# x64dbg 自动化控制插件

简体中文 | ENGLISH | русский язык



一款 x64dbg 自动化控制插件，通过Python控制x64dbg的行为，实现远程动态调试，解决了逆向工作者分析程序，反病毒人员脱壳，漏洞分析者寻找指令片段，原生脚本不够强大的问题，通过与Python相结合利用Python语法的灵活性以及其丰富的第三方库，加速漏洞利用程序的开发，辅助漏洞挖掘以及恶意软件分析。

Python 包请安装与插件一致的版本，在cmd命令行下执行pip命令即可安装，推荐两个包全部安装。

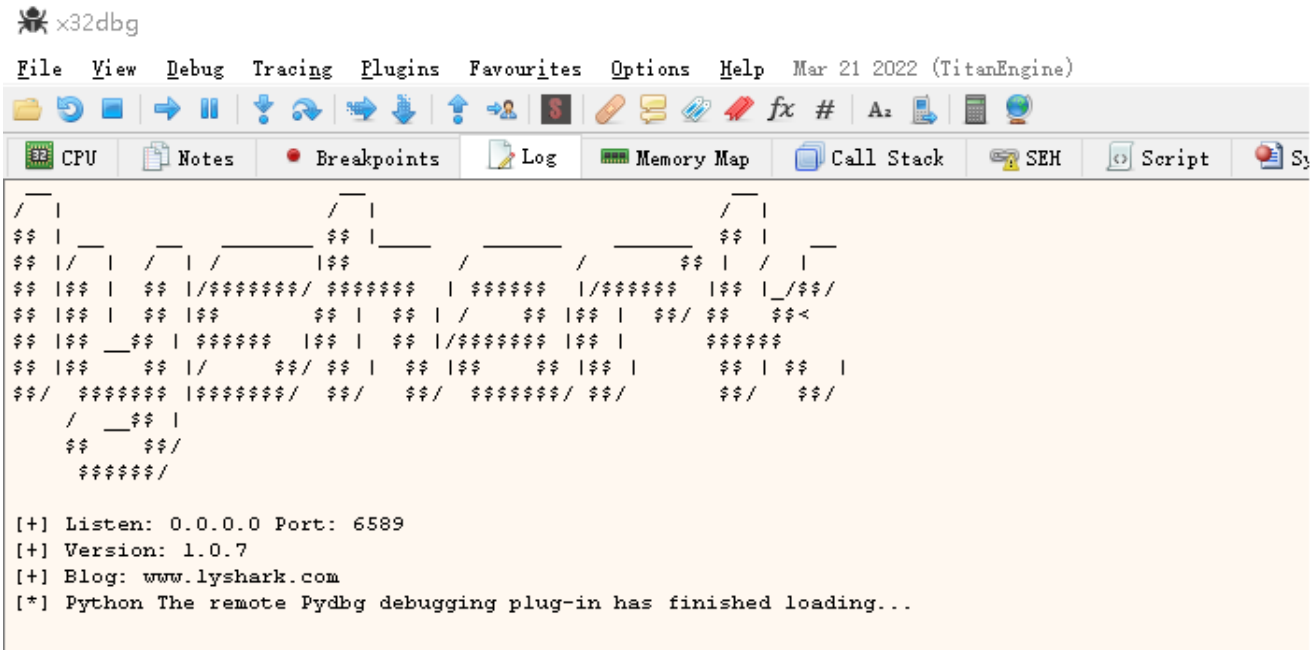
- 安装标准包：`pip install LyScript32` 或者 `pip install LyScript64`
- 安装扩展包：`pip install LyScriptTools32` 或者 `pip install LyScriptTools64`

其次您需要手动下载对应x64dbg版本的驱动文件，并放入指定的`plugins`目录下。

- 插件下载：LyScript32 1.0.11 (32位插件) 或者 LyScript64 1.0.11 (64位插件)

