Ksplice: An Automatic System for Rebootless Kernel Security Updates

by

Jeffrey Brian Arnold

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2008

Author .....
Department of Electrical Engineering and Computer Science
May 8, 2008

Certified by .....
M. Frans Kaashoek
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Ksplice: An Automatic System for Rebootless Kernel Security Updates

by

Jeffrey Brian Arnold

Submitted to the Department of Electrical Engineering and Computer Science
on May 8, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Ksplice allows system administrators to apply security patches to their operating system kernels without having to reboot. Based on a source code patch and the kernel source code to be patched, Ksplice applies the patch to the corresponding running kernel, without requiring work from a programmer. To be fully automatic, Ksplice's design is limited to patches that do not introduce semantic changes to data structures, but a study of all significant x86-32 Linux security patches from May 2005 to December 2007 finds that only eight patches of 50 make semantic changes. An evaluation with Debian and kernel.org Linux kernels shows that Ksplice can automatically apply the remaining 42 patches, which means that 84% of the Linux kernel vulnerabilities from this interval can be corrected by Ksplice without the need for rebooting.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Acknowledgments

I thank Frans Kaashoek for his invaluable guidance on this project. I also thank Nelson Elhage, Sam Hartman, Anders Kaseorg, Russ Cox, and Robert Morris for useful discussions. Lastly, I thank the MIT Student Information Processing Board (SIPB) for providing an inspiring environment for working on computing projects.

Contents

Chapter 1

Introduction

Several contemporary operating systems release kernel security patches many times per year. Some of these kernel patches repair vulnerabilities which would otherwise potentially allow an attacker to gain full administrator privileges on the operating system. Applying these security patches typically requires rebooting the kernel, which results in downtime and loss of state (e.g., all active network connections). Since rebooting can cause disruption, system administrators often delay performing security updates, despite the risk of compromises. This paper describes and evaluates *Ksplice*, a system for performing *hot updates*, which change a running kernel without rebooting it.

When software developers correct a security problem in the source code of a C program (which is the language of many kernels), they create and distribute a patch, which consists of a set of changes to the source code. In the case of a kernel change, software vendors or system administrators apply the patch to their copy of the source code, build a new kernel, and then distribute that new binary kernel to servers and end-user machines, which must be rebooted in order to run the new kernel.

Safely performing a hot update, instead of a traditional update, on an unmodified C program currently requires significant programmer involvement. Existing hot update practices, described in Chapter 5, rely on a programmer to write source code files with certain properties (e.g., [7, 20]) or require manual inspection of the running binary to achieve safety guarantees and resolve ambiguous symbols (e.g., [1]).

Requiring significant programmer involvement in creating a hot update increases both the cost and the risk of the update, which discourages the adoption of hot updates. Ideally, we would like to be able to safely construct hot updates, with no human involvement, from existing information, such as the patch used to correct the security vulnerability.

Although performing an arbitrary source code change as a hot update is not feasible without a programmer writing new code to perform the transition, some patches can safely be performed as a hot update based solely on the source code differences. Specifically, we have sufficient information for a hot update when a patch does not make *semantic changes* to data structures—that is, changes that would require existing instances of kernel data structures to be transformed (e.g., a patch that adds a field to a global data structure would require the existing data structures to change). Since kernel security patches tend to make as few software modifications as possible, we can expect many of these patches to make no semantic changes to data structures.

This paper focuses on developing techniques for safely performing hot updates without programmer involvement, in the common case that the security patch does not make semantic changes to data structures. This problem contains several challenges that have not been solved by previous hot update systems, and effectively solving these problems helps many hot update systems (not just systems that focus on patches without semantic changes to data structures). The techniques that we describe are, compared to previous hot update systems, oriented much more towards analysis at the object code level, which offers various advantages, as described in Chapter 2 and Chapter 5.

Ksplice can, without restarting the kernel, apply any source code patch that only needs to modify the kernel text (including patches to kernel modules and assembly files). Unlike other hot update systems, Ksplice takes as input only a source code patch and the original kernel source code, and it updates the running kernel correctly, with no further human assistance required.

Ksplice does not require any preparation before the system is originally booted.

The running kernel does not need to have been specially compiled, for example. Other hot update systems require a special kernel design that is conducive to hot updates [3, 4], require a customized compiler [1, 21], or require a virtual machine monitor [7].

The performance impact of inserting a Ksplice update into a kernel is minimal. A small amount of additional kernel memory, proportional to the size of the replacement functions, will be expended, and function calls to the replaced functions will take a few cycles longer because of inserted jump instructions. On modern systems, this overhead should be negligible under most circumstances.

Although generating a Ksplice hot update from a compatible patch requires no human effort, Ksplice requires a person to perform a single check before invoking Ksplice: a person is expected to confirm that the target security patch does not make any semantic changes to data structures. Performing this check requires only seconds or a few minutes for most security patches. This check is much simpler than the programmer work required by other hot update systems.

We implemented Ksplice for Linux, but the techniques that Ksplice employs apply to other operating systems. To evaluate Ksplice's approach, we applied Ksplice to 50 Linux patches for kernel security vulnerabilities from May 2005 to December 2007. The 50 include all documented x86-32 Linux kernel vulnerabilities from this time interval with greater consequences than denial of service. We applied the patches to five different running Debian kernels and six different running kernel.org kernels. Of the 50 patches, eight make semantic changes and cannot be applied by Ksplice. The remaining 42 of the 50 patches can be applied using Ksplice without the need for rebooting, which is a significant advance over the current state in which system administrators always have to reboot their systems.

The contributions of this paper are a new binary-level approach for constructing hot updates and an evaluation of this system against Linux security patches. This Ksplice evaluation is, to our knowledge, the first evaluation of any kernel hot update system against a comprehensive list of the significant security vulnerabilities within a commodity operating system over a period of time. We believe that this evaluation

clearly demonstrates that hot updates currently have the potential to eliminate most kernel security reboots, while requiring little additional work from any programmer.

The rest of this paper is organized as follows: The next chapter presents Ksplice's design for performing hot kernel updates. Chapter 3 describes Linux-specific implementation considerations. Chapter 4 tests Ksplice against security patches from May 2005 to December 2007. Chapter 5 and Chapter 6 relate Ksplice to previous work and kernel developer community discussions. Chapter 7 summarizes our conclusions and directions for future work.

Chapter 2

Design

To apply a patch, Ksplice replaces all of the functions that the patch changes. If any code within a function is patched, then Ksplice will replace the entire function. Ksplice replaces a function by linking the patched version of the function into the kernel's address space and by causing all callers of the original function to invoke the patched version. Ksplice replaces entire functions since they tend to have a well-defined entry point, at the beginning of the function, which is convenient for redirecting execution flow away from an obsolete function to a replacement function.

Although replacing entire functions is relatively convenient, this replacement must be done with care. It involves generating the object code for the patched function, resolving symbols in the object code of the patched function, stopping the kernel temporarily, rewriting the initial instructions in the obsolete function to point callers at the patched function, and starting the kernel again. The rest of the section discusses how Ksplice performs these operations.

2.1 Overview

Ksplice accepts as input the original kernel source code and a source code patch. Through a multi-stage process, Ksplice uses this input to create a kernel module that contains the patched functions (see Figure 2-1). Most of the stages of this process make few assumptions about the underlying operating system. The design assumes

a reasonable binary format for object code and a basic facility for kernel modules. In order to make our examples specific, we assume in our discussion that the object code format is ELF, the Executable and Linkable Format [11], which is widely used by Linux, BSD, and Solaris.

