

三年前面朝黄土背朝天的我，写了一篇如何在 Windows 7 系统下枚举内核 SSDT 表的文章《驱动开发：内核读取 SSDT 表基址》三年过去了我还是个单身狗，开个玩笑，微软的 Windows 10 系统已经覆盖了大多数个人 PC 终端，以前的方法也该进行迭代更新了，或许在网上你能够找到类似的文章，但我可以百分百肯定都不能用，今天 Lyshark 将带大家一起分析 Win10 x64 最新系统 SSDT 表的枚举实现。

看一款闭源 ARK 工具的枚举效果：

进程	驱动模块	内核层	内核钩子	应用层钩子	设置	监控	启动信息	注册表	服务	文件	网络	调试引擎
SSDT	Shadow SSDT	内核钩子	系统中断表	Object钩子								

索引	函数名	原始函数地址	当前函数地址	当前函数地址所在模块
0	NtAccessCheck	0xFFFFF8036EB12340	0xFFFFF8036EB12340	C:\Windows\system32\ntoskrnl.exe
1	NtWorkerFactoryWorkerReady	0xFFFFF8036EB1C3D0	0xFFFFF8036EB1C3D0	C:\Windows\system32\ntoskrnl.exe
2	NtAcceptConnectPort	0xFFFFF8036F0DE450	0xFFFFF8036F0DE450	C:\Windows\system32\ntoskrnl.exe
3	NtMapUserPhysicalPagesScatter	0xFFFFF8036F2992F0	0xFFFFF8036F2992F0	C:\Windows\system32\ntoskrnl.exe
4	NtWaitForSingleObject	0xFFFFF8036EFF3B50	0xFFFFF8036EFF3B50	C:\Windows\system32\ntoskrnl.exe
5	NtCallbackReturn	0xFFFFF8036EBC4E10	0xFFFFF8036EBC4E10	C:\Windows\system32\ntoskrnl.exe
6	NtReadFile	0xFFFFF8036EFE5220	0xFFFFF8036EFE5220	C:\Windows\system32\ntoskrnl.exe
7	NtDeviceIoControlFile	0xFFFFF8036EFE8770	0xFFFFF8036EFE8770	C:\Windows\system32\ntoskrnl.exe

直接步入正题，首先 SSDT 表中文为系统服务描述符表，SSDT 表的作用是把应用层与内核层联系起来起到桥梁的作用，枚举 SSDT 表也是反内核工具最基本的功能，通常在 64 位系统中要想找到 SSDT 表，需要先找到 KeServiceDescriptorTable 这个函数，由于该函数没有被导出，所以只能动态的查找它的地址，庆幸的是我们可以通过查找 msr(c0000082) 这个特殊的寄存器来替代查找 KeServiceDescriptorTable 这一步，在新版系统中查找 SSDT 可以归纳为如下这几个步骤。

- rdmsr c0000082 -> KiSystemCall64Shadow -> KiSystemServiceUser -> SSDT

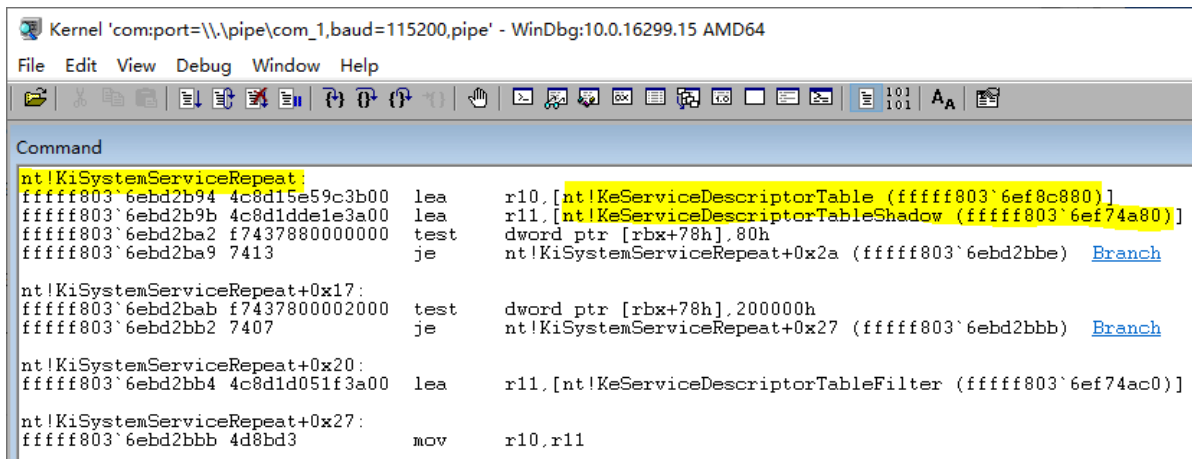
首先第一步通过 rdmsr c0000082 MSR 寄存器得到 KiSystemCall64Shadow 的函数地址，计算 KiSystemCall64Shadow 与 KiSystemServiceUser 偏移量，如下图所示。

