官方API使用例程 [文档]

本人结合LyScript插件API函数实现的一些通用案例，用于演示插件内置方法是如何灵活组合运用的，其目的是让用户可以自行研究学习API函数的参数传递，并能够通过案例的学习快速掌握官方API函数的使用方法。

PEFile载入PE程序到内存: 将PE可执行文件中的内存数据通过PEfile模块打开并读入内存，实现PE参数解析。

```python
from LyScript32 import MyDebug
import pefile

if __name__ == "__main__":
    # 初始化
    dbg = MyDebug()
    dbg.connect()

    # 得到text节基地址
    local_base = dbg.get_local_base()

    # 根据text节得到程序首地址
    base = dbg.get_base_from_address(local_base)

    byte_array = bytearray()
    for index in range(0,4096):
        read_byte = dbg.read_memory_byte(base + index)
        byte_array.append(read_byte)

    oPE = pefile.PE(data = byte_array)
    timedate = oPE.OPTIONAL_HEADER.dump_dict()
    print(timedate)
```

如下通过`LyScirpt`模块配合`PEFile`模块解析内存镜像中的`section`节表属性。

```python
from LyScript32 import MyDebug
import pefile

if __name__ == "__main__":
    # 初始化
    dbg = MyDebug()
    dbg.connect()

    # 得到text节基地址
    local_base = dbg.get_local_base()

    # 根据text节得到程序首地址
    base = dbg.get_base_from_address(local_base)

    byte_array = bytearray()
    for index in range(0,8192):
        read_byte = dbg.read_memory_byte(base + index)
        byte_array.append(read_byte)
    oPE = pefile.PE(data = byte_array)

    for section in oPE.sections:
        print("%10s %10x %10x %10x"
    %(section.Name.decode("utf-8"), section.VirtualAddress,
    section.Misc_VirtualSize, section.SizeOfRawData))
    dbg.close()
```

验证PE启用的保护方式: 验证保护方式需要通过`dbg.get_all_module()`遍历加载过的模块，并依次读入`DllCharacteristics`与操作数进行与运算得到保护方式。

```
from LyScript32 import MyDebug
import pefile

if __name__ == "__main__":
    # 初始化
    dbg = MyDebug()
    dbg.connect()

    # 得到所有加载过的模块
    module_list = dbg.get_all_module()

    print("-" * 100)
    print("模块名 \t\t\t 基址随机化 \t\t DEP保护 \t\t 强制完整性 \t\t SEH异常保护
\t\t")
    print("-" * 100)

    for module_index in module_list:
        print("{:15}\t\t".format(module_index.get("name")),end="")

        # 依次读入程序所载入的模块
        byte_array = bytearray()
        for index in range(0, 4096):
            read_byte = dbg.read_memory_byte(module_index.get("base") + index)
            byte_array.append(read_byte)

        oPE = pefile.PE(data=byte_array)

        # 随机基址 => hex(pe.OPTIONAL_HEADER.DllCharacteristics) & 0x40 == 0x40
        if ((oPE.OPTIONAL_HEADER.DllCharacteristics & 64) == 64):
            print("True\t\t\t",end="")
        else:
            print("False\t\t\t",end="")
        # 数据不可执行 DEP => hex(pe.OPTIONAL_HEADER.DllCharacteristics) & 0x100 ==
0x100
        if ((oPE.OPTIONAL_HEADER.DllCharacteristics & 256) == 256):
            print("True\t\t\t",end="")
        else:
            print("False\t\t\t",end="")
        # 强制完整性=> hex(pe.OPTIONAL_HEADER.DllCharacteristics) & 0x80 == 0x80
        if ((oPE.OPTIONAL_HEADER.DllCharacteristics & 128) == 128):
            print("True\t\t\t",end="")
        else:
            print("False\t\t\t",end="")
        if ((oPE.OPTIONAL_HEADER.DllCharacteristics & 1024) == 1024):
            print("True\t\t\t",end="")
        else:
            print("False\t\t\t",end="")
        print()
    dbg.close()
```

计算PE节区内存特征: 通过循环读入节区数据，并动态计算出该节的MD5以及CRC32特征值输出。

```
import binascii
import hashlib
from LyScript32 import MyDebug

def crc32(data):
    return "0x{:X}".format(binascii.crc32(data) & 0xffffffff)

def md5(data):
    md5 = hashlib.md5(data)
    return md5.hexdigest()

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 循环节
    section = dbg.get_section()
    for index in section:
        # 定义字节数组
        mem_byte = bytearray()

        address = index.get("addr")
        section_name = index.get("name")
        section_size = index.get("size")

        # 读出节内的所有数据
        for item in range(0,int(section_size)):
            mem_byte.append( dbg.read_memory_byte(address + item))

        # 开始计算特征码
        md5_sum = md5(mem_byte)
        crc32_sum = crc32(mem_byte)

        print("[*] 节名: {:10s} | 节长度: {:10d} | MD5特征: {} | CRC32特征: {}"
              .format(section_name,section_size,md5_sum,crc32_sum))

    dbg.close()
```

文件FOA与内存VA转换: 封装实现`get_offset_from_va()`用于将VA内存虚拟地址转为FOA文件偏移地址，封装实现`get_va_from_foa()`用于将FOA文件偏移地址转为VA内存虚拟地址。

```
from LyScript32 import MyDebug
import pefile

# 传入一个VA值获取到FOA文件地址
```

```python
def get_offset_from_va(pe_ptr, va_address):
    # 得到内存中的程序基地址
    memory_image_base = dbg.get_base_from_address(dbg.get_local_base())

    # 与VA地址相减得到内存中的RVA地址
    memory_local_rva = va_address - memory_image_base

    # 根据RVA得到文件内的FOA偏移地址
    foa = pe_ptr.get_offset_from_rva(memory_local_rva)
    return foa

# 传入一个FOA文件地址得到VA虚拟地址
def get_va_from_foa(pe_ptr, foa_address):
    # 先得到RVA相对偏移
    rva = pe_ptr.get_rva_from_offset(foa_address)

    # 得到内存中程序基地址,然后计算VA地址
    memory_image_base = dbg.get_base_from_address(dbg.get_local_base())
    va = memory_image_base + rva
    return va

# 传入一个FOA文件地址转为RVA地址
def get_rva_from_foa(pe_ptr, foa_address):
    sections = [s for s in pe_ptr.sections if s.contains_offset(foa_address)]
    if sections:
        section = sections[0]
        return (foa_address - section.PointerToRawData) + section.VirtualAddress
    else:
        return 0

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 载入文件PE
    pe = pefile.PE(name=dbg.get_local_module_path())

    # 读取文件中的地址
    rva = pe.OPTIONAL_HEADER.AddressOfEntryPoint
    va = pe.OPTIONAL_HEADER.ImageBase + pe.OPTIONAL_HEADER.AddressOfEntryPoint
    foa = pe.get_offset_from_rva(pe.OPTIONAL_HEADER.AddressOfEntryPoint)
    print("文件VA地址: {} 文件FOA地址: {} 从文件获取RVA地址: {}".format(hex(va), foa,
hex(rva)))

    # 将VA虚拟地址转为FOA文件偏移
    eip = dbg.get_register("eip")
    foa = get_offset_from_va(pe, eip)
    print("虚拟地址: 0x{:x} 对应文件偏移: {}".format(eip, foa))

    # 将FOA文件偏移转为VA虚拟地址
    va = get_va_from_foa(pe, foa)
    print("文件地址: {} 对应虚拟地址: 0x{:x}".format(foa, va))

    # 将FOA文件偏移地址转为RVA相对地址
    rva = get_rva_from_foa(pe, foa)
    print("文件地址: {} 对应的RVA相对地址: 0x{:x}".format(foa, rva))

    dbg.close()
```

