第十二章 逆向辅助工具

北京邮电大学
崔宝江

第十二章 逆向辅助工具

- 一. IDAPython
- 二. Intel PIN
- 三. angr

一. IDAPython

- 当掌握了基本的逆向知识和分析能力之后，除了多多练习之外，还需要学习一些辅助逆向分析的工具
- 通过编写自动化脚本可以帮助逆向分析人员完成很多繁琐、重复性的工作，极大的节省逆向分析人员时间。

一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 1 基本介绍
  - IDAPython是IDA的一款插件，可以供使用者编写Python脚本，除了可以调用Python的所有已安装模块，还可以访问IDA提供的API接口。
  - 在学习IDAPyhon之前，可以先了解下IDA自带的脚本语言IDC。IDAPyhon集成了IDC脚本的所有功能，IDAPyhon在功能上要比IDC强大的多，并且由于是Pyhon语法，所以非常易懂。

一. IDAPython

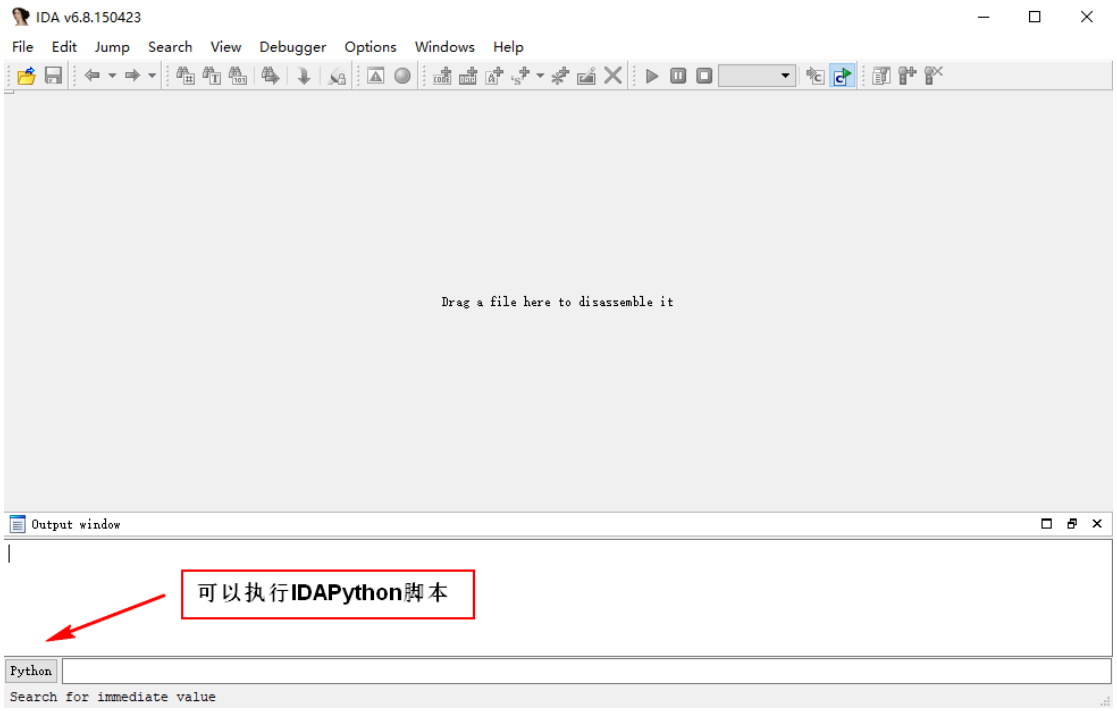
- （1）IDAPython安装
  - 安装方法可以参考github上的IDAPython project，下面只做简要介绍。
    - 安装python的2.6或者2.7环境，有一点要注意python位数必须与IDAPyhon位数一致（例如，都是32位或64位）
    - 将IDAPython中的整个python文件目录拷贝到IDA安装目录下。
    - 将IDAPython中plugin目录下的所有文件拷贝到IDA安装目录下的plugin文件夹内。
    - 将IDAPython中的python.cfg拷贝到IDA安装目录下的cfg文件夹内。

一. IDAPython

- 在python文件目录下，可以看到有以下三个文件idaapi.py、idautils.py、idc.py
  - idaapi模块提供了IDA核心的API函数
  - idc模块提供了IDC中所有的函数功能
  - idautils利用前两个模块里的函数，封装了一系列提供各种功能的实用函数，如生成一系列对象（函数、交叉引用等）的Python列表。

