

idapython 主要依赖的包由以下三部分组成:

idc

idautils

idaapi

## 一些基础的函数

位置相关函数(用于获取光标, 或特殊数据的地址)

idc.ScreenEA() or here() 返回当前光标位置

MinEA() 返回 idb 的起始地址

MaxEA() 返回 idb 的结束地址

hex(MinEA()) 将起始地址以 16 进制输出, 如下所示:

```
Python>ea = idc.ScreenEA()
Python>print "0x% x % s" % (ea, ea)
0x 10003645 268449349
Python>ea = here()
Python>print "0x% x % s" % (ea, ea)
0x 10003645 268449349
Python>hex(MinEA())
0x10000000L
Python>hex(MaxEA())
0x1000a000L
```

idc.SegName(ea) 获取 ea 地址的段, 比如 text

idc.GetDisasm(ea) 获取 ea 地址的反汇编指令

ida.GetMnem(ea) 获取 ea 地址的反汇编指令操作码

ida.GetOpnd(ea, n) 获取 ea 地址的反汇编指令的第 n 个操作数, 由 0 开始, 实例如下:

```
Python>ea = idc.ScreenEA()
Python>idc.SegName(ea)
.text
Python>idc.GetDisasm(ea)
sub     ecx, eax
Python>ida.GetMnem(ea)
sub
Python>ida.GetOpnd(ea,0)
ecx
Python>ida.GetOpnd(ea,1)
eax
```

有时需要用于判断一个地址是否无效, 可通过 idaapi.BADADDR 实现, 该函数返回一个内置的无效地址, 具体使用如下:

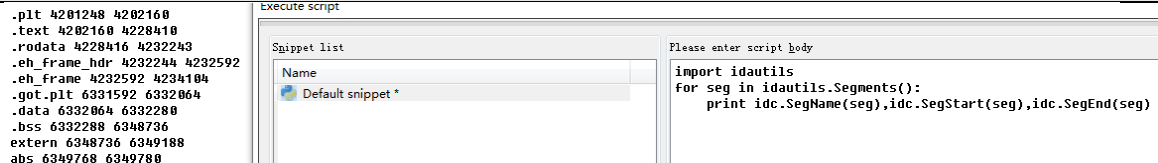
```
if idaapi.BADADDR != here():
    print "vaild addr"
```

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

## 节

idapython 的强大之处在于对 idb 数据库的循环迭代，其中包括指令，交叉引用等，这些会在下文中提到，反编译文件的数据节迭代会是一个不错的开始，代码如下所示：

```
import idutils
for seg in idutils.Segments():
    print idc.SegName(seg),idc.SegStart(seg),idc.SegEnd(seg)
```



idutils.Segments() 函数返回该反编译文件中所有的节信息，托 python 的福，这里我们可以非常方便的直接对这些节信息进行循环迭代，每一次迭代会对一个节进行处理。

每层迭代中使用到以下函数

idc.SegName(seg) 获取节名

idc.SegStart(seg) 获取该节的开始

idc.SegEnd(seg) 获取该节的结束

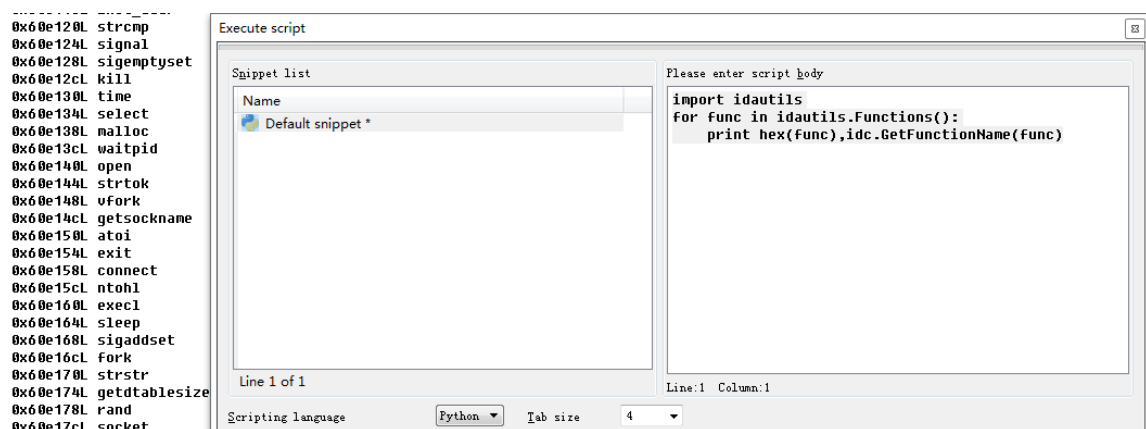
通过函数 idc.NextSeg(ea) 可以获取下一个节，参数为当前节地址范围内任意地址皆可。

```
Python>ea = here()
Python>print "0x% x" % idc.NextSeg(ea)
0x 10006000
```

## 函数

继数据节之后我们下一个目标是函数。

```
import idutils
for func in idutils.Functions():
    print hex(func),idc.GetFunctionName(func)
```



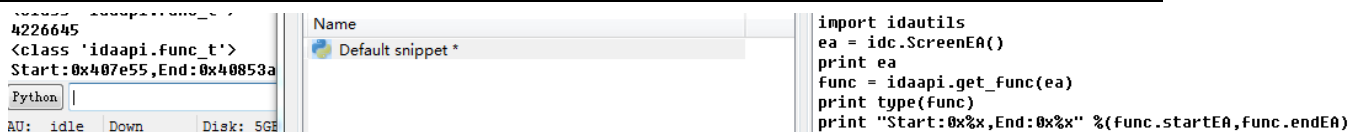
idutils.Functions() 会返回一个所有函数的起始地址列表, 该函数也支持区域查找, 通过给函数传入起始和终止地址, 可以控制搜寻的范围。

如 idutils.Functions(start\_addr,end\_addr), idc.GetFunctionName(ea) 返回一个函数名, ea 可以为该函数中的任意二进制地址。

```
.text:0000000000407E55      push     rbp
.text:0000000000407E56      mov      rbp, rsp
.text:0000000000407E59      push     rbx
.text:0000000000407E5A      sub      rsp, 1488h
.text:0000000000407E61      mov      [rbp+argc], edi
.text:0000000000407E67      mov      [rbp+argv], rsi
.text:0000000000407E6E      mov      edi, 0          ; timer
.text:0000000000407E73      mov      eax, 0
```

通过函数 idaapi.get\_func(ea), 获取一个 idaapi.func\_t 的类, 该类定义了函数的一些属性, 如下脚本所示, 通过该返回的数据结构获取该函数的边界值。

```
import idutils
ea = idc.ScreenEA()
print ea
func = idaapi.get_func(ea)
print type(func)
print "Start:0x%x,End:0x%x" %(func.startEA,func.endEA)
```



idaapi.get\_func(ea) 会返回一个函数类对象, 通过命令 dir(class) 可以查看该类的导出函数和属性。

```
Python>dir(func)
['_class_', '_del_', '_delattr_', '_dict_', '_doc_', '_eq_', '_format_', '_getattr_', '_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_', '_labeledqty_', '_llabels_', '_overlaps_', '_owner_', '_pntqty_', '_points_', '_referers_', '_refqty_',
'_regargqty_', '_regargs_', '_regvarqty_', '_regvars_', '_size_', '_startEA_', '_tailqty_', '_tails_', '_this_', '_thisown_']
```

通过 idc.NextFunction(ea) 和 idc.PrevFunction(ea) 分别可以获取当前地址前后的两个函数的地址, 传入的 ea 参数为当前函数区间的所有合法地址。

```
import idutils
ea = idc.ScreenEA()
```

```
print ea
next = idc.NextFunction(ea)
pre = idc.PrevFunction(ea)
print "the next is 0x%x,the pre is 0x%x" %(next,pre)
```

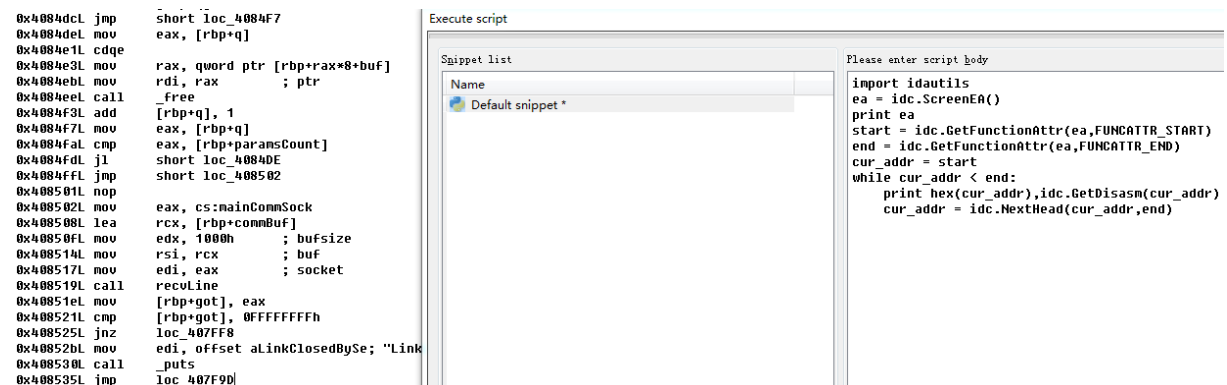


idapython 提供了多种获取数据信息的方式，另一种获取函数边界值的方式可以通过以下两个函数。

通过函数 `idc.GetFunctionAttr(ea, FUNCATTR_START)`，  
`idc.GetFunctionAttr(ea, FUNCATTR_END)`。获取地址 `ea` 所在函数的开始和结尾。  
 通过函数 `idc.GetDisasm(ea)` 获取当前地址的反汇编代码。  
 通过函数 `idc.NextHead(ea)` 获取当前地址之后下一条指令的起始地址。

但是有时候当一个函数中存在一个大跳时，通过以上两种方式获取的函数边界值就会出现问題，这个时候最好的方式是通过函数 `idutils.FuncItems(ea)` 实现该功能，具体的使用会在后文中提到。

```
import idutils
ea = idc.ScreenEA()
print ea
start = idc.GetFunctionAttr(ea, FUNCATTR_START)
end = idc.GetFunctionAttr(ea, FUNCATTR_END)
cur_addr = start
while cur_addr < end:
    print hex(cur_addr),idc.GetDisasm(cur_addr)
    cur_addr = idc.NextHead(cur_addr,end)
```



类似于 `idc.GetFunctionAttr(ea, attr)`，`idc.GetFuntionAttr(ea)` 也是用于获取函数的一些信息。

通过函数 `idc.GetFuntionAttr(ea)` 返回该函数的类型，idapython 中共提供了 9 类函数类型的标签。

可通过下列脚本列举函数标签，

```
import idutils
ea = idc.ScreenEA()
print ea
for func in idutils.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"

```

通过 `idautils.Functions()` 获取所有函数的地址列表，通过 `idc.GetFunctionGFlags(func)` 获取函数的 flag，通过 `flags&` 来判断函数的类型。

```

0x401e40L FUNC_HIDDEN
0x401e40L FUNC_THUNK
0x401e50L FUNC_HIDDEN
0x401e50L FUNC_THUNK
0x401e60L FUNC_HIDDEN
0x401e60L FUNC_THUNK
0x401e70L FUNC_HIDDEN
0x401e70L FUNC_THUNK
0x401e80L FUNC_HIDDEN
0x401e80L FUNC_THUNK
0x401e90L FUNC_HIDDEN
0x401e90L FUNC_THUNK
0x401ea0L FUNC_HIDDEN
0x401ea0L FUNC_THUNK
0x401eb0L FUNC_FRAME
0x401f2dL FUNC_FRAME
0x401fdbL FUNC_FRAME
0x4020a7L FUNC_STATIC
0x4020a7L FUNC_FRAME
0x4020f5L FUNC_STATIC
0x4020f5L FUNC_FRAME
0x4021e7L FUNC_STATIC
0x4021e7L FUNC_FRAME
0x402317L FUNC_STATIC

```

Python  
Convert to data

Execute script

Snippet list

Name
Default snippet *

Please enter script body

```

import idautils
ea = idc.ScreenEA()
print ea
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"

```

所有函数的类型如下所述：

`FUNC_NORET`

标示没有返回值的函数，内部用 1 标示。

```

.plt:00000000000401B70 ; void __noreturn __exit(int status)
.plt:00000000000401B70 __exit      proc near                ; CODE XREF: fdopen+192↓p
.plt:00000000000401B70                                     ; processCmd+184↓p ...
.plt:00000000000401B70 | jmp      cs:_exit_ptr
.plt:00000000000401B70 __exit      endp

```

`FUNC_FAR`

该标识的函数很少使用除非被逆向的软件中包含内存节，内部用 2 标示

FUNC\_USERFAR

同样很少见，ida 的文档中

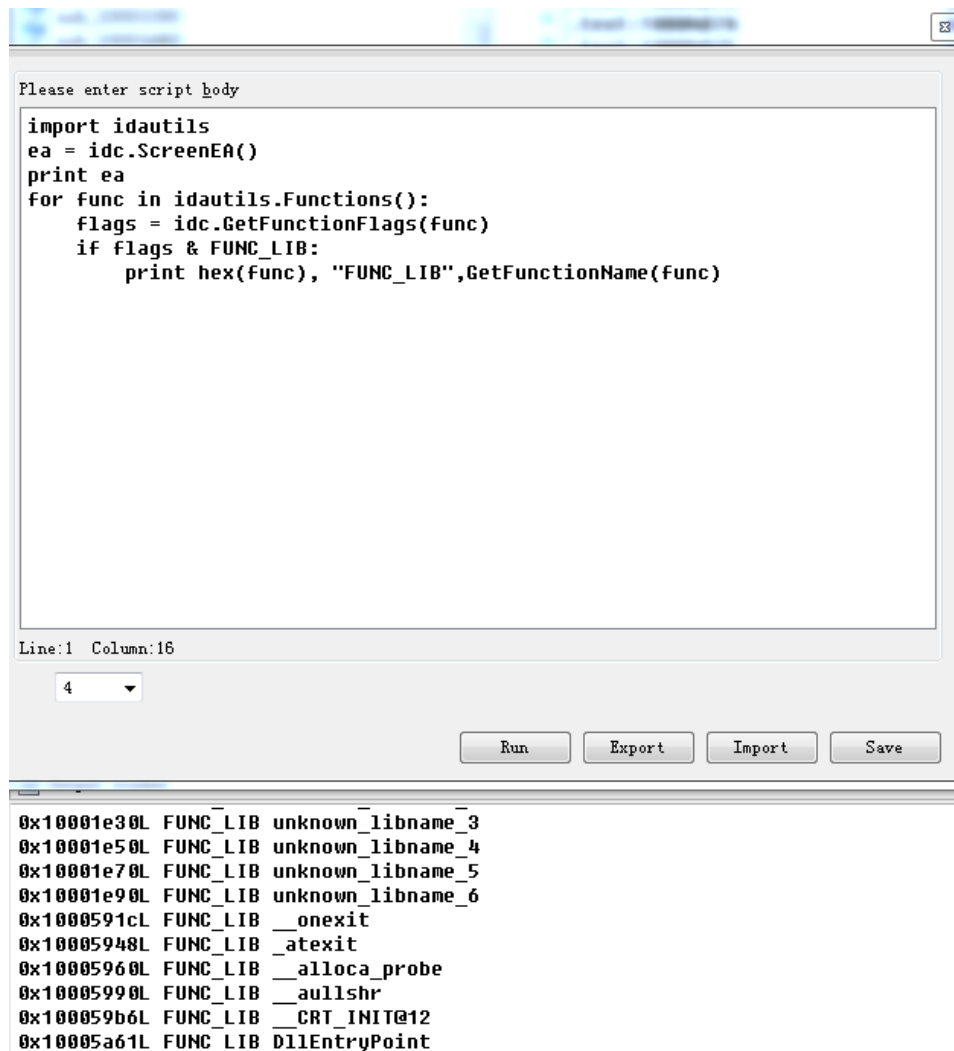
“user has specified far-ness of the function”，内部用 3 标示

FUNC\_LIB

该标识用于标记动态链接库的函数，内部用 4 标记。

以下脚本用于列举所有的链接库函数

```
import idutils
ea = idc.ScreenEA()
print ea
for func in idutils.Functions():
    flags = idc.GetFunctionFlags(func)
    if flags & FUNC_LIB:
        print hex(func), "FUNC_LIB", GetFunctionName(func)
```



FUNC\_STATIC

该标记用于标记 static 函数。

FUNC\_FRAME

该标记用于标示使用了 ebp 作为栈帧的函数，此类函数有典型的开头。具体如下：

```

.text:00000000004030EF ; int __cdecl getCores()
.text:00000000004030EF public getCores
.text:00000000004030EF getCores      proc near
.text:00000000004030EF
.text:00000000004030EF linebuf      = byte ptr -1010h
.text:00000000004030EF cmdline    = dword ptr -8
.text:00000000004030EF totalcores = dword ptr -4
.text:00000000004030EF
.text:00000000004030EF          push    rbp
.text:00000000004030F0          mov     rbp, rsp
.text:00000000004030F3          sub     rsp, 1010h

```

FUNC\_BOTTOMBP

该标记和 FUNC\_FRAME 类似。

FUNC\_THUNK

该标记用于标示中转函数，即只用一个 jmp 的函数

```

.plt:0000000000401D70 ; time_t time(time_t *timer)
.plt:0000000000401D70 _time      proc near          ; CODE XREF: StartTheLe1z+4C2↓p
.plt:0000000000401D70          jmp     cs:time_ptr      ; StartTheLe1z+798↓p ...
.plt:0000000000401D70 _time      endp

```

因此一个函数可以拥有多个函数标记

```

0x401d80L FUNC_HIDDEN
0x401d80L FUNC_THUNK

```

## 指令

目前我们已经知道如何对 idb 中的函数进行操作，接下来通过函数 `idautils.FuncItems(ea)` 获取一个函数中的所有指令地址，该函数会返回包含输入地址所属函数内所有指令地址的一个类 `list` (不是 `list`)。

```

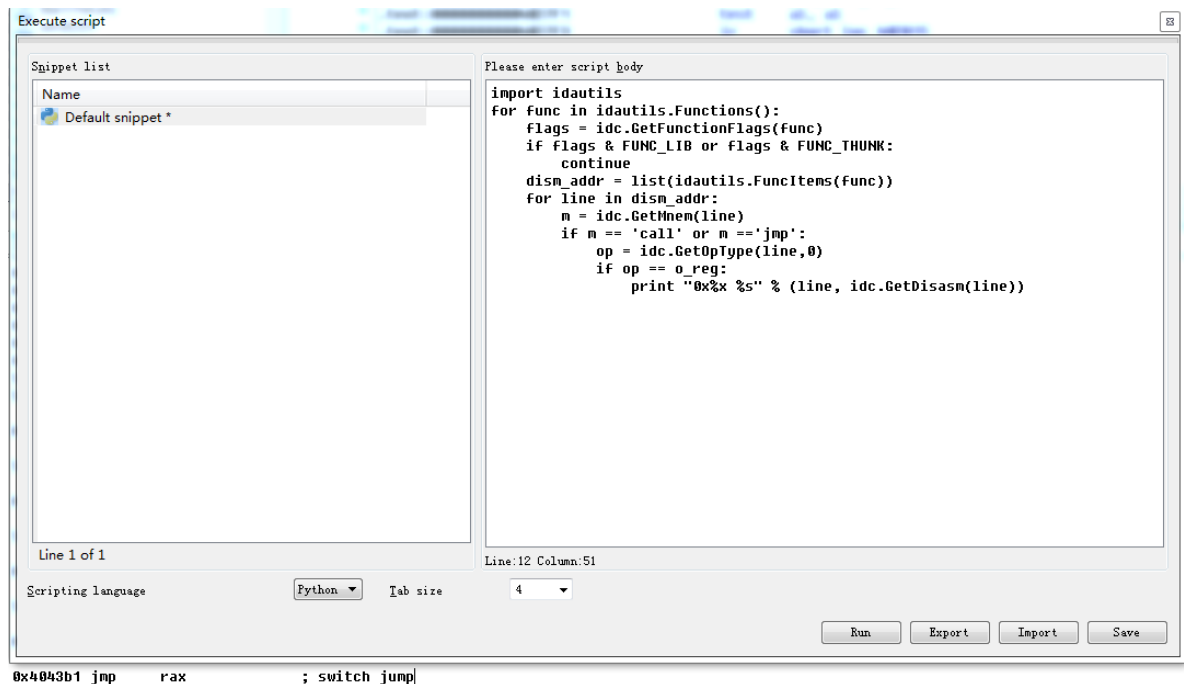
import idautils
ea = idc.ScreenEA()
print ea
dism_addr = list(idautils.FuncItems(ea))
print type(dism_addr)
print dism_addr
for line in dism_addr:
    print hex(line), idc.GetDisasm(line)

