址， 这个函数的返回值就是 Nt* 函数， Nt* 函数的地址就在 SSDT 中。

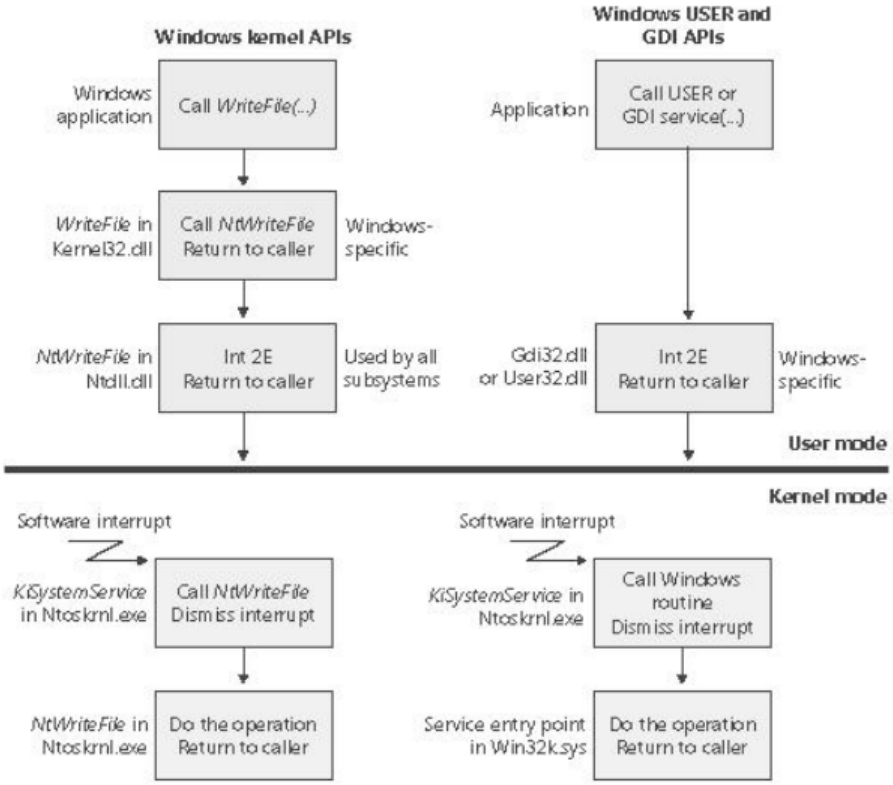
SYSCALL_INDEXmacro： 获得 Zw* 函数的地址并返回与之通信的函数在 SSDT 中的索引。这两个宏之所以能工作，是因为所有的 Zw* 函数都开始于 opcode： MOV eax, ULONG， 这

里的 ULONG 就是系统调用函数在 SSDT 中的索引。

HOOK_SYSCALL 和 UNHOOK_SYSCALLmacros： 获得 Zw* 函数的地址， 取得他的索引，

自动的交换 SSDT 中索引所对应的函数地址和我们 hook 函数的地址。

还有一个这样的表，叫做 KeServiceDescriptorTableShadow，它主要包含 GDI 服务，也就是我们常用的窗口，桌面相关，具体存在于 Win32k.sys。如下图：



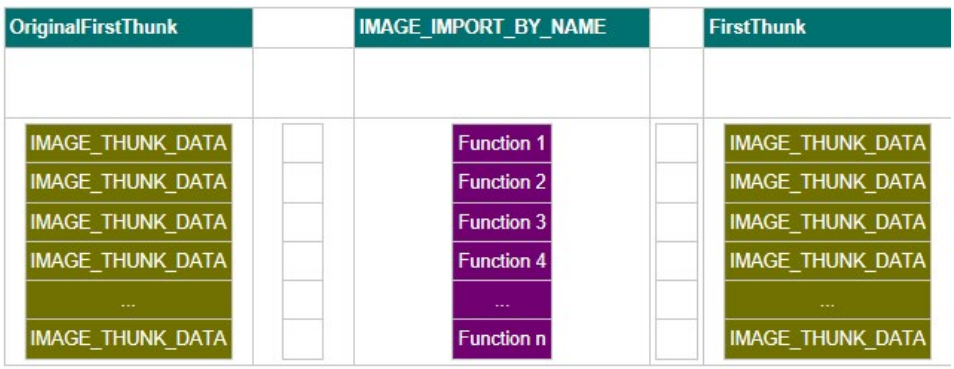
右侧的服务分布就通过 KeServiceDescriptorTableShadow。

