Script 脚本类

脚本类的功能实现都是调用的x64dbg命令，目前由于`run_command_exec()`命令无法返回参数，故通过中转eax寄存器实现了取值，目前只能取出整数类型的参数。

脚本类功能说明来源于：iBinary 的博客

Script 类内函数名	函数作用
party(addr)	获取模块的模式编号, addr = 0则是用户模块,1则是系统模块
base(addr)	获取模块基址
size(addr)	返回模块大小
hash(addr)	返回模块hash
entry(addr)	返回模块入口
system(addr)	如果addr是系统模块则为true否则则是false
user(addr)	如果是用户模块则返回true 否则为false
main()	返回主模块基地址
rva(addr)	如果addr不在模块则返回0,否则返回addr所位于模块的RVA偏移
offset(addr)	获取地址所对应的文件偏移量,如果不在模块则返回0
isexport(addr)	判断该地址是否是从模块导出的函数
valid(addr)	判断addr是否有效,有效则返回True
base(addr)	或者当前addr的基址
size(addr)	获取当前addr内存的大小
iscode(addr)	判断当前 addr是否是可执行页面,成功返回TRUE
decodepointer(ptr)	解密指针,相当于调用了DecodePointer ptr
ReadByte(addr/eg)	从addr或者寄存器中读取一个字节内存并且返回
Byte(addr)	从addr或者寄存器中读取一个字节内存并且返回
ReadWord(addr)	读取两个字节
ReadDDword(addr)	读取四个字节
ReadQword(addr)	读取8个字节,但是只能是64位程序方可使用
ReadPtr(addr)	从地址中读取指针(4/8字节)并返回读取的指针值
ReadPointer(addr)	从地址中读取指针(4/8字节)并返回读取的指针值
len(addr)	获取addr处的指令长度

Script 类内函数名	函数作用
iscond(addr)	判断当前addr位置是否是条件指令
isbranch(addr)	判断当前地址是否是分支指令
isret(addr)	判断是否是ret指令
iscall(addr)	判断是否是call指令
ismem(addr)	判断是否是内存操作数
isnop(addr)	判断是否是nop
isunusual(addr)	判断当前地址是否指示为异常地址
branchdest(addr)	将指令的分支目标位于addr处
branchexec(addr)	如果分支要执行
imm(addr)	获取当前指令位置的立即数
brtrue(addr)	下一条指令的地址
next(addr)	获取addr的下一条地址
prev(addr)	获取addr上一条低地址
iscallsystem(addr)	判断当前指令是否是系统模块指令
get(index)	获取当前函数堆栈中的第index个参数
set(index,value)	设置的索引位置的值
firstchance()	最后一个异常是否为第一次机会异常
addr()	最后一个异常地址
code()	最后一个异常代码
flags()	最后一个异常标志
infocount()	上次异常信息计数
info(index)	最后一个异常信息

如上是一些常用的脚本命令的封装，他们的调用方式如下面代码中所示。

```
from LyScript32 import MyDebug
from LyScriptTools32 import DebugControl
from LyScriptTools32 import Script

# 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))

    # 定义调试类与脚本类
    control = DebugControl(dbg)
    script = Script(dbg)

    # 得到EIP
    eip = control.get_eip()

    size = script.size(eip)
    print("当前模块大小: {}".format(hex(size)))

    entry = script.entry(eip)
    print("当前模块入口: {}".format(hex(entry)))

    # 得到hash值,默认有符号需要转换
    hash = script.hash(eip)
    print("有符号hash值: {}".format(hash))

    hash = long_to_ulong(script.hash(eip))
    print("无符号hash值: {}".format(hex(hash)))

    dbg.close()
```

如果觉得上面的函数封装不够，或自己需要调用特定命令，那么可以直接调用该类内的`script.GetScriptValue("")`方法，自定义一个参数传递，目前只能接受返回值是整数的命令。

```
from LyScript32 import MyDebug
from LyScriptTools32 import DebugControl
from LyScriptTools32 import Script

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

    # 定义控制类与脚本类
    control = DebugControl(dbg)
    script = Script(dbg)

    # 得到EIP
    eip = control.get_eip()

    # 调用脚本命令执行函数
```

```
    ref = script.GetScriptValue("mod.size(eip)")
    print("模块返回值: {}".format(hex(ref)))

    dbg.close()
```