查找SafeSEH内存地址: 读入PE文件到内存，验证该程序的SEH保护是否开启，如果开启则尝试输出SEH内存地址。

```
from LyScript32 import MyDebug
import struct

LOG_HANDLERS = True

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 得到PE头部基地址
    memory_image_base = dbg.get_base_from_address(dbg.get_local_base())

    peoffset = dbg.read_memory_dword(memory_image_base + int(0x3c))
    pebase = memory_image_base + peoffset

    flags = dbg.read_memory_word(pebase + int(0x5e))
    if(flags & int(0x400)) != 0:
        print("SafeSEH | NoHandler")

    numberofentries = dbg.read_memory_dword(pebase + int(0x74))
    if numberofentries > 10:

        # 读取 pebase+int(0x78)+8*10 | pebase+int(0x78)+8*10+4  读取八字节,分成两部分
读取
        sectionaddress, sectionsize =
[dbg.read_memory_dword(pebase+int(0x78)+8*10),

dbg.read_memory_dword(pebase+int(0x78)+8*10 + 4)
                                       ]
        sectionaddress += memory_image_base
        data = dbg.read_memory_dword(sectionaddress)
        condition = (sectionsize != 0) and ((sectionsize == int(0x40)) or
(sectionsize == data))

        if condition == False:
            print("[-] SafeSEH 无保护")
        if data < int(0x48):
            print("[-] 无法识别的DLL/EXE程序")

        sehlistaddress, sehlistsize =
[dbg.read_memory_dword(sectionaddress+int(0x40)),

dbg.read_memory_dword(sectionaddress+int(0x40) + 4)
                                       ]
        if sehlistaddress != 0 and sehlistsize != 0:
            print("[+] SafeSEH 保护中 | 长度: {}".format(sehlistsize))
            if LOG_HANDLERS == True:
                for i in range(sehlistsize):
                    sehaddress = dbg.read_memory_dword(sehlistaddress + 4 * i)
                    sehaddress += memory_image_base
                    print("SEHAddress = {}".format(hex(sehaddress)))

    dbg.close()
```

扫描所有模块匹配特征: 针对程序内加载的所有模块扫描特征码，如果找到了会返回内存地址，以及模块详细参数。

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 获取所有模块
    for entry in dbg.get_all_module():
        eip = entry.get("entry")
        base = entry.get("base")
        name = entry.get("name")

        if eip != 0:
            # 设置EIP到模块入口处
            dbg.set_register("eip",eip)

            # 开始搜索特征
            search = dbg.scan_memory_one("ff 25 ??")
            if search != 0 or search != None:

                # 如果找到了,则反汇编此行
                dasm = dbg.disasm_fast_at(search)

                print("addr = {} | base = {} | module = {} | search_addr = {} |
dasm = {}"
                      .format(eip,base,name,eip,dasm.get("disasm")))

    dbg.close()
```

简单的搜索汇编特征: 使用python实现方法，通过特定方法扫描内存范围，如果出现我们所需要的指令集序列，则输出该指令的具体内存地址。

```
from LyScript32 import MyDebug

# 检索指定序列中是否存在一段特定的指令集
def SearchOpCode(OpCodeList,SearchCode,ReadByte):
            pass

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    # 得到EIP位置
    eip = dbg.get_register("eip")
    # 反汇编前1000行
    disasm_dict = dbg.get_disasm_code(eip,1000)

    # 搜索一个指令序列,用于快速查找构建漏洞利用代码
    SearchCode = [
        ["push 0xC0000409", "call 0x003F1B38", "pop ecx"],
        ["mov ebp, esp", "sub esp, 0x324"]
    ]

    # 检索内存指令集
    for item in range(0,len(SearchCode)):
        Search = SearchCode[item]
        # disasm_dict = 返回汇编指令 Search = 寻找指令集 1000 = 向下检索长度
        ret = SearchOpCode(disasm_dict,Search,1000)
        if ret != None:
            print("指令集: {} --> 首次出现地址:
{}".format(SearchCode[item],hex(ret)))

    dbg.close()
```

得到汇编指令机器码: 该功能主要实现，得到用户传入汇编指令所对应的机器码，这段代码你可以这样来实现。

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    addr = dbg.create_alloc(1024)

    print("堆空间: {}".format(hex(addr)))

    asm_size = dbg.assemble_code_size("mov eax,1")
    print("汇编代码占用字节: {}".format(asm_size))

    write = dbg.assemble_write_memory(addr,"mov eax,1")

    byte_code = bytearray()

    for index in range(0,asm_size):
        read = dbg.read_memory_byte(addr + index)
        print("{:02x} ".format(read),end="")

    dbg.delete_alloc(addr)
```

封装上方代码，你就可以实现一个汇编指令获取工具了，如下`get_opcode_from_assemble()`函数。

```
from LyScript32 import MyDebug

# 传入汇编代码,得到对应机器码
def get_opcode_from_assemble(dbg_ptr,asm):
```

```
              pass

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    # 获取汇编代码
    byte_array = get_opcode_from_assemble(dbg,"xor eax,eax")
    for index in byte_array:
        print(hex(index),end="")
    print()

    # 汇编一个序列
    asm_list = ["xor eax,eax", "xor ebx,ebx", "mov eax,1"]
    for index in asm_list:
        byte_array = get_opcode_from_assemble(dbg, index)
        for index in byte_array:
            print(hex(index),end="")
        print()

    dbg.close()
```