一. IDAPython

- 当IDAPython环境安装成功时，IDA界面最下面的命令行窗口旁的按钮上会显示Python字符串



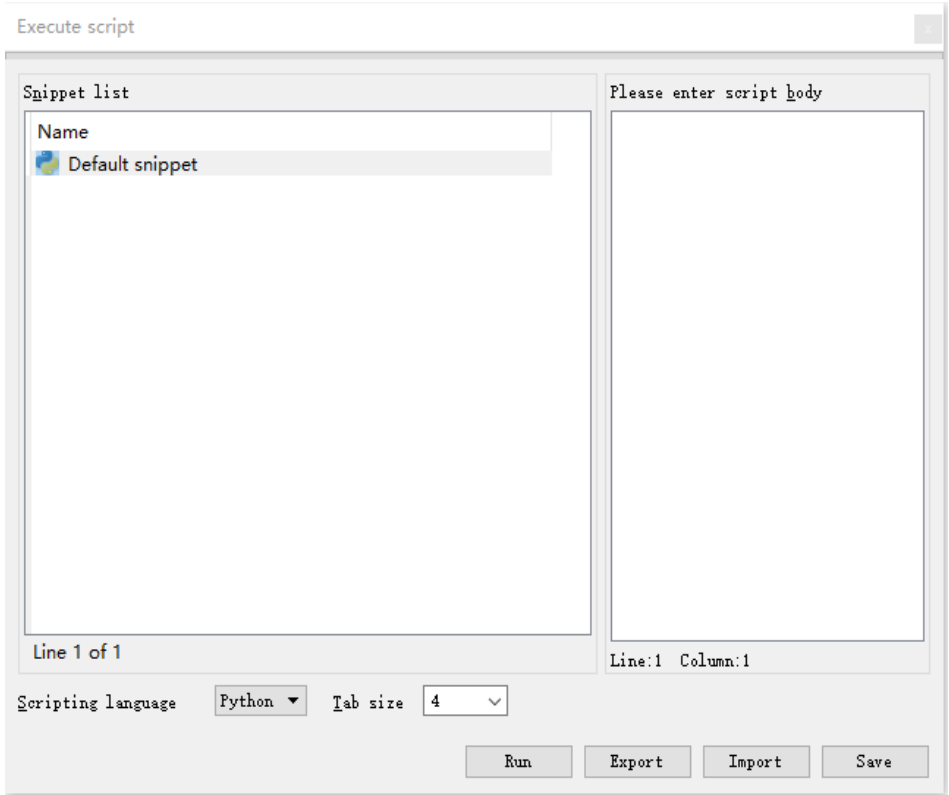
一. IDAPython

- （2）IDAPython脚本调用方式
  - 有三种方式可以执行Python脚本
    1. 通过IDAPython命令行，即在上面所说的命令行窗口中输入Python语句，但该方式只能执行一条Python语句
    2. 要执行一个独立的脚本文件，可以使用File->Script File，然后选择要执行的脚本文件。

一. IDAPython

（2）IDAPython脚本调用方式

③ 如果只想执行几条语句，又不愿意创建一个脚本文件，可以使用**File->Script Command**，即会弹出如下图所示的对话框，将**Scripting language**一项设定为**Python**即可



一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 在逆向分析过程中，如果可以读取或修改二进制数据是非常有帮助的。
  - IDAPython提供了接口函数来实现这些功能
  - 下面是程序的一段汇编指令，最左边的是指令地址，右边是汇编语句，中间则是汇编指令的16进制数据

```
804863e:   50                      push   eax
804863f:   68 16 87 04 08          push   0x8048716
8048644:   e8 77 fd ff ff          call   80483c0 <printf@plt>
8048649:   83 c4 10                add    esp,0x10
804864c:   b8 00 00 00 00          mov    eax,0x0
8048651:   8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
8048654:   c9                      leave
8048655:   8d 61 fc                lea    esp,[ecx-0x4]
8048658:   c3                      ret
```


一. IDAPython

- 在获取数据之前，需要确定两个信息：想获取数据的单位大小以及地址
  - 访问一字节数据可以用**idc.Byte(ea)**，**ea**为数据的地址
  - 访问一字的数据可以用**idc.Word(ea)**
  - 想要获取浮点数据可以用**idc.GetFloat(ea)**

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


一. IDAPython

❒尝试获取地址0x804863e处的数据

```
Python>ea = 0x804863e
Python>print hex(idc.Byte(ea))
0x50
Python>print hex(idc.Word(ea))
0x6850L
Python>print hex(idc.Dword(ea))
0x87166850L
Python>print hex(idc.Qword(ea))
0x77e8080487166850L
```


一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 当分析汇编语言时，重点关注的是操作码和操作数，**IDAPython**也提供了丰富接口供大家访问指令数据。
  - **here()**函数可以获取当前光标所在的地址，再通过**idc.GetDisasm(ea)**即可获取该地址处的汇编语句

```
Python>ea = here()
Python>print idc.GetDisasm(ea)
call    _printf
```


一. IDAPython

- 当知道一个地址时，便可以通过函数**idc.NextHead(ea)/idc.PrevHead(ea)**来获取下一条指令/上一条指令。
- 值得注意的是，这两个函数是返回下一条指令和上一条指令的地址，而非下一个地址和上一个地址，区别于**idc.NextAddr(ea)**和**idc.PrevAddr(ea)**。

一. IDAPython

- 除了**idc.GetDisasm(ea)，IDAPython**还提供更细的粒度去访问汇编指令。
  - **idc.GetMnem(ea)**获取指令的操作码
  - **idc.GetOpnd(ea, n)**获取操作数，操作数的下标从**0**开始

```
Python>ea = here()
Python>print idc.GetDisasm(ea)
add     esp, 10h
Python>print idc.GetOpnd(ea,0), idc.GetOpnd(ea,1)
esp 10h
```


一. IDAPython

- 既然可以获取操作数，那如何知道这个操作数的类型呢？就像上面这个例子，这条指令的操作数又有寄存器，又有立即数
- 可以通过函数**idc.GetOpType(ea, n)**获取参数的类型，**n**是操作数下标

一. IDAPython

- 操作数常见类型主要有以下几类
  - o_void
  - o_reg
  - o_mem
  - o_phrase
  - o_displ
  - o_imm

一. IDAPython

- **o_void**
  - **如果一条指令不包含操作数，就会返回o_void，值为0。**

```
Python>print hex(ea), idc.GetDisasm(ea)
0x8048654L leave
Python>print idc.GetOpType(ea,0)
0
```


一. IDAPython

- **o_reg**
  - **如果操作数是寄存器，就会返回o_reg，值为1。**

```
Python>print hex(ea), idc.GetDisasm(ea)
0x804863eL push    eax
Python>print idc.GetOpType(ea,0)
1
```


一. IDAPython

- o_mem
  - 如果操作数是个内存地址，就会返回o_mem，值为2。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x804853bL cmp dword ptr ds:8048000h, 0
Python>print idc.GetOpType(ea,0)
2
```


一. IDAPython

- **o_phrase**
  - **如果操作数是由寄存器组成的表达式，形如[Base Reg + Index Reg]，就会返回o_phrase，值为3。**
  - **其中BaseReg一般都保存着基址，IndexReg保存偏移，IndexReg可有可无。**
  - **一般在读写内存或快速运算时会用到该操作数。**

```
Python>ea=here()
Python>print hex(ea), idc.GetDisasm(ea)
0x804a093L mov   [eax+ebx], dl
Python>print idc.GetOpType(ea,0)
3
```


一. IDAPython

- **o_displ**
  - **这个类型和前一个类型差不多，只不过操作数里多了一项常数，形如[Base Reg + Index Reg + value]，值为4。**

```
Python>print hex(ea), idc.GetDisam(ea)
0x80485b1L
Python>print hex(ea), idc.GetDisasm(ea)
0x80485b1L mov eax, [ebp-14h]
Python>print idc.GetOpType(ea,1)
4
```


一. IDAPython

- o_imm
  - 如果操作数为立即数，就会返回o_imm，值为5。

