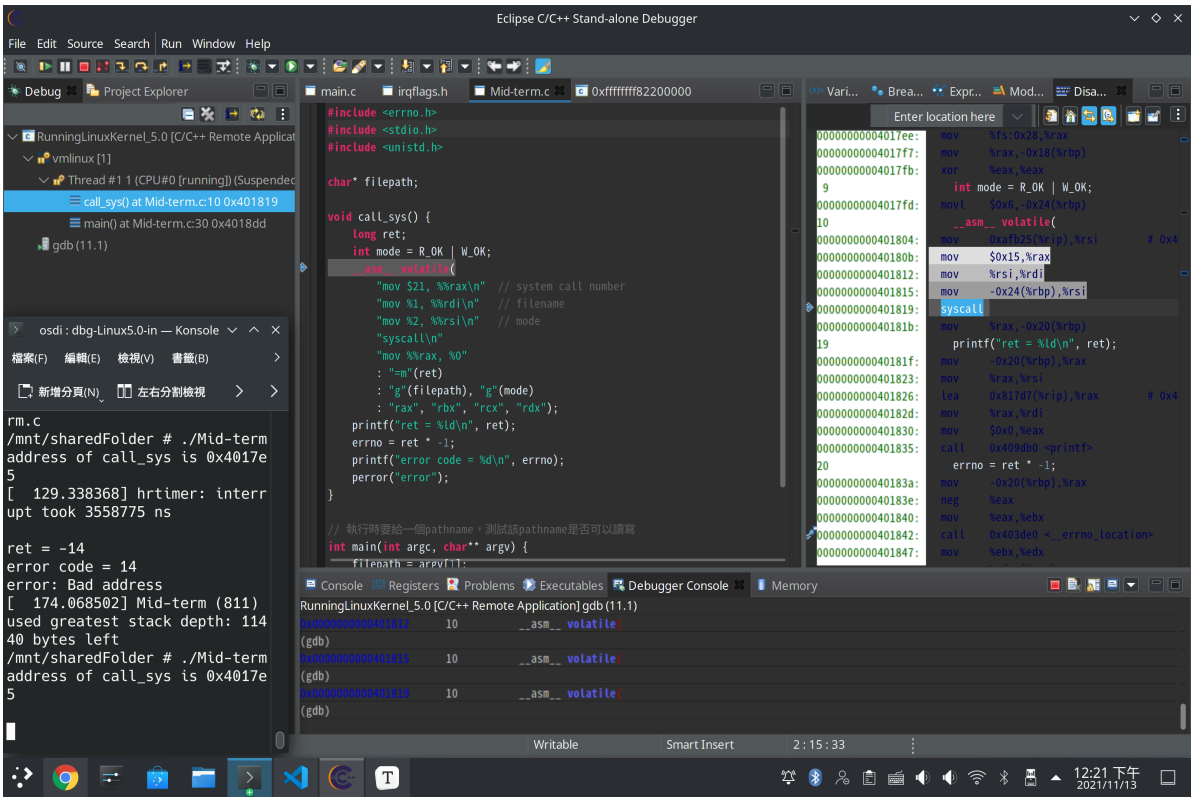
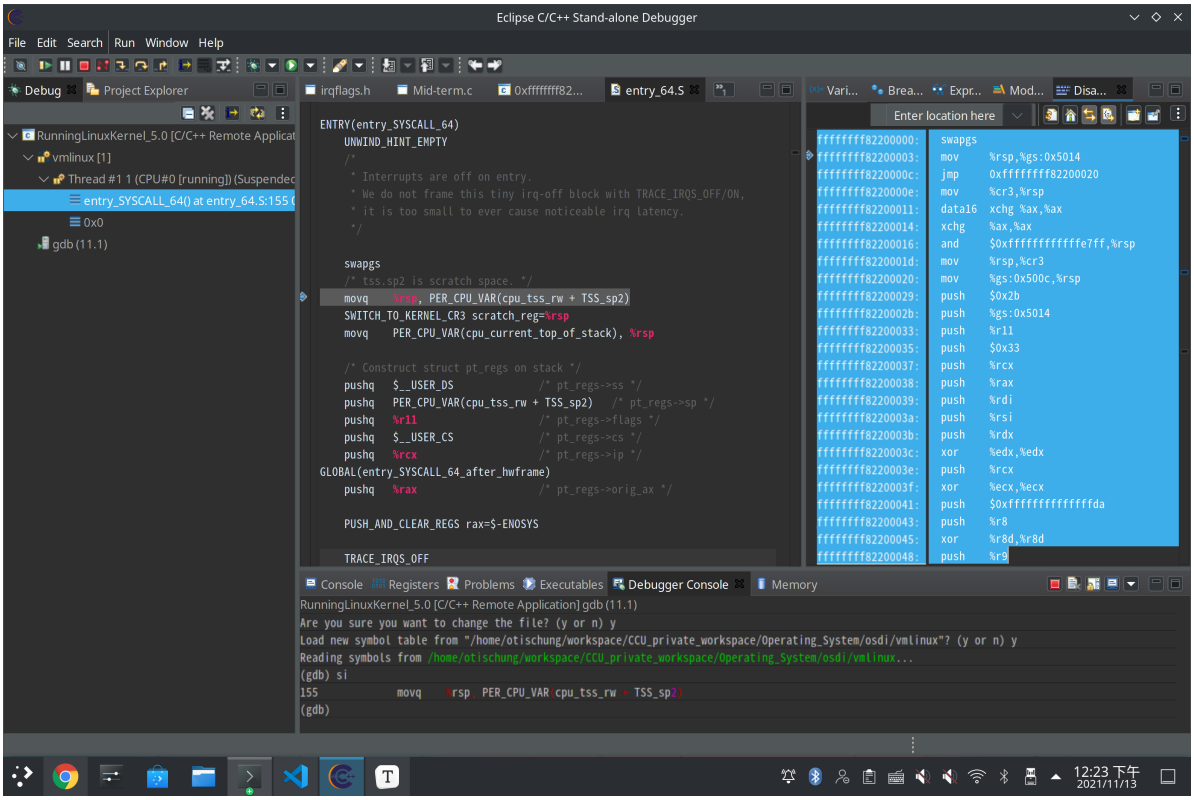
設定中斷點在 test_syscall 發出 system call 之前

