RSAConference2022
San Francisco & Digital | June 6 – 9

TRANSFORM

SESSION ID: HTA-T02

My Fuzzy Driver

Eran Shimony
Senior Security Researcher
CyberArk
@EranShimony

Mark Cherp
Senior Security Researcher
CyberArk
@OcamRazr

Disclaimer

Presentations are intended for educational purposes only and do not replace independent professional judgment. Statements of fact and opinions expressed are those of the presenters individually and, unless expressly stated to the contrary, are not the opinion or position of RSA Conference LLC or any other co-sponsors. RSA Conference does not endorse or approve, and assumes no responsibility for, the content, accuracy or completeness of the information presented.

Attendees should note that sessions may be audio- or video-recorded and may be published in various media, including print, audio and video formats without further notice. The presentation template and any media capture are subject to copyright protection.

RSAConference2022

TL;DR

kAFL vs Drivers == Bugs

Our Menu

Target

What to attack and how?

Internals

Healthy dose of Windows

Bugs

Vulnerabillities deep dive

Automation

Discovery, harness and grammar

Fuzzing

kAFL setup with some tweaks

RSAConference2022

Target

What to attack and why?

Don’t run away from Windows

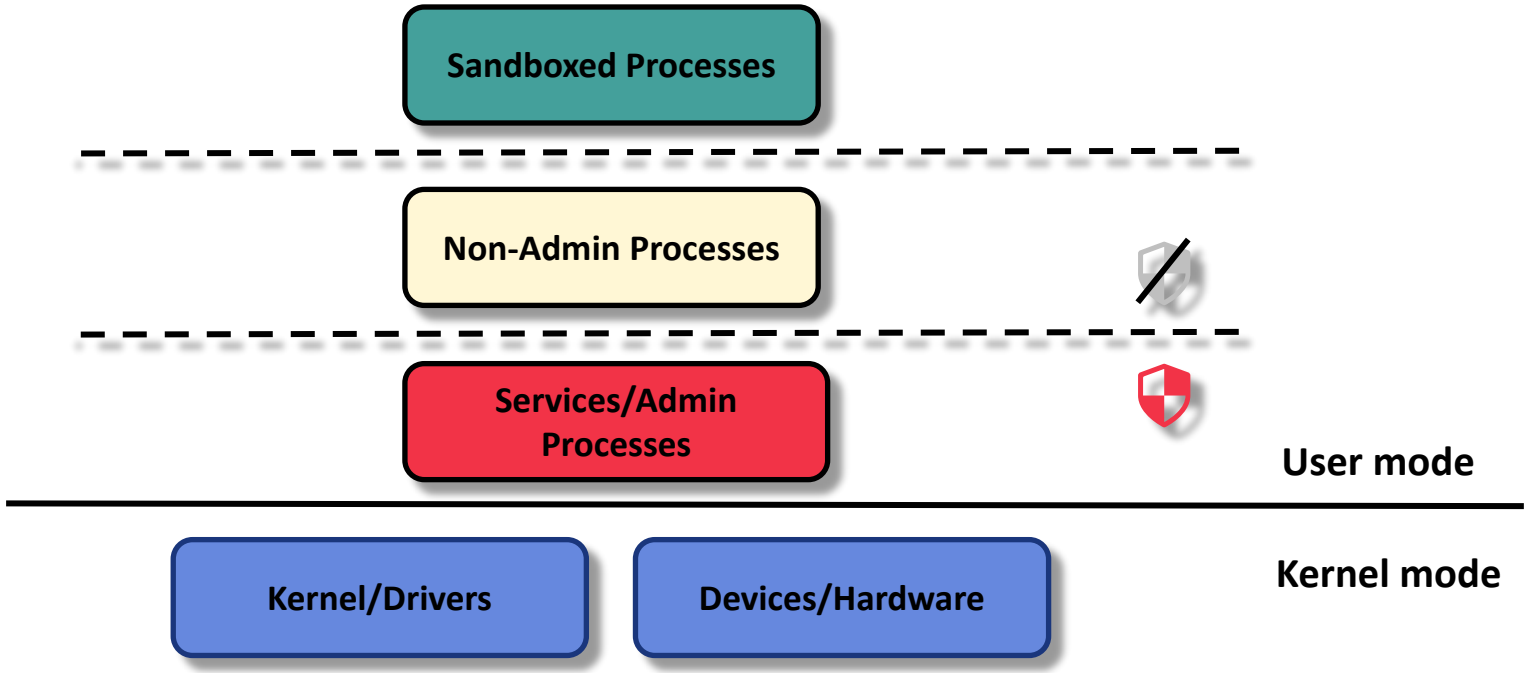
- We aim to escalate privilege from a weak point, accessible drivers are a good candidate
- Windows has more than 300 drivers with tons of legacy code written in C
- Windows is “considered” to be closed source, which makes it harder and less explored
- Windows drivers might be challenging but they are also rewarding ☺

RSAConference2022

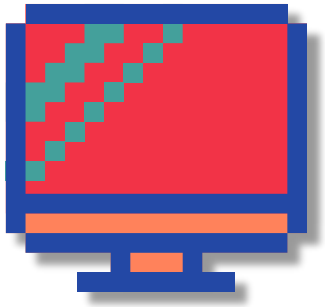
Internals

Healthy dose of Windows

Windows' privilege level architecture



Windows Drivers 101

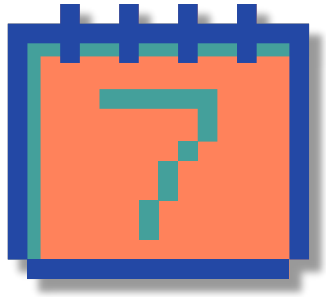


Part of the kernel

- An extension of the Windows kernel
- Share System's address space
- Essentially a part of the kernel

Signing

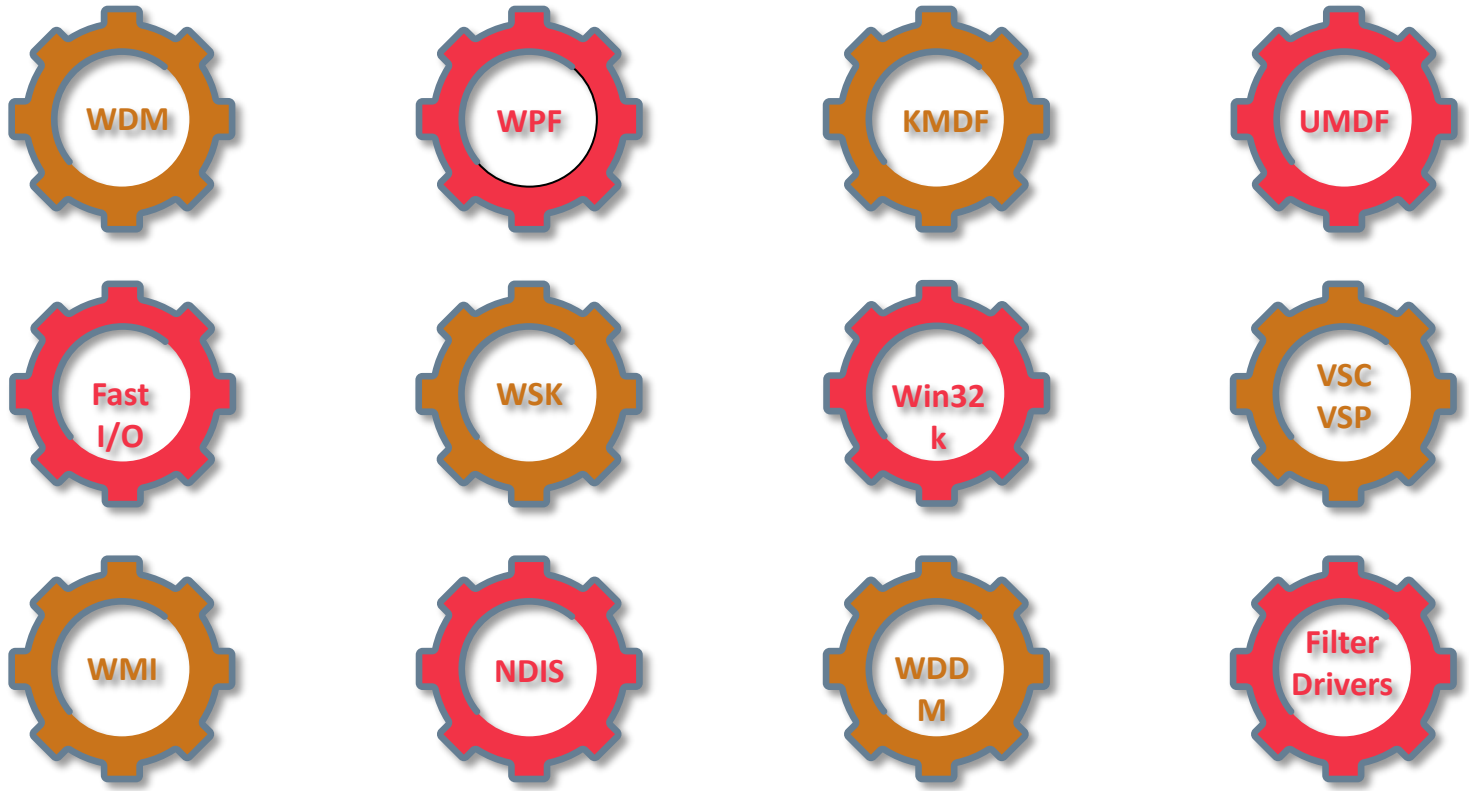
- Must be digitally signed
- Driver signing can be disabled
- Requires Admin+