```
Python>print hex(ea), idc.GetDisasm(ea)
0x804a0caL mov ebx, 8Ch
Python>print idc.GetOpType(ea,1)
5
```


一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 4函数处理
  - 通过IDAPython提供的函数接口，可以很容易获取到一个程序里的所有函数信息。

一. IDAPython

- 通过IDAPython提供的函数接口，可以很容易获取到一个程序里的所有函数信息。

```
import idautils
for func in idautils.Functions():
   print hex(func), idc.GetFunctionName(func)
#output
0x8048374L .init_proc
0x80483b0L .mprotect
......
0x804dd4cL puts
0x804dd50L strlen
0x804dd54L __libc_start_main
0x804dd5cL __imp___gmon_start__
```

- 这里idc.GetFunctionName(ea)函数根据字面意思，就可知道是获取某地址所在的函数名称。

一. IDAPython

- 反过来通过函数名称获取函数的起始地址也是可以的，就需要用到函数**idc.LocByName(name)**：

```
Python>print hex(idc.LocByName('main'))
0x804853bL
```


一. IDAPython

- 当知道某个函数的地址时，可以通过**idc.NextFunction(ea)/idc.PrevFunction(ea)**获取其下一个函数的地址或上一个函数的地址。
- **idautils.FuncItems(ea)**用来获取某函数里所有的指令地址

```
for ins in idautils.FuncItems(0x804853b):
    print hex(ins), idc.GetDisasm(ins)
#output
0x804853bL lea     ecx, [esp+4]
0x804853fL and     esp, 0FFFFFFF0h
0x8048542L push    dwordptr [ecx-4]
0x8048545L push    ebp
0x8048546L movebp, esp
……
0x8048654L leave
0x8048655L lea     esp, [ecx-4]
0x8048658L retn
```


一. IDAPython

- 除此之外，也可以通过获取函数的边界地址，再调用**idc.NextHead(ea)**来遍历函数内的指令地址。
- 获取函数边界地址的函数为**idc.GetFunctionAttr(ea, FUNCATTR_START)**和**idc.GetFunctionAttr(ea, FUNCATTR_END)**：

```
start = idc.GetFunctionAttr(0x804853b, FUNCATTR_START)
end = idc.GetFunctionAttr(0x804853b, FUNCATTR_END)
ins = start
while ins < end:
    print hex(ins), idc.GetDisasm(ins)
    ins = idc.NextHead(ins)
```


一. IDAPython

- 另外一个比较重要的函数是**idaapi.get_func(ea)，idc.GetFunctionAttr(ea, attr)**本质上也是调用的该函数。**idaapi.get_func(ea)**返回一个**function**类，可以用**dir(class)**函数查看一个类包含的变量、方法等。

```
Python>func = idaapi.get_func(0x804853b)
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']
```


一. IDAPython

- 也可以通过**func.startEA**和**func.endEA**获取函数的边界地址。

```
Python>func = idaapi.get_func(0x804853b)
Python>print "Start: 0x%x, End: 0x%x" % (func.startEA, func.endEA)
Start: 0x804853b, End: 0x8048659
```


一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 5交叉引用
  - 在使用IDA过程中，大家肯定都会用到交叉引用这个功能，通过交叉引用能帮助分析者快速理清程序的逻辑以及调用关系。IDAPython也提供了交叉引用模块，定位数据在哪里被使用，或者一个函数在哪里被调用。
  - 数据的交叉引用函数为idautils.DataRefsTo(ea)和idautils.DataRefsFrom(ea)，两个函数返回类型都是迭代器，DataRefsTo(ea)返回的是这个数据被引用的语句地址。DataRefsFrom(ea)返回的是地址ea处引用了哪里的数据。

一. IDAPython

- **DataRefsTo(ea)用法如下所示**

```
Python>print hex(ea), idc.GetDisasm(ea)
0x8048716L db 'Usage: %s input',0Ah,0
Python>for addr in idautils.DataRefsTo(ea): print hex(addr), idc.GetDisasm(addr)
0x804863fL push    offset aUsageSInput; "Usage: %s input\n"
```


一. IDAPython

- **DataRefsFrom(ea)用法如下所示**

```
Python>print hex(ea), idc.GetDisasm(ea)
0x804863fL push    offset aUsageSInput; "Usage: %s input\n"
Python>for addr in idautils.DataRefsFrom(ea): print hex(addr), idc.GetDisasm(addr)
0x8048716L db 'Usage: %s input',0Ah,0
```


一. IDAPython

- 代码的交叉引用函数为**idautils.CodeRefsTo(ea, flow)**和**idautils.CodeRefsFrom(ea, flow)**，两个函数同样都是返回迭代器。
- 当想查看哪里调用了**printf**函数时，可以用**idautils.CodeRefsTo(ea, flow)**，**flow**的值为**0**或者**1**，具体有什么区别之后会介绍，这里先关注这两个函数的用法。

一. IDAPython

- 这里首先是通过**idc.LocByName**得到**printf**函数的地址，然后通过函数**idautils.CodeRefsTo**获取程序中所有引用**printf**的地方。

```
Python>printf_addr = idc.LocByName('printf')
Python>print hex(printf_addr), idc.GetDisasm(printf_addr)
0x4192c0L extrnprintf:dword
Python>for addr in idautils.CodeRefsTo(printf_addr,0): print hex(addr), idc.GetDisasm(addr)
0x411414L call    ds:printf
0x41144aL call    ds:printf
```


一. IDAPython

❒**idautils.CodeRefsFrom用法如下所示，这里的0x4192c0是printf函数的导入地址表**

```
Python>print hex(ea), idc.GetDisasm(ea)
0x411414L call    ds:printf
Python>for addr in idautils.CodeRefsFrom(ea,0): print hex(addr), idc.GetDisasm(addr)
0x4192c0L extrn printf:dword
```


一. IDAPython

- 想要调用这些函数，还得先区分是代码还是数据，太麻烦
- **IDAPython还提供了idautils.XrefsTo和idautils.XrefsFrom函数，而无需区分参数是数据地址还是代码地址。**

一. IDAPython

❒下面代码示例就是通过idautils.XrefsTo函数找到调用'Hello %s'的地方。

