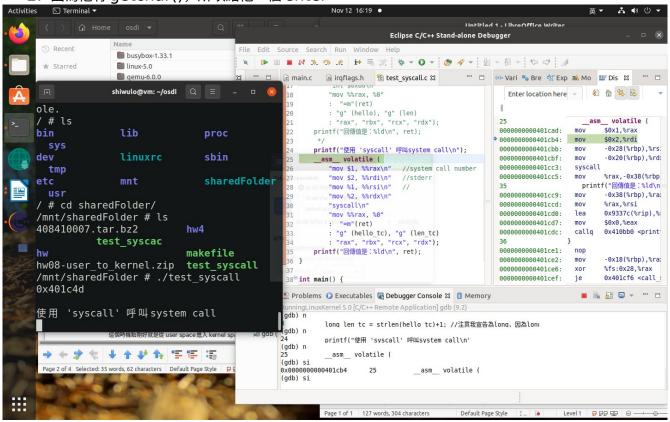
enter(跑出(gdb)), c讓他開機,

執行./test_syscall 後,要 ctrl_c,讓 eclipse 告訴 demu 說我拿到控制權(送 signal)file /home/shiwulo/osdi/sharedFolder/test_syscall

(沒有 start_kernel func,因為現在是對應用程式(test_syscall)做 debug,所以載入的是他的 symbol table。)

- 一:設定中斷點在 test_syscall 發出 system call 之前(例如:將中斷點設定在 call_sys),請在這個地方截圖
 - 1. b*(0x401c4d),按一個 c
 - 2. 因為他有 getchar(),所以給他一個 enter

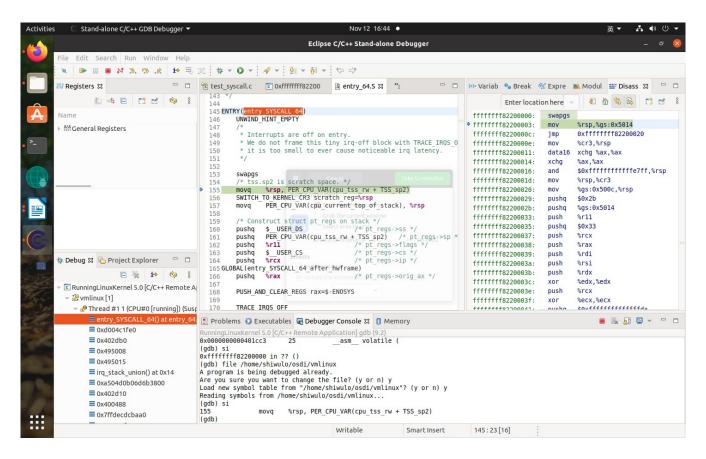


-:

使用單步追蹤(si),直到 syscall 後(注意: syscall 之後就是跳入到 Linux kernel),先在 Debugger Console 輸入「file /home/shiwulo/osdi/vmlinux」,然後對 Eclipse 拍照。這個時機點剛好就是從 user space 進入 kernel space 時。

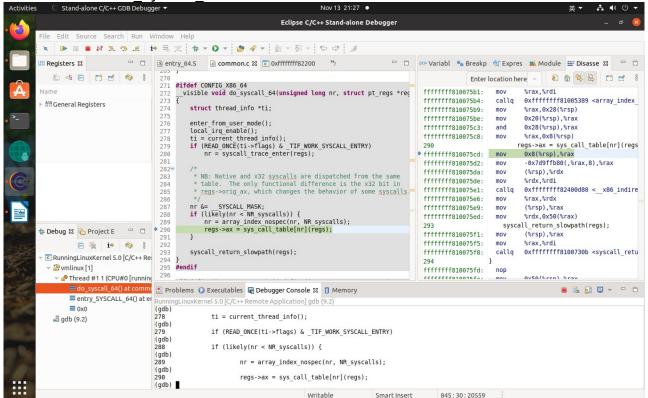
//這裡一開始的時候,就是把暫存器 push 到堆疊裡面。

- 1. 先進 swapgs,輸入「file /home/shiwulo/osdi/vmlinux」,把 debug sybmol 把 他切換成 linux kernel 的 debug symbol
- 2. 之後要再 si 一次,才看得到左側程式碼