In order to generate a hot update, Ksplice must determine what code within the kernel has been changed by the source code patch. Ksplice performs this analysis at the ELF object code layer, rather than at the C source code layer, for four reasons.

First, operating at the object code layer allows the C compiler to perform the work of parsing the C language. Other hot update systems parse C by modifying a mainstream C compiler [1] or by relying upon a research C parser [21], such as the C Intermediate Language tools [22]. Modifying a mainstream C compiler creates a dependency upon that version of that compiler and can make a hot update system more difficult to maintain over time. Using a research C parser creates a dependency upon a software system that is not widely used and that does not necessarily support all of the C extensions needed to compile the code in question. Operating at the object code layer and relying upon the well-established GNU BFD library, which is used extensively by the popular GNU binary utilities (including the GNU linker `ld`), does not have these disadvantages.

Second, looking for object code differences rather than source code differences naturally provides support for function signature changes, changes to functions with static local variables, and changes to assembly files, features which are notably missing from other hot update work [1].

Third, Ksplice's binary comparison techniques avoid a subtle problem with inline functions. This problem affects other hot update systems that, like Ksplice, are designed to update programs that were compiled without foresight of the update system. Compilers will sometimes inline a function in some places and not inline it in others, which can lead to a hot update system thinking that it replaced the only copy of a function, while other inline copies still exist and contain the outdated code. As a result, other hot update systems could leave vulnerable or obsolete code in the kernel, which could result in a security compromise or data corruption. This problem

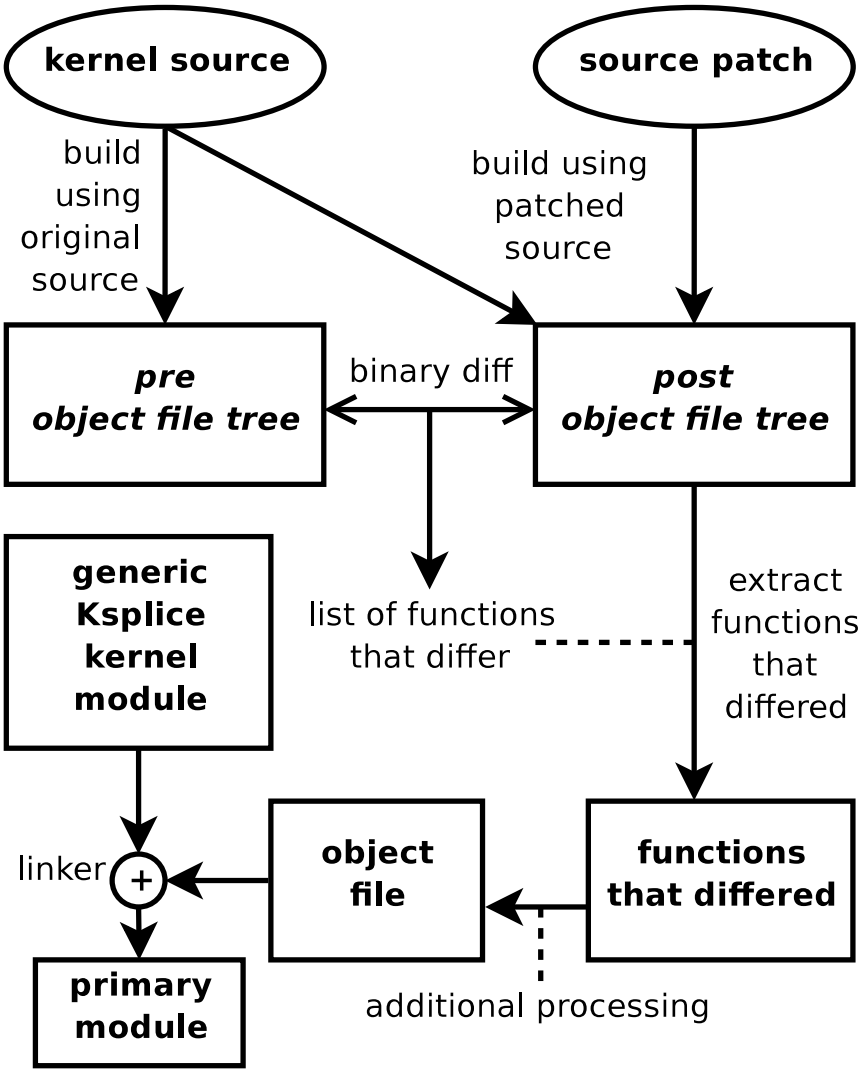


Figure 2-1: Ksplice's process for creating a hot update

essentially cannot be avoided without doing some kind of binary code comparison since any function can potentially be inlined by the compiler. In other words, a hot update system for legacy binaries cannot avoid this problem by simply declaring inline functions to be outside of its scope; ignoring this problem is always a safety risk.

Lastly, Ksplice's focus on information available in the object code helps Ksplice resolve symbols in difficult situations, as discussed in Section 2.6.

In order to understand the effect of a source code patch on the kernel, Ksplice performs two kernel builds and looks at how the resulting ELF object files differ. Ksplice builds the original unmodified kernel source in a directory tree that we will refer to as the *pre* tree. Next, Ksplice copies this directory tree into a new location and applies the source code patch. We refer to this patched directory tree as the *post* tree. Ksplice then performs a build of the *post* tree, which recompiles any object files whose constituent source files have been changed by the source code patch. Ksplice compares the *pre* and *post* object file trees in order to determine which functions were changed by the patch. It extracts the changed functions from the *post* tree and puts them into an object file.

Ksplice then creates a kernel module by combining this object file with generic code for performing hot updates. A system administrator can then insert the kernel module into the kernel using the operating system's standard kernel module facility. After the kernel has loaded and initialized the hot update, the hot update will locate entry point symbols in the running kernel's memory and insert trampolines at those locations. These trampolines will direct execution away from the vulnerable code to new code, located elsewhere, that has been loaded into kernel memory as part of the hot update.

The kernel module generated by Ksplice is not machine-specific and may be used on any machine running a kernel compiled from the same original source code. The patch can easily be reversed by running a user space program that communicates with the module. Reversing the patch involves removing the trampolines so that the original function text is once again executed.

2.2 Challenges

The described design must address several challenges in order to be practical:

- Avoiding extraneous differences. The corresponding *pre* and *post* object files will contain many differences that are only tangentially related to the source code patch. As an example, the GNU C compiler [25] will, by default, lay out an entire object file's executable text within a single ELF section named `.text`, and the C compiler will therefore generate much code within this section that performs relative jumps to other addresses within this ELF section. If a single function within an object file is changed in length as a result of the source code patch, then many relative jump offsets throughout the entire object file will potentially change as a result of what was originally a simple change to a single function.

- Determining entry points. Most C functions have one entry point, but assembly-enhanced parts of the kernel can have several.

- Resolving symbols to kernel addresses. We must use our own mechanism for performing relocations, rather than the kernel's provided mechanism, because we need to allow the *post* code that is being relocated to reference local functions and data structures. For example, we need to allow the replacement code to reference the static local variables of existing functions. The kernel symbol table contains entries for local symbols in addition to global symbols, but the kernel's provided relocation system only considers global symbols eligible for fulfilling relocations.

