CS 6V81-05: System Security and Malicious Code Analysis
Understanding the Implementation of Virtual Memory

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

February 29th, 2012
Outline

1. Overview
2. Implementation
3. Summary
Virtual Memory of a Linux Process

- **Kernel code and data**
- **Memory mapped region** for shared libraries
- **Runtime heap** (malloc)
- **Uninitialized data** (.bss)
- **Initialized data** (.data)
- **Program text** (.text)

- **User stack**
- **Process-specific data structs** (ptables, task and mm structs, kernel stack)
- **Physical memory**
- **Kernel virtual memory**

- **Different for each process**
- **Identical for each process**

Kernel virtual memory:
- Physical memory
- Kernel code and data
- Process-specific data structs (ptables, task and mm structs, kernel stack)

Process virtual memory:
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Process virtual memory:
- Physical memory
- Kernel code and data
- Process-specific data structs (ptables, task and mm structs, kernel stack)

- **Virtual Memory**
Address type in Linux

Linux is, of course, a virtual memory system, meaning that the addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Virtual memory introduces a layer of indirection, which allows a number of nice things. With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system’s physical memory. Virtual memory also allows playing a number of tricks with the process’s address space, including mapping in device memory.

Thus far, we have talked about virtual and physical addresses, but a number of the details have been glossed over. The Linux system deals with several types of addresses, each with its own semantics. Unfortunately, the kernel code is not always very clear on exactly which type of address is being used in each situation, so the programmer must be careful.

### Address Types

- **Kernel virtual addresses**
- **Kernel logical addresses**
- **High memory**
- **Low memory**
- **User process**
- **User process**
- **Kernel virtual addresses**
- **Kernel logical addresses**

The following is a list of address types used in Linux. Figure 13-1 shows how these address types relate to physical memory.

**User virtual addresses**
These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.
Linux Organizes VM as Collection of “Areas”

- **pgd:**
  - Page global directory address
  - Points to L1 page table

- **vm_prot:**
  - Read/write permissions for this area

- **vm_flags**
  - Pages shared with other processes or private to this process

**Diagram:**
- `task_struct` to `mm_struct`
- `mm` to `pgd`
- `mm_struct` to `vm_area_struct`
- `vm_end`, `vm_start`, `vm_prot`, `vm_flags` in `vm_area_struct`
- `Shared libraries`, `Data`, `Text` in `Process virtual memory`
- `vm_next`
i386+PAE

sources: http://linux-mm.org/PageTableStructure
x86_64

Page Table Layout

512 entries
4k object size

512 entries
4k object size

512 entries
4k object size

pte

4k Page

2M Page

sources: http://linux-mm.org/PageTableStructure
Powerpc-4k

PowerPC 64bit Page Table Layout With 4k Base Pages

sources: http://linux-mm.org/PageTableStructure
Powerpc-64k

sources: http://linux-mm.org/PageTableStructure
Simplifying Linking and Loading

Process virtual memory

- shared libraries
- data
- text

vm_area_struct

- vm_end
- vm_start
- vm_prot
- vm_flags
- vm_next

Segmentation fault:
accessing a non-existing page

1. read
2. write
3. read
Simplifying Linking and Loading

**Linux Page Fault Handling**

- **Segmentation fault:**
  - Accessing a non-existing page

- **Protection exception:**
  - e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

**Process virtual memory**

- `vm_area_struct`
  - `vm_end`
  - `vm_start`
  - `vm_prot`
  - `vm_flags`
  - `vm_next`

- **shared libraries**
- **data**
- **text**

**Actions**

1. **read**
2. **write**
3. **read**
Simplifying Linking and Loading

Process virtual memory

1. Segmentation fault:
   accessing a non-existing page

2. Protection exception:
   e.g., violating permission by writing to a read-only page
   (Linux reports as Segmentation fault)

3. Normal page fault
   reading data
Demand paging

- **Key point:** no virtual pages are copied into physical memory until they are referenced!
  - Known as **demand paging**
- Crucial for time and space efficiency
Shared Objects

Process 1 maps the shared object.

