# CORE SECURITY

## Breaking Out of VirtualBox through 3D Acceleration

Francisco Falcon (@fdfalcon)

*REcon 2014*

# About me

- Exploit writer for Core Security.
- From Argentina.
- Interested in the usual stuff: reverse engineering, vulnerability research, exploitation…
- This is my 2nd time presenting at REcon.

# Agenda

# Agenda

- Motivations and related work
- How VirtualBox implements 3D Acceleration
- Speaking the VBoxHGCM and Chromium protocols
- Chromium rendering commands
- The vulnerabilities
- The fixes
- Exploitation
- Live Demo!
- Conclusions/Q & A

# Motivations and related work

# Motivations

- Tarjei Mandt: Oracle VirtualBox Integer Overflow Vulnerabilities (specially CVE-2011-2305: VBoxSharedOpenGL Host Service Integer Overflow Vulnerability).

http://mista.nu/blog/2011/07/19/oracle-virtualbox-integer-overflow-vulnerabilities/

# Related work

- Cloudburst: Hacking 3D (and Breaking Out of VMware) [Kostya Kortchinsky, Black Hat US 2009]
- Virtunoid: Breaking out of KVM [Nelson Elhage, Black Hat US 2011]
- A Stitch in Time Saves Nine: A case of Multiple Operating System Vulnerability [Rafal Wojtczuk, Black Hat US 2012]

# An overview of VirtualBox

# An overview of VirtualBox

*“**VirtualBox** is a general-purpose full virtualizer for x86 hardware, targeted at server, desktop and embedded use.”*

- Supported Host OS: Windows, Linux, Mac OS X, Solaris.
- Supported Guest OS : Windows, Linux, Solaris, FreeBSD, OpenBSD, Mac OS X…

CORE SECURITY

# An overview of VirtualBox

VirtualBox provides hardware-based 3D Acceleration for Windows, Linux and Solaris guests.

This allows guest machines to use the host machine's hardware to process 3D graphics based on the OpenGL or Direct3D APIs.

# VirtualBox Guest Additions

- VirtualBox implements 3D Acceleration through its Guest Additions (Guest Additions must be installed on the guest OS).
- 3D Acceleration must be manually enabled in the VM settings.

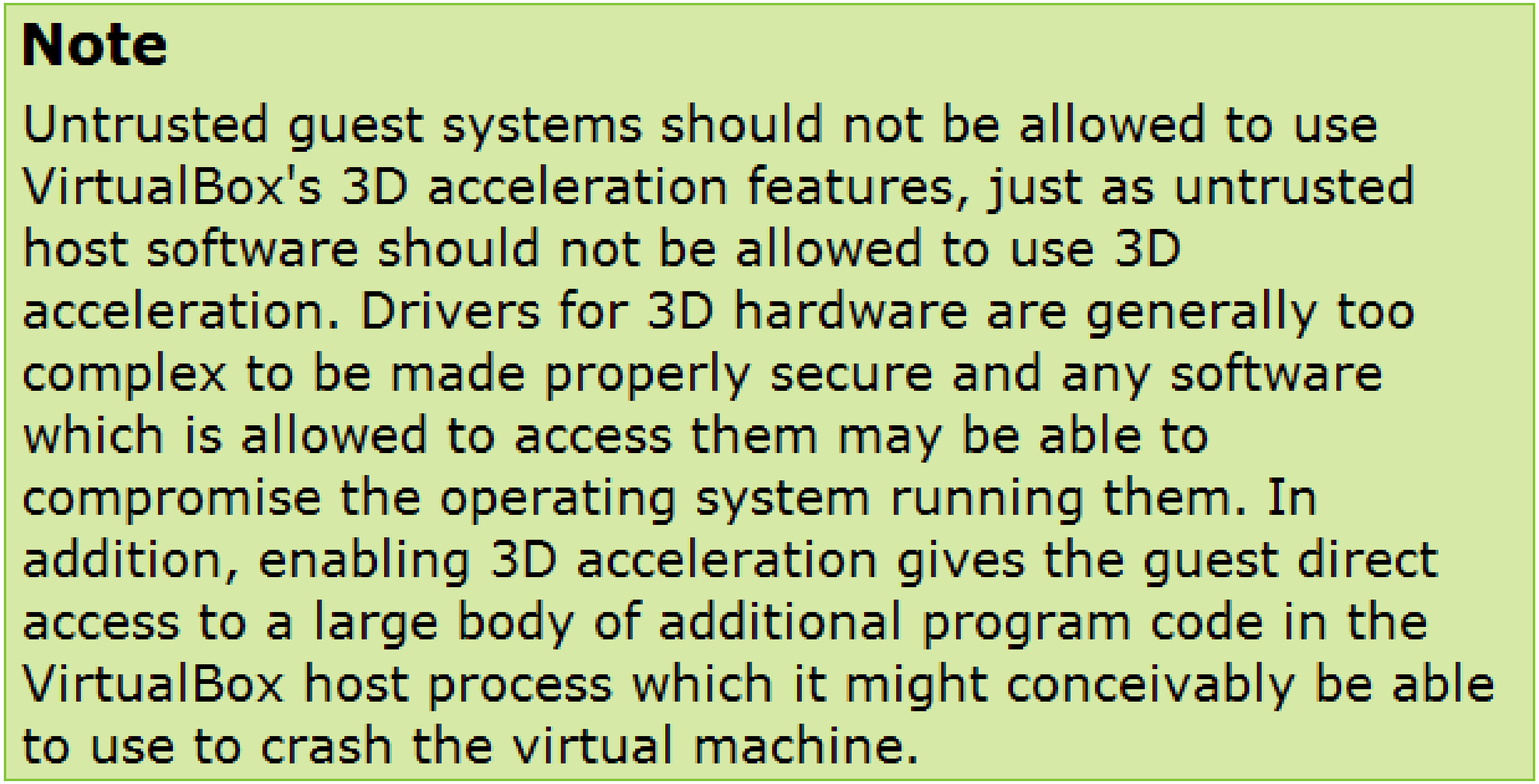
CORE SECURITY

# VirtualBox Guest Additions

- The Guest Additions install a device driver named VBoxGuest.sys in the guest machine.
- On Windows guests, this device driver can be found in the Device Manager under the “System Devices” branch.
- VBoxGuest.sys uses port-mapped I/O to communicate with the host.

# They warned you!

https://www.virtualbox.org/manual/ch04.html#guestadd-3d:



**Note**

Untrusted guest systems should not be allowed to use VirtualBox's 3D acceleration features, just as untrusted host software should not be allowed to use 3D acceleration. Drivers for 3D hardware are generally too complex to be made properly secure and any software which is allowed to access them may be able to compromise the operating system running them. In addition, enabling 3D acceleration gives the guest direct access to a large body of additional program code in the VirtualBox host process which it might conceivably be able to use to crash the virtual machine.

# The Chromium library

# Chromium

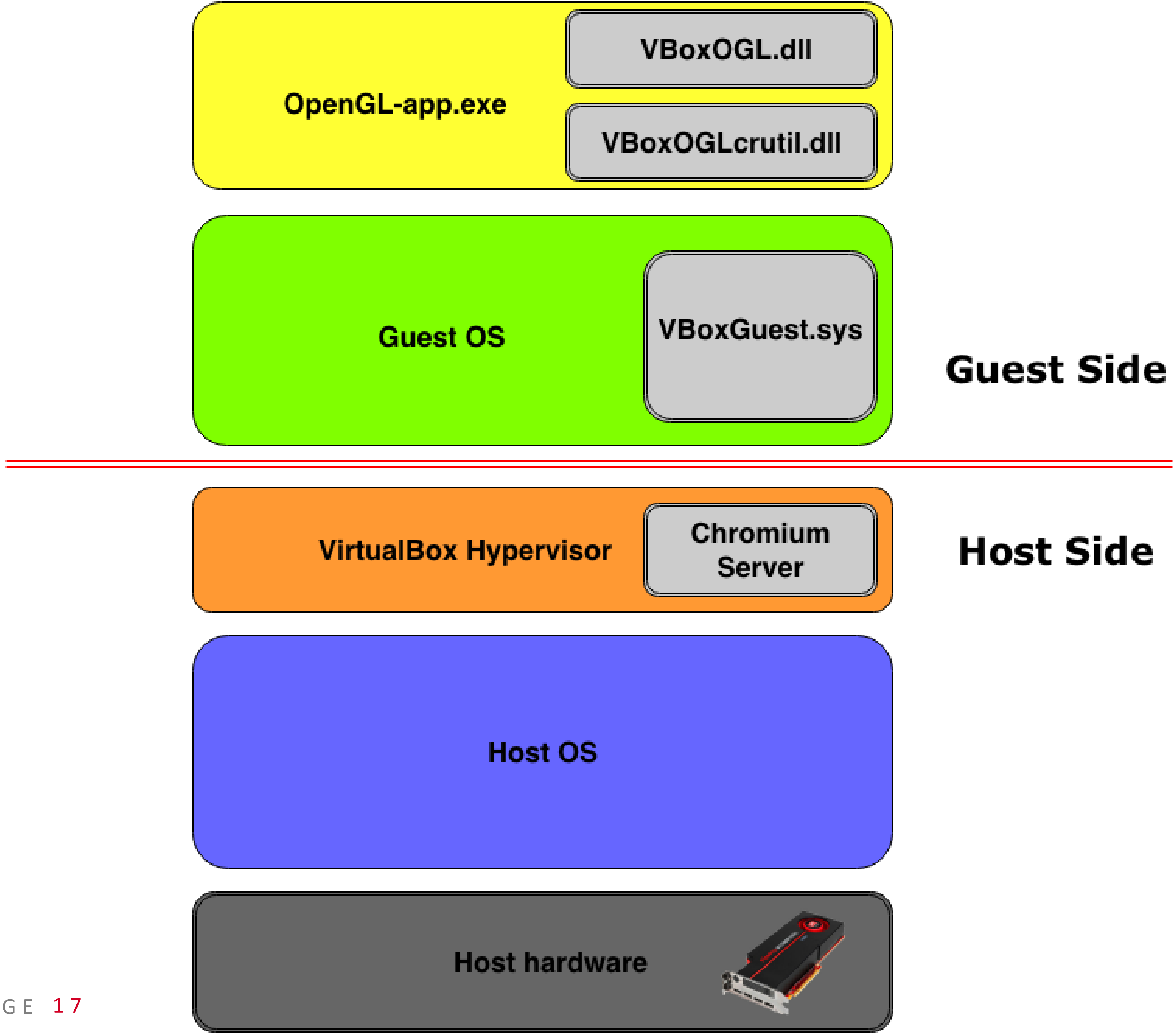
- VirtualBox 3D Acceleration is based on **Chromium**.
- Chromium is a library that allows for remote rendering of OpenGL-based 3D graphics.
- Client/server architecture.
- Not related at all with the Web browser!