SSSDT Hook 和 SSDT Hook 的方式差不多，在此不再进行介绍。

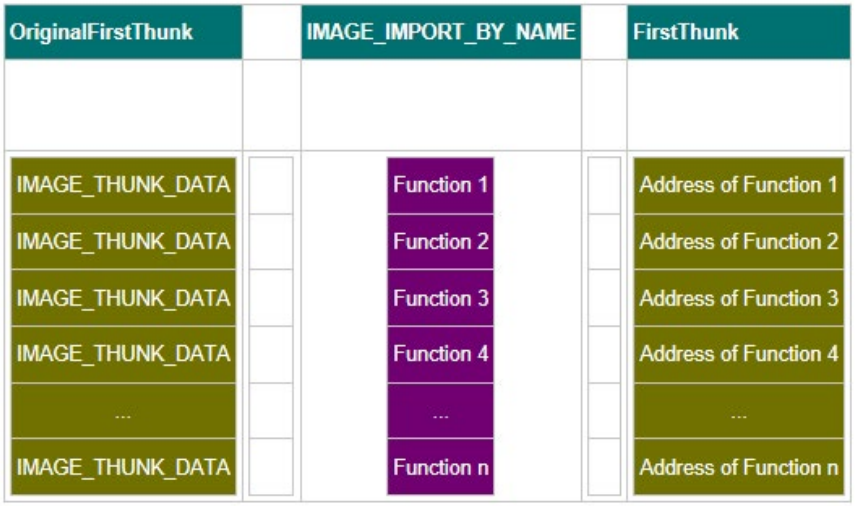
6.9 IAT Hook

IAT 即 Import Address Table 是 PE（可以理解为 EXE）的输入地址表，我们知道一个程序运行时可以要调用多个模块，或者说要调用许多 API 函数，但这些函数不一定都在 EXE 本身中，例如你调用 Messagebox 来显示一个对话框时，你只需要调用它，你并没有编写 Messagebox 的函数的实现过程，Messagebox 的函数的实现过程实际上是在 user32.dll 这个库文件中，当这个程序运行时会在 user32.dll 中找到 Messagebox 并调用它。

下图是导入表中的部分结构图：



IMAGE_THUNK_DATA 指向 IMAGE_IMPORT_BY_NAME 结构的 RVA，OriginalFirstThunk 和 FirstThunk 所指向的这两个数组大小取决于 PE 文件从 DLL 中引入函数的数目。当 PE 文件被装载到内存时，PE 装载器将查找 IMAGE_THUNK_DATA 和 IMAGE_IMPORT_BY_NAME 这些结构数组，以此决定引入函数的地址。然后用引入函数真实地址来替代由 FirstThunk 指向的 IMAGE_THUNK_DATA 数组里的元素值。因此当 PE 文件准备执行时，上图已转换下图所示：



所以 IAT Hook 的原理就是把后面的目标函数的地址改成我们自己写的函数的地址。这样，当在此调用目标函数的时候，就会调用我们的函数的地址。

6.10 EAT Hook

函数导入的函数的地址是再运行时候才确定的，比如我们的一个驱动程序导入了 PsGetCurrentProcessId 这个 ntkrnlpa.exe 导出的函数，那在我们驱动程序加载运行的时候，装载程序会确定 ntkrnlpa.exe 在内存的基地址，接着遍历它的导出表，在 AddressOfNames 指向的 " 函数名字表 " 中找到 PsGetCurrentProcessId 的位置，也就是如果在 AddressOfNames[i] 中找到 PsGetCurrentProcessId, 那就用 i 在 AddressOfNameOrdinals 中索引，假使得到是 X, 那么 AddressOfFunctions[index] 的值就是 PsGetCurrentProcessId 的 RVA 了，最后就可以知道 PsGetCurrentProcessId 在内存的值是 MM=ntkrnlpa.exe 在内存的基地址 +PsGetCurrentProcessId 的 RVA, 然后转载程序就把这个值写到我们驱动程序的 IAT 中，好了知道这些后 ,EAT HOOK 就是修改 PsGetCurrentProcessId 的 RVA, 使得 PsGetCurrentProcessId 的 RVA(修改后的)+ntkrnlpa.exe 在内存的基地址 = 我们自己函数的值，这样装载程序会把我们的函数的地址写入那些调用 PsGetCurrentProcessId 的驱动程序的 IAT, 那么当那些驱动程序调用 PsGetCurrentProcessId 时，实际上是执行了我们自己的函数

