Blue-Pill Oxpecker: Enabling Transaction Based Writing for VMI Technical Details

I. LibVMI APIs

The main read/write APIs of LibVMI are *vmi_read_X* and *vmi_write_X* functions. These APIs get virtual/physical addresses alongside with the context that they must be evaluated in, before finding the appropriate host memory address. LibVMI can execute operation on kernel memory by using the zero as the process id. Table I lists the typical usage and description of important LibVMI read/write APIs.

TABLE I: LibVMI read/write APIs. The *X* character represents different bit size numbers in the function name. For example, *vmi_read_X_va* represents *vmi_read_4_va*, *vmi_read_8_va*, etc.

API Name	Description	Example
vmi_read_X	Read X bits from given (physical or virtual) address based on context and place output in buffer.	`access_context_t ctx = { .translate_mechanism = VMI_TM_PROCESS_PID, .addr = vaddr, .pid = pid }; vmi_read_32(vmi, &ctx, ret_val);`
vmi_read_addr	Read from (64 or 32 bits) address based on system and put the result in buffer.	`vmi_read_addr(vmi, &ctx, ret_val);`
vmi_read_str	Read from address as a string.	`vmi_read_str(vmi, &ctx);`
vmi_read_X_va	Read from virtual address and return the result in buffer.	`vmi_read_32_va(vmi, vaddr, pid, ret_val);`
vmi_read_X_pa	Read from physical address and return the result in buffer.	`vmi_read_32_pa(vmi, paddr, ret_val);`
vmi_write_X	Same as *vmi_read_X* but for writing.	`vmi_write_32_pa(vmi, paddr, value);`

II. Syscall Interception using Nitro Framework

Nitro employs different approaches for intercepting *iret*, *sysexit*, and *sysret* assembly instructions. To intercept interrupt-based syscalls (i.e. *"int 0x80"*), it sets the *Interrupt Descriptor Table Register (IDTR)* which indicates size of Interrupt Descriptor Table (IDT) to 255. Therefore, all higher interrupt numbers will raise a *general protection fault* which can be caught in VMM. For `sysret` instruction, the `SCE` flag in the Extended Feature Enable Register (EFER) is cleared in order to raise an *invalid opcode exception* upon execution of `sysret` instructions. When an invalid opcode exception is caught by hypervisor, it detects execution of *sysret* (and similarly *syscall*) instruction by looking at the corresponding instruction that has raised the exception. Finally, *sysexit* is intercepted similar to *sysret*, but by changing *CS* register to *NULL* value which raises a *general protection fault* exception during the execution of *sysexit/sysenter* instructions. All these scenarios lead to *VM_EXIT* events at hypervisor.

Nitro allows a userspace process to configure hypervisor through a series of *ioctl* calls. Following four *ioctl* calls provide the main API for intercepting syscalls:

1) *KVM_NITRO_ATTACH_VCPUS*: used to attach to a virtual CPU,
2) *KVM_NITRO_SYSCALL_TRAP*: used for setting syscall traps for a virtual CPU in GVM,
3) *KVM_NITRO_GET_EVENT*: used for registering a userspace listener for GVM events,
4) *KVM_NITRO_GET_CONTINUE*: used for resuming the execution of GVM.

III. Kill Example

Fig. 1 shows an example code using the Oxpecker to kill a guest process. More details are available in the Oxpecker Github repository [1].

```python
import libvirt, asyncio
from nitro.nitro import Nitro
from ox_api import Oxpecker

def main(args):
  vm_name = args['<vm_name>']
  con = libvirt.open('qemu:///system')
  domain = con.lookupByName(vm_name)

  nitro = Nitro(domain, True)
  ox = Oxpecker(nitro)
  pid = 619

  def callback():
    kill = Kill(ox)
    kill.do_exit(pid)
  loop = asyncio.get_event_loop()
  loop.run_until_complete(
    asyncio.wait_for(
      ox.beginTransaction(callback, loop), timeout=3
    )
  )
  loop.close()
  ox.cancelTransaction()
```

Fig. 1: Example of using Oxpecker for killing a guest process.

IV. Lock Module

The source code of the sample lock module which was used for evaluation of write consistency is depicted in Fig. 2. This module is used to demonstrate the LibVMI inconsistency.

```
int init_module(void)
{
  unsigned long j0,j1;
  int delay = 1*HZ;
  uint32_t dining_spoon = 30;
  uint32_t crash = 0;
  printk(KERN_INFO "The address is %p\n", &dining_spoon);
  while(!fatal_signal_pending(current))
  {
    j0 = jiffies;
    j1 = j0 + delay;
    spin_lock(&my_lock);
    dining_spoon = 30;
    while (time_before(jiffies, j1))
      schedule();
    if(dining_spoon != 30)
      crash += 1;
    spin_unlock(&my_lock);
    msleep_interruptible(1000);
    printk(KERN_INFO "Total crash number %d\n", crash);
    printk(KERN_INFO "The case is %d\n", dining_spoon);
  }
  return 0;
}
```

Fig. 2: The lock module which is used to create race conditions with concurrent Oxpecker transactions. This kernel module waits for a second with `msleep_interruptible` and busy waits while holding the `my_lock` for one second. Oxpecker is expected to limit its writes to the period that `my_lock` is released.

References

[1] (2019, May) Consistent writable vmi api. [Online]. Available: https://github.com/Oxpecker-VMI/oxpecker