IDAPython 初学者指南

作者：Alexander Hanel

翻译：foyjog

目录

1. 介绍

大家好，这是一本关于 IDAPython 的书籍。

我写这本书的本意其实是为了给自己做参考，在我想用 IDAPython 的时候能够随时找到一些关于 IDAPython 的功能的样例。自从我写完了这本书，我经常使用它帮助我来理解 IDAPython 的语法或者寻找一些样例中的代码。如果你关注我的博客的话，你也许会注意到一些非常相似的功能脚本，这些功能脚本都是我在网上记录的大二实验的结果。

这些年来我收到了大量的关于询问学习 IDAPython 的最好办法的信件，而通常情况下我都会让他们去读 Ero Carrera's 的"Introduction to IDAPython"或者让他们去学习 IDAPython 的公开仓库中的脚本样例。这两种方法都是学习 IDAPython 的好方法，但是有一点，这两种方法中都不包括我平常在编写脚本时所遇到的问题。所以，我打算自己写一本包含我所遇到问题的书籍，我相信这本书会给想要学习 IDAPython 的人或者说想要希望快速参考一些例子和片段的人一些帮助。作为一本电子书，我会定期的更新它。

2. 目标读者和免责声明

这本书不适合逆向工程的新手阅读，也不适合当一本 IDA 的启蒙书阅读。如果你对 IDA 很陌生，那么我推荐你去买一本 Chris Eagles 的"The IDA PRO Book"来读一读，绝对的物超所值。

想要买这本书的读者需要以下几个条件：1.能够轻松的阅读汇编代码。2.有逆向工程的经验。3.熟悉 IDA 的基本操作。如果你有时会冒出一个想法：我怎么利用 IDAPython 来自动化这个工作呢？那么这本书就很适合你了。如果你已经能够自己编写一些 IDAPython 的脚本，那么这本书也不适合你。说句实在话，这是一本面向 IDAPython 新手的书籍，它是一个常用的 IDAPython 的功能脚本的参考。

还有一点要说明，我的研究方向是恶意软件的逆向工程。本书不包括静态分析中使用的基本块或其他学术概念等编译器概念，因为在我进行恶意软件的逆向分析的时候，我很少使用这些概念。有可能我已经使用了这些概念混淆了代码，但是出现的频率应该不高，我感觉这对初学者还是有帮助的。相信读完这本书以后，读者可以自己深入研究 IDAPython 文档。最后声明一点，书中将不包含 IDA 调试器的函数功能。

3. 代码习惯

书中的例子将使用 IDA 的输出窗口（命令行接口）作为输出。为了简洁一些，在一些例子中将不会把当前的地址作为一个变量来对待，通常它会表示为 ea = here()。书中所有的代码都可以被复制粘贴到 IDA 的命令行或者 IDA 的脚本窗口（快捷键为 shift+f2）中执行。当然你得好好读这本书才能行的说。我不会把所有的代码都一行行的解释的，太费时间了。一千个程序员就有一千种代码写法。在 IDAPython 中有人会写 idc.SegName(ea)，有人会写 SegName(ea)。本书的话使用第一种写法，因为我发现这种写法更好阅读和调试。但有时候这种写法会报如下的错误：

```
Python>DataRefsTo(here())
<generator object refs at 0x05247828>
Python>idautils.DataRefsTo(here())
Traceback (most recent call last):
    File "<string>", line 1, in <module>
NameError: name 'idautils' is not defined
Python>import idautils # manual importing of module
Python>idautils.DataRefsTo(here())
<generator object refs at 0x06A398C8>
```

如果报错，只要像上面那样手动引入 idautils 就可以了。

4. IDAPython 背景

IDAPython 出现于 2004 年，它是 Gergely Erdelyi 和 Ero Carrera 共同努力的成果。他们的目标在于打造一个既拥有 Python 的强大特性又拥有 IDC 的强大的自动化分析功能的脚本语言。IDAPython 由三个分离的模块组成，他们分别是 idc，idautils 和 idaapi。idc（注意大小写，不是 IDA 中的 IDC）是一个封装了 IDA 的 IDC 的兼容性模块，idautils 是 IDA 的高级实用功能模块，idaapi 允许了我们访问更加底层的数据。

5. 基本操作

在深入探索之前我们应该定义一些关键字段，顺便还要复习一下 IDA 的反汇编输出的结构，下面是个 IDA 反汇编的例子：

```
.text:00012529 mov esi, [esp+4+arg_0]
```

.text 的意思是 section 的名称，它的地址是 0x00012529，为十六进制显示。mov 是啥咱就不说了，操作数为 esi，esp+4+arg0 咱也就不讲了。在使用 IDA 的函数中最常见被传递的变量是地址。在 IDAPython 的文档中地址被记作 ea。获取当前地址有几种不同的方式，最常见的方式就是利用 idc.ScreenEA()函数或者 here()函数。它将返回一个 Interger 值。如果你想要获取当前 IDB 中的最小地址和最大地址的话就可以使用 MinEA()和 MaxEA()这两个函数。

```
Python>ea = idc.ScreenEA()
Python>print "0x%x %s" % (ea, ea)
0x12529 75049
Python>ea = here()
Python>print "0x%x %s" % (ea, ea)
0x12529 75049
Python>hex(MinEA())
0x401000
Python>hex(MaxEA())
```

```
0x437000
-----------------------------------------------
```

在 IDA 的反汇编窗口中，每一个能够被描述的元素我们都可以使用 IDAPython 来获取。下面的例子展示了如何获取这些元素。记住下面的代码要和上面的代码连起来读哦。

```
-----------------------------------------------
Python>idc.SegName(ea) # get text
.text
Python>idc.GetDisasm(ea) # get disassembly
mov esi, [esp+4+arg_0]
Python>idc.GetMnem(ea) # get mnemonic
mov
Python>idc.GetOpnd(ea,0) # get first operand
esi
Python>idc.GetOpnd(ea,1) # get second operand
[esp+4+arg_0]
-----------------------------------------------
```

上述代码中，我们利用了 idc.SegName(ea)来获取当前地址所在的段（segment）名称，还利用了 idc.GetDisasm(ea)来获取当前地址的反汇编语句，同时我们利用了 idc.GetMnem(ea)来获取当前反汇编语句的操作符（mov），利用了 idc.GetOpnd(ea,long n)来获取每个操作数（esi 和[esp+4+arg_0]）。

有些时候我们必须确认一个地址是否在当前的程序中存在，那么我们可以使用 idaapi.BADADDR 或者 BADADDR 来确认该地址。

```
-----------------------------------------------------
Python>idaapi.BADADDR
4294967295
Python>hex(idaapi.BADADDR)
0xffffffffL
Python>if BADADDR != here(): print "valid address"
valid address
-----------------------------------------------------
```

6. 段

打印一行数据好像并没什么卵用，但是 IDAPython 的强大之处在于它能遍历所有的指令，所有的交叉引用地址，还有搜索所有的代码和数据。后面两项功能稍后再做介绍。我们先从遍历所有段的指令开始讲起。

```
-----------------------------------------------------
Python>for seg in idautils.Segments():
print idc.SegName(seg), idc.SegStart(seg), idc.SegEnd(seg)
HEADER 65536 66208
.idata 66208 66636
.text 66636 212000
.data 212000 217088
.edata 217088 217184
```

```
INIT 217184 219872
.reloc 219872 225696
GAP 225696 229376
--------------------------------------------------
```

上述代码中，idautils.Segments()返回一个可迭代的对象数组，包含的元素是每个段的起始地址。通过这个起始地址我们可以获取段的名称（idc.SegName(ea)），段的起始地址（idc.SegStart(ea)），段的结束地址（idc.SegEnd(ea)）。通过 idc.NextSeg(ea)我们可以获取到当前地址所在段的下一个段的起始地址 。如果我们想通过名字得到一个段的起始地址，我们可以使用 idc.SegByName（segname）。

7. 函数

既然我们已经知道了如何遍历所有的段，现在我们也应该知道如何遍历所有的段中的所有函数：

```
------------------------------------------------------------------------
Python>for func in idautils.Functions():
print hex(func), idc.GetFunctionName(func)
Python>
0x401000 ?DefWindowProcA@CWnd@@MAEJIIJ@Z
0x401006 ?
LoadFrame@CFrameWnd@@UAEHIKPAVCWnd@@PAUCCreateContext@@@Z
0x40100c ??2@YAPAXI@Z
0x401020 save_xored
0x401030 sub_401030
....
0x45c7b9 sub_45C7B9
0x45c7c3 sub_45C7C3
0x45c7cd SEH_44A590
0x45c7e0 unknown_libname_14
0x45c7ea SEH_43EE30
------------------------------------------------------------------------
```

idautils.Functions()将会返回一个保存着已知函数首地址的数组，idautils.Functions()也可以指定要查找函数的起始地址和结束地址，就像这样子：idautils.Functions(start_addr, end_addr)。通过 idc.GetFunctionName(ea)这个函数可以通过某个地址获取函数的名称。ea 这个参数可以是处于任何函数中的地址。IDAPython 拥有大量关于函数的 API 调用。我们先来看一个简单的功能：函数表示啥意思咱暂时不需要太清楚，只要让函数的地址在脑海里有个印象即可：

```
-----------------------------------------
.text:0045C7C3 sub_45C7C3 proc near
.text:0045C7C3 mov eax, [ebp-60h]
.text:0045C7C6 push eax ; void *
.text:0045C7C7 call w_delete
.text:0045C7CC retn
.text:0045C7CC sub_45C7C3 endp
```

我们可以利用 idaapi.get_func(ea)这个函数来获取函数的边界地址（起始和结束地址）

```
Python>func = idaapi.get_func(ea)
Python>type(func)
<class 'idaapi.func_t'>
Python>print "Start: 0x%x, End: 0x%x" % (func.startEA,
func.endEA)
Start: 0x45c7c3, End: 0x45c7cd
```

---

idaapi.get_func(ea)返回一个 idaapi.func_t 的类给我们，有时候你并不清楚这个类能够干嘛，所以你可以使用 dir(class)函数来获取这个类究竟有哪些属性可以使用。

```
Python>dir(func)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__',
'__eq__', '__format__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__swig_destroy__', '__weakref__', '_print', 'analyzed_sp',
'argsize', 'clear', 'color', 'compare', 'contains', 'does_return',
'empty', 'endEA', 'extend', 'flags', 'fpd', 'frame', 'frregs',
'frsize', 'intersect', 'is_far', 'llabelqty', 'llabels',
'overlaps', 'owner', 'pntqty', 'points', 'referers', 'refqty',
'regargqty', 'regargs', 'regvarqty', 'regvars', 'size', 'startEA',
'tailqty', 'tails', 'this', 'thisown']
```