插件下载好以后，请将该插件复制到x64dbg的plugins目录下，程序运行后会自动加载插件。


当插件加载成功后，会在日志位置看到具体的绑定信息以及输出调试，该插件并不会在插件栏显示。



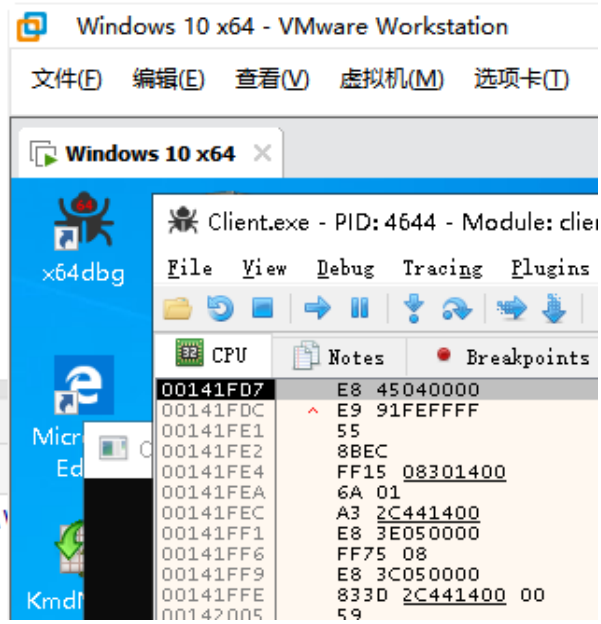

如果需要远程调试，则只需要在初始化`MyDebug()`类时传入对端IP地址即可，如果不填写参数则默认使用`127.0.0.1`地址，请确保对端放行了`6589`端口，否则无法连接。

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug(address="192.168.150.129")
    connect_flag = dbg.connect()

    ref = dbg.get_register("eip")
    print("EIP地址 = {}".format(hex(ref)))
    dbg.close()
```

```
main
C:\Users\lyshark\PycharmProjects\pythonProject\v
EIP地址 = 0x141fd7
```




运行x64dbg程序并手动载入需要分析的可执行文件，然后我们可以通过`connect()`方法连接到调试器，连接后会创建一个持久会话直到python脚本结束则连接会被强制断开，在此期间可调用`is_connect()`检查该链接是否还存在，具体代码如下所示。

```
from LyScript32 import MyDebug

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))

    # 检测套接字是否还在
    ref = dbg.is_connect()
```

```
    print("是否在连接: ", ref)

    dbg.close()
```

## 寄存器系列函数

**get_register() 函数:** 该函数主要用于实现，对特定寄存器的获取操作，用户需传入需要获取的寄存器名字即可。

- 参数1：传入寄存器字符串

可用范围：“DR0”, “DR1”, “DR2”, “DR3”, “DR6”, “DR7”, “EAX”, “AX”, “AH”, “AL”, “EBX”, “BX”, “BH”, “BL”, “ECX”, “CX”, “CH”, “CL”, “EDX”, “DX”, “DH”, “DL”, “EDI”, “DI”,“ESI”, “SI”, “EBP”, “BP”, “ESP”, “SP”, “EIP”, “CIP”, “CSP”, “CAX”, “CBX”, “CCX”, “CDX”, “CDI”, “CSI”, “CBP”, “CFLAGS”

```
from LyScript32 import MyDebug

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

    eax = dbg.get_register("eax")
    ebx = dbg.get_register("ebx")

    print("eax = {}".format(hex(eax)))
    print("ebx = {}".format(hex(ebx)))

    dbg.close()
```

如果您使用的是64位插件，则寄存器的支持范围将变为E系列加R系列。

可用范围扩展： “DR0”, “DR1”, “DR2”, “DR3”, “DR6”, “DR7”, “EAX”, “AX”, “AH”, “AL”, “EBX”, “BX”, “BH”, “BL”, “ECX”, “CX”, “CH”, “CL”, “EDX”, “DX”, “DH”, “DL”, “EDI”, “DI”, “ESI”, “SI”, “EBP”, “BP”, “ESP”, “SP”, “EIP”, “RAX”, “RBX”, “RCX”, “RDX”, “RSI”, “SIL”, “RDI”, “DIL”, “RBP”, “BPL”, “RSP”, “SPL”, “RIP”, “R8”, “R8D”, “R8W”, “R8B”, “R9”, “R9D”, “R9W”, “R9B”, “R10”, “R10D”, “R10W”, “R10B”, “R11”, “R11D”, “R11W”, “R11B”, “R12”, “R12D”, “R12W”, “R12B”, “R13”, “R13D”, “R13W”, “R13B”, “R14”, “R14D”, “R14W”, “R14B”, “R15”, “R15D”, “R15W”, “R15B”