# Chromium

- VirtualBox added support for a new protocol to Chromium: **VBoxHGCM** (HGCM stands for Host/Guest Communication Manager).

- This protocol allows Chromium clients running in the guest machine to communicate with the Chromium server running in the host machine.

- The **VBoxHGCM** protocol works through the VBoxGuest.sys driver.


VBoxOGL.dll
OpenGL-app.exe
VBoxOGLcrutil.dll
Guest OS
VBoxGuest.sys
Guest Side
VirtualBox Hypervisor
Chromium Server
Host Side
Host OS
Host hardware

# Speaking the VBoxHGCM protocol

# Speaking the VBoxHGCM protocol

- Step 1: obtain a handle to the VBoxGuest.sys device driver.

```
HANDLE hDevice = CreateFile("\\\\.\\VBoxGuest",
                 GENERIC_READ|GENERIC_WRITE,
                 FILE_SHARE_READ|FILE_SHARE_WRITE,
                 NULL,
                 OPEN_EXISTING,
                 FILE_ATTRIBUTE_NORMAL,
                 NULL);
```

- No privileges needed for this at all; even *guest* users can open the device!

# Speaking the VBoxHGCM protocol

- Step 2: Send a message to the VBoxGuest driver through DeviceIoControl.

```
BOOL rc = DeviceIoControl(hDevice,
                       VBOXGUEST_IOCTL_HGCM_CONNECT,
                       &info,
                       sizeof(info),
                       &info,
                       sizeof(info),
                       &cbReturned,
                       NULL);
```

# IoControl codes

- The VBoxGuest driver handles DeviceIoControl messages in the *`VBoxGuestCommonIOCtl()`* function [*`src/VBox/Additions/common/VBoxGuest/VBoxGuest.cpp`*].

- Some of the accepted IoControl codes:
  - VBOXGUEST_IOCTL_GETVMMDEVPORT
  - VBOXGUEST_IOCTL_VMMREQUEST
  - VBOXGUEST_IOCTL_SET_MOUSE_NOTIFY_CALLBACK
  - **VBOXGUEST_IOCTL_HGCM_CONNECT**
  - **VBOXGUEST_IOCTL_HGCM_CALL**
  - **VBOXGUEST_IOCTL_HGCM_DISCONNECT**
  - [...]

# Connecting to the service

Connecting to the “VBoxSharedCrOpenGL” service:

```
VBoxGuestHGCMConnectInfo info;
memset(&info, 0, sizeof(info));

info.Loc.type = VMMDevHGCMLoc_LocalHost_Existing;
strcpy(info.Loc.u.host.achName, "VBoxSharedCrOpenGL");

rc = DeviceIoControl(hDevice,
      VBOXGUEST_IOCTL_HGCM_CONNECT, &info,
      sizeof(info), &info, sizeof(info),
      &cbReturned, NULL);
```

# Speaking the Chromium protocol

# crOpenGL guest functions

- *include/VBox/HostServices/VBoxCrOpenGLSvc.h* has definitions for Input Buffer types, available Chromium guest functions and parameters count:

```
/* crOpenGL guest functions */
#define SHCRGL_GUEST_FN_WRITE       (2)
#define SHCRGL_GUEST_FN_READ        (3)
#define SHCRGL_GUEST_FN_WRITE_READ  (4)
#define SHCRGL_GUEST_FN_SET_VERSION (6)
#define SHCRGL_GUEST_FN_INJECT      (9)
#define SHCRGL_GUEST_FN_SET_PID     (12)
#define SHCRGL_GUEST_FN_WRITE_BUFFER        (13)
#define SHCRGL_GUEST_FN_WRITE_READ_BUFFERED (14)
```

Sending an HGCM Call message to the “VBoxSharedCrOpenGL” service:

```
CRVBOXHGCMSETPID parms;
memset(&parms, 0, sizeof(parms));
parms.hdr.u32ClientID = u32ClientID;
parms.hdr.u32Function = SHCRGL_GUEST_FN_SET_PID;
parms.hdr.cParms      = SHCRGL_CPARMS_SET_PID;

parms.u64PID.type     = VMMDevHGCMParmType_64bit;
parms.u64PID.u.value64 = GetCurrentProcessId();

BOOL rc = DeviceIoControl(hDevice,
          VBOXGUEST_IOCTL_HGCM_CALL, &parms,
          sizeof(parms), &parms, sizeof(parms),
          &cbReturned, NULL);
```

# HGCM_CALL handling

- When the VBoxGuest.sys driver receives a VBOXGUEST_IOCTL_HGCM_CALL message, it does the following:
- It copies our Input Buffer from the guest to the host
- It performs a call to host code
- It copies back the results (changed params and buffers) from the host to the guest

# Starting a Chromium communication

A Chromium client must start this way:

- Open the VBoxGuest.sys driver.
- Send a **VBOXGUEST_IOCTL_HGCM_CONNECT** message.
- Send a **VBOXGUEST_IOCTL_HGCM_CALL** message, calling the **SHCRGL_GUEST_FN_SET_VERSION** function.
- Send a **VBOXGUEST_IOCTL_HGCM_CALL** message, calling the **SHCRGL_GUEST_FN_SET_PID** function.

# Starting a Chromium communication

- After that, the Chromium client can start sending **VBOXGUEST_IOCTL_HGCM_CALL** messages, specifying which crOpenGL guest function it wants to invoke.

Final steps:

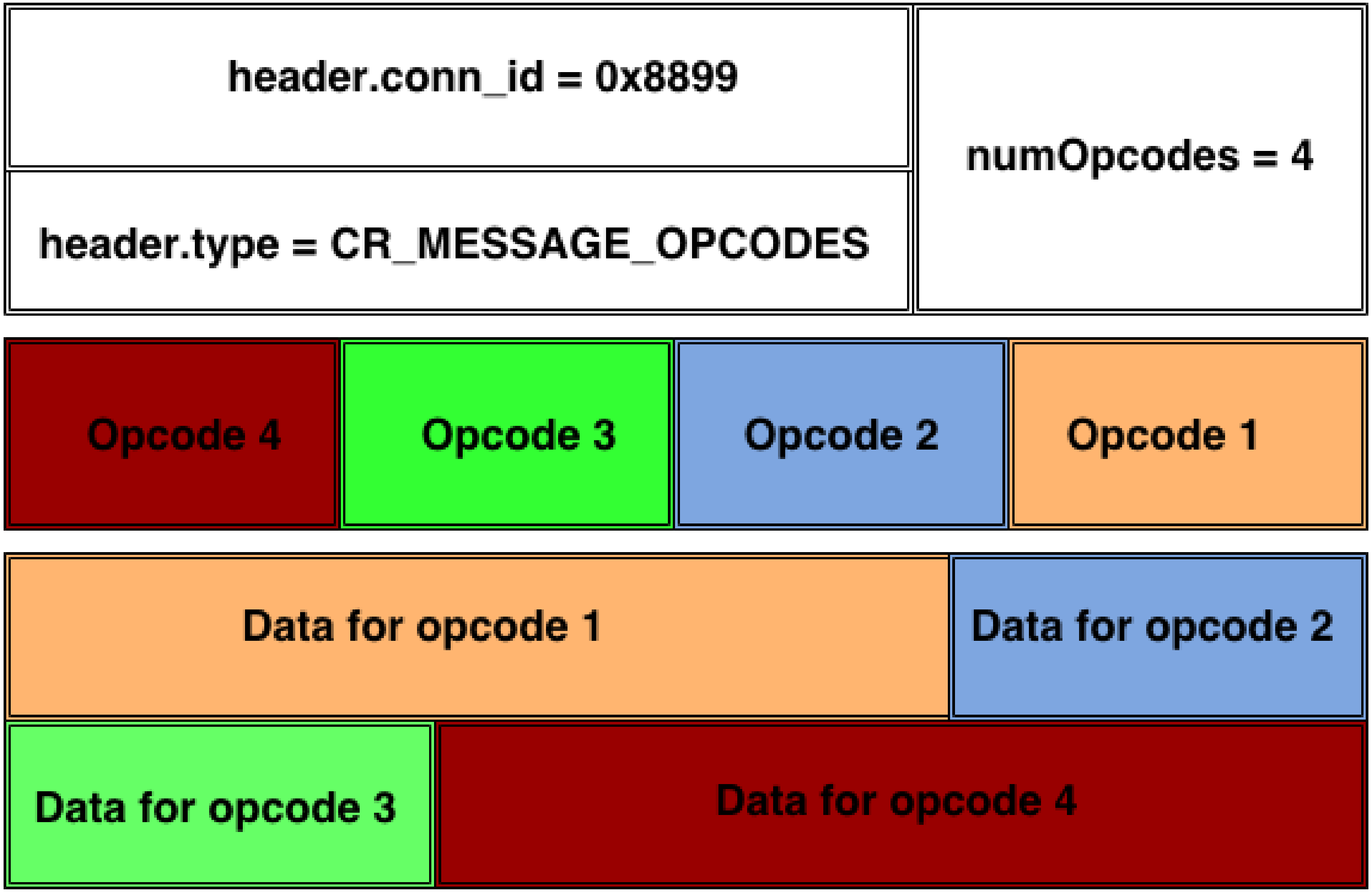
- Send a **VBOXGUEST_IOCTL_HGCM_DISCONNECT** message.
- Close the handle to the VBoxGuest.sys driver.

