

Eliminating the Threat of Kernel Stack Overflows

Yair Wiseman
*Computer Science
Department
The Open University
Israel*
wiseman@cs.huji.ac.il

Joel Isaacson
*Ascender Technologies
Israel*
joel@ascender.com

Eliad Lubovsky
*Computer Science
Department
Bar-Ilan University
Israel*
eliadl@013.net

Abstract

The Linux kernel stack has a fixed size. There is no mechanism to prevent the kernel from overflowing the stack. Hackers can exploit this bug to put unwanted information in the memory of the operating system and gain control over the system. In order to prevent this problem, we introduce a dynamically sized kernel stack that can be integrated into the standard Linux kernel. The well-known paging mechanism is reused with some changes, in order to enable the kernel stack to grow.

1. Introduction

The management of virtual memory and the relationship of software and hardware to this management is an old research subject [1]. In this paper we would like to focus on the kernel mode stack. Our discussion will deal with the Linux operating system running on an IA-32 architecture machine. However, the proposed solutions may be relevant for other platforms and operating systems as well.

The memory management architecture of IA-32 machines uses a combination of segmentation (memory areas) and paging to support a protected multitasking environment [2]. The x86 enforces the use of segmentation which provides a mechanism of isolating individual code, data and stack modules.

Therefore, Linux splits the memory address space of a user process into multiple segments and assigns a different protection mode for each of them. Each segment contains a logical portion of a process, e.g. the code of the process. Linux uses the paging mechanism to implement a conventional demand-paged, virtual-memory system and to isolate the memory spaces of user processes [3].

Paging is a technique of mapping small fixed size regions of a process address space into chunks of real, physical memory called page frames. The

size of the page is constant, e.g. IA-32 machines use 4KB of physical memory.

In point of fact, IA-32 machine support also large pages of 4MB. Linux (and Windows) do not use this ability of large pages (also called super-pages) and actually the 4KB page support fulfills the needs for the implementation of Linux [4].

Linux enables each process to have its own virtual address space. It defines the range of addresses within this space that the process is allowed to use. The addresses are segmented into isolated section of code, data and stack modules.

Linux provides processes a mechanism for requesting, accessing and freeing memory [5,6]. Allocations are made to contiguous, virtual addresses by arranging the page table to map physical pages. Processes, through the kernel, can dynamically add and remove memory areas to its address space. Memory areas have attributes such as the start address in the virtual address space, length and access rights. User threads share the process memory areas of the process that has spawned them; therefore, threads are regular processes that share certain resources. The Linux facility known as “kernel threads” are scheduled as user processes but lack any per-process memory space and can only access global kernel memory.

Unlike user mode execution, kernel mode does not have a process address space. If a process executes a system call, kernel mode will be invoked and the memory space of the caller remains valid. Linux gives the kernel a virtual address range of 3GB to 4GB, whereas the processes use the virtual address range of 0 to 3GB. Therefore, there will be no conflict between the virtual addresses of the kernel and the virtual addresses of whichever process.

In addition, a globally defined kernel address space becomes accessible which is not process

unique but is global to all processes running in kernel mode. If kernel mode has been entered not via a system call but rather via a hardware interrupt, a process address space is defined but it is irrelevant to the current kernel execution.

2. Stack Allocations

2.1. Fixed Size Allocations

User space allocations are transparent with a large and dynamically growing stack. In the Linux kernel's environment the stack is small-sized and fixed. It is possible to determine the stack size as from 2.6.x kernel series during compile time choosing between 4 to 8KB. The current tendency is to limit the stack to 4KB.

The allocation of one page is done as one non-swappable base-page of 4KB. If a 8KB stack is used, two non-swappable pages will be allocated, even if the hardware support an 8KB super-page [7]; in point of fact, IA-32 machines do not support 8KB super-pages, so 8KB is the only choice.

The rational for this choice is to limit the amount of memory and virtual memory address space that is allocated in order to support a large number of user processes. Allocating an 8KB stack increases the amount of memory by a factor of two. In addition the memory must be allocated as two contiguous pages which are relatively expensive to allocate.

