IDAPython 手册（翻译 by y0n）

介绍

这是一本关于 IDAPython 的手册，我最初写它是作为自己参考的，在我通常使用 IDAPython（忘记了）我想在某处能找到函数的例子。自从我开始这本书以来，我多次使用它作为快速参考来理解语法或者查看一些例子代码。

如果你跟随我的博客，你可能会注意到一些熟悉的面孔－很多脚本，我在下面贴出了一些在线实验的结果。

多年来，我收到许多电子邮件，问对于 IDAPython 什么是最好的学习指南？通常我会给他们指出 Ero Carrera 介绍的 idapython 或在 idapython 的公共的 repo 示例脚本。它们是学习的极好来源，但它们并没有涉及到我遇到的一些常见问题。我想写一本书，涵盖了这些问题。我觉得对于学习 idapython 的人或者想快速参考学习例子和代码片段的人是有价值的。作为电子书，它不会是静态的。我计划在未来定期更新它。

如果您遇到任何问题，错别字或有疑问，请发邮件给我。

更新

版本：1.0

免责声明

这本书不是为初学逆向工程的人设计的。它也不能作为一本介绍 IDA 的书。如果你是刚开始接触 IDA，我建议你买 Chris Eagles 的 IDA Pro。这本书是优秀的且值得花钱的。

购买这本书有两个先决条件，你应该能很轻松的阅读汇编在一个你想工程的背景下并且知道你所使用的 IDA。如果你碰到一个问题，问自己“如何使用 IDAPython 自动化完成一个任务？”那么这本书可能适合你。如果你已经有一些 IDAPython 编程，在这种情况下这本书不适合你。这本书适合初学 IDAPython 的人，它将作为一个手册方便查找例子，常用函数，但是有经验的你已经有了自己的一些脚本参考。

应该说，我的背景是恶意代码逆向工程，本书不涉及编译器概念，如静态分析中使用的基本块或其他学术概念。原因是，当逆向工程恶意软件时，我很少使用这些概念。偶尔我也用它们去混淆代码基本够用，我觉得他们会对初学者来说是有价值的。读了这本书之后，读者会感觉轻松并挖掘自己 idapython 文档。最后一个免责声明，IDA 调试器的功能不包括在内。

规定

IDA 的输出窗口（命令行接口）用于示例和输出。为了简洁起见，有些示例不包含当前地址对变量的赋值。通常表示为 EA = here()。所有的代码都可以剪切和粘贴到命令行或 IDA 的脚本命令的选项 shift-f2。从头到尾阅读是本书推荐的方法。有许多例子并不是一行一行解释的，因为它假定读者理解前面例子中的代码。不同的作者调用 IDAPython 的方式不同，有时候被调用为 idc.SegName(ea) 或者 SegName(ea)。在这本书中，我们将使用第一个风格。我发现这个约定更容易阅读和调试。有时使用这个约定时，会抛出一个错误，如下所示：

```
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>
```

如果这种情况发生则需要在显示前包含模块。

IDAPython 背景

IDAPython 创建于 2004 年。这是 Gergely Erdelyi 和 Ero Carrera 的共同努力。他们的目标是结合强大的 python 与自动化分析的 IDA 的类 C 脚本语言 IDC。IDAPython 由三个独立模块组成。第一个是 idc，它是封装 IDA 的 IDC 函数的兼容性模块。第二个模块是 idautils，这是 IDA 里的一个高级实用函数。第三个模块是 idaapi，它允许访问更多低级数据，这些数据能够被类使用通过 IDA。

基础

在挖掘得太深之前，我们应该定义一些关键字，并检查 IDA 的反汇编输出的结构。我们可以使用下面的代码行作为示例。

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

.text 是节的名称，地址是 00012529。显示的地址是 16 进制格式。mov 这个指令被称作助记符。助记符后面的第一个操作是 esi 和第二个操作是[esp+4+arg_0]，当使用 IDAPython 函数工作时，最常用的是传递变量的地址。在 IDAPython 文档中地址作为 ea 被引用。地址可以通过几个不同的函数手动访问。最常用的功能是 idc.ScreenEA()或 here()。它们将返回一个整型值。如果我们想要获得一个最小地址是在一个 idb 中，我们可以使用 MinEA()或者获取最大地址我们可以使用 MaxEA()。

```
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
```

每一个被反汇编的函数能被 IDAPython 访问。下面是一个例子，如何访问每个元素。请重新调用我们预先存储在 ea 中的地址。

```
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]
```

获取表示段名的字符串我们会使用 idc.SegName(ea)与 ea 是一个段地址。打印一个反汇编的字符串可以用 idc.GetDisasm(ea)。值得注意的是函数的拼写。获取助记符或者指令名称，我们可以调用 idc.GetMnem(ea)。获取操作数的助记符我们可以调用 idc.GetOpnd(ea，long n)。第一个参数是地址，第二个 long n 是操作数索引。第一个操作数是 0 和第二个是 1。

在某些情况下，验证一个地址是否存在是很重要的。idaapi.BADADDR 或 BADADDR 可以用来检验有效地址。

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

Segments(段)