实现劫持EIP指针: 这里我们演示一个案例，你可以自己实现一个`write_opcode_from_assemble()`函数批量将列表中的指令集写出到内存。

```
from LyScript32 import MyDebug

# 传入汇编指令列表,直接将机器码写入对端内存
def write_opcode_from_assemble(dbg_ptr,asm_list):
              pass

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 写出指令集到内存
    asm_list = ['mov eax,1','mov ebx,2','add eax,ebx']
    write_addr = write_opcode_from_assemble(dbg,asm_list)
    print("写出地址: {}".format(hex(write_addr)))

    # 设置执行属性
    dbg.set_local_protect(write_addr,32,1024)

    # 将EIP设置到指令集位置
    dbg.set_register("eip",write_addr)

    dbg.close()
```

如何执行函数呢？很简单，看以下代码是如何实现的，相信你能看懂，运行后会看到一个错误弹窗，说明程序执行流已经被转向了。

```
from LyScript32 import MyDebug
```

```
# 传入汇编指令列表,直接将机器码写入对端内存
def write_opcode_from_assemble(dbg_ptr,asm_list):
              pass

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 得到messagebox内存地址
    msg_ptr = dbg.get_module_from_function("user32.dll","MessageBoxA")
    call = "call {}".format(str(hex(msg_ptr)))
    print("函数地址: {}".format(call))

    # 写出指令集到内存
    asm_list = ['push 0','push 0','push 0','push 0',call]
    write_addr = write_opcode_from_assemble(dbg,asm_list)
    print("写出地址: {}".format(hex(write_addr)))

    # 设置执行属性
    dbg.set_local_protect(write_addr,32,1024)

    # 将EIP设置到指令集位置
    dbg.set_register("eip",write_addr)

    # 执行代码
    dbg.set_debug("Run")

    dbg.close()
```

得到_PEB_LDR_DATA线程环境块: 想要得到线程环境块最好的办法就是在目标内存中执行获取线程块的汇编指令，我们可以写出到内存并执行取值。

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug(address="127.0.0.1")
    dbg.connect()

    # 保存当前EIP
    eip = dbg.get_register("eip")

    # 创建堆
    heap_addres = dbg.create_alloc(1024)
    print("堆空间地址: {}".format(hex(heap_addres)))

    # 写出汇编指令
    # mov eax,fs:[0x30] 得到 _PEB
    dbg.assemble_at(heap_addres,"mov eax,fs:[0x30]")
    asmfs_size = dbg.get_disasm_operand_size(heap_addres)

    # 写出汇编指令
    # mov eax,[eax+0x0C] 得到 _PEB_LDR_DATA
    dbg.assemble_at(heap_addres + asmfs_size, "mov eax, [eax + 0x0C]")
    asmeax_size = dbg.get_disasm_operand_size(heap_addres + asmfs_size)
    # 跳转回EIP位置
    dbg.assemble_at(heap_addres+ asmfs_size + asmeax_size , "jmp
{}".format(hex(eip)))

    # 设置EIP到堆首地址
    dbg.set_register("eip",heap_addres)

    dbg.close()
```

内存字节变更后回写: 封装字节函数`write_opcode_list()`传入内存地址，对该地址中的字节更改后再回写到原来的位置。

- 加密算法在内存中会通过S盒展开解密，有时需要特殊需求，捕捉解密后的S-box写入内存，或对矩阵进行特殊处理后替换，这样写即可实现。

```
from LyScript32 import MyDebug

# 对每一个字节如何处理
def write_func(x):
    x = x + 10
    return x

# 对指定内存地址机器码进行相应处理
def write_opcode_list(dbg_ptr, address, count, function_ptr):
    for index in range(0, count):
        read = dbg_ptr.read_memory_byte(address + index)

        ref = function_ptr(read)
        print("处理后结果: {}".format(ref))

        dbg.write_memory_byte(address + index, ref)
    return True

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 得到EIP
    eip = dbg.get_register("eip")

    # 对指定内存中的数据+10后写回去
    write_opcode_list(dbg,eip,100,write_func)

    dbg.close()
```

将ShellCode从文本中注入到内存: 从文本中读取`ShellCode`代码，将该代码注入到目标进程的堆中，并把当前内存属性设置为可执行。

```
from LyScript32 import MyDebug

# 将shellcode读入内存
def read_shellcode(path):
    shellcode_list = []
    with open(path,"r",encoding="utf-8") as fp:
        for index in fp.readlines():
            shellcode_line = index.replace('"',"").replace("
","").replace("\n","").replace(";","")
            for code in shellcode_line.split("\\x"):
                if code != "" and code != "\\n":
                    shellcode_list.append("0x" + code)
    return shellcode_list

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 开辟堆空间
    address = dbg.create_alloc(1024)
    print("开辟堆空间: {}".format(hex(address)))
    if address == False:
        exit()

    # 设置内存可执行属性
    dbg.set_local_protect(address,32,1024)

    # 从文本中读取shellcode
    shellcode = read_shellcode("d://shellcode.txt")

    # 循环写入到内存
    for code_byte in range(0,len(shellcode)):
        bytef = int(shellcode[code_byte],16)
        dbg.write_memory_byte(code_byte + address, bytef)

    # 设置EIP位置
    dbg.set_register("eip",address)

    input()
    dbg.delete_alloc(address)

    dbg.close()
```

如果把这个过程反过来，就是将特定位置的汇编代码保存到本地。

```
from LyScript32 import MyDebug

# 将特定内存保存到文本中
def write_shellcode(dbg,address,size,path):
    with open(path,"a+",encoding="utf-8") as fp:
        for index in range(0, size - 1):
            # 读取机器码
            read_code = dbg.read_memory_byte(address + index)

            if (index+1) % 16 == 0:
                print("\\x" + str(read_code))
                fp.write("\\x" + str(read_code) + "\n")
            else:
                print("\\x" + str(read_code),end="")
                fp.write("\\x" + str(read_code))
```

```
if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")
    write_shellcode(dbg,eip,128,"d://lyshark.txt")
    dbg.close()
```

指令集探针快速检索: 快速检索当前程序中所有模块中是否存在特定的指令集片段，存在则返回内存地址。

```
from LyScript32 import MyDebug

# 将bytearray转为字符串
def get_string(byte_array):
    ref_string = str()
    for index in byte_array:
        ref_string = ref_string + "".join(str(index))
    return ref_string

# 传入汇编代码,得到对应机器码
def get_opcode_from_assemble(dbg_ptr,asm):
  pass

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    # 需要搜索的指令集片段
    search_asm = ['pop ecx','mov edi,edi', 'push eax', 'jmp esp']
    opcode = []