為了方便，我將 filename 改為在執行目錄底下的 Makefile，這樣我就不用在 GDB 裡面輸入 argv[1]



使用單步追蹤（si），直到 syscall 後



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

在 entry_64.S 裡，有這麼一段程式碼

```
1  ENTRY(entry_SYSCALL_64)
2      UNWIND_HINT_EMPTY
3      /*
4       * Interrupts are off on entry.
5       * We do not frame this tiny irq-off block with TRACE_IRQS_OFF/ON,
6       * it is too small to ever cause noticeable irq latency.
7       */
8
9      swapgs
10     /* tss.sp2 is scratch space. */
11     movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
12     SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
13     movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp
14
15     /* Construct struct pt_regs on stack */
16     pushq   $__USER_DS              /* pt_regs->ss */
17     pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)   /* pt_regs->sp */
18     pushq   %r11                    /* pt_regs->flags */
19     pushq   $__USER_CS              /* pt_regs->cs */
20     pushq   %rcx                    /* pt_regs->ip */
21 GLOBAL(entry_SYSCALL_64_after_hwframe)
22     pushq   %rax                    /* pt_regs->orig_ax */
23
24     PUSH_AND_CLEAR_REGS rax=$-ENOSYS
25
26     TRACE_IRQS_OFF
27
28     /* IRQs are off. */
29     movq    %rax, %rdi
30     movq    %rsp, %rsi
31     call    do_syscall_64       /* returns with IRQs disabled */
```

我們觀察一下第 31 行的 do_syscall_64，可以發現在 common.c 裡，有這麼一段程式碼

```
1  __visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
2  {
3      struct thread_info *ti;
4
5      enter_from_user_mode();
6      local_irq_enable();
7      ti = current_thread_info();
8      if (READ_ONCE(ti->flags) & _TIF_WORK_SYSCALL_ENTRY)
9          nr = syscall_trace_enter(regs);
10
11     /*
12      * NB: Native and x32 syscalls are dispatched from the same
13      * table.  The only functional difference is the x32 bit in
14      * regs->orig_ax, which changes the behavior of some syscalls.
15      */
```

```
16      nr &= __SYSCALL_MASK;
17      if (likely(nr < NR_syscalls)) {
18          nr = array_index_nospec(nr, NR_syscalls);
19          regs->ax = sys_call_table[nr](regs);
20      }
21
22      syscall_return_slowpath(regs);
23  }
```

其中第 9 行的 `nr = syscall_trace_enter(regs);`，我們看到程式將傳進來的 regs 放入 nr 裡面，在第 19 行裡面，將查完 sys_call_table 的結果放入 rax 暫存器裡面，這樣就完成放置 system call 的第一個變數了