A process that executes in kernel mode, i.e. executing a system call, will use its own kernel stack. The entire call chain of a process executing inside the kernel must be capable of fitting on the stack. In an 8KB stack size configuration, interrupt handlers use the stack of the process they interrupt. This means that the kernel stack size might need to be shared by a deep call chain of multiple functions and an interrupt handler. In a 4KB stack size configuration, interrupts have a separate stack, making the exception mechanism slower and more complicated [8].

The strict size of the stack may cause an overflow. Any system call must be aware of the stack size. If large stack variables are declared and/or too many function calls are made, an overflow may occur.

Memory corruption caused by a stack overflow may cause the system to be in an undefined state.

The kernel makes no effort to manage the stack and no essential mechanism oversees the stack size.

In [9] the authors present an empirical study of Linux bugs. The study compares errors in different subsections of Linux kernels, discovers how bugs are distributed and generated, calculates how long, on average, bugs live, clusters bugs according to error types, and compares the Linux kernel bugs to the OpenBSD kernel bugs. The data used in this study was collected from snapshots of the Linux kernel across seven years. The study refers to the versions until the 2.4.1 kernel series, as it was published in 2001. 1025 bugs were reported in this study. The reason for 102 of these bugs is large stack variables on the fixed-size kernel stack. Most of the fixed-size stack overflow bugs are located in device drivers. Device drivers are written by many developers who may understand the device more than the kernel, but are not aware of the kernel stack limitation. Hence, no attempt is made to confront this setback. In addition, only a few users may have a given device; thus, only a minimal check might be made for some device drivers. In addition, Cut-and-Paste bugs are very common in device drivers and elsewhere [10]; therefore, the stack overflow bugs are incessantly and unwarily spread.

The goal of malicious attackers is to drive the system into an unexpected state, which can help the attacker to infiltrate into the protected portion of the operating system. Overflowing the kernel stack can provide the attacker this option which can have very devastating security implications [11]. The attackers look for rare failure cases that almost never happen in normal system operations. It is hard to track down all the rare cases of kernel stack overflow, thus the operating system remains vulnerable. This leads us to the unavoidable conclusion: Since the stack overflows are difficult to detect and fix, the necessary solution is letting the kernel stack grow dynamically.

A small fixed size stack is a liability when trying to port code from other systems to Linux. The kernel thread capability would seem offer an ideal platform for porting user code and non-Linux OS code. This facility is limited both by the lack of a per-process memory space and by a small fixed sized size stack.

An example of the inadequacy of the fixed size stack is in the very popular use of the Ndiswrapper project [12] to implement Windows kernel API and NDIS (Network Driver Interface Specification) API within the Linux kernel. This can allow the use of a Windows binary driver for a wireless network card

running natively within the Linux kernel, without binary emulation. This is frequently the solution used when hardware manufacturers refuse to release detail of their product so a native Linux driver is not available.

The problem with this approach is that the Windows kernel provides a minimum of 12KB kernel stack whereas Linux in the best case uses an 8KB stack. This mismatch of kernel stack sizes can and cause system stack corruptions leading to kernel crashes. This would ironically seem to be the ultimate revenge of an OS (MS Windows) not known for long term reliability on an OS (Linux) which normally is known for its long term stability.

2.2. Dynamic Size Allocations

In the 1980s, a new operating system concept was introduced: the *microkernels* [13,14]. The objective of microkernels was to minimize the kernel code and to implement anything possible outside the kernel. This concept is still alive and embraced by some operating systems researchers [15], although the classic operating systems like Linux still employ the traditional monolithic kernel.

The microkernels concept has two main advantages: First, the system is flexible and extensible, i.e. the operating system can easily adapt a new hardware. Second, many malfunctions are isolated like in a regular application; because many parts of the operating system are standard processes and thus are independent. A permanent failure of a standard process does not induce a reboot; therefore, the microkernel based operating systems tend to be more robust [16].