    # 将汇编指令转为机器码,放入opcode
    for index in range(len(search_asm)):
        byt = bytearray()
        byt = get_opcode_from_assemble(dbg, search_asm[index])
        opcode.append(get_string(byt))

    # 循环搜索指令集内存地址
    for index,entry in zip(range(0,len(opcode)), dbg.get_all_module()):
        eip = entry.get("entry")
        base_name = entry.get("name")
        if eip != 0:
            dbg.set_register("eip",eip)
            search_address = dbg.scan_memory_all(opcode[index])

            if search_address != False:
                print("指令: {} --> 模块: {} --> 个数: {}".
    format(search_asm[index],base_name,len(search_address)))

                for search_index in search_address:
                    print("[*] {}".format(hex(search_index)))
            else:
                print("a")
        time.sleep(0.3)
    dbg.close()
```

使用内置脚本得到返回值: 使用内置 `run_command_exec()` 函数时，用户只能得到内置脚本执行后的状态值，如果想要得到内置命令的返回值则你可以这样写。

```
from LyScript32 import MyDebug

dbg = MyDebug()
conn = dbg.connect()

# 首先定义一个脚本变量
ref = dbg.run_command_exec("$addr=1024")

# 将脚本返回值放到eax寄存器，或者开辟一个堆放到堆里
dbg.run_command_exec("eax=$addr")

# 最后拿到寄存器的值
hex(dbg.get_register("eax"))
```

通过中转的方式，可以很好的得到内置脚本的返回值，如下我将其封装成了一个独立的方法。

```
from LyScript32 import MyDebug

# 得到脚本返回值
def GetScriptValue(dbg,script):
    try:
        ref = dbg.run_command_exec("push eax")
        if ref != True:
            return None
        ref = dbg.run_command_exec(f"eax={script}")
        if ref != True:
            return None
        reg = dbg.get_register("eax")
        ref = dbg.run_command_exec("pop eax")
        if ref != True:
            return None
        return reg
    except Exception:
        return None
    return None

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eax = "401000"
    ref = GetScriptValue(dbg,"mod.base({})".format(eax))
    print(hex(ref))

    dbg.close()
```

获取节表内存属性: 默认情况下插件可以得到当前节表，但无法直接取到节表内存属性，如果需要得到某个节的内存属性，可以这样写。

节表属性是一个数字，数字含义如下:

- ER 执行/读取 32
- E 执行 30
- R 读取 2
- W 写入 2
- C 未知 4
- N 空属性 1

```python
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    conn = dbg.connect()

    section = dbg.get_section()
    for i in section:
        section_address = hex(i.get("addr"))
        section_name = i.get("name")
        section_size = i.get("size")
        section_type = dbg.get_local_protect(i.get("addr"))
        print(f"地址: {section_address} -> 名字: {section_name} -> 大小:
{section_size}bytes -> ",end="")

        if(section_type == 32):
            print("属性: 执行/读取")
        elif(section_type == 30):
            print("属性: 执行")
        elif(section_type == 2):
            print("属性: 只读")
        elif(section_type == 4):
            print("属性: 读写")
        elif(section_type == 1):
            print("属性: 空属性")
        else:
            print("属性: 读/写/控制")

    dbg.close()
    pass
```

封装获取反汇编代码: 反汇编时可以使用`get_disasm_code()`函数，但如果想要自己实现可以这样写。

```python
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    conn = dbg.connect()

    # 获取当前EIP地址
    eip = dbg.get_register("eip")
    print("eip = {}".format(hex(eip)))

    # 向下反汇编字节数
    count = eip + 15
    while True:
        # 每次得到一条反汇编指令
        dissasm = dbg.get_disasm_one_code(eip)

        print("0x{:08x} | {}".format(eip, dissasm))

        # 判断是否满足退出条件
        if eip >= count:
            break
        else:
            # 得到本条反汇编代码的长度
            dis_size = dbg.assemble_code_size(dissasm)
            eip = eip + dis_size

    dbg.close()
    pass
```

得到当前EIP机器码: 获取到当前EIP指针所在位置的机器码，你可以灵活运用反汇编代码的组合实现。

```
from LyScript32 import MyDebug

# 得到机器码
def GetHexCode(dbg,address):
    ref_bytes = []
    # 首先得到反汇编指令,然后得到该指令的长度
    asm_len = dbg.assemble_code_size( dbg.get_disasm_one_code(address) )

    # 循环得到每个机器码
    for index in range(0,asm_len):
        ref_bytes.append(dbg.read_memory_byte(address))
        address = address + 1
    return ref_bytes

if __name__ == "__main__":
    dbg = MyDebug()
    conn = dbg.connect()

    # 获取当前EIP地址
    eip = dbg.get_register("eip")
    print("eip = {}".format(hex(eip)))

    # 得到机器码
    ref = GetHexCode(dbg,eip)
    for i in range(0,len(ref)):
        print("0x{:02x} ".format(ref[i]),end="")

    dbg.close()
    pass
```

如果将如上的两个方法结合起来，那么基本上就实现了获取反汇编窗口中的所有内容。

```
from LyScript32 import MyDebug

# 得到机器码
```

```
def GetHexCode(dbg,address):
    ref_bytes = []
    # 首先得到反汇编指令,然后得到该指令的长度
    asm_len = dbg.assemble_code_size( dbg.get_disasm_one_code(address) )
    # 循环得到每个机器码
    for index in range(0,asm_len):
        ref_bytes.append(dbg.read_memory_byte(address))
        address = address + 1
    return ref_bytes

if __name__ == "__main__":
    dbg = MyDebug()
    conn = dbg.connect()

    # 获取当前EIP地址
    eip = dbg.get_register("eip")
    print("eip = {}".format(hex(eip)))

    # 向下反汇编字节数
    count = eip + 20
    while True:
        # 每次得到一条反汇编指令
        dissasm = dbg.get_disasm_one_code(eip)

        print("0x{:08x} | {:50} | ".format(eip, dissasm),end="")

        # 得到机器码
        ref = GetHexCode(dbg, eip)
        for i in range(0, len(ref)):
            print("0x{:02x} ".format(ref[i]), end="")
        print()

        # 判断是否满足退出条件
        if eip >= count:
            break
        else:
            # 得到本条反汇编代码的长度
            dis_size = dbg.assemble_code_size(dissasm)
            eip = eip + dis_size

    dbg.close()
    pass
```

封装汇编指令提取器: 当用户传入指定汇编指令的时候，自动的将其转换成对应的机器码，代码很简单，首先`dbg.create_alloc(1024)`在进程内存中开辟堆空间，用于存放我们的机器码，然后调用`dbg.assemble_write_memory(alloc_address,"sub esp,10")`将一条汇编指令变成机器码写到对端内存，然后再`op = dbg.read_memory_byte(alloc_address + index)`依次将其读取出来即可。