从输出我们可以看出来 startEA 和 endEA 是用来获取函数的起始地址和结束地址的。上面的这些属性只适用于当前的函数。同时我们可以使用 idc.NextFunction(ea)和 idc.PrevFunction (ea)来获取当前函数的前一个或者后一个函数。ea 的值需要在被分析的函数地址之内。在枚举函数的时候，只有 IDA 将这段代码标记为函数的时候才行，不然会在枚举的过程中被跳过。没有被标记为函数的代码将在图例（ida 顶部的彩色条）中标为红色。当然我们可以手工的修复这些无法被标记为函数的代码。

IDAPython 有非常多获取数据的方式，最常用的是通过 idc.GetFunctionAttr(ea, FUNCATTR_START) 和 idc.GetFunctionAttr(ea, FUNCATTR_END)两个调用来获取一个函数的边界地址。

```
Python>ea = here()
Python>start = idc.GetFunctionAttr(ea, FUNCATTR_START)
Python>end = idc.GetFunctionAttr(ea, FUNCATTR_END)
Python>cur_addr = start
Python>while cur_addr <= end:
print hex(cur_addr), idc.GetDisasm(cur_addr)
cur_addr = idc.NextHead(cur_addr, end)
```

```
Python>
0x45c7c3 mov eax, [ebp-60h]
0x45c7c6 push eax ; void *
0x45c7c7 call w_delete
0x45c7cc retn
```

---

idc.GetFunctionAttr(ea, attr)可以用来获取某个函数的首尾地址。同时利用idc.GetDisasm(ea)来获取当前地址的反汇编代码。然后我们利用 idc.NextHead()来不停的获取下一个指令的地址，直到函数的结束地址才停止。这种方法的一个缺陷是它依赖于指令被包含在函数开始和结束的边界内。打个比方说，函数内有个 jmp 指令，它跳转到比这个函数结束地址还要高的地址中去，意思是这个函数的所有指令可能并不是线性的，它可能会通过 jmp 跳出函数边界（起始地址和结束地址），但其实这段指令仍是属于这个函数的，那么我们使用上述的方法就不能够遍历到该函数要执行的所有指令。这种跳转在代码混淆中非常的常见，所以说我们最好还是使用 idautils.FuncItems(ea)来循环函数内的指令，我们会在下一个章节中披露更多的细节。

GetFunctionFlags(ea)是一个类似于 idc.GetFunctionAttr(ea, attr)的信息收集函数。它可以用来检索关于函数的信息，例如它是否是库中代码，或者函数是否有返回值等。对于一个函数来说有九个可能的标志，我们可以使用下面的代码枚举所有函数的所有标志：

```
Python>import idautils
Python>for func in idautils.Functions():
flags = idc.GetFunctionFlags(func)
if flags & FUNC_NORET:
print hex(func), "FUNC_NORET"
if flags & FUNC_FAR:
print hex(func), "FUNC_FAR"
if flags & FUNC_LIB:
print hex(func), "FUNC_LIB"
if flags & FUNC_STATIC:
print hex(func), "FUNC_STATIC"
if flags & FUNC_FRAME:
print hex(func), "FUNC_FRAME"
if flags & FUNC_USERFAR:
print hex(func), "FUNC_USERFAR"
if flags & FUNC_HIDDEN:
print hex(func), "FUNC_HIDDEN"
if flags & FUNC_THUNK:
print hex(func), "FUNC_THUNK"
if flags & FUNC_LIB:
print hex(func), "FUNC_BOTTOMBP"
```

我们利用了 idautils.Fucntions()来获取所有已知的函数首地址，然后利用 idc.GetFunctionFlags (ea)获取标志。代码中利用&符号来检查每个函数是否拥有某个标志，让我们看一看这些标志，有些非常常用，有些就非常少用的说，下面介绍一下每个标志的作

用：

FUNC_NORET

这个标志表示某个函数是否有返回值，它本身的值是 1，下面是一个没有返回值的函数，注意它没有函数的最后并不是 ret 或者 leave 指令

```
----------------------------------------
CODE:004028F8 sub_4028F8 proc near
CODE:004028F8
CODE:004028F8 and eax, 7Fh
CODE:004028FB mov edx, [esp+0]
CODE:004028FE jmp sub_4028AC
CODE:004028FE sub_4028F8 endp
----------------------------------------
```

FUNC_FAR

这个标志非常少的出现，标志程序是否使用分段内存，它的值为 2。

FUNC_USERFAR

这个标志也非常少见，也很少有文档，HexRays 把它描述为“user has specified far-ness of the function”，它的值是 32.

FUNC_LIB

这个表示用于寻找库函数的代码。识别库函数代码是非常有必要的，因为我们在分析的时候一般将其跳过，它的值是 4。下面的例子展示了如何使用这个标志。

```
----------------------------------------------------
Python>for func in idautils.Functions():
flags = idc.GetFunctionFlags(func)
if flags & FUNC_LIB:
print hex(func), "FUNC_LIB", GetFunctionName(func)
Python>
0x1a711160 FUNC_LIB _strcpy
0x1a711170 FUNC_LIB _strcat
0x1a711260 FUNC_LIB _memcmp
0x1a711320 FUNC_LIB _memcpy
0x1a711662 FUNC_LIB __onexit
...
0x1a711915 FUNC_LIB _exit
0x1a711926 FUNC_LIB __exit
0x1a711937 FUNC_LIB __cexit
0x1a711946 FUNC_LIB __c_exit
```

0x1a711955 FUNC_LIB _puts
0x1a7119c0 FUNC_LIB _strcmp

FUNC_STATIC

这个标志作用在于识别该函数在编译的是否是一个静态函数。在 c 语言中静态函数被默认为是全局的。如果作者把这个函数定义为静态函数，那么这个函数在只能被本文件中的函数访问。利用静态函数的判定我们可以更好的理解源代码的结构。

FUNC_FRAME

这个标志表示函数是否使用了 ebp 寄存器（帧指针），使用 ebp 寄存器的函数通常有如下的语法设定，目的是为了保存栈帧。

.text:1A716697 **push** ebp
.text:1A716698 **mov** ebp, esp
.text:1A71669A **sub** esp, 5Ch

FUNC_BOTTOMBP

和 FUNC_FRAME 一样，该标志用于跟踪帧指针（ebp）。它作用是识别函数中帧指针是否等于堆栈指针（esp）。

FUNC_HIDDEN

带有 FUNC_HIDDEN 标志的函数意味着它们是隐藏的，这个函数需要展开才能查看。如果我们跳转到一个标记为 HIDDEN 的地址的话，它会自动的展开。

FUNC_THUNK

表示这个函数是否是一个 thunk 函数，thunk 函数表示的是一个简单的跳转函数。

.text:1A710606 **Process32Next proc near**
.text:1A710606 **jmp ds**:__imp_Process32Next
.text:1A710606 **Process32Next endp**

需要注意的是一个函数可能拥有多个标志的组合。

0x1a716697 FUNC_LIB
0x1a716697 FUNC_FRAME
0x1a716697 FUNC_HIDDEN
0x1a716697 FUNC_BOTTOMBP

---

8. 指令

上一节我们已经知道如何操作函数来获取他们的指令， 如果我们拥有一个函数中的指令地址，我们可以使用 idautils.FuncItems(ea)来获取该函数中所有指令地址的集合。

```
Python>dism_addr = list(idautils.FuncItems(here()))
Python>type(dism_addr)
<type 'list'>
Python>print dism_addr
[4573123, 4573126, 4573127, 4573132]
Python>for line in dism_addr: print hex(line),
idc.GetDisasm(line)
0x45c7c3 mov eax, [ebp-60h]
0x45c7c6 push eax ; void *
0x45c7c7 call w_delete
0x45c7cc retn
```

---

idautils.FuncItems(ea)实际上返回的是一个迭代器，但是我们将它强制转换为 list 类型。这个 list 以一种连续的方式存储着所有指令的起始地址。现在我们已经完全掌握了如何循环遍历程序中的段，函数和指令，那我们就开始 show 一个非常有用的例子。有时候我们会逆向一个加壳的代码，这时知道代码中哪里进行了动态调用对分析是非常有帮助的。一个动态的调用可能是由 call 或者 jump 加上一个操作数来实现的，比如说 call eax，或者 jmp edi。

```
Python>
for func in idautils.Functions():
flags = idc.GetFunctionFlags(func)
if flags & FUNC_LIB or flags & FUNC_THUNK:
continue
dism_addr = list(idautils.FuncItems(func))
for line in dism_addr:
m = idc.GetMnem(line)
if m == 'call' or m == 'jmp':
op = idc.GetOpType(line, 0)
if op == o_reg:
print "0x%x %s" % (line, idc.GetDisasm(line))
Python>
0x43ebde call eax ; VirtualProtect
```

我们使用 idautils.Functions()获取所有的函数，然后利用 idc.GetFunctionFlags(ea)获取每个函数的标志，如果这个函数是库函数或者这个函数是 thunk 函数，那我们就跳过对它的处理， 接下来我们使用 idautils.FuncItems(ea) 获取函数中每个指定的起始地址。然后使用 idc.GetMnem(ea)来获取每条指令的操作符，判断操作符是否为 call 或者 jmp。如果是的话，

我们就使用 idc.GetOpType(ea, n)来获取操作数，这个函数返回的一个 opt.type 类型的数值（int 类型），这个数值可表示的类型为寄存器，内存引用等等。我们检查每条 call 或者 jmp 指令的操作数，如果是寄存器的话就打印出该行指令。将 idautils.FuncItems(ea)的返回类型强制转换成 list 是非常有用的，因为迭代器是没有 len()函数的。利用一个强制转换我们就能够简单的获取到函数中行数和指令内容，请看以下的代码：

```
-----------------------------------------------
Python>ea = here()
Python>len(idautils.FuncItems(ea))
Traceback (most recent call last):
File "<string>", line 1, in <module>
TypeError: object of type 'generator' has no len()
Python>len(list(idautils.FuncItems(ea)))
39
-----------------------------------------------
```