打印一行作用不大。IDAPython 的强大来自于遍历所有的指令，交叉引用地址和搜索代码或数据。后面两部分将在后面更详细地描述。遍历所有段将是一个不错的开始的位置。

```
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
```

idautils.Segments()返回一个遍历类型对象，我们可以循环这个对象通过使用一个 for 循环。列表中的每个项都是段的起始地址。如果我们把它作为作为一个参数去调用 idc.SegName(ea)，地址可以被用来获取名称。开始和结束的段可以通过调用 idc.SegStart(ea)或 idc.SegEnd(ea)获得。地址或 ea 需要位于段的开始或结束的范围内。如果我们不想遍历所有段，但想找到下一段我们可以使用 idc.NextSeg(ea)。地址可以是段范围内的任何我们希望找到的下一段的地址。如果有机会我们想要通过名称获取一个段的开始地址，我们可以使用 idc.SegByName(segname)。

Functions(函数)

既然我们知道如何遍历所有段，我们就应该研究如何遍历所有已知函数。

```
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()将返回一个已知函数列表。这个列表将包含起始地址的每一个函数。idautils.Functions()可传递的参数范围内搜索。如果我们想要搜索可以通过开始地址和结束地址 idautils.Funtions(start_addr, end_addr)。获得一个函数的名称我们使用 idc.GetFunctionName(ea)。ea 可以是函数边界的任何地址。IDAPython 含有大量的 API 集合提供使用的函数。让我们从一个简单的功能开始。这个函数的语义不重要，但我们应该在心里创建一个地址的记录。

```
.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
```

获得边界我们可以使用 idaapi.get_func(ea)。

```
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)返回一个类的 idaapi.func_t。有时它并不总是显而易见的如何使用一个类的返回通过一个函数调用。一个有用的命令去查询在 Python 中的类是 dir(class)函数。

```
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']
```

从这个输出我们能明白 startEA 和 endEA 用来访问函数的开始和结束。这些属性只适用于当前函数。如果我们想要访问其他函数我们可以使用 idc.NextFunction(ea) 和 idc.PrevFunction(ea)。ea 的值仅需要在分析的函数的边界的地址里。枚举函数的一个警告是，只有当 IDA 将代码块标识为函数时，它才起作用。在代码块被标记为一个函数之前，它将在函数枚举过程中跳过。未标记为函数的代码将在图例中标记为红色（顶部的颜色栏）。这些可以手动固定或自动。IDAPython 有很多不同的方法来访问相同的数据。访问边界内的一种常见的方法是使用一个函数 idc.GetFunctionAttr(ea, FUNCATTR_START) 和 idc.GetFunctionAttr(ea, FUNCATTR_END)。

```
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)是用来获取开始和结束的函数，然后我们打印当前地址和反汇编通常使用 idc.GetDisasm(ea)。我们使用 idc.NextHead(eax)来获取下个指令的开始和继续

直到我们到达这个函数的末尾。这种方式的一个缺陷是包含在开始和结束的功能边界。如果有一个跳转地址高于函数的末尾循环也将过早的退出。这些类型的跳转在混淆技术（如代码转换）中非常常见。由于边界是不可靠的最好的实践是调用 idautils.FuncItems(ea)去循环函数的每个地址。我们将进入更详细的关于这个方法在下面的部分。

类似于 idc.GetFunctionAttr(ea, attr)另一个有用的函数收集有关于函数的信息是 GetFunctionFlags(ea)。它可以用来检索有关函数的信息，如库代码或函数不返回值。一个函数可能有 9 个标志。如果我们想列举所有的功能，我们可以使用以下代码。

```
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"
```

我们使用 idautils.Functions()来获取所有已知函数列表的地址同时我们使用 idc.GetFunctionFlags(ea)来获取标志。我们检测值通过使用逻辑‘&’在返回值的时候。例如检测是否函数没有返回值我们将使用接下来的比较 if flags & FUNC_NORET。接下来我们重温所有的标志。这些标志是常见的，其他的是罕见的。

FUNC_NORET

这个标志用来标识一个函数没有执行一个返回指令。它的内部表示等于 1。一个不返回值的函数的例子，如下所示：

```
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
```

注意如何 ret 或者 leave 不是上一个指令。

FUNC_FAR

这个标志很少出现，除非逆向软件使用分段内存。它的内部表示为一个整数 2。

FUNC_USERFAR

这个标志是罕见的，具有非常小的文件。hexrays 描述标志为“用户已指定远性功能”。它的内部值为 32。

FUNC_LIB

此标志用于查找库代码。识别库代码非常有用，因为它是在执行分析时通常可以忽略的代码。它的内部表示为整数 4。下面是一个例子，它的使用具有识别功能。

```
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

此标志用于标识作为静态函数编译的函数。在 C 函数中默认是全局的。如果作者定义了一个函数为静态只能访问内部文件等功能。在有限的方式下，这可以用来帮助理解源代码是如何构造的。

FUNC_FRAME

这个标志表明该函数使用帧指针 EBP。使用帧指针的函数通常以设置堆栈框架的标准函数序言开始。

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

FUNC_BOTTOMBP

类似 FUNC_FRAME 此标记用于跟踪帧指针。它将确定帧指针等于堆栈指针函数。

FUNC_HIDDEN

函数带 FUNC_HIDDEN 标志意味着他们是隐藏的将需要扩展到视图。如果我们转到一个被标记为隐藏的函数的地址，它会自动扩展。

FUNC_THUNK

这标志标识函数是 thunk 函数。一个简单的功能是跳到另一个函数。

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

应该指出的是，一个函数可以由多个标志。

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

Instructions(指令)