三:請說明 Linux kernel 如何用 RAX 暫存器判斷要呼叫哪個 Linux 內部的函數,請說明該函數的名稱





2. sys call table 是函數指標,指向一開始 push 的那些暫存器,regs 是那些 reg

3. <u>SYSCALL_DEFINE3</u> 是 maincore,展開後叫 do_sys_write,呼叫 ksys_write Stand-alone C/C++ GDB Debugger ▼ Nov 12 17:01 • Eclipse C/C++ Stand-alone Debugger File Edit Source Search Run Window Help ##Registers ⊠ □ □ Sentry_64.S Common.c Se barrier.h Se read_write.c ⊠ "4 □ □ (*)• Variab ** Break ** Expre ** Modul *** Disass ⊠ □ □ Enter location here - & h 5 6 1 1 1 1 if (f.file) {
 loff t pos = file pos_read(f.file);
 ret = vfs_write(f.file, buf, count, &pos);
 if (ret >= 0) _do_sys_write: • fffffffff8147013b: sub \$0x18,%rsp ▶ ∰ General Registers file_pos_write(f.file, pos);
fdput_pos(f); 599 ffffffff8147013f: fffffffff81470143: mov %rsi.0x8(%rsp) 601 ffffffff81470148: %rdx,(%rsp) return ksys write(fd, buf, count) 610 663
694 return ret;
605 }
606
607 SYSCALL_DEFINE3(write, unsigned int, fd, const char user *, buf,
688 size_t, count) fffffffff8147014c: (%rsp),%rdx ffffffff81470150: 0x8(%rsp),%rcx mov ffffffff81470155: 0x14(%rsp).%eax ffffffff81470159: %rcx,%rsi mov 609 { fffffffff8147015c %eax.%edi 0xffffffff8146ffd2 <ksys_wr return ksys write(fd, buf, count); fffffffff8147015e: 610 611 } 611 } ffffffff81470163: add \$0x18,%rsp 613 ssize_t ksys_pread64(unsigned int fd, char _user *buf, size_t count, 614 _____loff_t pos) fffffffff81470167: reta 614 615 { 616 617 618 619 ☼ Debug ☼ Project E struct fd f;
ssize_t ret = -EBADF; ksys pread64: □ ¾ i→ 6 8 ffffffff81470168: %rbp ffffffff81470169: mov %rsp,%rbp ▼ © RunningLinuxKernel 5.0 [C/C++ Re if (pos < 0)
 return -EINVAL;</pre> ffffffff8147016c: %r10 ▼ ? vmlinux [1] fffffffff8147016e: \$0x38,%rsp 621 ▼ P Thread #1 1 (CPU#0 [running Av1A/9rhn) = _se_sys_write() at read_ RunningLinuxKernel 5.0 [c/c++ Remote Application] gdb (9.2)
= _x64_sys_write() at read_ (gdb) s (gdb) s
 se_sys_write (fd=-60473137103128, buf=582, count=16) at fs/read_write.c:607
 SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, **≡** 0x0 adb (9.2) do sys_write (fd=4294967295, buf=0x1 <irq_stack_union+1> <error: Cannot access memory at address 0x1>, count=1844668360057 2448600) at fs/read_write.c:609 (gdb) **[** #

```
nr = array_index_nospec(nr, NR_syscalls);
regs->ax = sys_call_table[nr](regs);
```

4. 以下都要用 s, 因為不是組合語言

SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,size_t, count) {return ksys_write(fd, buf, count);}

四:<mark>請用 50~200 個「有意義的文字」大致說明作業系統如何處理該 system call</mark>(例如:可以將註解英翻中, 或者列出呼叫流程。請盡可能表達。這一題會依照所寫的內容的完整性給分)。

- * Lightweight file lookup no <u>refcnt</u> increment if <u>fd</u> table isn't shared.
- * You can use this instead of <u>fget</u> if you satisfy all of the following * conditions:
- * 1) You must call fput_light before exiting the <u>syscall</u> and returning control

 * to <u>userspace</u> (i.e. you cannot remember the returned <u>struct</u> file * after