```
from LyScript64 import MyDebug

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

    rax = dbg.get_register("rax")
    eax = dbg.get_register("eax")
    ax = dbg.get_register("ax")

    print("rax = {} eax = {} ax ={}".format(hex(rax),hex(eax),hex(ax)))

    r8 = dbg.get_register("r8")
```

```
    print("获取R系列寄存器: {}".format(hex(r8)))

    dbg.close()
```

**set_register() 函数:** 该函数实现设置指定寄存器参数，同理64位将支持更多寄存器的参数修改。

- 参数1：传入寄存器字符串
- 参数2：十进制数值

可用范围："DR0", "DR1", "DR2", "DR3", "DR6", "DR7", "EAX", "AX", "AH", "AL", "EBX", "BX", "BH", "BL", "ECX", "CX", "CH", "CL", "EDX", "DX", "DH", "DL", "EDI", "DI", "ESI", "SI", "EBP", "BP", "ESP", "SP", "EIP"

```
from LyScript32 import MyDebug

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

    eax = dbg.get_register("eax")

    dbg.set_register("eax",100)

    print("eax = {}".format(hex(eax)))

    dbg.close()
```

**get_flag_register() 函数:** 用于获取某个标志位参数，返回值只有真或者假。

- 参数1：寄存器字符串

可用寄存器范围："ZF", "OF", "CF", "PF", "SF", "TF", "AF", "DF", "IF"

```
from LyScript32 import MyDebug

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

    cf = dbg.get_flag_register("cf")
    print("标志: {}".format(cf))

    dbg.close()
```

**set_flag_register() 函数:** 用于设置某个标志位参数，返回值只有真或者假。

- 参数1：寄存器字符串
- 参数2：[ 设置为真 True] / [设置为假 False]

可用寄存器范围："ZF", "OF", "CF", "PF", "SF", "TF", "AF", "DF", "IF"

```
from LyScript32 import MyDebug

if __name__ == "__main__":
```

```
    dbg = MyDebug()
    connect_flag = dbg.connect()
    zf = dbg.get_flag_register("zf")
    print(zf)

    dbg.set_flag_register("zf",False)

    zf = dbg.get_flag_register("zf")
    print(zf)

    dbg.close()
```

## 调试系列函数

**set_debug() 函数:** 用于影响调试器，例如前进一次，后退一次，暂停调试，终止等。

- 参数1: 传入需要执行的动作

可用动作范围：[暂停 Pause] [运行 Run] [步入 StepIn]  [步过 StepOut] [到结束 StepOver] [停止 Stop] [等待 Wait]

```
from LyScript32 import MyDebug

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

    while True:
        dbg.set_debug("StepIn")

        eax = dbg.get_register("eax")

        if eax == 0:
            print("找到了")
            break

    dbg.close()
```

**set_debug_count() 函数:** 该函数是`set_debug()`函数的延续，目的是执行自动步过次数。

- 参数1：传入需要执行的动作
- 参数2：执行重复次数

可用动作范围：[暂停 Pause] [运行 Run] [步入 StepIn]  [步过 StepOut] [到结束 StepOver] [停止 Stop] [等待 Wait]

```
from LyScript32 import MyDebug

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

    dbg.set_debug_count("StepIn",10)

    dbg.close()
```

**is_debugger() /is_running() 函数:** is_debugger可用于验证当前调试器是否处于调试状态，is_running则用于验证是否在运行。

- 无参数传递

```
from LyScript32 import MyDebug

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

    ref = dbg.is_debugger()
    print(ref)

    ref = dbg.is_running()
    print(ref)

    dbg.close()
```

**set_breakpoint() 函数:** 设置断点与取消断点进行了分离，设置断点只需要传入十进制内存地址。

- 参数1：传入内存地址（十进制）

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    ref = dbg.set_breakpoint(eip)

    print("设置状态: {}".format(ref))
    dbg.close()
```

**delete_breakpoint() 函数:** 该函数传入一个内存地址，可取消一个内存断点。

- 参数1：传入内存地址（十进制）

```
from LyScript32 import MyDebug

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

```
    connect_flag = dbg.connect()

    eip = dbg.get_register("eip")
    ref = dbg.set_breakpoint(eip)
    print("设置状态: {}".format(ref))

    del_ref = dbg.delete_breakpoint(eip)
    print("取消状态: {}".format(del_ref))

    dbg.close()
```

**check_breakpoint() 函数:** 用于检查下过的断点是否被命中，命中返回True否则返回False。

- 参数1：传入内存地址（十进制）

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    ref = dbg.set_breakpoint(eip)
    print("设置状态: {}".format(ref))

    is_check = dbg.check_breakpoint(4134331)
    print("是否命中: {}".format(is_check))

    dbg.close()
```