---

step in 進去第 19 行，可以在 open.c 裡看到以下程式碼

```
1  SYSCALL_DEFINE2(access, const char __user *, filename, int, mode)
2  {
3      return do_faccessat(AT_FDCWD, filename, mode);
4  }
```

step in 進去第 3 行，可以在 open.c 裡看到以下程式碼

```
1  /*
2   * access() needs to use the real uid/gid, not the effective uid/gid.
3   * We do this by temporarily clearing all FS-related capabilities and
4   * switching the fsuid/fsgid around to the real ones.
5   */
6  long do_faccessat(int dfd, const char __user *filename, int mode)
7  {
8      const struct cred *old_cred;
9      struct cred *override_cred;
10     struct path path;
11     struct inode *inode;
12     int res;
13     unsigned int lookup_flags = LOOKUP_FOLLOW;
14
15     if (mode & ~S_IRWXO)    /* where's F_OK, X_OK, W_OK, R_OK? */
16         return -EINVAL;
17
18     override_cred = prepare_creds();
19     if (!override_cred)
20         return -ENOMEM;
21
22     override_cred->fsuid = override_cred->uid;
23     override_cred->fsgid = override_cred->gid;
24
25     if (!issecure(SECURE_NO_SETUID_FIXUP)) {
26         /* Clear the capabilities if we switch to a non-root user */
27         kuid_t root_uid = make_kuid(override_cred->user_ns, 0);
28         if (!uid_eq(override_cred->uid, root_uid))
29             cap_clear(override_cred->cap_effective);
30         else
31             override_cred->cap_effective =
32                 override_cred->cap_permitted;
33     }
```

```
34
35      old_cred = override_creds(override_cred);
36  retry:
37      res = user_path_at(dfd, filename, lookup_flags, &path);
38      if (res)
39          goto out;
40
41      inode = d_backing_inode(path.dentry);
42
43      if ((mode & MAY_EXEC) && S_ISREG(inode->i_mode)) {
44          /*
45           * MAY_EXEC on regular files is denied if the fs is mounted
46           * with the "noexec" flag.
47           */
48          res = -EACCES;
49          if (path_noexec(&path))
50              goto out_path_release;
51      }
52
53      res = inode_permission(inode, mode | MAY_ACCESS);
54      /* SuS v2 requires we report a read only fs too */
55      if (res || !(mode & S_IWOTH) || special_file(inode->i_mode))
56          goto out_path_release;
57      /*
58       * This is a rare case where using __mnt_is_readonly()
59       * is OK without a mnt_want/drop_write() pair.  Since
60       * no actual write to the fs is performed here, we do
61       * not need to telegraph to that to anyone.
62       *
63       * By doing this, we accept that this access is
64       * inherently racy and know that the fs may change
65       * state before we even see this result.
66       */
67      if (__mnt_is_readonly(path.mnt))
68          res = -EROFS;
69
70  out_path_release:
71      path_put(&path);
72      if (retry_estale(res, lookup_flags)) {
73          lookup_flags |= LOOKUP_REVAL;
74          goto retry;
75      }
76  out:
77      revert_creds(old_cred);
78      put_cred(override_cred);
79      return res;
80  }
```

可以發現，該程式在第 15 行時檢查 mode 是 F_OK, X_OK, W_OK, R_OK，故推斷**第 21 號 system call 是 access**

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

在第 37 行時 `user_path_at` 這個函數檢查 filename 的權限並將結果寫回 res (result)，我們 step in，發現在 namei.h 裡有一段程式碼

```
static inline int user_path_at(int dfd, const char __user *name, unsigned
flags,
         struct path *path)
{
    return user_path_at_empty(dfd, name, flags, path, NULL);
}
```

再次 step in，發現在 namei.h 裡有一段程式碼

```
int user_path_at_empty(int dfd, const char __user *name, unsigned flags,
         struct path *path, int *empty)
{
    return filename_lookup(dfd, getname_flags(name, flags, empty),
                    flags, path, NULL);
}
```

在 namei.c 裡，找到以下程式碼

```
static int filename_lookup(int dfd, struct filename *name, unsigned flags,
                struct path *path, struct path *root)
{
    int retval;
    struct nameidata nd;
    if (IS_ERR(name))
        return PTR_ERR(name);
    if (unlikely(root)) {
        nd.root = *root;
        flags |= LOOKUP_ROOT;
    }
    set_nameidata(&nd, dfd, name);
    retval = path_lookupat(&nd, flags | LOOKUP_RCU, path);
    if (unlikely(retval == -ECHILD))
        retval = path_lookupat(&nd, flags, path);
    if (unlikely(retval == -ESTALE))
        retval = path_lookupat(&nd, flags | LOOKUP_REVAL, path);

    if (likely(!retval))
        audit_inode(name, path->dentry, flags & LOOKUP_PARENT);
    restore_nameidata();
    putname(name);
    return retval;
}
```