既然我们知道函数如何访问它们的指令，如果我们有一个函数的地址，我们能使用 idautils.FuncItems(ea)获取列表中所有地址。

```
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)实际返回一个迭代器类型但是被强转成一个 list。该列表将包含顺序连续的每个指令的起始地址。现在我们已经有了一个很好的知识库来遍历段、函数和指令，让我们展示一个有用的例子。有时当逆向包代码是唯一知道在哪里发生动态调用的。一个动态的调用将调用或跳转到一个操作数是一个寄存器，例如调用 eax 或 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
```

我们调用 idautils.Functions()去获取所有已知函数列表。每个函数我们检索函数的标志通过调用 idc.GetFunctionFlags(ea)。如果这个函数是库代码或者 thunk 函数这个函数被跳过。接下来我们调用 idautils.FuncItems(ea)来获取所有的地址在函数里。我们使用 for 循环循环遍历列表。由于我们只对 call 和 jmp 指令感兴趣我们需要通过调用 idc.GetMnem(ea)。然后我们用一个简单的字符串比较检查法。如果是一个 jump 或 call 我们通过操作的类型调用 idc.GetOpType(ea, n)。这个函数将返回一个整数是内部调用 op_t.type。这个值可以用来确定如果操作数是一个寄存器、内存引用等。然后我们检查 op_t.type 是一个寄存器。如果是这样，我们打印行强制返回的 idautils.FuncItems(ea)到一个列表是有用的，因为迭代器没有对象如 len()。通过把它作为一个列表，我们可以很容易地获得函数中的行数或指令数。

```
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
```

在前一个示例中，我们使用了一个包含函数中所有地址的列表。我们循环每条指令并访问下条指令。如果我们只有一个地址，想获得下一个指示？移动到下一条指令的地址可以使用 idc.NextHead(ea)和获得前一条指令地址我们使用 idc.PrevHead(ea)。这些功能将得到下一个指令的开始而不是下一个地址，得到下一个地址我们使用 idc.NextAddr(ea)，得到前一个地址我们使用 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
```

Operands(操作数)

操作数的类型是常用的，它将有助于复习所有的类型。正如前面所述，我们可以使用idc.GetOpType(ea,n)得到的操作数的类型。ea 是地址，n 是索引。这里有 8 中不同类型的操作数类型。

o_void

如果一个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

如果一个操作数是一个普遍的寄存器将返回此类型。这个值在内部表示为 1。

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

o_mem

如果一个操作数是直接内存引用它将返回这个类型。这个值在内部表示为 2。这种类型是有用的在 DATA 段查找引用。

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

o_phrase

这个操作数被返回则这个操作数包含一个基本的寄存器或一个索引寄存器。这个值在内部表示为 3。

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

o_displ

这个操作数被返回则操作数包含寄存器和一个位移值，这个为位移值是一个整数，例如 0x18。这是常见的当一条指令访问值在一个结构中。在内部，它表示为 4 的值。

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

o_imm

操作数是这样一个为整数的 0xc 的值的类型。它在内部表示为 5。

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

o_far

这个操作数不是很常见当逆向 x86 或 x86_64 时。它是用来寻找操作数的访问立即数远地址的。它在内部表示为 6。

o_near

这个操作数不是很常见当逆向 x86 或 x86_64 时。它是用来寻找操作数的访问立即数近地址的。它在内部表示为 7。

Example-1

在逆向一个可执行文件时，我们可能注意到代码不断引用重复的位移值。这是一个可能的指标，代码是通过结构功能的不同。通过一个例子来创建一个 Python 字典，它包含所有的位移作为键，每个键都有一个地址列表。下面的代码中会有一个新的函数尚未被描述。这个函数是 idc.GetOpType(ea, n)。

```
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)
```

代码的开始应该已经很熟悉。我们结合 idautils.Functions()和 GetFunctionFlags(ea)来获取所有合适的函数，而忽略了库和 thunks。我们得到每个指令在一个函数里通过调用 idautils.FuncItems(ea)。从这里一个新的函数 idaapi.decode_insn(ea)被调用。这个函数获得我们想要解码的指令的地址。一旦它被解码，我们可以通过访问它的 idaapi.cmd 访问指令的不同性质。

```
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']
```

当我们了解到 dir()命令 idaapi.cmd 具有大量属性。现在回来看我们的例子。这些操作数类型被访问通过使用 idaapi.cmd.Op1.type。请注意这些操作数开始在 1 而不是 0，这不同于 idc.GetOpType(ea,n)。我们检查如果操作数 1 和操作数 2 是 o_displ 的类型。我们使用 idaapi.tag_remove(idaapi.ua_outop2(ea, n))来获取一个字符串表示的操作数。它将变得更短更容易阅读如果我们调用 idc.GetOpnd(ea, n)。例如这个目的表明这是一个好的方式来显示有多个函数访问属性使用 IDAPython。如果我们查看 IDAPython 的 idc.GetOpnd(ea, n)源代码，我们将看到更底层的方法。

```
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)
```

现在回来看我们的例子，当我们有了字符串的时候我们需要检查这个字符串时候包含“bp”。这是一个快速的方式决定是否寄存器 bp，ebp，或 rbp 中是存在操作数的。我们检查“bp”因为我们需要决定这个偏移值是否合法。访问偏移值我们使用 idaapi.cmd.Op1.addr。这将返回一个字符串。既然这样我们有一个地址我们把它转换成整数，在我们需要的时候，然后增加到我们的字典名称为 displace 中。如果有一个我们想搜索的位移值，我们可以使用下面的 for 循环来访问它。

