anjapepunkt, photocase.com

Applying updates to an active kernel with Ksplice

PIECE OF CAKE

Uptime is often just as important as updates. But doesn't a kernel patch require a reboot? Ksplice lets you have your cake and eat it too.

BY NILS MAGNUS

Many administrators don't relish the thought of installing a new kernel, so most distributions come with package management and installation tools that simplify the task. After you load and install the new kernel and register it with the bootloader, the reboot should take just a couple of minutes. But a couple of minutes of downtime is too long for some applications.

If you manage a system used for number crunching, such as in weather simulations, or a system that has to manage a large number of network connections, such as a phone server or online shop, you probably would prefer to avoid a reboot.

On the other hand, administrators are also responsible for the security of the systems they manage. Surveys have revealed that developers discover a Linux kernel bug every three weeks on average. The open development model means that patches appear often.

Jeffrey Brian Arnold from the Massachusetts Institute of Technology established in a survey [1] that patches were very simple in most cases. Eighty percent of them comprised fewer than 15 lines of code, and more than half were little more than one-liners. Problems are often caused by "off-by-one errors," in which a developer has simply miscounted a parameter such as an array boundary. Bugs of this type are very easy to resolve. For example, Listing 1 shows a patch for the *prctl()* system call, which resolves the vulnerability listed as CVE-2006-2451. The problem, for which exploits exist, takes just one line of code to fix.

Arnold developed the Ksplice package [2] with the idea of patching the active kernel directly, without the need for a reboot. The program just needs the source code for the current kernel, the configu-

Listing 1: Patch for CVE-2006-2451

```
01 diff --git a/kernel/sys.c b/kernel/sys.c
02
03 --- a/kernel/sys.c
04 +++ b/kernel/sys.c
05 @@ -1991,7 +1991,7 @@ asmlinkage long sys_prctl(int option, unsigned
   long arg2, unsigned long arg3,
06         case PR_SET_DUMPABLE:
07 -               if (arg2 < 0 || arg2 > 2) {
08 +               if (arg2 < 0 || arg2 > 1) {
09                         error = -EINVAL;
10                         break;
11                 }
```

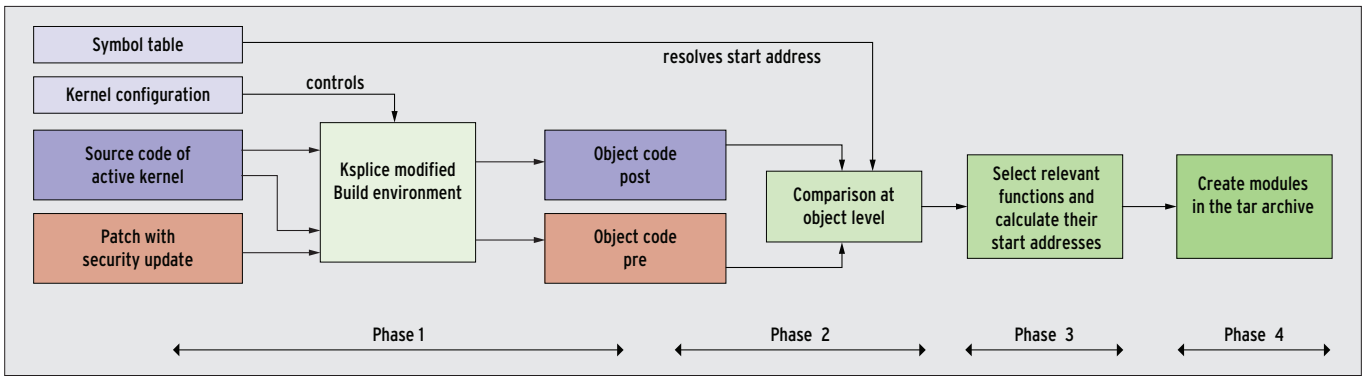

Figure 1: To prepare changes to the running kernel, Ksplice builds two kernel trees (Phase 1), ascertains differences at object code level (Phase 2), optimizes them (Phase 3), and bundles them to create a new module (Phase 4).

ration data, and the symbol table. The best thing is that you do not even need to prepare the running system to use Ksplice. The program can modify any kernel as of version 2.6.8.

Linux distributions offer the kernel source code, or, if you built your own kernel, you will find it in */usr/src*. Distributions typically store the two files *config* and *System.map* in */boot*. On top of this, Ksplice obviously needs a patch file or one or multiple files with changes. The program builds two new kernels: *pre* designates a version of the current system, and *post* designates the kernel after the updates.