Stack 堆栈类

Stack 类是通过二次封装stack堆栈操作API函数得到的，并在此基础上增加了全新的调用函数。

Stack 类内函数名	函数作用
create_alloc(decimal_size=1024)	开辟堆,传入长度,默认1024字节
delete_alloc(decimal_address=0)	销毁一个远程堆
push_stack(decimal_value=0)	将传入参数入栈
pop_stack()	从栈顶弹出元素,默认检查栈顶,可传入参数
peek_stack(decimal_index=0)	检查指定位置栈针中的地址,返回一个地址
peek_stack_list(decimal_count=0)	检查指定位置处前index个栈针中的地址,返回一个地址列表
get_current_stack_top()	获取当前栈帧顶部内存地址
get_current_stack_bottom()	获取当前栈帧底部内存地址
get_current_stackframe_size()	获取当前栈帧长度
get_stack_frame_list(decimal_count=0)	获取index指定的栈帧内存地址,返回列表
get_current_stack_disassemble()	堆当前栈地址反汇编
get_current_stack_frame_disassemble()	对当前栈帧地址反汇编
get_current_stack_base()	得到当前栈地址的基地址
get_current_stack_return_name()	得到当前栈地址返回到的模块名
get_current_stack_return_size()	得到当前栈地址返回到的模块大小
get_current_stack_return_entry()	得到当前栈地址返回到的模块入口
get_current_stack_return_base()	得到当前栈地址返回到的模块基地址

我们以输出当前堆栈中模块地址为例，演示堆栈类如何使用。

```
from LyScript32 import MyDebug
from LyScriptTools32 import Stack
from LyScriptTools32 import DebugControl

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

```
print("连接状态: {}".format(connect_flag))

# 定义堆栈类
control = DebugControl(dbg)
stack = Stack(dbg)

# 得到EIP
eip = control.get_eip()

return_name = stack.get_current_stack_return_name()
print("堆栈返回到: {}".format(return_name))

return_size = stack.get_current_stack_return_size()
print("堆栈返回大小: {}".format(return_size))

return_entry = stack.get_current_stack_return_entry()
print("堆栈返回模块的入口: {}".format(hex(return_entry)))

return_base = stack.get_current_stack_return_base()
print("堆栈返回模块入口基地址: {}".format(hex(return_base)))

dbg.close()
```

Module 模块类

Module 类主要封装自Module系列函数，在提供了标准函数的同时还提供了更多。

Module 类内函数名	函数作用
get_local_full_path()	得到程序自身完整路径
get_local_program_name()	获得加载程序的文件名
get_local_program_size()	得到被加载程序的大小
get_local_program_base()	得到基地址
get_local_program_entry()	得到入口地址
check_module_imported(module_name)	验证程序是否导入了指定模块
get_name_from_module(address)	根据基地址得到模块名
get_base_from_module(module_name)	根据模块名得到基地址
get_oep_from_module(module_name)	根据模块名得到模块OEP入口
get_all_module_information()	得到所有模块信息
get_module_base(module_name)	得到特定模块基地址
get_local_base()	得到当前OEP位置处模块基地址
get_local_size()	获取当前OEP位置长度

Module 类内函数名	函数作用
get_local_protect()	获取当前OEP位置保护属性
get_module_from_function(module,function)	获取指定模块中指定函数内存地址
get_base_from_address(address)	根据传入地址得到模块首地址,开头4D 5A
get_base_address()	得到当前.text节基地址
get_base_from_name(module_name)	根据名字得到模块基地址
get_oep_from_name(module_name)	传入模块名得到OEP位置
get_oep_from_address(address)	传入模块地址得到OEP位置
get_module_from_import(module_name)	得到指定模块的导入表
get_import_inside_function(module_name,function_name)	检查指定模块内是否存在特定导入函数
get_import_iatva(module_name,function_name)	根据导入函数名得到函数iat_va地址
get_import_iatrva(module_name,function_name)	根据导入函数名得到函数iat_rva地址
get_module_from_export(module_name)	传入模块名,获取模块导出表
get_module_export_va(module_name,function_name)	传入模块名以及导出函数名,得到va地址
get_module_export_rva(module_name,function_name)	传入模块名以及导出函数,得到rva地址
get_local_section()	得到程序节表信息
get_local_address_from_section(section_name)	根据节名称得到地址
get_local_size_from_section(section_name)	根据节名称得到节大小
get_local_section_from_address(address)	根据地址得到节名称

此处只提供一个演示案例，获取当前被调试进程详细参数，包括路径，名称，入口地址，基地址，长度等。

```
from LyScript32 import MyDebug
from LyScriptTools32 import Module

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

    # 连接到调试器
    connect_flag = dbg.connect()
```

```
print("连接状态: {}".format(connect_flag))