```
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
```

我们对 0x130 这个偏移值很感兴趣。这可以修改打印其他偏移。

Example-2

有时候当我们逆向一个内存 dump 出的一个可执行文件时操作数不会被识别成一个偏移。

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

第二个值被设置成一个内存偏移。入股哦鸣们右键点击它，改变他的数据类型；我们将看到偏移到一个字符串，这是做一次或两次，但之后我们最好可以自动处理。

```
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)
```

运行上面的代码，我们将立即看到的字符串。

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

我们通过调用 MinEA()和 MaxEA()得到最小值和最大值的地址开始。我们遍历所有功能及说明。我们检查每个指令如果操作数类型是 o_imm 代表立即数是 5。o_imm 类型类似整型值或偏移。一旦找到这个值我们读取这个值通过调用 idc.GetOperandValue(ea,n)。看这个值是否在最小和最大地址范围内。如果是这样，我们使用 idc.OpOff(ea, n, base)来转换操作数的偏移。第一个参数 ea 是地址，n 是操作数的索引和 base 是基地址。我们的例子仅仅需要 base 为 0。

Xrefs(交叉引用)

能够找到交叉引用又名外部参考数据或代码是非常重要的。交叉引用是重要的因为他们因为它提供在某些数据被使用或一个函数被调用的位置。例如，如果我们想找到 WriteFile 被调用的地址。使用交叉引用我们需要做的是确定 WriteFile 的地址在导入表中然后找到所有的交叉引用。

```
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
```

第一行我们得到了 API WriteFile 的地址通过使用 idc.LocByName(str)。这个函数将返回一个 API 的地址。我们输出 WriteFile 的地址是一个字符串表示的。然后遍历所有的交叉引用通过调用 idautils.CodeRefsTo(ea, flow)。它能通过遍历将返回一个迭代器。ea 是我们想要得到的交叉引用的地址。参数流是一个 bool 类型。它被用来指定是否要遵照正常的代码流。然后显示每一个交叉引用的地址。一个快速的注释关于使用 idc.LocByName(str)。所有的重命名函数和 APIs 在一个 IDB 中能被访问通过调用 idautils.Names()。这个函数返回一个迭代器对象能够循环遍历输出或者访问名称。每个名称的项是一个(ea, str_name)的元组。

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

如果我们想要获得代码在哪里被引用，我们需要使用 idautisl.CodeRefsFrom(ea,flow)。例如让我们获取在 0x10004932 被引用的地址。

```
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
```

如果我们复查 idautils.CodeRefsTo(ea, flow)这个例子，我们将看见地址 0x10004932 是一个到 WriteFile 的地址。idautils.CodeRefsTo(ea, flow)和 idautils.CodeRefsFrom(ea, flow)是用来搜索交叉引用和代码。使用 idautils.CodeRefsTo(ea, flow)的限制是它是 API，是动态导入的，然后手动重命名将不会显示为代码交叉引用。我们手动重命名一个 dword "RtlCompareMemory"地址使用 idc.MakeName(ea, name)。

```
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 将不标记这些 APIs 作为交叉引用。接下来我们将描述一个一般技术获得所有交叉引用。如果我们想要搜索交叉引用从数据中我们能使用 idautils.DataRefsTo(e) 或者 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)获取参数的地址，并返回所有交叉引用数据的地址的迭代器。

```
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
```

对于逆向和显示地址我们调用 idautils.DataRefsFrom(ea)，通过地址作为参数。他返回一个一个所有地址交叉引用返回给数据的迭代器。使用不同的代码和数据可能有点混乱。如前所述，让我们描述一个更通用的技术。这种方法可以通过调用单个函数获得所有对地址的交叉引用。我们获得到一个地址的所有交叉引用使用 idautils.XrefsTo(ea, flags=0)，获得从一个地址到所有交叉引用通过调用 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"
```

第一行显示我的们的地址和一个字符串命名的<Path>。我们使用 idautils.XrefsTo(ea, 1) 获得的所有交叉引用的字符串。我们然后使用 xref.type 来打印交叉引用类型值。idautils.XrefTypeName(xref.type)被用来打印这个类型的字符串表示。有十二个不同的文件引用类型的值。该值可以在左边看到，它的对应名字可以被看到，如下所示。

```
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 打印从哪输出地址，xref.to 打印两个地址。xref.iscode 打印是否交叉引用是在一个代码段中。在前面的例子中，我们使用 idautils.XrefsTo(ea, 1)的标志设置值为 1。如果这个标志为 0 任何交叉引用将不被显示。我们有如下反汇编。

```
.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], 1
```

我们在 1000AB02 出进行标记。这个地址从 1000AAF6 有一个交叉引用，但是它也有第二个交叉引用。

```
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
```

第二个交叉引用是从 1000AAFF 到 1000AB02。交叉引用不会造成分支指令。他们也可以被普通的代码流影响。如果我们设置标志为 1 Ordinary_Flow 引用类型将不被增加。回到之前的 RtlCompareMemory 例子。我们能使用 idautils.XrefsTo(ea,flow)来获取交叉引用。

```
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
```