# Chromium Rendering Commands

# Rendering commands

- The Chromium client (VM) sends a bunch of rendering commands (opcodes + data for those opcodes).
- The Chromium server (Hypervisor) interprets those opcodes + data, and stores the result into a frame buffer.
- The content of the frame buffer is transmitted back to the client in the VM.

# CRMessageOpcodes struct



header.conn_id = 0x8899

header.type = CR_MESSAGE_OPCODES

numOpcodes = 4

Opcode 4

Opcode 3

Opcode 2

Opcode 1

Data for opcode 1

Data for opcode 2

Data for opcode 3

Data for opcode 4

# Rendering commands

- That sequence can be performed by the Chromium client in different ways:

1. **Single-step**: send the rendering commands and receive the resulting frame buffer with one single message.
2. **Two-step**: send a message with the rendering commands and let the server interpret them, then send another message requesting the resulting frame buffer.
3. **Buffered**: send the rendering commands and let the server store them in a buffer without interpreting it, then send a second message to make the server interpret the buffered commands and return the resulting frame buffer.

# Buffered Mode

**`SHCRGL_GUEST_FN_WRITE_BUFFER:`**

- Allocates a buffer that is not freed until the Chromium client sends a message to invoke the **`SHCRGL_GUEST_FN_WRITE_READ_BUFFERED`** function.

- This allows us to **allocate (and deallocate)** at will **an arbitrary number of buffers of arbitrary size** and with **arbitrary contents** in the address space of the hypervisor process that runs **on the host machine** (A.K.A. Heap Spray)

- We'll make use of this later!

# crUnpack() function

- The function *crUnpack*() handles the opcodes + data sent by a Chromium client through a **CR_MESSAGE_OPCODES** message.

- The code for this function is generated by the Python script located at *`src/VBox/HostServices/SharedOpenGL/unpacker/unpack.py`*.

- *`Unpack.py`* parses a file named APIspec.txt containing the definition of the whole OpenGL API, and **generates C code to dispatch Chromium opcodes to the corresponding OpenGL functions.**

```
void crUnpack( const void *data, const void *opcodes,
        unsigned int num_opcodes, SPUDispatchTable *table )
{
    [...]
    unpack_opcodes = (const unsigned char *)opcodes;
    cr_unpackData = (const unsigned char *)data;

    for (i = 0 ; i < num_opcodes ; i++)
    {
      /*crDebug("Unpacking opcode \%d", *unpack_opcodes);*/
      switch( *unpack_opcodes )
      {
      case CR_ALPHAFUNC_OPCODE: crUnpackAlphaFunc(); break;
      case CR_ARRAYELEMENT_OPCODE: crUnpackArrayElement(); break;
      case CR_BEGIN_OPCODE: crUnpackBegin(); break;
      [...]
```

# The Vulnerabilities

# CVE-2014-0981, CVE-2014-0982

*crVBoxServerClientWrite*() ends up calling *crNetDefaultRecv*() [net.c] before calling *crUnpack*():

```
void
crNetDefaultRecv( CRConnection *conn, CRMessage *msg,
unsigned int len ){
[...]
    switch (pRealMsg->header.type) {
        [...]
        case CR_MESSAGE_WRITEBACK:
            crNetRecvWriteback( &(pRealMsg->writeback) );
            return;
        case CR_MESSAGE_READBACK:
            crNetRecvReadback( &(pRealMsg->readback), len );
            return;
        [...]
```

# Network pointers

- Turns out that *crNetRecvWriteback*() and *crNetRecvWriteback*() are the implementation of Chromium's so-called **NETWORK POINTERS.**
- From Chromium's documentation page:

***"Network pointers are simply memory addresses that reside on another machine.[...] The networking layer will then take care of writing the payload data to the specified address."***

# Vulnerability Nº 1: CVE-2014-0981

# CVE-2014-0981

```
static void
crNetRecvReadback( CRMessageReadback *rb, unsigned int
len )
{
    int payload_len = len - sizeof( *rb );
    int *writeback;
    void *dest_ptr;
    crMemcpy( &writeback, &(rb->writeback_ptr), sizeof(
writeback ) );
    crMemcpy( &dest_ptr, &(rb->readback_ptr), sizeof(
dest_ptr ) );

    (*writeback)--;
    crMemcpy( dest_ptr, ((char *)rb) + sizeof(*rb),
payload_len );
}
```

# CVE-2014-0981

- **CVE-2014-0981**: VirtualBox crNetRecvReadback Memory Corruption Vulnerability

- The attacker from the VM fully controls both function params: `CRMessageReadback *rb, unsigned int len`

- It's a **write-what-where** memory corruption primitive **by design, within the address space of the hypervisor.**

# Vulnerability Nº 2: CVE-2014-0982

# CVE-2014-0982

```
static void
crNetRecvWriteback( CRMessageWriteback *wb )
{
    int *writeback;
    crMemcpy( &writeback, &(wb->writeback_ptr), sizeof( writeback ) );
    (*writeback)--;
}
```

# CVE-2014-0982

- **CVE-2014-0982**: VirtualBox crNetRecvWriteback Memory Corruption Vulnerability

- The attacker from the VM fully controls the function parameter: `CRMessageReadback *rb`

- Another memory corruption primitive **by design, within the address space of the hypervisor.**

# Vulnerability Nº 3: CVE-2014-0983

# CVE-2014-0983

When the opcode being processed is **CR_VERTEXATTRIB4NUBARB_OPCODE (0xEA)**, the function to be invoked is *crUnpackVertexAttrib4NubARB()*:

```
switch( *unpack_opcodes ) {
[...]
case CR_VERTEXATTRIB4NUBARB_OPCODE:
    crUnpackVertexAttrib4NubARB(); break;
[...]
```

# CVE-2014-0983

*crUnpackVertexAttrib4NubARB*() reads 5 values from the opcode data and just invokes *cr_unpackDispatch.VertexAttrib4NubARB*() with those 5 attacker-controlled values as arguments:

```
static void crUnpackVertexAttrib4NubARB(void)
{
  GLuint index = READ_DATA( 0, GLuint );
  GLubyte x = READ_DATA( 4, GLubyte );
  GLubyte y = READ_DATA( 5, GLubyte );
  GLubyte z = READ_DATA( 6, GLubyte );
  GLubyte w = READ_DATA( 7, GLubyte );
  cr_unpackDispatch.VertexAttrib4NubARB( index, x, y, z, w );
  INCR_DATA_PTR( 8 );
}
```

# CVE-2014-0983

```
void SERVER_DISPATCH_APIENTRY
crServerDispatchVertexAttrib4NubARB( GLuint index, GLubyte x,
GLubyte y, GLubyte z, GLubyte w )
{
  cr_server.head_spu->dispatch_table.VertexAttrib4NubARB( index,
x, y, z, w );
  cr_server.current.c.vertexAttrib.ub4[index] = cr_unpackData;
}
```

# CVE-2014-0983

- **CVE-2014-0983**: VirtualBox crServerDispatchVertexAttrib4NubARB Memory Corruption Vulnerability

- Allows the attacker to corrupt arbitrary memory with a pointer to attacker-controlled data.