- 得到相对偏移 6ed53180(KiSystemCall64Shadow) - 6ebd2a82(KiSystemServiceUser) = 1806FE
- 也就是说 6ed53180(rdmsr) - 1806FE = KiSystemServiceUser

```
Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:10.0.16299.15 AMD64
File Edit View Debug Window Help
[Icons]
Command
0: kd> .printf "hello lyshark.com \n\n"
hello lyshark.com
0: kd> rdmsr c0000082
msr[c0000082] = fffff803`6ed53180
0: kd> u KiSystemCall64Shadow
nt!KiSystemCall64Shadow:
fffff803`6ed53180 0f01f8 swapgs
fffff803`6ed53183 654889242510700000 mov     qword ptr gs:[7010h],rsp
fffff803`6ed5318c 65488b242500700000 mov     rsp,qword ptr gs:[7000h]
fffff803`6ed53195 650fba24251870000001 bt     dword ptr gs:[7018h],1
fffff803`6ed5319f 7203 jb     nt!KiSystemCall64Shadow+0x24 (fffff803`6ed531a4)
fffff803`6ed531a1 0f22dc mov     cr3,rsp
fffff803`6ed531a4 65488b242508700000 mov     rsp,qword ptr gs:[7008h]
fffff803`6ed531ad 6a2b push   2Bh
0: kd> u KiSystemServiceUser
nt!KiSystemServiceUser:
fffff803`6ebd2a82 c645ab02 mov     byte ptr [rbp-55h],2
fffff803`6ebd2a86 65488b1c2588010000 mov     rbx,qword ptr gs:[188h]
fffff803`6ebd2a8f 0f0d8b9000000000 prefetchw [rbx+90h]
fffff803`6ebd2a96 0fae5dac stmxcsr dword ptr [rbp-54h]
fffff803`6ebd2a9a 650fae142580010000 ldmxcsr dword ptr gs:[180h]
fffff803`6ebd2aa3 807b0300 cmp     byte ptr [rbx+3],0
fffff803`6ebd2aa7 66c78580000000000000 mov     word ptr [rbp+80h],0
fffff803`6ebd2ab0 0f84a800000000 je     nt!KiSystemServiceUser+0xdc (fffff803`6ebd2b5e)
```

如上我们找到了 KiSystemServiceUser 的地址以后，在 KiSystemServiceUser 向下搜索可找到 KiSystemServiceRepeat 里面就是我们要找的 SSDT 表基址。