A microkernel feature that is worthy of note is the address space memory management [17]. A dedicated process is in charge of the memory space allocation, reallocations and free. The process is executed in user mode; thus, the page faults are forwarded and handled in user mode and cannot cause a kernel bug. Moreover, most of the kernel services are implemented outside the kernel and specifically the device drivers; hence these services are executed in user mode and are not able to use the kernel stack.

Although the microkernel has many theoretical advantages [18], its performance and efficiency are somewhat disappointing. Nowadays, most of the modern operating systems use a monolithic kernel. In addition, even when an operating system uses a

microkernel scheme, there still will be minimal use of the kernel stack.

We propose an approach that suggests a dynamically growing stack. However, unlike the microkernel approach, we will implement the dynamically growing stack within the kernel.

2.3. Real Time Considerations

Linux is designed as a non-preemptive kernel. Therefore, by its nature, is not well suited for real time applications that require deterministic response time.

The 2.4.x Linux kernel versions introduced several new amendments. One of them was the preemptive patch which supports soft real-time applications [19]. This patch is now a standard in the new Linux kernel versions [20]. The objective of this patch is executing the scheduler more often by finding places in the kernel code that preemptions can be executed safely. On such cases more data is pushed onto the kernel stack. This additional data can worsen the kernel overflow problem. In addition, these cases are hard to be predicted [21].

For hard real-time applications, RTLinux [22] or RTAI [23] can be used. These systems use a nano-kernel that runs Linux as its lowest priority execution thread. This thread is fully preemptive hence real-time tasks are never delayed by non-real-time operations.

Another interesting solution for a high-speed kernel-programming environment is the KML (Kernel Mode Linux) project [24,25,26]. KML allows executing user programs in kernel mode and a direct access to the kernel address space. The kernel mode execution eliminates the system call overhead, because every system call is merely a function call. The main disadvantage of KML is that any user can write to the kernel memory. In order to trim down the aforementioned problem, the author of KML suggests using TAL (Typed Assembly Language) which checks the program before loading. However, this check does not always find the memory leak. As a result, the security is very poor. It is difficult to prevent illegal memory access and illegal code execution. On occasion, memory illegal accesses are done deliberately, but they also can be performed accidentally.

Our approach to increase the soft real-time applications responsiveness is to run them as kernel threads while using fundamental normal process

facilities such as a large and dynamically growing stack. While running in kernel context, it is possible to achieve a better predictive response time as the kernel is the highest priority component in the system. The solution provides the most important benefits you find in the KML project, although this solution is a more intuitive and straightforward implementation.

3. Implementation

The objective of this implementation is to support the demand paging mechanism for the kernel mode stack. The proposed solution is a patch for the kernel that can be enabled or disabled using the kernel configuration tools. In the following sections the design, implementation and testing utilities are described.

3.1. Process Descriptor

In order to manage its processes, Linux has for each process a *process descriptor* containing the information related to this process [27]. Linux stores the process descriptors in a circular doubly linked list called the *task list*. The process descriptor's pointer is a part of a structure named "thread_info" that is stored under the bottom of the kernel mode stack of each process as shown in Figure 1.

This feature allows referencing the process descriptor using the stack pointer without any memory referencing. The reason for this method of implementation is improved performance. The stack pointer address is frequently used; hence, it is stored in a special purpose register. In order to get a reference for the current process descriptor faster, the stack pointer is used. This is done by a macro called "current".

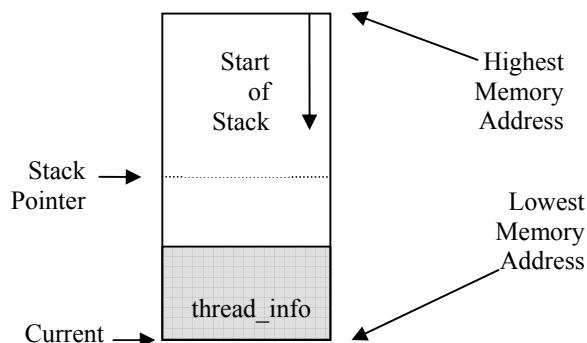


Fig. 1 Kernel Memory Stack and the Process Descriptor

In order to benefit the performance and leave the "current" mechanism untouched, a new allocation interface is introduced which allocates one physical page and a contiguous virtual address space that is aligned to the new stack size.