我們繼續追蹤第 13 行 `path_lookupat`，可以在 namei.c 裡，找到以下程式碼

```
/* Returns 0 and nd will be valid on success; Retuns error, otherwise. */
static int path_lookupat(struct nameidata *nd, unsigned flags, struct path
*path)
{
    const char *s = path_init(nd, flags);
    int err;

```

```
    if (unlikely(flags & LOOKUP_DOWN) && !IS_ERR(s)) {
        err = handle_lookup_down(nd);
        if (unlikely(err < 0))
            s = ERR_PTR(err);
    }

    while (!(err = link_path_walk(s, nd))
        && ((err = lookup_last(nd)) > 0)) {
        s = trailing_symlink(nd);
    }
    if (!err)
        err = complete_walk(nd);

    if (!err && nd->flags & LOOKUP_DIRECTORY)
        if (!d_can_lookup(nd->path.dentry))
            err = -ENOTDIR;
    if (!err) {
        *path = nd->path;
        nd->path.mnt = NULL;
        nd->path.dentry = NULL;
    }
    terminate_walk(nd);
    return err;
}
```

所以我們已經知道，如果該 filename 的權限符合的話，就會 return 0，所以在 `do_faccessat` 裡面的第 37 行 `res = user_path_at(dfd, filename, lookup_flags, &path);` 會 return 0

接下來，看看 `do_faccessat` 第 53 行 `res = inode_permission(inode, mode | MAY_ACCESS);` 裡面是什麼，我們 step in 進去看，看到在 `namei.c` 裡面有以下程式碼

```
int inode_permission(struct inode *inode, int mask)
{
    int retval;

    retval = sb_permission(inode->i_sb, inode, mask);
    if (retval)
        return retval;

    if (unlikely(mask & MAY_WRITE)) {
        /*
         * Nobody gets write access to an immutable file.
         */
        if (IS_IMMUTABLE(inode))
            return -EPERM;

        /*
         * Updating mtime will likely cause i_uid and i_gid to be
         * written back improperly if their true value is unknown
         * to the vfs.
         */
        if (HAS_UNMAPPED_ID(inode))
            return -EACCES;
    }

    retval = do_inode_permission(inode, mask);
```

```
26      if (retval)
27          return retval;
28
29      retval = devcgroup_inode_permission(inode, mask);
30      if (retval)
31          return retval;
32
33      return security_inode_permission(inode, mask);
34  }
```

我們看一下第 5 行的 `sb_permission`，可以看到在 namei.c 裡有以下程式碼

```
1   /**
2    * sb_permission - Check superblock-level permissions
3    * @sb: Superblock of inode to check permission on
4    * @inode: Inode to check permission on
5    * @mask: Right to check for (%MAY_READ, %MAY_WRITE, %MAY_EXEC)
6    *
7    * Separate out file-system wide checks from inode-specific permission
    checks.
8    */
9   static int sb_permission(struct super_block *sb, struct inode *inode, int
    mask)
10  {
11      if (unlikely(mask & MAY_WRITE)) {
12          umode_t mode = inode->i_mode;
13
14          /* Nobody gets write access to a read-only fs. */
15          if (sb_rdonly(sb) && (S_ISREG(mode) || S_ISDIR(mode) ||
    S_ISLNK(mode)))
16              return -EROFS;
17      }
18      return 0;
19  }
```

可以發現，只要 permission 符合，就 return 0

我們看一下 `inode_permission` 第 25 行的 `do_inode_permission`，可以看到在 namei.c 裡有以下程式碼

```
1   /*
2    * We _really_ want to just do "generic_permission()" without
3    * even looking at the inode->i_op values. So we keep a cache
4    * flag in inode->i_opflags, that says "this has not special
5    * permission function, use the fast case".
6    */
7   static inline int do_inode_permission(struct inode *inode, int mask)
8   {
9       if (unlikely(!(inode->i_opflags & IOP_FASTPERM))) {
10          if (likely(inode->i_op->permission))
11              return inode->i_op->permission(inode, mask);
12
13          /* This gets set once for the inode lifetime */
14          spin_lock(&inode->i_lock);
15          inode->i_opflags |= IOP_FASTPERM;
```

```
16          spin_unlock(&inode->i_lock);
17      }
18      return generic_permission(inode, mask);
19  }
```