- Handling ambiguous or missing symbols. Attempting to resolve symbols based on the names in the kernel's symbol table can commonly cause problems when a symbol name appears more than once or does not appear at all. For example, the replacement code might reference a function by the name `translate_table`, and two or more local symbols with that name might appear in the kernel.

In this situation, the hot update system needs a way of determining which `translate_table` address should be used to fulfill this relocation.

- Handling compiler differences. The compiler used to generate the hot update may not behave the same as the compiler used to compile the original kernel. The *pre* code could therefore be different from the code in the running kernel, which we will refer to as the *run* version of the machine code. These differences could cause various serious problems with the hot update process, such as causing the hot update to neglect to update sections of vulnerable code. Consider the situation in which a patched function (`patched_func`) is called from a non-patched function (`calling_func`) that is within the same compilation unit. Assume that, in the *run* code, the compiler decided to inline `calling_func`'s call to `patched_func`. If, in the *pre* and *post* code, the compiler chooses to have `calling_func` perform a call to `patched_func` rather than making the call be inline, then the *pre* and *post* code for `calling_func` will be identical. As a result, the system would not think that `calling_func` needs to be replaced in the running kernel. One would not expect this kind of situation to arise often, but silently failing to update sections of vulnerable code is unacceptable even in rare circumstances.

- Finding a safe time to update. After the hot update module has been inserted into the kernel, the system must find an appropriate time to insert the trampolines and thereby switch over to using the replacement functions. In order for Ksplice to be able to safely replace a function, that function should not be in the middle of being executed by any thread on the system. Consider a situation in which one patched function (`calling_func`) calls another patched function (`called_func`). If the patch changes the interface between `calling_func` and `called_func` in any way, then it could be important to ensure that the obsolete `calling_func` only calls the obsolete `called_func`. Unfortunately, if `calling_func` is in the middle of being executed when `called_func` has its trampoline inserted, then the obsolete `calling_func` could call the replace-

ment `called_func`. This incorrect call could cause serious problems.

- Patching a previously-patched kernel. After one hot update has been applied to a running system, safely applying additional hot updates to that system should remain possible.

The rest of this chapter describes how Ksplice addresses these challenges.

2.3 Handling extraneous differences

In order to identify a more minimal set of changes caused by the source code patch, Ksplice would like to be able to generate object code that makes no assumptions in its executable text about where functions are positioned in memory. Avoiding these function layout assumptions is also useful for generating the replacement code for the hot update.

To reduce location assumptions, all of Ksplice's kernel builds are performed with certain compiler options enabled to ensure that every C function and data structure within the kernel receives its own dedicated ELF section within the resulting object files. These options, which are included in the standard GNU C compiler but are disabled by default, are known as `-ffunction-sections` and `-fdata-sections`. Enabling these options forces the compiler to generate relocations for functions and data structures, which results in more general code that does not make assumptions about where functions and data structures are located in memory. Instead, the resulting object code contains more general assembly instructions along with ELF relocation entries so that arbitrary addresses can be plugged-in for functions and data structures.

When compiling with these options, kernel functions that have not been directly changed by the source code patch will often have identical ELF sections in the *pre* and *post* kernel trees. For various reasons, such as nondeterministic compiler optimizations, some of the resulting ELF sections could differ in places not caused by the source code patch, but such differences are unusual, difficult to avoid, and harmless. Although Ksplice would like to replace as few functions as possible, we can safely

replace a function with a different binary representation of the same source code, even if doing so is unnecessary.

2.4 Determining entry points

After determining what ELF sections differ between the *pre* and *post* trees, Ksplice generates a list of all of the entry points to those ELF text sections. Every ELF symbol pointing into a text section is considered a potential entry point to that section. C functions typically have one entry point at the start of the function, but assembly code can result in an ELF section having multiple entry points. Looking for an arbitrary number of entry points per ELF section allows Ksplice to handle patches to certain assembly-enhanced parts of the kernel that Ksplice would not otherwise be able to handle.

2.5 Resolving symbols to kernel addresses

In order to implement its own symbol resolution mechanism, which looks at both local and global symbols, Ksplice removes the original ELF relocation entries present in the ELF object files so that the kernel's loader will not recognize the ELF relocations and try to fulfill them. Instead, Ksplice stores the needed relocation information in special Ksplice-specific ELF sections that Ksplice uses to perform the relocations during the module's initialization procedure.

Ksplice must perform these relocations during the module initialization procedure, rather than in user space, in order to properly support cryptographic verification [19] of Ksplice kernel updates. Since kernel modules can be linked into and unlinked from kernel memory after the kernel has booted, the addresses of some kernel functions and data structures will vary across machines. This variation means that, if we want to distribute a single cryptographically-signed hot update kernel module that will work on all machines running a particular kernel version, then we must defer completing the relocations for that hot update until after the update module has been inserted

into the kernel and the kernel loader has had the opportunity to verify the module's cryptographic signature.

2.6 *run-pre* matching solution

Both the ambiguous symbol name problem and the compiler variation problem can be solved using an approach that we call *run-pre* matching. The compiler variation problem arises because of unexpected and undetected differences between the *run* code and the *pre* code. Ksplice can avoid failing silently in this situation by adding a step to the hot update process to check the *run* code against the *pre* code. Specifically, we should be concerned if we can find a difference between the *run* code and the *pre* code in the kernel compilation units that are being modified by the hot update.

During the process of comparing the *run* code against the *pre* code, the hot update system can also gain information about symbols that Ksplice was previously having difficulty mapping to addresses because of the ambiguous symbol name problem. The *run* code contains all of the information needed to complete the relocations for the *pre* code.

run-pre matching passes over every byte of the *pre* code, making sure that the *pre* code corresponds to the *run* code. When this process comes to a *pre* word of memory that is unknown because of a *pre* relocation entry with an ambiguous symbol name, Ksplice can compute the correct final *pre* address based on the corresponding *run* bytes in memory.

For example, consider a situation in which the *pre* code contains a function that calls `translate_table`, but two local symbols with that name appear in the kernel. The *pre* object code generated by the compiler will, as in all relocation situations, not contain a final `translate_table` address at the to-be-relocated position. Instead, the *pre* code's metadata will know that a symbol name (`translate_table`) and an "addend" * value are associated with that to-be-relocated position in the *pre* code.

*The "addend" is an offset chosen by the compiler to affect the final to-be-stored value. For x86 32-bit relative jumps, this value tends to be -4 to account for the fact that the x86 jump instructions expect an offset that is relative to the starting address of the *next* instruction.

The ELF specification says that the to-be-relocated position's final value in memory will be computed from the addend (A), the `translate_table` symbol value (S), and the final address (P) of the to-be-relocated position. Specifically, this position will take on the value $A + S - P$.

When *run-pre* matching gets to the to-be-relocated location in the *pre* code, it will note that this relocation has not yet been fulfilled, and it will examine the *run* code in order to gain the information needed to fulfill it. The *run* code contains the already-relocated value *val*, which is $val = A + S - P_{run}$. The *run-pre* matching system also knows the *run* address of that position in memory (P_{run}). The *pre* code metadata contains the addend A, and so the symbol value can be computed as $S = val + P_{run} - A$.

Although Ksplice does not require that the hot update be prepared using exactly the same compiler and assembler version that were used to prepare the original binary kernel, doing so is advisable since the *run-pre* check will, in order to be safe, abort the upgrade if it detects unexpected binary code differences.