Spot the Difference

After building both kernels, Ksplice looks for differences in the object code (see Figure 1). If Ksplice were to analyze the source code directly, it would need to emulate all of the compiler's decisions, which is far too complicated for the task in hand. For this reason, Ksplice uses the GNU BFD library [3] to search for functions in the object code that have changed in the new kernel. Ksplice then adds the new code to modules and inserts trampoline jumps at the start of the original functions that then point to the new versions. When the administrator finally enables the changes, Ksplice loads two kernel modules on the running system that then perform the modifications (see Figure 2).

One critical issue is timing as to when Ksplice is allowed to install the trampoline jumps. Trouble occurs if a kernel thread is currently running one of the functions that needs to be replaced. To avoid this, the program calls *stop_machine_run()* to stop the thread execution because the function creates a high-priority process for each CPU. The Ksplice module now checks to see whether the change candidate contains threads. If it does, the module waits for a while and then retries. This approach will not work for some functions, such as the scheduler, because the scheduler will always be doing something. In that case, Ksplice gives up; in all other cases, it installs the jump addresses. From now on, the Linux kernel executes the patched version of the functions.

Inner Values

Ksplice has to find the right functions and vector addresses in relocatable code. The program is capable of detecting changes in relative jump addresses where the function itself is not affected by a modification, thanks to the new length of the patched function. The kernel typically enters functions implemented in C at the start only. In contrast, the program has to search for the vector in the case of assembler code.

If the program were to use a different compiler to create the *pre* kernel, it could lead to incorrect assumptions about the running system. Ksplice relies on substantial logic to analyze the symbol tables, which many distributions store in the */boot/Symbol.map* file. On top of this, it builds the two kernels with a number of compiler options that assign a separate ELF text segment to each function to make it easier to identify modified, relative jumps.

Because it handles code like a black box, Ksplice can't detect changes in data structures. For example, if a patch adds a new attribute to a data structure or changes its layout, unpredictable issues will occur. Trampoline jumps will typically pick up function pointers, but there is no guarantee Ksplice will handle more complex pointer arithmetic or "creative" typecasting gracefully in each case.

Off Limits

In the documentation, the author emphasizes that he mainly designed the tool for minor security patches and that it is the system administrator's own responsibility to read, understand, and evaluate the patch before applying it.

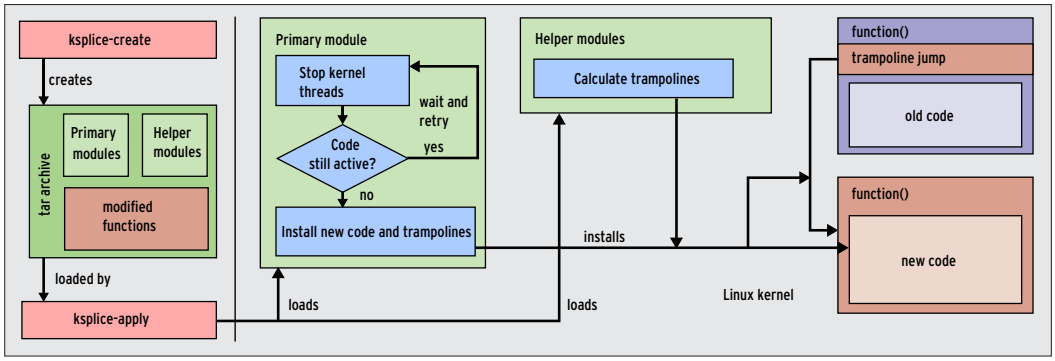

Figure 2: To enable the new functions, Ksplice attempts to write trampoline jumps at the start of the old functions. To do so, the program first stops the processes and makes sure that the code does not use threads.

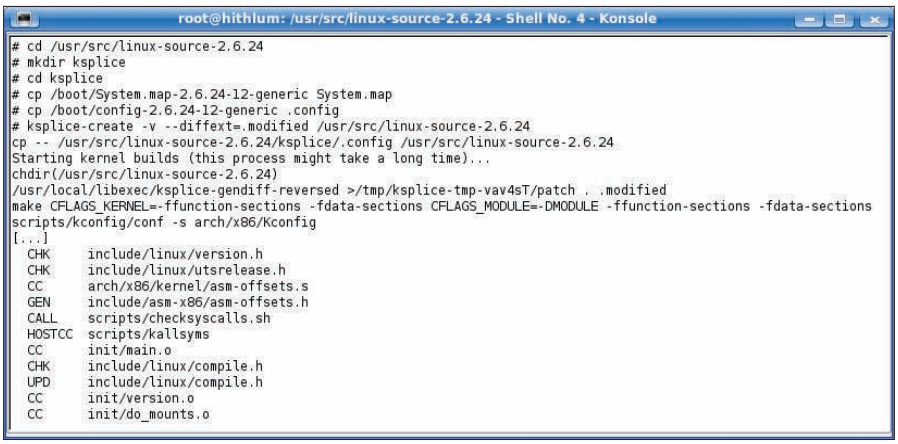

Figure 3: To prepare Ksplice, the administrator types ksplice-create in the kernel source directory and specifies the patch type. Ksplice then builds the old and new kernels and bundles the changes into an update module.