# CVE-2014-0983

The same vulnerability affects several functions whose code is generated by the *crserverlib/server_dispatch.py* Python script:

```
CR_VERTEXATTRIB1DARB_OPCODE   [0xDE]-> crServerDispatchVertexAttrib1dARB()
CR_VERTEXATTRIB1FARB_OPCODE   [0xDF]-> crServerDispatchVertexAttrib1fARB()
CR_VERTEXATTRIB1SARB_OPCODE   [0xE0]-> crServerDispatchVertexAttrib1sARB()
CR_VERTEXATTRIB2DARB_OPCODE   [0xE1]-> crServerDispatchVertexAttrib2dARB()
CR_VERTEXATTRIB2FARB_OPCODE   [0xE2]-> crServerDispatchVertexAttrib2fARB()
CR_VERTEXATTRIB2SARB_OPCODE   [0xE3]-> crServerDispatchVertexAttrib2sARB()
CR_VERTEXATTRIB3DARB_OPCODE   [0xE4]-> crServerDispatchVertexAttrib3dARB()
CR_VERTEXATTRIB3FARB_OPCODE   [0xE5]-> crServerDispatchVertexAttrib3fARB()
CR_VERTEXATTRIB3SARB_OPCODE   [0xE6]-> crServerDispatchVertexAttrib3sARB()
CR_VERTEXATTRIB4NUBARB_OPCODE [0xEA]-> crServerDispatchVertexAttrib4NubARB()
CR_VERTEXATTRIB4DARB_OPCODE   [0xEF]-> crServerDispatchVertexAttrib4dARB()
CR_VERTEXATTRIB4FARB_OPCODE   [0xF0]-> crServerDispatchVertexAttrib4fARB()
CR_VERTEXATTRIB4SARB_OPCODE   [0xF2]-> crServerDispatchVertexAttrib4sARB()
```

# The Fixes

# The fixes

**CVE-2014-0981** and **CVE-2014-0982** (design errors): support for CR_MESSAGE_WRITEBACK and CR_MESSAGE_READBACK messages was removed from the host-side code [changeset 50437].

```
#ifdef IN_GUEST
    case CR_MESSAGE_WRITEBACK:
        crNetRecvWriteback( &(pRealMsg->writeback) );
        [...]
    case CR_MESSAGE_READBACK:
        crNetRecvReadback( &(pRealMsg->readback), len );
#endif
```

# The fixes

**CVE-2014-0983**: The *server_dispatch.py* Python script now outputs an extra conditional statement that checks if **index** is within the bounds of the array [changeset 50441].

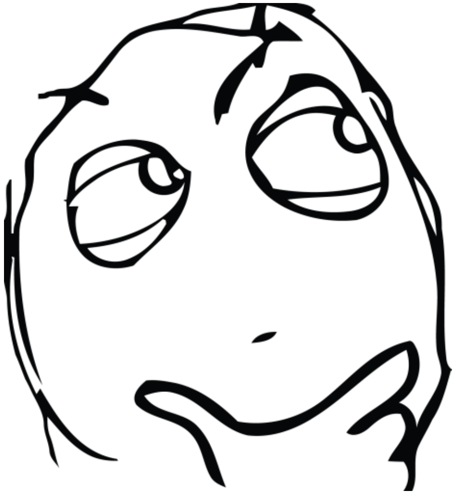
```
condition = "if (index < CR_MAX_VERTEX_ATTRIBS)"
[...]
print '\t%s' % (condition)
print '\t{'
print '\n\tcr_server.head_spu->dispatch_table.%s( %s );' %
(func_name, apiutil.MakeCallString(params) )
print "\t\tcr_server.current.c.%s.%s%s = cr_unpackData;" %
(name,type,array)
```

# Exploitation

# Exploitation

- One of the design flaws is pretty ideal for exploitation: it’s a **write-what-where primitive**.

- On a host system with ASLR, we still need to figure out where to write.



# Exploitation

*VertexAttrib4NubARB* vulnerability to the rescue!

```
void SERVER_DISPATCH_APIENTRY
crServerDispatchVertexAttrib4NubARB( GLuint index, GLubyte x,
GLubyte y, GLubyte z, GLubyte w )
{
  cr_server.head_spu->dispatch_table.VertexAttrib4NubARB( index,
x, y, z, w );
  cr_server.current.c.vertexAttrib.ub4[index] = cr_unpackData;
}
```

**cr_server** is a **global variable** (CRServer struct, as defined in *src/VBox/GuestHost/OpenGL/include/cr_server.h*) holding a lot of info on the state of the Chromium server.

# Exploitation

- We can write to a memory address **RELATIVE** to the beginning of the **cr_server.current.c.vertexAttrib.ub4** array.
- Since **cr_server** is a **global variable**, that array is located in the .data section of the *VBoxSharedCrOpenGL.dll* module.
- We can safely corrupt memory within that DLL, without having to worry about its base address being randomized due to ASLR on the host side!

# Exploitation

- So far, with opcode 0xEA we can write a pointer to data we control (the opcode data) into a memory address relative to the base address of an array that belongs to a global struct variable.

- **Anything interesting to overwrite?**

- The cr_server global variable (CRServer struct) contains (among many other fields) a field **SPU *head_spu**;

# Exploitation

- The pointer stored at **cr_server.head_spu** is dereferenced to access a **dispatch_table** field (which efectively is a table of function pointers) in the functions that handle several opcodes, for example, opcode 0x02:

```
/*Function that handles opcode CR_BEGIN_OPCODE (0x02) */

void SERVER_DISPATCH_APIENTRY crServerDispatchBegin( GLenum mode
)
{
    crStateBegin( mode );
    cr_server.head_spu->dispatch_table.Begin( mode );
}
```

# Exploitation

- That’s it! We have achieved code execution on the host!
- Step 1: send opcode 0xEA; opcode data must contain a crafted index argument to overwrite **cr_server.head_spu**; now you control the function pointers table.
- Step 2: send opcode 0x02 to hijack the execution flow of the hypervisor.

# Need to bypass ASLR?

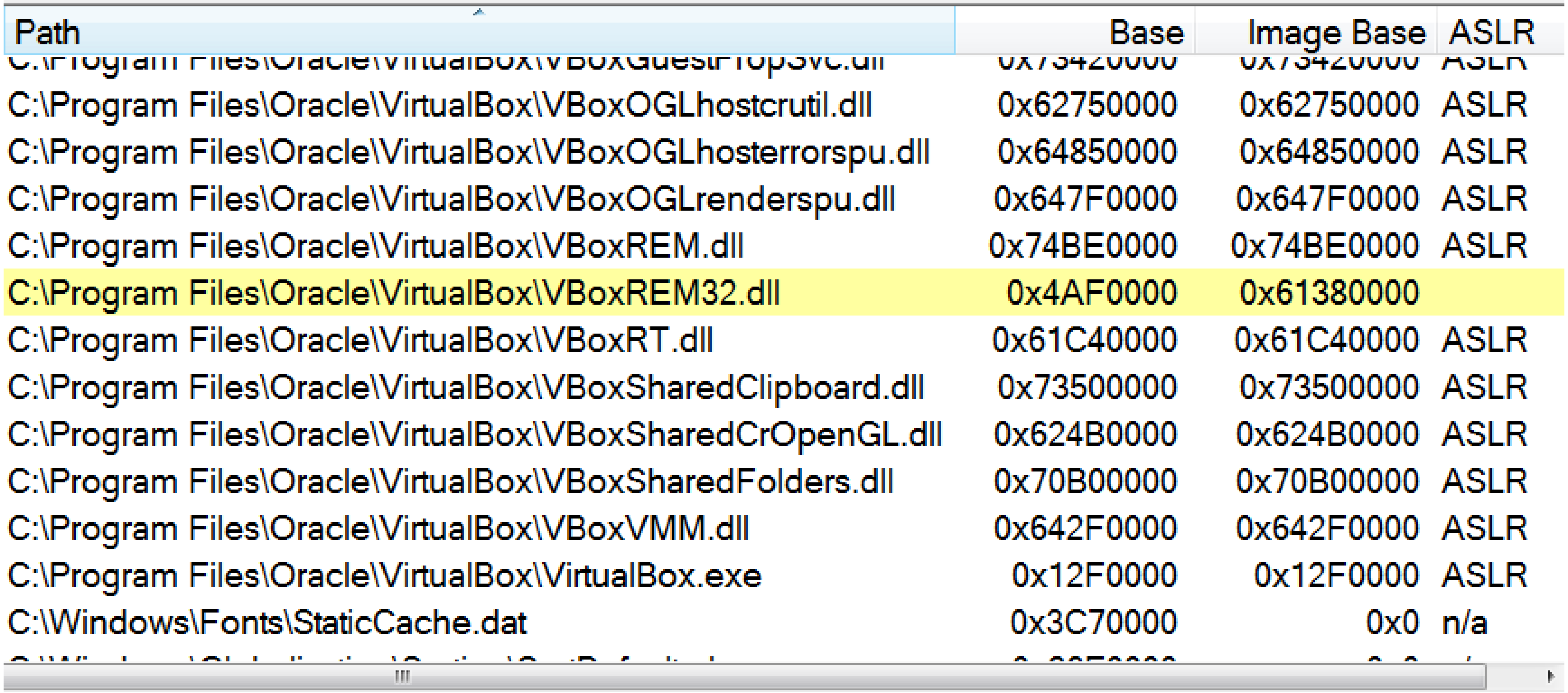
- VirtualBox versions 4.2.x ship VBoxREM32.dll compiled without ASLR support -> base address 0x61380000

- VirtualBox versions 4.3.x ship VBoxREM64.dll compiled without ASLR support -> base address 0x6D380000