```

```
<type 'list'>
[4204483L, 4204484L, 4204487L, 4204494L, 4204501L, 4204508L, 4204515L, 4204522L, 4204529L, 4204531L, 4204533L, 4204537L, 4204541L, 4204545L, 4204549L,
4204553L, 4204557L, 4204561L, 4204565L, 4204572L, 4204582L, 4204592L, 4204596L, 4204603L, 4204610L, 4204617L, 4204624L, 4204631L, 4204638L, 4204641L,
4204644L, 4204649L, 4204650L]
0x4027c3L push    rbp
0x4027c4L mov     rbp, rsp
0x4027c7L sub     rsp, 0E0h
0x4027ceL mov     [rbp+out], rdi
0x4027d5L mov     [rbp+var_80], rdx
0x4027dcL mov     [rbp+var_98], rcx
0x4027e3L mov     [rbp+var_90], r8
0x4027eaL mov     [rbp+var_88], r9
0x4027f1L test    al, al
0x4027f3L jz      short loc_402815
0x4027f5L movaps [rbp+var_80], xmm0
0x4027f9L movaps [rbp+var_70], xmm1
0x4027fdL movaps [rbp+var_60], xmm2
0x402801L movaps [rbp+var_50], xmm3
0x402805L movaps [rbp+var_40], xmm4
0x402809L movaps [rbp+var_30], xmm5
0x40280dL movaps [rbp+var_20], xmm6
0x402811L movaps [rbp+var_10], xmm7
```

现在已经知道如何循环迭代节，函数，以及指令，接下来看一个用于搜寻被逆向软件中动态调用的脚本，所谓的动态调用即通过 call 或 jmp 直接跳转到寄存器地址的调用。

```
import idutils
for func in idutils.Functions():
    flags = idc.GetFunctionFlags(func)
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idutils.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))
```



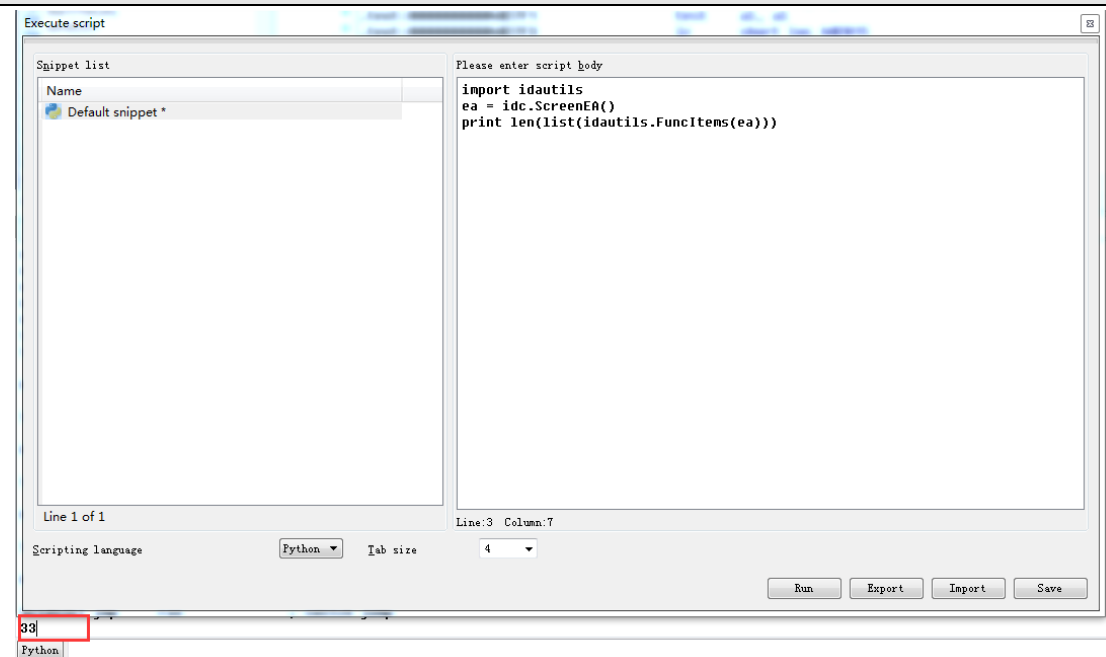
在该脚本中通过调用 idutils.Functions 获取已知的函数地址列表，对于每一个函数通过 idc.GetFunctionFlags() 获取其对应的 flags, 其中过滤掉 flag 为 FUNC\_LIB 和 FUNC\_THUNK 的函数类型，



通过 `idautils.FuncItems(ea)` 获取选中函数的所有指令地址，循环遍历该地址，通过 `idc.GetMnem(ea)` 获取所有操作码为 `call` 和 `jmp` 的指令地址。判断通过之后通过 `idc.GetOpType(ea, n)` 该地址上指令的第一个操作数。判断该操作数的类型，函数会返回一个整形变量，分别表示该地址是一个寄存器，内存还是引用，如果该操作数的类型是一个寄存器，则将该行指令打印出来。

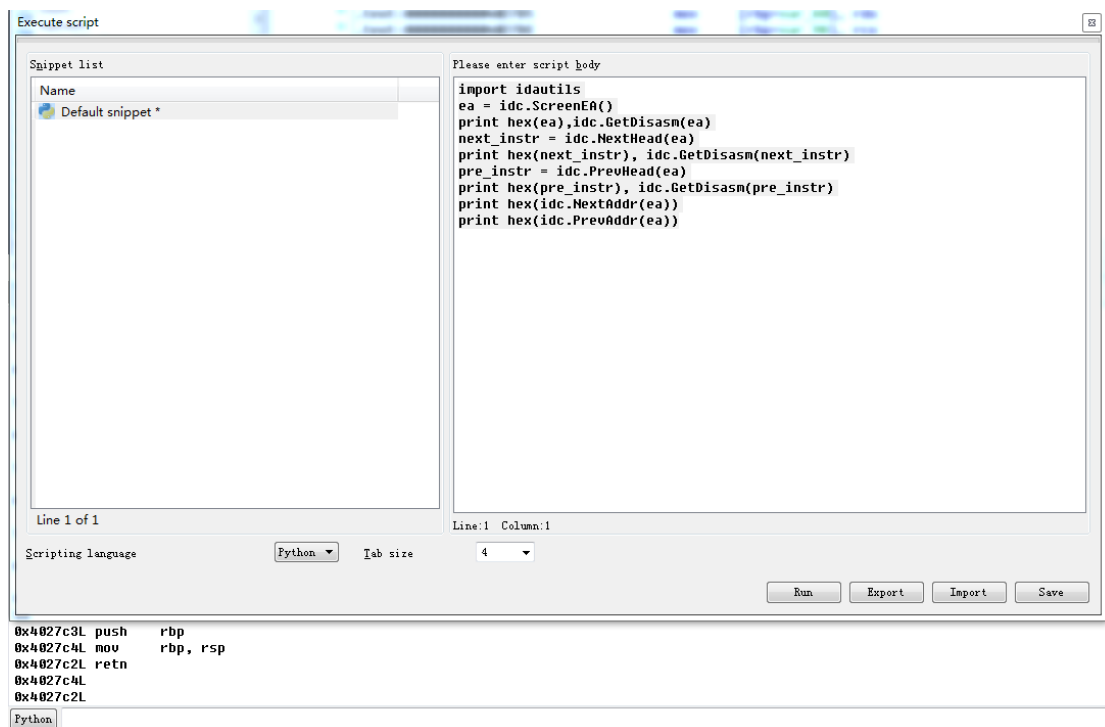
通过 `idautils.FuncItems(ea)` 返回的不是一个列表所以不能直接获取长度，可以通过将其放到 `list` 中来实现获取指令数的操作。如下所示：

```
import idautils
ea = idc.ScreenEA()
print len(list(idautils.FuncItems(ea)))
```



当我们需要获取某一个确切的地址附近的指令情况时，可以使用之前所过的 `idc.NextHead(ea)`, `idc.PreHead(ea)`, 但是这组函数只是获取对应的指令而不是其地址，想要获取地址可以使用以下这种函数

```
idc.NextAddr(ea), idc.PreAddr(ea)。
import idautils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
next_instr = idc.NextHead(ea)
print hex(next_instr), idc.GetDisasm(next_instr)
pre_instr = idc.PreHead(ea)
print hex(pre_instr), idc.GetDisasm(pre_instr)
print hex(idc.NextAddr(ea))
print hex(idc.PreAddr(ea))
```



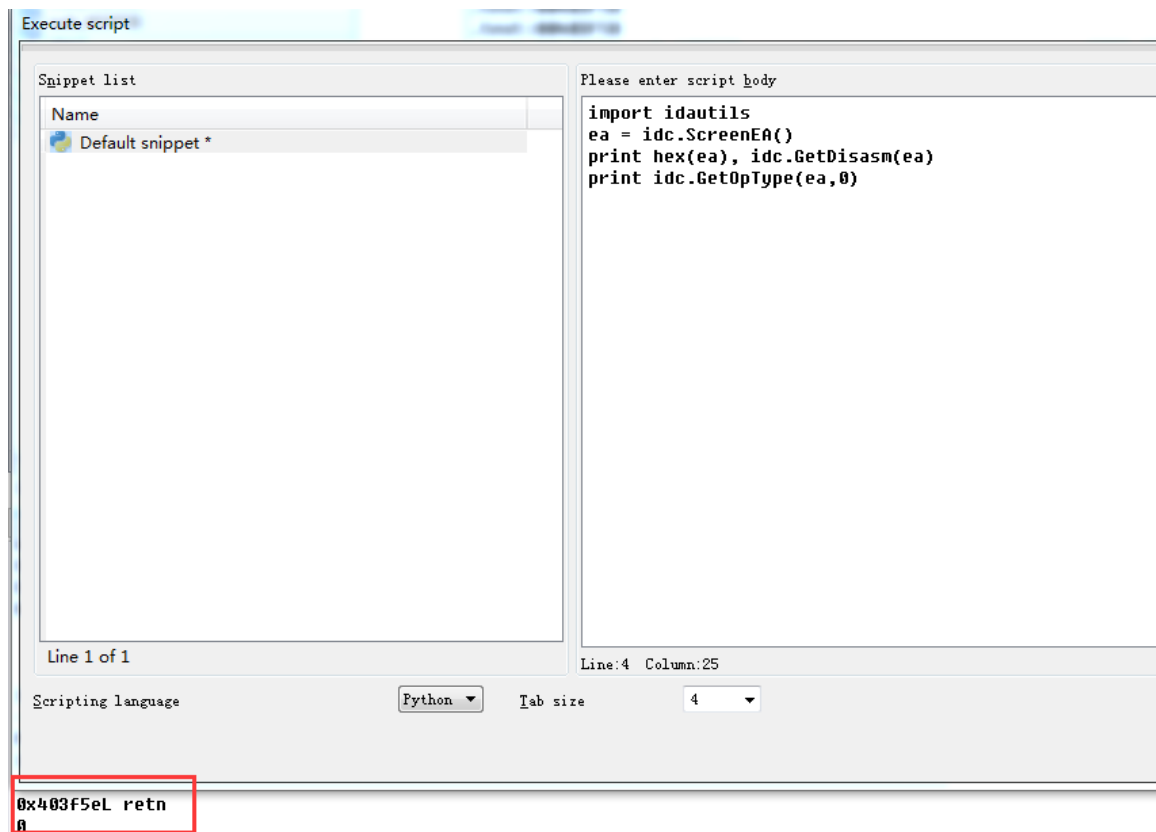
## 运算对象

运算对象(操作数)会被经常使用,所以获取运算对象的类型就显得至关重要,可以通过 `idc.GetOpType(ea, n)` 获取操作数的类型,其中 `ea` 为对应指令的地址, `n` 为操作数的索引,由 0 开始,在 `idapython` 中操作数的类型一共有 8 种。

## `o_void`

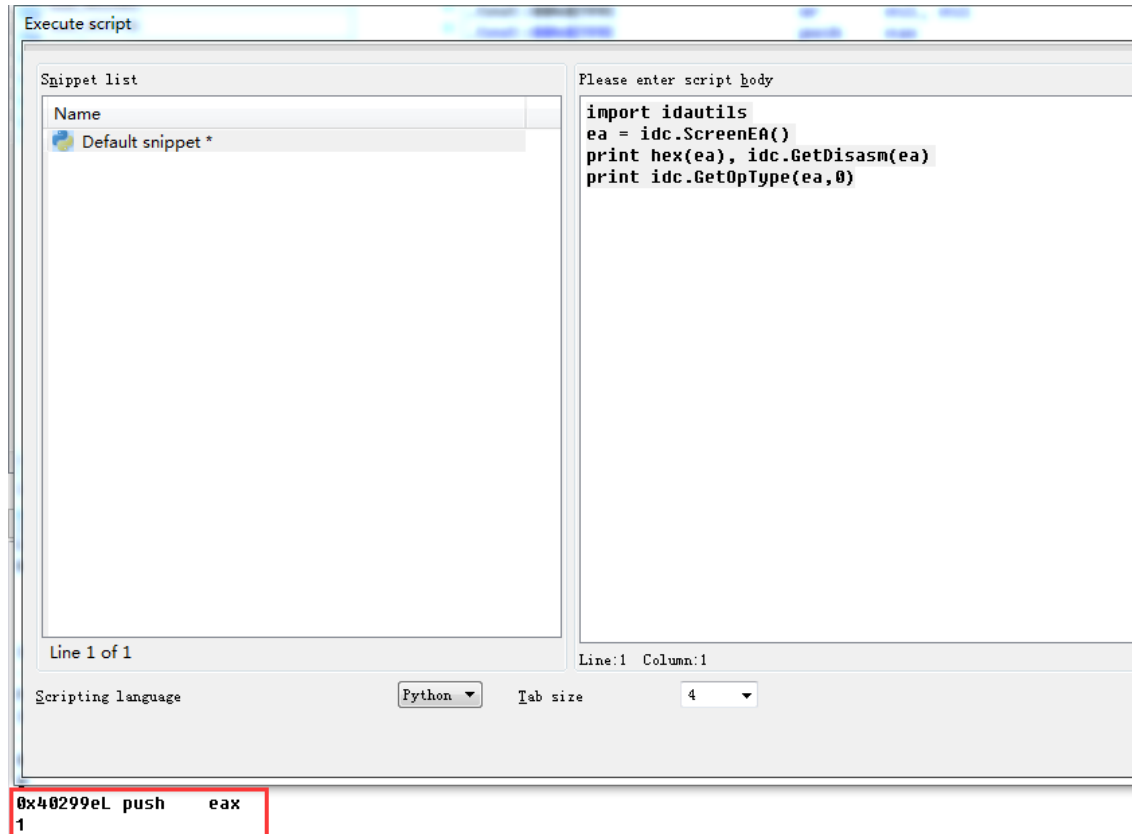
当一个指令没有任何操作数是返回 0.

```
import idautils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.GetOpType(ea, 0)
```



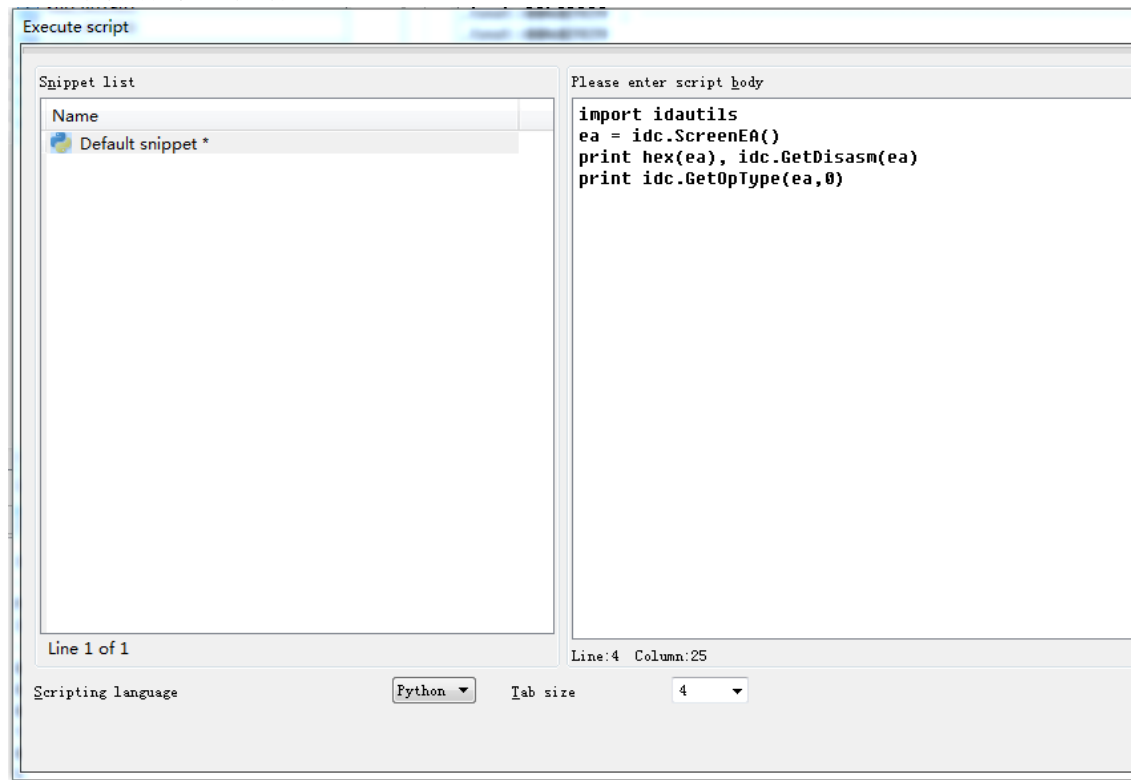
o\_reg

当一个操作数为寄存器时，返回 1.



o\_mem

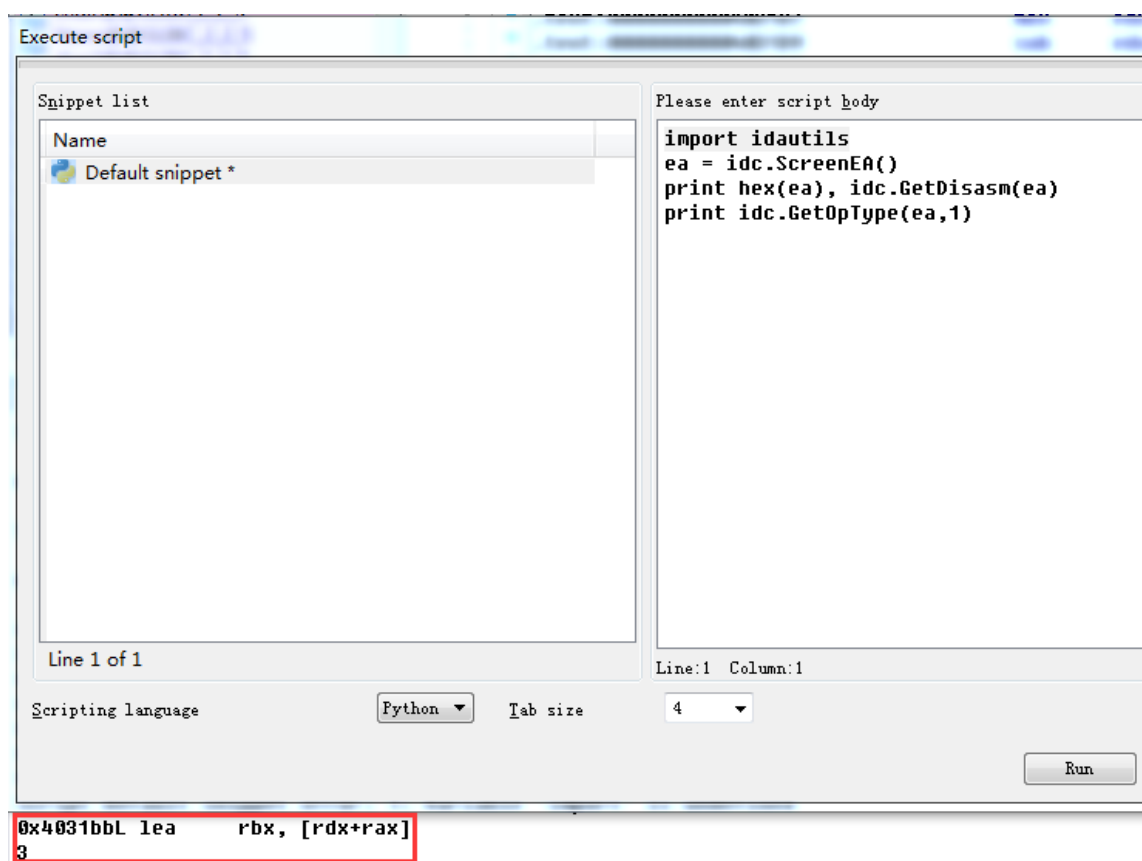
当一个操作数为内存引用时，返回 2



0x402913L call ds:GetTickCount  
2

o\_phrase

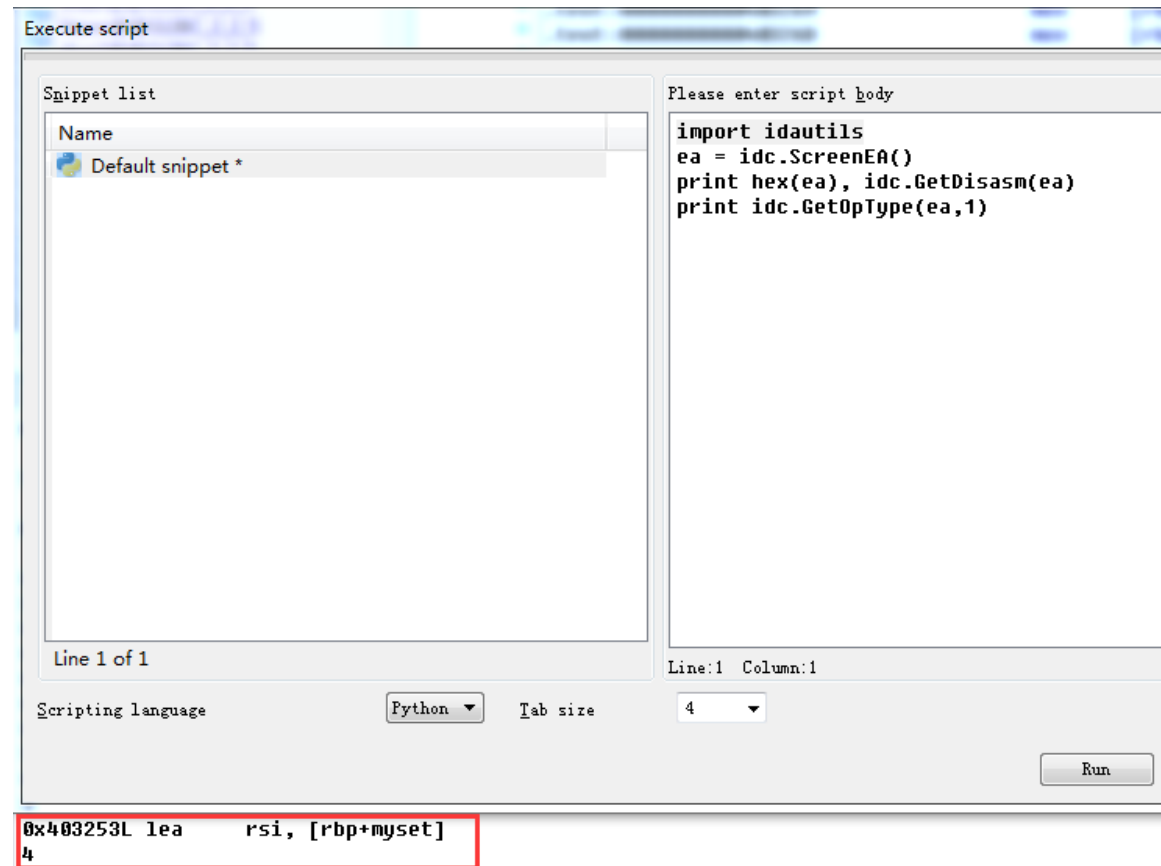
当一个操作数为基址寄存器加间址寄存器时，返回 3.