```
from LyScript32 import MyDebug

# 传入汇编指令,获取该指令的机器码
def get_assembly_machine_code(dbg,asm):
    pass
```

```
if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    machine_code_list = []

    # 开辟堆空间
    alloc_address = dbg.create_alloc(1024)
    print("分配堆: {}".format(hex(alloc_address)))

    # 得到汇编机器码
    machine_code = dbg.assemble_write_memory(alloc_address,"sub esp,10")
    if machine_code == False:
        dbg.delete_alloc(alloc_address)

    # 得到汇编指令长度
    machine_code_size = dbg.assemble_code_size("sub esp,10")
    if machine_code == False:
        dbg.delete_alloc(alloc_address)

    # 读取机器码
    for index in range(0,machine_code_size):
        op = dbg.read_memory_byte(alloc_address + index)
        machine_code_list.append(op)

    # 释放堆空间
    dbg.delete_alloc(alloc_address)

    # 输出机器码
    print(machine_code_list)
    dbg.close()
```

我们继续封装如上方法，封装成一个可以直接使用的`get_assembly_machine_code`函数。

```
from LyScript32 import MyDebug

# 传入汇编指令,获取该指令的机器码
def get_assembly_machine_code(dbg,asm):
    machine_code_list = []

    # 开辟堆空间
    alloc_address = dbg.create_alloc(1024)
    print("分配堆: {}".format(hex(alloc_address)))

    # 得到汇编机器码
    machine_code = dbg.assemble_write_memory(alloc_address,asm)
    if machine_code == False:
        dbg.delete_alloc(alloc_address)

    # 得到汇编指令长度
    machine_code_size = dbg.assemble_code_size(asm)
    if machine_code == False:
        dbg.delete_alloc(alloc_address)
    # 读取机器码
    for index in range(0,machine_code_size):
        op = dbg.read_memory_byte(alloc_address + index)
        machine_code_list.append(op)

    # 释放堆空间
    dbg.delete_alloc(alloc_address)
    return machine_code_list

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    # 转换第一对
    opcode = get_assembly_machine_code(dbg,"mov eax,1")
    for index in opcode:
        print("0x{:02X} ".format(index),end="")
    print()

    # 转换第二对
    opcode = get_assembly_machine_code(dbg,"sub esp,10")
    for index in opcode:
        print("0x{:02X} ".format(index),end="")
    print()

    dbg.close()
```

扫描符合条件的内存: 通过使用上方封装的`get_assembly_machine_code()`并配合`scan_memory_one(scan_string)`函数，在对端内存搜索是否存在符合条件的指令。

```
from LyScript32 import MyDebug

# 传入汇编指令,获取该指令的机器码
def get_assembly_machine_code(dbg,asm):
    machine_code_list = []

    # 开辟堆空间
    alloc_address = dbg.create_alloc(1024)
    print("分配堆: {}".format(hex(alloc_address)))

    # 得到汇编机器码
    machine_code = dbg.assemble_write_memory(alloc_address,asm)
    if machine_code == False:
        dbg.delete_alloc(alloc_address)

    # 得到汇编指令长度
    machine_code_size = dbg.assemble_code_size(asm)
    if machine_code == False:
        dbg.delete_alloc(alloc_address)

    # 读取机器码
    for index in range(0,machine_code_size):
        op = dbg.read_memory_byte(alloc_address + index)
        machine_code_list.append(op)
    # 释放堆空间
    dbg.delete_alloc(alloc_address)
    return machine_code_list

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    for item in ["push eax","mov eax,1","jmp eax","pop eax"]:
        # 转换成列表
        opcode = get_assembly_machine_code(dbg,item)
        #print("得到机器码列表: ",opcode)

        # 列表转换成字符串
        scan_string = " ".join([str(_) for _ in opcode])
        #print("搜索机器码字符串: ", scan_string)

        address = dbg.scan_memory_one(scan_string)
        print("第一个符合条件的内存块: {}".format(hex(address)))

    dbg.close()
```

获取下一条汇编指令: 下一条汇编指令的获取需要注意如果是被命中的指令则此处应该是CC断点占用一个字节，如果不是则正常获取到当前指令即可。

- 1.我们需要检查当前内存断点是否被命中，如果没有命中则说明此处我们需要获取到原始的汇编指令长度，然后与当前eip地址相加获得。
- 2.如果命中了断点，则此处有两种情况
    - 1.1 如果是用户下的断点，则此处调试器会在指令位置替换为CC，也就是汇编中的init停机指令，该指令占用1个字节，需要eip+1得到。
    - 1.2 如果是系统断点，EIP所停留的位置，则我们需要正常获取当前指令地址，此处调试器没有改动汇编指令仅仅只下下了异常断点。

```
from LyScript32 import MyDebug

# 获取当前EIP指令的下一条指令
def get_disasm_next(dbg,eip):
    next = 0

    # 检查当前内存地址是否被下了绊子
    check_breakpoint = dbg.check_breakpoint(eip)

    # 说明存在断点，如果存在则这里就是一个字节了
    if check_breakpoint == True:

        # 接着判断当前是否是EIP，如果是EIP则需要使用原来的字节
        local_eip = dbg.get_register("eip")

        # 说明是EIP并且命中了断点
        if local_eip == eip:
            dis_size = dbg.get_disasm_operand_size(eip)
            next = eip + dis_size
```

```
                next_asm = dbg.get_disasm_one_code(next)
                return next_asm
            else:
                next = eip + 1
                next_asm = dbg.get_disasm_one_code(next)
                return next_asm
            return None

        # 不是则需要获取到原始汇编代码的长度
        elif check_breakpoint == False:
            # 得到当前指令长度
            dis_size = dbg.get_disasm_operand_size(eip)
            next = eip + dis_size
            next_asm = dbg.get_disasm_one_code(next)
            return next_asm
        else:
            return None

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    next = get_disasm_next(dbg,eip)
    print("下一条指令: {}".format(next))

    prev = get_disasm_next(dbg,12391436)
    print("下一条指令: {}".format(prev))

    dbg.close()
```

获取上一条汇编指令: 上一条指令的获取难点就在于，我们无法确定当前指令的上一条指令到底有多长，所以只能用笨办法，逐行扫描对比汇编指令，如果找到则取出其上一条指令即可。

```
from LyScript32 import MyDebug

# 获取当前EIP指令的上一条指令
def get_disasm_prev(dbg,eip):
    prev_dasm = None
    # 得到当前汇编指令
    local_disasm = dbg.get_disasm_one_code(eip)