在前面的例子中我们获得了一个包含某个函数所有地址的 list。我们通过遍历 list 中的每个实体来获取下一条指令，那如果我们拥有一个确定的地址，然后想要获取这个地址的下一条指令该怎么办？我们可以使用 idc.NextHead(ea)来获取下一条指令的地址或者利用 idc.PrevHead(ea)来获取前一条指令的地址。这两个函数的功能获取的是下一条指令的地址而不是下一个地址，如果要获取下一个地址或者上一个的话，我们使用的是 idc.NextAddr(ea)和 idc.PrevAddr(ea)。

```
-----------------------------------------------
Python>ea = here()
Python>print hex(ea), idc.GetDisasm(ea)
0x10004f24 call sub_10004F32
Python>next_instr = idc.NextHead(ea)
Python>print hex(next_instr), idc.GetDisasm(next_instr)
0x10004f29 mov [esi], eax
Python>prev_instr = idc.PrevHead(ea)
Python>print hex(prev_instr), idc.GetDisasm(prev_instr)
0x10004f1e mov [esi+98h], eax
Python>print hex(idc.NextAddr(ea))
0x10004f25
Python>print hex(idc.PrevAddr(ea))
0x10004f23
-----------------------------------------------
```

9. 操作数

操作数在逆向分析中经常被使用，所以说了解所有的操作数类型对逆向分析是非常有帮助的。在前面文中提到我们可以使用 idc.GetOpType(ea,n)来获取操作数类型，ea 是一个地址，n 是一个索引。操作数总共有八种不同的类型。

o_void

如果指令没有任何操作数，它将返回 0。

```
----------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0xa09166 retn
Python>print idc.GetOpType(ea,0)
0
----------------------------------------
```

o_reg

如果操作数是寄存器，则返回这种类型，它的值为 1

```
----------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0xa09163 pop edi
Python>print idc.GetOpType(ea,0)
1
----------------------------------------
```

o_mem

如果操作数是直接寻址的内存，那么返回这种类型，它的值是 2，这种类型对寻找 DATA 的引用非常有帮助。

```
----------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0xa05d86 cmp ds:dword_A152B8, 0
Python>print idc.GetOpType(ea,0)
2
----------------------------------------
```

o_phrase

如果操作数是利用基址寄存器和变址寄存器的寻址操作的话，那么返回该类型，值为 3

```
----------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0x1000b8c2 mov [edi+ecx], eax
Python>print idc.GetOpType(ea,0)
3
----------------------------------------
```

o_displ

如果操作数是利用寄存器和位移的寻址操作的话，返回该类型，值为 4，位移指的是像

如下代码中的 0x18，这在获取结构体中的某个数据是非常常见的。

```
-----------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0xa05dc1 mov eax, [edi+18h]
Python>print idc.GetOpType(ea,1)
4
-----------------------------------------------
```

o_imm

如果操作数是一个确定的数值的话，那么返回类型，值为 5

```
----------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0xa05da1 add esp, 0Ch
Python>print idc.GetOpType(ea,1)
5
----------------------------------------
```

o_f ar

这种返回类型在 x86 和 x86_64 的逆向中不常见。它用来判断直接访问远端地址的操作数，值为 6

o_near

这种返回类型在 x86 和 x86_64 的逆向中不常见。它用来判断直接访问近端地址的操作数，值为 7

例子 1

当我们在逆向一个可执行文件的时候，我们可能会注意到一些代码会不断的重复使用某个偏移量。这种操作感觉上是代码在传递某个结构体给不同的函数使用。接下来的这个例子的目的是创建一个 python 的字典，字典包含了可执行文件中使用的所有偏移量，让偏移量作为字典的 key，而每个 key 对应的 value 存储着所有使用该偏移量的地址。在下面的代码中有一个前面没有提及过的新函数（idaapi.tag_remove(idaapi.ua_outop2(curr_addr,0))），这个函数的作用和 idc.GetOpType(ea, n)的作用是一样。(我在 ida7.0 中试了一下，发现这个函数好像被抛弃了)。

```
-------------------------------------------------------------------------
import idautils
import idaapi
displace = {}
# for each known function
for func in idautils.Functions():
    flags = idc.GetFunctionFlags(func)
```

```
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for curr_addr in dism_addr:
        op = None
        index = None
        # same as idc.GetOptype, just a different way of accessing the types
        idaapi.decode_insn(curr_addr)
        if idaapi.cmd.Op1.type == idaapi.o_displ:
            op = 1
        if idaapi.cmd.Op2.type == idaapi.o_displ:
            op = 2
        if op == None:
            continue
        if "bp" in idaapi.tag_remove(idaapi.ua_outop2(curr_addr,0)) or \
        "bp" in idaapi.tag_remove(idaapi.ua_outop2(curr_addr, 1)):
            # ebp will return a negative number
            if op == 1:
                index = (~(int(idaapi.cmd.Op1.addr) - 1) &0xFFFFFFFF)
            else:
                index = (~(int(idaapi.cmd.Op2.addr) - 1) &0xFFFFFFFF)
            else:
                if op == 1:
                    index = int(idaapi.cmd.Op1.addr)
                else:
                    index = int(idaapi.cmd.Op2.addr)
            # create key for each unique displacement value
            if index:
                if displace.has_key(index) == False:
                    displace[index] = []
                displace[index].append(curr_addr)
```

代码的最开始大家应该很熟悉了，我们利用了 idautils.Functions() 和 GetFunctionFlags(ea)来获取所有的函数，然后剔除掉其中库函数和 trunk 函数，随后我们又利用 idautils.FuncItems(ea)来获取每个函数的指令。到这里我们使用了一个新函数 idaapi.decode_insn(ea)，这个函数需要传入我们想要解读的每个指令的开始地址。一旦这个指令成功的被解读，我们就可以通过 idaapi.cmd 来获取该指令的不同属性和参数。

```
Python>dir(idaapi.cmd)
['Op1', 'Op2', 'Op3', 'Op4', 'Op5', 'Op6', 'Operands', .....,
'assign', 'auxpref', 'clink', 'clink_ptr', 'copy', 'cs', 'ea',
'flags', 'get_canon_feature', 'get_canon_mnem', 'insnpref', 'ip',
'is_canon_insn', 'is_macro', 'itype', 'segpref', 'size']
```

我们可以使用 dir()指令来获取 idaapi.cmd 的所有属性，对，就是有很多的属性。现在让我们回到我们的例子中，我们使用了 idaapi.cmd.Op1.type 来获取操作数的类型，需要注意的是这里操作数的起始索引并不是 idc.GetOpType(ea,n)中的 0，而是 1。然后我们检查两个操作数中哪个的类型是 o_displ。我们使用了 idaapi.tag_remove(idaapi.ua_outop2(ea, n))来获取操作数的字符串表示，当然你使用 idc.GetOpnd(ea, n)来完成这项操作的话是更好，更易于阅读的。我们这里只是为了凸显出在 IDAPython 中有不止一种方法来获取函数的各项参数和属性。如果我们查看一下 IDAPython 中 idc.GetOpnd(ea, n)的源代码，可以发现如下的封装：

```
def GetOpnd(ea, n):
"""
Get operand of an instruction
@param ea: linear address of instruction
@param n: number of operand:
0 - the first operand
1 - the second operand
@return: the current text representation of operand
"""
res = idaapi.ua_outop2(ea, n)
if not res:
return ""
else:
return idaapi.tag_remove(res)
```

好，现在又回到我们的例子当中，我们已经获取了操作符的字符串表示，那么我们检查操作符中是否包含了“bp”字符串，这是一个快速判断操作符的中寄存器是否为 bp，ebp 或者 rbp 的方法。检查"bp"字符串的目的在于确定偏移量是否是一个负数。我们使用 idaapi.cmd.Op1.addr 来获取偏移量，这个方法会返回一个字符串。然后我们把他转换成为一个 integer 类型，如果需要的话把它转换为正数，然后我们把它放进脚本最开始定义的字典 display 中去。这样就完成了我们的操作，之后如果你想要查找使用某个偏移量的所有地址，请使用如下所示的代码即可：

```
Python>for x in displace[0x130]: print hex(x), GetDisasm(x)
0x10004f12 mov [esi+130h], eax
0x10004f68 mov [esi+130h], eax
0x10004fda push dword ptr [esi+130h] ; hObject
0x10005260 push dword ptr [esi+130h] ; hObject
0x10005293 push dword ptr [eax+130h] ; hHandle
0x100056be push dword ptr [esi+130h] ; hEvent
0x10005ac7 push dword ptr [esi+130h] ; hEvent
```

0x130 是我们感兴趣的偏移量，你当然可以修改它来展示其他偏移量的使用情况。

例子 2

有时候我们在逆向分析一个可执行文件的内存转储的时候, 有些操作数就不是一个偏移量了。看如下代码：

```
-----------------------------------------------
seg000:00BC1388 push 0Ch
seg000:00BC138A push 0BC10B8h
seg000:00BC138F push [esp+10h+arg_0]
seg000:00BC1393 call ds:_strnicmp
-----------------------------------------------
```

第二个被 push 的值是一个存在内存中的偏移。如果我们通过右键把这个偏移定义为 data 类型，我们可以看到这个偏移其实是一个字符串，当然完成这个定义操作很简单，但是，你懂的，有时候这种操作太多了话就需要写一个脚本来自动完成这件事情的说。

```
-------------------------------------------------------
min = MinEA()
max = MaxEA()
# for each known function
for func in idautils.Functions():
    flags = idc.GetFunctionFlags(func)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for curr_addr in dism_addr:
        if idc.GetOpType(curr_addr, 0) == 5 and \
            (min < idc.GetOperandValue(curr_addr,0) < max):
            idc.OpOff(curr_addr, 0, 0)
        if idc.GetOpType(curr_addr, 1) == 5 and \
            (min < idc.GetOperandValue(curr_addr,1) < max):
            idc.OpOff(curr_addr, 1, 0)
-------------------------------------------------------
```

做完如上操作我们可以看到代码变成了如下的状态：

```
-----------------------------------------------
seg000:00BC1388 push 0Ch
seg000:00BC138A push offset aNtoskrnl_exe ;
"ntoskrnl.exe"
seg000:00BC138F push [esp+10h+arg_0]
seg000:00BC1393 call ds:_strnicmp
-----------------------------------------------
```