0x4031bbL lea rbx, [rdx+rax]  
3

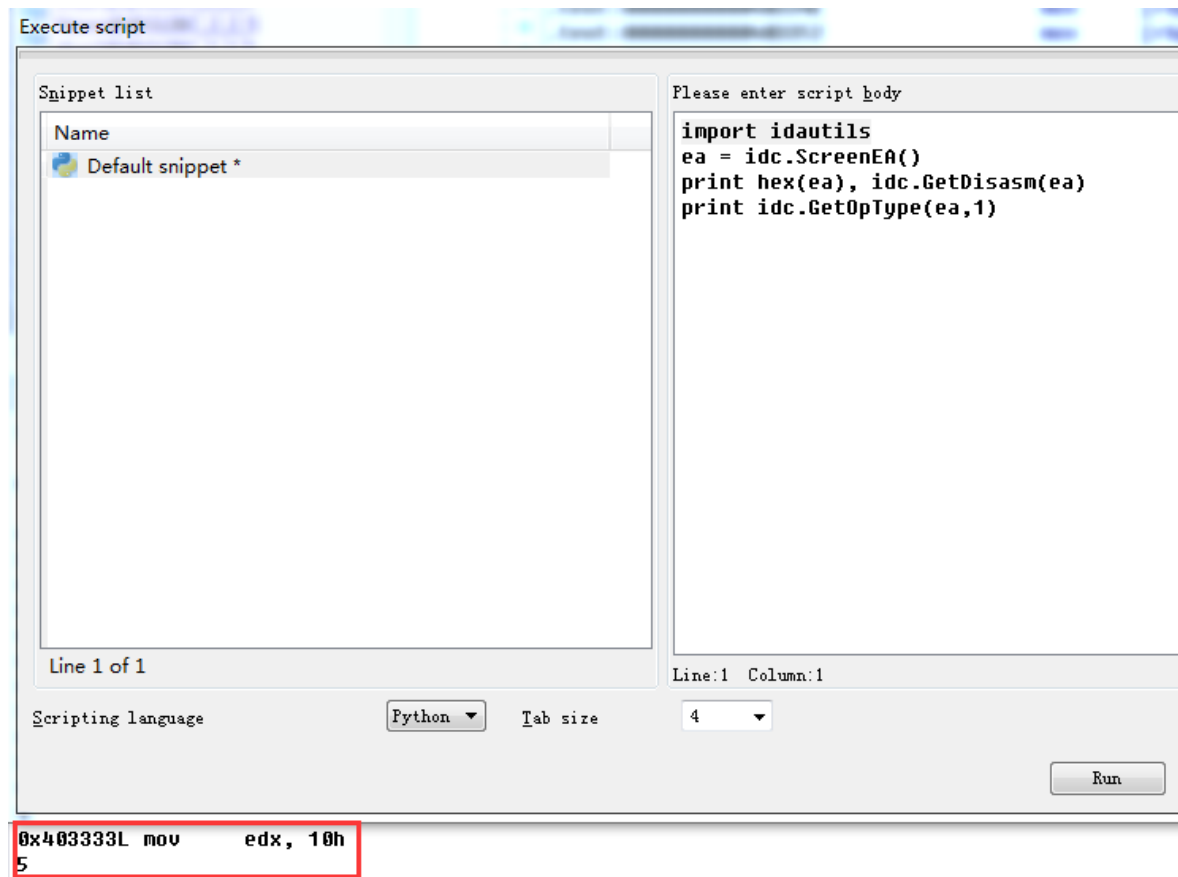
o\_displ

当操作数为一个寄存器加数字偏移时，返回 4。



o\_imm

当操作数为数字变量时，返回 5。

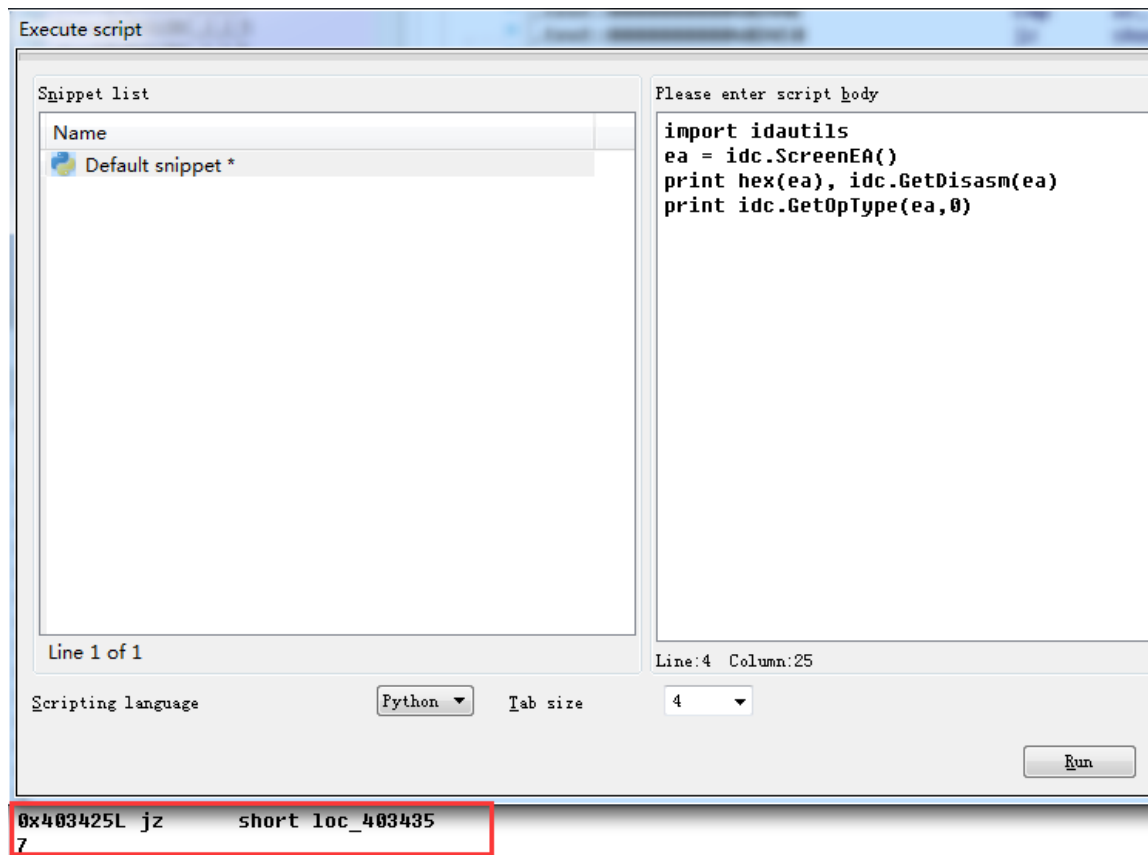


o\_far

少见的跳转指令之后的地址，短返回 6

o\_near

常见的跳转指令之后的地址，短返回 7



### 实例 1

```
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 "b
p" 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)

```

脚本的一开始不是不很熟悉，通过 `idautils.Functions()` 和 `GetFunctionFlags(ea)` 的组合获取 idb 中的说有函数类型并过滤掉库函数类型，之后通过函数 `idautils.FuncItems(ea)` 获取符合条件的函数中的所有指令，在这有一个新函数 `idaapi.decode_insn(ea)`，该函数用于解码传入地址的指令，一旦解码，我们就可以通过 `idaapi.cmd` 获取指令的各个属性参数。

```

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']

```

如上图所示，`idaapi.cmd` 有多个属性，现在回到我们的实例，通过 `idaapi.cmd.Op1.type` 可以获取指令的操作码，注意操作码的标号为 1 而不是 0，这点需要和 `idc.GetOpType(ea, n)` 函数的使用区分开来，之后通过 `o_displ` 判断该条指令的操作数为 1 还是 2，我们可以使用 `idaapi.tag_remove(idaapi.ua_outop2(ea, n))` 获取 string 类型的操作数，对比之后我们发现使用函数 `idc.GetOpnd(ea, n)` 来进行一条指令的操作是如此简洁优雅，但是作为学习的话，实现的思路还是多多益善的，查看 idapython 中 `idc.GetOpnd(ea, n)` 的源码。



```
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)
```

再次回到我们的实例，获取操作数对应的字符，我们需要检测其是否包含“bp”，当你需要判断该操作数是否为寄存器 bp，ebp 或 rbp 时这是一个一劳永逸的方法，检测 bp 的话，还需要检测其是否具有偏移量，通过函数 `idaapi.cmd.Op1.addr` 来获取对应偏移的值，他的返回值是 `stirng`，获取之后将其转化为 `int` 类型，之后将其添加到我们名为 `displace` 的字典中，如果存在一个这样的偏移，可以通过以下循环获取。

```
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
```

如上图所示，0x130

\*\*\*\*\*

## 实例 2

对于汇编代码中某些直接使用内存地址的指令，可以通过脚本将其中的内存地址以 `offset` 的形式展现

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

其中地址 0BC10B8 地址上的字符没有被 ida 解析出来，每次查看的双击该地址，通过以下的脚本可以实现自动化的操作。

```
import idutils
import idaapi
min = MinEA()
max = MaxEA()
```

```

for func in idutils.Functions():
    flags = idc.GetFunctionFlags(func)
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idutils.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)

```

运行之后结果如下：

```

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

```

在该脚本中首先通过 MinEA 和 MaxEA 获取整个 idb 文件中的起始和结束地址，之后通过函数 `idutils.Functions()` 获取所有函数的列表，通过 `idc.GetFunctionFlags` 获取函数对应的函数类型列表，过滤掉其中 `FUNC_LIB`，`FUNC_THUNK` 类型的函数，获取符合条件函数的所有指令，通过 `idc.GetOpType(curr_addr, 0)`，`idc.GetOpType(curr_addr, 1)` 获取第一和第二个操作数的类型，如果类型为数字，通过 `idc.GetOperandValue(curr_addr, 0)` 和 `idc.GetOperandValue(curr_addr, 1)` 获取对应操作数，如果该数字在 `min` 和 `max` 之间，说明该处为一个内存地址，通过函数 `idc.OpOff(curr_addr, 0, 0)` 将其转为一个 `offset` 的形式。

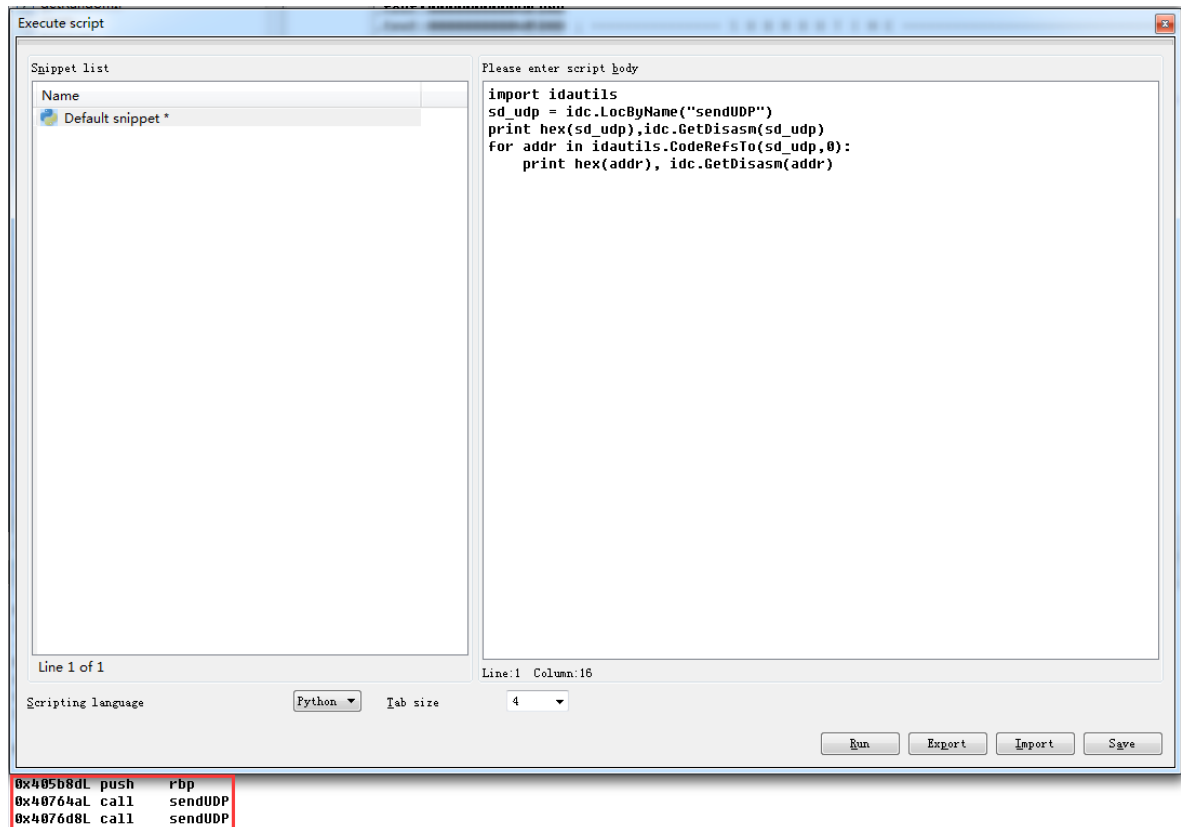
## 交叉引用

通常用于定位代码或数据的交叉引用非常重要，因为它提供了一处代码和数据的具体调用方是谁，比如我们想知道函数 `sendUDP` 是被那些地址调用，通过交叉引用即可实现改需求。

```

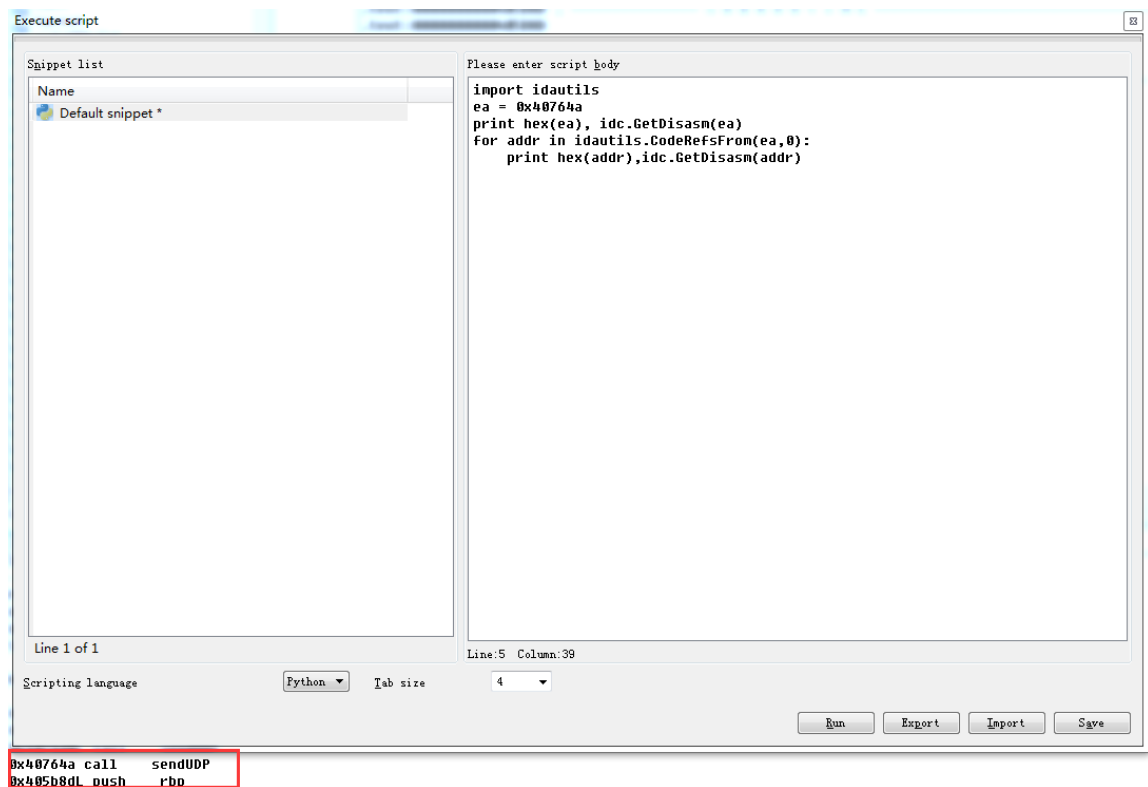
import idutils
sd_udp = idc.LocByName("sendUDP")
print hex(sd_udp), idc.GetDisasm(sd_udp)
for addr in idutils.CodeRefsTo(sd_udp, 0):
    print hex(addr), idc.GetDisasm(addr)

```



通过 `idc.LocByName(str)` 函数可以获取函数“sendUDP”的地址，`str` 为函数名称，返回地址为该 api 的地址。通过 `idautils.CodeRefsTo(ea, flow)` 返回所有调用了 sendUDP 的地址。该返回的结果可以通过循环进行迭代，`ea` 是用于查找相应交叉引用的地址，`flow` 是一个布尔值，用于标记是否监视一个正常的代码流。

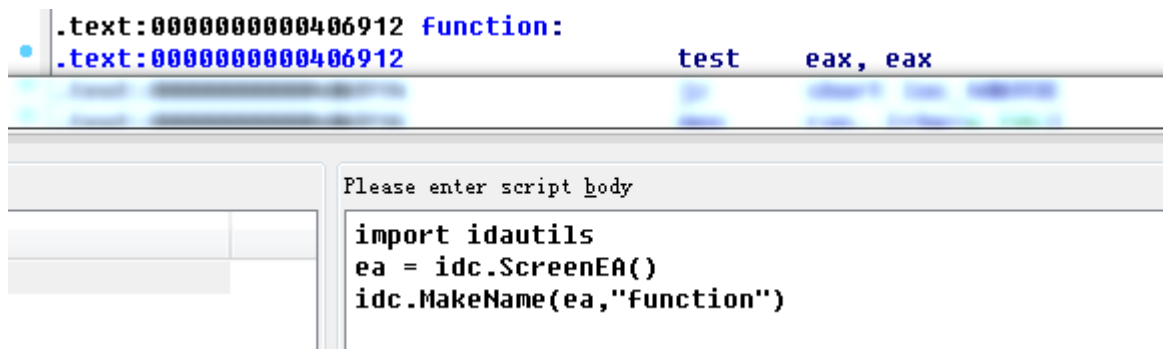
```
import idautils
ea = 0x40764a
print hex(ea), idc.GetDisasm(ea)
for addr in idautils.CodeRefsFrom(ea, 0):
    print hex(addr), idc.GetDisasm(addr)
```



可以通过 `idutils.CodeRefsFrom(ea, 0)` 获取 `ea` 地址调用的地方。如果想获取一个确切地址引用了哪些地址的数据可以使用函数 `idutilsl.CodeRefsFrom(ea, flow)`，以上脚本就展示了 `0x40764a` 这个地址引用了哪个地址。