**get_all_breakpoint() 函数:** 用于获取当前调试程序中，所有下过的断点信息，包括是否开启，命中次数等。

- 无参数传递

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    ref = dbg.get_all_breakpoint()
    print(ref)
    dbg.close()
```

**set_hardware_breakpoint() 函数:** 用于设置一个硬件断点，硬件断点在32位系统中最多设置4个。

- 参数1：内存地址（十进制）
- 参数2：断点类型

断点类型可用范围：[类型 0 = HardwareAccess / 1 = HardwareWrite / 2 = HardwareExecute]

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug(address="127.0.0.1",port=6666)
    connect_flag = dbg.connect()

    eip = dbg.get_register("eip")

    ref = dbg.set_hardware_breakpoint(eip,2)
    print(ref)

    dbg.close()
```

**delete_hardware_breakpoint() 函数:** 用于删除一个硬件断点，只需要传入地址即可，无需传入类型。

- 参数1：内存地址（十进制）

断点类型可用范围：[类型 0 = HardwareAccess / 1 = HardwareWrite / 2 = HardwareExecute]

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug(address="127.0.0.1",port=6666)
    connect_flag = dbg.connect()

    eip = dbg.get_register("eip")

    ref = dbg.set_hardware_breakpoint(eip,2)
    print(ref)

    # 删除断点
    ref = dbg.delete_hardware_breakpoint(eip)
    print(ref)

    dbg.close()
```

## 模块系列函数

**get_module_base() 函数:** 该函数可用于获取程序载入的指定一个模块的基地址。

- 参数1：模块名字符串

```
from LyScript32 import MyDebug

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

    user32_base = dbg.get_module_base("user32.dll")
    print(user32_base)

    dbg.close()
```

**get_all_module() 函数:** 用于输出当前加载程序的所有模块信息，以字典的形式返回。

- 参数：无参数

```
from LyScript32 import MyDebug

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

    ref = dbg.get_all_module()

    for i in ref:
        print(i)

    print(ref[0])
    print(ref[1].get("name"))
    print(ref[1].get("path"))

    dbg.close()
```

**get_local_() 系列函数:** 获取当前EIP所在模块基地址，长度，以及内存属性，此功能无参数传递，获取的是当前EIP所指向模块的数据。

- dbg.get_local_base()　获取模块基地址
- dbg.get_local_size()　获取模块长度
- dbg.get_local_protect() 获取模块保护属性

```
from LyScript32 import MyDebug

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

    ref = dbg.get_local_base()
    print(hex(ref))

    ref2 = dbg.get_local_size()
    print(hex(ref2))

    ref3 = dbg.get_local_protect()
    print(ref3)

    dbg.close()
```

**get_module_from_function() 函数:** 获取指定模块中指定函数的内存地址，可用于验证当前程序在内存中指定函数的虚拟地址。

- 参数1：模块名
- 参数2：函数名

成功返回地址，失败返回false

```
from LyScript32 import MyDebug

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

    ref = dbg.get_module_from_function("user32.dll","MessageBoxW")
    print(hex(ref))

    ref2 = dbg.get_module_from_function("kernel32.dll","test")
    print(ref2)

    dbg.close()
```

**get_module_from_import() 函数:** 获取当前程序中指定模块的导入表信息，输出为列表嵌套字典。

- 参数1：传入模块名称

```
from LyScript32 import MyDebug

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

    ref = dbg.get_module_from_import("ucrtbase.dll")
    print(ref)

    ref1 = dbg.get_module_from_import("win32project1.exe")

    for i in ref1:
        print(i.get("name"))

    dbg.close()
```

**get_module_from_export() 函数:** 该函数用于获取当前加载程序中的导出表信息。

- 参数1：传入模块名

```
from LyScript32 import MyDebug

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

    ref = dbg.get_module_from_export("msvcr120d.dll")

    for i in ref:
        print(i.get("name"), hex(i.get("va")))

    dbg.close()
```

**get_section() 函数:** 该函数用于输出主程序中的节表信息。

- 无参数传递

```
 from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug(address="127.0.0.1",port=6666)
    connect_flag = dbg.connect()

    ref = dbg.get_section()
    print(ref)

    dbg.close()
```

**get_base_from_address() 函数:** 根据传入的内存地址得到该模块首地址。

- 参数1：传入内存地址（十进制）

```
from LyScript32 import MyDebug

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

    ref = dbg.get_base_from_address(eip)
    print("模块首地址: {}".format(hex(ref)))
```

**get_base_from_name() 函数:** 根据传入的模块名得到该模块所在内存首地址。

- 参数1：传入模块名