脚本的开始通过 MinEA()和 MaxEA()获取可执行文件的最大地址和最小地址，我们遍历了所有的函数和指令。对于每条指令我们检查他的操作数类型是否为 o_imm（值为 5），o_imm 类型的操作数就是一个确定的数值或者偏移，一旦这个发现这种类型的操作数，我们利用 idc.GetOperandValue(ea,n)来获取它的值，然后检测一下是否在可执行文件的最大地

址和最小地址中。最后我们利用 idc.OpOff(ea, n, base)来将操作数转换成为一个偏移，该函数的第一个参数为地址，第二个参数为操作数的索引，第三个参数为基地址，在该例子中我们只需要将其设为 0 即可。

10. 交叉引用（Xrefs）

能够定位 data 段和 code 段的交叉引用是非常重要的，交叉引用的重要性在于它能够提供某个确定的数据或者某个函数被调用的位置。举个栗子，如果我们想要知道哪些地址调用了 WriteFile()函数，我们所要做的就是在导入表中找到 Write File()函数，然后查看其交叉引用即可。

```
-------------------------------------------------------------
Python>wf_addr = idc.LocByName("WriteFile")
Python>print hex(wf_addr), idc.GetDisasm(wf_addr)
0x1000e1b8 extrn WriteFile:dword
Python>for addr in idautils.CodeRefsTo(wf_addr, 0):\
print hex(addr), idc.GetDisasm(addr)
0x10004932 call ds:WriteFile
0x10005c38 call ds:WriteFile
0x10007458 call ds:WriteFile
-------------------------------------------------------------
```

第一行中我们使用了 idc.LocByName(str)来获取 API（WriteFile()）的地址，该函数返回 API 的地址，我们将其打印出来。然后我们利用 idautils.CodeRefsTo(ea, flow)该函数循环打印出该 API 的所有交叉引用。在函数中，ea 参数是我们想要寻找交叉引用的地址，flow 参数是一个 bool 值，它用于指定是否遵循正常的代码流。上述便是 idc.LocByName(str)的一个简要说明。我们可以通过调用 idautils.Names()函数来获取在 IDA 中任何 API 和被重命名的函数的相关信息，该函数将返回一个类型为(ea, str_name)的元组。

```
-----------------------------------------------------------------
Python>[x for x in Names()]
[(268439552, 'SetEventCreateThread'), (268439615, 'StartAddress'),
(268441102, 'SetSleepClose'),....
-----------------------------------------------------------------
```

我们同样可以利用 idautisl.CodeRefsFrom(ea,flow)该函数来获取任意地址所引用的代码，下面的例子展示了如果获取 0x10004932 该地址的引用信息。

```
-------------------------------------------------------------
Python>ea = 0x10004932
Python>print hex(ea), idc.GetDisasm(ea)
0x10004932 call ds:WriteFile
Python>for addr in idautils.CodeRefsFrom(ea, 0):\
print hex(addr), idc.GetDisasm(addr)
Python>
0x1000e1b8 extrn WriteFile:dword
-------------------------------------------------------------
```

如果我们观察上面使用 idautils.CodeRefsTo(ea, flow)的例子我们可以看到 0x10004932 是一个调用了 WriteFile 函数的地址。idautils.CodeRefsTo(ea, flow) 和

idautils.CodeRefsFrom(ea, flow)两个函数具有相反的作用，一个是查找某处地址在哪儿被调用，一个是查找某处地址调用了哪些地址（具体情况大家可以操作一下便知）。但有一点要注意：使用 idautils.CodeRefsTo（ea，flow）的限制是，动态导入并手动重命名的 API 不会显示为代码交叉引用。比如下面我们利用 idc.MakeName(ea, name)将一个 dword 的地址重命名为"RtlCompareMemory"。

```
Python>hex(ea)
0xa26c78
Python>idc.MakeName(ea, "RtlCompareMemory")
True
Python>for addr in idautils.CodeRefsTo(ea, 0):\
print hex(addr), idc.GetDisasm(addr)
```

IDA 并不会将这些 API 标记为交叉引用代码。稍后我们将会使用一个通用的技术来获得所有的交叉引用。

如果我们想要查找数据的交叉引用或者调用，我们使用 idautils.DataRefsTo(e) or idautils.DataRefsFrom(ea) 。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x1000e3ec db 'vnc32',0
Python>for addr in idautils.DataRefsTo(ea): print hex(addr),
idc.GetDisasm(addr)
0x100038ac push offset aVnc32 ; "vnc32"
```

---

idautils.DataRefsTo(ea)函数只有一个地址参数，它返回该数据地址的所有交叉引用（迭代器）。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x100038ac push offset aVnc32 ; "vnc32"
Python>for addr in idautils.DataRefsFrom(ea): print hex(addr),
idc.GetDisasm(addr)
0x1000e3ec db 'vnc32',0
```

---

idautils.DataRefsFrom(ea)只携带一个地址作为参数，它返回改、该地址所引用的数据地址。在查找数据和代码的交叉引用的时候可能会有一些困惑，这里我们使用前面所提到的有一种更加通用的方法来获取交叉引用，该方法调用两个函数就能完成获取所有交叉引用地址和调用地址的效果，这两个函数就是 idautils.XrefsTo(ea, flags=0)和 idautils.XrefsFrom(ea, flags=0)。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x1000eee0 unicode 0, <Path>,0
Python>for xref in idautils.XrefsTo(ea, 1):
print xref.type, idautils.XrefTypeName(xref.type), \
hex(xref.frm), hex(xref.to), xref.iscode
```

```
Python>
1 Data_Offset 0x1000ac0d 0x1000eee0 0
Python>print hex(xref.frm), idc.GetDisasm(xref.frm)
0x1000ac0d push offset KeyName ; "Path"
```

第一行显示了一个<Path>字符串的的地址，我们使用 idautils.XrefsTo(ea, 1)来获取该字符串的所有交叉引用。我们使用 xref.type 来指明该交叉引用的类型，idautils.XrefTypeName(xref.t ype)用来打印表示该类型的含义，这其中有十二种不同的类型。

```
0 = 'Data_Unknown'
1 = 'Data_Offset'
2 = 'Data_Write'
3 = 'Data_Read'
4 = 'Data_Text'
5 = 'Data_Informational'
16 = 'Code_Far_Call'
17 = 'Code_Near_Call'
18 = 'Code_Far_Jump'
19 = 'Code_Near_Jump'
20 = 'Code_User'
21 = 'Ordinary_Flow'
```

---

xref.frm 打印出该地址的交叉引用，xref.to 打印出改地址本身，xref.iscode 打印出该交叉引用是否在代码段中，上述的代码我们使用了 idautils.XrefsTo(ea, 1)并将其 flag 位设为了 1，如果我们将 flag 设为 0 的话，那么它将会显示该地址的任意交叉引用，举个栗子：

```
.text:1000AAF6 jnb    short loc_1000AB02 ; XREF
.text:1000AAF8 mov eax, [ebx+0Ch]
.text:1000AAFB mov ecx, [esi]
.text:1000AAFD sub eax, edi
.text:1000AAFF mov [edi+ecx], eax
.text:1000AB02
.text:1000AB02 loc_1000AB02: ; ea is  here()
.text:1000AB02 mov byte ptr [ebx],
```

现在我们的光标停留在 1000AB02 上，该地址有来自 1000AAF6 的交叉引用，但是其实它还有着另一个交叉引用。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x1000ab02 mov byte ptr [ebx], 1
Python>for xref in idautils.XrefsTo(ea, 1):
print xref.type, idautils.XrefTypeName(xref.type), \
hex(xref.frm), hex(xref.to), xref.iscode
Python>
```

```
19 Code_Near_Jump 0x1000aaf6 0x1000ab02 1
Python>for xref in idautils.XrefsTo(ea, 0):
print xref.type, idautils.XrefTypeName(xref.type), \
hex(xref.frm), hex(xref.to), xref.iscode
Python>
21 Ordinary_Flow 0x1000aaff 0x1000ab02 1
19 Code_Near_Jump 0x1000aaf6 0x1000ab02 1
```

上述地址的第二个交叉引用是从 1000AAFF 到 1000AB02 .对，你没有看错，交叉引用不只是来自于分支跳转指令，同时还会来自于正常的指令流程，所以我们将 flag 设为 1 来略过正常指令流程造成的交叉引用。现在让我们回到之前 RtlCompareMemory 的栗子，现在我们可以使用 idautils.XrefsTo(ea,flow)来获取它所有的交叉引用了。

```
Python>hex(ea)
0xa26c78
Python>idc.MakeName(ea, "RtlCompareMemory")
True
Python>for xref in idautils.XrefsTo(ea, 1):
print xref.type, idautils.XrefTypeName(xref.type), \
hex(xref.frm), hex(xref.to), xref.iscode
Python>
3 Data_Read 0xa142a3 0xa26c78 0
3 Data_Read 0xa143e8 0xa26c78 0
3 Data_Read 0xa162da 0xa26c78 0
```

当然打印所有的交叉引用可能会感觉有点儿重复的说。

```
Python>print hex(ea), idc.GetDisasm(ea)
0xa21138 extrn GetProcessHeap:dword
Python>for xref in idautils.XrefsTo(ea, 1):
print xref.type, idautils.XrefTypeName(xref.type), \
hex(xref.frm), hex(xref.to), xref.iscode
Python>
17 Code_Near_Call 0xa143b0 0xa21138 1
17 Code_Near_Call 0xa1bb1b 0xa21138 1
3 Data_Read 0xa143b0 0xa21138 0
3 Data_Read 0xa1bb1b 0xa21138 0
Python>print idc.GetDisasm(0xa143b0)
call ds:GetProcessHeap
```

重复性来自于 Data_Read 和 Code_Near 类型的交叉引用都被识别出来，所以下面利用 set 的方法可以 精简一下这些地址：

```
def get_to_xrefs(ea):
```

```
xref_set = set([])
for xref in idautils.XrefsTo(ea, 1):
xref_set.add(xref.frm)
return xref_set
def get_frm_xrefs(ea):
xref_set = set([])
for xref in idautils.XrefsFrom(ea, 1):
xref_set.add(xref.to)
return xref_set
```

以下是精简过后的栗子：

```
Python>print hex(ea), idc.GetDisasm(ea)
0xa21138 extrn GetProcessHeap:dword
Python>get_to_xrefs(ea)
set([10568624, 10599195])
Python>[hex(x) for x in get_to_xrefs(ea)]
['0xa143b0', '0xa1bb1b']
```