其中 fffff8036ef8c880 则是 SSDT 表的基地址，紧随其后的 fffff8036ef74a80 则是 SSSDT 表的基地址。



```
Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:10.0.16299.15 AMD64
File Edit View Debug Window Help
Command
nt!KiSystemServiceRepeat
fffff803`6ebd2b94 4c8d15e59c3b00 lea     r10,[nt!KeServiceDescriptorTable (fffff803`6ef8c880)]
fffff803`6ebd2b9b 4c8d1dde1e3a00 lea     r11,[nt!KeServiceDescriptorTableShadow (fffff803`6ef74a80)]
fffff803`6ebd2ba2 f7437880000000 test    dword_ptr [rbx+78h],80h
fffff803`6ebd2ba9 7413                      je      nt!KiSystemServiceRepeat+0x2a (fffff803`6ebd2bbe) Branch
nt!KiSystemServiceRepeat+0x17:
fffff803`6ebd2bab f7437800002000 test    dword_ptr [rbx+78h],200000h
fffff803`6ebd2bb2 7407                      je      nt!KiSystemServiceRepeat+0x27 (fffff803`6ebd2bbb) Branch
nt!KiSystemServiceRepeat+0x20:
fffff803`6ebd2bb4 4c8d1d051f3a00 lea     r11,[nt!KeServiceDescriptorTableFilter (fffff803`6ef74ac0)]
nt!KiSystemServiceRepeat+0x27:
fffff803`6ebd2bbb 4d8bd3                      mov     r10,r11
```

那么如果将这个过程通过代码的方式来实现，我们还需要使用《驱动开发：内核枚举IoTimer定时器》中所使用的特征码定位技术，如下我们查找这段特征。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#pragma intrinsic(__readmsr)

ULONGLONG ssdt_address = 0;

// 获取 KeServiceDescriptorTable 首地址
ULONGLONG GetLySharkCOMKeServiceDescriptorTable()
{
    // 设置起始位置
    PCHAR StartSearchAddress = (PCHAR)__readmsr(0xC0000082) - 0x1806FE;

    // 设置结束位置
    PCHAR EndSearchAddress = StartSearchAddress + 0x100000;
    DbgPrint("[LyShark Search] 扫描起始地址: %p --> 扫描结束地址: %p \n",
        StartSearchAddress, EndSearchAddress);

    PCHAR ByteCode = NULL;

    UCHAR OpCodeA = 0, OpCodeB = 0, OpCodeC = 0;
    ULONGLONG addr = 0;
    ULONG templong = 0;

    for (ByteCode = StartSearchAddress; ByteCode < EndSearchAddress; ByteCode++)
    {
        // 使用MmIsAddressValid()函数检查地址是否有页面错误
        if (MmIsAddressValid(ByteCode) && MmIsAddressValid(ByteCode + 1) &&
            MmIsAddressValid(ByteCode + 2))
        {

```

```

        OpCodeA = *ByteCode;
        OpCodeB = *(ByteCode + 1);
        OpCodeC = *(ByteCode + 2);

        // 对比特征值 寻找 nt!KeServiceDescriptorTable 函数地址
        /*
        nt!KiSystemServiceRepeat:
        ffffffff803`6ebd2b94 4c8d15e59c3b00 lea     r10,
[nt!KeServiceDescriptorTable (ffffffff803`6ef8c880)]
        ffffffff803`6ebd2b9b 4c8d1dde1e3a00 lea     r11,
[nt!KeServiceDescriptorTableShadow (ffffffff803`6ef74a80)]
        ffffffff803`6ebd2ba2 f743788000000000 test    dword ptr [rbx+78h],80h
        ffffffff803`6ebd2ba9 7413                                je
nt!KiSystemServiceRepeat+0x2a (ffffffff803`6ebd2bbe) Branch
        */
        if (OpCodeA == 0x4c && OpCodeB == 0x8d && OpCodeC == 0x15)
        {
            // 获取高位地址 ffffffff802
            memcpy(&templong, ByteCode + 3, 4);

            // 与低位 64da4880 地址相加得到完整地址
            addr = (ULONGLONG)templong + (ULONGLONG)ByteCode + 7;
            return addr;
        }
    }
}
return 0;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("驱动程序卸载成功! \n"));
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com");

    ssdt_address = GetLySharkCOMKeServiceDescriptorTable();
    DbgPrint("[LyShark] SSDT = %p \n", ssdt_address);