```
from LyScript32 import MyDebug

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

    ref_base = dbg.get_base_from_name("win32project.exe")
    print("模块首地址: {}".format(hex(ref_base)))

    dbg.close()
```

**get_oep_from_name() 函数:** 根据传入的模块名，获取该模块实际装载OEP位置。

- 参数1：传入模块名

```
from LyScript32 import MyDebug

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

    oep = dbg.get_oep_from_name("win32project.exe")
    print(hex(oep))

    dbg.close()
```

**get_oep_from_address() 函数:** 根据传入内存地址，得到该地址模块的OEP位置。

- 参数1：传入内存地址

```
from LyScript32 import MyDebug

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

    oep = dbg.get_oep_from_address(eip)
    print(hex(oep))

    dbg.close()
```

## 内存系列函数

**read_memory_() 系列函数:** 读内存系列函数，包括 ReadByte,ReadWord,ReadDword 三种格式，在64位下才支持Qword

- 参数1：需要读取的内存地址（十进制）

目前支持:

- read_memory_byte() 读字节
- read_memory_word() 读word
- read_memory_dword() 读dword
- read_memory_qword() 读qword （仅支持64位）
- read_memory_ptr() 读指针

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")

    ref = dbg.read_memory_byte(eip)
    print(hex(ref))
```

```
    ref2 = dbg.read_memory_word(eip)
    print(hex(ref2))

    ref3 = dbg.read_memory_dword(eip)
    print(hex(ref3))

    ref4 = dbg.read_memory_ptr(eip)
    print(hex(ref4))

    dbg.close()
```

**write_memory_() 系列函数:** 写内存系列函数，WriteByte,WriteWord,WriteDWORD,WriteQword

- 参数1：需要写入的内存
- 参数2：需要写入的byte字节

目前支持:

- write_memory_byte() 写字节
- write_memory_word() 写word
- write_memory_dword() 写dword
- write_memory_qword() 写qword （仅支持64位）
- write_memory_ptr() 写指针

```
from LyScript32 import MyDebug

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

    addr = dbg.create_alloc(1024)
    print(hex(addr))

    ref = dbg.write_memory_byte(addr,10)

    print(ref)

    dbg.close()
```

**scan_memory_one() 函数:** 实现了内存扫描，当扫描到第一个符合条件的特征时，自动输出。

- 参数1：特征码字段

这个函数需要注意，如果我们的x64dbg工具停在系统领空，则会默认搜索系统领空下的特征，如果像搜索程序里面的，需要先将EIP切过去在操作。

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    ref = dbg.scan_memory_one("ff 25")
    print(ref)
    dbg.close()
```

**scan_memory_all() 函数:** 实现了扫描所有符合条件的特征字段，找到后返回一个列表。

- 参数1：特征码字段

```
from LyScript32 import MyDebug

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

    ref = dbg.scan_memory_all("ff 25")

    for index in ref:
        print(hex(index))

    dbg.close()
```

**get_local_protect() 函数:** 获取内存属性传值，该函数进行更新，取消了只能得到EIP所指的位置的内存属性，用户可随意检测。

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print(eip)

    ref = dbg.get_local_protect(eip)
    print(ref)
```

**set_local_protect() 函数:** 新增设置内存属性函数，传入eip内存地址，设置属性32，以及设置内存长度1024即可。

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print(eip)

    b = dbg.set_local_protect(eip,32,1024)
    print("设置属性状态: {}".format(b))

    dbg.close()
```

**get_local_page_size() 函数:** 用于获取当前EIP所指领空下，内存pagesize分页大小。

- 无参数传递

```
from LyScript32 import MyDebug

if __name__ == "__main__":
    dbg = MyDebug()
    connect_flag = dbg.connect()
    size = dbg.get_local_page_size()
    print("pagesize = {}".format(size))

    dbg.close()
```

**get_memory_section() 函数:** 该函数主要用于获取内存映像中，当前调试程序的内存节表数据。

- 无参数传递

```
from LyScript32 import MyDebug

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

    ref = dbg.get_memory_section()
    print(ref)
    dbg.close()
```

## 堆栈系列函数

**create_alloc() 函数:** 函数`create_alloc()`可在远程开辟一段堆空间，成功返回内存首地址。

- 参数1：开辟的堆长度（十进制）