# 类定义,并传入调试器对象
module = Module(dbg)

# 得到当前被调试程序完整路径
full_path = module.get_local_full_path()
print("完整路径: {}".format(full_path))

# 得到进程名字
local_name = module.get_local_program_name()
print("调试名称: {}".format(local_name))

# 验证是否导入了user32.dll
is_import = module.check_module_imported("user32.dll")
print("是否导入: {}".format(is_import))

# 根据模块名得到基地址
module_base = module.get_base_from_module("kernelbase.dll")
print("根据模块名得到基地址: {}".format(hex(module_base)))

dbg.close()
```

Memory 内存类

内存操作类是对内存操作函数与断点系列函数，二次封装而成，新增了新方法可供用户使用。

Memory 类内函数名	函数作用
read_memory_byte(decimal_int=0)	读取内存byte字节类型
read_memory_word(decimal_int=0)	读取内存word字类型
read_memory_dword(decimal_int=0)	读取内存dword双字类型
read_memory_ptr(decimal_int=0)	读取内存ptr指针
read_memory(decimal_int=0,decimal_length=0)	读取内存任意字节数,返回列表格式,错误则返回空列表
write_memory_byte(decimal_address=0, decimal_int=0)	写内存byte字节类型
write_memory_word(decimal_address=0, decimal_int=0)	写内存word字类型
write_memory_dword(decimal_address=0, decimal_int=0)	写内存dword双字类型

Memory 类内函数名	函数作用
write_memory_ptr(decimal_address=0, decimal_int=0)	写内存ptr指针类型
write_memory(decimal_address=0, decimal_list = [])	写内存任意字节数,传入十进制列表格式
scan_local_memory_one(search_opcode="")	扫描当前EIP所指向模块处的特征码 (传入参数 ff 25 ??)
scan_local_memory_all(search_opcode="")	扫描当前EIP所指向模块处的特征码,以列表形式反回全部
scan_memory_all_from_module(module_name="", search_opcode="")	扫描特定模块中的特征码,以列表形式反汇所有
scan_memory_one_from_module(module_name="", search_opcode="")	扫描特定模块中的特征码,返回第一条
scanall_memory_module_one(search_opcode="")	扫描所有模块,找到了以列表形式返回模块名称与地址
get_local_protect()	获取EIP所在位置处的内存属性值
get_memory_protect(decimal_address=0)	获取指定位置处内存属性值
set_local_protect(decimal_address=0,decimal_attribute=32,decimal_size=0)	设置指定位置保护属性值 ER执行/读取=32
get_local_page_size()	获取当前页面长度
get_memory_section()	得到内存中的节表
memory_xchage(memory_ptr_x=0, memory_ptr_y=0, bytes=0)	交换两个内存区域
memory_cmp(dbg,memory_ptr_x=0,memory_ptr_y=0,bytes=0)	对比两个内存区域
set_breakpoint(decimal_address=0)	设置内存断点,传入十进制
delete_breakpoint(decimal_address=0)	删除内存断点

Memory 类内函数名	函数作用
check_breakpoint(decimal_address=0)	检查内存断点是否命中
get_all_breakpoint()	获取所有内存断点
set_hardware_breakpoint(decimal_address=0, decimal_type=0)	设置硬件断点[类型 0 = r / 1 = w / 2 = e]
delete_hardware_breakpoint(decimal_address=0)	删除硬件断点

我们以扫描当前程序中所有符合特定特征条件的代码为例说明使用方法。

```
from LyScript32 import MyDebug
from LyScriptTools32 import Memory

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

    eip = dbg.get_register("eip")

    # 定义内存类
    memory = Memory(dbg)

    # 扫描整个模块中包含特征的地址
    ref = memory.scanall_memory_module_one("ff 25 ??")
    print("输出列表: {}".format(ref))

    dbg.close()
```

Disassemble 反汇编类

Disassemble 反汇编类增加了新的API函数的让用户有更多选择，需要注意API定义中，地址后面带有0的说明其可以省略参数，缺省值默认取当前EIP位置。

Disassemble 类内函数名	函数作用
is_call(address=0)	是否是跳转指令
is_jmp(address=0)	是否是jmp
is_ret(address=0)	是否是ret
is_nop(address=0)	是否是nop
is_cond(address=0)	是否是条件跳转指令
is_cmp(address=0)	是否cmp比较指令

Disassemble 类内函数名	函数作用
is_test(address=0)	是否是test比较指令
is_(address,cond)	自定义判断条件
get_assembly(address=0)	得到指定位置汇编指令,不填写默认获取EIP位置处
get_opcode(address=0)	得到指定位置机器码
get_disasm_operand_size(address=0)	获取反汇编代码长度
assemble_code_size(assemble)	计算用户传入汇编指令长度
get_assemble_code(assemble)	用户传入汇编指令返回机器码
write_assemble(address,assemble)	将汇编指令写出到指定内存位置
get_disasm_code(address,size)	反汇编指定行数
get_disasm_one_code(address = 0)	向下反汇编一行
get_disasm_operand_code(address=0)	得到当前内存地址反汇编代码的操作数
get_disasm_next(eip)	获取当前EIP指令的下一条指令
get_disasm_prev(eip)	获取当前EIP指令的上一条指令

我们来举一个使用案例，其实和模块调用原理是一样的，调用时先初始化，然后就可以使用内部的函数了。