有时获得所有交叉引用有点多余。

```
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
```

冗长的来自 Data_read 和 Code_near 都添加到交叉引用。让所有的地址并将它们添加到一个集合可以使所有有用的地址被留下。

```
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
```

例如剩下的函数 GetProcessHeap 的例子。

```
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']
```

Searching(搜索)

我们已经遍历了一些基本的我们所知道的函数或指令。这是有用的，但是有时我们需要搜索特殊的字节例如 0x55 0x8B 0xEC。这些字节搭配是经典的功能语句 push ebp，mov ebp，esp。为了搜索这些字节或者二进制搭配我们可以使用 idc.FindBinary(ea, flag, searchstr, radix=16)。ea 是我们想要搜索的地址从标志是字典或条件中。这里有一些不同类型的标志。名称和值如下所示。

```
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 **
** Older versions of IDAPython do not support these
```

并不是所有这些标志都值得掌握，而是接触到最常用的标志。

SEARCH_UP 和 SEARCH_DOWN 被用来选择字典我们希望我们搜索到接下来的。

SEARCH_NEXT 被用来获得下一个找到的对象。

SEARCH_CASE 被用于指定大小写敏感度。

SEARCH_NOSHOW 将不显示搜索过程。

SEARCH_UNICODE 被用来处理所有的 Unicode 字符串搜索。

我们正在寻找的模式是 searchstr。radix 被使用当正在写入进程模块时。本课题是本书的范围之外。我将推荐阅读《IDA Pro 权威指南》第 19 章。现在 radix 字段可以留空。快速浏览一下前面提到的函数语句字节模式。

```
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
```

在第一行我们发现我们搜索的部分。被搜索的部分能以 16 进制的形式开始带 0x 作为在 0x55 0x8B 0xEC 或者作为字节 55 8B EC 出现在 IDA 的 16 进制窗口。\x55\x8B\xEC 不能被使用除非我们使用 idc.FindText(ea, flag, y, x, searchstr)。MinEA()被用来获得第一个地址在可执行的位置。然后我们将 idc.FindBinary(ea, flag, searchstr, radix=16)返回的给一个叫做 addr 的变量。

在搜索时验证搜索没有找到匹配部分是重要的。它被测试通过对比 addr 的 idc.BADADDR。然后打印地址和反汇编。注意地址没有增加吗？这是因为我们没有通过 SEARCH_NEXT 标志。如果这个标志是没有通过的当前地址被用来搜索匹配部分。如果上一个地址包含我们的自节部分搜索将不在通过。如下所示。

```
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
```

搜索字节是有有利的但是有时候我们也许想要搜索字符串例如“chrome.dll”。我们能转换字符串到 16 进制字节使用[hex(y) for y in bytearray("chrome.dll")]但是这有一点丑陋。另外，如果字符串是 Unicode，我们必须解释这种格式。最简单的方式是使用 FindText(ea, flag, y, x, searchstr)。大多数字节应该看起来相似，因为他们是相同的作为 idc.FindBinary。ea 是开始地址，flag 是方向和类型用于搜索。y 是要搜索的 ea 中的行数，x 是行中的坐标。这些字段通常分配为 0。现在搜索出现的字符串“Accept”。任何字符串从字符串窗口按 shift+F12 能使用对于这个例子。

```
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
```

我们使用 MinEA()获得最小地址和将其赋值给变量名 cur_addr。同样的做法对于最大地址调用 MaxEA()和赋值返回给变量名 end。由于我们不知道字符串出现的次数，我们需要检查搜索是否继续下去，并且小于最大地址。我们然后赋值 idc.FindText 的返回值给当前地址。因为我们将手动增加地址通过调用 idc.NextHead(ea)。我们不需要 SEARCH_NEXT 这个标志。理由是我们手动增加当前地址到接下来的行中是因为一个字符串能在一行中出现多次。这便使它很难获得下一个字符串的地址。

除了前面描述的模式搜索外，还有两个函数可用于查找其他类型。根据 API 的名称可以很容易的推断出函数的整体功能。在讨论寻找不同类型之前，我们首先通过地址来识别类型。有一个 API 的子集，首先是可以用来确定一个地址类型。这个 APIs 返回一个 True 或 False 的布尔值。

idc.isCode(f)

如果 IDA 标记地址为代码，返回 True。

idc.isData(f)

如果 IDA 标记地址为数据，返回 True。

idc.isTail(f)

如果 IDA 标记地址为尾部，返回 True。

idc.isUnknown(f)

如果 IDA 标记地址为未知，返回 True。这个类型被使用当 IDA 没有标识地址是代码还是数据。

idc.isHead(f)

如果 IDA 标记地址为头部，返回 True。

f 对我们来说是新的。而不是通过一个地址我们首先需要一个内部的标志来表示然后通过 idc. is 设置函数。获得这个内部标志我们使用 idc.GetFlags(ea)。现在我们有了一个基本的功能，如何使用对不同类型的函数，让我们举一个简单的例子。

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

idc.FindCode(ea, flag)

它被用来找到下一个被标记为代码的地址。这可能是有用的如果我们想找到一个数据块的结束。如果 ea 是一个已经作为为代码被标记的地址它将返回下一个地址。这个 flag 被用来提前描述 idc.FindText。

```
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
```