---

11. 搜索

我们其实已经能够通过遍历所有已知的函数及其指令来达到一种基本的搜索效果，这当然很有用，但是有时候我们需要搜索一些特定的字节，比如说 0x55,0x8b,0xec 这种字节序列，这 3 个字节其实代表的汇编代码为 push ebp, mov ebp, esp 。所以我们可以使用 idc.FindBinary(ea,flag, searchstr, radix=16)来实行字节或者二进制的搜索。ea 代表啥就不说了，flag 代表搜索方向或者条件。flag 有好几种不同的类型，喏，都在下面了：

```
SEARCH_UP = 0
SEARCH_DOWN = 1
SEARCH_NEXT = 2
SEARCH_CASE = 4
SEARCH_REGEX = 8
SEARCH_NOBRK = 16
SEARCH_NOSHOW = 32
SEARCH_UNICODE = 64 **
SEARCH_IDENT = 128 **
SEARCH_BRK = 256 **
```

** 表示旧版本的 idapython 不支持这些东西

上面的类型不必要都看一遍，但是还是要看看一些常用的类型：

- SEARCH_UP 和 SEARCH_DOWN 用来指明搜索的方向
- SEARCH_NEXT 用来获取下一个已经找到的对象
- SEARCH_CASE 用来指明是否区分大小写

- SEARCH_NOSHOW 用来指明是否显示搜索的进度
- SEARCH_UNICODE 用于将所有搜索字符串视为 Unicode

searchstr 是我们要查找的形态，radix 参数在写处理器模块时使用，这超出本书要讲解的范围，所以我推荐你去看一看 Chris Eagle 的“The IDA Pro Book”的第 19 章，所以这里我们把 radix 参数留空。现在让我们来实现刚才提到的那三个字节的搜索好了：

```
----------------------------------------------------------
Python>pattern = '55 8B EC'
addr = MinEA()
for x in range(0,5):
    addr = idc.FindBinary(addr, SEARCH_DOWN, pattern);
if addr != idc.BADADDR:
    print hex(addr), idc.GetDisasm(addr)
Python>
0x401000 push ebp
0x401000 push ebp
0x401000 push ebp
0x401000 push ebp
0x401000 push ebp
----------------------------------------------------------
```

第一行我们定义了要搜索的形式，搜索形式可以是 16 进制格式，比如 0x55 0x8B 0xEC 和 55 8B EC 都是可以的，\x55\x8B\xEC 这种格式可不行，除非你使用 idc.FindText(ea, flag, y, x, searchstr)这个函数。MinEA()用来获取可执行文件的最开始的地址，接下来我们将 idc.FindBinary(ea, flag,searchstr, radix=16)的返回结果设置为 addr 变量。

在搜索时最重要的是验证结果是否符合预期，首先我们将地址与 idc.BADADDR 进行比较，然后打印出该地址和该地址的反汇编结果。注意到为什么地址没有增长没，那是因为我们在写程序的时候遗漏了 SEARCH_NEXT 这个标记，哈哈，下面才是正确的写法：

```
----------------------------------------------------------------
Python>pattern = '55 8B EC'
addr = MinEA()
for x in range(0,5):
    addr = idc.FindBinary(addr, SEARCH_DOWN|SEARCH_NEXT,
pattern);
if addr != idc.BADADDR:
    print hex(addr), idc.GetDisasm(addr)
Python>
0x401040 push ebp
0x401070 push ebp
0x4010e0 push ebp
0x401150 push ebp
0x4011b0 push ebp
----------------------------------------------------------------
```

搜索字节序列是很有用的，但是有时候我们想要查找类似于“chrome.dll”这样的字符串。我们可以利用 hex(y)函数来将“chrome.dll”字符串转换成为 16 进制的字节序列，但是，太蠢了，如果字符串是 unicode 格式呢，你还得重新考虑了，好了，不瞒各位，最好的方法是

FindText(ea,flag, y, x, searchstr)。这个函数看起来是不是很熟悉，因为它很像函数idc.FindBinary 的说，参数里面，ea 是地址，flag 是搜索方向和搜索类型。y 是从 ea 开始搜索的行数，x 是行中的坐标。这两个参数通常置 0，现在我们开始查找字符串“Accept”的出现的次数。当然你可以换换其他的字符串，可以从字符串窗口（shift+F12）获得。

```
Python>cur_addr = MinEA()
end = MaxEA()
while cur_addr < end:
    cur_addr = idc.FindText(cur_addr, SEARCH_DOWN, 0, 0,
"Accept")
if cur_addr == idc.BADADDR:
    break
else:
    print hex(cur_addr), idc.GetDisasm(cur_addr)
    cur_addr = idc.NextHead(cur_addr)
Python>
0x40da72 push offset aAcceptEncoding; "Accept-Encoding:\n"
0x40face push offset aHttp1_1Accept; " HTTP/1.1\r\nAccept: */*\r\n "
0x40fadf push offset aAcceptLanguage; "Accept-Language: ru\r\n"
...
0x423c00 db 'Accept',0
0x423c14 db 'Accept-Language',0
0x423c24 db 'Accept-Encoding',0
0x423ca4 db 'Accept-Ranges',0
```

我们利用 MinEA()来获取可执行文件的最小地址， 将其赋给 cur_addr。利用 MaxEA()赋值给 end。因为不知道该字符串出现的次数，所以我们从上往下搜索直到最大地址。我们把 idc.FindText 的返回值赋给 cur_addr。因为我们利用了 idc.NextHead(ea)使当前地址不断增长，所以就不需要在 idc. FindText()中添加 SEARCH_NEXT 的标志。为什么我们要手动的增加地址呢，因为一行字符串中可能出现多次要查找的字符串，往上翻认真阅读 SEARCH_NEXT 的标志的意思，你就发现不手动的话会陷入一个死循环（但是测试的时候，算了你们自己测吧，估计我测的有问题）。

除了以上述描述的搜索方式以为，还有一系列的函数用来查找其他类型。通过这写 API 的名称就能容易的识别出它们的功能，在讨论查找其他类型之前我们首先来看一下如何通过地址来识别他们的类型，IDAPython 中有如下的一些函数用来判断一个地址的类型，这些 APIs 返回的是 bool 值 True 或者 False。

idc.isCode(f)

判断 IDA 是否将其判定为代码。

idc.isData(f)

判断 IDA 是否将其判定为数据。

idc.isTail(f)

判断 IDA 是否将其判定为尾部。

idc.isUnknown(f)

判断 IDA 是否将其判定为未知，即既不是数据，也不是代码。

idc.isHead(f)

判断 IDA 是否将其判定为头部。
f 这个参数是新出现的，相比起于传递地址，我们还要先通过 idc.GetFlags(ea)获取地址的内部标志表示，然后再传给 idc.is 系列函数当参数，代码如下：

```
------------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0x10001000 push ebp
Python>idc.isCode(idc.GetFlags(ea))
True
------------------------------------------------
```

idc.FindCode(ea, flag)

该函数用来寻找被标志为代码的下一个地址。这对我们想要查找数据块的末尾是很有帮助的。如果 ea 是代码地址，那么该函数返回下一个代码地址，flag 参数看前面的 idc.FindText 就可以了。

```
------------------------------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0x4140e8 dd offset dword_4140EC
Python>addr = idc.FindCode(ea, SEARCH_DOWN|SEARCH_NEXT)
Python>print hex(addr), idc.GetDisasm(addr)
0x41410c push ebx
------------------------------------------------------------------
```

可以看到 0x4140e8 是某个数据的地址。我们将 idc.FindCode(ea, SEARCH_DOWN|SEARCH _NEXT)的返回值赋给 addr，然后打印出该地址和它的反汇编代码。通过该函数我们直接跳过了 36 个字节获取到了代码段的首地址。

idc.FindData(ea, flag)

该函数和上一个函数 FindCode()差不多，除了它返回的是数据段的地址。我们反转一下上一个场景，然后通过代码段地址去寻找数据段首地址，代码如下：

```
--------------------------------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0x41410c push ebx
Python>addr = idc.FindData(ea, SEARCH_UP|SEARCH_NEXT)
Python>print hex(addr), idc.GetDisasm(addr)
0x4140ec dd 49540E0Eh, 746E6564h, 4570614Dh, 7972746Eh, 8, 1,4010BCh
--------------------------------------------------------------------
```

idc.FindUnexplored(ea, flag)

该功能用于查找 IDA 未识别为代码或数据的字节地址。未知类型需要通过观察或脚本进一步手动分析。

```
--------------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0x406a05 jge short loc_406A3A
Python>addr = idc.FindUnexplored(ea, SEARCH_DOWN)
Python>print hex(addr), idc.GetDisasm(addr)
0x41b004 db 0DFh ; ?
--------------------------------------------------
```

idc.FindExplored(ea, flag)

它用于查找 IDA 标识为代码或数据的地址。

```
--------------------------------------------------
0x41b900 db ? ;
Python>addr = idc.FindExplored(ea, SEARCH_UP)
Python>print hex(addr), idc.GetDisasm(addr)
0x41b5f4 dd ?
--------------------------------------------------
```

好像并没什么卵用，但是如果我们打印 addr 的交叉引用，就可以发现写其他的东西：

```
------------------------------------------
Python>for xref in idautils.XrefsTo(addr, 1):
print hex(xref.frm), idc.GetDisasm(xref.frm)
Python>
0x4069c3 mov eax, dword_41B5F4[ecx*4]
------------------------------------------
```

idc.FindImmediate(ea, flag, value)

相比于寻找一些类型，我们有些时候其实更希望能够找到特定的值，举个栗子，你感觉代码里面肯定是用了 rand()函数来产生随机数的，但是你就是找不到它，咋办？如果我们直到这个 rand()函数采用了 0x343fd 作为种子那么我们就可以去寻找这个数值：

```
------------------------------------------------------------------------
Python>addr = idc.FindImmediate(MinEA(), SEARCH_DOWN, 0x343FD )
Python>addr
[268453092, 0]
Python>print "0x%x %s %x" % (addr[0], idc.GetDisasm(addr[0]),
addr[1] )
0x100044e4 imul eax, 343FDh 0
------------------------------------------------------------------------
```