`idutils.CodeRefsTo(ea, flow)` 的一个缺陷是对应动态导入的 `api` 无法找到交叉引用，手工直接修改的函数名又无法识别，这时可以使用函数 `idc.MakeName(ea, name)`。

```
import idutils
ea = idc.ScreenEA()
idc.MakeName(ea,"function")
```

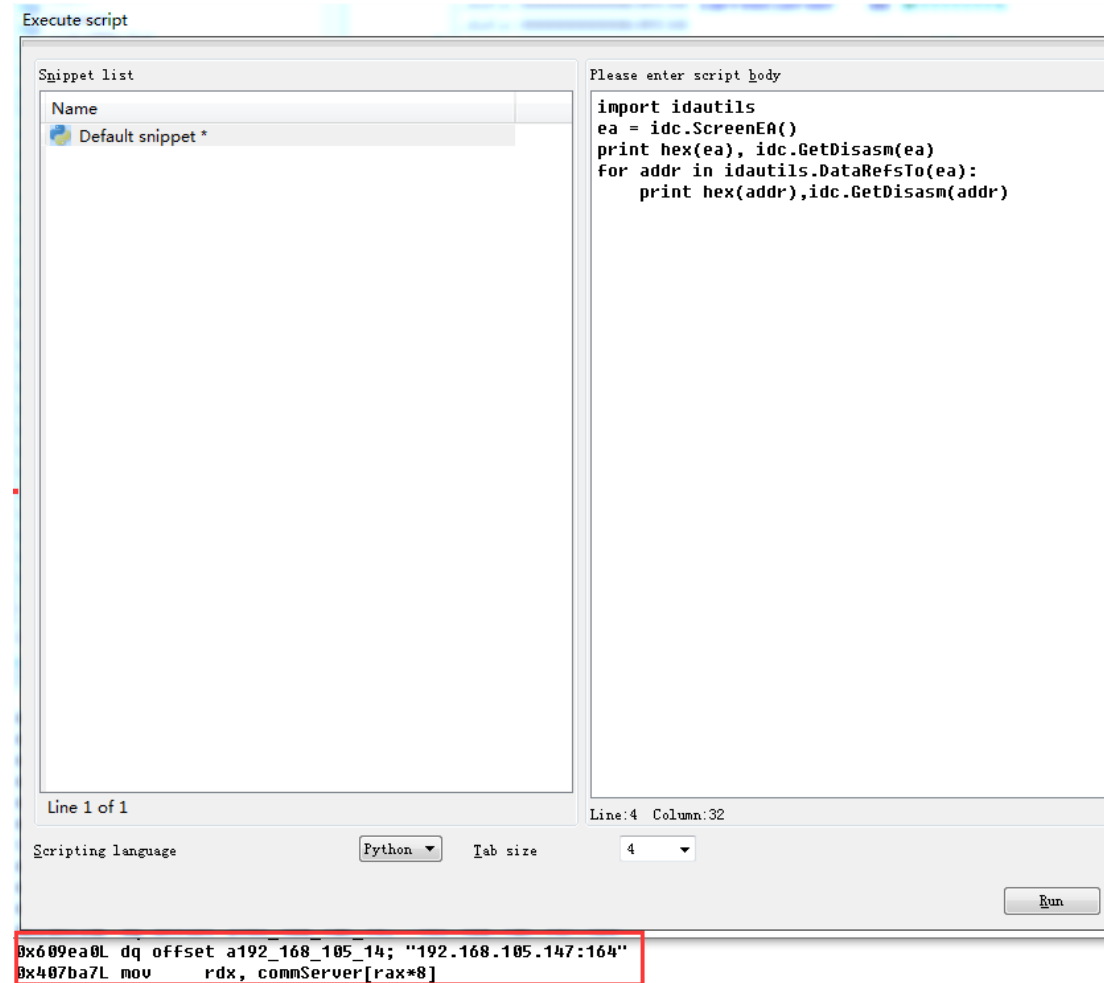


通过函数 `idc.MakeName(ea, "function")` 给地址加一个标签，此时就可以被交叉引用识别了。

想要找到一个数据的相关交叉引用可以使用函数 `idautils.DataRefsTo(e)` 和 `idautils.DataRefsFrom(ea)`，如下所示 `idautils.DataRefsTo(ea)` 用于获取所有引用了 `ea` 地址的调用。

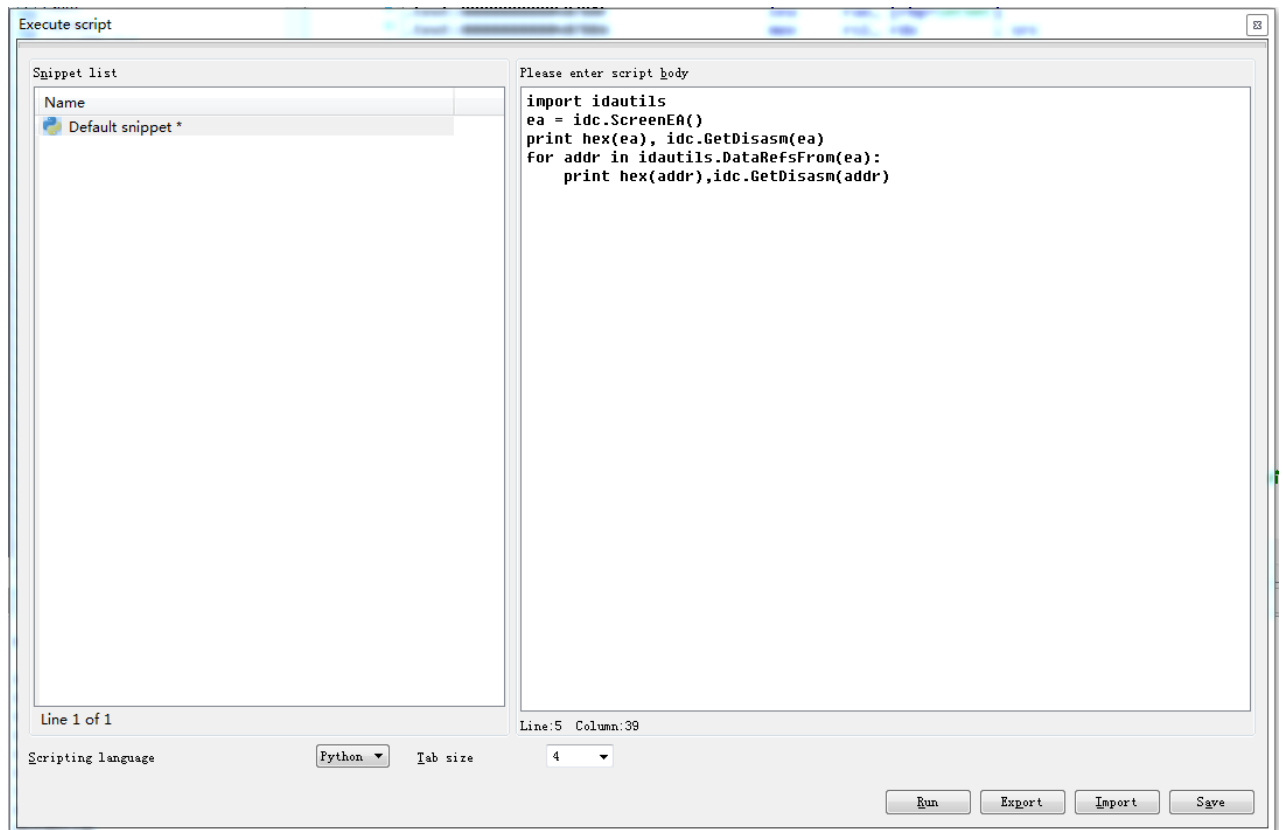
```
import idautils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
for addr in idautils.DataRefsTo(ea):
    print hex(addr), idc.GetDisasm(addr)
```

`idautils.DataRefsTo(ea)` 获取使用到该变量的所有地址



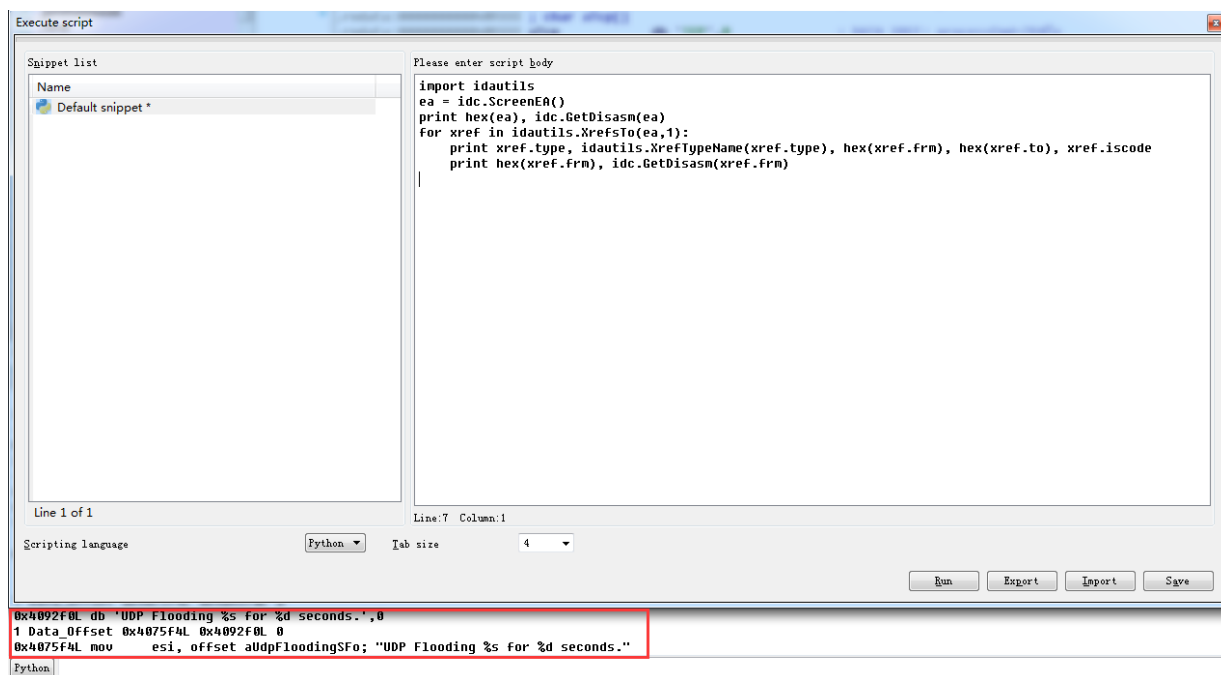
```
import idautils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
for addr in idautils.DataRefsFrom(ea):
    print hex(addr), idc.GetDisasm(addr)
```

获取该地址使用到的变量使用 `idautils.DataRefsFrom(ea)`

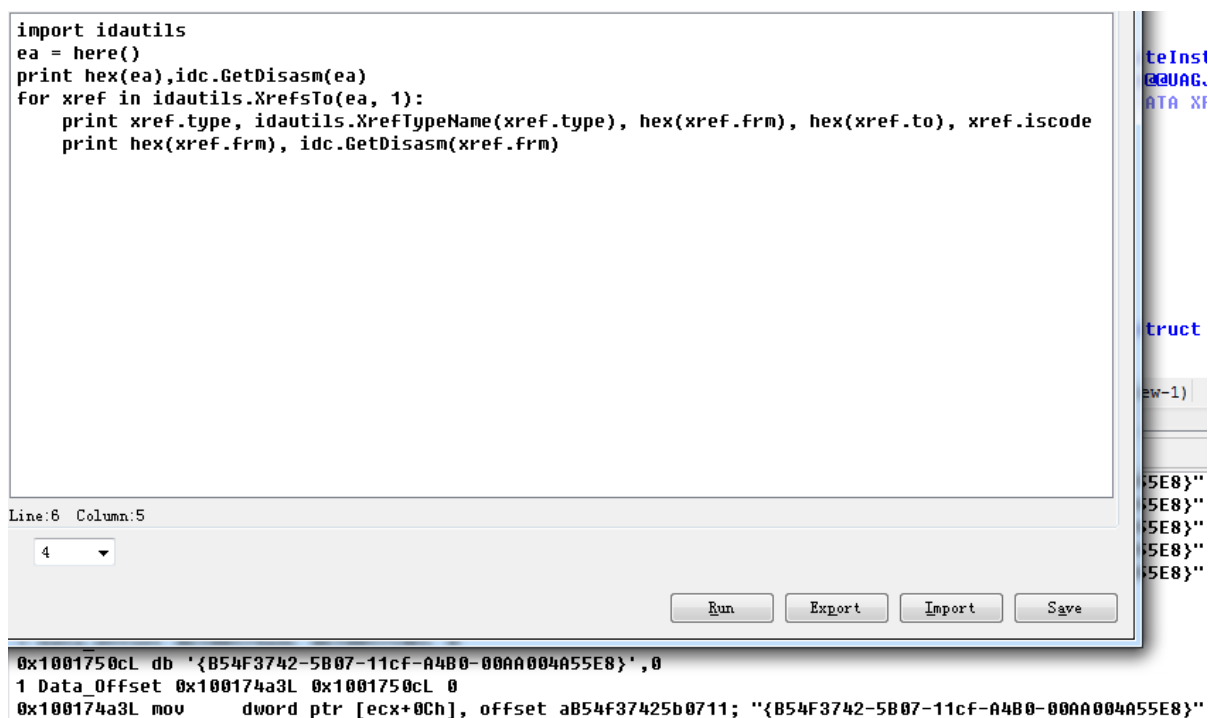


```
0x407ba7L mov     rdx, commServer[rax*8]
0x609ea0L dq offset a192 168 105 14; "192.168.105.147:164"
```

```
import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
for xref in idutils.XrefsTo(ea,1):
    print xref.type, idutils.XrefTypeName(xref.type), hex(xref.frm),
hex(xref.to), xref.iscode
    print hex(xref.frm), idc.GetDisasm(xref.frm)
```



可以通过 `idutils.XrefsTo(ea, flag=0)`, `idutils.XrefsFrom(ea, flags=0)` 获取对任意一个地址的所有交叉引用（无论是函数还是数据），其中的 `flags` 用于设置返回的类型，当为 0 时，会返回该地址的前后指令也默认为交叉引用，一般设置为 1，之后该函数会返回一个交叉引用的对象，该对象包含相关信息，如上所示. `type` 为 1，通过 `idutils.XrefTypeName` 返回 `type` 编号的实际类型名称，`.frm` 返回该地址的被引用的地址，`.to` 返回该地址的引用地址，`.iscode` 返回该地址是否为代码引用，如下所示。

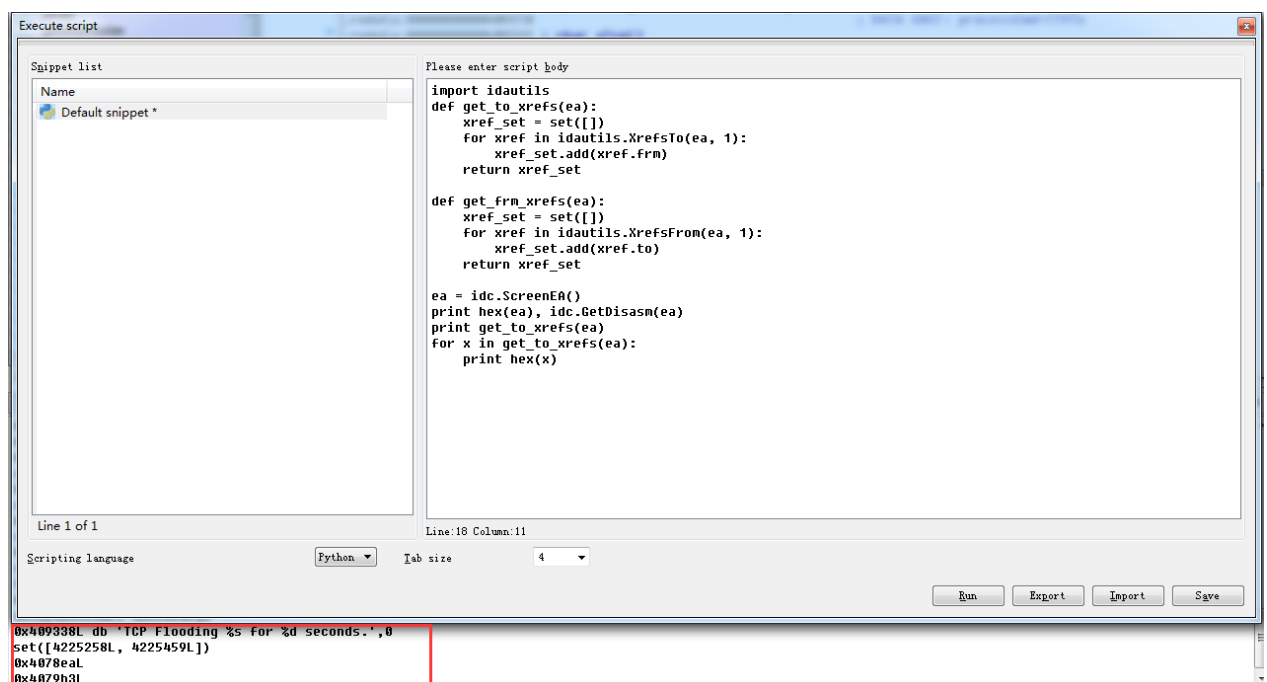


通过 `idutils.XrefTypeName(xref.type)` 可以获取交叉引用的类型，在 `idapython` 中有以下几类交叉引用，如下所示：

```

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'

```



```

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

ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print get_to_xrefs(ea)

```



```
for x in get_to_xrefs(ea):  
    print hex(x)
```

该实例中定义了两个函数用于返回指定的地址的交叉引用。

AddCodeXref(easource, eades, type) 添加交叉引用

easource 为增加交叉引用的源指令地址

eades 为交叉引用的实际目标地址

type: fl\_JF                      普通的 call 指令类型

      fl\_CF                      普通的跳转指令类型

## 搜寻

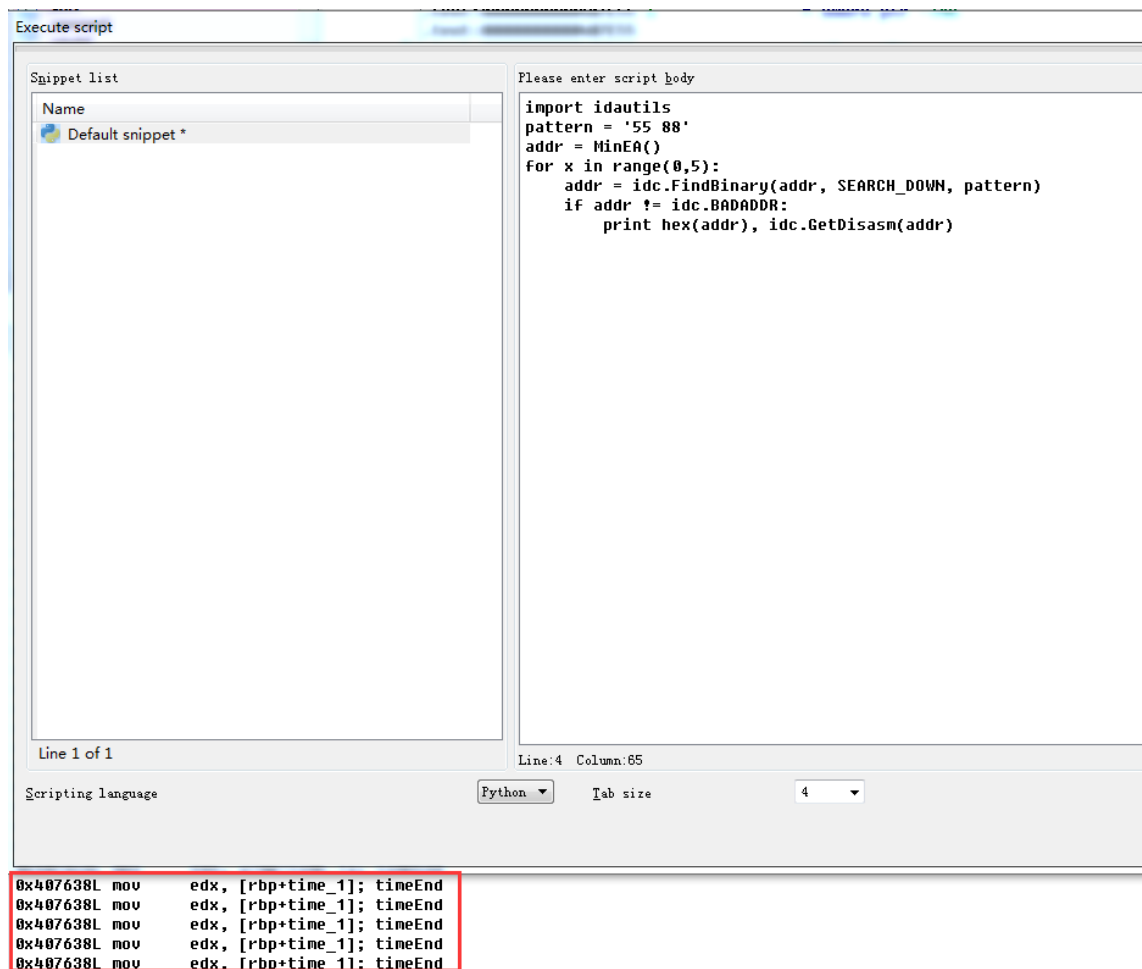
前面我们已经知道如何迭代函数和指令搜寻指定的目标, 这很有用但是有时候我们需要更精确的搜寻比如: 针对 0x55 0x8b 0xEC, 这一 16 进制比特其实就是典型的 push ebp, mov ebp, esp 栈处理过程。