七．断点

7.1 软件断点

x86 系列处理器从其第一代英特尔 8086 开始就提供了一条专门用来支持调试的指令，即 INT3。简单的说，这条指令的目的就是使 CPU 中断（break）到调试器，以供调试者对执行现场进行各种分析。当我们调试程序时，可以在可能有问题的地方插入一条 INT3 指令，使 CPU 执行到这一点时停下来。这便是软件调试中经常用到的断点功能，因此 INT3 指令又称为断点指令。

当我们在调试器中对代码的某一行设置断点时，调试器会把这里的本来指令的第一个字节保存起来，然后写入一条 INT3 指令。因为 INT3 指令的机器码是 0xCC，仅有一个字节，所以设置和取消断点时也只需要保存和恢复一个字节。

当 CPU 执行到 INT3 指令时，由于 INT3 指令的设计目的就是中断到调试器，因此，CPU 执行执行这条指令的过程也就是产生断点异常并转去执行异常处理的过程。在跳转到处理历程之前，CPU 会保存当前的执行上下文，包括段寄存器，程序指针寄存器等内容。

在保护模式下，在保存当前执行上下文之后，cpu 会从 IDTR 寄存器中获得 IDT 的地址，在 IDT 表中查询异常处理函数。

在 Windows 系统中，INT 3 异常处理函数是操作系统内核函数 KiTrap03。因此遇到 INT 3 会导致执行 nt!KiTrap03 函数。因为我们现在讨论的是应用程序调试，断点指令位于用户模式下的应用程序代码中，因此 CPU 会从用户模式转入内核模式。接下来，经过几个内核函数的分发和处理，因为这个异常来自用户模式，而且该异常的拥有进程正在被调试，所以内核例程会把这个异常通过调试子系统以调试事件的形式分发给用户模式的调试器，通知完用户模式调试器后，内核的调试子系统函数会等待调试器的回复。受到调试器的回复后，调试子系统的函数会层层返回，最后返回到异常处理例程，异常处理例程执行中断返回指令，使被调试的程序继续执行。

在调试器收到调试事件后，会根据调试事件数据结构中的程序指针，得到断点发生的位置，然后在自己的断点列表中寻找与其匹配的项。如果能找到说明是自己设置的断点。如果找不到，则说明导致这个异常的 INT 3 指令不是自己放进去的。会告诉用户：一个用户插入的断点被触发了。

在调试器下，我们看不到动态替换到程序的 INT 3 指令。大多数调试器的做法是在调试目标被中断到调试器之前，会先将所有断点位置被替换为 INT 3 的指令恢复成原来的指令，然后再把控制权交给用户。

当用户结束分析希望恢复被调试程序时，调试器通过调试 API 通知调试子系统，这会导致系统内核的异常分发函数返回到异常处理例程，然后异常处理例程通过 IRET/IRETD 指令触发一个异常返回动作，使 CPU 恢复执行上下文，从发生异常的位置继续执行。注意，这时的程序指针是指向断点所在的那条指令的，此时刚才的断点指令已经被替换成本来的指令，于是程序会从断点位置的原来指令继续执行。

软件断点虽然使用方便，但是也有局限性：

- 属于代码类断点，即可以让 CPU 执行到代码段内的某个地址是停下来，不使用于数据段和 I/O 控件
- 对于 ROM 中执行的程序，无法动态增加软件断点。因为目标内存是只读的，无法动态写入断点指令。这时就需要使用硬件断点。
- 在中断向量表或中断描述表（IDT）没有准备好或遭破坏的情况下，这类断点是无法或不能正常工作的，比如系统刚刚启动时或 IDT 被病毒篡改后，这时只能用硬件级的调试工具

7.2 硬件断点

IA-32 处理器定义了 8 个调试寄存器，分别称为 DR0-DR7。在 32 位模式下，他们都是 32 位的；在 64 位模式下，都是 64 位的。下面以 32 位的情况来介绍。

DR4 和 DR5 是保留的，当调试扩展功能被启用（CR4 寄存器的 DE 位设为 1）时，任何对 DR4 和 DR5 的引用都会导致一个非法指令异常，当此功能被禁止时，DR4 和 DR5 分别是 DR6 和 DR7 的别名寄存器，即等价于访问后者。其他的 6 个寄存器：

- 4 个 32 位的调试地址寄存器（DR0-DR3），64 位下是 64 位的
- 1 个 32 位调试控制寄存器（DR7），64 位时，高 32 位保留未用
- 1 个 32 位调试状态寄存器（DR6），64 位时，高 32 位保留未用