Responsibility

- Talking with hardware, devices
- Filtering
- Not blue screening your machine

Too many Driver models



WDM Drivers

PERSONAL DATA

Age	23 Years
Usage	Abundant
Successor	KMDF
Location	Windows OS
Amount	Over 200

STATS

PERCENTAGE OF TOTAL

PURPOSE

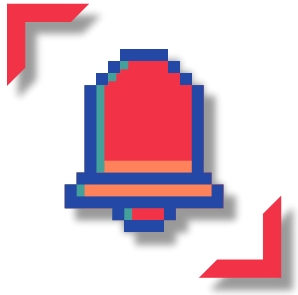
- Talking with devices, power management…
- Supply kernel support for applications

RSAConference2022

Bugs

Vulnerabillities deep dive

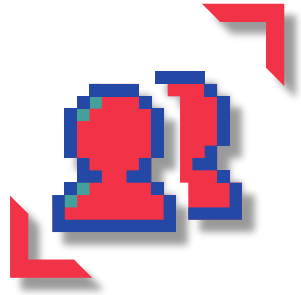
Driver Bug Hunting 101



Access

Which driver do I have access to?

What are my permissions?



Communication

How do we communicate with the driver?

How does it parse user data?

Driver's Anatomy I

Access

In Windows, you talk to a driver via a device

The driver creates a device, and it should specify which users can access it. It uses SDDL string for it, or defines it in the inf file:

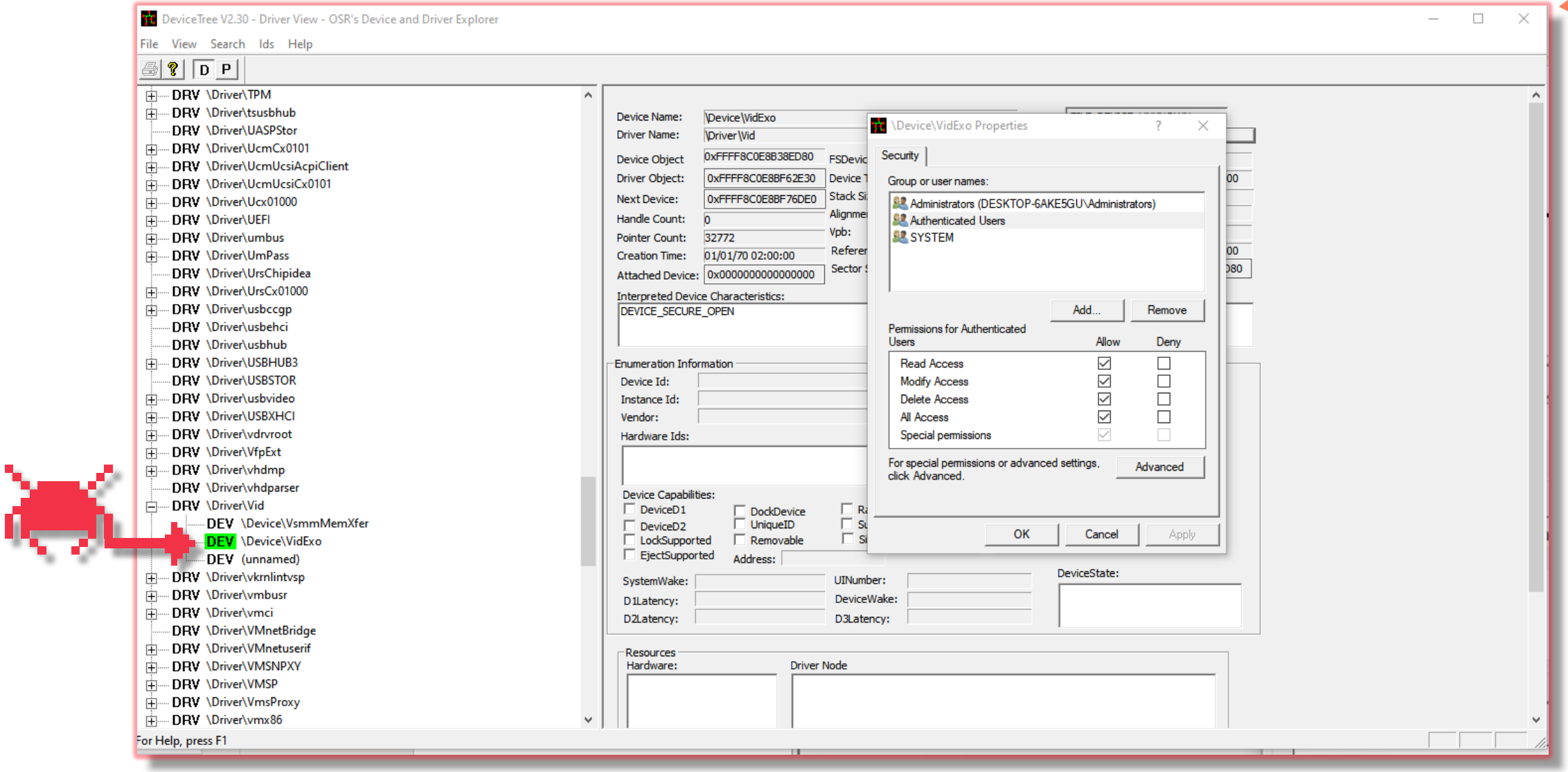
- *IoCreateDevice*() or *IoCreateDeviceSecure*()

Followed by exposing the device to the user via calling:

- *IoCreateSymbolicLink() or IoRegisterDeviceInterface()*

The user opens a handle to the device via calling *CreateFile*()*NtCreateFile*()

Show me some permissions



Driver's Anatomy II

Communication

The Driver object registers dispatch routines through *DriverObject->MajorFunction[IRP_MJ_XXX]*

The Dispatch routines are invoked by the *IoManager* when the following operations are done on the device:

- Create - *NtCreateFile*
- Close - *NtCloseFile*
- Read - *NtReadFile*
- Write - *NtWriteFile*
- Device_Control – *DeviceIoControl -> FileIoControlDevice*

These dispatch routines our initial go-to places

Typical DriverEntry

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING  RegistryPath)
{
    UNICODE_STRING DeviceName, SymbolicLink, sddlString;
    PDEVICE_OBJECT deviceObject;
    RtlInitUnicodeString(&DeviceName, L"\\Device\\testydrv");
    RtlInitUnicodeString(&SymbolicLink, L"\\DosDevices\\testydrv");
    RtlInitUnicodeString(&sddlString, L"D:P(A;;GA;;;SY)(A;;GA;;;BA)");

    //Create a device
    IoCreateDevice(DriverObject, 65535, &DeviceName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &deviceObject);
    //IoCreateDeviceSecure(DriverObject, sizeof(65533), &DeviceName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &sddlString, NULL, &deviceObject);
    //Create a symbolic so the user can access the device
    IoCreateSymbolicLink(&SymbolicLink, &DeviceName);

    //Populating Driver's object dispatch table
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = TestyDispatchIoctl;
    DriverObject->MajorFunction[IRP_MJ_CREATE] = TestyDispatchCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = TestyDispatchClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = TestyDispatchRead;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = TestyDispatchWrite;
    DriverObject->MajorFunction[IRP_MJ_CLEANUP] = TestyDispatchCleanup;
    DriverObject->DriverUnload = TestyUnloadDriver;
    return STATUS_SUCCESS;
}
```

Dispatch Routine

```
NTSTATUS TestyDispatchIoctl(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    PIO_STACK_LOCATION  CurrentStackLocation;
    NTSTATUS            ntStatus = STATUS_SUCCESS;
    PVOID               SystemBuffer = NULL;
    ULONG               InputBufferLength = 0;
    ULONG               OutputBufferLength = 0;
    DWORD               IoControlCode = 0;