```
from LyScript32 import MyDebug
from LyScriptTools32 import Module
from LyScriptTools32 import Disassemble

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

    # 反汇编类
    dasm = Disassemble(dbg)

    # 无参数使用
    ref = dasm.is_jmp()
    print("是否是JMP: {}".format(ref))

    # 携带参数使用
    eip = dbg.get_register("eip")
    eip_flag = dasm.is_jmp(eip)
    print("EIP位置: {}".format(eip_flag))

    dbg.close()
```

DebugControl 调试控制类

DebugControl 调试控制类，封装自寄存器系列函数。

debugcontrol 类内函数名	函数作用
get_eax()	获取通用寄存器系列
set_eax(decimal_value)	设置特定寄存器中的值(十进制)
get_zf()	获取标志寄存器系列
set_zf(decimal_bool)	设置标志寄存器的值(布尔型)
script_initdebug(path)	传入文件路径,载入被调试程序
script_closedebug()	终止当前被调试进程
script_detachdebug()	让进程脱离当前调试器
script_rundebug()	让进程运行起来
script_erun()	释放锁并允许程序运行，忽略异常
script_serun()	释放锁并允许程序运行，跳过异常中断
script_pause()	暂停调试器运行
script_stepinto()	步进
script_estepinfo()	步进,跳过异常
script_sestepinto()	步进,跳过中断
script_stepover()	步过到结束
script_stepout()	普通步过f8
script_skip()	跳过执行
script_inc(register)	递增寄存器
script_dec(register)	递减寄存器
script_add(register,decimal_int)	对寄存器进行add运算
script_sub(register,decimal_int)	对寄存器进行sub运算
script_mul(register,decimal_int)	对寄存器进行mul乘法
script_div(register,decimal_int)	对寄存器进行div除法
script_and(register,decimal_int)	对寄存器进行and与运算
script_or(register,decimal_int)	对寄存器进行or或运算
script_xor(register,decimal_int)	对寄存器进行xor或运算
script_neg(register,decimal_int)	对寄存器参数进行neg反转

debugcontrol 类内函数名	函数作用
script_rol(register,decimal_int)	对寄存器进行rol循环左移
script_ror(register,decimal_int)	对寄存器进行ror循环右移
script_shl(register,decimal_int)	对寄存器进行shl逻辑左移
script_shr(register,decimal_int)	对寄存器进行shr逻辑右移
script_sal(register,decimal_int)	对寄存器进行sal算数左移
script_sar(register,decimal_int)	对寄存器进行sar算数右移
script_not(register,decimal_int)	对寄存器进行not按位取反
script_bswap(register,decimal_int)	进行字节交换也就是反转
script_push(register_or_value)	对寄存器入栈
script_pop(register_or_value)	对寄存器弹出元素
pause()	内置api暂停
run()	内置api运行
stepin()	内置api步入
stepout()	内置api步过
stepout()	内置api到结束
stop()	内置api停止
wait()	内置api等待
is_debug()	内置api判断调试器是否在调试
is_running()	内置api判断调试器是否在运行

自动控制类主要功能如上表示，其中Script开头的API是调用的脚本命令实现，其他的是API实现，我们以批量自动载入程序为例，演示该类内函数是如何使用的。

```
import os
import pefile
import time
from LyScript32 import MyDebug
from LyScriptTools32 import Module
from LyScriptTools32 import Disassemble
from LyScriptTools32 import DebugControl

# 得到特定目录下的所有文件,并返回列表
def GetFullFilePaht(path):
    ref = []
    for root,dirs,files in os.walk(str(path)):
        for index in range(0,len(files)):
            ref.append(str(root + "/" + files[index]))
    return ref
```

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

    # 初始化调试控制器
    debug = DebugControl(dbg)

    # 得到特定目录下的所有文件
    full_path = GetFullFilePaht("d://test/")

    for i in range(0,len(full_path)):
        debug.Script_InitDebug(str(full_path[i]))
        time.sleep(0.3)
        debug.Script_RunDebug()

        time.sleep(0.3)
        local_base = dbg.get_local_base()
        print("当前调试进程: {} 基地址: {}".format(full_path[i],local_base))

        time.sleep(0.3)
        # 关闭调试
        debug.Script_CloseDebug()

    dbg.close()
```