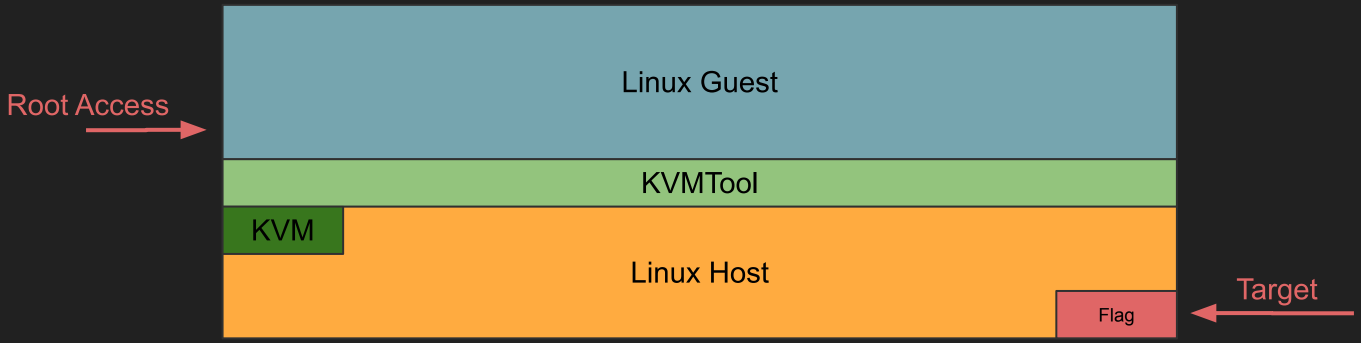
KalmarVM

CTF Challenge Presentation

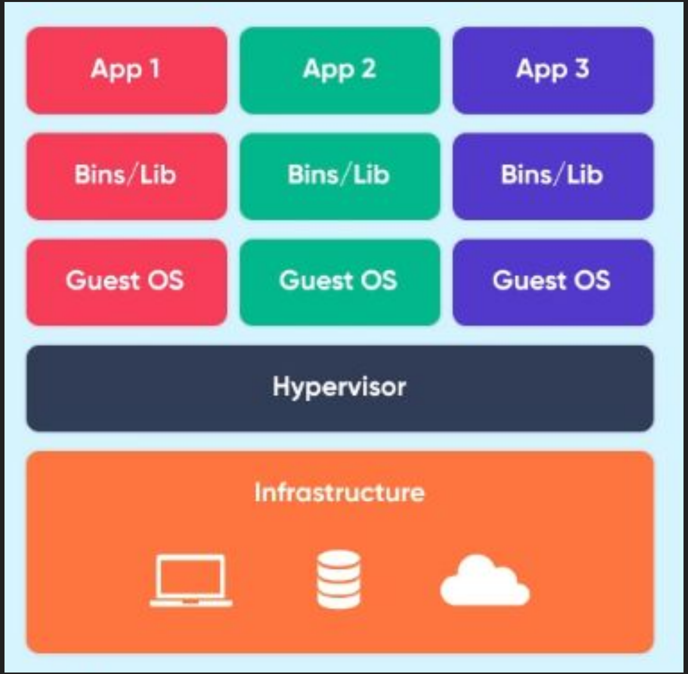
Challenge Summary

- Linux host (with flag) running guest virtual machine
- Given root access to guest
- Goal: escape guest VM and get flag from host