通过函数 idc.FindBinary(ea, flag, searchstr, radix=16) 可以实现这一功能。其中 ea 用于标记搜索的起始地址。flag 用于标记搜索方向及条件, 以下注释的几个为比较常用的 flag

SEARCH_UP	= 0	#方向向上
SEARCH_DOWN	= 1	#方向向下
SEARCH_NEXT	= 2	#获取下一个代码对象
SEARCH_CASE	= 4	
SEARCH_REGEX	= 8	
SEARCH_NOBRK	= 16	
SEARCH_NOSHOW	= 32	
SEARCH_UNICODE	= 64	**
SEARCH_IDENT	= 128	**
SEARCH_BRK	= 256	**

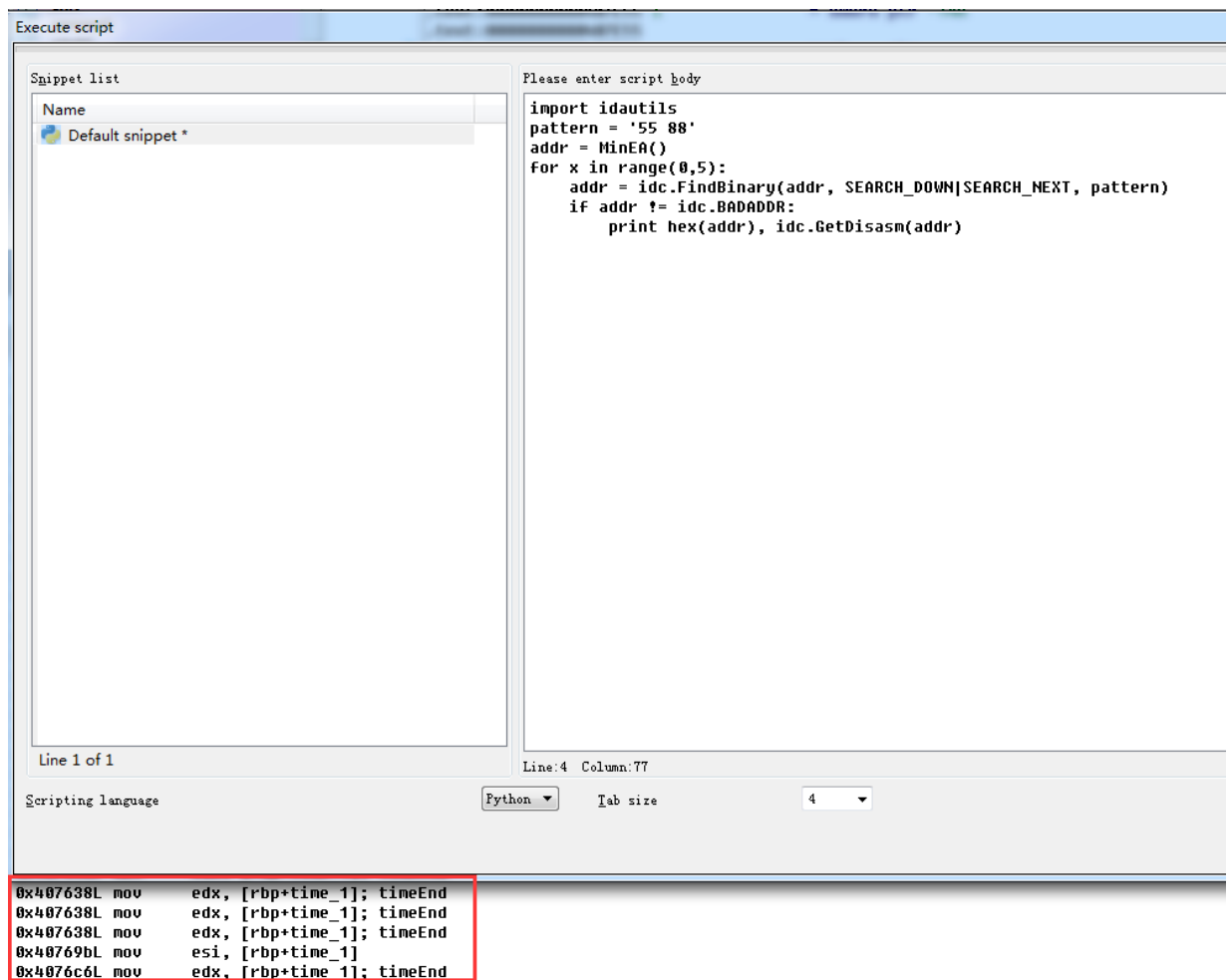
searchstr 是用于搜寻的目标。

radix\*\*\*\*可以暂时忽略。

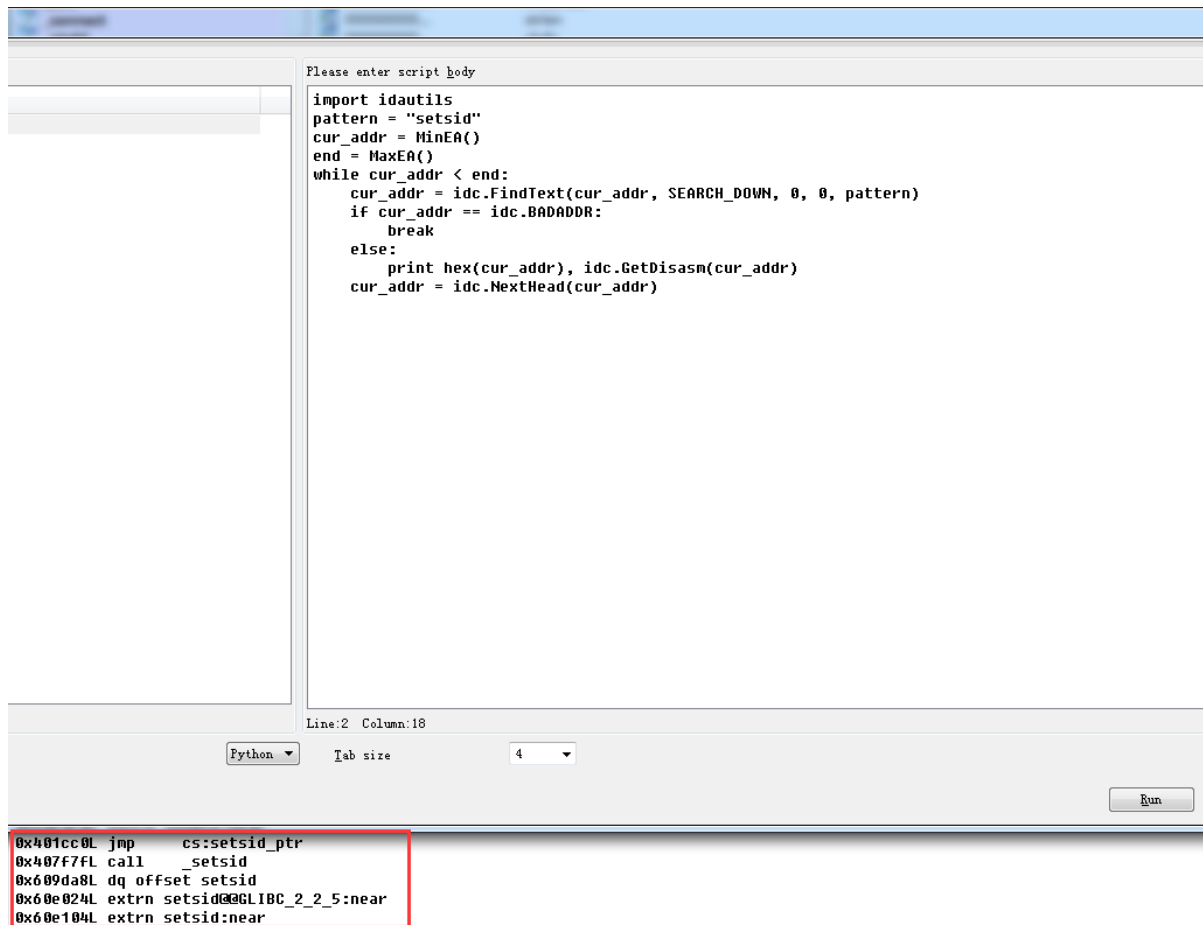


```
import idautils
pattern = '55 88'
addr = MinEA()
for x in range(0,5):
    addr = idc.FindBinary(addr, SEARCH_DOWN, pattern)
if addr != idc.BADADDR:
    print hex(addr), idc.GetDisasm(addr)
```

该脚本中第一行定义了查询的目标，16 进制的情况下可以写成 0x55 0x88 0xEC，或者直接写成 55 88，之后通过 MinEA() 函数获取执行体中的第一个地址，之后将搜寻到的结果赋值给变量 addr，通过之前说过的 idc.BADADDE 来校验返回的是不是一个合法的地址，如果合法就输出该地址以及对应的反汇编指令。注意此处我们返回的结果都是一个地址，那是因为 flags 中没有加上让该函数向下寻址的条件，下面是修改过之后的脚本执行情况。



有时候需要搜寻 ascii 或 unicode 形式的字符串，可以使用函数 `FindText(ea, flag, y, x, searchstr)`，是不是和之前的 `FindBinary` 很类似？  
`ea` 为搜寻的起始地址。  
`flag` 为搜寻的条件  
`y, x` 一般设置为 0  
`searchstr` 为搜寻的目标。  
以下脚本通过该函数实现对“setsid”字符的一个搜索功能。



```
import idutils
pattern = "setuid"
cur_addr = MinEA()
end = MaxEA()
while cur_addr < end:
    cur_addr = idc.FindText(cur_addr, SEARCH_DOWN, 0, 0, pattern)
    if cur_addr == idc.BADADDR:
        break
    else:
        print hex(cur_addr), idc.GetDisasm(cur_addr)
    cur_addr = idc.NextHead(cur_addr)
```

该脚本中我们用于查询字符“setuid”，cur\_addr 标记当前地址，end 标记搜寻 idb 的结束地址，通过函数 FindText 进行搜寻，注意此处的 flag 中没有使用 SEARCH\_NEXT 选项，这是因为在搜寻的循环中我们使用了 cur\_addr = idc.NextHead(cur\_addr) 将每次操作的当前地址更新为下一行地址。当然 ida 中还提供了其他用于地址搜寻的 find 方法，学习之前需要先学习另一组用于鉴定地址类型的方法，这组方法以 is 开头，返回的结果是布尔类型，如下所示：

idc.isCode(f)

如果该地址为代码则返回 true。

idc.isData(f)

如果该地址为数据则返回 true。

`idc.isUnknown(f)`

如果该地址 `ida` 无法鉴别其为数据还是代码时，返回 true

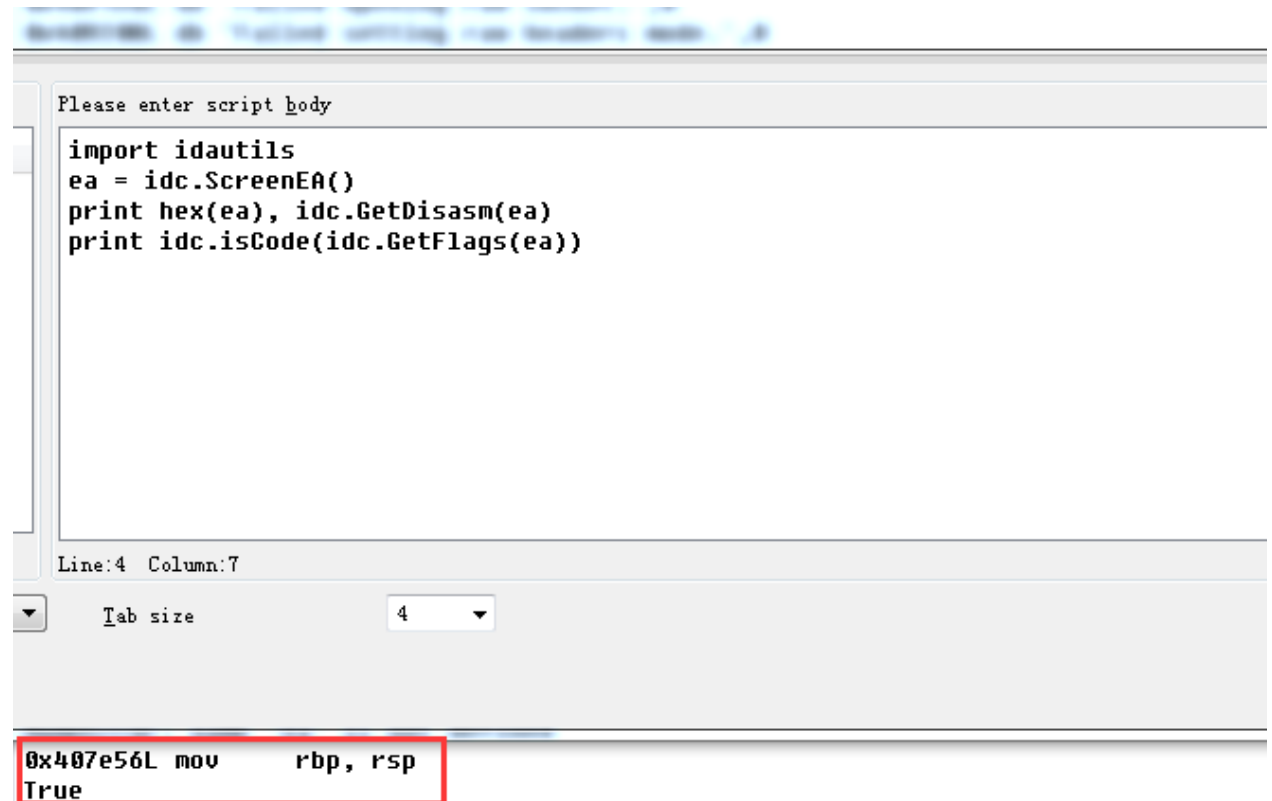
`idc.isHead(f)`

如果该地址为函数的卡头则返回 true

`idc.isTail(f)`

如果该地址为函数的结尾则返回 true

注意使用这一系列的函数是不能直接传递地址的，需要通过函数 `GetFlags(ea)` 对该地址进行一下转换，具体使用如下



The screenshot shows a text editor window titled "Please enter script body". Inside, there is a Python script that uses the `idautils` module. The script defines a function `ea = idc.ScreenEA()` and then prints the hexadecimal value of `ea`, the disassembly at that address, and whether it is code. The output of the script is shown at the bottom of the window, with the first line `0x407e56L mov rbp, rsp` and the second line `True` highlighted with a red box.

```
Please enter script body

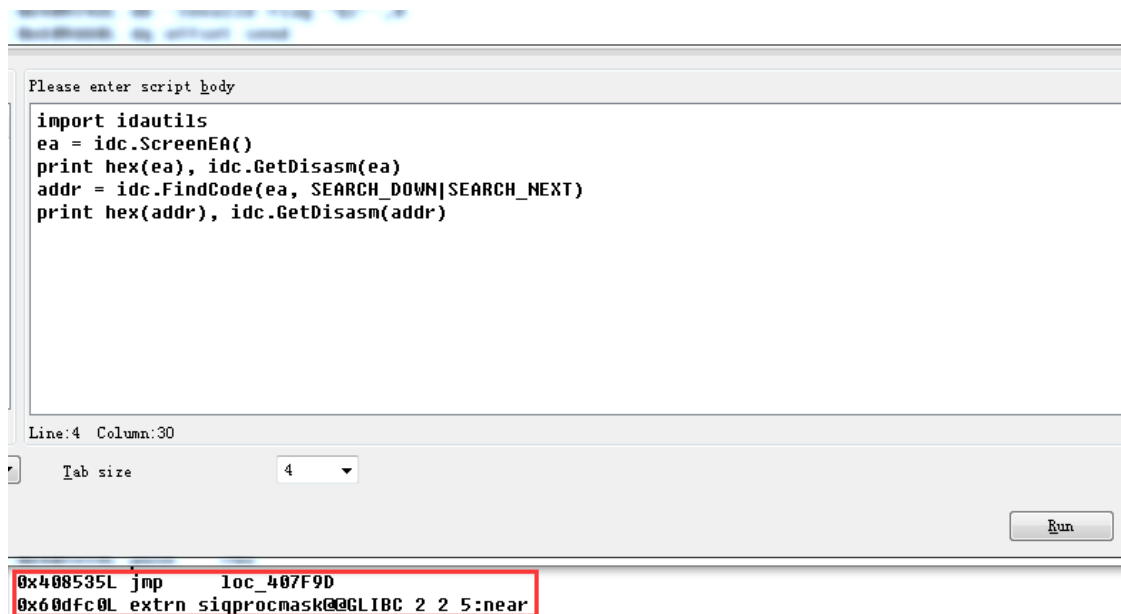
import idautils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.isCode(idc.GetFlags(ea))

Line:4 Column:7
Tab size 4

0x407e56L mov rbp, rsp
True
```

```
import idautils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.isCode(idc.GetFlags(ea))
```

通过函数 `idc.FindCode(ea, flag)` 用于获取下一个代码指令的地址，比如当我们需要获取某一块数据的结尾时，如果当前的 `ea` 是一个代码段的地址，其将会返回下一个代码指令的地址。



```
import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
addr = idc.FindCode(ea, SEARCH_DOWN|SEARCH_NEXT)
print hex(addr), idc.GetDisasm(addr)
```

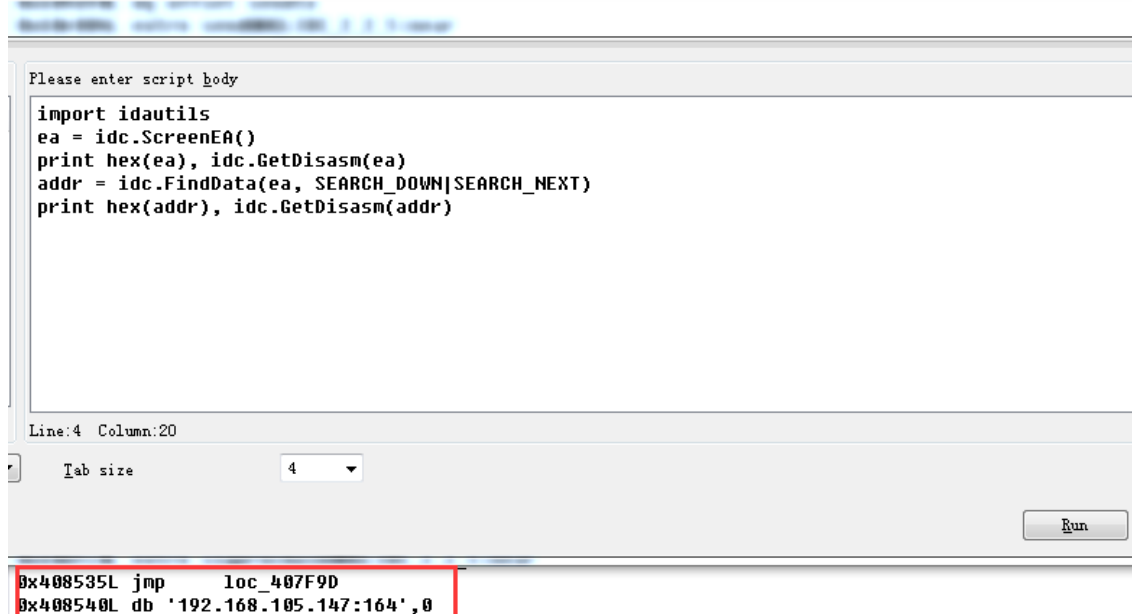
该脚本中 ea 地址之后实际是一段数据，通过函数 FindCode，可以实现一个跨区域的搜索。

idc.FindData(ea, flag)

同样类似的函数还有 idc.FindData(ea, flag)