    CurrentStackLocation = IoGetCurrentIrpStackLocation(Irp);
    InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
    OutputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength;
    SystemBuffer = Irp->AssociatedIrp.SystemBuffer;
    IoControlCode = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
```

IRPs

I/O request packet, is a struct that has all the parameters any of the dispatch routines would ever need, generated when an action is performed on a device

The driver receives a pointer to the generated **IRP structure** by the *IoManager*, it consists of:

StackLocation – holds many important members

Requestor mode – Kernel or User

Buffers – Depends on the transfer type

IoStatus.Information – How many bytes are written to the output buffer

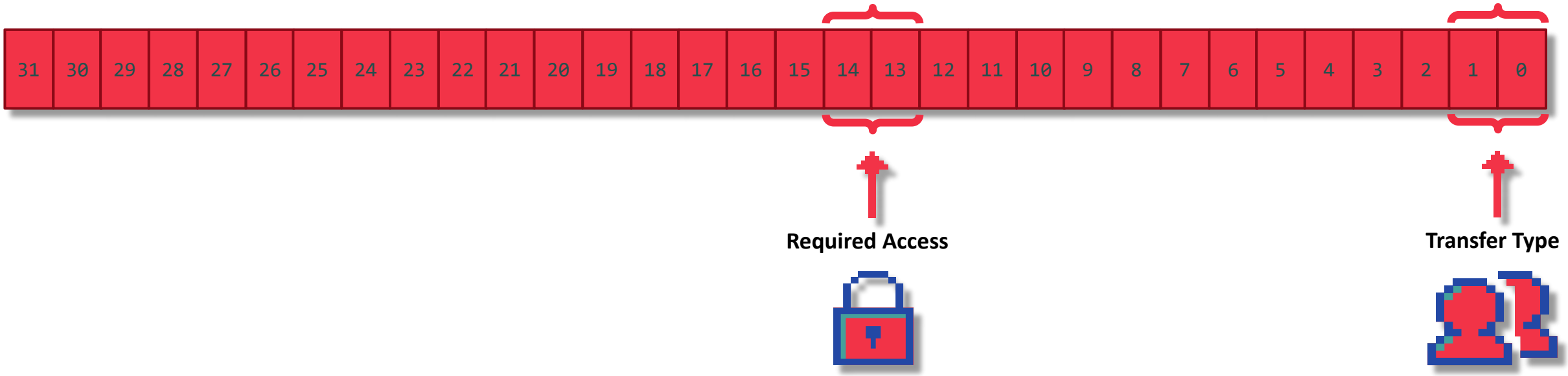
A misuse of any of these fields would probably cause a bug

How to generate IRPs

```
BOOL DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);
```

IOCTL Code

32-bit IOCTL

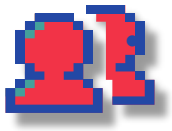


Required Access

Two bit number – describes the required permissions to send the *DeviceIoControl* request, it is based on your permissions in the call to *NtCreateFile*.

- There are four different options:
  - *FILE_ANY_ACCESS*
  - *FILE_READ_ACCESS*
  - *FILE_WRITE_ACCESS*
  - *FILE_READ_ACCESS | FILE_WRITE_ACCESS*

Transfer Type



The two least significant bits – describe how the *IoManager* treats user's data, there are many nuances the driver developer must know

- There are four different options:
  - *METHOD_NEITHER*
  - *METHOD_BUFFERED*
  - *METHOD_IN_DIRECT*
  - *METHOD_OUT_DIRECT*

METHOD_NEITHER = Mother of all evil

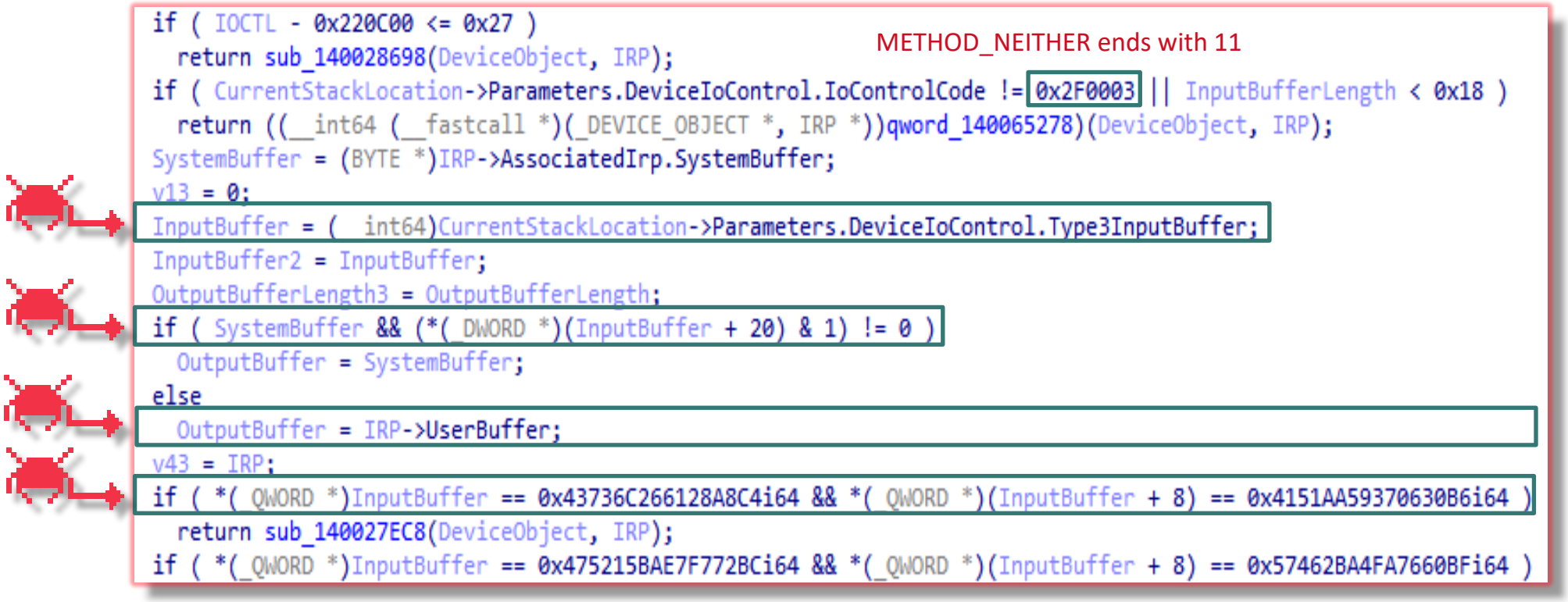
- The two bits are on -> ioctl number ends with 11
- The IoManager is lazy, the buffers and their lengths reside in User-Mode, the kernel does not copy them to kernel space, they can be paged-out
- The input and output buffers are:
  - *Irp->CurrentStackLocation.Parameters.DeviceIoControl.Type3InputBuffer*
  - *Irp->UserBuffer*
- The user can allocate and deallocate the buffers making the pages invalid, so you must be cautious when dealing with them
- Every access to the buffers must be in a try except block

Probing

- Probing validates that an address resides in User-Mode:
  - *ProbeForRead* (*Address, Length, Alignment)
  - *ProbeForWrite* (*Address, Length, Alignment)

- But if you don’t probe or you do not use the function correctly, then:
  - It throws an exception on invalid address so the call must be inside a try except block
  - If Length is 0 it does nothing, it passes the validation without probing
  - If not probed and the kernel reads form it = **BSoD or info leak**
  - If not probed and the kernel writes to it = **BSoD or arbitrary write**

Bug Example I



Method_Buffered

- The two bits are off -> ioctl number ends with 00
- In *METHOD_BUFFERED* the IoManager copies the buffers and their lengths to the kernel in a secure manner. Therefore, they don't reside in the user's memory space.
- No need for probing, but if the buffer has embedded fields, like pointer addresses, lengths and so on, they need to be treated properly!
- The IRP buffer is used both for input and output:
  - *Irp->AssociatedIrp.SystemBuffer*

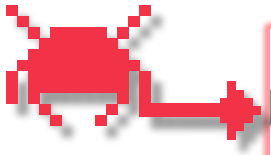
Method_Buffered - Continued

- The *Irp->IoStatus.Information* indicates how many bytes are to be copied to user's *OutputBuffer*

- If *Irp->IoStatus.Inforamtion > InputBufferLen* and *OutputBufferLen > InputBufferLen*
  - The rest of the system buffer data copied to (*OutputBuffer*) is uninitialized data
  - *SizeOfData = OutputBufferLen – InputBufferLen*

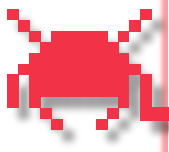
- Would cause kernel leak unless the buffer is properly initialized
  - Assuming:
    - *OutputBufferLen = 0x1000, InputBufferLen = 0x8*
    - *Irp->IoStatus.Inforamtion = OutputBufferLen*
  - Leakage of 0xFF8 bytes to user-mode

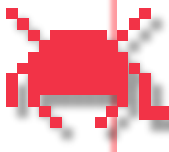
Bug Example II



```
DisptachIoctlFDO (PDEVICE_OBJECT Device_Object, IRP *IRP)
{
case 0x2D2324u:
      NTStatus2 = DoSomething(Device_Object, IRP);
...