```
from LyScript32 import MyDebug

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

    ref = dbg.create_alloc(1024)
    print("开辟地址: ", hex(ref))

    dbg.close()
```

**delete_alloc() 函数:** 函数`delete_alloc()`用于注销一个远程堆空间。

- 参数1：传入需要删除的堆空间内存地址。

```
from LyScript32 import MyDebug

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

    ref = dbg.create_alloc(1024)
    print("开辟地址: ", hex(ref))

    flag = dbg.delete_alloc(ref)
    print("删除状态: ",flag)

    dbg.close()
```

**push_stack() 函数:** 将一个十进制数压入堆栈中，默认在堆栈栈顶。

- 参数1：十进制数据

```
from LyScript32 import MyDebug

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

    ref = dbg.push_stack(10)

    print(ref)

    dbg.close()
```

**pop_stack() 函数:** pop函数用于从堆栈中推出一个元素，默认从栈顶弹出。

- 无参数传递

```
from LyScript32 import MyDebug

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

    tt = dbg.pop_stack()
    print(tt)

    dbg.close()
```

**peek_stack() 函数:** peek则用于检查堆栈内的参数，可设置偏移值，不设置则默认检查第一个也就是栈顶。

- 参数1：十进制偏移

```
from LyScript32 import MyDebug

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

```
    # 无参数检查
    check = dbg.peek_stack()
    print(check)

    # 携带参数检查
    check_1 = dbg.peek_stack(2)
    print(check_1)

    dbg.close()
```

## 进程线程系列函数

**get_thread_list() 函数:** 该函数可输出当前进程所有在运行的线程信息。

- 无参数传递

```
from LyScript32 import MyDebug

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

    ref = dbg.get_thread_list()
    print(ref[0])
    print(ref[1])

    dbg.close()
```

**get_process_handle() 函数:** 用于获取当前进程句柄信息。

- 无参数传递

```
from LyScript32 import MyDebug

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

    ref = dbg.get_process_handle()
    print(ref)

    dbg.close()
```

**get_process_id() 函数:** 用于获取当前加载程序的PID

- 无参数传递

```
from LyScript32 import MyDebug

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

    ref = dbg.get_process_id()
    print(ref)

    dbg.close()
```

**get_teb_address() / get_peb_address() 系列函数:** 用于获取当前进程环境块，和线程环境快。

- get_teb_address() 传入参数是线程ID
- get_peb_address() 传入参数是进程ID

```
from LyScript32 import MyDebug

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

    ref = dbg.get_teb_address(6128)
    print(ref)

    ref = dbg.get_peb_address(9012)
    print(ref)

    dbg.close()
```

## 反汇编系列函数

**get_disasm_code() 函数:** 该函数主要用于对特定内存地址进行反汇编，传入两个参数。

- 参数1：需要反汇编的地址(十进制)
- 参数2：需要向下反汇编的长度

```
from LyScript32 import MyDebug

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100行
    disasm_dict = dbg.get_disasm_code(eip,100)

    for ds in disasm_dict:
        print("地址: {} 反汇编: {}".format(hex(ds.get("addr")),ds.get("opcode")))
```

```
    dbg.close()
```

**get_disasm_one_code() 函数:** 在用户指定的位置读入一条汇编指令，用户可根据需要对其进行判断。

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print("EIP = {}".format(eip))

    disasm = dbg.get_disasm_one_code(eip)
    print("反汇编一条: {}".format(disasm))

    dbg.close()
```

**get_disasm_operand_code() 函数:** 用于获取汇编指令中的操作数，例如`jmp 0x0401000`其操作数就是`0x0401000`。

- 参数1：传入内存地址（十进制）

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print("EIP = {}".format(eip))

    opcode = dbg.get_disasm_operand_code(eip)
    print("操作数: {}".format(hex(opcode)))

    dbg.close()
```

**get_disasm_operand_size() 函数:** 用于得当前内存地址下汇编代码的机器码长度。

- 参数1：传入内存地址（十进制）

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print("EIP = {}".format(eip))

    opcode = dbg.get_disasm_operand_size(eip)

    print("机器码长度: {}".format(hex(opcode)))
```

```
    dbg.close()
```

**assemble_write_memory() 函数:** 实现了用户传入一段正确的汇编指令，程序自动将该指令转为机器码，并写入到指定位置。

- 参数1：写出内存地址（十进制）
- 参数2：写出汇编指令

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print(eip)

    ref = dbg.assemble_write_memory(eip,"mov eax,1")
    print("是否写出: {}".format(ref))

    dbg.close()
```