第一行我们通过 MinEA()传递最小地址作为参数，SEARCH_DOWN 为 flag，0x343fd 作为查找的值。不像前几个函数返回地址，idc.FindImmediate 返回一个元组，元组的第一项为

地址，第二项为标识是第几个操作数。这里操作数的算法和 idc.GetOpnd 一样，也是从 0 开始的。我们打印出改地址和反汇编代码可以看出，0x343FD 是作为第二个操作数。如果我们想查找所有的立即数的使用的话可以利用如下的代码：（感觉上面的代码有问题+ +，那个 addr[1]的值应该是 1）

```
--------------------------------------------------------------------------------
Python>addr = MinEA()
while True:
    addr, operand = idc.FindImmediate(addr,SEARCH_DOWN|SEARCH_NEXT, 0x7a )
    if addr != BADADDR:
        print hex(addr), idc.GetDisasm(addr), "Operand ", operand
    else:
        break
Python>
0x402434 dd 9, 0FF0Bh, 0Ch, 0FF0Dh, 0Dh, 0FF13h, 13h, 0FF1Bh, 1Bh Operand 0
0x40acee cmp eax, 7Ah Operand 1
0x40b943 push 7Ah Operand 0
0x424a91 cmp eax, 7Ah Operand 1
0x424b3d cmp eax, 7Ah Operand 1
0x425507 cmp eax, 7Ah Operand 1
--------------------------------------------------------------------------------
```

上述大部分代码看起来很熟悉，但是因为搜索不止一个值，循环的时候需要使用 SEARCH_ DOWN|SEARCH_NEXT 标志。

12. 数据选取

有时候我们并不是只想写自动查找数据和代码的脚本，也许你已经知道了代码或者数据的地址，你现在想选取它们分析一下。可能我们只是想要把突出某些代码然后在这些代码上面利用 IDAPython 进行工作。为了获得选定数据的边界，我们可以使用 idc.SelStart()来获取开始地址和 idc.SelEnd()来获得结束地址，举个栗子，现在我们选取了以下的代码:

```
-----------------------------------------------
.text:00408E46 push ebp
.text:00408E47 mov ebp, esp
.text:00408E49 mov al, byte ptr dword_42A508
.text:00408E4E sub esp, 78h
.text:00408E51 test al, 10h
.text:00408E53 jz short loc_408E78
.text:00408E55 lea eax, [ebp+Data]
-----------------------------------------------
```

我们使用如下代码来打印地址：

```
-----------------------------------
Python>start = idc.SelStart()
Python>hex(start)
0x408e46
Python>end = idc.SelEnd()
```

```
Python>hex(end)
0x408e58
----------------------------------
```

我们利用 idc.SelStart()赋值给 start，idc.SelEnd()赋值给 end，请注意 end 的地址，它是下一条指令的起始地址，并不是这段代码最后一条指令的起始地址。如果你想要只用一个函数就完成上面的操作，那么你可以使用 idaapi.read_selection()，它返回一个元组，第一个值为 bool 值，判断是否读取成功，第二个值为开始地址，最后一个值为结束地址。

```
-----------------------------------------------------
Python>Worked, start, end = idaapi.read_selection()
-----------------------------------------------------
```

注意在分析 64 位可执行文件的时候容易出错，因为 64 位地址容易导致整数 integer 的溢出。

13. 注释和重命名

个人信仰的是如果你不在写写画画那你就不在做逆向。添加添加注释，重命名一些函数等等，和文件做交互是理解代码的重要途径。有时候一些交互的操作比较麻烦的说，所以你可以自动化它，当然，我们得利用 IDAPython。

在开始举栗子之前我们得讨论一下注释的基本操作。总共有两种注释，第一种常规注释，第二种重复性注释。0041136B 中包含的是常规注释，很简单就是文本，00144372，00411386，00411392 包含了重复性注释。只有最后一个注释是我们手工输入的，其他的注释因为某个指令引用了输入注释的地址而自动产生的（比如条件分支）。

```
--------------------------------------------------------------------
00411365 mov [ebp+var_214], eax
0041136B cmp [ebp+var_214], 0 ; regular comment
00411372 jnz short loc_411392 ; repeatable comment
00411374 push offset sub_4110E0
00411379 call sub_40D060
0041137E add esp, 4
00411381 movzx edx, al
00411384 test edx, edx
00411386 jz short loc_411392 ; repeatable comment
00411388 mov dword_436B80, 1
00411392
00411392 loc_411392:
00411392
00411392 mov dword_436B88, 1 ; repeatable comment
0041139C push offset sub_4112C0
--------------------------------------------------------------------
```

我们使用 idc.MakeComm(ea,comment)来增加注释，idc.MakeRptCmt(ea, comment)来增加重复性注释，ea 是要添加注释的地址。下面代码在一条指令（用 XOR 清零寄存器）出现的时候自动增加注释。

```
------------------------------------------------------------
for func in idautils.Functions():
```

```
    flags = idc.GetFunctionFlags(func)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for ea in dism_addr:
        if idc.GetMnem(ea) == "xor":
            if idc.GetOpnd(ea, 0) == idc.GetOpnd(ea, 1):
                comment = "%s = 0" % (idc.GetOpnd(ea,0))
                idc.MakeComm(ea, comment)
```

如前所述，我们通过调用 idautils.Functions()来循环遍历所有函数，并通过调用 list（idautils. FuncItems(func)）遍历所有指令。然后利用 idc.GetMnem(ea)获取操作符，并判断是否为 XOR。然后利用 idc.GetOpnd(ea, n)判断两个操作数是否相等，如果是的话，那么给它加上常规注释。效果如下：

```
0040B0F7 xor al, al ; al = 0
0040B0F9 jmp short loc_40B163
```

如果要增加重复性注释，那么我们使用 MakeRptCmt(ea, comment)去代替 idc.MakeComm(ea, comment)。 这样做好像更有用一点，因为我们可以看到一些分支做了清零，就像是返回 0 的操作。要获取一个注释我们可以使用 GetCommentEx(ea, repeatable)该函数，ea 是地址，repeatable 是 bool 值。要获取上述的注释，我们可以使用如下的代码：

```
Python>print hex(ea), idc.GetDisasm(ea)
0x40b0f7 xor al, al ; al = 0
Python>idc.GetCommentEx(ea, False)
al = 0
```

如果要获取重复性注释使用 idc.GetCommentEx(ea, True)就可以了。当然不只是指令可以做注释，函数也可以做注释的说，我们利用 idc.SetFunctionCmt(ea, cmt, repeatable)注释函数和利用 idc.GetFunctionCmt(ea, repeatable)获取函数的注释。ea 可以是函数中的任何地址，cmt 是我们要添加的注释，repeatable 同上面一样。将函数的注释标记为可重复性的话，那么它会在任何调用该函数的地方增加注释。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x401040 push ebp
Python>idc.GetFunctionName(ea)
sub_401040
Python>idc.SetFunctionCmt(ea, "check out later", 1)
True
```

第一行我们打印该函数的地址和反汇编，第二行我们打印函数的名字，然后我们利用 idc.SetFunctionCmt(ea, comment, repeatable)来设置一个可重复性注释（check out later）给

该函数。如果我们去观察函数的起始地址就可以发现我们增加的注释。

```
-------------------------------------------------------
00401040 ; check out later
00401040 ; Attributes: bp-based frame
00401040
00401040 sub_401040 proc near
00401040 .
00401040 var_4 = dword ptr -4
00401040 arg_0 = dword ptr 8
00401040
00401040 push ebp
00401041 mov ebp, esp
-------------------------------------------------------
```

因为函数的注释是可重复性的，所以函数的交叉引用地方我们同样能看到注释。这样的话就能够很好的提示我们函数的功能。

```
-----------------------------------------------
00401C07 push ecx
00401C08 call sub_401040 ; check out later
00401C0D add esp, 4
-----------------------------------------------
```

重命名函数和地址是一项非常常见的自动化任务，特别是在一些于地址无关代码（PIC），加壳或者封装函数中，因为在 PIC 代码和脱壳代码中，导入表可能并不存在于转储中。而封装函数的功能只是简单的调用 API 而已。

```
-----------------------------------------
10005B3E sub_10005B3E proc near
10005B3E
10005B3E dwBytes = dword ptr 8
10005B3E
10005B3E push ebp
10005B3F mov ebp, esp
10005B41 push [ebp+dwBytes] ; dwBytes
10005B44 push 8 ; dwFlags
10005B46 push hHeap ; hHeap
10005B4C call ds:HeapAlloc
10005B52 pop ebp
10005B53 retn
10005B53 sub_10005B3E endp
-----------------------------------------
```

上述的代码其实可以被成为 w_HeapAlloc，w 的意思是封装。重命名该地址的话可以使用 idc.MakeName(ea, name)，ea 是地址，name 是重命名的名称（w_HeapAlloc）。重命名一个函数的话 ea 一定要是函数的起始地址，我们用以下的代码来重命名上面的那个函数：

```
-------------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0x10005b3e push ebp
```

```
Python>idc.MakeName(ea, "w_HeapAlloc")
True
```

---

ea 是函数的起始地址，name 是 w_HeapAlloc。

```
10005B3E w_HeapAlloc proc near
10005B3E
10005B3E dwBytes = dword ptr 8
10005B3E
10005B3E push ebp
10005B3F mov ebp, esp
10005B41 push [ebp+dwBytes] ; dwBytes
10005B44 push 8 ; dwFlags
10005B46 push hHeap; hHeap
10005B4C call ds:HeapAlloc
10005B52 pop ebp
10005B53 retn
10005B53 w_HeapAlloc endp
```

上面可以看到函数已经被重命名了，验证一下，利用 idc.GetFunctionName(ea)来打印该函数的名称。

```
Python>idc.GetFunctionName(ea)
w_HeapAlloc
```

要重命名一个操作数，我们需要拿到它的地址先，比如在 004047B0 有一个双字我们想要重命名：

```
.text:004047AD lea ecx, [ecx+0]
.text:004047B0 mov eax, dword_41400C
.text:004047B6 mov ecx, [edi+4BCh]
```

要获取操作数的数值我们可以使用 GetOperandValue(ea, n) .