    # 只能向上扫描10行
    eip = eip - 10
    disasm = dbg.get_disasm_code(eip,10)

    # 循环扫描汇编代码
    for index in range(0,len(disasm)):
        # 如果找到了,就取出他的上一个汇编代码
        if disasm[index].get("opcode") == local_disasm:
            prev_dasm = disasm[index-1].get("opcode")
            break

    return prev_dasm
```

```python
if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    next = get_disasm_prev(dbg,eip)
    print("上一条指令: {}".format(next))

    dbg.close()
```

判断是否为特定指令: 判断当前传入的内存地址处的汇编指令集，看是否是`call,nop,jmp,ret`指令，如果是返回True否则返回False。

```python
from LyScript32 import MyDebug

# 是否是跳转指令
def is_call(dbg,address):
    try:
        dis = dbg.get_disasm_one_code(address)
        if dis != False or dis != None:
            if dis.split(" ")[0].replace(" ","") == "call":
                return True
            return False
        return False
    except Exception:
        return False
    return False

# 是否是jmp
def is_jmp(dbg,address):
    try:
        dis = dbg.get_disasm_one_code(address)
        if dis != False or dis != None:
            if dis.split(" ")[0].replace(" ","") == "jmp":
                return True
            return False
        return False
    except Exception:
        return False
    return False

# 是否是ret
def is_ret(dbg,address):
    try:
        dis = dbg.get_disasm_one_code(address)
        if dis != False or dis != None:
            if dis.split(" ")[0].replace(" ","") == "ret":
                return True
            return False
        return False
    except Exception:
        return False
    return False
```

```
# 是否是nop
def is_nop(dbg,address):
    try:
        dis = dbg.get_disasm_one_code(address)
        if dis != False or dis != None:
            if dis.split(" ")[0].replace(" ","") == "nop":
                return True
            return False
        return False
    except Exception:
        return False
    return False

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    call = is_call(dbg, eip)
    print("是否是call指令: {}".format(call))
    dbg.close()
```

同如上所示，我们也可以验证指令集是否是条件跳转命令，验证方式只需要稍微改动代码即可。

```
from LyScript32 import MyDebug

# 是否是条件跳转指令
def is_cond(dbg,address):
    try:
        dis = dbg.get_disasm_one_code(address)
        if dis != False or dis != None:

            if dis.split(" ")[0].replace(" ","")
            in
["je","jne","jz","jnz","ja","jna","jp","jnp","jb","jnb","jg","jng","jge","jl","j
le"]:
                return True
            return False
        return False
    except Exception:
        return False
    return False

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    call = is_cond(dbg, eip)
    print("是否是条件跳转指令: {}".format(call))
    dbg.close()
```

内存区域交换与差异对比: 内存交换可调用`memory_xchage`将两个内存颠倒顺序，内存差异对比可调用`memory_cmp`对比内存差异并返回列表。

```
from LyScript32 import MyDebug

# 交换两个内存区域
def memory_xchage(dbg,memory_ptr_x,memory_ptr_y,bytes):
    ref = False
    for index in range(0,bytes):
        # 读取两个内存区域
        read_byte_x = dbg.read_memory_byte(memory_ptr_x + index)
        read_byte_y = dbg.read_memory_byte(memory_ptr_y + index)

        # 交换内存
        ref = dbg.write_memory_byte(memory_ptr_x + index,read_byte_y)
        ref = dbg.write_memory_byte(memory_ptr_y + index, read_byte_x)
    return ref

# 对比两个内存区域
def memory_cmp(dbg,memory_ptr_x,memory_ptr_y,bytes):
    cmp_memory = []
    for index in range(0,bytes):

        item = {"addr":0, "x": 0, "y": 0}

        # 读取两个内存区域
        read_byte_x = dbg.read_memory_byte(memory_ptr_x + index)
        read_byte_y = dbg.read_memory_byte(memory_ptr_y + index)

        if read_byte_x != read_byte_y:
            item["addr"] = memory_ptr_x + index
            item["x"] = read_byte_x
            item["y"] = read_byte_y
            cmp_memory.append(item)
    return cmp_memory

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    # 内存交换
    flag = memory_xchage(dbg, 12386320,12386352,4)
    print("内存交换状态: {}".format(flag))

    # 内存对比
    cmp_ref = memory_cmp(dbg, 12386320,12386352,4)
    for index in range(0,len(cmp_ref)):
        print("地址: 0x{:08X} -> X: 0x{:02x} -> y: 0x{:02x}"

.format(cmp_ref[index].get("addr"),cmp_ref[index].get("x"),cmp_ref[index].get("y
")))

    dbg.close()
```

内存与磁盘机器码比较: 通过调用`read_memory_byte()`函数，或者`open()`打开文件，等就可以得到程序磁盘与内存中特定位置的机器码参数，然后通过对每一个列表中的字节进行比较，就可得到特定位置下磁盘与内存中的数据是否一致的判断。

```
#coding: utf-8
import binascii,os,sys
from LyScript32 import MyDebug

# 得到程序的内存镜像中的机器码
def get_memory_hex_ascii(address,offset,len):
    count = 0
    ref_memory_list = []
    for index in range(offset,len):
        # 读出数据
        char = dbg.read_memory_byte(address + index)
        count = count + 1

        if count % 16 == 0:
            if (char) < 16:
                print("0" + hex((char))[2:])
                ref_memory_list.append("0" + hex((char))[2:])
            else:
                print(hex((char))[2:])
                ref_memory_list.append(hex((char))[2:])
        else:
            if (char) < 16:
                print("0" + hex((char))[2:] + " ",end="")
                ref_memory_list.append("0" + hex((char))[2:])
            else:
                print(hex((char))[2:] + " ",end="")
                ref_memory_list.append(hex((char))[2:])
    return ref_memory_list

# 读取程序中的磁盘镜像中的机器码
def get_file_hex_ascii(path,offset,len):
    count = 0
    ref_file_list = []

    with open(path, "rb") as fp:
        # file_size = os.path.getsize(path)
        fp.seek(offset)

        for item in range(offset,offset + len):
            char = fp.read(1)
            count = count + 1
            if count % 16 == 0:
                if ord(char) < 16:
                    print("0" + hex(ord(char))[2:])
                    ref_file_list.append("0" + hex(ord(char))[2:])
                else:
                    print(hex(ord(char))[2:])
                    ref_file_list.append(hex(ord(char))[2:])
            else:
                if ord(char) < 16:
                    print("0" + hex(ord(char))[2:] + " ", end="")
                    ref_file_list.append("0" + hex(ord(char))[2:])
                else:
                    print(hex(ord(char))[2:] + " ", end="")
                    ref_file_list.append(hex(ord(char))[2:])
    return ref_file_list

if __name__ == "__main__":
    dbg = MyDebug()

