Rootkit Detection

CYB 5285

Maksym Hryhorenko

Kernel Data Structures

- Store necessary information that is used by OS Kernel to keep track of the state of the system as well as manage it.
- Always stored in physical memory.
- Always stored in kernel memory space. Protected only from user's apps.
- Can be accessed and modified by any kernel module, if it knows the address of a specific data structure.
- Usually stored in doubly linked list.
- Examples: tasks list (processes), sockets list, network ports list, allocated memory, hardware list, **loaded modules list.**

Module Data Structure

- All information about the module: name, size, exported functions, base address, references etc.
- Also stores doubly linked list pointers for previous list item and next.
- Module data structure object is created during module load, specifically, during memory allocation and module's file image mapping.
- Every module knows the address of its own module object. This address can be accessed in module's source code through *THIS_MODULE* preprocessor.
- Every module can read and write to its own module object. In addition, every module can traverse through the modules list with help of list pointers. Hence, module can read and write to any module's kernel object.

Module Data Structure

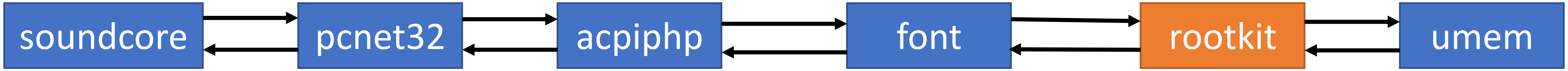
```
struct module
{
    enum module_state state;

    /* Member of list of modules */
    struct list_head list;

    /* Unique handle for this module */
    char name[MODULE_NAME_LEN];

    /* Sysfs stuff. */
    struct module_attribute *modinfo_attrs;
    const char *version;
    const char *srcversion;
    struct kobject *holders_dir;

    /* Startup function. */
    int (*init)(void);
    …
} // sizeof = 360
```

```
struct list_head {
        struct list_head *next;
        struct list_head *prev;
};
```

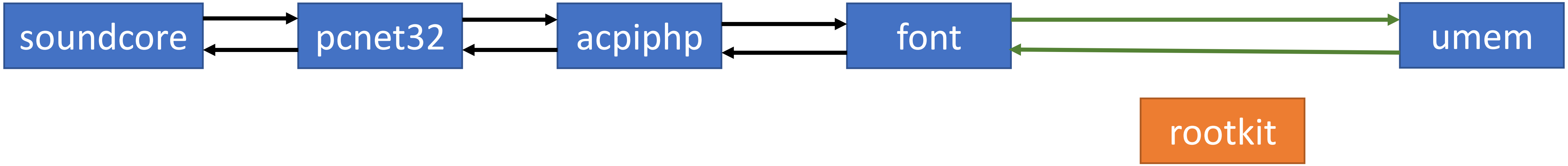
Loaded Kernel Modules List

- Stores information about every loaded kernel module.
- For every loaded module it's **`struct`** `module` object is inserted into the modules list. If module was unloaded, its object would be removed from the list.
- Helps Kernel to keep track of loaded modules' location (virtual address), name, state, size, references number and many other important stuff. Prevents multiple load of the same module. Makes module unload feasible.
- If module (pointer) was removed from the list and was not unloaded, Kernel would lose all the information about it. Kernel becomes unstable. Same module can be loaded again. Behavior is unpredictable.
- HOWEVER, module object still exists in the kernel memory, thus this module will function normally (until OS crashes).