```
ea=0x415860
print hex(ea), idc.GetDisasm(ea)
for xref in idautils.XrefsTo(ea, 0):
    print xref.type, idautils.XrefTypeName(xref.type), hex(xref.frm), hex(xref.to), xref.iscode
    print hex(xref.frm), idc.GetDisasm(xref.frm)
#output
0x415860 db 'Hello %s',0Ah,0
1 Data_Offset 0x41140fL 0x415860L 0
0x41140fL push    offset Format   ; "Hello %s\n"
```


一. IDAPython

- 交叉引用的这些API，有个遗留问题还没解决，就是这些API的第二个参数flow有什么功能，传入0或1有什么区别。

```
.text:004119E1                  cmp     dword_41856C, 0
.text:004119E8                  jz      short loc_411A07
.text:004119EA                  push    offset dword_41856C
.text:004119EF                  call    sub_41117C
.text:004119F4                  add     esp, 4
.text:004119F7                  test    eax, eax
.text:004119F9                  jz      short loc_411A07
.text:004119FB                  push    0
.text:004119FD                  push    2
.text:004119FF                  push    0
.text:00411A01                  call    dword_41856C
.text:00411A07
.text:00411A07 loc_411A07:
.text:00411A07
.text:00411A07                  push    1
```


一. IDAPython

- 将光标指向0x00411A07地址处，调用XrefsTo查看其交叉引用，当第二个参数为0时，结果如下

```
ea=here()
print hex(ea), idc.GetDisasm(ea)
for xref in idautils.XrefsTo(ea, 0):
    print xref.type, idautils.XrefTypeName(xref.type), hex(xref.frm), hex(xref.to), xref.iscode
#output
0x411a07L push    1
21 Ordinary_Flow 0x411a01L 0x411a07L 1
19 Code_Near_Jump 0x4119e8L 0x411a07L 1
19 Code_Near_Jump 0x4119f9L 0x411a07L 1
```


一. IDAPython

- 可以看到，一共输出了三个会执行到**0x00411A07**地址的指令语句，除了比较显而易见的**0x4119e8**和**0x4119f9**两个是通过**jmp**指令跳转到目标地址，还有**0x411a01**地址也会引用目标地址，顺序执行时其下一条指令便是**0x00411A07**。

一. IDAPython

- 当XrefsTo第二个参数为1时，结果如下：

```
ea=here()
print hex(ea), idc.GetDisasm(ea)
for xref in idautils.XrefsTo(ea, 1):
    print xref.type, idautils.XrefTypeName(xref.type), hex(xref.frm), hex(xref.to), xref.iscode
#output
0x411a07L push    1
19 Code_Near_Jump 0x4119e8L 0x411a07L 1
19 Code_Near_Jump 0x4119f9L 0x411a07L 1
```

- 可以看到输出结果中不包含Ordinary_Flow类型的引用，这就是两者的区别

一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 6 搜索模块
  - 有时候也需要查找特定的二进制数据，就像在写漏洞利用脚本时，需要寻找一些跳板指令。
    - 比方读者想要查找这样的十六进制数据0x55 0x89 0xE5，这个翻译成指令就是push ebp;mov ebp,esp。
  - IDAPython提供了函数idc.FindBinary(ea, flag, searchstr, radix=16)来方便搜索二进制数据。
    - ea是开始搜索的起始地址
    - flag表明搜索的方向和条件

一. IDAPython

- flag表明搜索的方向和条件
  - 常用的flag有SEARCH_UP、SEARCH_DOWN、SEARCH_NEXT、SEARCH_REGEX、SEARCH_UNICODE等
    - SEARCH_UP和SEARCH_DOWN表示搜索的方向，SEARCH_NEXT会返回下一个找到的对象，SEARCH_REGEX表示支持正则查找，SEARCH_UNICODE会将搜索的字符串当作Unicode处理。
- searchstr是需要查找的模式字符串
- radix是指数据的进制表示方式，默认为16进制

一. IDAPython

- 函数idc.FindBinary(ea, flag, searchstr, radix=16) 搜索二进制数据例子

```
pattern = "55 89 E5"
addr = MinEA()
for x in range(0,5):
addr = idc.FindBinary(addr, SEARCH_DOWN | SEARCH_NEXT,
pattern, 16)
    if addr != idc.BADADDR:
        print hex(addr), idc.GetDisasm(addr)
#output
0x8048498L push    ebp
0x80484d1L push    ebp
0x80484f9L push    ebp
0x8048529L push    ebp
0x8048545L push    ebp
```


一. IDAPython

- 如果不加**SEARCH_NEXT**参数，**addr**会一直是同一个值，并不会返回下一个找到的对象，因此会输出**5**个**0x8048498L push ebp**。

一. IDAPython

- 查找字符串用函数idc.FindText(ea, flag, y, x, searchstr)，代码样例