```
Python>print hex(ea), idc.GetDisasm(ea)
0x4047b0 mov eax, dword_41400C
Python>op = idc.GetOperandValue(ea,1)
Python>print hex(op), idc.GetDisasm(op)
0x41400c dd 2
Python>idc.MakeName(op, "BETA")
True
Python>print hex(ea), idc.GetDisasm(ea)
0x4047b0 mov eax, BETA[esi]
```

第一行我们打印了当前地址信息，然后我们把第二个操作数 dword_41400C 赋值给 op，然后使用 idc.MakeName(ea, name)进行重命名，最后打印出新的被重命名的名字。
好了，既然我们现在基础已经很好了，我们现在就用我们所学的来自动化命名封装函数。请注意注释的意思来保证你能够理解代码逻辑。

```
import idautils
def rename_wrapper(name, func_addr):
    if idc.MakeNameEx(func_addr, name, SN_NOWARN):
        print "Function at 0x%x renamed %s" %( func_addr,idc.GetFunctionName(func))
    else:
        print "Rename at 0x%x failed. Function %s is being used."% (func_addr, name)
    return
def check_for_wrapper(func):
    flags = idc.GetFunctionFlags(func)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        return
    dism_addr = list(idautils.FuncItems(func))
    # get length of the function
    func_length = len(dism_addr)
    # if over 32 lines of instruction return
    if func_length > 0x20:
        return
    func_call = 0
    instr_cmp = 0
    op = None
    op_addr = None
    op_type = None
    # for each instruction in the function
    for ea in dism_addr:
        m = idc.GetMnem(ea)
        if m == 'call' or m == 'jmp':
            if m == 'jmp':
                temp = idc.GetOperandValue(ea,0)
                # ignore jump conditions within the function boundaries
                if temp in dism_addr:
                    continue
            func_call += 1
            # wrappers should not contain multiple function calls
            if func_call == 2:
                return
            op_addr = idc.GetOperandValue(ea , 0)
            op_type = idc.GetOpType(ea,0)
        elif m == 'cmp' or m == 'test':
```

```
            # wrappers functions should not contain much logic.
                instr_cmp += 1
                if instr_cmp == 3:
                    return
            else:
                continue
        # all instructions in the function have been analyzed
        if op_addr == None:
            return
        name = idc.Name(op_addr)
        # skip mangled function names
        if "[" in name or "$" in name or "?" in name or "@" in name or name == "":
            return
        name = "w_" + name
        if op_type == 7:
            if idc.GetFunctionFlags(op_addr) & FUNC_THUNK:
                rename_wrapper(name, func)
                return
        if op_type == 2 or op_type == 6:
            rename_wrapper(name, func)
            return
for func in idautils.Functions():
    check_for_wrapper(func)
```

输出样例：

```
Function at 0xa14040 renamed w_HeapFree
Function at 0xa14060 renamed w_HeapAlloc
Function at 0xa14300 renamed w_HeapReAlloc
Rename at 0xa14330 failed. Function w_HeapAlloc is being used.
Rename at 0xa14360 failed. Function w_HeapFree is being used.
Function at 0xa1b040 renamed w_RtlZeroMemory
```

所有的代码都很熟悉，但是值得注意的是 rename_wrapper 中的 idc.MakeNameEx(ea, name, flag)用法，因为使用 idc.MakeName 的话，如果某一个函数名称已经被使用了，那么 ida 会抛出一个警告的对话。为了跳过该对话框，我们将 flag 的值设置为 256 或者 SN_NOWARN 即可。我们可以应用一些逻辑来将函数重命名为 w_HeapFree_1 等，但为简洁起见，我们会将其忽略。

14. 访问原始数据

在逆向工程中获取原始数据是非常重要的。原始数据是 16 进制的字节，它们被解释为数据或者代码，ida 中我们可以在反汇编代码窗口的左侧看到这些原始数据（需要在 IDA 里面进行设置的说）。

```
--------------------------------------------------
00A14380 8B 0D 0C 6D A2 00    mov ecx, hHeap
00A14386 50                   push eax
00A14387 6A 08                push 8
00A14389 51                   push ecx
00A1438A FF 15 30 11 A2 00    call ds:HeapAlloc
00A14390 C3                   retn
--------------------------------------------------
```

要获取原始数据的话我们首先要指定获取的单元大小，这些获取原始数据 API 的命名规则就是以单元大小。我们使用 idc.Byte(ea)获取单字节，idc.Word(ea)获取字等等。

```
----------------------
```

- idc.Byte(ea)
- idc.Word(ea)
- idc.Dword(ea)
- idc.Qword(ea)
- idc.GetFloat(ea)
- idc.GetDouble(ea)

```
----------------------
```

如果当前的地址是 00A14380 的话，我们可以得出以下输出：

```
------------------------------------------------------
Python>print hex(ea), idc.GetDisasm(ea)
0xa14380 mov ecx, hHeap
Python>hex( idc.Byte(ea) )
0x8b
Python>hex( idc.Word(ea) )
0xd8b
Python>hex( idc.Dword(ea) )
0x6d0c0d8b
Python>hex( idc.Qword(ea) )
0x6a5000a26d0c0d8bL
Python>idc.GetFloat(ea) # Example not a float value
2.70901711372e+27
Python>idc.GetDouble(ea)
1.25430839165e+204
------------------------------------------------------
```

在编写解码脚本是获取单个字节或者单个字并没有太多卵用，所以我们可以使用 idc.GetManyBytes(ea, size, use_dbg=False)来获取某个地址开始的更多的字节。最后一个参数是可选的，用来指定是否正在调试内存。

```
-------------------------------------------
ython>for byte in idc.GetManyBytes(ea, 6):
print "0x%X" % ord(byte),
0x8B 0xD 0xC 0x6D 0xA2 0x0
-------------------------------------------
```

要注意的是 idc.GetManyBytes(ea, size)返回的是字节的字符表示，这个和 idc.Word(ea)

或者 idc.Qword(ea)返回的 integer 不一样。

15. 补丁

有时候我们在逆向一个恶意软件的时候，样本会有被加密的字符串。这会阻碍我们分析的过程和组织我们通过字符串来定位关键点。这种情况下给 idb 文件打补丁就很有用了。重命名地址但是好像并没有卵用，因为命名是有约束限制的，所以我们需要给某些地址做 patch 了，我们可以使用如下的函数来 patch：

- idc.PatchByte(ea, value)
- idc.PatchWord(ea, value)
- idc.PatchDword(ea, value)

---

ea 是地址，value 是值，注意值要和你选择的函数相对应即可，现在我们举个栗子：下面是一些被加密的字符串：

```
.data:1001ED3C aGcquEUdg_bUfuD db 'gcqu^E]~UDG_B[uFU^DC',0
.data:1001ED51 align 8
.data:1001ED58 aGcqs_cuufuD db 'gcqs\_CUuFU^D',0
.data:1001ED66 align 4
.data:1001ED68 aWud@uubQU db 'WUD@UUB^Q]U',0
.data:1001ED74 align 8
```

在我们分析的过程中我们识别出了解密的函数如下：

```
100012A0 push esi
100012A1 mov esi, [esp+4+_size]
100012A5 xor eax, eax
100012A7 test esi, esi
100012A9 jle short _ret
100012AB mov dl, [esp+4+_key] ; assign key
100012AF mov ecx, [esp+4+_string]
100012B3 push ebx
100012B4
100012B4 _loop: ;
100012B4 mov bl, [eax+ecx]
100012B7 xor bl, dl ; data ^ key
100012B9 mov [eax+ecx], bl ; save off byte
100012BC inc eax ; index/count
100012BD cmp eax, esi
100012BF jl short _loop
100012C1 pop ebx
100012C2
100012C2 _ret: ;
```

```
100012C2 pop esi
100012C3 retn
-------------------------------------------------
```

这是一个标注的 xor 加密函数，它具有大小，密钥和加密字符串三个参数。

```
---------------------------------------
Python>start = idc.SelStart()
Python>end = idc.SelEnd()
Python>print hex(start)
0x1001ed3c
Python>print hex(end)
0x1001ed50
Python>def xor(size, key, buff):
for index in range(0,size):
cur_addr = buff + index
temp = idc.Byte( cur_addr ) ^ key
idc.PatchByte(cur_addr, temp)
Python>
Python>xor(end - start, 0x30, start)
Python>idc.GetString(start)
WSAEnumNetworkEvents
---------------------------------------
```

我们使用 idc.SelStart() 和 idc.SelEnd()来获取我们选取的字符串的起始地址和结束地址，然后我们使用了 idc.Byte(ea)获取字节，解密之后，利用 idc.PatchByte(ea, value)函数把补丁打上。

16. 输入和输出

在 IDAPython 中当我们并不知道文件的位置或者并不知道用户想要把他们的数据存储在什么地方，输入输出文件就很重要了。导入导出文件我们可以使用 AskFile(forsave, mask, prompt)这个函数，当 forsave 参数为 0，打开一个文件对话框，当 forsave 的参数为 1，打开一个文件保存对话框，mask 用来指定文件后缀或者模式，比如我只想打开.dll 文件的话就可已使用“*.dll”作为 mask 的参数，prompt 是窗口的名称，以下是一个展示输入输出好栗子，利用选择的数据形成以下 IO_DATA 这么一个类。

```
-----------------------------------------------------
import sys
import idaapi
class IO_DATA():
    def __init__(self):
        self.start = SelStart()
        self.end = SelEnd()
        self.buffer = ''
        self.ogLen = None
        self.status = True
        self.run()
```

```
    def checkBounds(self):
        if self.start is BADADDR or self.end is BADADDR:
            self.status = False
    def getData(self):
    '''get data between start and end put them into object.buffer'''
        self.ogLen = self.end - self.start
        self.buffer = ''
        try:
            for byte in idc.GetManyBytes(self.start, self.ogLen):
                self.buffer = self.buffer + byte
        except:
            self.status = False
        return
    def run(self):
    '''basically main'''
        self.checkBounds()
        if self.status == False:
            sys.stdout.write('ERROR: Please select valid data\n')
            return
        self.getData()
    def patch(self, temp = None):
    '''patch idb with data in object.buffer'''
    if temp != None:
        self.buffer = temp
        for index, byte in enumerate(self.buffer):
            idc.PatchByte(self.start+index, ord(byte))
    def importb(self):
    '''import file to save to buffer'''
        fileName = idc.AskFile(0, "*.*", 'Import File')
        try:
            self.buffer = open(fileName, 'rb').read()
        except:
            sys.stdout.write('ERROR: Cannot access file')
    def export(self):
    '''save the selected buffer to a file'''
        exportFile = idc.AskFile(1, "*.*", 'Export Buffer')
        f = open(exportFile, 'wb')
        f.write(self.buffer)
        f.close()
    def stats(self):
        print "start: %s" % hex(self.start)
        print "end: %s" % hex(self.end)
        print "len: %s" % hex(len(self.buffer))
-----------------------------------------------------
```

利用该类的话可以很方便的选择数据并将其存到一个文件中去。这在解密某个 ida 文件中的加密数据是很有用的，我们使用 IO_DATA 获取数据，并在 python 中解密，在重新 patch 回 idb 文件。下面的栗子展示了如何使用 IO_DATA 类。

