ISSN:0975-9646

Linux Kernel Memory Protection (ARM)

Manjeet Singh, Vaneet

Birla Institute of Technology and Science
University in Pilani, Rajasthan, India

***Abstract*— Memory protection is a way to control memory access rights on an embedded system. The main purpose of memory protection is to prevent a task from accessing memory without proper access permissions. Without memory Protection memory segment like code and data segment are vulnerable to memory related bugs and code injection attacks. Memory related bugs arise due to corruption from one task to another that grow in complexity over time until it crashes the entire system. Such corruptions are hard to debug. Furthermore code injection attacks continue to be one of the major ways of computer break-ins and malware propagation by injecting malicious code. For example: a. Kernel code section has been corrupted by some task and later the same section accessed by another task resulting in system crash or system may display some undesired behaviour. In aforementioned scenario it's very difficult to catch the culprit task. B. a hacker can inject malicious code into a running process having modified the text section and transfer execution to the injected code resulting in system's wreckage.**

***Keywords*— Memory, Kernel, uImage, sections**

Introduction

Memory protection is required to intercept corruption at earlier stage which would be helpful in debugging and also for security reasons for not allowing malicious code to get executed.

Memory protection can be provided if system forbids code section and read only data to get written at run time along with barred execution from data section. Above mentioned requirement can be summarized as:

- Kernel code section should always be executable and read only.
- Kernel read only data (RO) should be read only and not executable (NE).
- Kernel data should always be not executable (NE) but should have read write access (RW).

To achieve above requirements is also the main purpose of this document.

I. HOW IT WORKS

In this paper, a mechanism is explained how to provide kernel code and data protection. To test this feature a kernel code is written which tries to modify the behaviour by changing the function (created by module) address. Fig.1. displays an example when a code section corrupted by a task which latterly accessed by another task resulted in crash.

A. Protection on Kernel module function

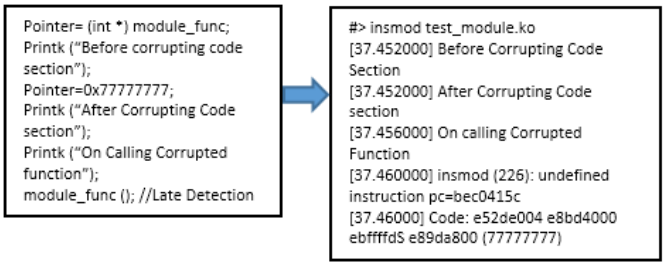

Fig.1. Prototype code (existing scenario)

Before producing this scenario (memory protected), the kernel sections are protected by applying permission over them. Since kernel divides physical memory into pages and sections. Sections are updated with permissions like kernel text will be made as RX, kernel data and stack will be made as RW. The trade-off is that each region is padded into section-size (1MiB) boundaries.

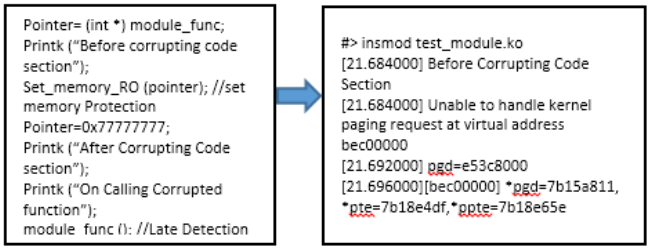

Fig.2.kernel module function protection

B. Protection on Kernel own function

Here kernel functions are used to check sections protections.

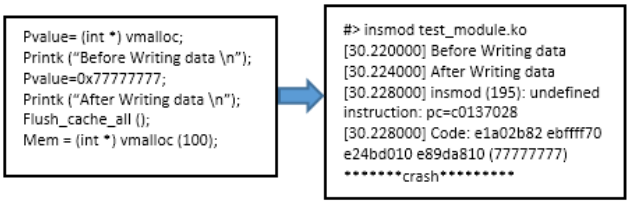

Fig.3. Existing scenario

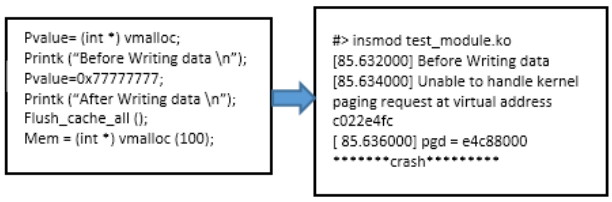

Fig.4. kernel function protection

II. IMPLEMENTATION

ARMV7 supports variants of memory system architectures.