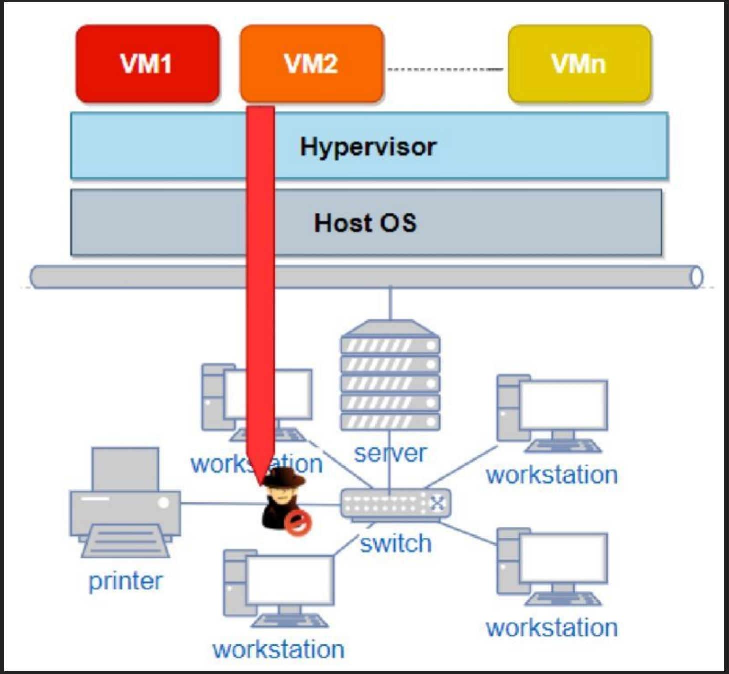
Virtualization and VMs

- Run “guest” OS from within host OS
  - Developers, security products want sandbox environment
  - Cloud providers want isolation (e.g. multiple customers on the same server)
- Simplest option: fully emulate hardware
  - Write a normal program that reads every x86 instruction and does exactly what a real CPU would do
  - Possible, but extremely slow
- Better: Safely lend the guest “virtual” access to real hardware
  - “Hypervisor” kernel-level application manages hardware, multiplexes different guests across hardware
  - Guest able to execute kernel-mode instruction on the CPU
  - Common hypervisors: KVM, Hyper-V, VMWare



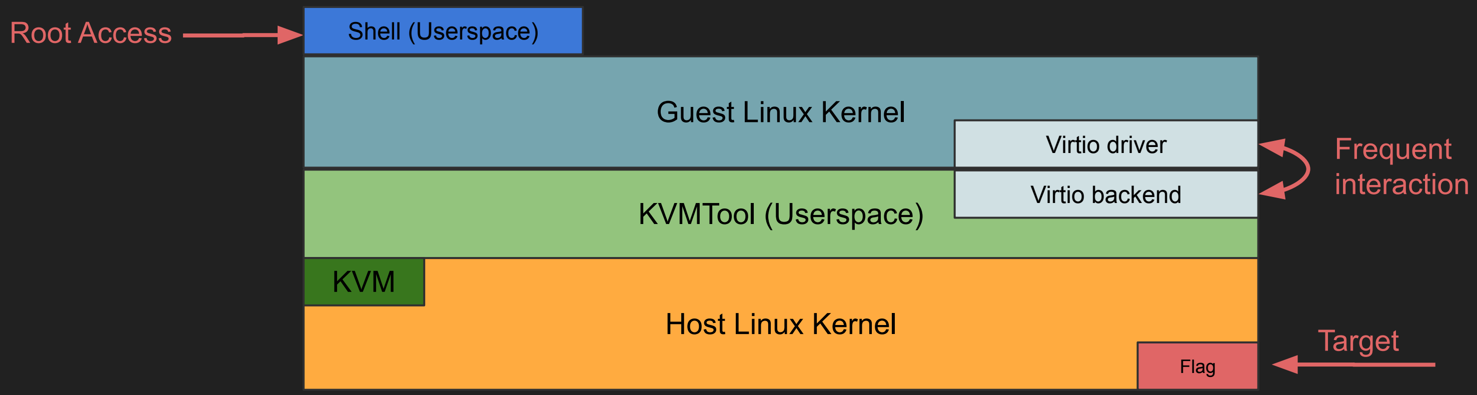
VM Escapes

- Guest VM to host (or guest VM to guest VM) is strong security boundary
    - In cloud environment, host may be managing many guests for different customers
    - Guests shouldn't see or affect each other or the host
- VM escapes exploit vulnerabilities in the hypervisor
    - Most extreme allow guests to take control of host and other guests
- Vulnerabilities can be in hypervisor itself, or device emulator/manager