Loaded Kernel Modules List
Rootkit Hiding

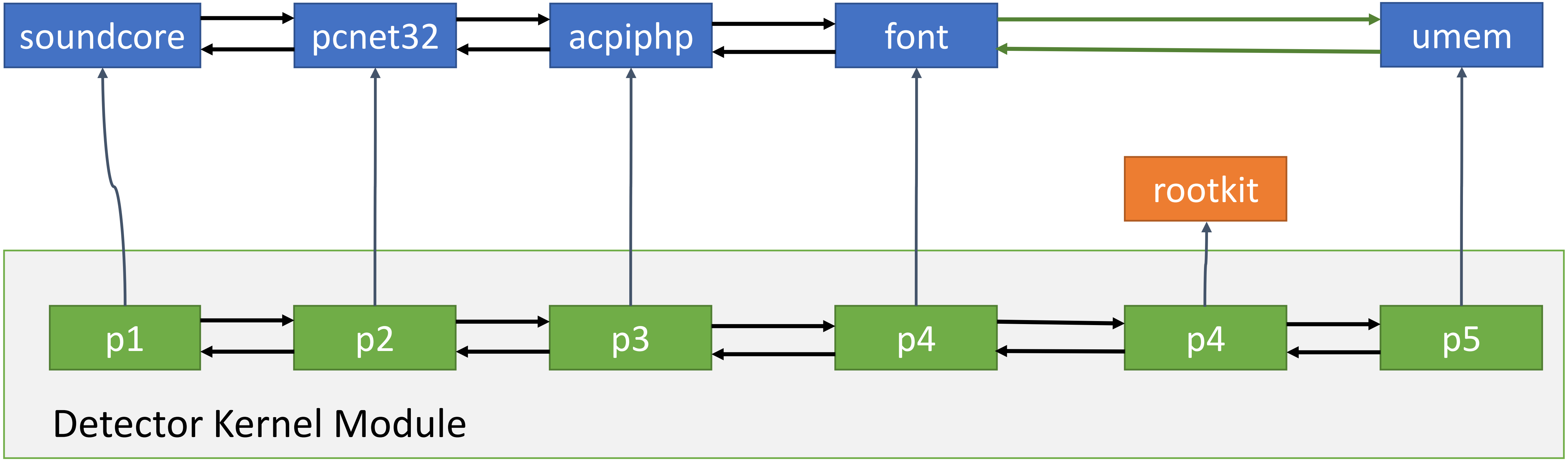


- Rootkit can manipulate list pointers in its own module `struct module` object.
- Simple doubly linked list item removal operation. However, the memory for the removed item is not deallocated, thus object still exists.



Detection

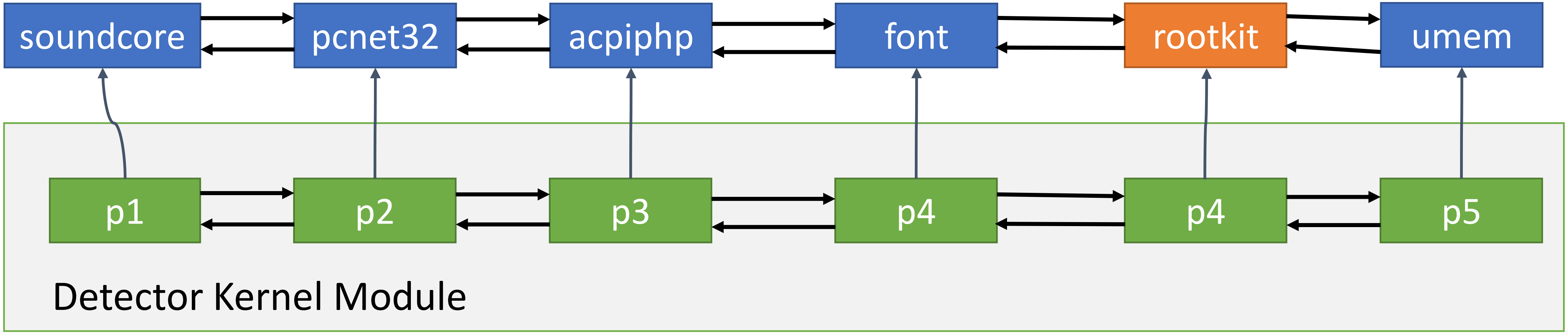
- Store object's address for every loaded module.
  Doubly linked list of pointers.



p - pointer

Detection

- Insert rootkit's module object back to the kernel modules list.



- Now rootkit is not hidden and can be detected by any standard tools: lsmod, modprobe and others.

Implementation

Problems:

- How to get the kernel object's address of loaded module?
- Only a single instance of kernel data structures are stored in memory, so it is not possible to get rootkit's module address from somewhere else except the modules list (but rootkit has deleted yourself from this list).
- The address is undefined before the memory allocation function was called for the module that is loading.

After memory for the module was allocated, its file image will be mapped. After image mapping kernel will build module's kernel object `struct module` and insert it into the modules list. Finally, OS will execute modules init function and waits till it finish. During its init function, rootkit can easily remove yourself from the modules list (hide).

Implementation

Solution:

- Intercept the load module syscall - `init_module().`
- But when? Intercept before – rootkit's module object still would be undefined.

  After – rootkit already removed yourself from the list.

So the only option is to intercept function during its execution (*mid-function hooking*).

Intercept **AFTER** the memory allocation and module's file image mapping and **BEFORE** module's init function execution.

Implementation

```
SYSCALL_DEFINE3(init_module, void __user *, umod,
          unsigned long, len, const char __user *, uargs)
{
      struct module *mod;
      int ret = 0;
      …
      /* Do all the hard work */
      mod = load_module(umod, len, uargs);
      …
      /* Start the module */
      if (mod->init != NULL)
            ret = do_one_initcall(mod->init);
      …
}
```

Pointer stores address of a module kernel object

Module file mapping and allocation. Module inserted into list in this function.

Good place for a function hook

Implementation

What to do after syscall interception:

- It is possible to traverse the loaded modules list and store the address of last loaded module, e.g. rootkit.

- However, there is a better solution. Extract value of the local variable
  ```
  struct module *mod;
  ```

- OR even better - extract the return value (register ***EAX)*** of the function call:
  ```
  load_module(umod, len, uargs);
  ```

Implementation

```
SYSCALL_DEFINE3(…)
{
      struct module *mod;
      int ret = 0;

      /* Do all the hard work */
      mod = load_module(umod, len, uargs);
----> goto DKM;
      BACK:
      …
      /* Start the module */
      if (mod->init != NULL)
            ret = do_one_initcall(mod->init);
      …
}
```

Somewhere in the detector module:

```
-----------------------
struct module *g_mod;

DKM:
  g_mod = EAX;
  goto BACK;
-----------------------
```

g_mod - global variable.