# Exploitation

CORE SECURITY

# Can’t rely on non-ASLR modules



| Path | Base | Image Base | ASLR |
|---|---|---|---|
| C:\Program Files\Oracle\VirtualBox\VBoxGuestPropSvc.dll | 0x73420000 | 0x73420000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxOGLhostcrutil.dll | 0x62750000 | 0x62750000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxOGLhosterrorspu.dll | 0x64850000 | 0x64850000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxOGLrenderspu.dll | 0x647F0000 | 0x647F0000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxREM.dll | 0x74BE0000 | 0x74BE0000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxREM32.dll | 0x4AF0000 | 0x61380000 | |
| C:\Program Files\Oracle\VirtualBox\VBoxRT.dll | 0x61C40000 | 0x61C40000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxSharedClipboard.dll | 0x73500000 | 0x73500000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxSharedCrOpenGL.dll | 0x624B0000 | 0x624B0000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxSharedFolders.dll | 0x70B00000 | 0x70B00000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VBoxVMM.dll | 0x642F0000 | 0x642F0000 | ASLR |
| C:\Program Files\Oracle\VirtualBox\VirtualBox.exe | 0x12F0000 | 0x12F0000 | ASLR |
| C:\Windows\Fonts\StaticCache.dat | 0x3C70000 | 0x0 | n/a |

# Exploitation with full ASLR bypass

***“Infoleaks are made, not found”***

- Halvar Flake -

# Exploitation with full ASLR bypass

- **cr_server** global variable holds all the information about the state of the Chromium server.

- It holds an array of currently connected clients:

```
typedef struct {
[...]
int numClients;
CRClient *clients[CR_MAX_CLIENTS];
[...]
} CRServer;
```

# Exploitation with full ASLR bypass

The **CRClient** struct (*cr_server.h*) contains this interesting field:

```
typedef struct _crclient {
int spu_id;
CRConnection *conn;   /**< network connection from the client */
[...]
} CRClient;
```

# Exploitation with full ASLR bypass

The **CRConnection** struct (*cr_net.h*) contains some juicy data:

```
struct CRConnection {
[...]
 uint8_t  *pHostBuffer;
 uint32_t cbHostBuffer;
[...]
};
```

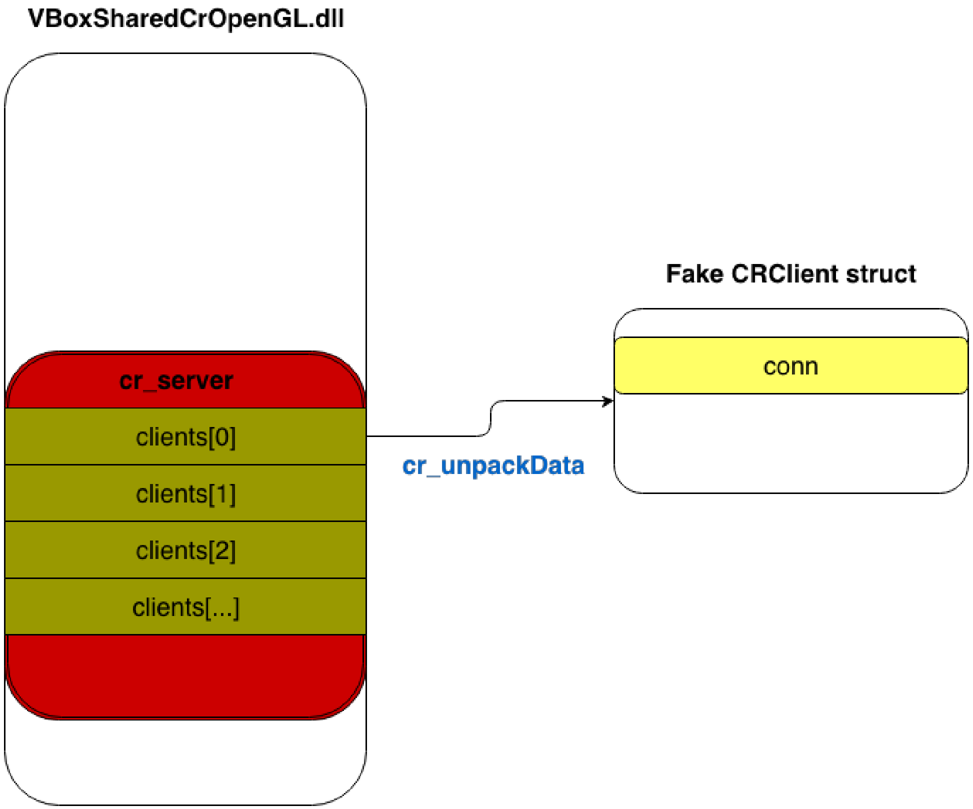
**pHostBuffer** and **cbHostBuffer** define address and size of the resulting frame buffer that will be copied to the guest.

