Chapter 12
Katana: Towards Patching as a Runtime Part of the Compiler–Linker–Loader Toolchain

Sergey Bratus
Dartmouth College, USA

Ashwin Ramaswamy
Dartmouth College, USA

James Oakley
Dartmouth College, USA

Sean W. Smith
Dartmouth College, USA

Michael E. Locasto
George Mason University, USA

ABSTRACT

The mechanics of hot patching (the process of upgrading a program while it executes) remain understudied, even though it offers capabilities that act as practical benefits for both consumer and mission-critical systems. A reliable hot patching procedure would serve particularly well by reducing the downtime necessary for critical functionality or security upgrades. However, hot patching also carries the risk—real or perceived—of leaving the system in an inconsistent state, which leads many owners to forgo its benefits as too risky; for systems where availability is critical, this decision may result in leaving systems un-patched and vulnerable. In this paper, the authors present a novel method for hot patching ELF binaries that supports synchronized global data and code updates, and reasoning about the results of applying the hot patch. In this regard, the Patch Object format was developed to encode patches as a special type of ELF re-locatable object file. The authors then built a tool, Katana, which automatically creates these patch objects as a by-product of the standard source build process. Katana also allows an end-user to apply the Patch Objects to a running process.

1. INTRODUCTION

It is somewhat ironic that users and organizations hesitate to apply patches — whose stated purpose is to support availability or reliability — precisely *because* the process of doing so can lead to downtime (both from the patching process itself as well as unanticipated issues with the patch). Periodic reboots in desktop systems — irrespective of the vendor — are at best annoying. Reboots in enterprise environments (e.g., trading, e-commerce, core network systems), even for a few minutes,

DOI: 10.4018/978-1-4666-1580-9.ch012

imply large revenue loss — or require an extensive backup and failover infrastructure with rolling updates to mitigate such loss.

We question whether this *de facto* acceptance of significant downtime and redundant infrastructure should not be abandoned in favor of a reliable hot patching process.

Software, the product of an inherently human process, remains a flawed and incomplete artifact. This reality leads to the uncomfortable inevitability of future fixes, upgrades, and enhancements. Given the way such fixes are currently applied (i.e., patch and reboot), developers accept downtime as a foregone conclusion even as the software is released — and deployers who resist downtime resist the patches.

While patches themselves are a necessity, we believe that the process of *applying* them remains rather crude. First, the target process is terminated; the new binary and corresponding libraries (if any) are then written over the older versions; the system is restarted if necessary; and finally the upgraded application begins execution. Besides the appreciable loss in uptime, all context held by the application is also lost, unless the application had saved its state to persistent storage (Candea & Fox, 2003;Brown & Patterson, 2002) and later restored it (which is expensive to design for, implement, and execute). In the case of mission-critical services, even after a major flaw is unveiled and a patch subsequently created, administrators must choose between security (applying a patch) and availability. This conundrum serves as our motivation for *hot patching,* without restarting the program and losing state and time. We focus on systems, such as those found in the cyber infrastructure for the power grid, which require high availability and which store significant state (that would be lost on a restart).

Challenges of Patching

Requiring and encouraging the adoption of the latest security patches is a matter of common wisdom and prudent policy. It appears, however, that this wisdom is routinely ignored in practice. This disconnect suggests that we should look for the reasons underlying users' hesitancy to apply patches, as these reasons might be due to fundamental technical challenges that are not yet recognized as such. We believe that the current mechanics of applying patches prove to be just such a stumbling block, and we contend that the underlying challenges need to and can be addressed in a fundamental manner *by extending the core elements of the ABI and the executable file format*.

Mission-critical systems seem hardest to patch. They can ill afford downtime, and the owner may be reluctant to patch due to the real or perceived risk of the patch breaking essential functionality. For example, patching a component of a distributed system might lead to a loss or corruption of state for the entire system. An administrator might also suspect that the patch is incompatible with some legacy parts of the system. Even so, the patch may target a latent vulnerability in a software feature that is not now in active use, but also cannot be easily made unreachable via configuration or module unloading. The administrator is forced to accept a particularly thorny choice: inaction holds as much risk as a proactive "responsible" approach. Since the risks of patching must be weighed against those of staying un-patched, we seek to *shift the balance of this decision toward hot patching by making it not only possible, but also less risky in a broad range of circumstances.* We contend that this can only be done through good engineering and making patching a part of the standard toolchain.

Our key observation is that current binary patches, whether "hot" or static, are almost entirely opaque and do not support any form of reasoning about the impact of the patch (short of reverse engineering both the patch and the targeted binary). In particular, it is hard for the software owner to find out whether and how a patch would affect any particular subsystem in any other way than applying the patch on a test system and trying it out, somehow finding a way to faithfully replicate the conditions of the production environment.