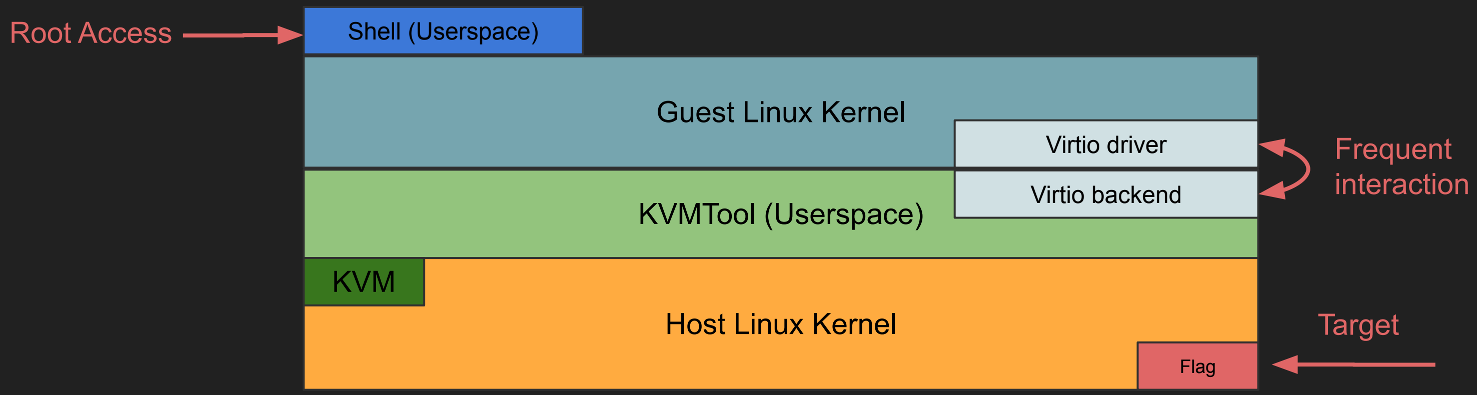
KalmarVM Structure

- Hypervisor is KVM (kernelspace) managed by KVMTool (userspace)
  - KVM only manages core hardware (e.g. CPU and memory, but not I/O device emulation)
- KVMTool is real software, but closer to hobby project (unlike KVM)
  - Minimal alternative to QEMU, provides, everything else needed to boot images
  - Main interaction with guest is via backend device emulation, tends to be complex
- Approach: Look for bugs in device emulation code



Virtio Communication with Virtqueues

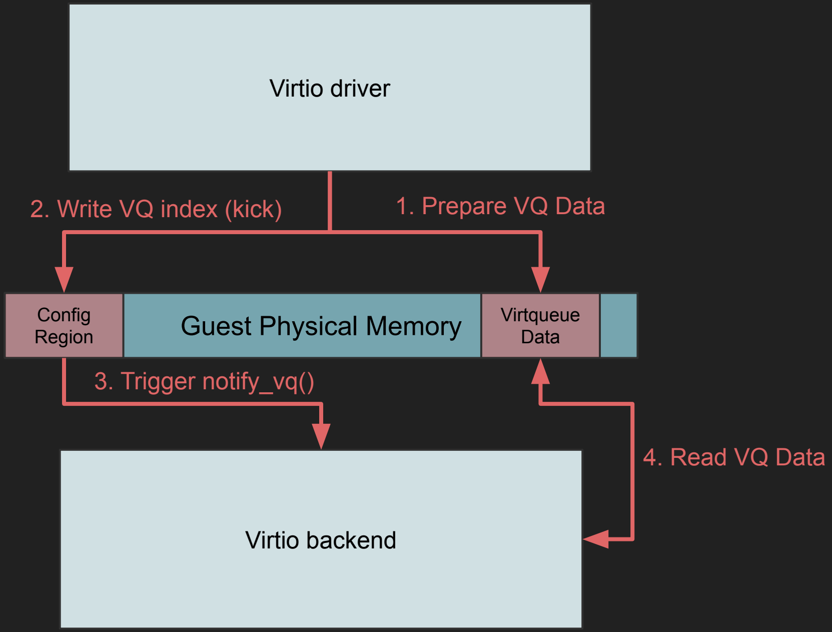
- "Paravirtualized" communication between guest kernel and host
    - Guest kernel driver builds queue in guest physical memory, then writes to special mapped memory to "kick" host
    - Callback triggered in host backend, which reads from guest physical memory
- KVMTool example: Virtio Balloon devices
    - Balloon driver sends memory statistics to host (Virtqueue 1), host sends inflate/deflate commands (Virtqueues 2 and 3)



Virtio Balloon Device Bug

- `notify_vq()` called when guest driver kicks backend
  - Picks job based on virqueue index from guest memory, passes to threadpool
- Bug: Does not check bounds of virtqueue index

```
// kvmtool/virtio/balloon.c
static int notify_vq(struct kvm *kvm, void *dev, u32 vq)
{
        struct bln_dev *bdev = dev;

        thread_pool__do_job(&bdev->jobs[vq]);

        return 0;
}
```