The new virtual area of the stack size can be of any size. The *thread_info* structure is set to the top of the highest virtual address minus the *thread_info* structure size. The stack pointer starts from beneath the *thread_info*. Additional physical pages will be allocated and populated in the virtual address space if the CPU triggers a page fault exception.

3.2. Exceptions

The IA-32 architecture provides 4 protection levels of code execution. Usually they are called "rings" and numbered as 0,1,2,3 whereas 0 is the most privileged ring and 3 is the least privileged. Linux uses just ring 0 and 3. Ring 0 is used when the kernel is executed, whereas 3 is used for non-privileged user space applications.

When a process is executed and an exception occurs, the ring is switched from 3 to 0. One of the consequences of this switch is changing of the stack. The process' user space stack is replaced by the process' kernel mode stack while the CPU pushes several registers to the new stack. When the execution is completed, the CPU restores the interrupted process user space stack using the registers it pushed to the kernel stack.

If an exception occurs during a kernel execution in the kernel mode stack, the stack is not replaced because the task is already running in ring 0. The CPU cannot push the registers to the kernel mode stack, thus it generates a double fault exception. This is called the stack starvation problem.

3.3. Interrupt Task

Interrupts divert the processor to code outside the normal flow of control. The CPU stops what it is currently doing and switches to a new activity. This activity is usually held in the context of the process that is currently running, i.e. the interrupted process. As mentioned, current scheme may lead to a stack starvation problem if a page fault exception happens in the kernel mode stack.

The IA-32 provides a special task management facility to support process management in the kernel.

Using this facility while running in the kernel mode causes the CPU to switch an execution context to a special context, therefore preventing the stack starvation problem.

The current Linux kernel release uses this kind of mechanism to handle double fault exceptions that are non-recoverable exceptions in the kernel. This mechanism uses a system segment called a Task State Segment that is referenced via the IDT (Interrupt Descriptor Table) and the GDT (Global Descriptor Table) tables. This mechanism provides a protected way to manage processes although it is not widely used because of a relatively larger context switch time.

We suggest adding the special task management facility to handle page fault exceptions in the kernel mode stack. Using this mechanism it is possible to handle the exceptions by allocating a new physical page, mapping it to the kernel page tables and resuming the interrupted process. Current user space page faults handling will remains as is.

4. Evaluation

We performed a functionality test to check that when the CPU triggers a page fault in the kernel mode stack, a new page is actually allocated and mapped to the kernel page tables.

This feature was accomplished by writing a kernel module and intentionally overloading the stack by a large vector variable. We then added printing to the page fault handler and were able to assess that the new mechanism worked as expected.

It has to be noted that only page faults that are in the kernel mode stack are handled using the task management facility, whereas page faults triggered from user space processes are handled as in the original kernel.

Triggering of page faults from the user processes stack and even more so from the kernel mode stack rarely happens. In both scenarios performance decrement in the system is negligible.

In spite of the aforementioned, we obtained several measurements to ensure that the new mechanism does not demonstrate anomalous results.

5. Conclusions

An overflow in kernel stack is a common bug in the Linux operating system. These bugs are difficult to detect because they are created as a side effect of the code and not as an inherent mistake in the algorithm implementation.

This paper shows how the size of the kernel stack can dynamically grow using the common mechanism of page faults giving a number of advantages:

1.) Stack pages are allocated on demand. If a kernel process needs minimal stack only one page is allocated. Only kernel processes that need larger stacks will have more pages allocated.

2.) The stack pages allocated per kernel process need not be contiguous but rather non-contiguous physical pages are mapped contiguously by the MMU.

3.) Stack overflows can be caught and damage to other kernel process stacks prevented.

4.) Larger kernel stacks can be efficiently provided. This facilitates porting of code that has not been designed for minimal stack usage into the Linux kernel.