Now I can just copy g_mod value to another pointer and insert this pointer to the list

Implementation

Detector creates 2 hooks

1. Intercept syscall before its execution (hook through syscall table)
2. Intercept syscall during its execution.

The first hook is needed:

- To prepare g_mod variable (g_mod = null)
- Examine the return value of **`init_module().`** If module load was not successful just return.

Implementation

```c
asmlinkage long hooked_init_module(void __user *umod, unsigned long len, const char *uargs)
{
        int ret = 0;
        struct dkm_module *mod; // has pointer to struct module* and list pointers
        // Prepare g_mod
        g_mod = NULL;
        // Call original init_module
        ret = orig_init_module(umod, len, uargs); // g_mod gets value during execution

        if (!ret && g_mod) {
                mod = kmalloc(sizeof(struct dkm_module), GFP_KERNEL);
                if (mod) {
                        strncpy(mod->name, g_mod->name, MODULE_NAME_LEN);
                        mod->mod_ptr = g_mod;
                        list_add(&mod->list, &g_modlist);
                }
                else { pr_err("Failed to allocate memory for new mod\n");
                }
        }
        return ret; }
```

Implementation

```
struct dkm_module {
    char name[MODULE_NAME_LEN];
    struct module *mod_ptr;
    struct list_head list;
};
```

Implementation

```
MEMORY:C0183353 E8 98 CC 40 00          call    near ptr mutex_lock_interruptible
MEMORY:C0183358 85 C0                   test    eax, eax
MEMORY:C018335A 75 D8                   jnz     short loc_C0183334
MEMORY:C018335C 8B 4D 10                mov     ecx, [ebp+arg_8]
MEMORY:C018335F 8B 55 0C                mov     edx, [ebp+arg_4]
MEMORY:C0183362 8B 45 08                mov     eax, [ebp+arg_0]
MEMORY:C0183365 E8 36 F1 FF FF          call    near ptr load_module
MEMORY:C018336A 3D 00 F0 FF FF          cmp     eax, 0FFFFF000h
MEMORY:C018336F 89 C6                   mov     esi, eax
MEMORY:C0183371 0F 87 06 01 00 00       ja      loc_C018347D
```

AFTER HOOK

```
MEMORY:C0183353 E8 98 CC 40 00          call    near ptr mutex_lock_interruptible
MEMORY:C0183358 85 C0                   test    eax, eax
MEMORY:C018335A 75 D8                   jnz     short loc_C0183334
MEMORY:C018335C 8B 4D 10                mov     ecx, [ebp+arg_8]
MEMORY:C018335F 8B 55 0C                mov     edx, [ebp+arg_4]
MEMORY:C0183362 8B 45 08                mov     eax, [ebp+arg_0]
MEMORY:C0183365 E8 36 F1 FF FF          call    near ptr load_module
MEMORY:C018336A 68 00 B0 29 F8          push    0F829B000h
MEMORY:C018336F C3                      retn
MEMORY:C0183370                         ; ---------------------------------------
MEMORY:C0183370 90                      nop
```

Implementation

Hook handler for mid-func

```
MEMORY:F829B000                                        ; START OF FUNCTION CHUNK FOR sys_init_module
MEMORY:F829B000
MEMORY:F829B000                                        loc_F829B000:                        ; CODE
MEMORY:F829B000 A3 3C 53 29 F8                         mov     dword_F829533C, eax
MEMORY:F829B005 3D 00 F0 FF FF                         cmp     eax, 0FFFFF000h
MEMORY:F829B00A 89 C6                                  mov     esi, eax
MEMORY:F829B00C 68 71 33 18 C0                         push    offset loc_C0183371
MEMORY:F829B011 C3                                     retn
MEMORY:F829B011                                        ; END OF FUNCTION CHUNK FOR sys_init_module
MEMORY:F829B011                                        ; ---------------------------------------
```

Disabling Rootkit Functions

Besides hiding yourself, rootkit can also hide **user's space processes** and **files.**

Usually implemented by hooking certain **system calls,** e.g. `open(), read(), write().` Rootkit checks whether the target of a system call is **ID** of the process or **NAME** of the file that should be hidden. If yes, rootkit does not call the original system call and just returns error (common scenario).

To hide files, it is also possible to hook **file_operations**.

Disabling Rootkit Functions

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    …
};
```

Disabling Rootkit Functions

Solution: **create snapshots.**

- Copy pointers of original system calls and original file_operations.
- Copy first bytes (instructions) of system calls and file_operations.
- Verify system calls and file_operations using valid pointers and function bytes

Disabling Rootkit Functions

```
struct fops_snapshot {
    unsigned char open[SNAPSHOT_SIZE];
    unsigned char readdir[SNAPSHOT_SIZE];
    unsigned char read[SNAPSHOT_SIZE];
    unsigned char write[SNAPSHOT_SIZE];
    const char *fpath; // absolute file path
};

struct syscallt_snapshot {
    char op[SNAPSHOT_SIZE]; // op codes
    unsigned int call_id; // sys call number
    unsigned long addr; // addr of original syscall
};
```

Demonstration