    DriverObject->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

如上代码中所提及的步骤我想不需要再做解释了，这段代码运行后即可输出SSDT表的基址。

DebugView on \\DESKTOP-B53PAVI (local)		
File Edit Capture Options Computer Help		
#	Time	Debug Print
24	18.90486145	2925468750 - STORMINI: StorNVMe - POWER: ACTIVE
25	19.13070488	hello lyshark.com
26	19.13571739	[LyShark Search] 扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBD2A82
27	19.14109993	[LyShark] SSDT = FFFFF8036EF8C880
28	20.04025650	2935468750 - STORMINI: StorNVMe - POWER: IDLE
29	20.15188217	2936562500 - STORMINI: StorNVMe - POWER: ACTIVE
30	21.17152405	2946718750 - STORMINI: StorNVMe - POWER: IDLE
31	24.79397392	2982812500 - STORMINI: StorNVMe - POWER: ACTIVE
32	24.80368996	2982812500 - STORMINI: StorNVMe - POWER: IDLE
33	25.69594955	2991718750 - STORMINI: StorNVMe - POWER: ACTIVE
34	25.70223618	2991718750 - STORMINI: StorNVMe - POWER: IDLE
35	25.71316338	2991718750 - STORMINI: StorNVMe - POWER: ACTIVE
36	26.72927666	3001875000 - STORMINI: StorNVMe - POWER: IDLE
37	29.84280968	3032968750 - STORMINI: StorNVMe - POWER: ACTIVE
38	29.85138512	3032968750 - STORMINI: StorNVMe - POWER: IDLE
39	29.85584068	3032968750 - STORMINI: StorNVMe - POWER: ACTIVE
40	30.86251831	3042968750 - STORMINI: StorNVMe - POWER: IDLE

如上通过调用 `GetLySharkCOMKeServiceDescriptorTable()` 得到 SSDT 地址以后我们就需要对该地址进行解密操作。

得到 `ServiceTableBase` 的地址后，就能得到每个服务函数的地址。但这个表存放的并不是 SSDT 函数的完整地址，而是其相对于 `ServiceTableBase[Index]>>4` 的数据，每个数据占四个字节，所以计算指定 `Index` 函数完整地址的公式是；

- 在x86平台上: $\text{FuncAddress} = \text{KeServiceDescriptorTable} + 4 * \text{Index}$
- 在x64平台上: $\text{FuncAddress} = [\text{KeServiceDescriptorTable} + 4 * \text{Index}] >> 4 + \text{KeServiceDescriptorTable}$

如下汇编代码就是一段解密代码，代码中 `rcx` 寄存器传入SSDT的下标，而 `rdx` 寄存器则是传入SSDT表基址。

```

48:8BC1          | mov rax,rcx          |
rcx=index
4C:8D12          | lea r10,qword ptr ds:[rdx] | rdx=ssdt
8BF8           | mov edi,eax          |
C1EF 07         | shr edi,7            |
83E7 20         | and edi,20           |
4E:8B1417       | mov r10,qword ptr ds:[rdi+r10] |
4D:631C82       | movsxd r11,dword ptr ds:[r10+rax*4] |
49:8BC3         | mov rax,r11          |
49:C1FB 04      | sar r11,4            |
4D:03D3         | add r10,r11          |
49:8BC2         | mov rax,r10          |
C3             | ret                  |

```

有了解密公式以后代码的编写就变得很容易，如下是读取SSDT的完整代码。

```

// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>