```
searchstr = "usage"
addr = MinEA()
end = MaxEA()
while addr< end:
addr = idc.FindText(addr, SEARCH_DOWN, 0, 0, searchstr)
    if addr == idc.BADADDR:
        break
    else:
        print hex(addr), idc.GetDisasm(addr)
     addr = idc.NextHead(addr)
#output
0x804863fL push    offset aUsageSInput; "Usage: %s input\n"
0x8048716L db 'Usage: %s input',0Ah,0
```


一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

- 7调试模块
  - 前面所介绍的IDAPython用法，大多数都用在静态分析中，如果想在程序动态运行过程中处理些操作，就需要用到调试模块函数。

一. IDAPython

- 这里重点介绍以下几种，注意的是这些函数基本上都只能在程序运行的时候调用：
  - **RunTo(ea)**
    - 该函数可以让程序运行到参数ea指定的地址。
  - **StepInto()**
    - 单步运行程序，遇到call指令会跟进，相当于IDA调试的快捷键F7。
  - **StepOver()**
    - 单步运行程序，遇到call指令会步过，相当于IDA调试的快捷键F8。

一. IDAPython

- **AddBptEx(ea, size, bpttype)**
  - 添加一个新的断点，**ea**是断点的地址，**bpttype**是断点的类型，主要有五种，**BPT_WRITE**为硬件写断点，**BPT_RDWR**为硬件读写断点，**BPT_EXEC**是硬件执行断点，**BPT_SOFT**是软件断点，**BPT_DEFAULT**为软件断点和硬件执行断点的组合。
  - 当断点类型为硬件断点时，**size**指明断点大小，软件断点时**size**为**0**。
- **GetDebuggerEvent(wfne, timeout)**
  - 等待调试器事件，并且根据参数**wfne**来决定下一步处理。参数**wfne**常用的有三种**WFNE_ANY**、**WFNE_SUSP**、**WFNE_CONT**，**WFNE_ANY**可以捕捉所有的调试事件，包括断点、单步、加载动态库等，**WFNE_SUSP**是等待程序中断事件，**WFNE_CONT**可以从中断事件中恢复并且程序继续执行。

一. IDAPython

- **DelBpt(ea)**
  - 删除参数ea地址处的断点。
- **EnableBpt(ea, enable)**
  - 根据enable的值来决定激活还是禁用断点。
- **CheckBpt(ea)**
  - 检查参数ea指定地址处断点的状态，断点状态主要有以下几种：存在、激活、禁用等，不同状态返回不同的值。
- **SetBptCnd(ea, cnd)**
  - 在地址ea处设置条件断点，条件表达式由cnd决定。

一. IDAPython

- **DbgByte(ea)**
  - 获取内存中一字节数据，相应的函数有**DbgWord(ea)**、**DbgDword(ea)**、**DbgQword(ea)**。
- **DbgRead(ea, size)**
  - 读取**ea**地址处**size**字节大小的内容。
- **DbgWrite(ea, data)**
  - 往**ea**地址处写**size**字节大小的内容。
- **GetRegValue(name)**
  - 获取寄存器值。
- **SetRegValue(value, name)**
  - 设置寄存器值为**value**。

一. IDAPython

- 以实验练习12-1-1的程序为例，来简单掌握一下这些函数的用法。IDA加载该程序，可以看到main函数伪代码如下：

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    int v4; // [sp+0h] [bp-Ch]@1
4    int v5; // [sp+4h] [bp-8h]@1
5    __int16 v6; // [sp+8h] [bp-4h]@1
6    char v7; // [sp+Ah] [bp-2h]@1
7
8    v4 = dword_407044;
9    v5 = dword_407048;
10   v7 = byte_40704E;
11   v6 = word_40704C;
12   if ( argc == 2 )
13   {
14     if ( !strcmp(argv[1], (const char *)&v4) )
15     {
16       sub_4010A0(aCorrect, v4);
17       return 0;
18     }
19     sub_4010A0(aWrong, v4);
20   }
21   return 0;
22 }
```

一. IDAPython

- 该段代码主要功能就是将**main**函数的参数**argc[1]**和地址**0x407044**处的字符串比较，字符串内容如图所示，当正确时输出**Correct!**，错误时输出**Wrong!**

```
.data:00407030 aWrong          db 'Wrong!',0Ah,0       ; DATA XREF: _main:loc_401087↑o
.data:00407038 aCorrect        db 'Correct!',0Ah,0     ; DATA XREF: _main+74↑o
.data:00407042                 align 4
.data:00407044 aHelloworld     db 'HelloWorld',0       ; DATA XREF: _main+3↑r
.data:00407044                                         ; _main+8↑r ...
.data:0040704F                 align 10h
.data:00407050 off_407050      dd offset __exit        ; DATA XREF: __amsg_exit+1C↑r
.data:00407054 dword_407054    dd 1                    ; DATA XREF: __FF_MSGBANNER+E↑r
.data:00407054                                         ; sub_402829+46↑r ...
```


一. IDAPython

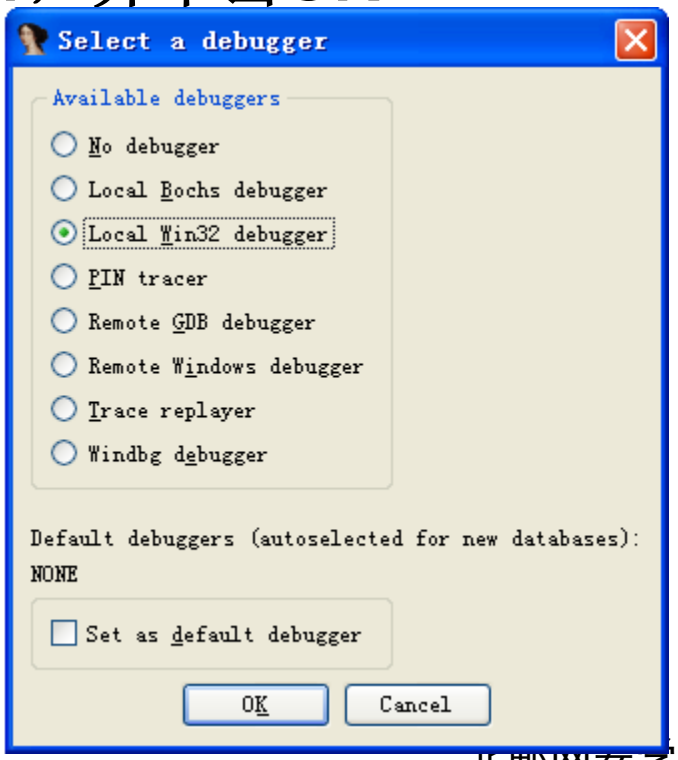
❒练习

- ❍现在编写一个小脚本，即使当程序命令行参数不为"HelloWorld"时，也能输出Correct!。
- ❍具体做法为，修改strcmp函数的返回值即可，对应就是修改0x401070地址处的寄存器eax的值，将其设为0。

```
.text:0040106E loc_40106E:                             ; CODE XREF: _main+67↑j
.text:0040106E                 pop     esi
.text:0040106F                 pop     ebx
.text:00401070                 test    eax, eax
.text:00401072                 jnz     short loc_401087
.text:00401074                 push    offset aCorrect ; "Correct!\n"
.text:00401079                 call    sub_4010A0
.text:0040107E                 add     esp, 4
.text:00401081                 xor     eax, eax
.text:00401083                 add     esp, 0Ch
.text:00401086                 retn
```