通过以上寄存器可以最多设置 4 个断点，基本分工是 DR0-DR3 用来指定断点的内存（线性地址）或 I/O 地址。DR7 用来进一步定义断点的中断条件。DR6 的作用是当调试事件发生时，向调试器报告事件的详细信息，以供调试器判断发生的是何种事件。

调试地址寄存器（DR0-DR3）用来指定断点的地址。对于设置在内存中的断点，这个地址应该是断点的线性地址而不是物理地址，因为 CPU 是在线性地址被翻译为物理地址之前来做断点匹配工作的。这意味着，在保护模式下，我们不能使用调试寄存器来针对一个物理内存地址设置断点。

调试控制寄存器（DR7）中，有 24 位被划分成四组分别与四个调试地址寄存器相对应。

R/W0-R/W3：读写域，分别与 DR0-DR3 四个调试地址寄存器相对应，用来指定被监控的访问类型，含义如下：

- 00：仅当执行对应地址的指令时中断
- 01：仅当向对应地址写数据时中断
- 10：当向相应地址进行输入输出（即 I/O 读写）时中断
- 11：当向相应地址读写数据时都中断，但是从该地址读取指令除外

LEN0-LEN3：长度域。分别与 DR0-DR3 四个调试地址寄存器相对应，用来指定要监控的区域长度，含义如下：

- 00：1 字节长
- 01：2 字节长
- 10：8 字节长或未定义（其他处理器）
- 11：4 字节长

L0-L3: 局部断点启用：分别与 DR0-DR3 四个调试地址寄存器相对应，用来启用或禁止对应断点的局部匹配，如果该值设为 1，当 CPU 在当前任务中检测到满足所定义的断点条件时便中断，并且自动清除此位，如果该位设为 0，便禁止此断点。

G0-G3：全部断点启用。分别与 DR0-DR3 四个调试地址寄存器相对应，用来全局启用和禁止对应的断点。如果该位设为 1，当 CPU 在任何任务中检测到满足所定义的断电条件时都会中断；如果该位设为 0，便禁止此断点。与 L0-L3 不同，断点条件发生时，CPU 不会自动清除此位。

LE 和 GE：这个在 P6 以下系列 CPU 上不被支持，在升级版的系列里面：如果被置位，那么 cpu 将会追踪精确的数据断点。LE 是局部的，GE 是全局的。

GD：启用或禁止调试寄存器的保护。当设为 1 时，如果 CPU 检测到将修改调试寄存器（DR0-DR7）的指令时，CPU 会在执行这条指令前产生一个调试异常。

我们可以通过设置读写域来指定断点的访问类型。读写域占两个二进制位，可以指定 4 中访问方式。这里介绍三种典型的方式：

读 / 写内存中的数据时中断：这种断点又被称为数据访问断点。利用数据访问断点，可以监控全局变量或局部变量的读写操作。

执行内存中的代码时中断：这种断点又被称为代码访问断点或指令断点。代码访问断点在实现的功能上看与软件断点类似，都是当 CPU 执行指定地址开始的指令时中断。但是通过寄存器实现的代码访问断点与软件断点相比有个优点，就是不需要像软件断点那样像目标代码中插入断点指令。例如：当我们调试位于 ROM 上的代码（比如 BIOS 中的 POST 程序）时，根本没有办法向那里插入软件断点（INT3）指令，因为目标内存是只读的。另外，软件断点的另一个局限性是，只有当目标代码被加载进内存后才可以向该区域设置软件断点。而调试寄存器断点没有这些限制，因为只要把需要终端的内存地址放入调试地址寄存器（DR0-DR3），并设置好调试控制寄存器（DR7）的相应位就可以了。

读写 I/O（输入输出）端口时中断：这种断点又被称为 I/O 访问断点。I/O 访问断点对于调试使用输入输出端口的设备驱动程序非常有用。也可以利用 I/O 访问断点来监视 I/O 空间的非法读写操作，提高系统的安全性。因为某些恶意程序在实现破坏动作时，需要对特定的 I/O 端口进行读写操作。

7.3 条件断点

条件断点是一个带有条件表达式的普通 INT3 断点，是软件断点的一种，只有某些条件得到满足时这个断点才能中断执行程序。对于频繁调用的 API 函数，仅当特定参数传给他时才中断程序执行，这种情况下，条件断点特别有用，它可以节省调试的时间。