```

```

#pragma intrinsic(__readmsr)

typedef struct _SYSTEM_SERVICE_TABLE
{
    PVOID      ServiceTableBase;
    PVOID      ServiceCounterTableBase;
    ULONGLONG  NumberOfServices;
    PVOID      ParamTableBase;
} SYSTEM_SERVICE_TABLE, *PSYSTEM_SERVICE_TABLE;

ULONGLONG ssdt_base_address;
PSYSTEM_SERVICE_TABLE KeServiceDescriptorTable;

typedef UINT64(__fastcall *SCFN)(UINT64, UINT64);
SCFN scfn;

// 解密算法
VOID DecodeSSDT()
{
    UCHAR strShellCode[36] =
        "\x48\x8B\xC1\x4C\x8D\x12\x8B\xF8\xC1\xEF\x07\x83\xE7\x20\x4E\x8B\x14\x17\x4D\x63\x1C\x82\x49\x8B\xC3\x49\xC1\xFB\x04\x4D\x03\xD3\x49\x8B\xC2\xC3";
    /*
        48:8BC1          | mov rax,rcx          |
rcx=index
        4C:8D12          | lea r10,qword ptr ds:[rdx]          |
rdx=ssdt
        8BF8          | mov edi,eax          |
        C1EF 07        | shr edi,7            |
        83E7 20        | and edi,20           |
        4E:8B1417      | mov r10,qword ptr ds:[rdi+r10]      |
        4D:631C82      | movsxd r11,dword ptr ds:[r10+rax*4] |
        49:8BC3        | mov rax,r11          |
        49:C1FB 04     | sar r11,4            |
        4D:03D3        | add r10,r11          |
        49:8BC2        | mov rax,r10          |
        C3            | ret                  |
    */
    scfn = ExAllocatePool(NonPagedPool, 36);
    memcpy(scfn, strShellCode, 36);
}

// 获取 KeServiceDescriptorTable 首地址
ULONGLONG GetKeServiceDescriptorTable()
{
    // 设置起始位置
    PCHAR StartSearchAddress = (PCHAR)__readmsr(0xC0000082) - 0x1806FE;

    // 设置结束位置
    PCHAR EndSearchAddress = StartSearchAddress + 0x8192;
    DbgPrint("扫描起始地址: %p --> 扫描结束地址: %p \n", StartSearchAddress, EndSearchAddress);

    PCHAR ByteCode = NULL;

```

```

    UCHAR OpCodeA = 0, OpCodeB = 0, OpCodeC = 0;
    ULONGLONG addr = 0;
    ULONG templong = 0;

    for (ByteCode = StartSearchAddress; ByteCode < EndSearchAddress; ByteCode++)
    {
        // 使用MmIsAddressValid()函数检查地址是否有页面错误
        if (MmIsAddressValid(ByteCode) && MmIsAddressValid(ByteCode + 1) &&
MmIsAddressValid(ByteCode + 2))
        {
            OpCodeA = *ByteCode;
            OpCodeB = *(ByteCode + 1);
            OpCodeC = *(ByteCode + 2);

            // 对比特征值 寻找 nt!KeServiceDescriptorTable 函数地址
            // LyShark.com
            // 4c 8d 15 e5 9e 3b 00 lea r10,[nt!KeServiceDescriptorTable
(fffff802`64da4880)]
            // 4c 8d 1d de 20 3a 00 lea r11,[nt!KeServiceDescriptorTableShadow
(fffff802`64d8ca80)]
            if (OpCodeA == 0x4c && OpCodeB == 0x8d && OpCodeC == 0x15)
            {
                // 获取高位地址fffff802
                memcpy(&templong, ByteCode + 3, 4);

                // 与低位64da4880地址相加得到完整地址
                addr = (ULONGLONG)templong + (ULONGLONG)ByteCode + 7;
                return addr;
            }
        }
    }
    return 0;
}

// 得到函数相对偏移地址
ULONG GetOffsetAddress(ULONGLONG FuncAddr)
{
    ULONG dwtmp = 0;
    PULONG ServiceTableBase = NULL;
    if (KeServiceDescriptorTable == NULL)
    {
        KeServiceDescriptorTable =
(PSYSTEM_SERVICE_TABLE)GetKeServiceDescriptorTable();
    }
    ServiceTableBase = (PULONG)KeServiceDescriptorTable->ServiceTableBase;
    dwtmp = (ULONG)(FuncAddr - (ULONGLONG)ServiceTableBase);
    return dwtmp << 4;
}

// 根据序号得到函数地址
ULONGLONG GetSSDTFunctionAddress(ULONGLONG NtApiIndex)
{
    ULONGLONG ret = 0;
    if (ssdt_base_aaddress == 0)
    {