我们能俩姐 ea 是地址 0x4140e8 的一些数据。我们赋值 idc.FindCode(ea, SEARCH_DOWN|SEARCH_NEXT)的返回值给 addr。然后我们打印 addr 和它的汇编。通过调用一个函数，我们跳过 36 字节数据的开始标记为代码的部分。

idc.FindData(ea, flag)

这的使用和 idc.findcode 用完全一样的除了它将返回下一个地址被标记为一个数据块的开始。如果我们逆向前面的场景，从代码的地址开始，查找数据的开始。

```
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
```

只有一点比前面的例子稍有不同的是 SEARCH_UP|SEARCH_NEXT 的方向和搜索的数据。

idc.FindUnexplored(ea, flag)

这个函数是用来寻找的字节地址，IDA 没有识别码或数据。unknown 类型将需要进一步的人工分析或通过脚本观察。

```
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)

它是用来寻找一个地址，IDA 确定为代码和数据。

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

这可能看上去不是真实的值，但如果我们要打印的 addr 的交叉引用，我们会看到它被使用。

```
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)

并不是寻找一种我们可能要搜索的一个特定的值。比方说，我们有一种感觉，代码调用 rand 来生成一个随机数，但是我们找不到代码。如果我们知道 rand 使用值 0x343fd 作为种子，我们可以寻找一些数。

```
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
```

在第一行我们获取最小地址通过 MinEA()，接下来，然后搜索值 0x343fd。而不是返回一个地址来显示找到的 APIs idc.FindImmediate 返回一个 tupple。tupple 的第一项将成为地址第二项将成为操作数。类似于 idc.GetOpnd 第一个操作数开始返回零。当我们打印的地址和反汇编，我们可以看到的值是第二个操作数。如果我们想搜索所有使用立即数我们可以做如下内容。

```
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
```

大部分的代码应该看起来很熟悉，但因为我们是搜索多个值我们将使用一个 while 循环和 SEARCH_DOWN|SEARCH_NEXT 标志。

Selecting Data(选择数据)

不要总是将我们想要写的代码自动搜索代码或数据。在某些情况下，我们已经知道代码或数据的位置，但我们希望选择它进行分析。在这样的情况下，我们可能只是想高亮的代码开始工作在 IDAPython 中。获得选择数据的边界我们可以使用 idc.SelStart()来获得开始，idc.SelEnd()来获得结束。我们有下面的代码选择。

```
.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]
```

我们可以使用下面的代码打印地址。

```
Python>start = idc.SelStart()
Python>hex(start)
0x408e46
Python>end = idc.SelEnd()
Python>hex(end)
0x408e58
```

我们把 idc.SelStart()的返回值复制给 start。这将是第一个选定的地址的地址。我们然后使用 idc.SelEnd()的返回值和赋值给 end。有一点需要注意的是 end 不是上一次被选择的地址而是下一次地址的开始。如果我们宁愿只调用一次 API，我们能使用 idaapi.read_selection()。它将返回一个元祖，第一个值是 bool，表示是否选择读取，第二个是开始的地址和上一次地址的结束。

```
Python>Worked, start, end = idaapi.read_selection()
Python>print Worked, hex(start), hex(end)
True 0x408e46 0x408e58
```

当工作在 64 位样本时需要谨慎。基本地址不总是正确的，因为选中的起始地址会导致整数溢出，而前导数字会是不正确的。

Comments & Renaming(注释和重命名)

一个人相信自己这是不是不是我写的不是我逆向的。添加注释，重命名函数和交互的集合是理解代码在做什么的最好的方式之一。随着时间的推移，一些互动变得多余。在这样的情况下自动化过程是有用的。

我们翻阅一些例子的时候我们应该首先讨论注释和重命名的基本知识。这里有两种类型的注释。第一个是普通的注释，第二个是可重复的注释。普通的注释出现在地址 0041136B 作为 regular comment 文本。可重复注释能被看见在 00411372，00411386 和 00411392。只有最后的注释是手动输入的。当一个指令引用一个包含重复注释的地址（比如分支条件）时，其他注释就会出现。

```
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
```

为了增加注释，我们使用 idc.MakeComm(ea, comment)和重复注释我们使用 idc.MakeRptCmt(ea, comment)。ea 是地址，comment 是字符串，我们要自己添加。下面的代码添加一个注释每次指令为 0 了 xor 寄存器或者这个值。

```
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)
```

如前面所述，我们循环遍历所有函数通过调用 idautils.Functions()，循环遍历所有的指令通过调用 list(idautils.FuncItems(func))。我们读取内存使用 idc.GetMnem(ea)。校验是否等于 xor。倘若，我们验证的操作数是等于 idc.GetOpnd(ea, n)。如果相等，我们创建一个操作数的字符串然后增加一个不重复的注释。

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

为了添加一个重复注释我们将替换 idc.MakeComm(ea, comment)为 MakeRptCmt(ea, comment)。这可能更有用一些，因为我们将看到对分支的引用，这些分支将值清零，可能返回 0。为了获得注释我们简单的使用 GetCommentEx(ea, repeatable)。ea 是包含注释的地址 repeatable 是 True 或者 False 的布尔值。获得以上注释我们将使用下面的代码片段。