    __int64 __fastcall DoSomething(PDEVICE_OBJECT Device_Object, PIRP irp)
    {
     _IO_STACK_LOCATION *CurrentStackLocation; // rbx
      PVOID Device_Extension; // rbp
      BYTE SystemBuffer; // si
      __int64 InputBufferLength; // r9
      ULONG OutputBufferLength; // [rsp+50h] [rbp+8h] BYREF

      CurrentStackLocation = irp->Tail.Overlay.CurrentStackLocation;
        Device_Extension = Device_Object->DeviceExtension;
      irp->IoStatus.Information = 0i64;
      SystemBuffer = (BYTE *)irp->AssociatedIrp.SystemBuffer;
      InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
      OutputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength;

      if ( OutputBufferLength >= 0x107C0 && *(_QWORD *)&SystemBuffer )
       {
          FillOutputDataAndRestWithZeros((__int64)Device_Extension, *(_BYTE **)&SystemBuffer, &OutputBufferLength);
          IRP->IoStatus.Information = OutputBufferLength;
          return 0i64;
       }
      else
      {
       IRP->IoStatus.Information = 0x107C0i64;
        return 0x80000005i64;
```

METHOD_BUFFRED ends with 00





Direct I/O

- The Right bit is on = *METHOD_IN_DIRECT* and left is off 01
- The Left bit is on = *METHOD_OUT_DIRECT* and right is off 10

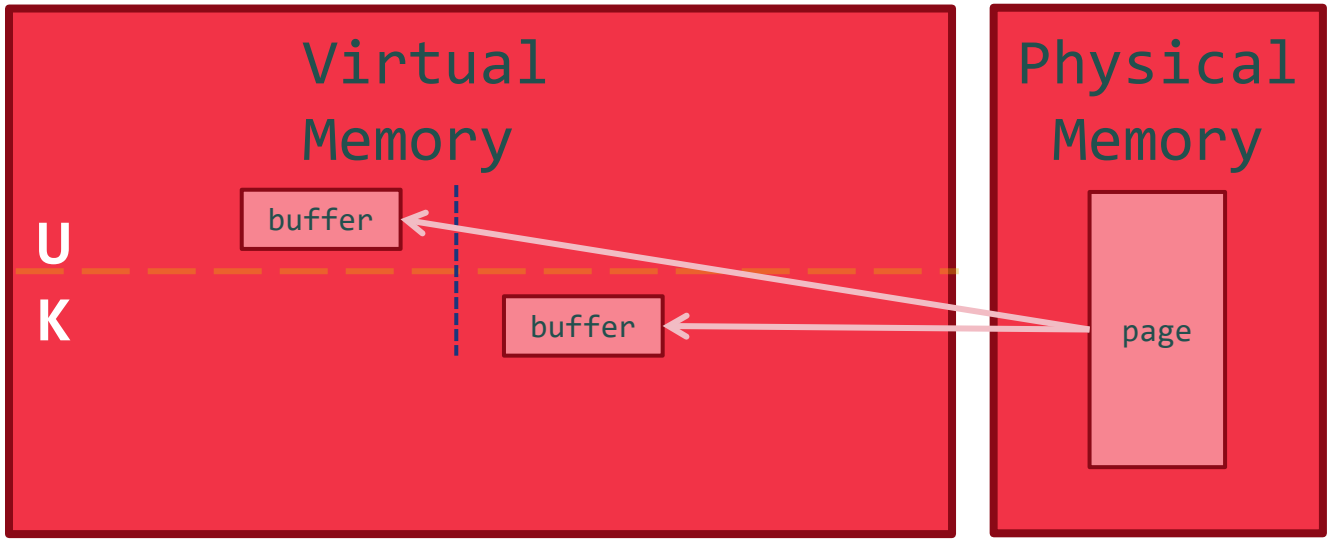
- The IRP supply two pointer to buffers, both in kernel:
  - *Irp->AssociatedIrp.SystemBuffer* – The input buffer
  - *Irp->MdlAddress* – A second buffer used or buffered

- The second buffer is the “direct” which always paged = faster access, this buffer can be used for input or output

- The kernel can also create an MDL with *IoCreateMdl*() followed by locking it with *MmProbeAndLockPages*() which would throw an exception on invalid page

Direct I/O MDL

- *"An MDL is a structure that describes the fixed physical memory locations that comprise a contiguous data buffer in virtual memory"*
- In simpler words, The MDL describes the data buffer at a fixed position in physical memory, which will always be paged in and locked (you need to do that) in memory. It is a double mapping, one for the user, and another for the kernel



Bug Example III

- Most of the bugs involve not checking for null, as the user can send a null buffer, thus causing *MmGetSystemAddressForMDLSafe*() to be sad
- Also, if creating an MDL yourself in the kernel, make sure you use the correct virtual address of a buffer not like in here:

```
ControlableVirtualAddress = (void *)*((_QWORD *)SystemBuffer2 + 3);
if ( ControlableVirtualAddress )
{
  AllocatedMdl = IoAllocateMdl(ControlableVirtualAddress, dwMdlSize, 0, 0, Irp);
  IrqlLevel = GetIRQLLevel(KeGetCurrentIrql());
  LODWORD(Irpb) = *((_DWORD *)SystemBuffer2 + 3);
  // Local pages in physcial memory, would crash on a bad address
  MmProbeAndLockPages(AllocatedMdl, 0, IoModifyAccess);
  if ( (AllocatedMdl->MdlFlags & 5) != 0 )
    LocalMappedSystemVA = AllocatedMdl->MappedSystemVa;
  else
    LocalMappedSystemVA = MmMapLockedPagesSpecifyCache(AllocatedMdl, 0, MmCached, 0i64, 0, dword_1400E8D84 | 0x10u);
}
```

How to Be Lazy

Enumerate

Enumerate device names and discover IOCTLs

Generate

Generate harness

Fuzz

Fuzz with the generated harness

RSAConference2022

Automation

Discovery, harness and grammar

Device Handles and IOCTLs Discovery

Enumerating All Device Drivers in the System

05/31/2018 • 2 minutes to read •

The following sample code uses the EnumDeviceDrivers function to enumerate the current device drivers in the system. It passes the load addresses retrieved from this function call to the GetDeviceDriverBaseName function to retrieve a name that can be displayed.

C++ Copy

```
#include <windows.h>
#include <psapi.h>
#include <tchar.h>
#include <stdio.h>

// To ensure correct resolution of symbols, add Psapi.lib to TARGETLIBS
// and compile with -DPSAPI_VERSION=1

#define ARRAY_SIZE 1024

int main( void )
{
    LPVOID drivers[ARRAY_SIZE];
    DWORD cbNeeded;
    int cDrivers, i;

    if( EnumDeviceDrivers(drivers, sizeof(drivers), &cbNeeded) && cbNeeded < sizeof(drivers))
    {
        TCHAR szDriver[ARRAY_SIZE];

        cDrivers = cbNeeded / sizeof(drivers[0]);

        _tprintf(TEXT("There are %d drivers:\n"), cDrivers);
        for (i=0; i < cDrivers; i++ )
        {
            if(GetDeviceDriverBaseName(drivers[i], szDriver, sizeof(szDriver) /              sizeof(szDriver[0])))
            {
                _tprintf(TEXT("%d: %s\n"), i+1, szDriver);
            }
        }
    }
    else
    {
        _tprintf(TEXT("EnumDeviceDrivers failed; array size needed is %d\n"),              cbNeeded / sizeof(LPVOID));
        return 1;
    }

return 0;
}
```

The **SetupDiGetClassDevs** function returns a handle to a device information set that contains requested device information elements for a local computer.

Syntax

C++ Copy