只是该函数返回的是下一个数据块类型地址的起始。

在之前脚本中将 FindCode 换成该函数，效果如下：



```
import idutils
ea = idc.ScreenEA()
```

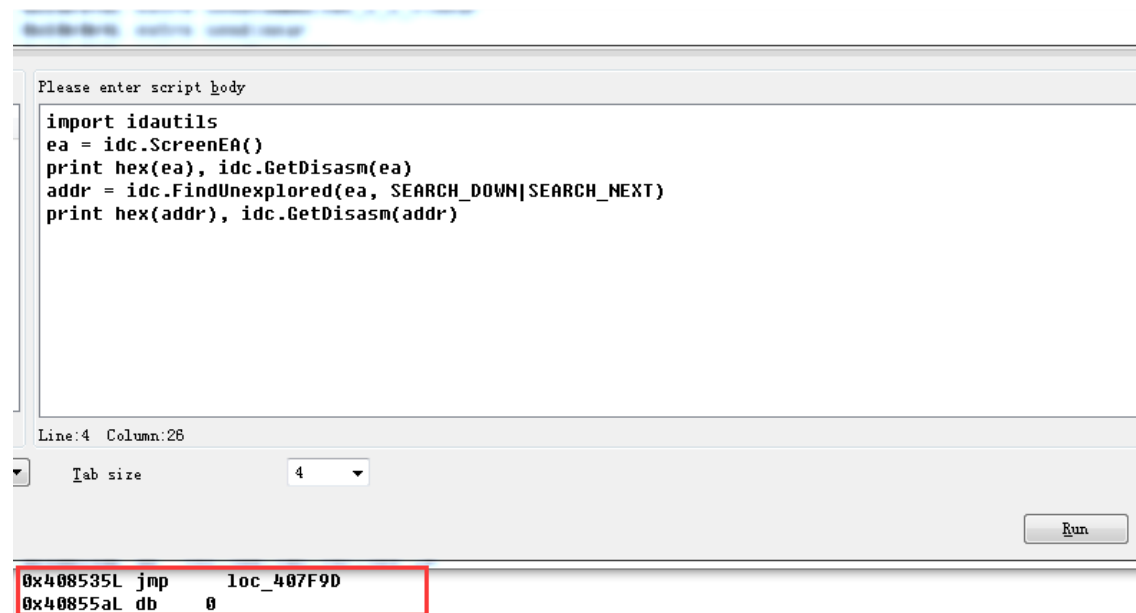
```
print hex(ea), idc.GetDisasm(ea)
addr = idc.FindData(ea, SEARCH_DOWN|SEARCH_NEXT)
print hex(addr), idc.GetDisasm(addr)
```

当然你也可以将其中的 SEARCH\_DOWN 替换成 SEARCH\_UP。

idc.FindUnexplored(ea, flag)

函数 idc.FindUnexplored(ea, flag)

该函数用于找到一个既不是代码也不是数据的地址，通过脚本就可以减少大量的类似手工操作。



```
import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
addr = idc.FindUnexplored(ea, SEARCH_DOWN|SEARCH_NEXT)
print hex(addr), idc.GetDisasm(addr)
```

idc.FindExplored(ea, flag)

函数 idc.FindExplored(ea, flag) 用于返回一个 ida 中判定为数据或代码的地址。

函数 idc.FindImmediate(ea, flag, value)

相比于按类型搜索，有时候我们更喜欢按某一数值进行搜索，比如一个 sock 函数，没有定位到具体的函数代码地址，但是我们知道该 sock 初始化是的端口为 1A0B，通过该函数我们就可以进行一次以该端口数值为目标的搜索。

```
Please enter script body

import idutils
addr = idc.FindImmediate(MinEA(), SEARCH_DOWN, 0x1A0B)
print addr
print "0x%x %s %x" % (addr[0],idc.GetDisasm(addr[0]),addr[1])

Line:4 Column:60
Tab size 4 Run

[4225985L, 1]
0x407bc1 mov [rbp+port], 1A0Bh 1
```

```
import idutils
addr = idc.FindImmediate(MinEA(), SEARCH_DOWN, 0x1A0B)
print addr
print "0x%x %s %x" % (addr[0],idc.GetDisasm(addr[0]),addr[1])
```

不同于之前的搜寻函数，函数 FindImmediate 函数返回的是一个结构数组，该数组的第一个元素为找到的地址，第二个元素为该地址下的的操作数。

```
Please enter script body

import idutils
addr = MinEA()
while True:
    addr, operand = idc.FindImmediate(addr,SEARCH_DOWN|SEARCH_NEXT,0x3A)
    if addr != BADADDR:
        print hex(addr), idc.GetDisasm(addr), "Operand", operand
    else:
        break

Line:4 Column:72
Python Tab size 4 Run

0x40228eL sub eax, 3Ah Operand 1
0x402ff8L cmp al, 3Ah Operand 1
0x407bcfL mov esi, 3Ah ; c Operand 1
0x407be8L mov esi, 3Ah ; c Operand 1
0x407c0bL mov esi, 3Ah ; c Operand 1
```

在该脚本中搜寻了所有带 3A 的地址，如上所示。

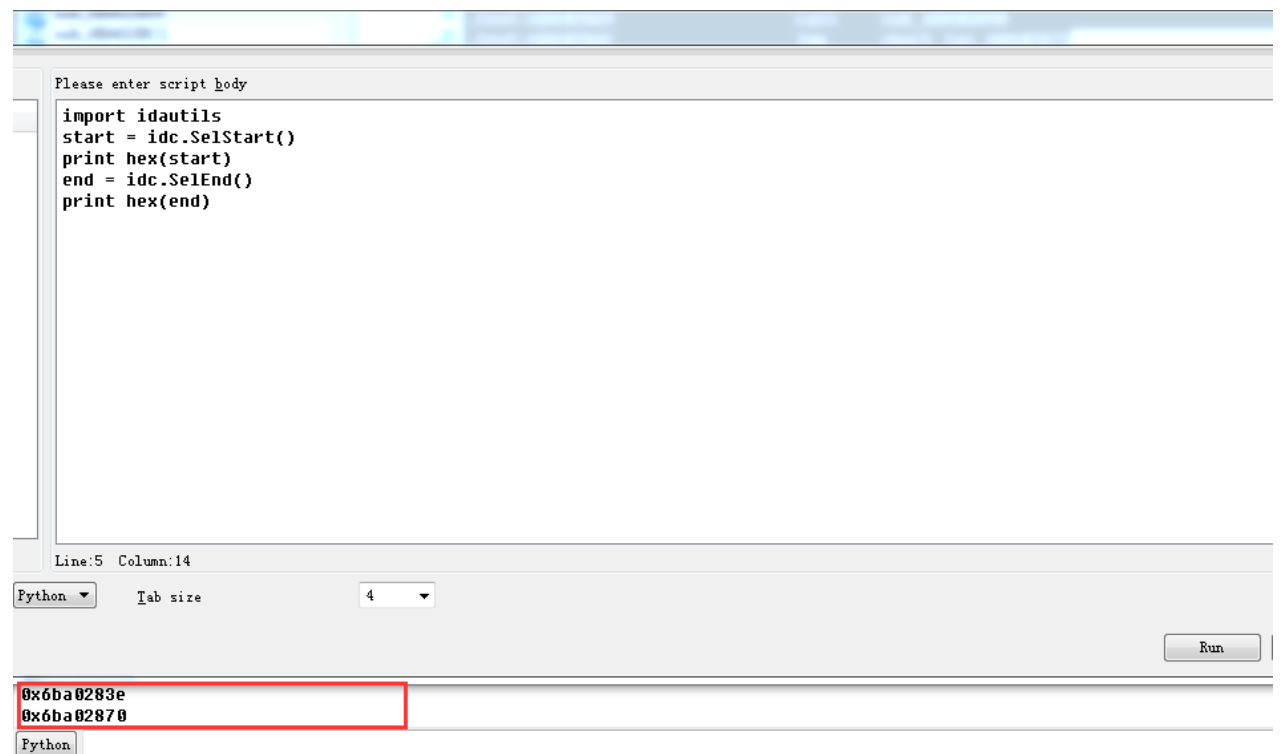


# 数据选取

并不是所有的时候我们都需要写一段代码来实现自动化的代码或数据的搜寻，在某些情况下我们已经知道了目标代码或数据的位置，我们仅仅需要一种数据选取的手段，获取指定区域的代码可以通过函数 `idc.SelStart()` 和 `idc.SelEnd()` 来获取，

```
.text:6BA0283E loc_6BA0283E: ; CODE XREF: sub_6BA02809+2B↑j
.text:6BA0283E      mov     ebx, [ebp+arg_8]
.text:6BA02841      push    ebx
.text:6BA02842      push    [ebp+arg_0]
.text:6BA02845      mov     [ebp+var_E4], esi
.text:6BA02848      call   sub_6BA0659E
.text:6BA02850      cmp     [ebp+arg_14], esi
.text:6BA02853      pop     ecx
.text:6BA02854      pop     ecx
.text:6BA02855      push    ebx
.text:6BA02856      push    [ebp+arg_0]
.text:6BA02859      lea     edi, [ebx+6FCh]
.text:6BA0285F      lea     eax, [ebp+var_184]
.text:6BA02865      push    edi
.text:6BA02866      push    eax
.text:6BA02867      jz      short loc_6BA02870
.text:6BA02869      call   sub_6BA03A9B
.text:6BA0286E      jmp     short loc_6BA02875
```

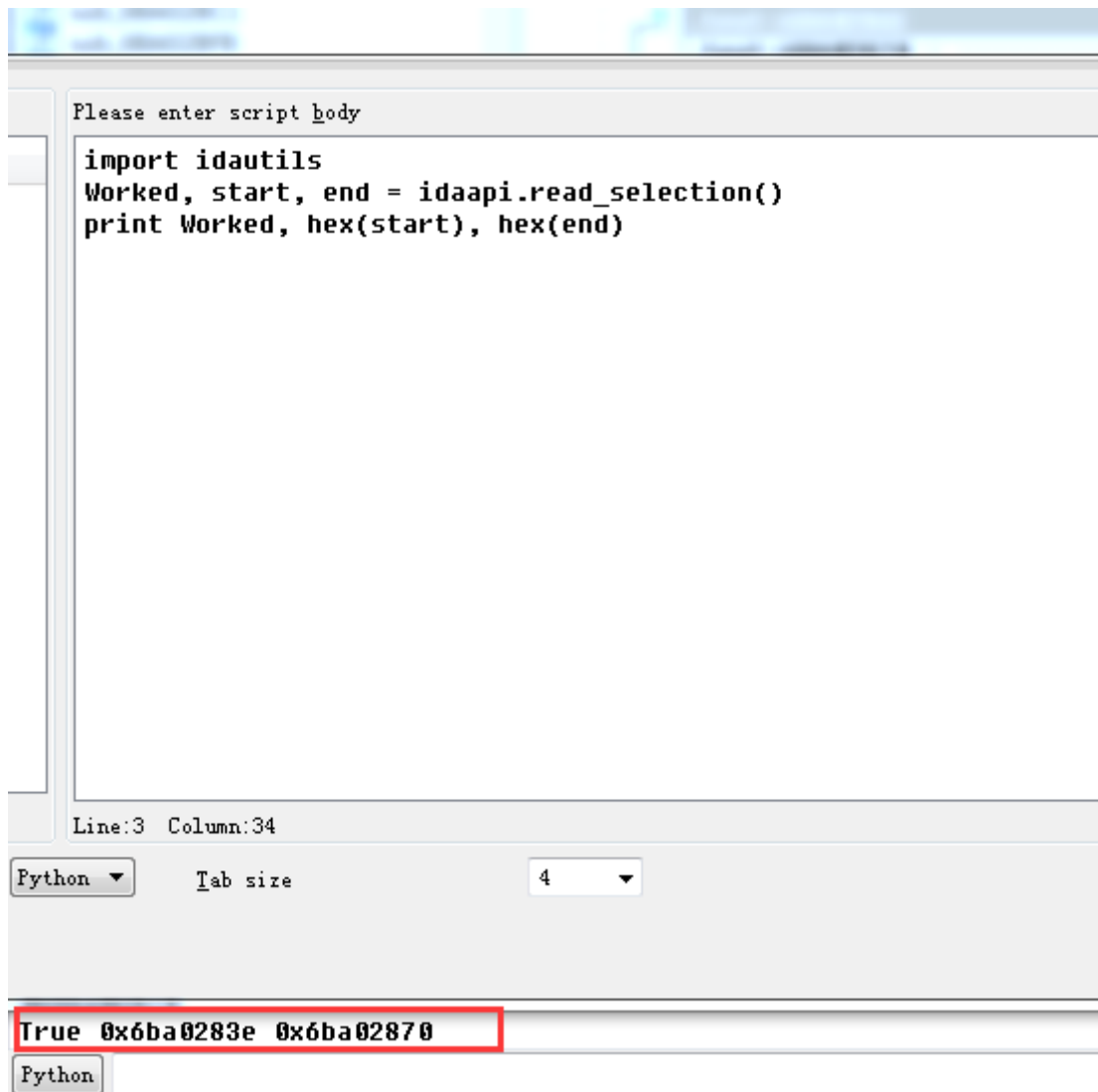
我们可以使用以下脚本来实现所选区域地址的获取。



```
import idutils
start = idc.SelStart()
print hex(start)
end = idc.SelEnd()
print hex(end)
```

在该脚本中 start 为所选区域的起始地址,但是 end 却并不是该区域的最后一条地址, idc.SelEnd() 返回的是该区域结束地址的下一条地址。

当然还有一个更专业的函数 idaapi.read\_selection(), 该函数返回一个 3 元组, 元组中的第一个参数为布尔值, 标记所选区域是否可读, 第二和第三个参数为所选区域的起始和结束地址。



```
import idutils
Worked, start, end = idaapi.read_selection()
print Worked, hex(start), hex(end)
```

同样一个区域使用 read\_selection 返回的结果。

注意: 在 64 位的分析样本中, 返回的地址有可能出错, 毕竟有可能发生整数溢出嘛 0(∩\_∩)0 哈哈~

## 注释&重命名

在逆向中注释有助于理解代码的结构和功能。

在开始实例脚本之前，我们首先要来聊聊一些注释和重命名到的常识，一般来说有两类注释，第一类是常规的注释，第二类是重复的注释，其中第一类注释如 0041136B 出所示，第二类注释如地址 00411372, 00411386, 00411392 所示，在这三个地址中只是最后一处进行了手动注释，当重复注释生效之后，其他所有引用到该地址的地方都会自动生成注释。

```

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

```

增加一个普通注释可以使用函数 `idc.MakeComm(ea, comment)`，重复的注释可以使用函数 `idc.MakeRptCmt(ea, comment)`，其中 `ea` 是注释那一行指令的地址，`comment` 是实际的注释内容。

下面的代码的功能：为 `idb` 中所有的 `xor` 置零指令注释。

```

loc_407D7D:                                     ; CODE XREF: sub_407C22+45↑j
                                                ; sub_407C22+5B↑j ...
xor     eax, eax                               ; eax = 0

```

Please enter script body

```

import idutils
for func in idutils.Functions():
    flags = idc.GetFunctionFlags(func)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idutils.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)

```

```

import idutils
for func in idutils.Functions():
    flags = idc.GetFunctionFlags(func)
    # skip library & thunk functions

```

```

if flags & FUNC_LIB or flags & FUNC_THUNK:
    continue
dism_addr = list(idutils.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)

```

在该脚本中首先通过 `idutils.Functions()` 循环遍历了所有的函数，之后通过 `list(idutils.FuncItems(func))` 将每一个函数对应的所有指令放到一个 `list` 列表中汇总，通过 `idc.GetMnem(ea)` 获取 `list` 中每一条指令的操作码，并判断该操作码是否为“xor”，之后通过函数 `idc.GetOpnd(ea, n)` 获取该条指令的两个操作数，并判断这两个操作数是否相等，如果相等则给改句指令增加一个普通注释。

```

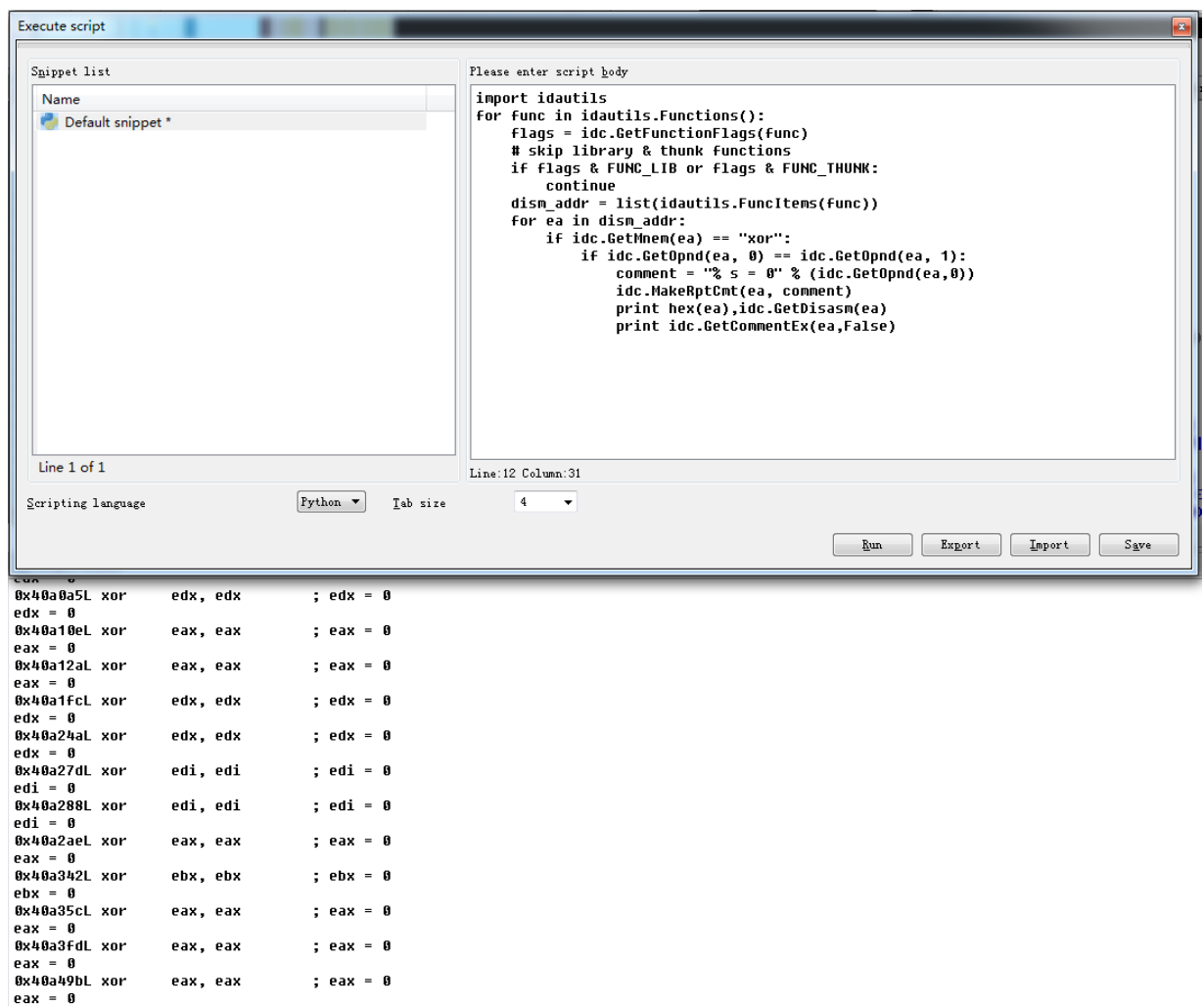
.text:00407D7D loc_407D7D:                                ; CODE XREF: sub_407C22+45↑j
.text:00407D7D                                         ; sub_407C22+5B↑j ...
.text:00407D7D      xor      eax, eax                        ; eax = 0

```

将函数 `idc.MakeComm(ea, comment)` 替换为 `MakeRptCmt(ea, comment)`，可以获得一条重复的注释。

`GetCommentEx(ea, repeatable)`

通过函数 `GetCommentEx(ea, repeatable)` 函数可以获取当前地址的注释内容，其中 `ea` 为指定的地址，`repeatable` 是一个布尔值，通过在脚本中补充一下两行代码可以获取该行对应的注释。



注意以上调用中，因为我们申明的注释为重复注释，所以在函数 `idc.GetCommentEx(ea, bool)` 中第二个参数使用了 `False`，如果需要获取的注释为普通注释，函数调用时的布尔值应该为 `true`，即 `idc.GetCommentEx(ea, True)`。

除了指令之外，函数同样可以进行注释。

`idc.SetFunctionCmt(ea, cmt, bool)` `idc.GeFunctionCmt(ea, bool)`

通过函数 `idc.SetFunctionCmt(ea, cmt, bool)` 进行一个函数注释。

`ea` 为函数体中任意一处指令的地址，`cmt` 为需要注释的内容，`bool` 为布尔值，其中 `flase` 表示重复注释，`true` 为普通注释。

通过函数 `idc.GeFunctionCmt(ea, bool)` 用于获取某一行指令所在函数对对应的注释。

`ea` 为改行的地址。

```

.text:00401070 ; check out later
.text:00401070 ; Attributes: bp-based frame
.text:00401070
.text:00401070 ; int __cdecl sub_401070(double)
.text:00401070 sub_401070      proc near                ; CODE XREF: WinMain(x,x,x,x)+1B2↓p
.text:00401070
.text:00401070 var_18          = qword ptr -18h
.text:00401070 var_10          = qword ptr -10h
.text:00401070 var_8           = qword ptr -8
.text:00401070 arg_0           = qword ptr 8
.text:00401070
.text:00401070      push      ebp

```

Please enter script body

```

import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.GetFunctionName(ea)
idc.SetFunctionCmt(ea, "check out later",1)

```

```

import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.GetFunctionName(ea)
idc.SetFunctionCmt(ea, "check out later",1)

```

在以上的脚本中我们打印出当前指令对应的汇编指令形式及地址，并通过函数 `idc.SetFunctionName(ea, comment, bool)` 对该指令进行注释，因为使用的是重复注释的形式，所以通过交叉引用的时候可以看到引用处的该函数也进行的注释。

```

.text:0040126F      fstp      [esp+400h+var_400] ; double
.text:00401272      call     sub_401070 ; check out later
.text:00401277      add      esp, 8
.text:0040127A      mov      edx, [ebp+var_3F0]
.text:00401280      fstp      [ebp+edx*8+var_1C0]
.text:00401287      fld      [ebp+var_1B8]
.text:0040128D      fstp      [ebp+var_3A8]
.text:00401293      jmp      short loc_401247