**assemble_code_size() 函数:** 该函数实现了用户传入一个汇编指令，自动计算出该指令占多少个字节。

- 参数1：汇编指令字符串

```
from LyScript32 import MyDebug

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

    ref = dbg.assemble_code_size("sub esp, 0x324")
    print(ref)

    dbg.close()
```

## 其他系列函数

**set_comment_notes() 函数:** 给指定位置代码增加一段注释，如下演示在eip位置增加注释。

- 参数1：注释内存地址
- 参数2：注释内容

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    ref = dbg.set_comment_notes(eip,"hello lyshark")
    print(ref)

    dbg.close()
```

**run_command_exec() 函数:** 执行内置命令，例如bp,dump等。

- 参数1：命令语句

```
from LyScript32 import MyDebug

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

    ref = dbg.run_command_exec("bp MessageBoxA")
    print(ref)

    dbg.close()
```

**set_loger_output() 函数:** 日志的输出尤为重要，该模块提供了自定义日志输出功能，可将指定日志输出到x64dbg日志位置。

- 参数1：日志内容

```
from LyScript32 import MyDebug

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

    for i in range(0,100):
        ref = dbg.set_loger_output("hello lyshark -> {} \n".format(i))
        print(ref)

    dbg.close()
```

## LyScript 1.0.11 新版特性

LyScript 1.0.11 插件在原有函数基础上封装实现了更多有用的功能，并解决了旧版本插件中x64无法反汇编的问题，新版本插件与旧版本保持兼容，原函数不发生变化，您依然可以使用，如果需要使用新版本中的新函数，请安装以下新版本插件，并更新您的LyScript标准包。

---

| LyScript 1.0.11 新增函数 | 函数作用 |
| --- | --- |
| run_command_exe_ref(command) | 执行脚本命令(返回整数) |
| set_status_bar_message(message) | 在状态栏上面输出字符串提示 |
| get_window_handle() | 取出自身进程模块句柄 |
| get_disassembly(address) | 反汇编一条指令(新增) |
| assemble_at(address,assemble) | 传入汇编指令,直接写出到内存 |
| disasm_fast_at(address) | 反汇编一条指令,返回完整字典 |
| get_module_at(eip) | 获取EIP所在位置处模块名 |
| get_xref_count_at(eip) | 获取EIP位置处交叉引用计数 |
| get_xref_type_at(eip) | 得到EIP位置处交叉引用类型 XREFTYPE |
| get_bpx_type_at(address) | 得到指定地址处BP断点类型 BPXTYPE |
| get_function_type_at(eip) | 获得EIP位置处函数类型 FUNCTYPE |
| is_bp_disable(address) | 验证指定地址处BP断点是否被禁用 |
| is_jmp_going_to_execute(eip) | 是否跳转到可执行内存块 |
| is_run_locked() | 检查调试器是否被锁定(暂停) |
| mem_find_base_addr(eip) | 返回EIP位置处内存模块基地址和大小(字典) |
| mem_get_page_size(eip) | 得到EIP位置处内存页面长度 |
| mem_is_valid(eip) | 验证EIP位置处内存是否可读 |
| script_loader(file_path) | 从文件中加载x64dbg内置脚本 |
| script_unloader() | 关闭打开的脚本 |
| script_run() | 运行x64dbg内置脚本 |
| script_set_ip(index) | 脚本指定运行到第index条 |
| open_debug(file_path) | 打开硬盘中的被调试程序(打开功能) |
| close_debug() | 关闭被调试进程 |
| detach_debug() | 进程脱离调试器 |
| input_string_box(message) | 弹出输入框,用户输入后得到输入值 |
| message_box_yes_no(title) | 弹出是否按钮选择框 |
| message_box(title) | 弹出信息框,用于提示用户 |
| get_branch_destination(address=0) | 获取call或者是跳转指令的跳转地址 |
| set_argument_brackets(start_address=0,end_address=0) | 在注释处增加括号 |

| LyScript 1.0.11 新增函数 | 函数作用 |
| --- | --- |
| del_argument_brackets(start_address=0) | 删除注释处的括号 |
| set_function_brackets(start_address=0,end_address=0) | 在机器码位置增加注释 |
| del_function_brackets(start_address=0) | 删除机器码位置处的注释 |
| set_loop_brackets(start_address=0,end_address=0) | 在反汇编位置添加注释 |
| del_loop_brackets(depth=1, start_address=0) | 删除反汇编位置处的注释 |
| get_section_from_module_name(module_name) | 传入模块名称,获取其节表并输出 |
| clear_log() | 清空日志 |
| switch_cpu() | 切换到CPU窗口 |
| update_all_view() | 刷新所有视图参数 |
| size_from_address(eip) | 传入基地址得到模块占用总大小 |
| size_from_name(module_name) | 传入模块名称得到模块占用总大小 |
| section_count_from_name(module_name) | 传入模块名称得到模块有多少个节区 |
| section_count_from_address(eip) | 传入模块基址得到模块有多少个节区 |
| path_from_name(module_name) | 传入模块名称得到模块路径 |
| path_from_address(eip) | 传入模块地址得到模块路径 |
| name_from_address(eip) | 传入模块地址得到模块名称 |
| get_local_module_size() | 获取当前程序的大小 |
| get_local_module_section_Count() | 获取自身节数量 |
| get_local_module_path() | 获取被调试程序完整路径 |
| get_local_module_name() | 获取自身模块名 |
| get_local_module_entry() | 获取自身模块入口 |
| get_local_module_base() | 获取自身模块基地址 |
| set_label_at(address,label) | 在特定位置设置标签 |
| location_label_at(label) | 定位到标签,返回内存地址 |
| clear_label() | 清空所有标签 |

新版本的更新增加和许多新函数，其中比较有代表性的要属下面这些用法。

**寄存器增加:** 无论32位还是64位，都可以直接获取`"CIP","CSP","CAX","CBX","CCX","CDX","CDI","CSI","CBP","CFLAGS"`这些寄存器的参数。