```
WINSETUPAPI HDEVINFO SetupDiGetClassDevsW(
  const GUID *ClassGuid,
  PCWSTR     Enumerator,
  HWND       hwndParent,
  DWORD      Flags
);
```

Device Console (DevCon.exe) Commands

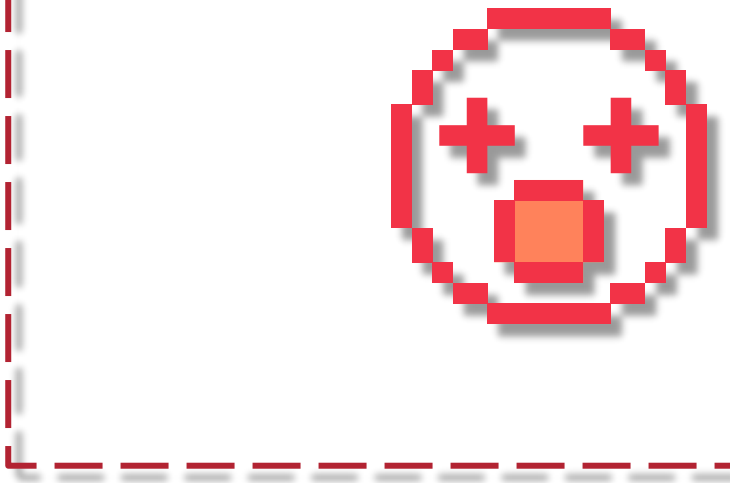
04/20/2017 • 4 minutes to read •

DevCon (DevCon.exe) is a command line tool that can display detailed information about devices on computers running Windows. You can also use DevCon to enable, disable, install, configure, and remove devices. DevCon uses the following syntax.

Copy

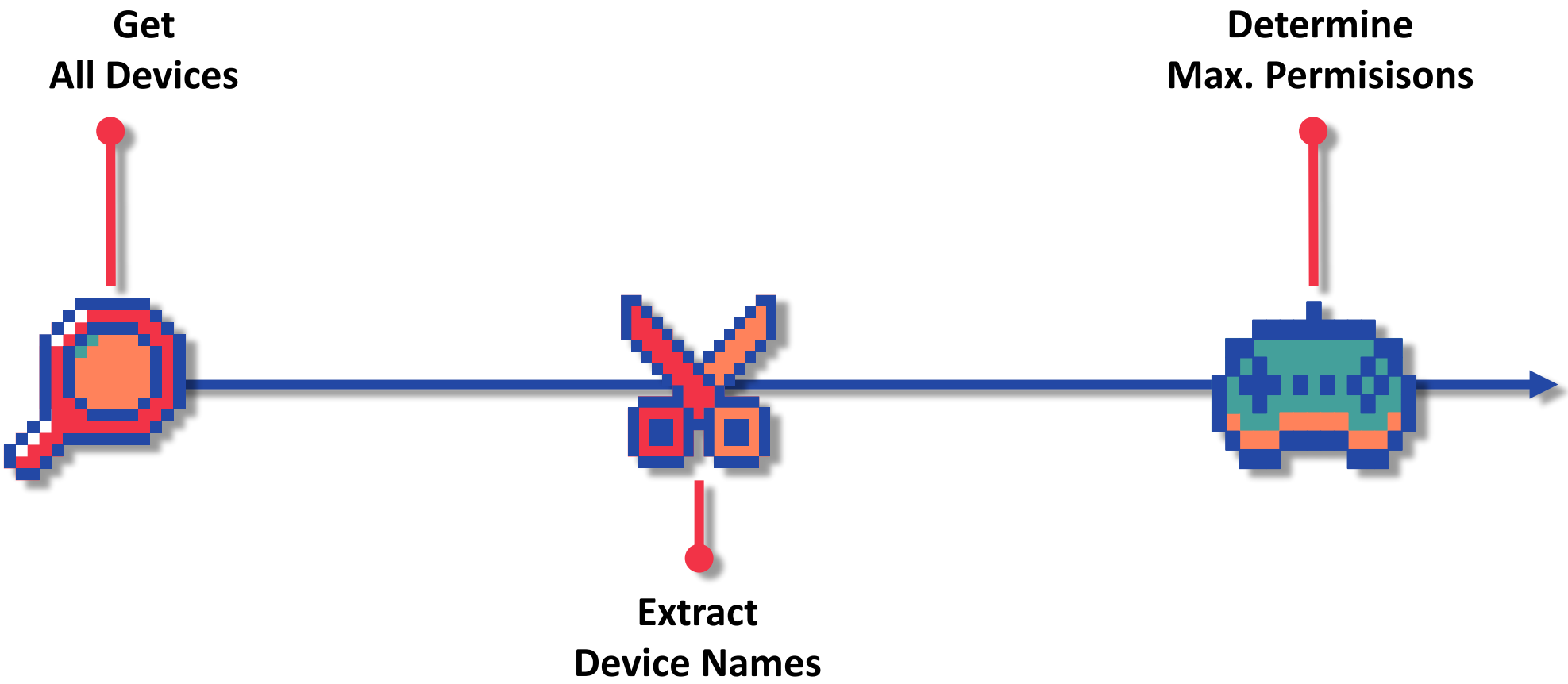
```
devcon [/m:\\computer] [/r] command [arguments]
```

Discover Devices

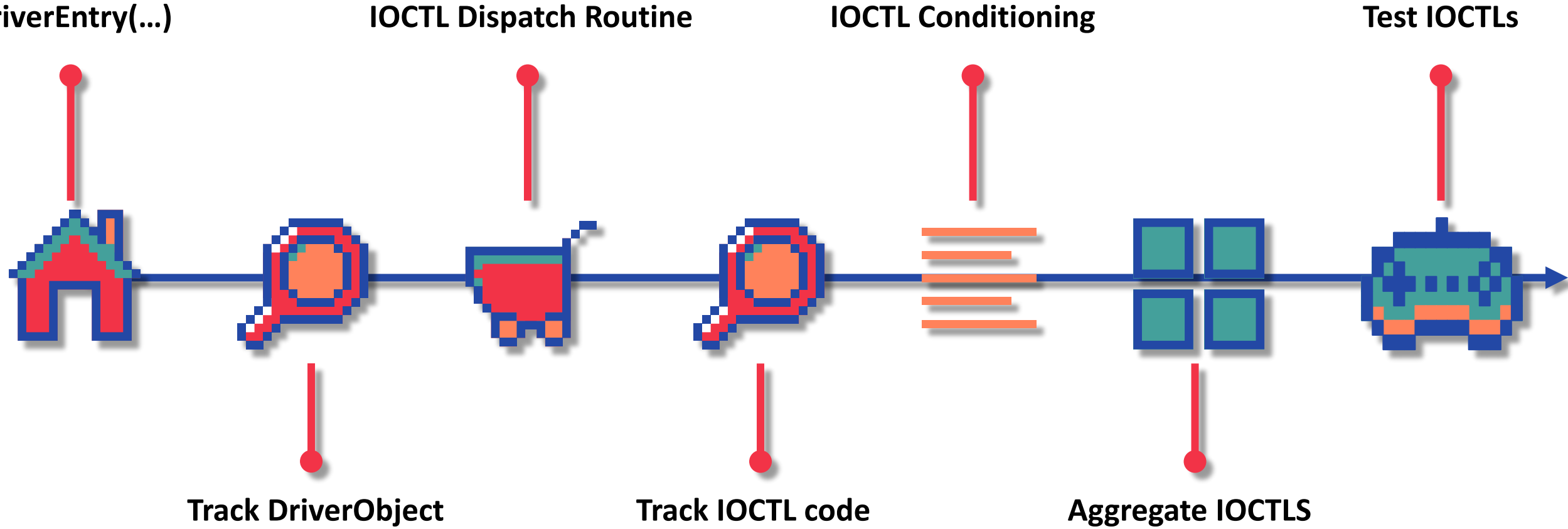


Enumurate Device Names and Discover IOCTLS

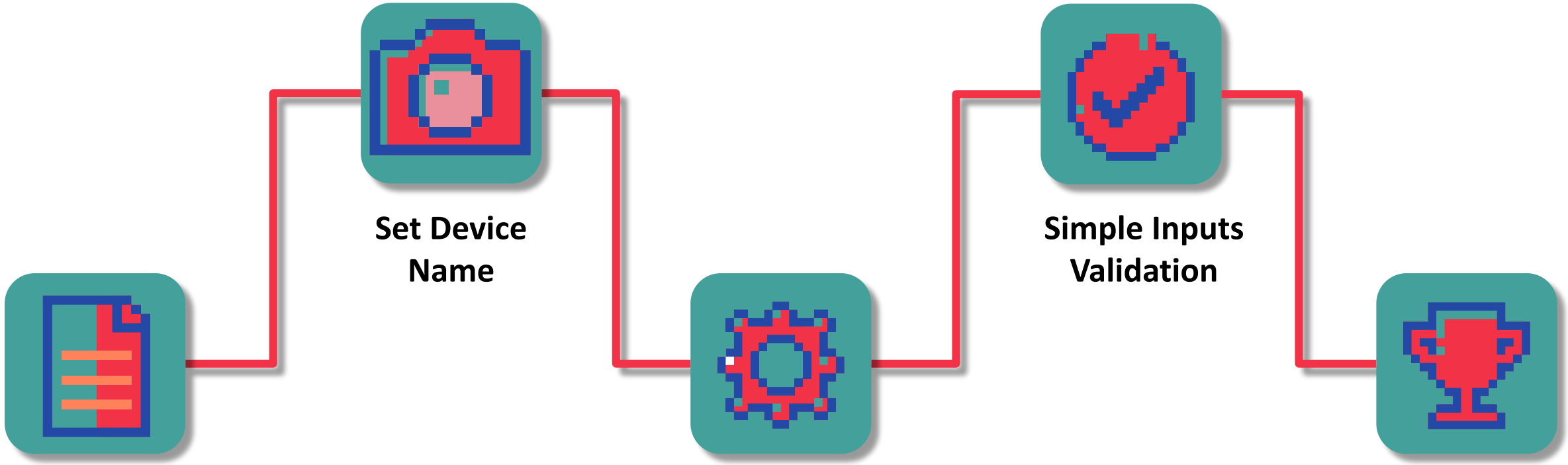
Discovering Accessible Devices



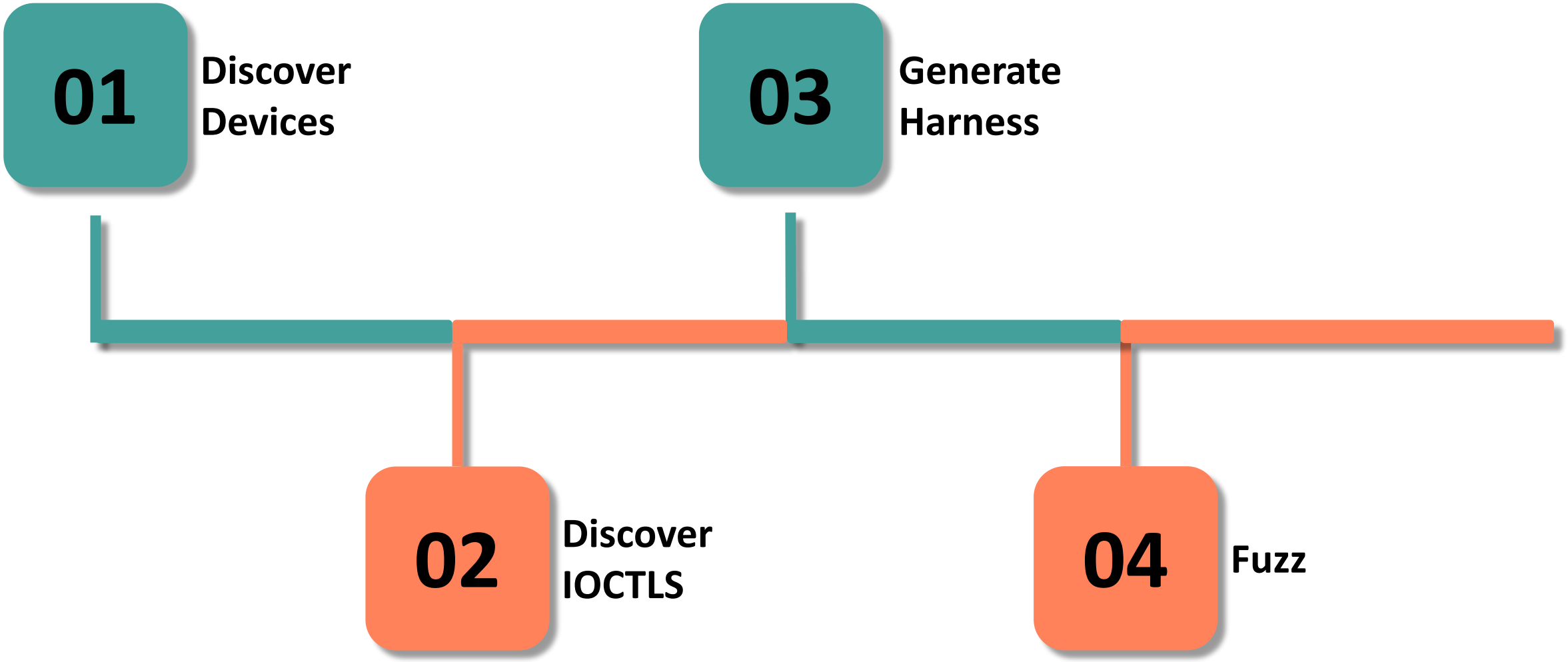
IOCTLs Discovery



Harness Generation



Automation Summary



Grammar 101

- After discovering some bugs with a simple grammar, we started examining more advanced approaches

- We know the shape of the “high-level” input passed to the IOCTL
- We don’t know how the “deserialized” data inside the driver looks like

- We want to guide our Fuzzer towards relevant inputs and that’s where grammar comes into play
- Grammar is essentially a way to tell the Fuzzer which inputs are more likely to increase coverage

Grammar Approaches

Simple template matching (i.e. Regex)