- Process 1 virtual memory
- Physical memory
- Process 2 virtual memory

Shared object
Shared Objects

- Process 1 maps the shared object.
- Notice how the virtual addresses can be different.
Private Copy-on-write (COW) Objects

- Two processes mapping a private copy-on-write (COW) object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only
Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!
Demand paging

- VM and memory mapping explain how fork provides private address space for each process.
- To create virtual address for new new process
  - Create exact copies of current mm_struct, vm_area_struct, and page tables.
  - Flag each page in both processes as read-only
  - Flag each vm_area_struct in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new pages using COW mechanism.
The execve Function

- To load and run a new program `a.out` in the current process using execve:
  - Free vm_area_struct’s and page tables for old areas
  - Create vm_area_struct’s and page tables for new areas
    - Programs and initialized data backed by object files.
    - `.bss` and stack backed by anonymous files.
  - Set PC to entry point in `.text`
    - Linux will fault in code and data pages as needed.
void *mmap(void *start, int len, int prot, int flags, int fd, int offset)

- Map len bytes starting at the offset of the file specified by file description fd, preferably at address start

  - **start**: may be 0 for “pick an address”
  - **prot**: PROT_READ, PROT_WRITE, ...
  - **flags**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be start)
User-Level Memory Mapping

void *mmap(void *start, int len, int prot, int flags, int fd, int offset)

Disk file specified by file descriptor fd

Process virtual memory

len bytes

start (or address chosen by kernel)

offset (bytes)
Memory management is the heart of operating systems; Each process in a multi-tasking OS runs in its virtual address space.
Memory management is the heart of operating systems; Each process in a multi-tasking OS runs in its virtual address space.

- **Kernel space**: User code CANNOT read from or write to these addresses, doing so results in a Segmentation Fault.
  - $0x00000000$ == TASK_SIZE
  - Random stack offset

- **Stack (grows down)**
  - RLIMIT_STACK (e.g., 8MB)
  - Random mmap offset

- **Memory Mapping Segment**: File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

- **Heap**
  - program break
  - brk
  - start_brk
  - Random brk offset

- **BSS segment**: Uninitialized static variables, filled with zeros. Example: static char *userName;

- **Data segment**: Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

- **Text segment (ELF)**: Stores the binary image of the process (e.g., /bin/gonzo)

Credit: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
How The Kernel Manages Process Memory

Credit: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory
Each address space consists of a number of page-aligned regions of memory that are in use. They never overlap and represent a set of addresses which contain pages that are related to each other in terms of protection and purpose. These regions are represented by a `struct vm_area` and are roughly analogous to the `vm_map` entry in BSD. For clarity, a region may represent the process heap for use with `malloc()`, a memory mapped file such as a shared library or a block of anonymous memory allocated with `mmap()`. The pages for this region may still have to be allocated, be active and resident or have been paged out.

If a region is backed by a file, its `vm_file` field will be set. By traversing `vm_file → f_dentry → d_inode → i_mapping`, the associated address space for the region may be obtained. The address space has all the filesystem specific information required to perform page-based operations on disk.

The relationship between the different address space related structures is illustrated in Figure 4.2. A number of system calls are provided which affect the address space and regions. These are listed in Table 4.1.

![Diagram of Relevant data structures in address space](image-url)
struct mm_struct {
    struct vm_area_struct *mmap;
    struct rb_root mm_rb;
    struct vm_area_struct *mmap_cache;
    long unsigned int (*get_unmapped_area)(struct file *, long unsigned int, long unsigned int, long unsigned int, long unsigned int);
    void (*unmap_area)(struct mm_struct *, long unsigned int);
    long unsigned int mmap_base;
    long unsigned int task_size;
    long unsigned int cached_hole_size;
    long unsigned int free_area_cache;
    pgd_t *pgd;
    atomic_t mm_users;
    atomic_t mm_count;
    int map_count;
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;
    struct list_head mmlist;
    mm_counter_t _file_rss;
    mm_counter_t _anon_rss;
    long unsigned int hiwater_rss;
    long unsigned int hiwater_vm;
}
struct mm_struct {
    [120] long unsigned int total_vm;
    [124] long unsigned int locked_vm;
    [128] long unsigned int shared_vm;
    [132] long unsigned int exec_vm;
    [136] long unsigned int stack_vm;
    [140] long unsigned int reserved_vm;
    [144] long unsigned int def_flags;
    [148] long unsigned int nr_ptes;
    [152] long unsigned int start_code;
    [156] long unsigned int end_code;
    [160] long unsigned int start_data;
    [164] long unsigned int end_data;
    [168] long unsigned int start_brk;
    [172] long unsigned int brk;
    [176] long unsigned int start_stack;
    [180] long unsigned int arg_start;
    [184] long unsigned int arg_end;
    [188] long unsigned int env_start;
    [192] long unsigned int env_end;
struct mm_struct {
    long unsigned int saved_auxv[44];
    unsigned int dumpable : 2;
    cpumask_t cpu_vm_mask
    mm_context_t context;
    long unsigned int swap_token_time;
    char recent_pagein;
    int core_waiters;
    struct completion *core_startup_done;
    struct completion core_done;
    rwlock_t ioctx_list_lock;
    struct kioctx *ioctx_list;
}
SIZE: 488
Figure 13-2. The three levels of Linux page tables