```

```

        // 得到ssdt基地址
        ssdt_base_address = GetKeServiceDescriptorTable();
    }
    if (scfn == NULL)
    {
        DecodeSSDT();
    }
    ret = scfn(NtApiIndex, ssdt_base_address);
    return ret;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("驱动程序卸载成功! \n"));
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    ULONGLONG ssdt_address = GetKeServiceDescriptorTable();
    DbgPrint("SSDT基地址 = %p \n", ssdt_address);

    // 根据序号得到函数地址
    ULONGLONG address = GetSSDTFunctionAddress(51);
    DbgPrint("[LyShark] NtOpenFile地址 = %p \n", address);

    // 得到相对SSDT的偏移量
    DbgPrint("函数相对偏移地址 = %p \n", GetOffsetAddress(address));

    DriverObject->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

运行后即可得到 SSDT 下标为 51 的函数也就是得到 NtOpenFile 的绝对地址和相对地址。

#	Time	Debug Print
11	4.58826303	hello lyshark.com
12	4.59556484	扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBDAC14
13	4.60104418	SSDT基地址 = FFFFF8036EF8C880
14	4.60638428	扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBDAC14
15	4.61171055	[LyShark] NtOpenFile地址 = FFFFF8036F0304D0
16	4.61703539	扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBDAC14
17	4.62237120	函数相对偏移地址 = 00000000020B8C00
18	4.64895201	13027812500 - STORMINI: StorNVMe - POWER: ACTIVE
19	5.65780687	13037968750 - STORMINI: StorNVMe - POWER: IDLE
20	5.86589098	驱动程序卸载成功!
21	6.67477417	13045312500 - STORMINI: StorNVMe - POWER: ACTIVE
22	6.68241358	13045312500 - STORMINI: StorNVMe - POWER: IDLE
23	6.68919277	13045312500 - STORMINI: StorNVMe - POWER: ACTIVE
24	7.69535208	13055312500 - STORMINI: StorNVMe - POWER: IDLE
25	8.67695332	13065156250 - STORMINI: StorNVMe - POWER: ACTIVE
26	8.68396473	13065156250 - STORMINI: StorNVMe - POWER: IDLE

你也可以打开ARK工具，对比一下是否一致，如下图所示，LyShark 的代码是没有任何问题的。

进程	驱动模块	内核层	内核钩子	应用层钩子	设置	监控	启动信息	注册表	服务	文件	网络	调试引擎
SSDT	Shadow SSDT	内核钩子	系统中断表	Object钩子								

索引	函数名	原始函数地址	钩子类型	当前函数地址
51	NtOpenFile	0xFFFFF8036F0304D0	-	0xFFFFF8036F0304D0
52	NtDelayExecution	0xFFFFF8036EFEB160	-	0xFFFFF8036EFEB160
53	NtQueryDirectoryFile	0xFFFFF8036F004980	-	0xFFFFF8036F004980
54	NtQuerySystemInformation	0xFFFFF8036EFE14E0	-	0xFFFFF8036EFE14E0

DebugView on \\DESKTOP-B53PAVI (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
11	4.58826303	hello lyshark.com
12	4.59556484	扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBDAC14
13	4.60104418	SSDT基地址 = FFFFF8036EF8C880
14	4.60638428	扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBDAC14
15	4.61171055	[LyShark] NtOpenFile地址 = FFFFF8036F0304D0
16	4.61703539	扫描起始地址: FFFFF8036EBD2A82 --> 扫描结束地址: FFFFF8036EBDAC14
17	4.62237120	函数相对偏移地址 = 00000000020B8C00
18	4.64895201	13027812500 - STORMINI: StorNVMe - POWER: ACTIVE
19	5.65780687	13037968750 - STORMINI: StorNVMe - POWER: IDLE
20	5.86589098	驱动程序卸载成功!

本书作者: 王瑞 (LyShark)

作者邮箱: me@lyshark.com

作者博客: <https://lyshark.cnblogs.com>

团队首页: www.lyshark.com