```
Python>f = IO_DATA()
Python>f.stats()
start: 0x401528
end: 0x401549
len: 0x21
```

授之以鱼不如授之以渔，在下面的解释中，obj 我们分配给类的任意变量，在 f = IO_DATA()中 f 就是 obj。

- obj.start

选取数据的开始地址

- obj.end

选取数据的结束地址

- obj.buffer

选取数据的二进制数

- obj.ogLen

buffer 的大小

- obj.getData()

先将选取数据的开始地址和结束地址之间的数据拷贝到 buffer 中，然后再获取 buffer 中数据

- obj.patch()

用 buffer 中的数据进行 patch

- obj.patch(d)

利用 d 中的数据进行 patch

- obj.importb()

打开一个文件，然后导入数据

- obj.buffer.obj.export()

将 buffer 中的数据导出，并存成一个文件

- obj.stats()

打印该对象的起始地址，结束地址和 buffer 的大小。

---

17. 因特尔 Pin 记录工具

Pin 是 IA-32 和 x86-64 的动态二进制检测框架。如果将 PIN 的动态分析和 IDA 的静态分析结合起来，爽歪歪。所以让我们先安装 Pin 和把它运行起来好了，以下步骤指导安装，执行跟踪可执行文件的 Pintool 并将执行的地址添加到 IDB。（只需要 30s）

```
Notes about steps
    * Pre-install Visual Studio 2010 (vc10) or 2012 (vc11)
```

* If executing malware **do** steps 1,2,6,7,8,9,10 & 11 in an analysis machine

1. Download PIN
    * **https**://software.intel.com/en-us/articles/pintool-downloads
    * Compiler Kit is **for version** of Visual Studio you are **using**.
2. Unzip pin **to** the root dir and **rename** the **folder to** "pin"
    * example path C:\pin\
    * There is a known but that Pin will not always parse the arguments correctly **if** there is spacing in the **file** path
3. Open the following **file** in Visual Studio
    * C:\pin\source\tools\MyPinTool\MyPinTool.sln
        - This **file** contains all the needed setting **for** Visual Studio.
        - Useful **to** back up and reuse the **directory** when starting **new** pintools.
4. Open the below **file**, **then** cut and paste the code **into** MyPinTool.cpp (currently opened in Visual Studio)
        * C:\pin\source\tools\ManualExamples\itrace.cpp
            - This **directory** along with ../SimpleExamples is very useful **for** example code.
5. Build Solution (F7)
6. Copy traceme.exe **to** C:\pin
7. Copy compiled MyPinTool.dll **to** C:\pin
    * path C:\pin\source\tools\MyPinTool\Debug\MyPinTool.dll
8. Open a **command line and set the working dir to C**:**pin**
9. Execute the following **command**
    * pin -t traceme.exe -- MyPinTool.dll
        - "-t" = name of **file to** be analyzed
        - "-- MyPinTool.dll" = specifies that pin is **to** use the following pintool/dll
10. While pin is executing **open** traceme.exe in IDA.
11. Once pin has completed (**command line will have returned**) **execute the following in IDAPython**
    * The pin output (itrace.out) must be in the working dir of the IDB. \

---

itrace.cpp 是一个 pintool，它可以打印出 itrace.out 中的每条语句执行时候的 EIP，打印的数据如下所示：

00401500
00401506
00401520
00401526
00401549
0040154F
0040155E
00401564

0040156A

在 pintools 执行完成之后我们可以执行如下的 IDAPython 的脚本来给所有执行过的地址加上注释。pintools 输出的文件 itrace.out 需要当 IDB 的工作目录下。

```
f = open('itrace.out', 'r')
lines = f.readlines()
for y in lines:
    y = int(y,16)
    idc.SetColor(y, CIC_ITEM, 0xffffff)
    com = idc.GetCommentEx(y,0)
    if com == None or 'count' not in com:
        idc.MakeComm(y, "count:1")
    else:
        try:
            count = int(com.split(':')[1],16)
        except:
            print hex(y)
        tmp = "count:0x%x" % (count + 1)
        idc.MakeComm(y, tmp)
f.close()
```

我们首先打开 itrace.out 然后读入一个 list 中，然后进行遍历，但是 pintools 输出的文件地址为 16 进制字符串，我们需要将其转换成为 integer 数值。

```
.text:00401500 loc_401500:  ;  CODE
XREF: sub_4013E0+106j
.text:00401500 cmp ebx, 457F4C6Ah;count:0x16
.text:00401506 ja short loc_401520;count:0x16
.text:00401508 cmp ebx, 1857B5C5h; count:1
.text:0040150E jnz short loc_4014E0; count:1
.text:00401510 mov ebx, 80012FB8h; count:1
.text:00401515 jmp short loc_4014E0; count:1
.text:00401515; ---------------------------------------
.text:00401517 align 10h
.text:00401520
.text:00401520 loc_401520: ; CODE
XREF: sub_4013E0+126j
.text:00401520 cmp ebx, 4CC5E06Fh ;count:0x15
.text:00401526 ja short loc_401549 ;count:0x15
```

---

18. 批生成文件

有时，为目录中的所有文件创建 IDB 或 ASM 可能很有用。 在分析属于同一系列恶意软件的一组样本时，这可以帮助节省时间。比起手工做这件事情，写一个批处理文件会容易许多，我们只需要将-B 该参数传给 idaw.exe 即可， 下面的代码可以被复制到包含我们想为其生成文件的所有文件的目录中。

```
import os
import subprocess
import glob
paths = glob.glob("*")
ida_path = os.path.join(os.environ['PROGRAMFILES'], "IDA",
"idaw.exe")
for file_path in paths:
if file_path.endswith(".py"):
continue
subprocess.call([ida_path, "-B", file_path
```

我们利用 glob.glob("*")来获取当前文件夹下面的所有文件，通过变动该参数可以选取各种不同的我们所想要的文件类型，举个栗子，如果我们只想要 exe 后缀的文件，那么使用 glob.glob("*.exe")就可以了。os.path.join(os.environ['PROGRAMFILES'], "IDA", "idaw.exe")用来获取 idaw.exe 文件路径，有些版本的 ida 的文件夹名称会有版本号存在，如果它不是“IDA “的话，就需要我们进行更改，当然，你更改过 ida 的安装路径的话，额，自己看着改吧。然后我们遍历想要处理文件目录下的文件，剔除掉 py 文件后，传给 IDA。单个文件的处理过程就像”C:\Prog ram Files\IDA\idaw.exe -B bad_file.exe “这样。脚本跑完之后，它会在当前目录下为所有的文件生成 ASM 和 IDB 文件。如下所示：

```
C:\injected>dir
0?/**/____ 09:30 AM <DIR> .
0?/**/____ 09:30 AM <DIR> ..
0?/**/____ 10:48 AM 167,936 bad_file.exe
0?/**/____ 09:29 AM 270 batch_analysis.py
0?/**/____ 06:55 PM 104,889 injected.dll
C:\injected>python batch_analysis.py
Thank you for using IDA. Have a nice day!
C:\injected>dir
0?/**/____ 09:30 AM <DIR> .
0?/**/____ 09:30 AM <DIR> ..
0?/**/____ 09:30 AM 506,142 bad_file.asm
0?/**/____ 10:48 AM 167,936 bad_file.exe
0?/**/____ 09:30 AM 1,884,601 bad_file.idb
0?/**/____ 09:29 AM 270 batch_analysis.py
0?/**/____ 09:30 AM 682,602 injected.asm
```

```
0?/**/___ 06:55 PM 104,889 injected.dll
0?/**/___ 09:30 AM 1,384,765 injected.idb
-------------------------------------------------------
```

bad_file.asm , bad_file.idb , injected.asm and injected.idb 是生成的文件。

19. 可执行脚本

IDAPython 脚本可以再命令行中执行，我们也可以使用下面计算 IDB 拥有指令个数的脚本，然后将其个数写进一个叫做“instru_count.txt”文件中。

```
-------------------------------------------------------
import idc
import idaapi
import idautils
idaapi.autoWait()
count = 0
for func in idautils.Functions():
# Ignore Library Code
    flags = idc.GetFunctionFlags(func)
    if flags & FUNC_LIB:
        continue
    for instru in idautils.FuncItems(func):
        count += 1

f = open("instru_count.txt", 'w')
print_me = "Instruction Count is %d" % (count)
f.write(print_me)
f.close()
idc.Exit(0)
-------------------------------------------------------
```

上面有两个非常重要的函数就是 idaapi.autoWait() 和 idc.Exit(0)，当 IDA 打开一个文件的时候，等待 IDA 分析完成时很重要的，因为 IDA 分析一个文件需要花大量的时间。这时候你不能执行 IDAPython 脚本，所以你可使用 idaapi.autoWait()来等待 IDA 分析文件结束，它会在 IDA 分析完成之前一直等待，一旦分析完成，控制权就会交到脚本身上。然后我们同样需要使用 idc.Exit(0)来结束脚本的执行，如果不这么做的话，IDB 可以会在关闭的时候出问题。

如果我们想要计算 IDB 包含多少行的话我们可以使用以下的指令完成：

```
-------------------------------------------------------------------
C:\Cridix\idbs>"C:\Program Files (x86)\IDA 6.3\idaw.exe" -A -
S count.py cur-analysis.idb
-------------------------------------------------------------------
```

-A 表示自动化，-S 表示 idb 在被打开后立即执行脚本。执行完成之后我们可以看到目录找那个有一个叫做 instru_count.txt 的文件，它包含了所有指令的计数。

20. 结束语

希望大家学到了关于使用 IDAPython 来解决你问题的知识哈，如果你有问题，评论，或者反馈给我的话，请给我邮件（我是不会收的），我会继续更新这本书，希望大家多多关注。