```
// Regex to match phone numbers
"^[+]*[(]{0,1}[0-9]{1,4}[)]{0,1}[-\\s\\./0-9]*$"
```

```
// Possible matches
"+(123) - 456-78-90"
"+972-548099912"
```

Input-to-state correspondence (i.e. Redqueen)

```
//Try Input = "SEEDVALUE"
cmp eax, "ABCD"
```

```
//Observe
eax == "VALU"
```

```
//Replace
"SEEDVALUE" with "SEEDABCDE"
```

Large-scale mutation (i.e. Grimoire)

```
# Original statement
print ("aabbccdd")
```

```
# Mutated
print ("aabbccdd")
```

```
# Mutated new paths
print ("aabbccdd")
```

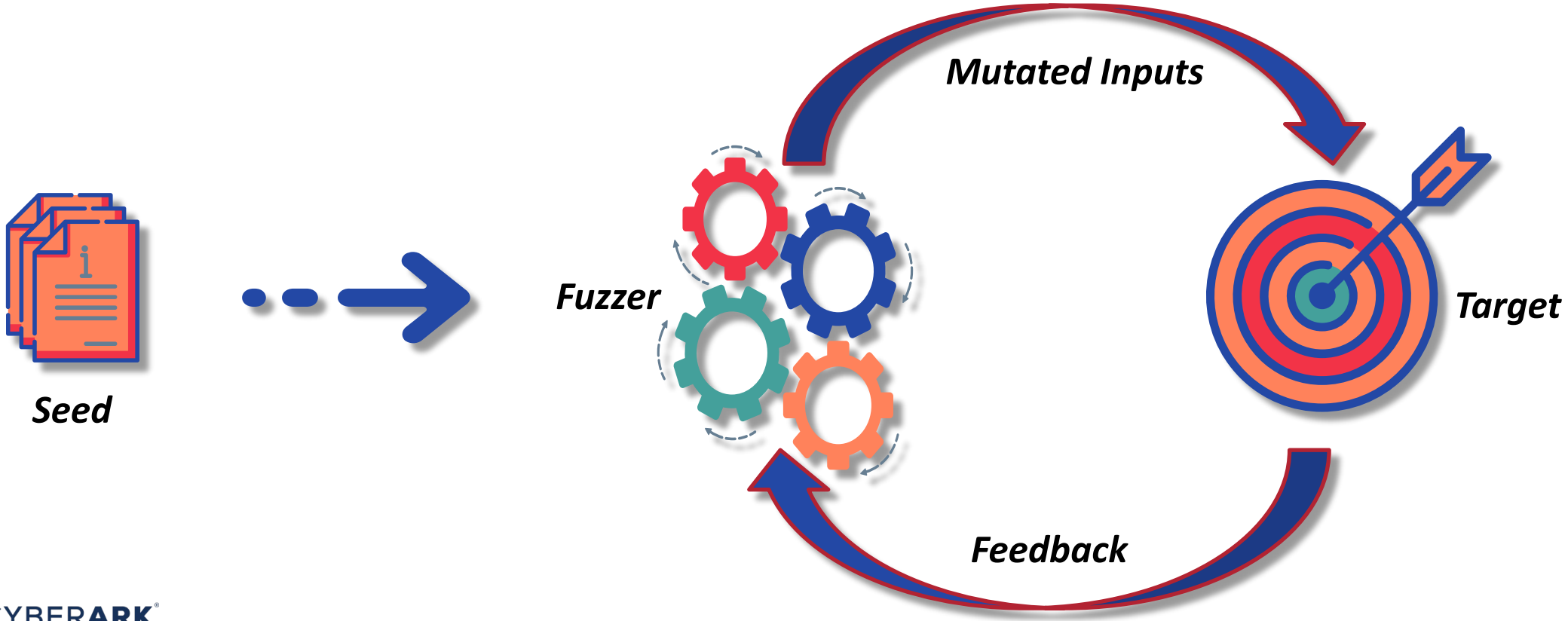
RSAConference2022

Kernel Fuzzing

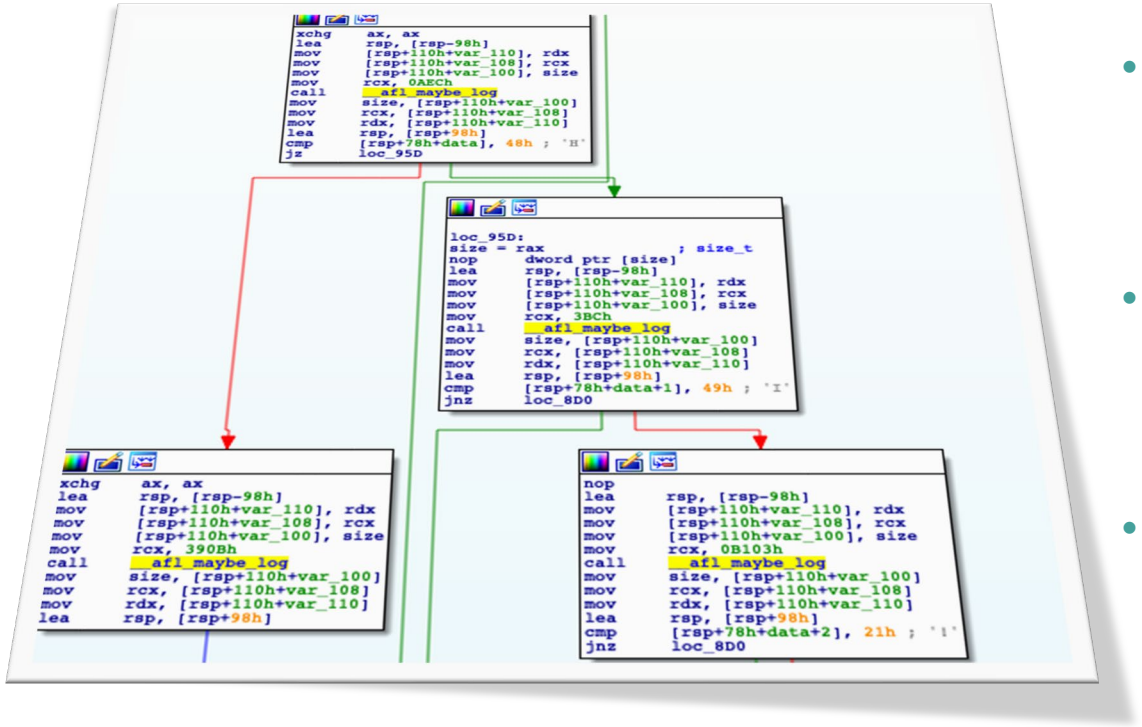
kAFL setup with some tweaks

Fuzzing Concepts

- **Feedback** – Get feedback on target execution state
- **Fitness** – Decide according to some metric (coverage) how



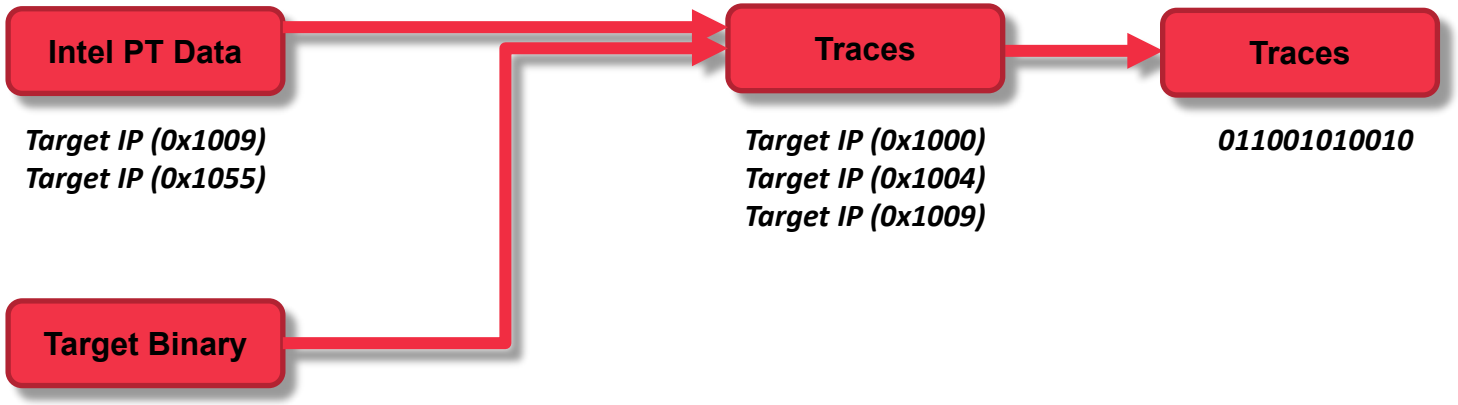
AFL Feedback



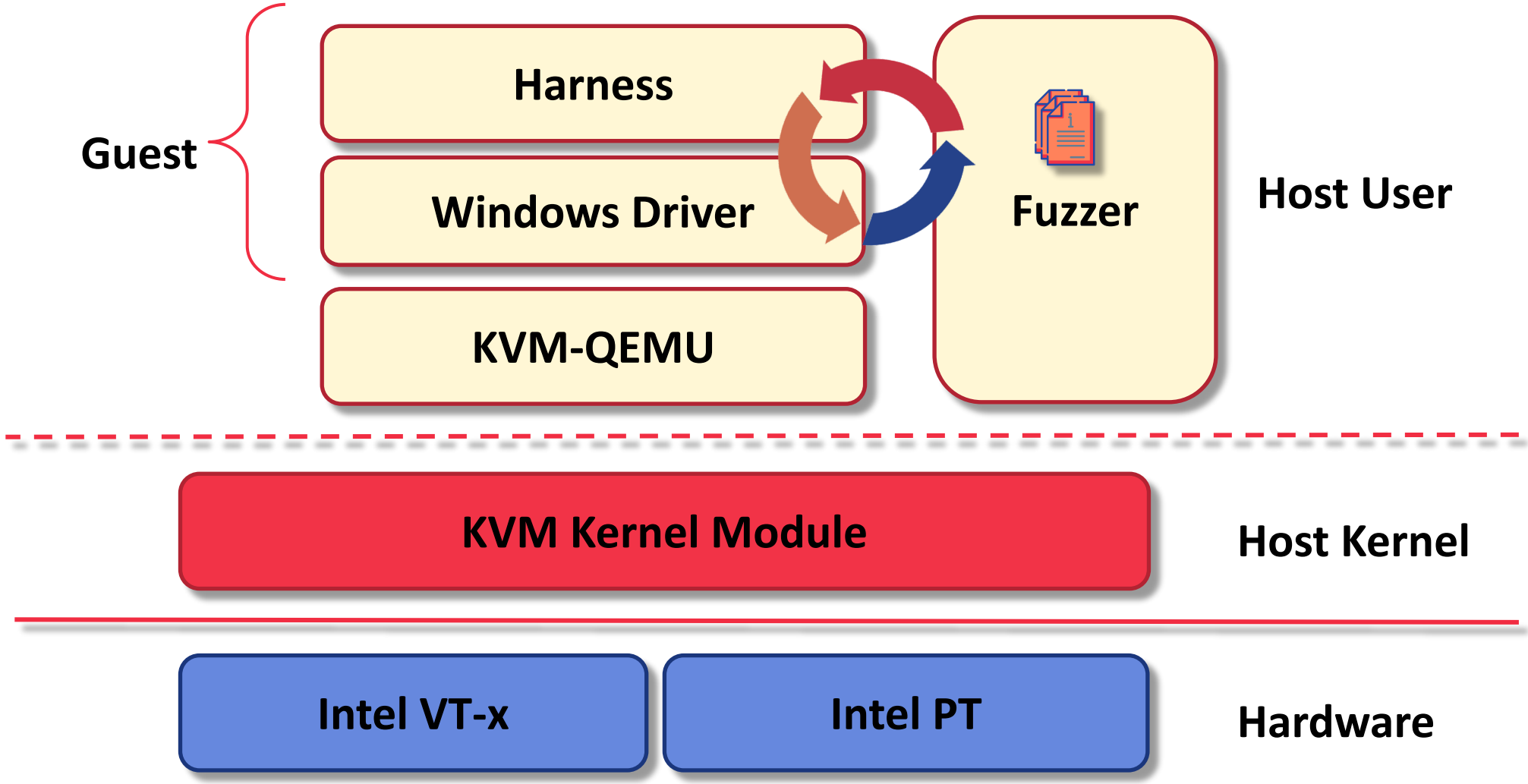
- The feedback is implemented by hooking each code block with a special report snippet
- Upon receiving a cue for entering a new block, the fuzzer updates the bitmap
- Instrumentation can be done during compile time, binary rewrite or dynamically

kAFL Feedback