我們看一下第 18 行的 `generic_permission`，可以看到在 namei.c 裡有以下程式碼

```
1   /**
2    * generic_permission -  check for access rights on a Posix-like filesystem
3    * @inode:  inode to check access rights for
4    * @mask:   right to check for (%MAY_READ, %MAY_WRITE, %MAY_EXEC, ...)
5    *
6    * Used to check for read/write/execute permissions on a file.
7    * We use "fsuid" for this, letting us set arbitrary permissions
8    * for filesystem access without changing the "normal" uids which
9    * are used for other things.
10   *
11   * generic_permission is rcu-walk aware. It returns -ECHILD in case an rcu-
     walk
12   * request cannot be satisfied (eg. requires blocking or too much
     complexity).
13   * It would then be called again in ref-walk mode.
14   */
15  int generic_permission(struct inode *inode, int mask)
16  {
17      int ret;
18
19      /*
20       * Do the basic permission checks.
21       */
22      ret = acl_permission_check(inode, mask);
23      if (ret != -EACCES)
24          return ret;
25
26      if (S_ISDIR(inode->i_mode)) {
27          /* DACs are overridable for directories */
28          if (!(mask & MAY_WRITE))
29              if (capable_wrt_inode_uidgid(inode,
30                                           CAP_DAC_READ_SEARCH))
31                  return 0;
32          if (capable_wrt_inode_uidgid(inode, CAP_DAC_OVERRIDE))
33              return 0;
34          return -EACCES;
35      }
36
37      /*
38       * Searching includes executable on directories, else just read.
39       */
40      mask &= MAY_READ | MAY_WRITE | MAY_EXEC;
41      if (mask == MAY_READ)
42          if (capable_wrt_inode_uidgid(inode, CAP_DAC_READ_SEARCH))
43              return 0;
44      /*
45       * Read/write DACs are always overridable.
46       * Executable DACs are overridable when there is
47       * at least one exec bit set.
48       */
```

```
49      if (!(mask & MAY_EXEC) || (inode->i_mode & S_IXUGO))
50          if (capable_wrt_inode_uidgid(inode, CAP_DAC_OVERRIDE))
51              return 0;
52
53      return -EACCES;
54  }
```

可以發現這裡有檢查 ACL 的權限，如果全部權限符合，就會進入第 51 行的 `return 0`

最後，看一下 `inode_permission` 第 33 行最後 return 的 `security_inode_permission`，我們在 `security.c` 裡找到以下程式碼

```
1  int security_inode_permission(struct inode *inode, int mask)
2  {
3      if (unlikely(IS_PRIVATE(inode)))
4          return 0;
5      return call_int_hook(inode_permission, 0, inode, mask);
6  }
```

我們看一下第 5 行的 `call_int_hook`，我們在 `hooks.c` 裡找到以下程式碼

```
1  static int selinux_inode_permission(struct inode *inode, int mask)
2  {
3      const struct cred *cred = current_cred();
4      u32 perms;
5      bool from_access;
6      unsigned flags = mask & MAY_NOT_BLOCK;
7      struct inode_security_struct *isec;
8      u32 sid;
9      struct av_decision avd;
10     int rc, rc2;
11     u32 audited, denied;
12
13     from_access = mask & MAY_ACCESS;
14     mask &= (MAY_READ|MAY_WRITE|MAY_EXEC|MAY_APPEND);
15
16     /* No permission to check.  Existence test. */
17     if (!mask)
18         return 0;
19
20     validate_creds(cred);
21
22     if (unlikely(IS_PRIVATE(inode)))
23         return 0;
24
25     perms = file_mask_to_av(inode->i_mode, mask);
26
27     sid = cred_sid(cred);
28     isec = inode_security_rcu(inode, flags & MAY_NOT_BLOCK);
29     if (IS_ERR(isec))
30         return PTR_ERR(isec);
31
32     rc = avc_has_perm_noaudit(&selinux_state,
33                   sid, isec->sid, isec->sclass, perms, 0, &avd);
34     audited = avc_audit_required(perms, &avd, rc,
35                      from_access ? FILE__AUDIT_ACCESS : 0,
```

```
36                      &denied);
37      if (likely(!audited))
38          return rc;
39
40      rc2 = audit_inode_permission(inode, perms, audited, denied, rc, flags);
41      if (rc2)
42          return rc2;
43      return rc;
44  }
```

至此，所有權限檢查完畢， 若權限都吻合，`open.c` 裡面 `do_faccessat` 裡的 `inode_permission` 會 return 0