    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    module_base = dbg.get_base_from_address(dbg.get_local_base())
    print("模块基地址: {}".format(hex(module_base)))

    # 得到内存机器码
    memory_hex_byte = get_memory_hex_ascii(module_base,0,1024)

    # 得到磁盘机器码
    file_hex_byte = get_file_hex_ascii("d://Win32Project1.exe",0,1024)

    # 输出机器码
    for index in range(0,len(memory_hex_byte)):
        # 比较磁盘与内存是否存在差异
        if memory_hex_byte[index] != file_hex_byte[index]:
            # 存在差异则输出
            print("\n相对位置: [{}] --> 磁盘字节: 0x{} --> 内存字节: 0x{}".
                  format(index,memory_hex_byte[index],file_hex_byte[index]))
    dbg.close()
```

内存ASCII码解析: 通过封装的`get_memory_hex_ascii`得到内存机器码，然后再使用如下过程实现输出该内存中的机器码所对应的ASCII码。

```
from LyScript32 import MyDebug
import os,sys

# 转为ascii
def to_ascii(h):
    list_s = []
    for i in range(0, len(h), 2):
        list_s.append(chr(int(h[i:i+2], 16)))
    return ''.join(list_s)

# 转为16进制
def to_hex(s):
    list_h = []
    for c in s:
        list_h.append(hex(ord(c))[2:])
    return ''.join(list_h)

# 得到程序的内存镜像中的机器码
def get_memory_hex_ascii(address,offset,len):
    count = 0
    ref_memory_list = []
    for index in range(offset,len):
        # 读出数据
        char = dbg.read_memory_byte(address + index)
        count = count + 1

        if count % 16 == 0:
            if (char) < 16:
                ref_memory_list.append("0" + hex((char))[2:])
            else:
                ref_memory_list.append(hex((char))[2:])
        else:
            if (char) < 16:
                ref_memory_list.append("0" + hex((char))[2:])
            else:
                ref_memory_list.append(hex((char))[2:])
    return ref_memory_list


if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    # 得到模块基地址
    module_base = dbg.get_base_from_address(dbg.get_local_base())

    # 得到指定区域内存机器码
    ref_memory_list = get_memory_hex_ascii(module_base,0,1024)

    # 解析ascii码
    break_count = 1
    for index in ref_memory_list:
        if break_count %32 == 0:
            print(to_ascii(hex(int(index, 16))[2:]))
        else:
            print(to_ascii(hex(int(index, 16))[2:]),end="")
        break_count = break_count + 1

    dbg.close()
```

内存特征码匹配: 通过二次封装`get_memory_hex_ascii()`实现扫描内存特征码功能，如果存在则返回True否则返回False。

```
from LyScript32 import MyDebug
import os,sys

# 得到程序的内存镜像中的机器码
def get_memory_hex_ascii(address,offset,len):
    count = 0
    ref_memory_list = []
    for index in range(offset,len):
        # 读出数据
        char = dbg.read_memory_byte(address + index)
        count = count + 1
```

```
            if count % 16 == 0:
                if (char) < 16:
                    ref_memory_list.append("0" + hex((char))[2:])
                else:
                    ref_memory_list.append(hex((char))[2:])
            else:
                if (char) < 16:
                    ref_memory_list.append("0" + hex((char))[2:])
                else:
                    ref_memory_list.append(hex((char))[2:])
        return ref_memory_list


# 在指定区域内搜索特定的机器码,如果完全匹配则返回
def search_hex_ascii(address,offset,len,hex_array):
    # 得到指定区域内存机器码
    ref_memory_list = get_memory_hex_ascii(address,offset,len)

    array = []

    # 循环输出字节
    for index in range(0,len + len(hex_array)):

        # 如果有则继续装
        if len(hex_array) != len(array):
            array.append(ref_memory_list[offset + index])

        else:
            for y in range(0,len(array)):
                if array[y] != ref_memory_list[offset + index + y]:
                    return False

        array.clear()
    return False

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    eip = dbg.get_register("eip")

    # 得到模块基地址
    module_base = dbg.get_base_from_address(dbg.get_local_base())

    re = search_hex_ascii(module_base,0,100,hex_array=["0x4d","0x5a"])

    dbg.close()
```

特征码扫描一般不需要自己写，自己写的麻烦，而且不支持通配符，可以直接调用我们API中封装好的`scan_memory_one()`它可以支持`??`通配符模糊匹配，且效率要高许多。

堆栈地址有符号无符号转换: peek_stack命令传入的是堆栈下标位置默认从0开始，并输出一个`十进制有符号`长整数，首先实现有符号与无符号数之间的转换操作，为后续堆栈扫描做准备。

```
from LyScript32 import MyDebug

# 有符号整数转无符号数
def long_to_ulong(inter,is_64 = False):
    if is_64 == False:
        return inter & ((1 << 32) - 1)
    else:
        return inter & ((1 << 64) - 1)

# 无符号整数转有符号数
def ulong_to_long(inter,is_64 = False):
    if is_64 == False:
        return (inter & ((1 << 31) - 1)) - (inter & (1 << 31))
    else:
        return (inter & ((1 << 63) - 1)) - (inter & (1 << 63))

if __name__ == "__main__":
    dbg = MyDebug()

    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    for index in range(0,10):

        # 默认返回有符号数
        stack_address = dbg.peek_stack(index)

        # 使用转换
        print("默认有符号数: {:15} --> 转为无符号数: {:15} --> 转为有符号数: {:15}".
              format(stack_address,
            long_to_ulong(stack_address),
            ulong_to_long(long_to_ulong(stack_address))))

    dbg.close()
```

扫描堆栈并反汇编一条: 我们使用`get_disasm_one_code()`函数，扫描堆栈地址并得到该地址处的第一条反汇编代码。

```
from LyScript32 import MyDebug

# 有符号整数转无符号数
def long_to_ulong(inter,is_64 = False):
    if is_64 == False:
        return inter & ((1 << 32) - 1)
    else:
        return inter & ((1 << 64) - 1)

# 无符号整数转有符号数
def ulong_to_long(inter,is_64 = False):
    if is_64 == False:
        return (inter & ((1 << 31) - 1)) - (inter & (1 << 31))
    else:
        return (inter & ((1 << 63) - 1)) - (inter & (1 << 63))

if __name__ == "__main__":
```

```python
    dbg = MyDebug()

    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    for index in range(0,10):