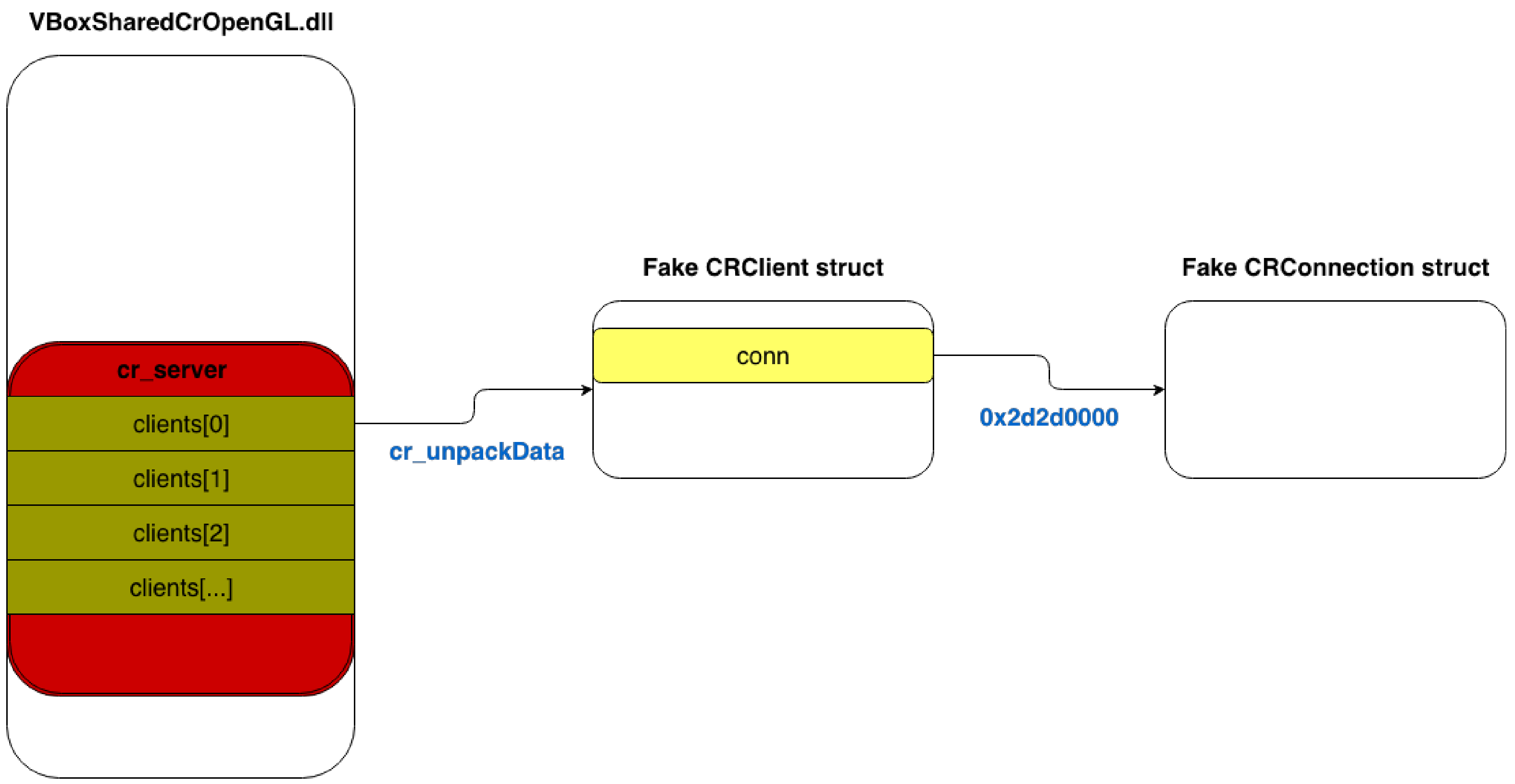
# Exploitation with full ASLR bypass

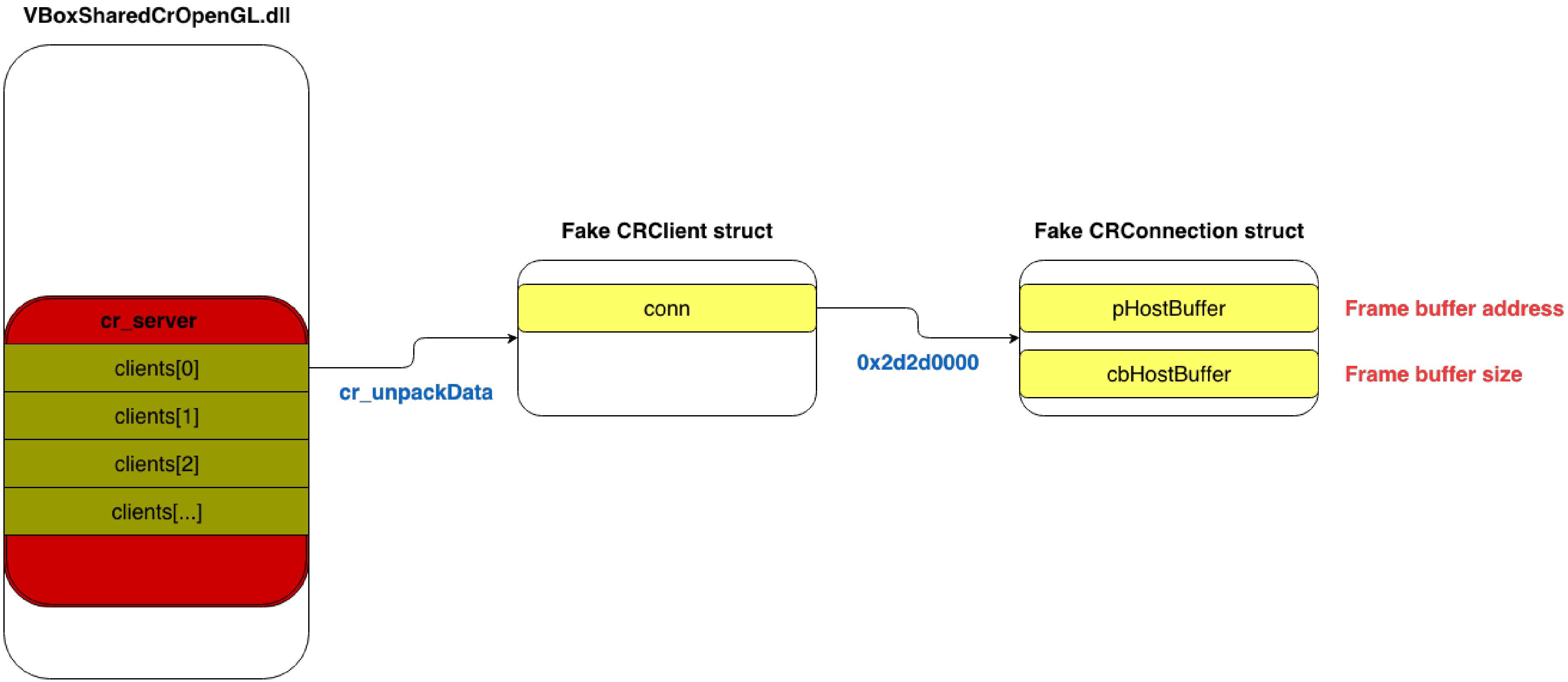
- So, we could overwrite **cr_server.clients[0]** with the pointer to our opcode data (cr_unpackData).
- cr_unpackData will be a fake **CRClient** struct.

- The 2nd DWORD of our fake **CRClient** struct will be interpreted as the **CRConnection *conn** field.
- We could make the **CRConnection *conn** field point to data controlled by us (a fake **CRConnection** struct).

- Finally, our fake **CRConnection** struct will contain crafted **pHostBuffer** and **cbHostBuffer** fields.


VBoxSharedCrOpenGL.dll
cr_server
clients[0]
clients[1]
clients[2]
clients[...]
cr_unpackData
Fake CRClient struct
conn


VBoxSharedCrOpenGL.dll
cr_server
clients[0]
clients[1]
clients[2]
clients[...]
cr_unpackData
Fake CRClient struct
conn
0x2d2d0000
Fake CRConnection struct


VBoxSharedCrOpenGL.dll
cr_server
clients[0]
clients[1]
clients[2]
clients[...]
cr_unpackData
Fake CRClient struct
conn
0x2d2d0000
Fake CRConnection struct
pHostBuffer
cbHostBuffer
Frame buffer address
Frame buffer size

# Exploitation with full ASLR bypass

- We still have a problem to solve! (caused by ASLR)
- We need the **CRConnection *conn** field point to data controlled by us (a fake **CRConnection** struct).
- How can we place a fake **CRConnection** struct at a known address (within the address space of the hypervisor on the Host side)?

# Exploitation with full ASLR bypass

- Heap Spray to the rescue!
- Remember that rendering commands sequence can be performed in Single-Step, Two-Step, or **Buffered Mode**.
- In Buffered Mode, we can allocate an arbitrary number of buffers with arbitrary size and content within the address space of the hypervisor with the SHCRGL_GUEST_FN_WRITE_BUFFER function.

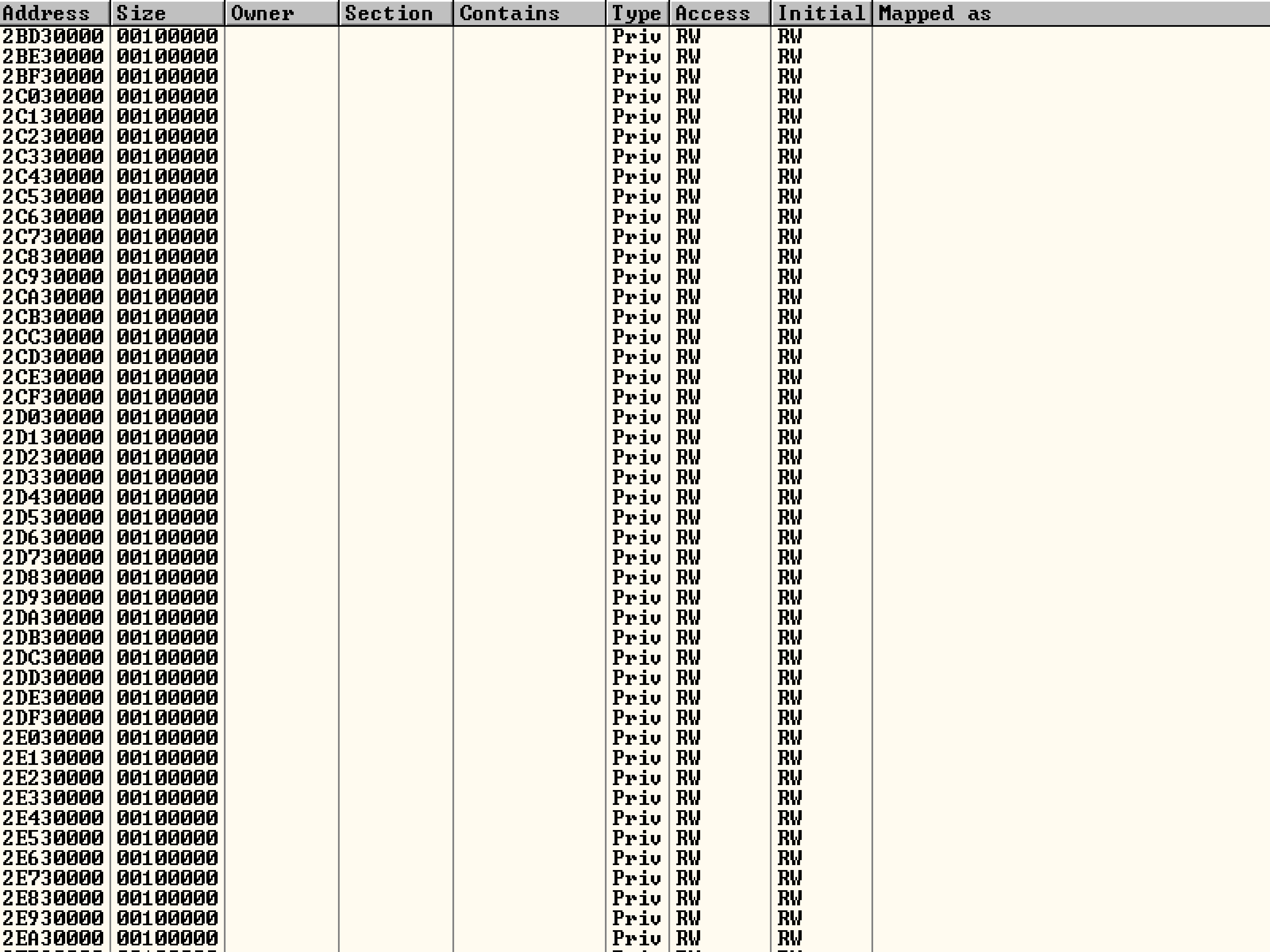
# Exploitation with full ASLR bypass

- Those buffers are not freed until we invoke  the SHCRGL_GUEST_FN_WRITE_READ_BUFFERED function.

- With this command we control which buffer we want to free, specifying it **BY ID!**

- That gives us some nice heap manipulation power on the hypervisor address space.