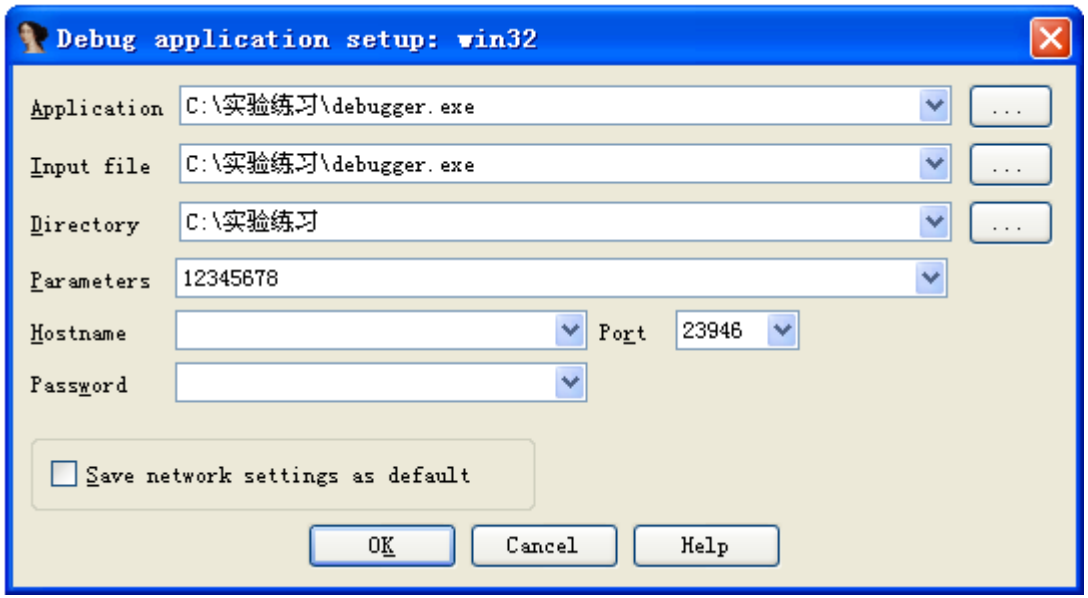

一. IDAPython

- 首先需要让IDA调试目标程序，点击Debugger->Select a debugger，选择LocalWin32 Debugger，并单击OK



一. IDAPython

- 然后选择**Debugger->Process option，在Parameters一栏里随便输入几个字符串**



一. IDAPython

- 接下来在**main**函数起始地址**0x401000**处用快捷键**F2**下断点，选择**Debugger->Start process**开始调试程序，程序会在**main**函数入口中断下来，然后加载脚本。
- **请练习编写此脚本：即使当程序命令行参数不为"HelloWorld"时，也能输出Correct!。**

一. IDAPython

- 脚本思路提示
  - 这段脚本先获得**EIP**的地址，由于在**main**函数入口地址下了断点，所以程序会停在**main**函数入口处，此时的**EIP**地址即**main**函数地址
    - 之所以需要获得**main**函数地址，因为在除**XP**操作系统上，其他操作系统程序加载的基址都是随机的。
  - 然后通过**RunTo(main+0x70)**语句执行到**test　eax, eax**指令处，用**SetRegValue**函数将其设为**0**，
  - 然后继续用**RunTo**函数将程序运行到地址**00401074**处，获取**printf**参数的地址并将其内容打印出来
  - 接下来调用**StepOver**函数便可以看到程序窗口输出**Correct!**字符串。

一. IDAPython

- 1 基本介绍
- 2 数据处理
- 3 指令处理
- 4 函数处理
- 5 交叉引用
- 6 搜索模块
- 7 调试模块
- 8 命令行脚本执行

一. IDAPython

8 命令行脚本执行

- 除了采用图形化界面的方式执行IDAPython脚本，还可以采用命令行的方式，具体命令如下

```
idaq.exe –A –c -S"脚本名称" 目标程序
```

- 其中各个参数详情如下：
  - **-A** 让**ida**自动运行，不需要人工干预。也就是在处理的过程中不会弹出交互窗口，但是如果从来没有使用过**ida**那么许可协议的窗口无论是否使用这个参数都将会显示。
  - **-c** 参数会删除所有与参数中指定的文件相关的数据库，并且生成一个新的数据库。

一. IDAPython

- **-S 参数用于指定ida在分析完数据之后执行的idc脚本，该选项和参数之间没有空格，并且搜索目录为ida目录下的idc文件夹。**
- **当读者想要ida执行完脚本就退出，还需要在脚本里添加两个函数idc.Wait()和idc.Exit(0)**
  - **idc.Wait()添加在脚本的最开始，目的是等待ida分析完成**
  - **idc.Exit(0)添加在脚本的最末尾，表示一旦执行完脚本就退出ida进程**

第十二章 逆向辅助工具

- 一. IDAPython
- 二. Intel PIN
- 三. angr

二. Intel PIN

- 1 介绍
- 2 环境搭建
- 3 实验练习

二. Intel PIN

- 1 介绍
  - PIN是Intel公司推出的一款动态二进制插桩工具，支持Windows/Linux平台，并且提供了丰富的API接口以及详细的用户手册
  - 开发人员可以利用PIN开发Pintools插件，来获取程序运行过程中的动态信息，包括指令、函数、寄存器内容、内存地址等。

