End to End Defense against Rootkits in Cloud Environment

Background

Sachin Shetty

Associate Professor
Electrical and Computer Engineering
Director, Cybersecurity Laboratory
Tennessee State University

Background

- Background on CPU and Linux kernel necessary to understand our detection system.
- x86 memory protection mechanisms followed by Linux kernel memory layout.
- Relocation work when kernel modules are loaded.

Background: Page-based Memory Protection

- Page-based protection divides the virtual address space of CPU into pages of fixed size.
- Memory Management Unit(MMU) translate the virtual address into physical address through a data structure called page directory.
- Page table entries in the page directory describe the access permissions of each page, including read/write and execute permissions.
- Particularly, execute permission indicated by NX-bit is available only when the PAE(Physical Address Extensions) of x86 32 mode or x86 64 mode is used.

Background: Page-based Memory Protection

- When execute permissions of page table entries are available, a page can be executed if and only if the NX-bit of its page table entry is cleared.
- Pages containing code/instructions are marked executable(NX-bit of the page table entry is cleared).
- Linux kernels which support PAE take advantage of this feature, and only mark pages of kernel code and modules executable to guarantee normal execution flows.
- Under this condition, when a rootkit is installed into the kernel space, the pages in which its code resides must also be marked executable

Background- Kernel Memory Layout

- In Linux system, all of the process share the same kernel address space.
- The kernel address space is also divided into pages, thus a page directory is used to describe the kernel address space.
- Page directory is the only one describing the kernel address space in the normal.
- In the kernel address space, some regions are mapped to the physical memory, including region of kernel code, region of kernel static data and regions allocating memory for dynamic kernel data..

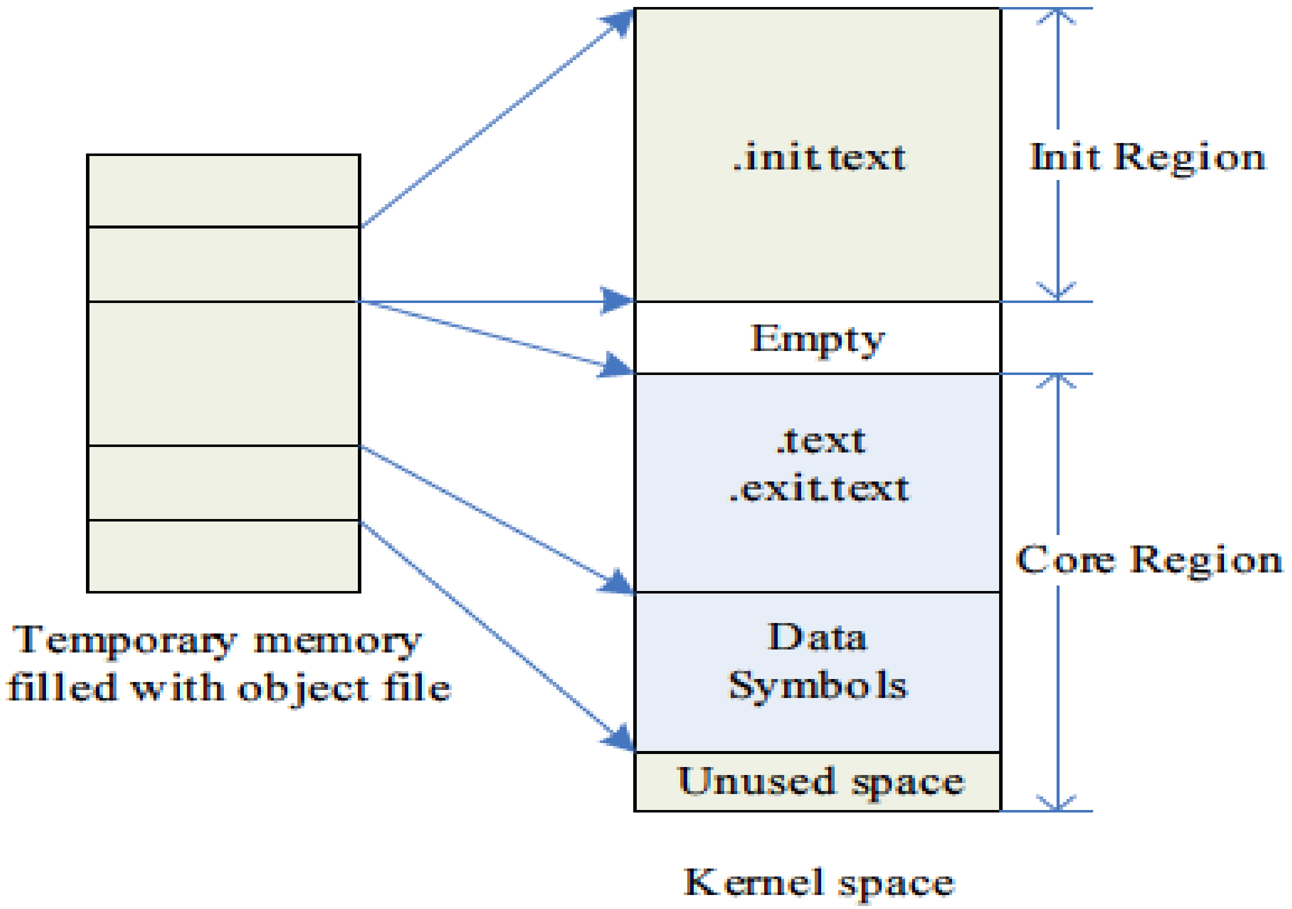
Background- Kernel Memory Layout

- Other regions that are not mapped are reserved for kernel modules and other usage.
- Kernel code consists of many functions, some of which are exported.
- Exported functions can be used by kernel modules through their names, which are called symbols.
- When the kernel is compiled and linked, the virtual addresses of the symbols(exported or not) of the kernel code and static data are determined and stored in the System:map file

Background- Kernel Memory Layout

- Regions of kernel code and modules must be marked executable in order to execute
- Kernel marks the other regions as non-executable so that GP exception will be raised if CPU try to get instructions from those nonexecutable regions.
- When a kernel-level rootkit is installed, its code may reside in the regions of kernel code, or modules
- Its code may also reside in the reserved regions or the regions for dynamic kernel data.
  - In this case, it has to clear NX-bits of the related page table entries in the page directory.
  - Creates a new executable region for its code.

Background: Loading Kernel Modules



Background: Loading Kernel Modules

- 1. Loads the relocatable file of the module from file system into temporary memory.
- 2. Checks the format of this module file, stores the parameters of this module, and makes sure that this module is not already loaded.
- 3. Allocates a page-aligned memory region for the initialization code of the module, and fills it with the corresponding content of the module. We call it init region.
- 4. Allocates a page-aligned memory region for the core executable code of the module, and fills it with the corresponding content of the module. We call it core region.

Background: Loading Kernel Modules

- 5. Relocates the module’s object code, using the kernel symbols table and module symbols table.
- 6. Processes the exported symbols of this module.
- 7. Frees the temporary memory allocated in step 1.
- 8. Executes the initialization code of this module.
- 9. Frees the init region allocated for the initialization code of this module.

Background: Loading Kernel Modules

- After a module is loaded, only its core region resides in the kernel space and this region is page-aligned.
- However, the size of its core executable code is unlikely to be page-aligned.
- Therefore, a small space at the last of this region is never used by the module or kernel itself. We name this small space as unused space.
- The size of unused space may vary between 0 and the page size.
- Each module loaded into the kernel space likely contains an executable unused space which may be exploited by the kernel-level rootkits.