- This technique may aid in other VM escape exploits, given that 3D acceleration is enabled.



| Address | Size | Owner | Section | Contains | Type | Access | Initial | Mapped as |
|---|---|---|---|---|---|---|---|---|
| 2BD30000 | 00100000 | | | | Priv | RW | RW | |
| 2BE30000 | 00100000 | | | | Priv | RW | RW | |
| 2BF30000 | 00100000 | | | | Priv | RW | RW | |
| 2C030000 | 00100000 | | | | Priv | RW | RW | |
| 2C130000 | 00100000 | | | | Priv | RW | RW | |
| 2C230000 | 00100000 | | | | Priv | RW | RW | |
| 2C330000 | 00100000 | | | | Priv | RW | RW | |
| 2C430000 | 00100000 | | | | Priv | RW | RW | |
| 2C530000 | 00100000 | | | | Priv | RW | RW | |
| 2C630000 | 00100000 | | | | Priv | RW | RW | |
| 2C730000 | 00100000 | | | | Priv | RW | RW | |
| 2C830000 | 00100000 | | | | Priv | RW | RW | |
| 2C930000 | 00100000 | | | | Priv | RW | RW | |
| 2CA30000 | 00100000 | | | | Priv | RW | RW | |
| 2CB30000 | 00100000 | | | | Priv | RW | RW | |
| 2CC30000 | 00100000 | | | | Priv | RW | RW | |
| 2CD30000 | 00100000 | | | | Priv | RW | RW | |
| 2CE30000 | 00100000 | | | | Priv | RW | RW | |
| 2CF30000 | 00100000 | | | | Priv | RW | RW | |
| 2D030000 | 00100000 | | | | Priv | RW | RW | |
| 2D130000 | 00100000 | | | | Priv | RW | RW | |
| 2D230000 | 00100000 | | | | Priv | RW | RW | |
| 2D330000 | 00100000 | | | | Priv | RW | RW | |
| 2D430000 | 00100000 | | | | Priv | RW | RW | |
| 2D530000 | 00100000 | | | | Priv | RW | RW | |
| 2D630000 | 00100000 | | | | Priv | RW | RW | |
| 2D730000 | 00100000 | | | | Priv | RW | RW | |
| 2D830000 | 00100000 | | | | Priv | RW | RW | |
| 2D930000 | 00100000 | | | | Priv | RW | RW | |
| 2DA30000 | 00100000 | | | | Priv | RW | RW | |
| 2DB30000 | 00100000 | | | | Priv | RW | RW | |
| 2DC30000 | 00100000 | | | | Priv | RW | RW | |
| 2DD30000 | 00100000 | | | | Priv | RW | RW | |
| 2DE30000 | 00100000 | | | | Priv | RW | RW | |
| 2DF30000 | 00100000 | | | | Priv | RW | RW | |
| 2E030000 | 00100000 | | | | Priv | RW | RW | |
| 2E130000 | 00100000 | | | | Priv | RW | RW | |
| 2E230000 | 00100000 | | | | Priv | RW | RW | |
| 2E330000 | 00100000 | | | | Priv | RW | RW | |
| 2E430000 | 00100000 | | | | Priv | RW | RW | |
| 2E530000 | 00100000 | | | | Priv | RW | RW | |
| 2E630000 | 00100000 | | | | Priv | RW | RW | |
| 2E730000 | 00100000 | | | | Priv | RW | RW | |
| 2E830000 | 00100000 | | | | Priv | RW | RW | |
| 2E930000 | 00100000 | | | | Priv | RW | RW | |
| 2EA30000 | 00100000 | | | | Priv | RW | RW | |

# Exploitation with full ASLR bypass

- We need to put arbitrary data (fake **CRConnection** struct) at an arbitrary address.

- First option: allocate multiple 1 Mb buffers, repeating a 64 Kb pattern (M. Dowd and A. Sotirov, 2008).

- Second (and lazy) option: allocate multiple 1 Mb buffers, write the desired data to an arbitrary address by using the first vulnerability.

# Infoleak

- The memory region specified by address **pHostBuffer** and size **cbHostBuffer** is copied to the Guest in function *crVBoxServerInternalClientRead* [*server_main.c*]:

```
*pcbBuffer = pClient->conn->cbHostBuffer;
if (*pcbBuffer)    {
    CRASSERT(pClient->conn->pHostBuffer);
    crMemcpy(pBuffer, pClient->conn->pHostBuffer, *pcbBuffer);
    pClient->conn->cbHostBuffer = 0;
}
```

# Infoleak

- So by controlling the **pHostBuffer** and **cbHostBuffer** fields of our fake **CRConnection** struct , we control the address and size of the data to be copied back to the guest.

- This way **we have created an information leak that will allow us to read arbitrary hypervisor memory from the Guest!**

- **The problem: where to read from?** (Remember ASLR)

# Leaking what?

- One of the many fields of the global struct **cr_server** is **CRHashTable *barriers**.

- We can overwrite **cr_server.barriers** with the pointer to our opcode data (cr_unpackData). cr_unpackData will be a fake **CRHashTable** struct (*hash.c*).

- **cr_server.barriers** is used when processing opcode 0xF7 (extended opcode) with subopcode 0x08.

# Leaking what?

```
void SERVER_DISPATCH_APIENTRY
crServerDispatchBarrierExecCR(GLuint name ){
[...]
barrier =
(CRServerBarrier*)crHashtableSearch(cr_server.barriers, name);
[...]
barrier->waiting[barrier->num_waiting++] = cr_server.run_queue;
[...]
}
```

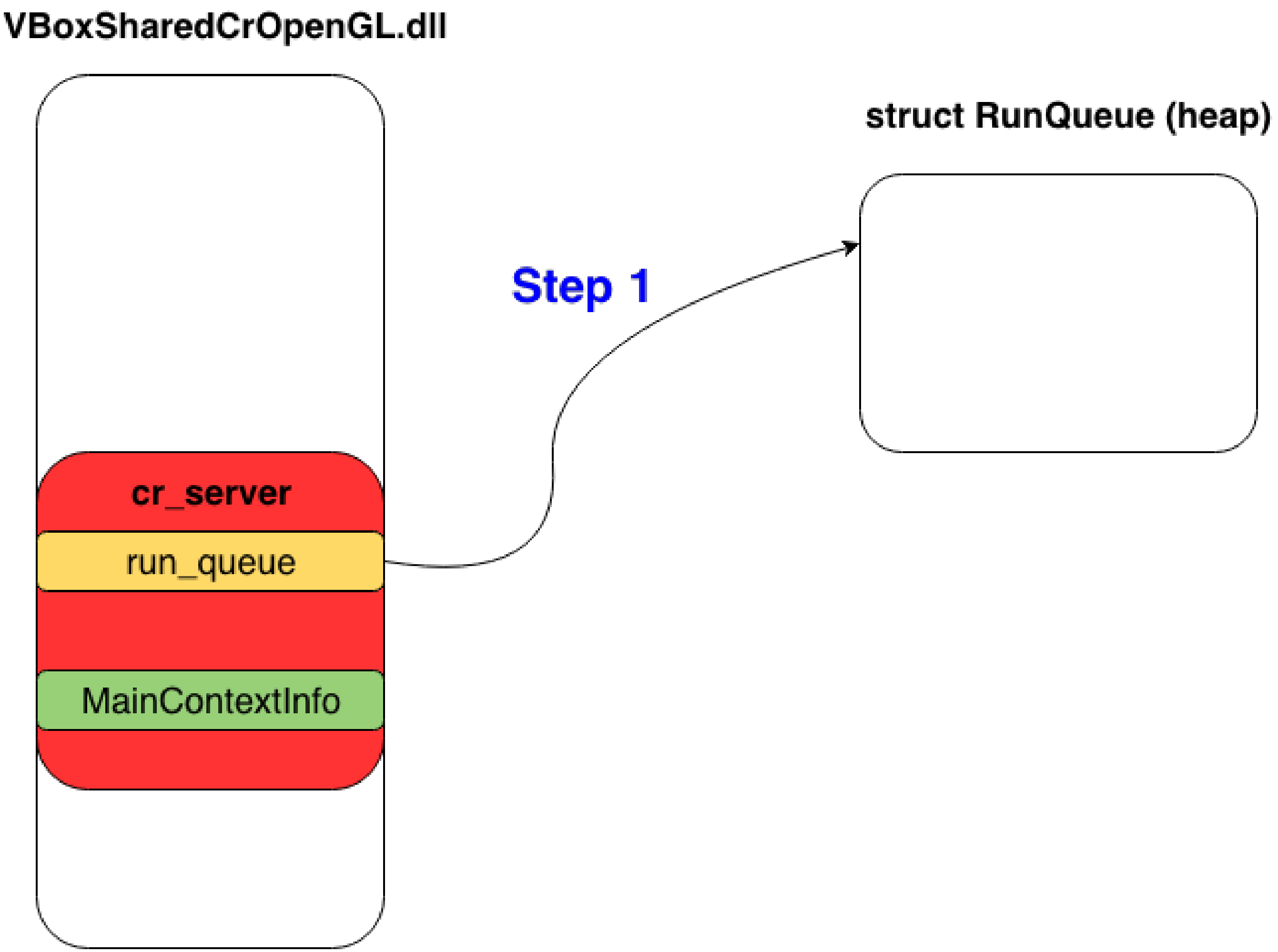
- We control both arguments in the call to *crHashtableSearch [hash.c].*
- *crHashtableSearch*() returns an arbitrary pointer controlled by us.