二. Intel PIN

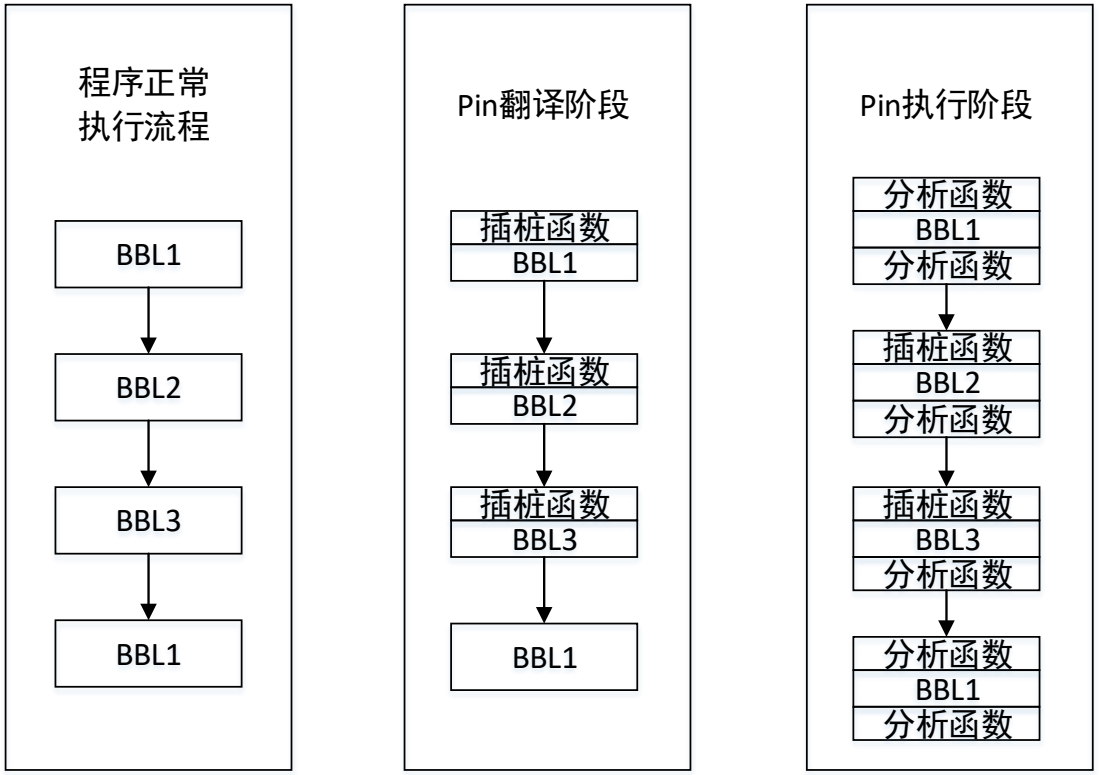
- PIN定义了六个对象：image、section、routine、trace、bbl、ins，用户可以根据需要使用不同级别的插桩粒度
  - Image：程序文件映像，表示一个完整的PE文件，包括主进程文件、加载的DLL等。
  - Section：程序区段，包括代码段、数据段等。
  - Routine：函数层级，可以通过Routine相关函数获取函数名、函数边界等信息。
  - Trace：指令流，定义为单入口多出口的结构。Trace由多个基本块组成。

二. Intel PIN

- **BBL**：基本块，定义为单入口单出口的结构。单入口单出口结构的意思是，一旦执行**BBL**入口处的汇编指令，则一定会顺序执行完其余指令，直到出口指令，**BBL**由多条汇编指令组成。
- **Instruction**：汇编指令，**Pin**可供插装的最小粒度单位，表示程序在执行中的单独汇编指令，可以在执行前后插装用户代码。

二. Intel PIN

- PIN工作流程分为两个阶段
  - 翻译阶段和执行阶段，这两个阶段是交叉进行的。
  - 以BBL插桩为例，具体流程如图所示



二. Intel PIN

- 当程序正常执行时，BBL执行顺序为BBL1、BBL2、BBL3、BBL1
- Pin首先执行翻译阶段，在翻译阶段，Pin不会对相同的代码片段重复翻译，而仅遇到新的代码片段时才会对该代码片段翻译，调用插桩函数在其前后插入用户自定义代码，即分析函数。
  - 所以该图中当程序第二次执行到BBL1时，并不会调用插桩函数。
- 翻译阶段结束后，Pin会先去执行分析函数代码，然后再将控制权交回原程序执行
- 当执行阶段完毕后，Pin又继续新的翻译阶段。所以Pin的工作过程也是一个即时编译的过程。

二. Intel PIN

❒2 环境搭建

❍1 Windows

- ❖以Visual Studio2012为例，在官网上下载Pin 2.14适用vc11（即VS2012）的版本。

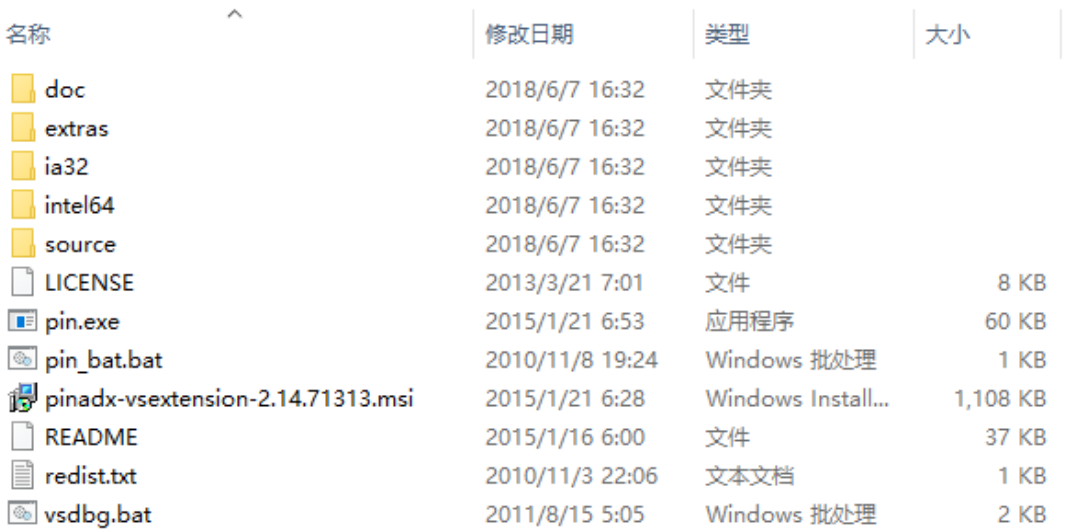
Windows

IA32 and intel64 (x86 32 bit and 64 bit)

Version	Date	Kit				Documentation		
Pin 3.6	Feb 11, 2018	97554				Manual	PinCRT	Release Notes
Pin 3.5	Nov 8, 2017	97503				Manual	PinCRT	Release Notes
Pin 3.4	Sep 6, 2017	97438				Manual	PinCRT	Release Notes
Pin 3.2	Feb 13, 2017	81205				Manual	PinCRT	Release Notes
Pin 2.14	Feb 03, 2015	vc9	vc10	vc11	vc12	Manual		Release Notes