In order to operate correctly, the code for the *run-pre* matching system needs two architecture-specific pieces of information. First, the matching system must know the list of valid jump instructions for the target architecture so that the matching system does not conclude that the *run* code and the *pre* code differ because of two relative jump instructions that point to the same location but that use differently-sized offsets for the jump.

Second, the *run-pre* matching system must know what instruction sequences are commonly used as no-op sequences by assemblers for that architecture. In order to manipulate code alignment, assemblers will sometimes insert efficient sequences of machine instructions that are equivalent to a no-op sequence. The *run-pre* matching system needs to be able to recognize these sequences so that they can be skipped during the *run-pre* matching process. The GNU assembler for x86-32 and x86-64 has a preferred no-op-equivalent sequence for each possible desired length between one byte and 15 bytes.

2.7 Finding a safe time to update

A safe time to update a function is when no thread's instruction pointer falls within that function's text in memory and when no thread's kernel stack contains a return address within that function's text in memory.

Before inserting the trampolines, Ksplice captures all of the machine's processors and checks whether the above safety condition is met for all of the functions being replaced. If this condition is not satisfied, then Ksplice tries again after a short delay. If multiple such attempts are unsuccessful, then Ksplice abandons the upgrade attempt and reports the failure.

Ksplice therefore cannot be used to upgrade *non-quiescent* kernel functions. A function is considered non-quiescent if that function is always on the call stack of some thread within the kernel. For example, the primary Linux scheduler function, `schedule`, is generally non-quiescent since sleeping threads block in the scheduler. This limitation does not prevent Ksplice from handling any of the 50 significant Linux security vulnerabilities from May 2005 to December 2007.

2.8 Patching a previously-patched kernel

When a system administrator wants to apply a new patch to a previously-patched running kernel, Ksplice needs to be provided with two inputs, which are similar to the standard Ksplice inputs:

- the source for the currently-running kernel, including any patches that have been hot-applied (this source is the "previously-patched source")
- the new source patch (which should be a difference between the previously-patched source and the desired new source)

The update process is almost exactly the same as before. The *pre* object code is generated from the previously-patched source code, and the *post* object code is generated from the previously-patched source code with the new patch applied. The

run-pre matching system will compare *pre* object code against the latest Ksplice replacement function code already in the kernel. When the hot update is applied, the previous trampoline is overwritten with a new trampoline.

Chapter 3

Implementation

We implemented Ksplice's design for Linux 2.6 on the x86-32 and x86-64 architectures. Although small parts of Ksplice, such as the trampoline assembly code, need to be implemented separately for each supported architecture, most of the system is architecture-independent.

3.1 Overview

The update "module" produced by Ksplice is actually two Linux kernel modules, so that part of the system can be unloaded after the update is complete, in order to save memory. Ksplice's implementation consists of three components:

- a generic "helper" Ksplice Linux kernel module, written in C, responsible for loading the *pre* object code and performing *run-pre* matching
- a generic "primary" Ksplice Linux kernel module, written in C, responsible for loading the *post* object code and inserting the trampolines
- user space software, written in C and Perl, that, using the kernel source and unified diff [9] provided to Ksplice as input, generates processed "helper" and "primary" object files (these object files are then linked with the generic Ksplice kernel modules to produce the ready-to-insert modules)

The primary module for a hot update must always be inserted before the helper module. The primary module will be inactive until the helper module has been loaded. Once the helper module has been loaded, the update process will begin, and after the update process is complete, the helper module can be removed. Since the helper module must contain the entire compilation unit corresponding to each patched function, it can be significantly larger than the primary module.

The user space responsibilities of performing a Ksplice update are managed by a Perl script, `ksplice-create`, that invokes C programs (written using the GNU Binary File Descriptor library) in order to perform specific operations on object files. For example, one such C program, `objdiff`, is responsible for detecting the differences between corresponding object files and reporting back what sections differ and the entry points of those sections. Another C program, `objmanip`, is responsible for removing the ELF relocations from an object file so that the relocation information can instead be stored in a Ksplice-specific ELF section, which will be processed after the module has been inserted. Essentially identical procedures are used for preparing the *pre* and *post* groups of ELF sections. The processed *pre* ELF sections, which potentially originate from several different kernel compilation units, are eventually combined with the helper kernel module. The processed *post* ELF sections are combined with the primary kernel module in a similar manner.

3.2 Capturing the CPUs to update safely

Ksplice uses Linux's `stop_machine_run` facility in order to help achieve an appropriate opportunity to insert the trampolines. The Linux kernel normally uses `stop_machine_run` for CPU hotplugging and suspend/resume functionality. When invoked, `stop_machine_run` creates one high priority kernel thread for each CPU on the machine, and it configures these threads so that each thread will only be scheduled on its own distinct CPU.

Once these high priority kernel threads have simultaneously captured all of the CPUs on the system, `stop_machine_run` will run a desired function on a single

CPU. Ksplice uses `stop_machine_run` to execute a function that checks for the safety condition discussed in Chapter 2.7. If this condition is met, the function inserts the trampolines for the hot update.

Chapter 4

Evaluation

4.1 Linux security patches

We compiled a list of significant Linux 2.6.x kernel security problems from May 2005 to December 2007. We compiled this list of vulnerabilities and the corresponding patches by matching entries in the Common Vulnerabilities and Exposures (CVE) vulnerability list [8] against the Git source control logs of Linus Torvalds' branch of the Linux kernel source [17]. Only security problems that could result in greater consequences than denial of service are included on this Linux kernel vulnerability list; specifically, all of the vulnerabilities on the list involve the potential for some kind of privilege escalation or unintended information disclosure. We excluded from this list any architecture-specific vulnerabilities that do not affect the x86-32 architecture.

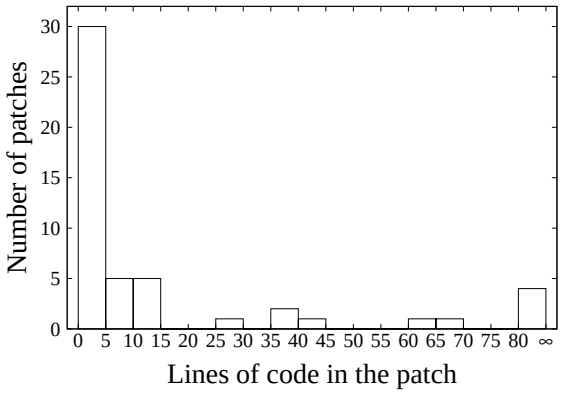
In this time interval, 50 significant x86-32 Linux kernel vulnerabilities were reported, which are listed in detail in the Appendix and summarized in Table 4.1 and Figure 4-1. Roughly one-third of the kernel vulnerabilities primarily involved information disclosure, and the rest potentially allowed for some kind of privilege escalation.

The frequency of new vulnerabilities with the potential for privilege escalation—about one vulnerability per month—is consistent with the conventional wisdom that the Linux kernel must be updated frequently in order to be secure against attack from users with local accounts on the system. Protecting against this kind of attack can be important even on systems where all of the people with accounts are trustworthy,

Table 4.1: Number of vulnerabilities by type

Type of Vulnerability	Quantity
Privilege escalation	32
Read kernel memory	10
Failure to clear memory	5
Other information disclosure	3

Figure 4-1: Number of patches by patch length



since a network attacker can commonly access a user account by compromising a password or ordinary program. In these situations, having an up-to-date kernel can mean the difference between whether an attacker gains limited access to a system or full control over it.