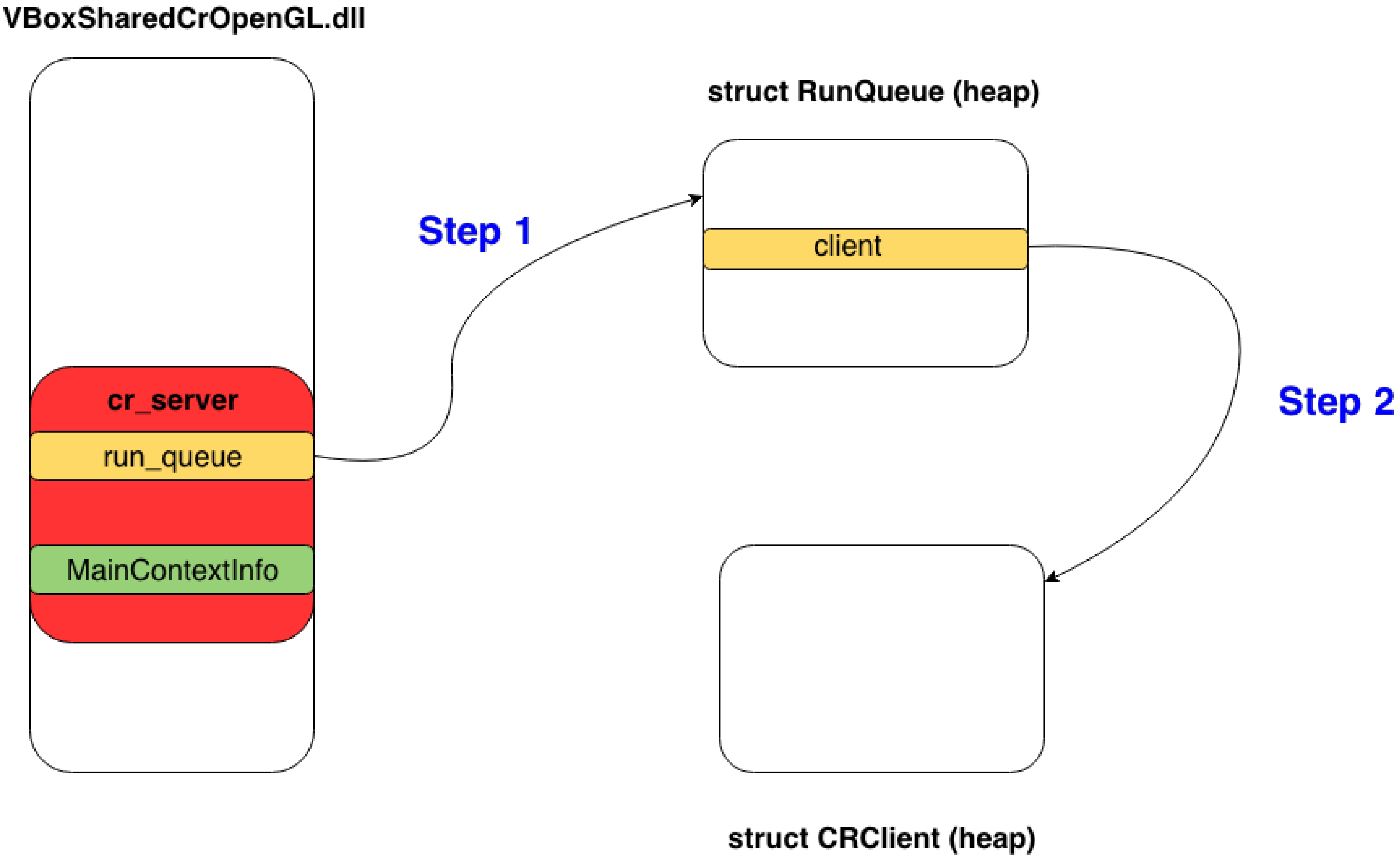
# Leaking what?

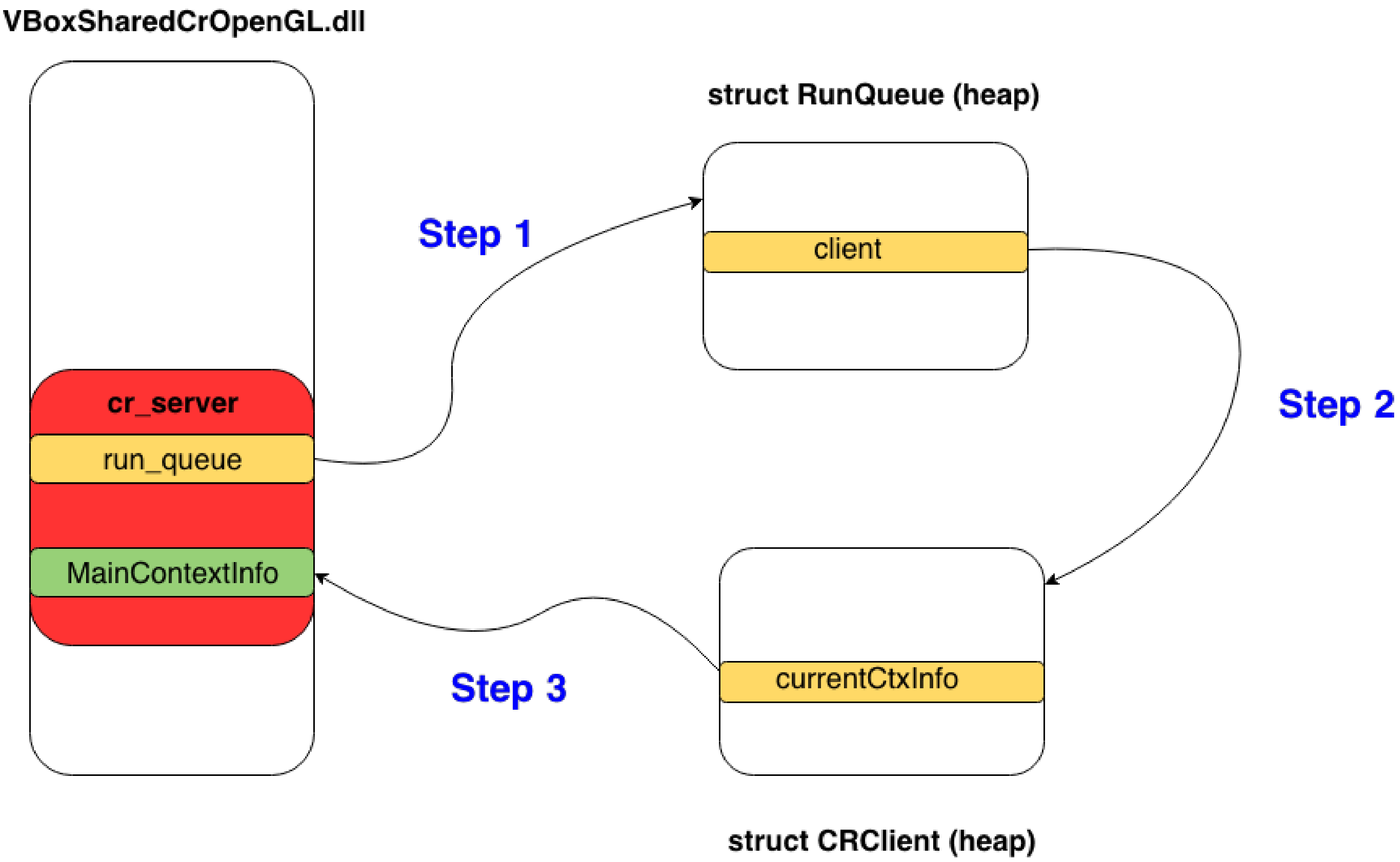
- Since we fully control the value of the **barrier** pointer, this line means that we can also fully control the address where **cr_server.run_queue** will be stored by crafting a fake **CRServerBarrier** at a known address:

```
barrier->waiting[barrier->num_waiting++] = cr_server.run_queue;
```

- So far we have managed to store the pointer **cr_server.run_queue** (type **RunQueue ***)in a known address, so **now we can leak it to the Guest side**.


VBoxSharedCrOpenGL.dll
struct RunQueue (heap)
Step 1
cr_server
run_queue
MainContextInfo


VBoxSharedCrOpenGL.dll
cr_server
run_queue
MainContextInfo
Step 1
struct RunQueue (heap)
client
Step 2
struct CRClient (heap)


VBoxSharedCrOpenGL.dll
cr_server
run_queue
MainContextInfo
struct RunQueue (heap)
client
Step 1
Step 2
Step 3
currentCtxInfo
struct CRClient (heap)

# Bye bye ASLR!

- Now we can calculate the base address of VBoxSharedCrOpenGL.dll. Bye bye ASLR on the Host side!
- Go build your ROP chain and break out of the hypervisor!

# Live Demo!

# Future Work and Conclusions

# Future Work

- Try to write an exploit that works on a 64-bit Host OS.
- Try to write an exploit that works on other Guest/Host Operating Systems.
- Work on process continuation (keep the hypervisor running after exploitation).

# Conclusions

- VirtualBox added a library to the hypervisor without even thinking about its security.
- Having a bytecode interpreter in the hypervisor running untrusted bytecode coming from a VM is not a good idea.
- Sometimes, even legit OpenGL apps crash the VM.
- Go grab the Proof-of-Concept! http://www.coresecurity.com/grid/advisories

# Contact info

@fdfalcon

[ffalcon@coresecurity.com](mailto:ffalcon@coresecurity.com)

# Questions?

Slowly please! Not an Engrish specialista ☺

Thank you!