 * returning to <u>userspace</u>).
- * 2) You must not call filp_close on the returned <u>struct</u> file * in between * calls to fget light and fput light.
- * 3) You must not clone the current task in between the calls to fget_light * and fput light.
- * The fput_needed flag returned by fget_light should be passed to the * corresponding fput_light.

```
759@ static unsigned long fget light(unsigned int fd, fmode t mask)
760 {
        struct files struct *files = current->files;
761
        struct file *file;
762
763
        if (atomic read(&files->count) == 1) {
764
            file = fcheck files(files, fd);
765
            if (!file || unlikely(file->f mode & mask))
766
767
                return 0;
768
            return (unsigned long)file;
        } else {
769
            file = fget(fd, mask);
770
            if (!file)
771
772
                return 0;
773
            return FDPUT FPUT | (unsigned long)file;
        }
774
775 }
776⊖ unsigned long fdget(unsigned int fd)
777 {
778
        return fget light(fd, FMODE PATH);
779 }
780 EXPORT SYMBOL( fdget);
781
782⊕ unsigned long fdget raw(unsigned int fd)
787⊖ unsigned long __fdget_pos(unsigned int fd)
788 {
        unsigned long v = fdget(fd);
789
        struct file *file = (struct file *)(v & ~3);
790
791
        if (file && (file->f mode & FMODE ATOMIC POS)) {
792
            if (file count(file) > 1) {
793
                v |= FDPUT POS UNLOCK;
794
                mutex lock(&file->f pos lock);
795
796
            }
        }
797
798
        return v;
```

在___fget_light 裡回傳的 file,給___fdget(上層呼叫的),最後是 v。把回傳值變成指標,用 fd 去查詢,得 到描述這個檔案的一個資料結構,file 這個指標。(在___fdgets_pos)

■ 在 ksys_write 中首先呼叫 fdget_pos(),這個函數似乎是「鎖定」這個檔案,同時他會回傳一個資料結構用以代表「檔案」,這個資料結構包含 fields 及 function pointers。

```
591 ssize t ksys write(unsigned int fd, const char user *buf, size t count)
592 {
        struct fd f = fdget pos(fd);
593
        ssize t ret = -EBADF;
594
595
        if (f.file) {
596
            loff t pos = file pos read(f.file);
597
             ret = vfs write(f.file, buf, count, &pos);
598
             if (ret >= 0)
599
                 file pos write(f.file, pos);
600
             fdput pos(f);
601
        }
602
603
        return ret;
604
605 }
```

597.598 行: 拿到檔案的資料結構,再拿到檔案的位置,位置往後讀,要把資料讀到 buf, buf 前面是定義一個 __user,所以這個指標是指到 user space 的指標。

■ 呼叫 vfs_write(f.file, buf, count, &pos)做真正的寫入。因為每個檔案可能對應到不同的裝置,因此 vfs write 的行為,取決於 f.file

```
◆ 這個地方應該繼續往下追, vfs write 應該是呼叫 f 裡面的某個函數
533 ssize t vfs write(struct file *file, const char user *buf, size t count, loff t *pos)
534 {
535
         ssize t ret;
536
         if (!(file->f mode & FMODE WRITE))
537
538
             return - EBADF;
         if (!(file->f mode & FMODE CAN WRITE))
539
540
             return -EINVAL;
         if (unlikely(!access ok(buf, count)))
541
             return -EFAULT;
542
543
544
         ret = rw verify area(WRITE, file, pos, count);
545
         if (!ret) {
             if (count > MAX RW COUNT)
546
                 count = MAX RW COUNT;
547
             file start write(file);
548
             ret = vfs write(file, buf, count, pos);
549
             if (ret > 0) {
550
                 fsnotify modify(file);
551
                 add wchar(current, ret);
552
553
             inc syscw(current);
554
             file end write(file);
555
         }
556
557
         return ret;
558
559 }
```

EBADF: fd 不是有效的文件描述符或未打開寫入

EINVAL: fd 附加到不適合寫入的對像上;或者文件是使用 0_DIRECT 標誌打開的,並且 buf 中指定的地址、count 中指定的值或文件偏移量未適當對齊。

EFAULT: buf 超出您的可訪問地址空間。

■ 如果上述的 write 成功,那麼會將 pos 更新到 f.file

```
566⊖ static inline void file_pos_write(struct file *file, loff_t pos)
567 {
568    file->f_pos = pos;
569 }
```

■ 因為已經更新完畢,呼叫 fdput_pos,解開「鎖定」