Figure 4-1 shows that most Linux kernel security vulnerabilities can be corrected by modifying relatively few lines of source code. Of the 50 patched vulnerabilities discussed above, 40 vulnerabilities were corrected in 15 or fewer lines of source code changes, and 30 vulnerabilities were corrected in 5 or fewer lines of changes.

Reviewing the text of these patches reveals that most Linux kernel vulnerabilities can be fully corrected without making any semantic changes to the kernel's global, dynamically-allocated, or static local data structures. We collectively refer to these data structures as the kernel's "long-lived" data structures. Since these data structures do not need to change in order to accommodate most patches, many security updates can be applied to a running system by simply updating the text of the kernel.

Table 4.2: Debian kernels used for implementation testing

Debian version number	Release date
2.6.8-2-686-smp_2.6.8-15_i386	2005-03-24
2.6.12-1-686-smp_2.6.12-1_i386	2005-07-22
2.6.16-1-686-smp_2.6.16-1_i386	2006-03-22
2.6.22-1-686_2.6.22-1_i386	2007-07-17
2.6.23-1-686_2.6.23-1_i386	2007-12-05

4.2 Methodology

To evaluate Ksplice, we applied Ksplice to the 50 patches described in Section 4.1. We are interested in how many security patches from this interval can be applied successfully to running kernels. Success means that the patch was applied and the kernel kept functioning without any observed problems while building a new kernel and sending and receiving network traffic. We expect a few of the patches to be incompatible with Ksplice because some of the patches require changes to the semantics of long-lived kernel data structures. For the vulnerabilities for which exploit code was readily available, we also tested that the exploit code worked before the hot update and did not work after the hot update.

Since no single Linux kernel version needs all of the 50 security patches (many of the vulnerabilities were introduced during the 32-month period of ongoing development), we tested Ksplice with these 50 patches using five different Linux kernels released by the Debian GNU/Linux distribution and six different "vanilla" Linux kernels released by kernel.org on behalf of Linus Torvalds. These kernels are shown in Table 4.2 and Table 4.3. The details of which patches were tested on each kernel are available in the Appendix.

The kernel.org kernels were only introduced into the evaluation process in order to test security patches that are not applicable to any released Debian kernel. For most of the patches in this category, the security problems corrected by the patch were completely absent from any released Debian kernel. Some Linux kernel security vulnerabilities are caught before they make it into many released kernels seen by users,

Table 4.3: Vanilla kernels used for implementation testing

kernel.org version number	Release date
2.6.11.9	2005-05-11
2.6.19	2006-11-29
2.6.20.4	2007-03-23
∼2.6.21.1*	2007-04-30
2.6.23-rc3	2007-08-13
2.6.24-rc2	2007-11-06

and other vulnerabilities affect portions of the kernel that are completely disabled by some Linux distributions.

We obtained the original binary and source Debian kernel packages from a historical archive of nearly all packages released by Debian since mid-2005 [26]. For each kernel, we began by fetching the compiler and assembler versions originally used by Debian in order to compile that binary kernel. We then used the Debian kernel source for that kernel as input to Ksplice, along with an unmodified security patch taken directly from Linus Torvalds' Git tree. In order to perform the hot update on a running machine, we installed the corresponding binary Debian kernel package on a machine, and we booted the machine into that kernel.

4.3 Results

Ksplice has been used to correct 42 of the 50 significant x86-32 kernel vulnerabilities during the time interval. Ksplice's system for resolving symbol names in difficult situations and its support for assembly code are important for achieving this percentage of supported patches. Nine of the 42 patches modify functions that contain ambiguous or missing symbols, and much of the kernel makes use of assembly code via common primitives that manage concurrency control and other operations.

For example, the patch for CVE-2005-1264 changes the function `raw_ioctl` in the file `drivers/char/raw.c`, but both the kernel's raw character device driver and its ipv4 implementation contain a local function named `raw_ioctl`. Since both of

*Git revision b7b5f487ab39bc10ed0694af35651a03d9cb97ff

Table 4.4: Kernel vulnerabilities that cannot be patched using the Ksplice design

CVE #	Reason for failure
2007-4571	changes data initialization value
2007-1217	changes data initialization value
2006-5753	changes data initialization value
2006-3626	changes data initialization value
2006-1056	changes data initialization value
2005-3179	changes data initialization value
2005-2709	adds field to structure
2005-2500	adds field to structure

these drivers can be included with the kernel as an optional module, a naive hot update system would find, when running on a kernel with only the ipv4 driver loaded, the single `raw_ioctl` symbol and replace it with the raw character device driver replacement function. Ksplice's *run-pre* matching system ensures that symbols are resolved correctly and that functions look as expected before they are patched.

As another example, the patch for CVE-2007-4573 modifies the x86-64 assembly file `ia32entry.S` in order to zero-extend all registers in order to avoid an arbitrary execution vulnerability in the 32-bit kernel entry path. Ksplice handles this patch using the same techniques that handle patches to pure C functions.

Eight of the 50 patches were not supported by the Ksplice design. As shown in Table 4.4, these patches change the semantics of long-lived kernel data structures, either by changing the default value of a data structure or by adding a field to a data structure. Some of the patches explicitly change the initial value of a data structure via the C variable declaration, and some patches change a data structure initialization function.

Working exploits for two recent x86 vulnerabilities are available on the web. We have used exploit code for CVE-2006-2451 [15] and CVE-2007-4573 [10] in order to confirm that these security vulnerabilities disappear when the corresponding hot updates are applied.

Chapter 5

Related Work

There are two streams of work related to Ksplice: academic papers with various approaches to the hot update problem and black hat publications describing how to perform malicious hot updates on commodity operating systems. We discuss the relationship of Ksplice to these in turn.

5.1 Research literature overview

The existing research hot update systems that are designed to update "legacy binaries" (mostly-unmodified binaries that are created with essentially no foresight of the update system) perform their patch analysis and update construction at the source code layer, rather than at the object code layer, which results in several disadvantages for these systems.

First, because of the complexity of analyzing a patch and constructing the replacement code at the source code level, DynAMOS [20] and LUCOS [7] leave essentially all of this work to a human kernel programmer. In these systems, a kernel programmer needs to construct replacement source code files with certain properties, which requires "tedious engineering effort" [7] and, as with any human involvement, is error-prone.

The one system which does implement in software the construction of replacement code, OPUS [1], suffers from several limitations related to its source-level approach

to this problem. Unlike OPUS, Ksplice's binary-level approach supports function signature changes, changes to functions with static local variables, and changes to assembly files. All of these features are natural consequences of approaching hot updates from the object code level rather than the source code level.

Intuitively, looking for object code differences works well because it comes closer than the source code changes to what we actually care about—how and where the machine's execution might be changed. Determining how to handle arbitrary source code changes can require much information about the semantics of the C and assembly languages. For example, because of implicit casting, simply changing a data type in a function prototype in a C header file (e.g., from an `int` to a `long long`) can result in changes to many different functions (which have not themselves had their source code changed at all, even after C preprocessing). The Ksplice approach to hot updates described in Chapter 2 does not require special cases in order to deal with language-level nuances such as macros, function signatures, static local variables, and whether code is written in C or raw assembly.