7.4 内存断点

改变内存分页的属性，如内存访问断点设为不可访问属性。由于分页粒度的限制，无法保证精度，最小改变一页的属性，不过内存断点不改变指令，不会被自校验检测到，并且没有个数限制，同时可以对一整段内存下断。归属于硬件断点。

八．调试器的原理

本文介绍的原理以大家所熟知的 OllyDbg 为例进行讲解。

Ollydbg 的断点功能是基于异常处理来实现的，通过捕获程序执行过程中的异常信息来中断程序的执行流程。Ollydbg 常用的断点类型有三种：INT3 断点，内存断点，硬件断点。每种断点都是一种制造异常的方法，首先使程序在运行过程中产生错误，然后由 Ollydbg 的异常处理来接管，从而实现断点的功能。

8.1 加载调试程序

调试程序的第一步就是使用 OllyDbg 来加载程序，加载的过程是通过创建新进程来完成的。OllyDbg 通过 CreateProcess 以调试的方式开启新进程。在创建调成程序前，OllyDbg 需要进行一些必要的检查工作。

首先是针对快捷方式的检查。OllyDbg 根据可执行程序的后缀名来判断分析程序是否为一个快捷方式，如果是快捷方式，则会找到这个快捷方式所对应的可执行程序的全路径。通过检查 DOS 头与 NT 头来判定分析文件是否为合法的 PE 文件。当调试文件为 DLL 动态库时，Olly/Dbg 会使用自带的 LoadDll.exe 将 Dll 文件进行加载。当调试文件为 exe 可执行程序时，会跳过 Dll 文件的处理部分，直接获取相关的配置文件信息并进行加载和调试。

8.2 异常处理机制

异常就是程序运行过程中产生的错误。OllyDbg 利用异常机制捕获调试程序在运行过程中产生的异常，对异常进行排查，从而实现断点功能，使程序暂停运行。OllyDbg 将异常处理过程放置在一个大消息循环中，捕获异常的流程如下：

1. 进入消息循环
2. 利用 WaitForDebugEvent 函数捕获异常信息，如果捕获失败，则回到循环起始处
3. 捕获到异常，率先由 OllyDbg 插件进行异常处理
4. 检查是否为调试异常，如果不是，则继续执行程序，回到循环起始处
5. 如果是调试异常，则进行相关检查，进入断点异常处理函数中

当进入最后一步时，程序已经被成功断下，调试程序出于挂起状态，等待调试者的处理。异常处理首先检查调试事件类型，如果调试信息为异常，则进入异常处理部分，判断异常类型。先判断异常是否为 INT3 断点所产生的，如果是，则通过跳转指令执行对应的代码。下面介绍 INT3 断点的捕获过程：OllyDbg 将调试程序停留在正确的 INT3 断点处，在显示反汇编代码的过程中，没有直接显示断点处机器码 0xCC 或 0xCD，而是通过查找断点信息表中所对应的原机器码的信息来进行显示，以防止因修改指令造成的指令混乱。

在调试人员发出在此运行的指令后，OllyDbg 将会先修复 INT3 断点处的内存数据，然后再次运行修复后的指令代码。INT3 断点处的指令被执行后，此处将会被再次设置为 INT3 断点。

如果检测 INT3 断点失败，则会开始内存断点的异常检查。内存断点的设置过程是通过修改内存属性来达到触发异常的目的的。因此，内存断点的触发便是内存访问类错误。其流程如下：

1. 得到线程信息
2. 跳转到相应的异常处理分支

3. 若得到线程信息，则根据线程信息的 eip 进行赋值，否则根据异常地址进行赋值
4. 得到异常所处的模块的信息，并解析反汇编信息，以进行相关检查
5. 若模块为自解压（SFX）模式，则进行相应的检查以及错误处理
6. 检查内存断点是否在 kernel32.dll 中，弹出提示窗口，并将断点去除
7. 最后调整优先级并退出

硬件断点的捕获过程是由调试寄存器来完成的，因此 OllyDbg 没有捕获处理过程。

8.3 INT3 断点

INT3 断点是最常用的断点，其工作流程时通过修改机器码为 0xCC 来制造异常。当程序执行 0xCC 代码时会触发 INT3 异常，OllyDbg 将捕获此异常并等待用户的处理。跳过 INT3 断点则是将 0xCC 处的代码恢复，在此运行，以保证程序的正常运行。

OllyDbg 实现 INT3 断点的主要流程如下：

1. 检查 INT3 断点是否在记录的断点信息表中
2. 将 INT3 断点信息记录到表中
3. 记录 INT3 断点处的机器码信息
4. 将 INT3 断点处的机器码修改为 0xCC
5. 设置断点信息表