In other words, you need a great deal of kernel expertise to apply the tool; otherwise, the effect might be far more destructive than a bit of reboot downtime.

Because Ksplice cannot make semantic changes to a running kernel, the administrator's pipe dream of measuring uptime in years is just that because most changes between one kernel release and the next add some new functionality.

Production Use

The current 0.8.6 version is available as a tarball with prebuilt binaries or as a source code archive under the GPLv2. Distribution packages do not exist as of this writing. If you build the tool from the source code, you will also need the BFD library, which you can retrieve on Debian or a derivative such as Ubuntu with the following command:

```
sudo aptitude install ➥
binutils-dev
```

Prominent developers, including Andi Kleen, have proposed adding Ksplice to the official kernel as an upstream extension. Kleen hopes this step would mean permanent support for the project, which would, in the long term, lead to an incremental compiler [4]. This would remove the need for developers to completely rebuild the kernel to test and modify patches.

The program itself consists of four Perl scripts that call a number of tools written in C to analyze the object code. *ksplice-create* only needs a path to the directory with the running kernel and details of the patch. One practical feature is that the administrator can specify a change in *diff* format with the *--patch* option or specify a file with the *--diffext* option in which the changes have already been completed. On top of this, the program needs a *ksplice* subdirectory in the kernel tree, where the administrator stores both the kernel configuration and the symbol table (see Figure 3).

Depending on what kind of system you are using, the first phase can take a while because Ksplice needs to build two complete kernels: one in *ksplice/pre* and one in *ksplice/post*. After doing so, the program searches for differences and merges the results to create two kernel modules.

By calling *ksplice-apply*, you can apply the hotfix. The program first loads a module that takes care of trampoline management, then waits for the right moment. When the moment occurs, Ksplice loads the changes into the kernel, executes them, then removes itself to save memory.

Patch Management

Ksplice can also change a patched kernel. To do so, the patches from the first phase must reside in the source code's *pre* tree. *ksplice-create* and *ksplice-apply* take the trampolines into consideration and modify them correspondingly. The same mechanism makes it possible to undo changes by calling *ksplice-undo* because the system "remembers" the vector addresses. *ksplice-view* shows the changes performed by Ksplice.

On his website, Ksplice author Jeffrey Brian Arnold shows another potential application scenario for the tool: debugging the active kernel. If you just want to add a couple of *printk()* calls at various points to view data structures that are otherwise difficult to access, Ksplice gives you a simple approach to injecting them into a running system. However, this approach does not lend itself to more complex applications, for which dynamically loadable modules, Kprobes, or Systemtap are more useful.

Patented Approach?

Developers have pointed out that Microsoft posted a patent application with the US Patent Office (USPO) in December 2002 titled "Patching of In-Use Functions on a Running Computer System." USPO had refused the application, and Microsoft had appealed and posted a whole bunch of additional applications, including one for Efficient Patching (USPO reference 20050257208).

In response to this, half a dozen developers piped up in various forums pointing out that this technology was public knowledge on various platforms from PDP-11 through a state-of-the-art PC long before the software patent application was filed.

Clever Helper for Some Scenarios

Ksplice includes clever mechanisms to support hot kernel updates at the binary level. Despite intelligent code pushing and vector analysis, administrators should be aware that they do need to check manually on a case-by-case basis to determine whether the tool is useful. Ksplice is useful for simple cases, but it is no replacement for a hardware failover solution in situations that require high system availability. ■

INFO

[1] "Ksplice: An Automatic System for Rebootless Linux Kernel Security Updates" by Jeffrey Brian Arnold, Massachusetts Institute of Technology, *http://web.mit.edu/ksplice/doc/ksplice.pdf*

[2] Ksplice download and installation: *http://web.mit.edu/ksplice/*

[3] GNU binutils and BFD library: *http://sourceware.org/binutils/*

[4] Announcement and discussion on the Linux Kernel mailing list: *http://thread.gmane.org/gmane.linux.kernel/669951*