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

如果注释是重复的我们将替换 idc.GetCommentEx(ea, False)为 idc.GetCommentEx(ea, True)。指令不仅仅是字段，可以添加注释。函数也能郑家注释。给函数增加注释我们使用 idc.SetFunctionCmt(ea, cmt, repeatable)，获得函数的注释我们调用 idc.GetFunctionCmt(ea, repeatable)。ea 能是任何地址在函数开始到结束的边界。cmt 是字符串注释我们将增加和重复是一个布尔值取决于我们是否想要这个注释重复。对于注释不能重复的可以表示为 0 或 false；或者对于注释是可重复的表示为 1 或正确。有的函数作为重复的将添加注释当注释被正在被使用。

```
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
```

我们打印地址，反汇编，函数名称在第一行中。我们然后使用 idc.SetFunctionCmt(ea, comment, repeatable)来设置重复注释"check out later"。如果我们在函数的开始处我们将看见我们的注释。

```
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
00401043 push ecx
00401044 push 723EB0D5h
```

既然注释是重复的，当函数有交叉引用的时候我们将看到注释。这是不错的地方增加提醒或记录函数相关信息。

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

重命名函数和地址是一种常见的自动化任务，特别是在处理位置无关代码（PIC）、加壳或包装函数时。这就是为什么常见于 PIC 或脱壳代码中是因为导入表可能不在 dump 的文件中。在封装函数的所有功能情况下简单调用一个 API。

```
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
```

在以上函数代码中可以调用 w_HeapAlloc。w_是 short 前缀。对于重命名一个地址我们使用函数 idc.MakeName(ea, name)。ea 是地址，name 是字符串名称例如"w_HeapAlloc"。对于重命名函数 ea 需要是函数的首地址。重命名我们的 HeapAlloc 封装的函数我们应该使用下列代码。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x10005b3e push ebp
Python>idc.MakeName(ea, "w_HeapAlloc")
True
```

ea 是函数的首地址，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
```

上面我们能看到函数被重命名了。为了确认已被重命名我们能使用idc.GetFunctionName(ea)打印新函数的名称。

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

现在我们有了一个好的知识基础。展示一个例子我们怎样使用我们所学的目的为了自动命名封装的函数。请看内部的注释获得逻辑思路。

```
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)
```

例子输出

```
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
```

大多是代码是相似的。一个可记录的不同点是使用 idc.MakeNameEx(ea, name, flag)来自 rename_wrapper。我们使用这个函数因为 idc.MakeName 将抛出警告对话框如果函数名称已

经被使用。通过 SN_NOWARN 的标志值或者 256 我们可以避免对话框。我们可以应用一些逻辑来重命名函数，如 w_HeapFree_1，但为了简洁起见，我们将把它排除在外。

Accessing Raw Data(访问原数据)

能访问原始数据是扩展的当逆向的时候。原始数据是二进制表示的代码或数据。我们能看到原始指令的数据或字节在左边的地址中。

```
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
```

对于访问数据我们首先需要决定单元大小。命名约定通常是按照访问数据的单位大小来的。为了访问一个字节我们将调用 idc.Byte(ea)或者访问一个字我们将调用 idc.Word(ea)等。

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

如果光标在 00A14380 根据以上在汇编里我们将有如下输出。

```
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
```

当编写解码器的时候它不总是有用的来获取单字节或读取一个 dword，但是可以读取一块原数据。读一个指定大小的字节地址可以使用 idc.GetManyBytes(ea, size, use_dbg=False)。最后一个参数是可选的在必要的时候如果我们想调试内存。

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

有必要注意 idc.GetManyBytes(ea, size)返回 char 表示的字节。这不同于 idc.Word(ea)或

idc.Qword(ea)，它们返回整型。

Patching(打补丁)

有时当逆向病毒样本的时候将字符串被编码了。这样做是为了减慢分析过程，并阻止使用字符串查看器恢复指示器。在这种情况下修补 IDB 是有用的。我们可以对地址重命名但有限制。这是由于命名约定的限制。要用一个值修补地址，我们可以使用以下函数。

idc.PatchByte(ea, value)

idc.PatchWord(ea, value)

idc.PatchDword(ea, value)

ea 是地址和 value 是整型值，我们想要给 IDB 打补丁。值的大小需要与我们选择的函数名指定的大小匹配。例如，我们发现下面的编码字符串

```
.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
```

在我们的分析中，我们能够识别解码器的功能。

```
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
```

函数是一个标准的异或解码带函数参数大小，关键和解码缓冲区。

```
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
```

我们选择高亮数据地址开始和结束使用 idc.SelStart()和 idc.SelEnd()。然后我们有一个函数读字节通过 idc.Byte(ea)，用传递给函数的键对字节进行 XOR，然后通过调用 idc.PatchByte(ea, value)来 patch 字节。

Input and Output(输入输出)

导入导出文件到 IDAPython 可能有用，当我们不知道文件路径或当我们不知道用户想要保存他们的数据在哪里。对于导入和保存一个我呢见通过名称我们使用 AskFile(forsave, mask, prompt)。forsave 是一个 0 的值，如果我们想要打开一个对话框或者 1 表示我们想要打开一个保存对话框。mask 是文件扩展或模式。如果我们仅仅想要打开.dll 文件我们将要使用一个 mask 的"*.dll"，prompt 是窗口的标题。输入，输出和选择数据下面有一个很好的例子 IO_DATA。

```
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))
```

这类数据可以选择保存到缓冲区，然后存储到一个文件。这个编码或加密的数据在 IDB 中是有用的。我们可以用 IO_DATA 选择数据解码，在 Python 缓冲然后补丁 IDB。说明如何使用 io_data 类。