8.4 内存断点

内存断点用来监控内存，它可以对内存数据的访问和写入进行监控。内存断点的设置主要依靠两个 API 来完成：VirtualQuery 和 VirtualProtectEx。通过 VirtualQuery 来获取原内存页的属性，以便于还原；通过 VirtualProtectEx 修改内存页属性，以制造内存访问异常。被调试的目标进程发生异常后，首先处理这个异常的是调试器。因此调试器可以成功捕获这个异常。内存断点的处理过程是由异常处理部分来完成。

8.5 硬件断点

在寄存器中，有一些寄存器专门用于调试，称为调试寄存器，调试寄存器一共有 8 个：Dr0-Dr7；对于 Dr0-Dr3 四个寄存器，作用是存放中断的地址，Dr4 和 Dr5 一般不使用，保留，Dr6 和 Dr7 这两个寄存器的作用是用来记录 Dr0-Dr3 中下断的地址的属性，比如：对这个 401000 是硬件读还是写，或者是执行；是对字节还是对字，或者是双字。

关于硬件断点的详细信息，请参阅断点部分的硬件断点知识。

8.6 单步执行

SEH 即结构化异常处理（Structured Exception Handling），当程序出现错误时，系统把当前的一些信息压

入堆栈，然后转入我们设置好的异常处理程序中执行，在异常处理程序中我们可以终止程序或者修复异常后继续执行。

异常处理处理分两种，顶层异常处理和线程异常处理，下面介绍的是线程异常处理。每个线程的 FS: [0] 处都是一个指向包含异常处理程序的结构的指针，这个结构又可以指向下一个结构，从而形成一个异常处理程序链。当发生异常时，系统就沿着这条链执行下去，直到异常被处理为止。

下面以最常见的 OllyDbg 调试器为例讲解调试器单步执行时的工作方式。

当在调试器中选择“步过”某条指令时，程序自动在下一 条语句停下来，这其实也属于一种中断，而且可以说是最常用的一种形式了，当我们需要对某段语句详细分析，想找出程序的执行流程和注册算法时必须要进行这一 步。是 80386 以上的 INTEL CPU 中 EFLAGS 寄存器，其中的 TF 标志位表示单步中断。当 TF 为 1 时，CPU 执行完一条指令后会产生单步异常，进入异常处理程序后 TF 自动置 0。调试器通过处理这个单步异常实现对程序的中断控制。持续地把 TF 置 1，程序就可以每执行一句中断一次，从而实现调试器的单步跟踪功能。

单步执行中包含 StepIn 和 StepOver 两种：

StepIn：

StepIn 即逐条语句执行，遇到函数调用时进入函数内部，其实现方式如下：

1. 通过调试符号获取当前指令对应的行信息，并保存该行的信息
2. 设置 TF 位，开始 CPU 的单步执行
3. 在处理单步执行异常时，获取当前指令对应的行信息，与 1 中保存的行信息进行比较。如果相同，表示仍然在同一行上，转到 2；如果不相同，表示已到了不同的行，结束 StepIn

StepOver：

StepOver 即逐条语句执行，遇到函数调用时不进入函数内部，其实现方式如下：

1. 通过调试符号获取当前指令对应的行信息，并保存该行的信息
2. 检查当前指令是否 CALL 指令。如果是，则在下一条指令设置一个断点，然后让被调试进程继续运行；如果不是，则设置 TF 位，开始 CPU 的单步执行，跳到 4
3. 处理断点异常时，恢复断点所在指令第一个字节的内容。然后获取当前指令对应的行信息，与 1 中保存的行信息进行比较，如果相同，跳到 2；否则停止 StepOver
4. 处理单步执行异常时，获取当前指令对应的行信息，与 1 中保存的行信息进行比较。如果相同，跳到 2；否则停止 StepOver

《2017 绿盟科技恶意样本分析手册 - 理论篇》
由如下部门撰写

- 绿盟科技安全能力中心（SAC）

如需了解更多，请联系：

官方网站

技术博客

微信公众号

特别声明

为避免客户数据泄露，所有数据在进行分析前都已经匿名化处理，不会在中间环节出现泄露，任何与客户有关的具体信息，均不会出现在本报告中。

版权声明

THE EXPERT
BEHIND GIANTS
巨人背后的专家

多年以来，绿盟科技致力于安全攻防的研究，
为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供
具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。
在这些巨人的背后，他们是备受信赖的专家。

www.nsfocus.com