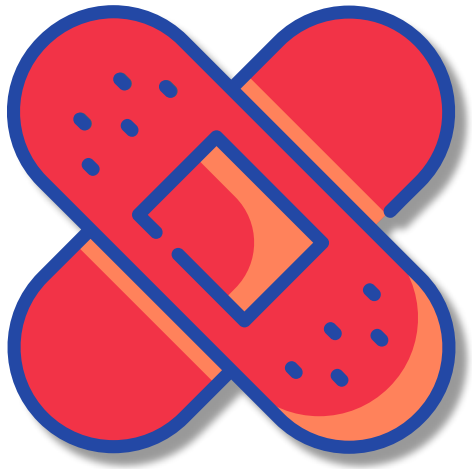
- kAFL doesn't modify the target, instead it maps Intel PT to the coverage bitmap
- Intel Processor Tracing is essentially a real-time CPU instructions tracing mechanism
- Intel PT can trace a specific driver at an address range



kAFL Virtualization Infra



Why Virtualize?



Crash Protection

Crashing the target guest won't crash the Fuzzer on the host

Closed Source

VT-x + Intel PT enbale efficient target agnostic fuzzing

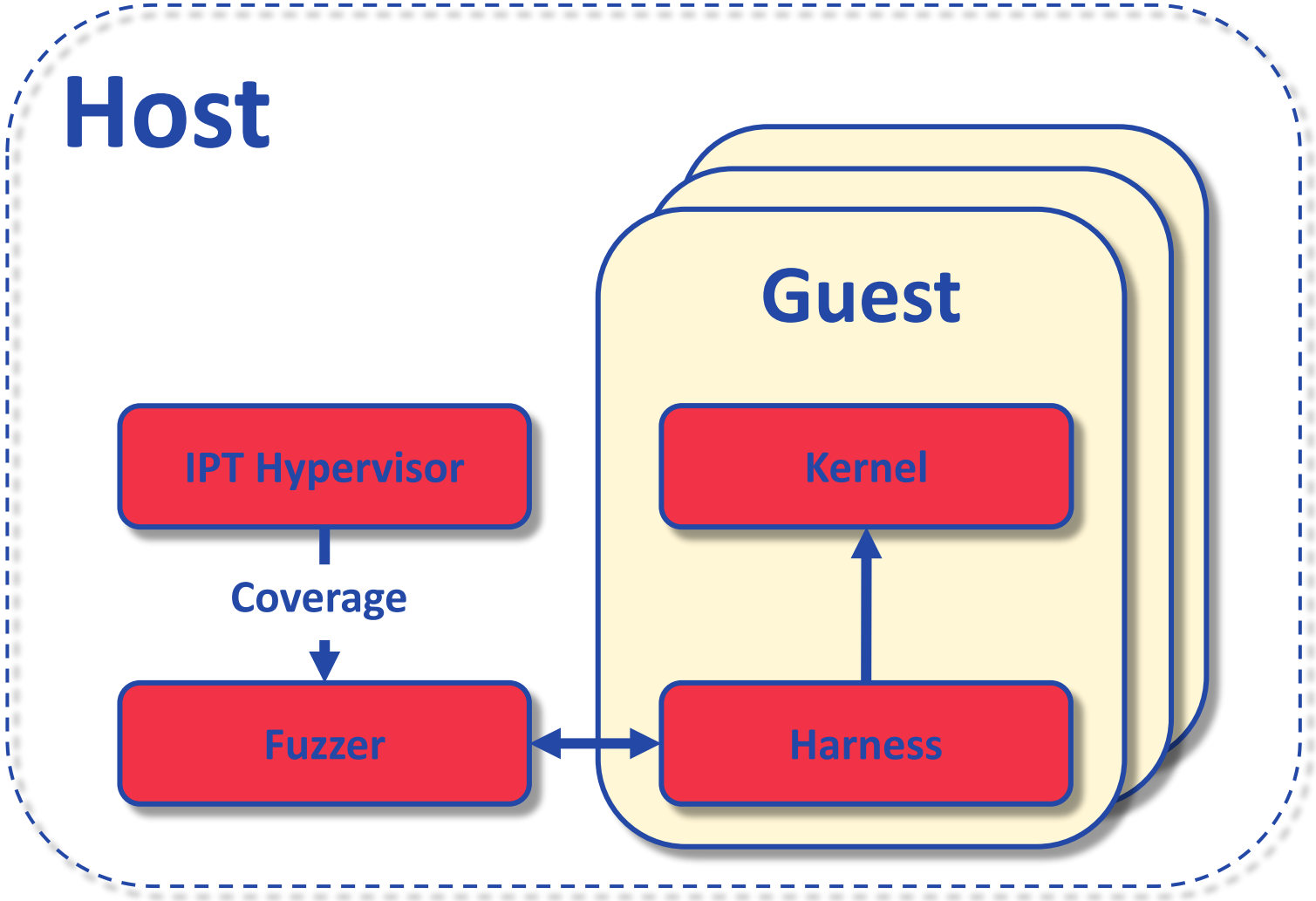
Host ←→ Guest

Host to Guest

- Send inputs
- Overwrite panic handler

Guest to Host

- Request and get next payload
- Disclose CR3 value
- Disclose panic handler address



Setup Story

- Fuzzer executes from a VT-x and KVM enabled host (Latest Ubuntu in our case)
- Fuzzing target is virtualized on the Guest (Latest Windows in our case)
- Inputs are requested by the guest via a Hypercall API
- Host passes the input via a shared memory buffer
- Guest **harness** runs inputs vs kernel
- Host collects Intel PT traces and they are converted to a coverage bitmap

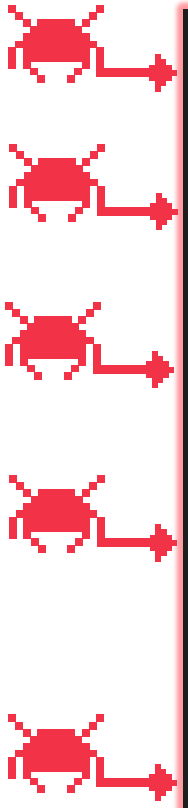
Harness: Initialization