```

将一个函数或地址进行重命名是逆向中一件常见的工作，尤其当你遇到一系列用于封装函数的代码时，如下面这种代码，一个自动化的脚本将避免你大量的手工操作。

```

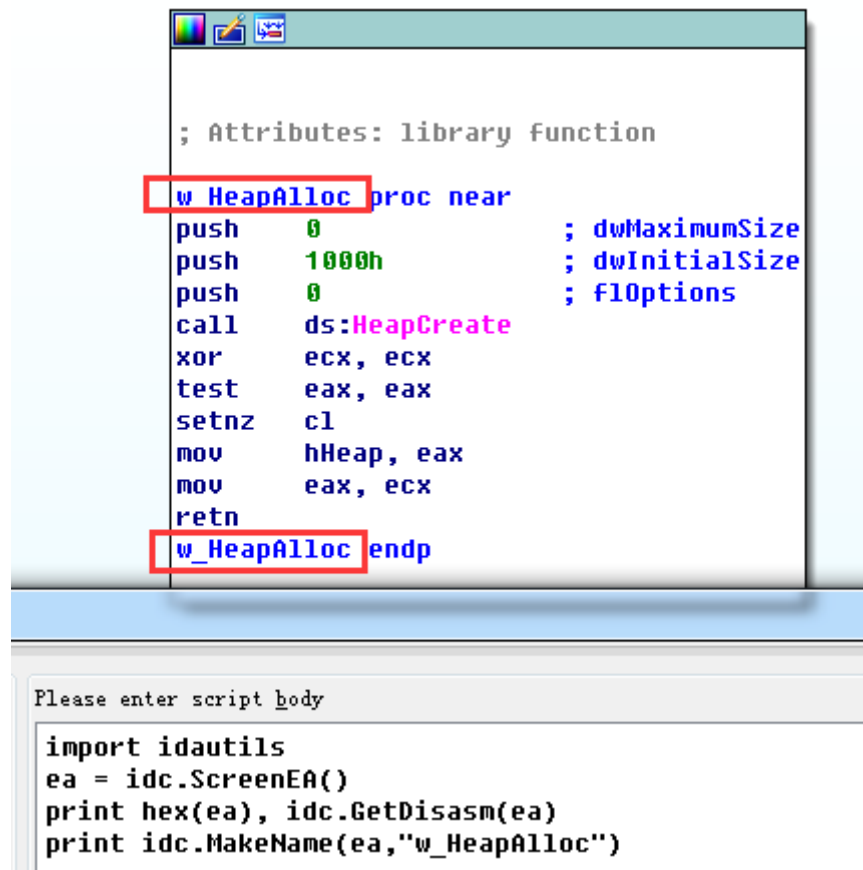
sub_10005B3E proc near

dwBytes = dword ptr 8

      push      ebp
      mov      ebp, esp
      push      [ebp+dwBytes] ; dwBytes
      push      8 ; dwFlags
      push      hHeap ; hHeap
      call     ds:HeapAlloc
      pop      ebp
      retn
sub_10005B3E endp

```

在上述的代码中，sub\_10005B3E 实际上是一个对函数 HeapAlloc 的封装，为了是代码更具可读性，我们需要将其重命名，比如改名叫 w\_HeapAlloc()，为一个地址所属函数进行重命名可以通过函数 idc.MakeName(ea, name)实现，其中 ea 为希望重命名的函数区内的第一条指令地址，name 为重命名之后的名字。



```

; Attributes: library function
w_HeapAlloc proc near
push    0                ; dwMaximumSize
push    1000h            ; dwInitialSize
push    0                ; flOptions
call    ds:HeapCreate
xor     ecx, ecx
test    eax, eax
setnz   cl
mov     hHeap, eax
mov     eax, ecx
retn
w_HeapAlloc endp

```

```

Please enter script body

import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.MakeName(ea, "w_HeapAlloc")

```

```

import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print idc.MakeName(ea, "w_HeapAlloc")

```

为了确保函数被重命名，可以通过函数 idc.GetFunctionName(ea) 打印出新生成的函数的地址。

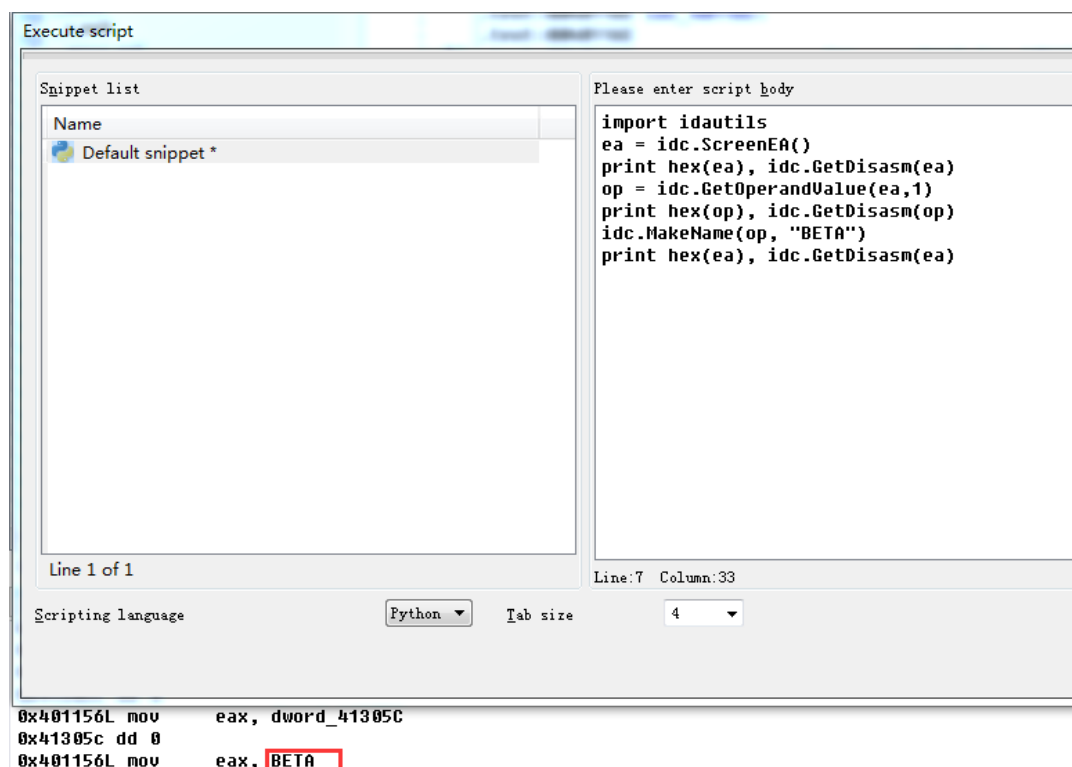
Python>idc.GetFunctionName(ea)

w\_HeapAlloc

为了给一个指令中的操作数进行重命名，首先需要获取需要重命名的操作数所在指令的地址，比如此处的地址为 00401156

.text:00401151	cmp	edx, 2Eh
.text:00401154	jz	short loc_40116C
.text:00401156	mov	eax, dword_41305C
.text:0040115B	mov	ecx, [eax]
.text:0040115D	movsx	edx, byte ptr [ecx+3]
.text:00401161	cmp	edx, 6Fh

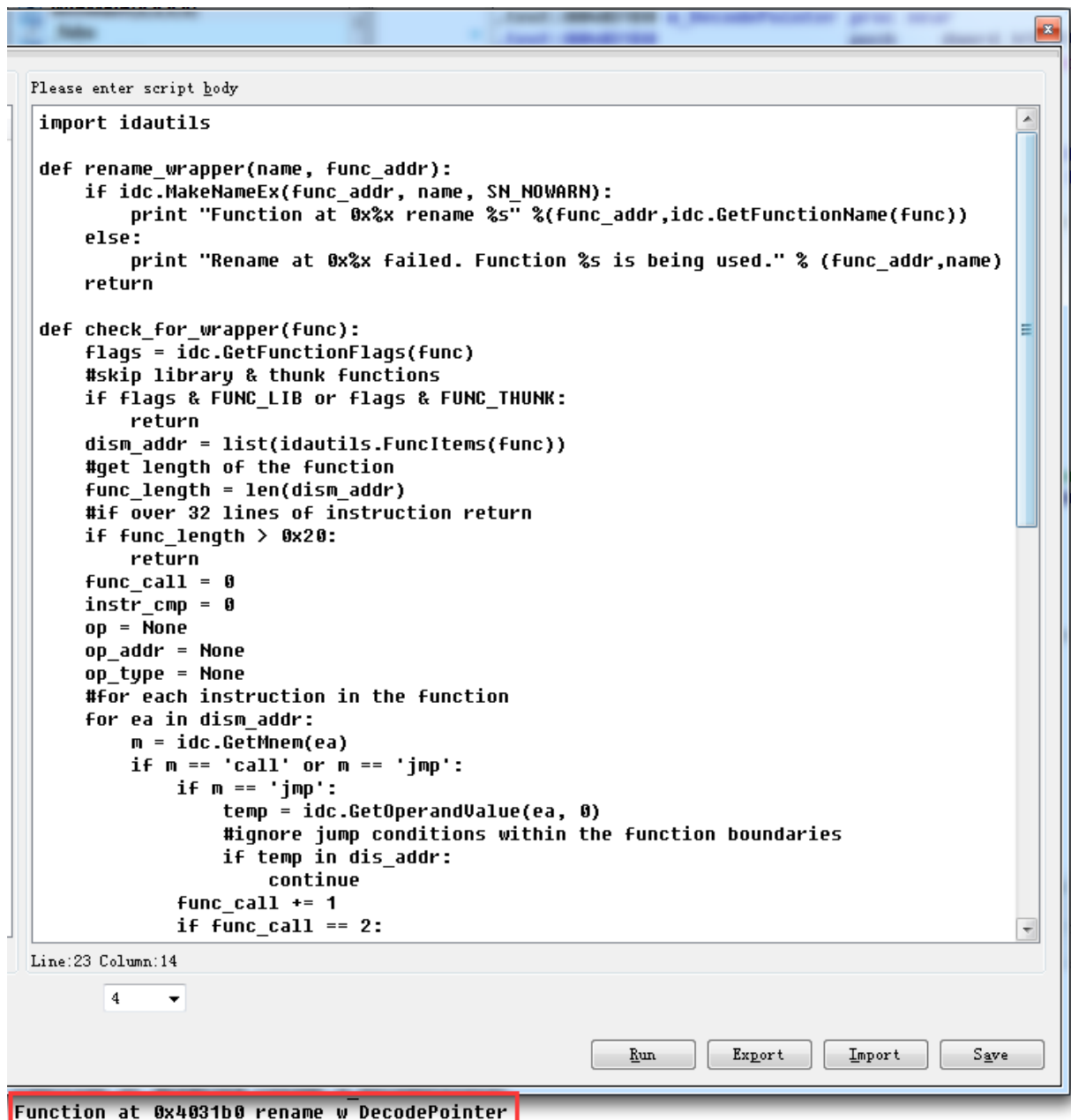
可以通过函数 GetOperandValue(ea, n)获取函数的操作数。



该脚本中，首先我们获取了该指令的地址，之后通过函数 `idc.GetOperandValue(ea, n)` 获取 `dword_41305c` 的值，最后使用函数 `idc.MakeName(ea, name)` 对该操作数进行重命名。

现在我们已经获取了足够多的基础知识，现在我们编写一个脚本用于动态注释并对函数进行重命名。





```
import idutils

def rename_wrapper(name, func_addr):
    if idc.MakeNameEx(func_addr, name, SN_NOWARN):
        print "Function at 0x%x rename %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
        if func_call == 2:
            return
        op_addr = idc.GetOperandValue(ea, 0)
        op_type = idc.GetOpType(ea, 0)
    elif m == 'cmp' or m == 'test':
        #wrapper 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 idutils.Functions():
    check_for_wrapper(func)

```

很多代码是不是很熟悉?其中一个不同的地方在于使用函数 `idc.MakeNameEx(ea, name, flag)` 代替 `rename_wrapper()`, 主要原因在于函数 `MakeNameEx(ea, name, flag)` 将会抛出一个警告当函数名字已经被使用过时。

## 处理原数据

获取原始数据的操作在逆向中至关重要, 原数据时代码或数据的二进制形式, 如下图所示, 每条指令左侧的十六进制数即为原数据。

```

.text:0040167C          loc_40167C:                                ; CODE XREF: _doexit+FD↓j
.text:0040167C  81 7D E0 90 E1 40 00          cmp     [ebp+var_20], offset unk_40E190
.text:00401683  73 11                        jnb     short loc_401696
.text:00401685  8B 45 E0                      mov     eax, [ebp+var_20]
.text:00401688  8B 00                        mov     eax, [eax]
.text:0040168A  85 C0                        test    eax, eax
.text:0040168C  74 02                        jz      short loc_401690

```

在获取原数据之前, 首先需要划分单位, 幸运的是这些操作原数据的 api 也是按获取的单位命名的。获取以比特的数据可以通过函数 `idc.Byte(ea)`, 获取一个字可以通过函数 `idc.Word(aea)`, 具体如下

```

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

```

如果光标放置到 00401685 处, 通过以上的函数, 可以获取以下输出。

```
Please enter script body

import idutils
ea = idc.ScreenEA()
print hex(ea), idc.GetDisasm(ea)
print hex(idc.Byte(ea))
print hex(idc.Word(ea))
print hex(idc.Dword(ea))
print hex(idc.Qword(ea))
print idc.GetFloat(ea)|
print idc.GetDouble(ea)

Line:8 Column:23
4 ▼

0x8b
0x458bL
0x8be0458bL
0x74c085008be0458bL
-8.63862981382e-32
2.42227971593e+254
```

有时在解密一段数据时，一次读取一个字或双字并不能满足我们的需求，这时候读取一块内存的原始数据成为我们的最佳选择，读取指定长度的内存可以使用函数 `idc.GetManyBytes(ea, size, use_dbg=False)`，最后一个参数仅当我们希望调试内存时会使用到。

```
Please enter script body

import idutils
ea = idc.ScreenEA()
for byte in idc.GetManyBytes(ea, 6):
    print "0x%x" % ord(byte)

Line:4 Column:29
4 ▼

0xff
0x15
0x98
0xe0
0x40
0x0
```

注意在 `idc.GetMangBytes(ea, size)` 中返回的是复数形式的二进制字符，不同于 `idc.Word(ea)`，`idc.Qword(ea)` 返回一个整形。

# 打补丁

有时候在逆向恶意样本时，样本中存在加密过的字符，通常这些字符会减慢分析的进度，这是对 idb 进行补丁将会很有用，可能你会想到重命名，但是重命名有覆盖率的限制，通过补丁改变某一个地址的值可以使用到一下函数。

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

其中 ea 表示需要补丁的地址，value 为补丁的需要写入的新值，value 的长度度量需要符合函数命中显示的度量单位。  
比如下列需要解密的字符。

```
.rodata:000B004C aM7a4nq_Na db 'm7A4nQ_/nA',0 ; DATA XREF: main+2CF↑to
.rodata:000B0057 aMN3 db 'm [(n3',0 ; DATA XREF: main+2F0↑to
.rodata:000B005E aM6_6n3 db 'm6_6n3',0 ; DATA XREF: main+30B↑to
.rodata:000B0065 aM4s4nacNa db 'm4S4nAC/nA',0 ; DATA XREF: main+382↑to
```

在开始分析之前需要定位到指定的解密函数。

```
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      jnl     short _loop
100012C1      pop     ebx
100012C2
100012C2 _ret:
100012C2      pop     esi
100012C3      retn
```

如上图所示，该函数是一个典型的 xor 解密函数，包括加密字符串和字符串的长度。

```
import idutils

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)

start = idc.SelStart()
end = idc.SelEnd()
print hex(start), hex(end)
xor(end-start, 0x42, start)
idc.GetString(start)
```

通过高亮选择需要解密的字符串，之后通过函数 `idc.SelStart()` 和 `idc.SelEnd()` 获取高亮区域的起始地址，之后通过函数 `idc.Byte(ea)` 读取需要加密的字符，每次 1 比特，通过 `xor` 指令与秘钥 `key` 操作解密之后，通过函数 `idc.PatchByte(ea, value)` 将解密之后的比特位通过补丁的形式写回 `idb` 文件中，该处补充了一个我自己的实例（见最后）

## 输入输出

文件的导入导出对于 `idapython` 很重要，尤其是当我们不知道文件路径，或用户不知道在哪儿存储自己的数据时，通过名字保存文件可以使用函数 `AskFile(forsave, mask, prompt)`，`forsave` 为 0 时表明我们需要打开一个 `dialog box`，1 表示需要打开一个保存的 `box`，`mask` 表示文件后缀，如果只想打开 `.dll` 的文件通过标记 `*.dll`，一个好的输入输出实例就是以下的 `IO_DATA` 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 idex, 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()
        print "the import file is: %s" % self.buffer
    except:
        sys.stdout.write('ERROR: Cannot access file')

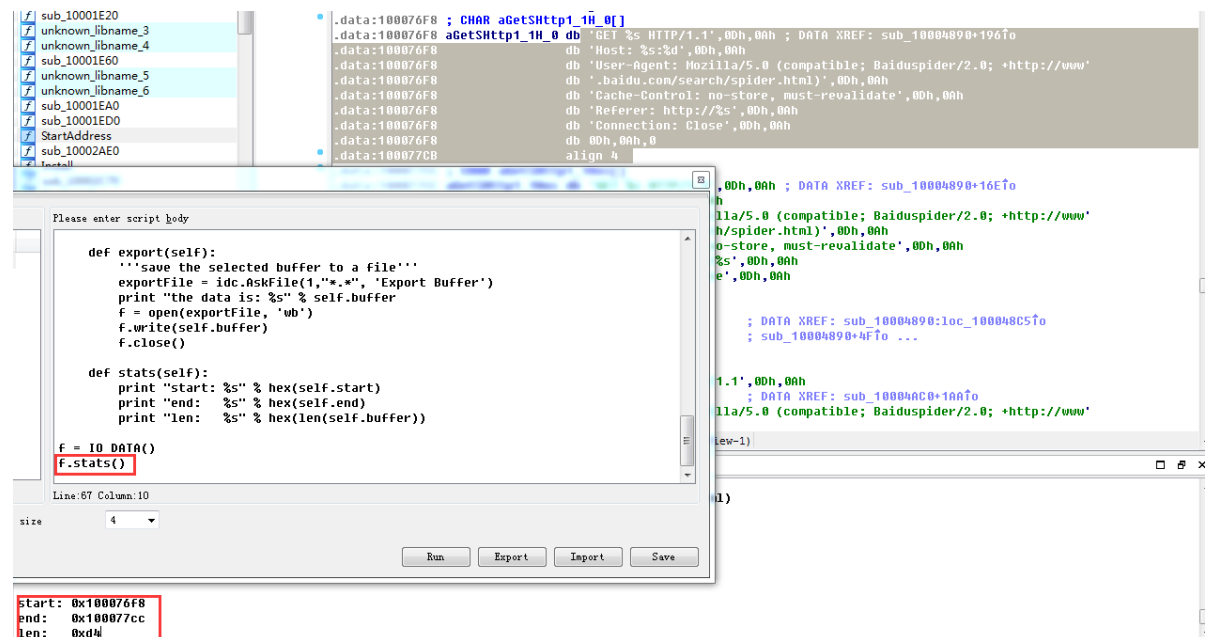
def export(self):
    '''save the selected buffer to a file'''
    exportFile = idc.AskFile(1, " *.*", 'Export Buffer')
    print "the data is: %s" % self.buffer
    f = open(exportFile, 'wb')
    f.write(self.buffer)
    f.close()

def stats(self):
    print "start: %s" % hex(self.start)
    print "end:    %s" % hex(self.end)
    print "len:    %s" % hex(len(self.buffer))