Second, and more importantly, essentially any hot update system which operates at the source code layer exclusively is vulnerable to an inescapable inline function problem described in detail in Section 2.1. Fundamentally, the binary version of the running program contains information that is not available in its source code, and looking at the binary code is necessary in order to guarantee the safety of the update. Any hot update system for legacy binaries needs to use a mechanism similar to Ksplice's *run-pre* matching system to be safe, in order to detect where code has been inlined by the compiler.

Third, Ksplice's binary-level *run-pre* matching system helps Ksplice resolve ambiguous symbol names, which other hot update systems either cannot handle or can only tolerate with significant human intervention. Ksplice's enhanced ability to resolve symbol names was needed for 9 of the 42 successful patches from the evaluation.

It is also worth noting that the Ksplice evaluation is structured differently from the evaluations of previous legacy binary hot update systems. Ksplice's evaluation measures Ksplice against all 50 of the significant Linux x86-32 security vulnerabilities

over a 32-month time interval. Previous evaluations of hot update systems for legacy binaries have not tested those systems against a comprehensive list of the patches over a time interval. The DynAMOS and LUCOS evaluations each describe testing 5 patches; the OPUS evaluation describes testing 26 patches from a corpus of 883 vulnerabilities.

5.2 Research literature individual systems

Since DynAMOS, LUCOS, and OPUS construct updates for legacy binaries at the source code layer, these systems are affected by the limitations described in Section 5.1.

DynAMOS is a recent hot update system that helps a programmer to manually prepare hot updates for Linux. Since DynAMOS requires a kernel programmer to write new code, its design can accomodate certain kinds of semantic changes to data structures, as long as a kernel programmer writes the appropriate code and debugs it. Ksplice's design could incorporate this functionality, but we do not do so because of the risks inherent in programmers writing code for each hot update. Ksplice implements and evaluates updates without any human component in order to judge the potential of hot updates at their safest.

LUCOS is a virtualization-based hot update system that enables a programmer to manually prepare hot updates for a Linux machine running on top of a customized version of the Xen [2] virtual machine monitor. LUCOS uses the virtual machine monitor in order to gain a high degree of control over the kernel during the update process. By controlling the kernel's underlying hardware, LUCOS can, for example, intervene when particular addresses in memory are accessed. Unlike LUCOS, Ksplice does not require virtualization.

OPUS is a user space hot update utility for C programs that shares several design choices with Ksplice. OPUS, like Ksplice, targets security updates and does not require source code design changes to the to-be-updated software. OPUS requires the least programmer work of any of the previous hot update systems, but OPUS requires

a programmer to intervene for many patches that are naturally handled by Ksplice's binary-level approach, as described in Chapter 5.1. For example, OPUS would not be able to resolve the ambiguous symbol names in 9 of the 42 kernel security patches successfully performed by Ksplice. Also, a programmer using OPUS needs to perform a tedious check for inline functions in the to-be-updated binary in order to ensure patch safety (looking for the `inline` keyword in the source code is not sufficient since modern compilers routinely inline functions that lack the keyword).

The K42 research operating system [3, 4] has implemented hot update capabilities in K42 by leveraging particular abstractions provided by that operating system's modular, object-oriented kernel. Ksplice's design focuses on immediately supporting existing, mainstream operating systems, with no strict expectations placed upon the original, running kernel. For example, a Ksplice user with an existing Linux system should not need to reboot into a redesigned, "hot update compatible" Linux kernel before they can start using hot updates.

Ginseng [21] is a user space hot update utility for C programs that handles more kinds of updates than either OPUS or Ksplice. Ginseng is capable of upgrading user space software such as OpenSSH across several years' worth of releases, but Ginseng makes significant compile-time changes to programs. Ginseng rewrites C programs at compile time in order to perform function indirection and type wrapping, and Ginseng expects a programmer to annotate the to-be-updated software to indicate safe update points. In contrast, OPUS and Ksplice do not rewrite the to-be-updated software at compile time and do not require any programmer annotations.

The DAS operating system [12] included hot update primitives in the operating system, but these primitives could not be used to upgrade the kernel.

Gupta et al. built an early system [13] for performing hot updates on C user space programs that is a predecessor to OPUS. Unlike OPUS, the system requires programs to be linked against a special library, and, during an upgrade, it loses program state stored in the kernel because of how it creates a new process instead of performing the upgrade in place.

In other work, Gupta et al. proved that verifying whether or not a programmer has

provided a correct transition function for accomplishing a source code upgrade is, in the general case, undecidable [14]. In other words, hot update system software cannot, in the general case, prove that a source code patch, along with a state transformation function, results in a valid state for the new program.

Many other systems have previously been designed for modifying a running program's behavior, but most such systems cannot perform updates to legacy C binaries, such as an unmodified kernel compiled from C.

5.3 Black hat techniques

The black hat community has been performing hot updates on commodity operating system kernels for many years as part of rootkits. Computer attackers benefit from modifying the kernel so that they can hide their activities and exert a high level of control over the system.

Publications on rootkits for the Linux, BSD, and Microsoft Windows operating systems [5, 16, 18, 24] describe techniques that aid in the construction of hot updates for these platforms. Black hats, however, have notably different hot update goals than system administrators; a black hat only needs one manually-constructed hot update in order to succeed, and, in general, a black hat is more willing than a system administrator to tolerate a slight chance that a hot update will destabilize the target machine.

Instead of pursuing a generalized approach for safely accomplishing arbitrary source code hot updates, these documents tend to focus on simple approaches which usually work for accomplishing particular goals. For example, these publications suggest using memory patterns (called "keys"), which are a few bytes long, in order to find particular parts of the kernel, such as particular data structures, in memory. These multibyte keys can have several problems, such as appearing several times in memory or not appearing at all on a different machine. Various strategies exist for obtaining a "reasonable guess at how useful a key is and if a key is not at all stable" [5] (for example, a person can try a key on multiple machines and insert a wildcard into the

key if necessary), but these strategies are laborious and still do not provide strong expectations about whether the update will work. Ksplice's approach for generating expectations for the contents of kernel memory and systematically mapping symbol names to values is significantly more general than the techniques described in these rootkit publications.

Although Ksplice is safer and easier to use than existing hot update practices, Ksplice does not provide malware authors with any troubling capabilities that they do not already possess. Black hats have known for many years how to create rootkits that accomplish their goals using ad hoc kernel inspection and modification techniques. For this reason, once an attacker has unrestricted access to kernel memory, a computer system must already be assumed to be completely compromised. The best way to protect against attackers is to promptly patch security vulnerabilities so that attackers never gain unrestricted access to kernel memory. The goal of Ksplice is to make this kernel patching process easier.

Chapter 6

Discussion

Within kernel developer communities, several arguments have been presented about why eliminating upgrade reboots is more trouble than it is worth. A summary of some of these arguments and Ksplice's perspective follows.

Isn't a general solution too hard to adopt? Some people have approached hot software upgrades by looking at the problem of how to migrate a running system from one arbitrary kernel version to the next-released version. Since adjacent kernel versions sometimes contain major code changes, automatically handling adjacent version transitions can present significant challenges. Systems for handling arbitrary upgrades inevitably increase the workload associated with releasing a new kernel version; for example, a developer might need to write and debug a state-transformer in order to bring data from an old format into a new format.

As discussed in Section 4.1, most Linux kernel security patches are quite limited in scope and do not change the semantics of any long-lived data structures. We can therefore leave the kernel's internal data entirely alone for most security hot updates, which allows for an easy-to-adopt solution.