```
from LyScript32 import MyDebug
```

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

    eip = dbg.get_register("eip")
    print("eip寄存器 = {}".format(hex(eip)))

    csp = dbg.get_register("csp")
    print("csp寄存器 = {}".format(hex(csp)))

    cflags = dbg.get_register("cflags")
    print("cflags寄存器 = {}".format(hex(cflags)))

    dbg.close()
```

**内置参数返回功能:** 在老版本中命令执行无法携带参数传出，新版本直接在插件内部实现了参数传递，目前只支持整数。

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print("eip寄存器 = {}".format(hex(eip)))

    exec_ref = dbg.run_command_exe_ref("mod.base(eip)")
    print("base基地址 = {}".format(hex(exec_ref)))

    dbg.close()
```

**反汇编携带更多参数:** 反汇编 `disasm_fast_at` 命令可以携带更多参数，可供用户自行判断是否使用本条指令。

```
from LyScript32 import MyDebug

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

    eip = dbg.get_register("eip")
    print("eip寄存器 = {}".format(hex(eip)))

    dic_ref = dbg.disasm_fast_at(eip)
    print("返回字典: {}".format(dic_ref))

    dbg.close()
```

**脚本载入执行功能:** 增加了脚本的载入与执行功能，用户可以载入已有的x64dbg原生脚本并通过命令执行。

```
from LyScript32 import MyDebug

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

```
    conn = dbg.connect()

    # 加载x64dbg脚本
    flag = dbg.script_loader("d://test.txt")

    # 运行脚本
    flag = dbg.script_run()

    # 指定行号运行
    flag = dbg.script_set_ip(1)

    # 关闭脚本
    flag = dbg.script_unloader()

    dbg.close()
```

**弹窗提醒功能:** 此功能提供了三种对话框，一种可输入文本，一种判断是否选中，另一种则是普通弹窗。

```
from LyScript32 import MyDebug

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

    # 弹出输入框
    flag = dbg.input_string_box("请输入反汇编入口地址?")
    print("用户的输入: {}".format(flag))

    # 弹出是否框
    flag = dbg.message_box_yes_no("是否继续执行脱壳操作?")
    if flag == True:
        print("脱壳")
    else:
        print("退出")

    # 提示框
    flag = dbg.message_box("这是第 {} 次,异常了".format(1))
    print("状态: {}".format(flag))

    dbg.close()
```

**自定义获取节表:** 用户可传入当前载入的模块名，即可直接取出指定模块的节表信息。

```
from LyScript32 import MyDebug

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

    ref = dbg.get_section_from_module_name("user32.dll")
    print(ref)

    dbg.close()
```

**打开关闭程序:** 本次更新还增加了打开关闭调试功能，用户可以传入文件路径让调试器打开，或者关闭指定程序。

```
from LyScript32 import MyDebug

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

    # 打开被调试进程
    ref = dbg.open_debug("d://lyshark.exe")

    # 关闭被调试进程
    ref = dbg.close_debug()

    dbg.close()
```