二. Intel PIN

- 下载好后进行解压，Pin根目录如下图所示



名称	修改日期	类型	大小
doc	2018/6/7 16:32	文件夹	
extras	2018/6/7 16:32	文件夹	
ia32	2018/6/7 16:32	文件夹	
intel64	2018/6/7 16:32	文件夹	
source	2018/6/7 16:32	文件夹	
LICENSE	2013/3/21 7:01	文件	8 KB
pin.exe	2015/1/21 6:53	应用程序	60 KB
pin_bat.bat	2010/11/8 19:24	Windows 批处理	1 KB
pinadx-vsextension-2.14.71313.msi	2015/1/21 6:28	Windows Install...	1,108 KB
README	2015/1/16 6:00	文件	37 KB
redist.txt	2010/11/3 22:06	文本文档	1 KB
vsdbg.bat	2011/8/15 5:05	Windows 批处理	2 KB

二. Intel PIN

- 选择工程文件
  - 启动**VS2012**，选择文件->打开->项目/解决方案，打开**source\tools\MyPinTool\MyPinTool.vcxproj**工程文件，默认是**debug**版本，只有一个源文件**MyPinTool.cpp**。
- 将工程文件编译为**DLL**
  - 点击生成->生成解决方案，会提示**fatal error LNK1281:** 无法生成 **SAFESEH** 映像，这是因为**Pin**不支持**SafeSEH**，右击工程选择属性，在链接器->高级里，将映像具有安全异常处理设置为否，重新生成解决方案即可，编译成功会生成一个**MyPinToll.dll**。

二. Intel PIN

- **Windows下Pin使用命令，将该dll注入到待插桩程序**

```
pin.exe -t MyPinTool.dll -- 可执行程序
```

- **如果想要调试自己开发的Pintools插件（即生成的dll），可以用以下命令**

```
pin.exe –pause_tool 15 -  t  MyPinTool.dll -- 可执行程序
```

- **15是秒数，换成其他整数也可以，表示程序暂停15秒之后再运行，输入完这条命令会回显一个进程的PID，然后选择VS2012附加到进程即可开始调试该插件。**

二. Intel PIN

- 2Linux
  - Linux安装就非常方便了，下载好适用Linux的版本，直接解压即可。具体使用方法可以参考官网上的使用手册，即下载页面中的Manual部分，或者下载后doc/html目录下的index.html文件，这里不做过多介绍。

二. Intel PIN

- 3 实验练习
  - 这里以实验练习12-2的SimpleVM为例，来给大家介绍一下如何用pintools解题。
  - SimpleVM是一道来自reversing.kr上的逆向题，直接运行程序发现会提示Access Denied，因此需要sudo运行该程序，提示输入数据，输入错误时会提示Wrong!。
  - 此题用IDA加载时，会发现该程序的代码段被加密了，想要解这道题的一种繁琐的方法就是只能动态调试慢慢分析其逻辑了。

二. Intel PIN

- 网上有人提出了一种很好的思路
  - 即通过统计程序运行的指令数来间接判断你的输入是否正确，主要思想就是如果程序对输入是一个字符一个字符校验的，那么每当多输对一个字符，程序执行的指令数就会增加
  - 假设正确输入长度为n，这样爆破的复杂度不再是256的n次方（一个字符有256种可能，包括不可见字符），而是O(256*n)了。
  - 统计程序运行的指令数就用到刚刚介绍的pintools，这里在source/tools/ManualExamples/inscount0.cpp的基础上做一下更改，原来inscount0.cpp代码里是将指令数输出到文件，更改为输出到屏幕即可

二. Intel PIN

- 练习的代码

第十二章 逆向辅助工具

- 一. IDAPython
- 二. Intel PIN
- 三. angr

三. angr

- **angr是一个二进制代码分析工具，能够自动化完成二进制文件的分析，并找出漏洞。**
- **angr是一个基于python的二进制漏洞分析框架，它将以前多种分析技术集成进来，它能够进行动态的符号执行分析（如，KLEE和Mayhem），也能够进行多种静态分析。**

三. angr

- 符号执行是在运行程序时，用符号来替代真实值。
  - 符号执行相较于真实值执行的优点在于，当使用真实值执行程序时，我们能够遍历的程序路径只有一条
  - 使用符号进行执行时，由于符号是可变的，就可以利用这一特性，尽可能的将程序的每一条路径遍历
  - 这样的话，必定存在至少一条能够输出正确结果的分支，每一条分支的结果都可以表示为一个离散关系式，使用约束求解引擎即可分析出正确结果。

三. angr

- 可以利用Angr这个工具尝试对一些CTF题目进行符号执行来找到正确的解答
- 要注意的是符号执行的路径选择问题到现在依旧是一个很大的问题
  - 也就是当我们的程序存在循环时，因为符号执行会尽量遍历所有的路径，所以每次循环之后会形成至少两个分支，当循环的次数足够多时，就会造成路径爆炸，整个机器的内存会被耗尽。

三. angr

- **Angr使用**
  - **Angr在求解REVERSE题目时很有用，但在处理PWN题目时，多用在一些辅助的位置，比如寻找 strcmp 等敏感的函数等**

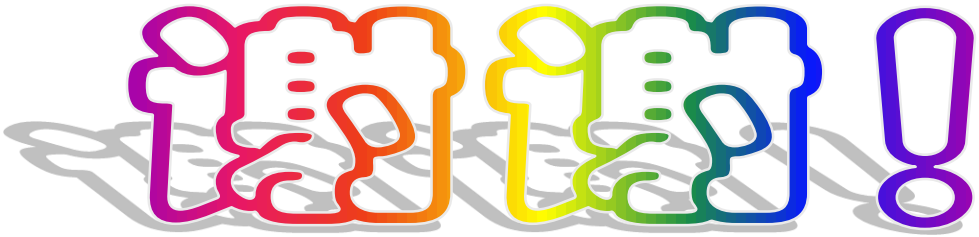
三. angr

- angr的基本过程：
  - 1）将二进制程序载入angr分析系统
  - 2）将二进制程序转换成中间语言（intermediate representation, IR）
  - 3）将IR语言转换成语义较强的表达形式，比如，这个程序做了什么，而不是它是什么。
  - 4）执行进一步的分析，比如，完整的或者部分的静态分析（依赖关系分析，程序分块）、程序空间的符号执行探索（挖掘溢出漏洞）、一些对于上面方式的结合

小结

- 一. IDAPython
- 二. Intel PIN
- 三. angr

Q & A



北邮网安学院　崔宝江