A page-aligned array of items, each of which is called a Page Table Entry. The kernel uses the \texttt{pte_t} type for the items. A \texttt{pte_t} contains the physical address of the data page.

The types introduced in this list are defined in \texttt{<asm/page.h>}, which must be included by every source file that plays with paging.

The kernel doesn't need to worry about doing page-table lookups during normal program execution, because they are done by the hardware. Nonetheless, the kernel must arrange things so that the hardware can do its work. It must build the page tables and look them up whenever the processor reports a page fault, that is,
vm_area_struct

Credit: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory
struct vm_area_struct {
    [0] struct mm_struct *vm_mm;
    [4] long unsigned int vm_start;
    [8] long unsigned int vm_end;
    [12] struct vm_area_struct *vm_next;
    [16] pgprot_t vm_page_prot;
    [20] long unsigned int vm_flags;
    [24] struct rb_node vm_rb;
    union {
        struct {...} vm_set;
        struct raw_prio_tree_node prio_tree_node;
    } shared;
    [36] } shared;
    [52] struct list_head anon_vma_node;
    [60] struct anon_vma *anon_vma;
    [64] struct vm_operations_struct *vm_ops;
    [68] long unsigned int vm_pgoff;
    [72] struct file *vm_file;
    [76] void *vm_private_data;
    [80] long unsigned int vm_truncate_count;
}
SIZE: 84
### Summary

**Figure 7.3. Relationship Between \textit{vmalloc()}, alloc page() and Page Faulting**

**7.3 Freeing a Noncontiguous Area**

The function \textit{vfree()} is responsible for freeing a virtual area as described in Table 7.2. It linearly searches the list of \textit{vm} structs looking for the desired region and then calls \textit{vmfree area pages()} on the region of memory to be freed, as shown in Figure 7.4.

```c
void vfree(void *addr)
```

Frees a region of memory allocated with \textit{vmalloc()}, \textit{vmalloc dma()} or \textit{vmalloc 32()}.

Table 7.2. **Noncontiguous Memory Free API**

\textit{vmfree area pages()} is the exact opposite of \textit{vmalloc area pages()}. It walks the page tables and frees up the page table entries and associated pages for the region.

**Diagram Description**

Process A Calls \textit{vmalloc()}

- Reserve Space in Reference Page Table
- After reserving space, allocate pages
- Buddy Allocator \textit{alloc_page()}

- Physically noncontiguous pages
- Virtually contiguous pages

Reference Page Table

- Inserted pages
  - in reference
  - page table

Userspace Portion

- Init_mm->pgd

Virtual pages

Process B page faults \textit{do_page_fault()}

- Fault is in \textit{vmalloc} region
- Copy in necessary page table entry from reference

Userspace Portion

- Copied entry

Process B Address Space managed by Process Page Tables

- Vmalloc_start
- Page_offset
do_page_fault
do_page_fault
Memory management is crucial
  - Machine introspection
  - Memory forensics
  - Traversing kernel data structures to understand the memory

Memory failures

Exploits
References

- Dening BEFORE MEMORY WAS VIRTUAL
  http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf
- Linux device drivers, Addison-wisely.
- Understanding Linux virtual memory manager
- Understanding Linux kernel (3rd edition)
  http://linux-mm.org/PageTableStructure