```
#define IOCTL_KAFL_INPUT    0x00010000

int main(int argc, char** argv)
{
```

```
    kAFL_payload* payload_buffer = (kAFL_payload*)VirtualAlloc(0, PAYLOAD_SIZE, MEM_COMMIT, PAGE_READWRITE);
    memset(payload_buffer, 0xff, PAYLOAD_SIZE);


    /* open vulnerable driver */
    HANDLE kafl_vuln_handle = NULL;
    BOOL status = -1;
```

```
    kafl_vuln_handle = CreateFile((LPCSTR)"\\.\\GLOBALROOT\\Device\\XXXXXXXX",
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
        NULL
    );
```

Harness: Fuzzing Loop



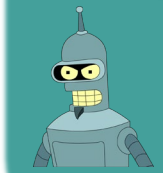
```
/* this hypercall submits the current CR3 value */
kAFL_hypercall(HYPERCALL_KAFL_SUBMIT_CR3, 0);

/* submit the guest virtual address of the payload buffer */
kAFL_hypercall(HYPERCALL_KAFL_GET_PAYLOAD, (UINT64)payload_buffer);

while(1){
        kAFL_hypercall(HYPERCALL_KAFL_NEXT_PAYLOAD, 0);
        /* request new payload (*blocking*) */
        kAFL_hypercall(HYPERCALL_KAFL_ACQUIRE, 0);

        /* validate input (simple grammar) */
        validate_input(payload_buffer->data, payload_buffer->size)

        /* kernel fuzzing */
        DeviceIoControl(kafl_vuln_handle,
            IOCTL_KAFL_INPUT,
            (LPVOID)(payload_buffer->data),
            (DWORD)payload_buffer->size,
            NULL,
            0,
            NULL,
            NULL
        );

        /* inform fuzzer about finished fuzzing iteration */
        kAFL_hypercall(HYPERCALL_KAFL_RELEASE, 0);
```

Demo

kAFL in action = Vulnerability

What's Next?

OTHER OSs



GRAMMAR

Fine tune drivers' specific grammar requirements

HYPERVISORS

Attack surfaces with more privilege levels fragmentation

Takeaways

- What Developers can do:
  - The first step in exploiting drivers is to access them
  - If you restrict that access then the attack surface is nullified
  - You can also fuzz your own drivers as a part of the QA process

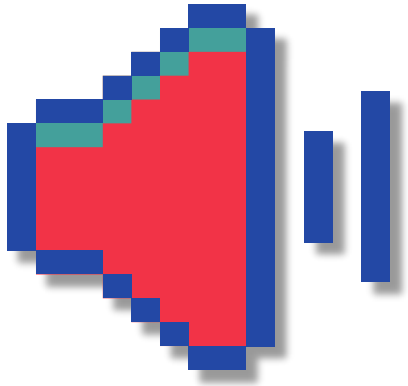
- What Attackers can do:
  - Use our automation scripts
  - Start fuzzing the kernel
  - Examine KMDF and NDIS drivers for more bugs

Takeaways

- What Defenders can do:
  - Naïve: Make sure every driver is up-to-date
  - Advanced:
    - Monitor *DeviceIoControl* requests to devices
    - Block / Alert in case the requesting executable is unsigned

Credits

- Google ☺
- Ilja van Sprundel
- James Forshaw
- kAFL
- Syzkaller
- Alex Ionescu
- HackSys Team



Thank You For Listening !

Do you have any questions?

mark.cherp@gmail.com | @OcamRazr

eranhelforc@gmail.com | @EranShimony