- ARMV7, a profile requires inclusion of Virtual Memory System Architecture (VMSA). This profile uses MMU for memory translation and protection.
- ARMV7, R profile requires inclusion of Protected Memory System Architecture (PMSA). This is for real time system that does not include MMU but provides memory protection using MPU.

Aforementioned memory system architectures provide mechanisms to split memory into different regions. Each region can be configured with specific memory types and attributes.

Linux kernel requires MMU for virtual address translation. VMSA (ARMv7-A) will be considered for memory protection in kernel. In VMSA MMU controls address translation, access permissions, and memory attribute determination and checking for memory.

For instruction and data access, the possible settings are:

- No access
- Read-only
- Write-only
- Read/Write

For instruction accesses, additional controls determine whether instructions can be fetched and executed from the memory region. If a processor attempts an access that is not permitted, a memory fault is signalled to the processor.

A. Address Translation

From virtual to physical address translation, MMU performs Translation table walks. First level table holds descriptor containing the base address and translation properties for a section. In case of pages it contains a pointer to a second level table.

Second level table holds descriptor containing the base address and translation properties for small or large pages.

Memory protection can be done by controlling access permission and execute permission of a memory.

A.1 AP, Access permissions

To control Read/Write access to any memory, AP bits needs to be controlled. For sections AP bits are present in first level descriptor and for pages AP are available in second level descriptor.

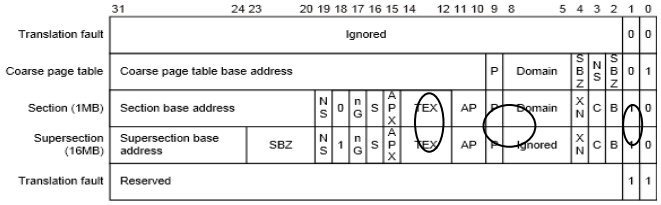

Fig.5. First level Descriptor

A.2 XN, Execute-Never

When the XN bit is 1, a Permission fault is generated if processor attempts to execute an instruction fetched from the corresponding memory region.

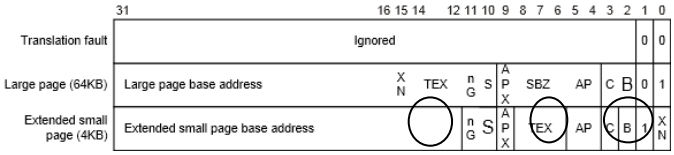

Fig.6. Second level Descriptor

B. How to protect Linux kernel code/data

The focus of this paper is to protect below three categories of kernel code.

B.1 Kernel static code and data (which is loaded as a part of uImage)

uImage are loaded in the form of sections and kernel modules are loaded as pages on arm. As explained above, to protect uImage we need to protect sections by configuring AP and NE bits of the first level descriptor. Kernel memory is divided into sections. [Text+RO data], [data section]. Both text [executable] and RO [not executable] section has different permission so it is required to preserve both indifferent sections. This may result in some memory wastage.

Without memory protection: - Section alignment by PAGE_SIZE
.text: 0xc0008000 – 0xc0518ddc (5188 kB)
.init: 0xc0519000 – 0xc0550040 (221 kB)
.data: 0xc0552000 – 0xc058a7a0 (226 kB)
.bss: 0xc058a7a0 – 0xc06fca70 (1481 kB)

After applying protections and modifying section alignment to kernel sections.

Memory Protection: - Section Alignment to 1MB
.text: 0xc0008000 – 0xc05c6dfc (5884 kB)
.init: 0xc0600000 – 0xc0633040 (205 kB)
.data: 0xc0634000 – 0xc0669a60 (215 kB)
.bss: 0xc0669a60 – 0xc078abd0 (1157 kB)

Since kernel sections are aligned by 1MB boundary. Minimizing size may lead to tlb performance issues. So it's not recommended. Above figure is observed after aligning sections to 1MB boundary.

B.2 Dynamic kernel modules

Loadable modules are loaded in the form of pages. To protect them we need to configure AP and NE bits of second level descriptor.

B.3 Init data in case of both static and dynamic code/data

There are certain set of data which are initialized once and then always treated as RO. We cannot protect such data from corruption as these are not in RO section. To protect such data we can declare them as constants so that loader can load them into RO section. Before initializing we need to disable the protection and after initialization we can enable it back.

III. CONCLUSION

As displayed above, having configured page table entries memory protection can be provided.

ACKNOWLEDGMENT

We take this opportunity to thanks and regards to our mentor Rakesh Kumar (Asst. Professor at BITS Pilani) for his exemplary guidance, monitoring and constant encouragement throughout this activity.

REFERENCES

[1] http://lwn.net/Articles/531148/
[2] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html
[3] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI.html
[4] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Caceaije.html