Demonstration: Efficient Code Certification for Open Firmware

Matt Stillerman
ATC-NY
33 Thornwood Drive, Suite 500
Ithaca, NY 14850
matt@atc-nycorp.com

Dexter Kozen
Department of Computer Science
Cornell University
Ithaca, New York 14853-7501
kozen@cs.cornell.edu

Abstract

BootSafe is a system for verifying the safety of boot firmware at load time. It employs inexpensive static checks of compiled code, based on the Efficient Code Certification (ECC) technique. We demonstrate a prototype of the BootSafe system that verifies safety of fcode programs for use with Open Firmware compliant boot platforms.

1. Introduction

Malicious boot firmware is a potentially serious problem for critical information systems. Boot firmware runs in a fully privileged mode on bare hardware, prior to the operating system and thus prior to most security mechanisms. It is responsible for detecting the hardware configuration, initializing hardware where necessary, and loading the operating system. Most security mechanisms depend on the integrity of the host operating system, and make use of operating system services. Thus, malicious boot firmware could cause serious harm by operating devices and by corrupting the operating system.

In spite of the evident sensitivity of boot firmware, we believe that it would be relatively easy for an attacker to corrupt it. Parts of the boot firmware on a typical workstation are contributed by many distinct vendors, most of whom have no direct contact with end users. Some parts of the boot program are quite volatile, as the hardware configuration changes, and as the firmware is upgraded.

ATC-NY is engaged in a research effort to develop a novel automated technique for detection of malicious boot firmware. The effort, entitled Efficient Code Certification for Open Firmware, is a DARPA-funded SBIR under the Organically Aware Survivable Information Systems (OASIS) program. We are constructing a prototype of our envisioned system[1].

Our BootSafe verifier will detect potentially harmful firmware with static checks on the compiled code. This process is inexpensive, allowing the verification to occur in every boot cycle. Untrusted boot firmware will be verified as it is loaded, prior to executing it.

We employ a form of verification called Efficient Code Certification[3]. The verification process depends on firmware being compiled with a certifying compiler, which produces particularly well-structured and annotated code. The verifier uses this structure and annotation to, in effect, check an implicit proof of safety. Proof checking (as opposed to proof construction) makes verification inexpensive.

End users only need to trust the verifier in order to have confidence in the safety of the compiled code. They do not need to trust the compiler, nor do they need access to the firmware source code. In addition to the verifier and compiler, the third principal component of BootSafe is a runtime support library.

Our BootSafe prototype targets Open Firmware[2] compliant boot platforms. Open Firmware is a mature and widely used standard for boot firmware. For instance, both Sun Microsystems and Apple use boot firmware that conforms to the standard. The most salient feature of Open Firmware, for our project, is that it includes an interpreter (or virtual machine) for fcode. Fcode is a lightly compiled form of the Forth programming language.

In Open Firmware, fcode is used for some, or all, of the boot program. At a minimum, it is used for modules representing boot-time device drivers for peripheral devices. Fcode device drivers, supplied by a wide range of relatively anonymous vendors, pose a significant risk of introducing malicious code into the boot program. In this effort we are focused on detection of malicious fcode.

2 Demonstration

We are demonstrating a prototype of the BootSafe system in three phases of operation. These correspond to the three major subsystems in our architecture:
• Compiler
• Verifier
• Runtime support library

Vendors of peripheral devices will use the BootSafe compiler (J2F) to compile Java source code into executable fcode device drivers that are shipped with their products. In Open Firmware compliant systems, boot-time device drivers are stored in ROM on the device.

Platform vendors will incorporate the BootSafe verifier and the runtime support library into their main boot programs. As the computer boots, the peripheral device drivers are loaded into main memory and checked by the verifier for compliance with the security policy. (If a device driver fails this check, the main program will take corrective action that is outside the scope of BootSafe.)

Drivers are then linked against the runtime support library. They are used by the main boot program to initialize and operate devices.

2.1 Compilation and Execution

We exhibit Java source code for an Open Firmware device driver. (Currently, such drivers are typically written in Forth.) This source is compiled into fcode with our BootSafe compiler.

We use SmartFirmware[4], running in a simulation mode, to exercise the device driver. SmartFirmware is a commercially available implementation of Open Firmware. SmartFirmware "boots" and loads the driver, linking it with the runtime support library. The corresponding device is emulated. We may then operate the device through SmartFirmware.

2.2 Verification

The verifier is exhibited as a stand-alone program. Our intention is that the verifier should be integrated with the main Open Firmware boot program, where it will be most effective.

The device driver compiled in the first step will pass the verifier check. We also show a selection of malformed fcode programs that could be the means of an attack. These "hacks" are detected by the verifier.

2.3 Runtime Support

The runtime support library provides an API so that programs written in Java can access Open Firmware services and data structures. The standard API is in Forth, so some mediation is necessary.

We illustrate how the Java API will enable device drivers to be easily written, and more easily checked for safety. We provide base classes that encapsulate much of the messy code that is common to all similar devices. We then require the use of these base classes (and employ other aspects of Java modularity). These constraints are designed to rule out specific forms of malicious code.

References