Aren't dynamic kernel modules enough? Some people have argued that a kernel module system, which allows the superuser to dynamically add and remove subsystems from the kernel, eliminates the need for other mechanisms for updating

the kernel. For example, a security vulnerability in a particular network driver could be corrected by unloading the old version of that module and then loading a new version.

Unfortunately, much kernel code cannot be treated as a freely-unloadable module either because it contains core functions that cannot be unloaded at all or because it contains important state that the user would not want to lose by unloading the module. For example, unloading a networking driver could cause the system to lose important networking state, which would potentially create as much inconvenience as a reboot.

Isn't rebooting just fine? Some people argue that rebooting simply is not a serious concern, because software—in particular, server software—should already handle machine reboots on an application level gracefully. Although some systems are indeed designed in order to minimize the negative impact of a machine reboot, many applications handle reboots poorly because they do not support saving and resuming their exact previous state. Handling these problems in every application is time-consuming for developers, and some applications will likely never handle restarts as well as one would like. Furthermore, even under the best circumstances (with ideal software), the interruption associated with reboots can be undesirable. Ksplice provides a low-cost, easy-to-adopt alternative.

Isn't redundancy, not a hot update system, the right way to achieve high availability? In some situations where high availability is desirable, redundancy might be difficult to achieve for cost or other reasons. More importantly, hot update systems can provide significant benefit even in applications where high availability is not important. Many desktop machines are not rebooted in order to apply kernel security updates because of the burden imposed by rebooting. High availability is not important on these machines, but they can still benefit significantly from hot updates (since hot updates allow them to be patched when they would not be patched otherwise).

Doesn't kexec already provide the ability to update the kernel? kexec [23] is a Linux feature which was designed to help users who have a buggy or slow BIOS to reboot their computers faster; it is not a hot update system. When kexec boots a machine into a new kernel, the entire state of the running machine is lost, including all kernel state and all programs in user space.

Doesn't any software that modifes a running program infringe on U.S. patent 10/307,902? No, that document is a patent application, not a patent, and that application received a "final rejection" in April 2006 because of prior art. Programmers have been making modifications to running programs since some of the earliest computer systems.

Chapter 7

Conclusions and Future Work

Ksplice's binary-level approach to hot updates has been demonstrated to be able to construct updates, without programmer involvement, for 84% of the 50 significant Linux kernel security patches from May 2005 to December 2007. The remaining patches make semantic changes to data structures and therefore would require programmer involvement in order to perform as hot updates.

A system administrator could, at the present time, use an implementation of Ksplice to eliminate most reboots associated with security upgrades, which is a notable advance over the current state.

Due to Ksplice's focus on enabling safe hot updates that require minimal programmer involvement, it should be possible for any Linux distributor—or other motivated individual—to start releasing Ksplice-based hot update packages for common starting kernel configurations. People who subscribe their systems to these updates would be able to transparently receive kernel hot updates along with the user space software updates to their system. This kind of distribution of hot updates would, without any ongoing effort from users, significantly reduce how frequently they are notified by their computer that they need to reboot for pending security updates to take effect. Distribution of hot kernel security updates can reduce downtime, decrease windows of security vulnerability, and improve the user experience.

Bibliography

[1] Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th USENIX Security Symposium*. August 2005.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., et al. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. October 2003.

[3] Baumann, A., Appavoo, J., Wisniewski, R. W., et al. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proceedings of the 2007 USENIX Annual Technical Conference*. June 2007.

[4] Baumann, A., Appavoo, J., Silva, D. D., Kerr, J., Krieger, O. and Wisniewski, R. W. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference*. April 2005.

[5] Cesare, S. Runtime Kernel kmem Patching. `http://doc.bughunter.net/rootkit-backdoor/kmem-patching.html`. November 1998.

[6] Chamberlain, S. LIB BFD, the Binary File Descriptor Library. `http://sourceware.org/binutils/docs-2.18/bfd/index.html`.

[7] Chen, H., Chen, R., Zhang, F., Zang, B., and Yew, P. Live updating operating systems using virtualization. In *Proceedings of the 2nd USENIX Symposium on Virtual Execution Environments*. June 2006.

[8] Common Vulnerabilities and Exposures List. `http://cve.mitre.org/cve`.

[9] Comparing and Merging Files: Unified Format. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[10] Elhage, N. Root exploit for CVE-2007-4573. http://web.mit.edu/nelhage/Public/cve-2007-4573.c.

[11] Executable and Linkable Format Specification. http://www.skyfree.org/linux/references/ELF_Format.pdf.

[12] Goullon, H., Isle, R., and Löhr, K. Dynamic Restructuring in an Experimental Operating System. In *IEEE Transactions on Software Engineering*. July 1978, pp. 298-307.

[13] Gupta, D., Jalote, P. On-line software version change using state transfer between processes. In *Software—Practice and Expereince*. September 1993, pp. 949-964.

[14] Gupta, D., Jalote, P., and Barua, G. A formal framework for on-line software version change. In *IEEE Transactions on Software Engineering*. 1996, pp. 120-131.

[15] Hernandez, R. Local r00t Exploit for PRCTL Core Dump Handling. http://seclists.org/fulldisclosure/2006/Jul/0235.html. July 2006.

[16] Hoglund, G., and Butler, J. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, Inc. 2006.

[17] How to Follow Linux Kernel Development with git. http://www.kernel.org/doc/local/git-quick.html.

[18] Kong, Joseph. *Designing BSD Rootkits*. No Starch Press, Inc. 2007.

[19] Kroah-Hartman, G. Signed Kernel Modules. http://www.linuxjournal.com/article/7130. January 2004.

[20] Makris, K. and Ryu, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the EuroSys Conference*. Lisbon, Portugal. March 2007.

[21] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. June 2006.

[22] Necula, G., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compiler Construction*. April 2002.

[23] Nellitheertha, H. Reboot Linux faster using kexec. `http://www.ibm.com/developerworks/linux/library/l-kexec.html`. May 2004.

[24] sd (`sd@sf.cz`), devik (`devik@cdi.cz`). Linux on-the-fly kernel patching without LKM. December 2001. `http://www.phrack.org/issues.html?issue=58&id=7#article`.

[25] Stallman, R. *Using and Porting GNU CC*. Free Software Foundation, Inc. July 1999.

[26] Ukai, Fumitoshi. snapshot.debian.net. `http://snapshot.debian.net`.

Appendix

The table below shows 50 security vulnerabilities from May 2005 to December 2007 along with the type of each vulnerability and the sum of the number of non-empty lines of non-comment code added and/or removed by the associated patch. The patch dates do not proceed in chronological order because CVE numbers are not necessarily assigned in the same order in which vulnerabilities are corrected in Linus' kernel tree. The next two columns indicate the date that Linus' kernel tree was modified to include the patch and the identification number that Git automatically assigned to the patch. This commit ID number provides a reliable mechanism for retrieving the patch and its associated developer commentary. The final column indicates the Linux kernel version on which the patch was successfully applied using Ksplice. "N/A" appears in the final column when that vulnerability's patch changes data structure semantics and therefore cannot be used with Ksplice.

We performed the vulnerability type classifications by reading the CVE vulnerability descriptions and the Linux source code patches. "Read kmem" refers to the potential for a local user to read the contents of sensitive kernel memory. "Clear mem" refers to the potential for a local user to receive a buffer that contains traces of sensitive information that should have previously been cleared from the buffer by the kernel. "Disclosure" refers to other vulnerabilities which directly enable a local user to gain access to sensitive information; further exploitation of the system once this information has been obtained might, or might not, be possible. "Escalation" refers to vulnerabilities which potentially allow a user to perform privileged operations, such as the ability to execute arbitrary code as the kernel or the ability to write to privileged I/O ports.