        # 默认返回有符号数
        stack_address = dbg.peek_stack(index)

        # 反汇编一行
        dasm = dbg.get_disasm_one_code(stack_address)

        # 根据地址得到模块基址
        if stack_address <= 0:
            mod_base = 0
        else:
            mod_base = dbg.get_base_from_address(long_to_ulong(stack_address))

        print("stack => [{}] addr = {:10} base = {:10} dasm = {}".
        format(index, hex(long_to_ulong(stack_address)),hex(mod_base), dasm))

    dbg.close()
```

得到堆栈返回模块基址: 首先我们需要得到程序全局状态下的所有加载模块的基地址，然后得到当前堆栈内存地址内的实际地址，并通过实际内存地址得到模块基地址，对比全局表即可拿到当前模块是返回到了哪里。

```python
from LyScript32 import MyDebug

# 有符号整数转无符号数
def long_to_ulong(inter,is_64 = False):
    if is_64 == False:
        return inter & ((1 << 32) - 1)
    else:
        return inter & ((1 << 64) - 1)

# 无符号整数转有符号数
def ulong_to_long(inter,is_64 = False):
    if is_64 == False:
        return (inter & ((1 << 31) - 1)) - (inter & (1 << 31))
    else:
        return (inter & ((1 << 63) - 1)) - (inter & (1 << 63))

if __name__ == "__main__":
    dbg = MyDebug()

    connect_flag = dbg.connect()
    print("连接状态: {}".format(connect_flag))

    # 得到程序加载过的所有模块信息
    module_list = dbg.get_all_module()

    # 向下扫描堆栈
    for index in range(0,10):
        # 默认返回有符号数
        stack_address = dbg.peek_stack(index)

        # 反汇编一行
        dasm = dbg.get_disasm_one_code(stack_address)

        # 根据地址得到模块基址
        if stack_address <= 0:
            mod_base = 0
        else:
            mod_base = dbg.get_base_from_address(long_to_ulong(stack_address))

        # print("stack => [{}] addr = {:10} base = {:10} dasm = {}"
        .format(index, hex(long_to_ulong(stack_address)),hex(mod_base), dasm))
        if mod_base > 0:
            for x in module_list:
                if mod_base == x.get("base"):
                    print("stack => [{}] addr = {:10} base = {:10} dasm = {:15}
return = {:10}"

.format(index,hex(long_to_ulong(stack_address)),hex(mod_base), dasm,
                                  x.get("name")))

    dbg.close()
```

第三方反汇编库应用: 通过LyScript插件读取出内存中的机器码，然后交给`capstone`反汇编库执行，并将结果输出成字典格式。

```
#coding: utf-8
import binascii,os,sys
import pefile
from capstone import *
from LyScript32 import MyDebug

# 得到内存反汇编代码
def get_memory_disassembly(address,offset,len):
    # 反汇编列表
    dasm_memory_dict = []

    # 内存列表
    ref_memory_list = bytearray()

    # 读取数据
    for index in range(offset,len):
        char = dbg.read_memory_byte(address + index)
        ref_memory_list.append(char)

    # 执行反汇编
    md = Cs(CS_ARCH_X86,CS_MODE_32)
    for item in md.disasm(ref_memory_list,0x1):
        addr = int(pe_base) + item.address
        dasm_memory_dict.append({"address": str(addr), "opcode": item.mnemonic +
" " + item.op_str})
    return dasm_memory_dict
```

```python
if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    pe_base = dbg.get_local_base()
    pe_size = dbg.get_local_size()

    print("模块基地址: {}".format(hex(pe_base)))
    print("模块大小: {}".format(hex(pe_size)))

    # 得到内存反汇编代码
    dasm_memory_list = get_memory_disassembly(pe_base,0,pe_size)
    print(dasm_memory_list)

    dbg.close()
```

如果我们将磁盘文件也反汇编到一个列表内，然后对比两者就实现了一个简易版应用层钩子扫描器。

```python
#coding: utf-8
import binascii,os,sys
import pefile
from capstone import *
from LyScript32 import MyDebug

# 得到内存反汇编代码
def get_memory_disassembly(address,offset,len):
    # 反汇编列表
    dasm_memory_dict = []

    # 内存列表
    ref_memory_list = bytearray()

    # 读取数据
    for index in range(offset,len):
        char = dbg.read_memory_byte(address + index)
        ref_memory_list.append(char)

    # 执行反汇编
    md = Cs(CS_ARCH_X86,CS_MODE_32)
    for item in md.disasm(ref_memory_list,0x1):
        addr = int(pe_base) + item.address
        dic = {"address": str(addr), "opcode": item.mnemonic + " " +
item.op_str}
        dasm_memory_dict.append(dic)
    return dasm_memory_dict

# 反汇编文件中的机器码
def get_file_disassembly(path):
    opcode_list = []
    pe = pefile.PE(path)
    ImageBase = pe.OPTIONAL_HEADER.ImageBase

    for item in pe.sections:
        if str(item.Name.decode('UTF-8').strip(b'\x00'.decode())) == ".text":
            # print("虚拟地址: 0x%.8X 虚拟大小: 0x%.8X" %
(item.VirtualAddress,item.Misc_VirtualSize))
            VirtualAddress = item.VirtualAddress
            VirtualSize = item.Misc_VirtualSize
            ActualOffset = item.PointerToRawData
    StartVA = ImageBase + VirtualAddress
    StopVA = ImageBase + VirtualAddress + VirtualSize
    with open(path,"rb") as fp:
        fp.seek(ActualOffset)
        HexCode = fp.read(VirtualSize)

    md = Cs(CS_ARCH_X86, CS_MODE_32)
    for item in md.disasm(HexCode, 0):
        addr = hex(int(StartVA) + item.address)
        dic = {"address": str(addr) , "opcode": item.mnemonic + " " +
item.op_str}
        # print("{}".format(dic))
        opcode_list.append(dic)
    return opcode_list

if __name__ == "__main__":
    dbg = MyDebug()
    dbg.connect()

    pe_base = dbg.get_local_base()
    pe_size = dbg.get_local_size()

    print("模块基地址: {}".format(hex(pe_base)))
    print("模块大小: {}".format(hex(pe_size)))

    # 得到内存反汇编代码
    dasm_memory_list = get_memory_disassembly(pe_base,0,pe_size)
    dasm_file_list = get_file_disassembly("d://win32project1.exe")

    # 循环对比内存与文件中的机器码
    for index in range(0,len(dasm_file_list)):
        if dasm_memory_list[index] != dasm_file_list[index]:
            print("地址: {:8} --> 内存反汇编: {:32} --> 磁盘反汇编: {:32}".
                  format(dasm_memory_list[index].get("address"),

dasm_memory_list[index].get("opcode"),dasm_file_list[index].get("opcode")))
    dbg.close()
```