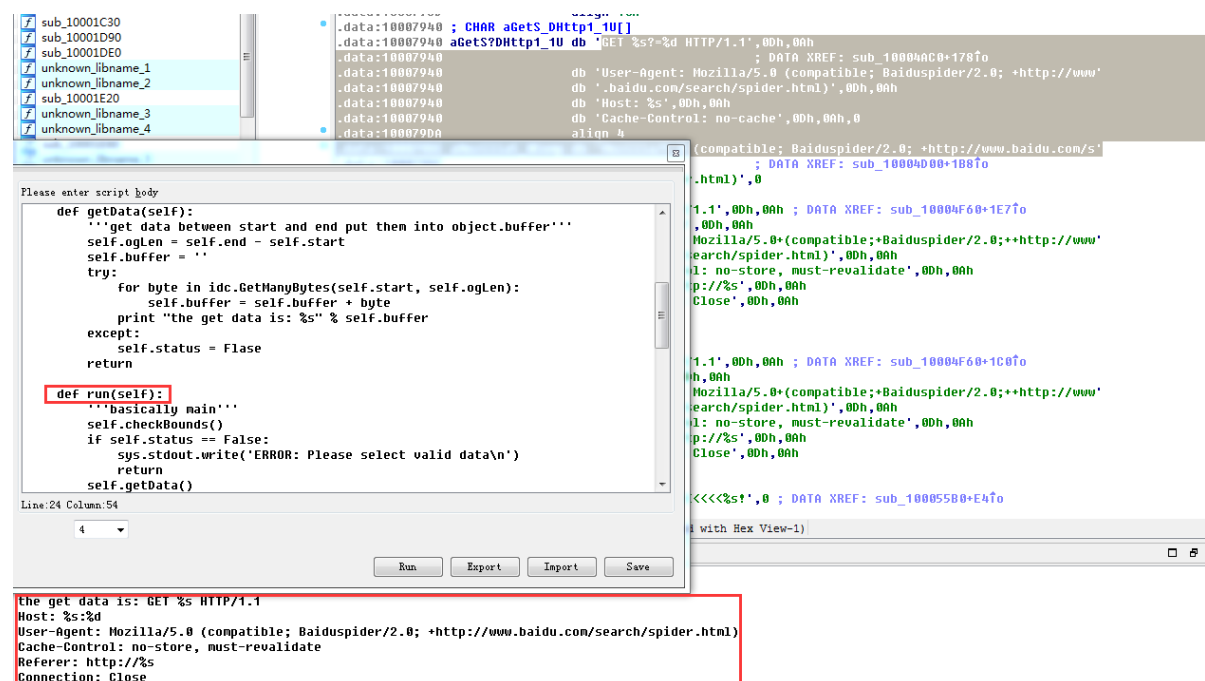
```

```
f = IO_DATA()
f.stats()
f.run()
f.importb()
```

通过这个类可以将 ida 中选中的数据转储到一个 buffer 中, 通过该 buffer 实现文件转储, 这对于处理 idb 中出现加密或解密的数据将非常有用, 比如我们可以将通过 python 脚本解密出的数据通过该类中的 patch 函数以补丁的形式写入到 idb 文件中, 以下是一个该类的简单利用, 函数 stats() 将选中的区域以 start, end, len 的形式输出。

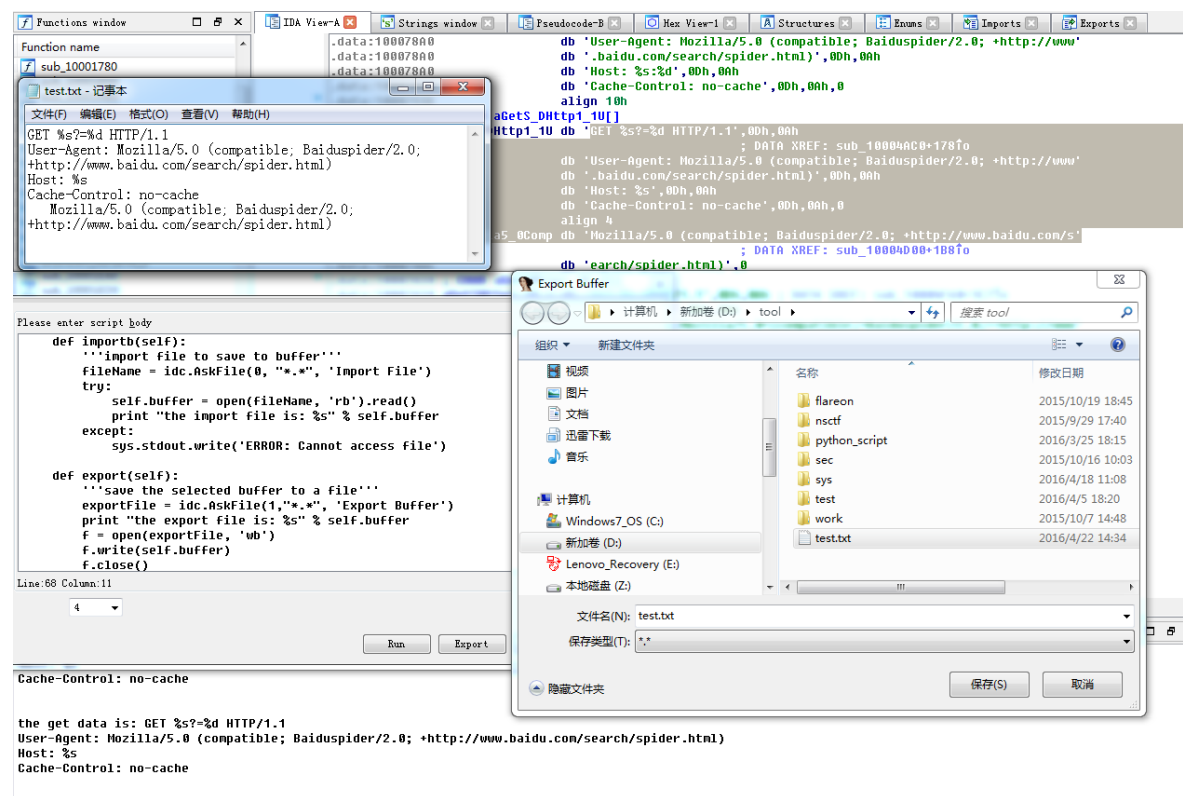


通过函数 run(), 实现将选中的数据读取到内存 buffer 变量中。





分别通过函数 export(),importb()导出导入。



相较于解释每一行代码，按函数来浏览每项功能会对读者更友好，以下会对每一个变量和函数的定义和作用进行解释，其中 f 是一个 IO\_DATA 类的对象。

1. obj.start  
包含选中地址的开始地址
2. obj.buffer  
用于转储读取的二进制变量的 buffer
3. obj.ogLen()  
buffer 的长度
4. obj.getData()  
将选中区间 obj.start 和 obj.end 中的数据以二进制的形式拷贝到 buffer 中
5. obj.patch()  
通过 obj.buffer 对选中区间 obj.start 和 obj.end 之间的数据进行补丁
6. obj.patch(d)  
通过参数对选中区间 obj.start 和 obj.end 之间的数据进行补丁
7. obj.importb()  
将指定文件中的内容读入到 obj.buffer 中
8. obj.export()  
将 buffer 的内容保存到指定的文件
9. obj.stats()  
以十六进制的形式打印出

# 批量处理解析文件

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。

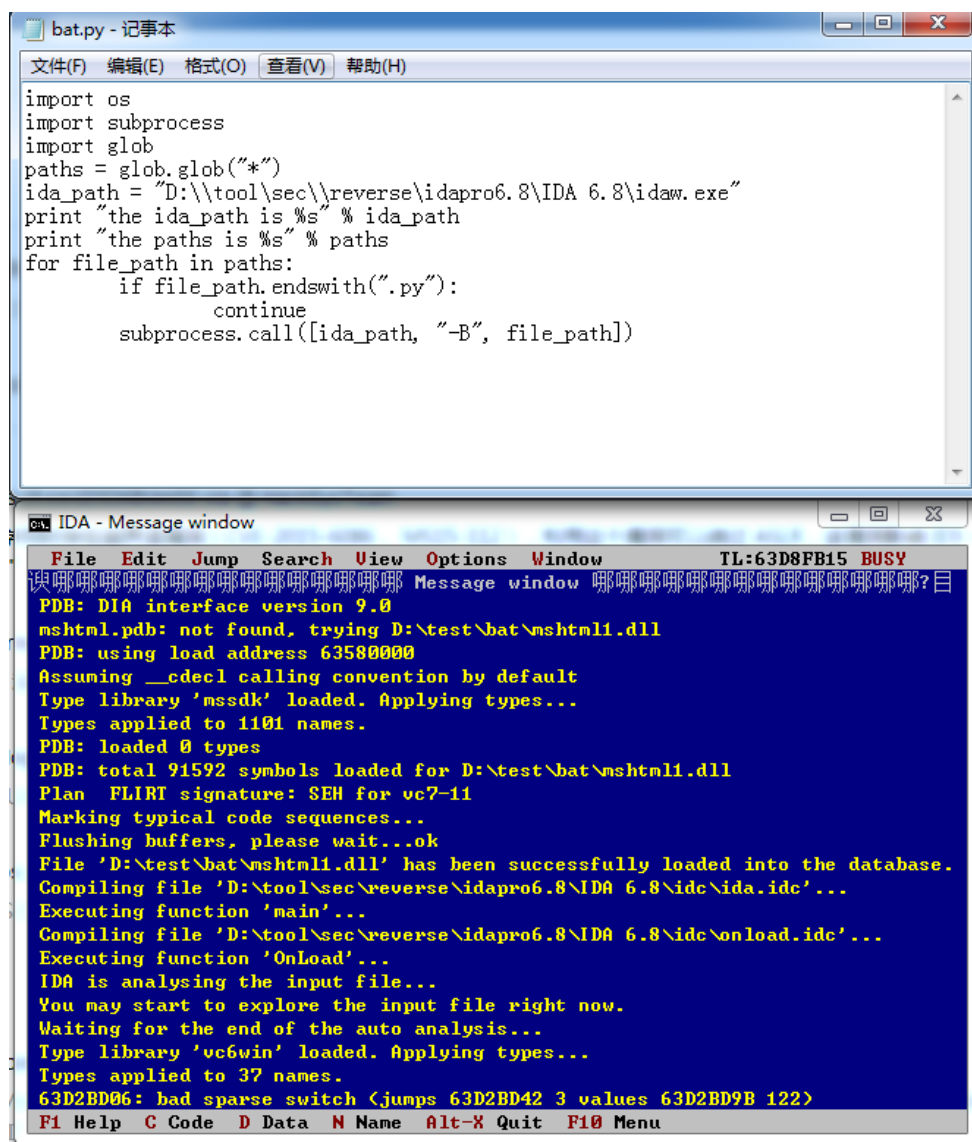
```
import os
import subprocess
import glob
paths = glob.glob("*")
ida_path = os.path.join(os.environ['PROGRAMFILES'], "IDA", "idaw.exe")

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\_file.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。



## 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 idutils.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。

"D:\tool\sec\reverse\idapro6.8\IDA 6.8\idaw.exe" -A -Scountrecord.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，具体如下所示：



## 实例

样本中使用了大量的 xor 加密，由于本身样本不全，无法运行（好吧我最稀饭的动态调试没了，样本很有意思，以后有时间做票大的分析），这个时候就只

好拜托 idapython 大法了（当然用 idc 也一样），期间用到几个问题，遂记录一番。

样本加密的字符如下，很简单，push 压栈之后，反复调用 sub\_1000204D 解密。

```
10012687      push    offset aDVjUxvkiUupUjg ; "磊Vi尝喝 尝握耽r"
1001268C      call   sub_1000204D
10012691      add     esp, 4
10012694      mov     dword_10066394, eax
10012699      push    offset aR0VcIdvKvki ; "愚 胆あ创炳 稻"
1001269E      call   sub_1000204D
100126A3      add     esp, 4
100126A6      mov     dword_10066258, eax
100126AB      push    offset aLijgloeJR ; "嬭r娶サ+緬"
100126B0      call   sub_1000204D
100126B5      add     esp, 4
100126B8      mov     dword_10066254, eax
100126BD      push    offset aAvKigLupjiglv ; "■3.姪2 護
100126C2      call   sub_1000204D
100126C7      add     esp, 4
100126CA      mov     dword_10066298, eax
100126CF      push    offset aBVvloeJ ; "忻ii娶サ+
100126D4      call   sub_1000204D
100126D9      add     esp, 4
100126DC      mov     dword_1006626C, eax
100126E1      push    offset aAvCIdjggU ; "■3.惋à哲5 4.
100126E6      call   sub_1000204D
100126EB      add     esp, 4
100126EE      mov     dword_10066414, eax
100126F3      push    offset aMvfvdvgvD0IUj ; "將敘当 4.さ 敵禄E "
100126F8      call   sub_1000204D
100126FD      add     esp, 4
```

此时，要写脚本的话，我们希望这个脚本能够足够通用，通常样本中的加密都是由一个函数实现，函数本身实现解密，传入的参数需要通常是解密字符，和 key 两个参数（当然肯定也有其他的模式），那么在写一个较为通用脚本前需要解决已下几个问题：

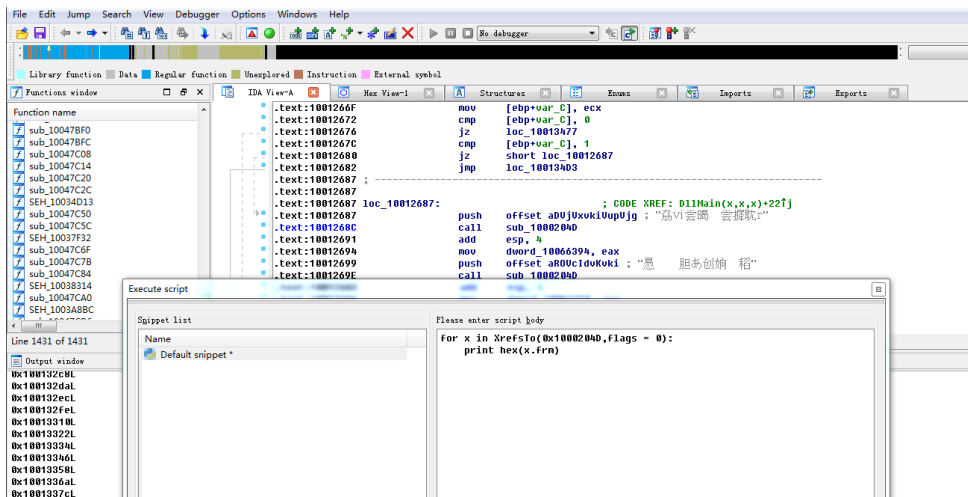
1. 如何获取所有调用解密函数的地址
2. 如何获取需要解密的字符
3. 解密算法如何
4. 解密之后的处理（最简单的比如注释）

首先针对第一个问题，如何获取调用解密函数的地址。

在 ida 中针对这一需求其实提供了一个函数 XrefsTo，该函数会返回一个地址的所有引用，在 ida 中通过一下两句脚本可以测试一下。

```
for x in XrefsTo(0x1000204D, flags = 0):
    print hex(x.frm)
```

测试结果如下：



第二个问题，如何获取加密的字符串。

在本例中函数的调用如下。

```
push    offset aROUcIdvKuki ; "愚 胆あ创始 稻"
call    sub_10002040
```

此时我们需要用到几个函数

第一个函数为 PrevHead，该函数用于获取一段代码段中的指令，ea 为开始，mines 为结尾，注意这个函数的搜索为降序搜索，说白了就是往前找。通过这个函数我们就可以获取解密函数之前的指令，即 push offset xxxxxx 指令所在的代码区域了

```
long PrevHead (long ea, long minea);
```

第二个函数为 GetMnem，用于获取 ea 指定地址中附近的指令

```
string GetMnem (long ea);
```

第三个函数为 GetOpnd，用于获取操作指定地址 ea 附近指令的操作码，注意此处的 n 为操作码的编号，由 0 开始，比如指令 push offset xxxxxx，0 号操作码为 push

```
string GetOpnd (long ea, long n);
```

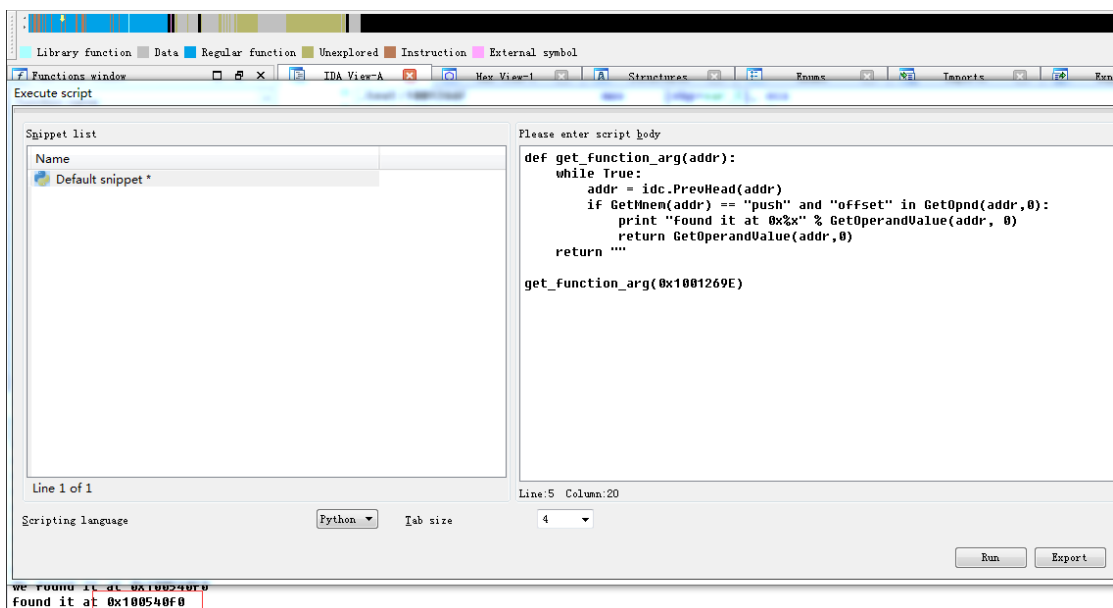
第四个函数为 GetOperandValue，用于获取指定地址 ea 附近指令的操作数，通用 n 为操作数的编号，由 0 开始，比如指令 push offset xxxxxx，0 号操作码为 000000

```
long GetOperandValue (long ea, long n);
```

通过这几个函数就可以获取压栈指令 push 压如的加密字符了，以下为参数获取函数

```
def get_function_arg(addr):
    while True:
        addr = idc.PrevHead(addr)
        if GetMnem(addr) == "push" and "offset" in GetOpnd(addr,0):
            return GetOperandValue(addr,0)
    return ""
```

测试效果如下：



但是此处我们只是获取了加密字符的地址，此处还需要计算出加密字符串的截止地址。

```
def get_stringlen(addr):
    out = ""
    while True:
        if idc.Byte(addr) != 0:
            out += chr(Byte(addr))
        else:
            break
        addr += 1
    return addr
```

第三个问题，解密字符。

由于此处只是简单的 xor 解密，含简单，当然以后有其他的加密方式，直接增加函数即可。

```
def decrypt_xor(stringstar,stringend):
    output = ""
    for i in range(stringstar, stringend):
        b = 0xC7 ^ idc.Byte(i)
        output += chr(b)
        idc.PatchByte(i,b)
    return output
```

第四个问题，注释

可以通过函数 MakeComm() 实现。ea 为注释地址，comment 为注释类容。

Success MakeComm (long ea,string comment); // give a comment

至此把所有的内容整合起来即可

```

def get_function_arg(addr):
    while True:
        addr = idc.PrevHead(addr)
        if GetMnem(addr) == "push" and "offset" in GetOpnd(addr,0):
            return GetOperandValue(addr,0)
    return ""

def get_string(addr):
    out = ""
    while True:
        if idc.Byte(addr) != 0:
            out += chr(Byte(addr))
        else:
            break
        addr += 1
    return out

def get_stringlen(addr):
    out = ""
    while True:
        if idc.Byte(addr) != 0:
            out += chr(Byte(addr))
        else:
            break
        addr += 1
    return addr

def decrypt_xor(stringstar,stringend):
    output = ""
    for i in range(stringstar, stringend):
        b = 0xC7 ^ idc.Byte(i)
        output += chr(b)
        idc.PatchByte(i,b)
    return output

def decrypt():
    print "[*] Attempting to decrypt strings in malware"
    for x in XrefsTo(0x1000204D,flags = 0):
        stringstart = get_function_arg(x.frm)
        stringend = get_stringlen(stringstart)
        retult = decrypt_xor(stringstart,stringend)
        print "Ref Addr: 0x%x | Decryptstring: %s" % (x.frm,retult)
        MakeComm(x.frm,retult)
        MakeComm(stringstart,retult)
    decrypt()

```

运行脚本之后的结果。

.text:10012687	push	offset aD0jUxvkiUupUjg ; "CreateRemoteThread"
.text:1001268C	call	sub_1000204D ; CreateRemoteThread
.text:10012691	add	esp, 4
.text:10012694	mov	dword_10066394, eax
.text:10012699	push	offset aR0UcIdvKvki ; "WriteProcessMemory"
.text:1001269E	call	sub_1000204D ; WriteProcessMemory
.text:100126A3	add	esp, 4
.text:100126A6	mov	dword_10066258, eax
.text:100126AB	push	offset alijglloeJR ; "LoadLibraryW"
.text:100126B0	call	sub_1000204D ; LoadLibraryW
.text:100126B5	add	esp, 4
.text:100126B8	mov	dword_10066654, eax
.text:100126BD	push	offset aAvKigLupjiglv ; "GetModuleHandleW"
.text:100126C2	call	sub_1000204D ; GetModuleHandleW
.text:100126C7	add	esp, 4
.text:100126CA	mov	dword_10066298, eax
.text:100126CF	push	offset aB0vloeJ ; "FreeLibrary"
.text:100126D4	call	sub_1000204D ; FreeLibrary
.text:100126D9	add	esp, 4
.text:100126DC	mov	dword_1006626C, eax
.text:100126E1	push	offset aAvCIdjggU ; "GetProcAddress"
.text:100126E6	call	sub_1000204D ; GetProcAddress
.text:100126EB	add	esp, 4
.text:100126EE	mov	dword_10066414, eax