Linux x86-32 kernel vulnerabilities from May 2005 to December 2007

CVE #	Type	Lines ±	Patch Date	Linus Git Commit # (Patch+Log)	Test kernel
2005-1263	Escalation	4	2005-05-11	a84a505956f5c795a9ab3d60d97b6b91a27aa571	2.6.8
2005-1264	Escalation	3	2005-05-13	68f66feb300423bb9ee5daecb1951af394425a38	2.6.8
2005-1589	Escalation	4	2005-05-14	118326e940bdecef6c459d42ccf05256ba86daa7	2.6.11.9
2005-2456	Escalation	3	2005-07-26	a4f1bac62564049ea4718c4624b0fadc9f597c84	2.6.8
2005-2492	Read kmem	4	2005-09-19	6d1cfe3f1752f17e297df60c8bcc6cd6e0a58449	2.6.12
2005-2500	Escalation	3	2005-08-10	58fcb8df0bf663bb6b8f46cd3010bfe8d13d97cf	N/A
2005-2709	Escalation	115	2005-11-04	330d57fb98a916fa8e1363846540dd420e99499a	N/A
2005-3179	Disclosure	2	2005-10-03	c0758146adbe39514e75ac860ce7e49f865c2297	N/A
2005-3180	Clear mem	10	2005-10-04	9bc39bec87ee3e35897fe27441e979e7c208f624	2.6.12
2005-3276	Clear mem	1	2005-07-27	71ae18ec690953e9ba7107c7cc44589c2cc0d9f1	2.6.12
2005-3784	Escalation	2	2005-11-10	7ed0175a462c4c30f6df6fac1cccac058f997739	2.6.12
2005-4605	Read kmem	38	2005-12-30	8b90db0df7187a01fb7177f1f812123138f562cf	2.6.12
2005-4639	Escalation	2	2005-11-08	5c15c0b4fa850543b8ccfcf93686d24456cc384d	2.6.12
2006-0039	Read kmem	4	2006-05-19	2c8ac66bb2ff89e759f0d632a27cc64205e9ddd9	2.6.16
2006-0095	Clear mem	4	2006-01-06	9d3520a339d62f942085e9888f66905eb8b350bd	2.6.12
2006-0457	Read kmem	15	2006-02-03	6d94074f0804143eac6bce72dc04447c0040e7d8	2.6.12
2006-1056	Disclosure	42	2006-04-20	18bd057b1408cd110ed23281533430cfc2d52091	N/A
2006-1343	Clear mem	2	2006-05-28	6c813c3fe9e30fcf3c4d94d2ba24108babd745b0	2.6.16
2006-1524	Escalation	3	2006-04-17	69cf0fac6052c5bd3fb3469a41d4216e926028f8	2.6.16
2006-1857	Escalation	4	2006-05-19	a601266e4f3c479790f373c2e3122a766d123652	2.6.16
2006-1858	Escalation	6	2006-05-19	dd2d1c6f2958d027e4591ca5d2a04dfe36ca6512	2.6.16
2006-1863	Escalation	9	2006-04-21	296034f7de8bdf111984ce1630ac598a9c94a253	2.6.16
2006-1864	Escalation	3	2006-05-15	3b7c8108273bed41a2fc04533cc9f2026ff38c8e	2.6.16
2006-2071	Escalation	2	2006-04-12	b78b6af66a5fbaf17d7e6bfc32384df5e34408c8	2.6.16
2006-2451	Escalation	2	2006-07-12	abf75a5033d4da7b8a7e92321d74021d1fcfb502	2.6.16
2006-2935	Escalation	2	2006-07-10	454d6fbc48374be8f53b9bafaa86530cf8eb3bc1	2.6.16
2006-3626	Escalation	1	2006-07-14	18b0bbd8ca6d3cb90425aa0d77b99a762c6d6de3	N/A

Linux x86-32 kernel vulnerabilities from May 2005 to December 2007 (continued)

CVE #	Type	Lines ±	Patch Date	Linus Git Commit # (Patch+Log)	Test kernel
2006-3745	Escalation	69	2006-08-22	c164a9ba0a8870c5c9d353f63085319931d69f23	2.6.16
2006-4813	Clear mem	2	2006-10-11	8c58165108e26d18849a0138c719e680f281197a	2.6.16
2006-5751	Escalation	7	2006-11-20	ba8379b220509e9448c00a77cf6c15ac2a559cc7	2.6.16
2006-5753	Escalation	285	2007-01-05	be6aab0e9fa6d3c6d75aa1e38ac972d8b4ee82b8	N/A
2006-6106	Escalation	30	2007-01-08	f4777569204cb59f2f04fbe9ef4e9a6918209104	2.6.16
2006-6304	Escalation	3	2006-12-06	6d4df677f8a60ea6bc0ef1a596c1a3a79b1d4882	2.6.19
2007-0005	Escalation	3	2007-03-06	059819a41d4331316dd8ddcf977a24ab338f4300	2.6.16
2007-0958	Escalation	4	2007-01-26	1fb844961818ce94e782acf6a96b92dc2303553b	2.6.19
2007-1000	Read kmem	2	2007-03-09	d2b02ed9487ed25832d19534575052e43f8e0c4f	2.6.16
2007-1217	Escalation	364	2007-02-28	17f0cd2f350b90b28301e27fe0e39f34bfe7e730	N/A
2007-1353	Read kmem	11	2007-05-05	0878b6667f28772aa7d6b735abff53efc7bf6d91	2.6.16
2007-1730	Read kmem	1	2007-03-16	d35690beda1429544d46c8eb34b2e3a8c37ab299	2.6.16
2007-1734	Read kmem	4	2007-03-28	39ebc0276bada8bb70e067cb6d0eb71839c0fb08	2.6.20.4
2007-2480	Escalation	38	2007-04-30	de34ed91c4ffa4727964a832c46e624dd1495cf5	∼2.6.21.1*
2007-2875	Read kmem	15	2007-05-09	85badbdf5120d246ce2bb3f1a7689a805f9c9006	2.6.20.4
2007-3105	Escalation	9	2007-07-19	5a021e9ffd56c22700133ebc37d607f95be8f7bd	2.6.22
2007-3848	Escalation	13	2007-08-17	d2d56c5f51028cb9f3d800882eb6f4cbd3f9099f	2.6.23-rc3
2007-3851	Escalation	15	2007-08-07	21f16289270447673a7263ccc0b22d562fb01ecb	2.6.22
2007-4308	Escalation	4	2007-11-07	5f78e89b5f7041895c4820be5c000792243b634f	2.6.23
2007-4571	Read kmem	65	2007-09-17	ccec6e2c4a74adf76ed4e2478091a311b1806212	N/A
2007-5904	Escalation	199	2007-11-13	133672efbc1085f9af990bdc145e1822ea93bcf3	2.6.24-rc2
2007-6063	Escalation	5	2007-12-01	eafe1aa37e6ec2d56f14732b5240c4dd09f0613a	2.6.23
2007-6206	Disclosure	2	2007-11-28	c46f739dd39db3b07ab5deb4e3ec81e1c04a91af	2.6.23

*Git revision b7b5f487ab39bc10ed0694af35651a03d9cb97ff