Exploiting Bug

1. Overflow in notify_vq allows launching with future thread job structure

2. Can send arbitrary bytes into stats buffer

Strategy: construct malicious job in stats buffer, then overflow to launch it

```c
// kvmtool/virtio/balloon.c
struct bln_dev {

      /* ... */

      struct virt_queuevqs[NUM_VIRT_QUEUES];
      struct thread_pool__job jobs[NUM_VIRT_QUEUES];

      /* ... */

      // stats buffer, controllable from guest
      struct virtio_balloon_stat stats[VIRTIO_BALLOON_S_NR];

      /* ... */

};
```

```c
// kvmtool/virtio/balloon.c
static int notify_vq(struct kvm *kvm, void *dev, u32 vq)
{
      struct bln_dev *bdev = dev;

      thread_pool__do_job(&bdev->jobs[vq]);

      return 0;
}
```

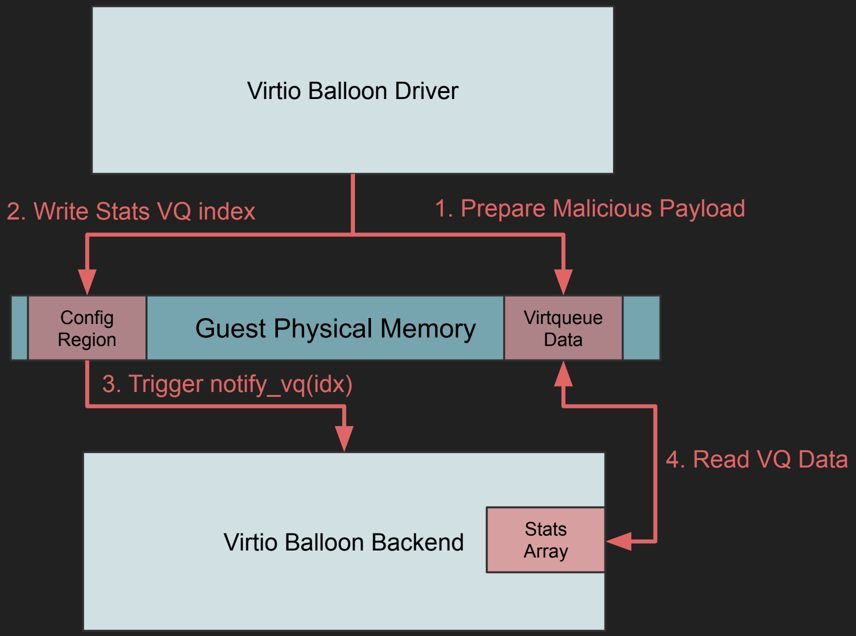
Construct Malicious Job Struct

```
// Expected thread structure from KVMTool (88 bytes)
struct thread_pool__job
{
      kvm_thread_callback_fn_t callback       // ptr: virtio_net_exec_script(char *script, char *tag)
      void *kvm;                              // First argument: pointer to arg_str that holds "/bin/sh"
      void *data;                 // Second argument: NULL [8 bytes]

      int signalcount;            // Threads already processing job, set to 0 [4 bytes]
      char padding[4];            // Alignment padding [4 bytes]
      char mutex[40];             // Job mutex, must start in unlocked state [40 bytes]

      struct list_head queue;     // List entry for job queue, point to self [16 bytes]

};   // Overall 88 bytes

// Payload with additional bytes for “/bin/sh”
struct job_payload {
      struct thread_pool__job job;
      char arg_str[8];      // Store "/bin/sh", pointed to by job.kvm [8 bytes]
  };  // Overall 96 bytes
```

Payload structure to be constructed in guest, and sent to stats array in KVMTool on host

Deliver Payload via Virtqueue

1. Wrap malicious job struct in Virtqueue structure
2. Kick backend by writing stats VQ index to config region
3. Backend copies payload into stats array in host memory
4. Trigger payload by writing out-of-bounds VQ index to config region



Exploit Implementation

Must deliver to guest physical memory, only via root shell:

1. Write kernel module containing exploit code (escape_mod.c)
2. Compile module against local 6.13.4 Kernel (escape_mod.ko)
3. Chunk binary and send in ASCII via shell print statements
4. Insert and execute module

Demo

Mitigation and Implications

- Mitigations
  - Specific mitigation: add array bound checking
  - General mitigation: check all inputs from guest VM
- Implications
  - Real bug in KVMTool, but KVMTool is “a hobby project”
  - Not that big of a deal
  - Other solutions found other bugs