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

与其解释代码的每一行，读者一个接一个地检查这些函数，看看它们是如何工作的。下

面的要点各解释变量和功能。obj 是我们赋值的变量的类。f 是 obj 在 f = IO_DATA()。

obj.start

包含选择偏移的开始的地址

. obj.end

包含选择偏移的结束的地址

obj.buffer

包含二进制数据

obj.ogLen

包含缓冲区大小

obj.getData()

复制二进制数数据在 obj.start 和 obj.end 之间到 obj.buffer，obj.run()选择数据是复制到缓冲区的二进制格式

obj.patch()

补丁 IDB 在 obj.start 处的数据在 obj.buffer

obj.patch(d)

补丁 IDB 在 obj.start 处的数据带参数

obj.importb()

打开一个文件和保存数据

obj.buffer. obj.export()

导出数据在 obj.buffer 到保存的文件中

obj.stats()

打印 obj.start 的 hex, obj.end 和 obj.buffer 长度.

Intel Pin Logger(引脚日志)

引脚是一个 IA-32 和 x86-64 的动态二进制仪表框架。将引脚的动态分析结果与 IDA 的静态分析相结合，使其成为一个强大的组合。IDA 的结合和引脚的一个初始化安装和运行。下面的步骤是 30 秒（减去下载）指导安装，执行 pintool 的过程，可执行文件和添加执行地址到 IDB。

```
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 打印 EIPs 的每个指令执行 itrace.out。数据将进行如下输出。

```
00401500
00401506
00401520
00401526
00401549
0040154F
0040155E
00401564
0040156A
```

Pintools 执行后我们能运行 IDAPython 代码来增加注释到所有的执行地址。输出文件 itrace.out 将需要工作在 IDB 字典中。

```
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()
```

我们打开 itrace.out 和读取所有行到列表中。我们然后遍历每一行在列表中。既然地址在输出的我呢进啊中是 16 进制字符串格式，我们需要转换成整型。

```
.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
```

Batch File Generation(批量处理文件)

有时候需要一次性对目录下的所有文件进行反编译生成 idb 和 asm 文件，尤其是当你在分析一个家族式的样本时这可以给你节约大量的时间，运行批量处理文件可以通过给 idaw.exe 文件加上参数-B 来实现，将以下的代码保存到脚本中并拷贝到需要分析文件的目录下运行。

```
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])
```

通过 glob.glob(*)获取目录下所有的文件路径，该方法也可以获取指定文件的路径，比如需要获取所有.exe 后缀的文件，可以直接使用 glob.glob("*.exe")。os.path.join(os.environ['PROGRAMFILES'], "IDA", "idaw.exe")，用于获取 idaw.exe 的路径，一些版本的 ida 会有一个版本号的父目录，如果是这样的话，其中的参数 IDA 就需要修改为该版本号的父目录名，当然该命令仍需要修改如果你的 ida 安装的时候不是使用标准目录的话。这里我们假设 ida 的安装目录为 C:\Program Files\IDA，之后脚本会循环获取该目录中所有需要反编译的 exe(当然.py 后缀的除外)，如下所示，最后的结果就是对于每一个该目录下的文件，实际上是运行了该命令 C:\Program Files\IDA\idaw.exe -B bad_f ile.exe，一旦该条命令执行完毕，对应的程序的 sam 和 idb 文件就生成了。如下图所示：注意此处由于我的 ida 并不是使用的默认安装，所以此处直接使用硬编码将路径写入到脚本中，运行之后，idaw.exe 启动，并开始批量对该目录下的二进制程序进行反编译。

```
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。

Executing Scripts(执行脚本)

idapython 可以直接在命令行中执行，我们可以使用下面这个脚本记录 idb 文件中的所有指令总数并将其记录到一个名为 instru_count.txt 的文件中。

```
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)
```

在以上的脚本中有两个函数的作用至关重要，它们分别是 idaapi.autoWait()及 idc.Exit()，当 ida 打开一个文件时，需要等待 ida 的分析过程完毕，在这个过程中 ida 会分析所有的函数，结构，以及变量，为了让我们的脚本等待该分析过程完毕，我们需要使用函数 idaapi.autoWait(),该函数运行之后会使脚本阻塞，一旦 ida 分析完毕，将会通过回调函数通知脚本，因此在脚本的一开始调用该函数就显得至关重要，尤其是当你的 ida 脚本的作用是建立在 ida 分析引擎的前提上的时候，脚本执行完毕之后调用函数 idc.Exit(0)，该函数会停止执行的脚本，关闭 idb 数据库。如果我们想通过 idapython 统计某一个 idb 中的所有指令时可以使用到以下指令，注意此处-S 和 countrecord.py 之间没有空格。

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

py rasmedia.idb-A 是自动模式，-S 用于告诉 ida 当打开 idb 是自动加载运行 idapython 脚本，注意-S 之后没有空格，运行之后在生成的 instru_count.txt 中保存了统计的所以指令数。

翻译说明

(2017.9)总算完成了，第一次翻译，主要目的是在于提高英语文档的阅读能力，同时也是为了学习文档中的知识点，方便以后查阅，之后将原文和翻译一起打包分享在我的博客主页中(博客账号被冻结了暂时分享在群 347477763)，其中翻译必定会有很多不足之处，欢迎交流讨论。