References

- [1] P. Denning, "Virtual Memory", ACM Computing Surveys (CSUR), Vol. 2(3), ACM Press, New York, NY, USA, pp. 153-189, 1970.
- [2] "Intel Pentium Processor User's Manual", Intel Corporation, Mt. Prospect IL, 1993.
- [3] "IA-32 Intel Architecture Software Developer's Manual", Volume3: System Programming Guide, September 2005
- [4] S. J. Winwood, Y. Shuf, and H. Franke. Multiple page size support in the Linux kernel. roceedings of Ottawa Linux Symposium, Ottawa, Canada, June 2002.
- [5] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel". O'reilly; 2nd Edition, 2003.

- [6] R. Love, "Linux Kernel Development", Sams; 1st edition, September 8, 2003.
- [7] Itshak M. & Wiseman Y., AMSQM: Adaptive Multiple SuperPage Queue Management, Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008), Las Vegas, Nevada, 2008.
- [8] A. Robbins, "Linux Programming by Example", Pearson Education Inc., 2004.
- [9] A. Chou, J.F. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors", In Proceedings of the 18th ACM, Symposium on Operating System Principles (SOSP), pp. 73-88, Lake Louise, Alta. Canada, October 2001.
- [10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code", The 6th Symposium on Operating Systems Design and Implementation (OSDI '04), San Francisco, California, December 6-8, 2004.
- [11] "Analysis of the Linux kernel", Coverity Corporation, 2004.
- [12] P Fuchs, G Pemmasani, "NdisWrapper", <http://ndiswrapper.sourceforge.net/>, 2005.
- [13] J. Liedtke, "Toward Real Microkernels", Communications of the ACM, Vol. 39(9), September 1996.
- [14] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage and E. Gun Sirer, "SPIN - An Extensible Microkernel for Application-specific Operating System Services", ACM Operating Systems Review, Vol. 29(1), January 1995.
- [15] T. Leschke, "Achieving speed and flexibility by separating management from protection: embracing the Exokernel operating system", Operating Systems Review, Vol. 38(4), pp. 5-19, October 2004.
- [16] X. Lu and S. F. Smith, "A Microkernel Virtual Machine: Building Security with Clear Interfaces", ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Ottawa, Canada, June 10, pp. 47-56, 2006.
- [17] J. Liedtke, "On Micro-Kernel Construction", Proceedings of the 15th ACM Symposium on Operating System Principles, ACM, December 1995.
- [18] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos and D. Magenheimer, "Are Virtual Machine Monitors Microkernels Done Right?", Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS-X), June 12-15, Santa-Fe, NM, 2005.
- [19] G. Anzinger and N. Gamble, "Design of a Fully Preemptable Linux Kernel", MontaVista Software, September 2000.
- [20] B. Kuhn "The Linux real time interrupt patch", <http://linuxdevices.com/articles/AT6105045931.html>, 2004.
- [21] C. Williams, "Linux Scheduler Latency", Red Hat Inc., March 2002.
- [22] K. Dankwardt, "Real Time and Linux, Part 3: Sub-Kernels and Benchmarks", <http://www.linuxdevices.com/articles/AT6320079446.html>, May 2001.
- [23] P. Mantegazz, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes, "RTAI: Real-Time Application Interface", <http://www.linuxdevices.com/articles/AT6605918741.html>, April 2000.
- [24] T. Maeda, "Safe Execution of User programs in Kernel Mode Using Typed Assembly Language", Master Thesis, The University of Tokyo, February, 2002.
- [25] T. Maeda, "Kernel Mode Linux: Execute user process in kernel mode", <http://www.yl.is.s.u-tokyo.ac.jp/~tosh/kml/>, 2002.
- [26] T. Maeda, "Kernel Mode Linux", Linux Journal, Issue 109, pp. 62-67, May 2003.
- [27] M. Gorman, "Understanding The Linux Virtual Memory Manager", Prentice Hall, Bruce Perens' Open Source Series, April 2004.