Defeating Kernel Driver Purifier

Jidong Xiao[1](✉), Hai Huang[2], and Haining Wang[3]

[1] College of William and Mary, Williamsburg, VA 23185, USA
jxiao@email.wm.edu
[2] IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA
[3] University of Delaware, Newark, DE 19716, USA

Abstract. Kernel driver purification is a technique used for detecting and eliminating malicious code embedded in kernel drivers. Ideally, only the benign functionalities remain after purification. As many kernel drivers are distributed in binary format, a kernel driver purifier is effective against existing kernel rootkits. However, in this paper, we demonstrate that an attacker is able to defeat such purification mechanisms through two different approaches: (1) by exploiting self-checksummed code or (2) by avoiding calling kernel APIs. Both approaches would allow arbitrary code to be injected into a kernel driver. Based on the two proposed offensive schemes, we implement prototypes of both types of rootkits and validate their efficacy through real experiments. Our evaluation results show that the proposed rootkits can defeat the current purification techniques. Moreover, these rootkits retain the same functionalities as those of real world rootkits, and only incur negligible performance overhead.

1 Introduction

Modern operating systems are often divided into a base kernel and various loadable kernel modules. Kernel drivers are often loaded into the kernel space as modules. The ability to quickly load and unload these modules makes driver upgrade effortless, as the new code can take an immediate effect without rebooting the machine. While the base kernel is trusted, kernel drivers are sometimes released by third-party vendors (i.e., untrusted) in binary format. This creates a problem as it is much more difficult to detect malicious code at the binary level than at the source level. Therefore, kernel drivers have been heavily exploited for hosting malicious code in the past. Sony's infamous XCP rootkit in 2005 [1,22] and its USB device driver rootkit in 2007 [18] have exemplified this risk. In addition, kernel drivers, which constitute 70% of modern operating system's code base [16],are a significant source of software bugs [7,10], making them substantially more vulnerable to various malicious attacks than the base kernel.

During an attack, once an attacker gains root access, rootkits are then installed to hide their track and provide backdoor access. Rootkits normally hook to the kernel and modify its data structures such as system call table, task list, interrupt descriptor table, and virtual file system handlers. Rootkits can be either installed as a separate kernel module, or injected into an existing kernel module. To protect against rootkits, different defense mechanisms have

B. Thuraisingham et al. (Eds.): SecureComm 2015, LNICST 164, pp. 116–134, 2015.
DOI: 10.1007/978-3-319-28865-9_7

been proposed and can be categorized into two basic approaches: kernel rootkit detection and kernel module isolation. In the former, various detection frameworks are created using either an extra device to monitor system memory [23] or virtual machine introspection techniques [9,15]. And in the latter, strict isolation techniques are introduced to further isolate kernel modules from the base kernel [5,30].

While the idea of enhancing the isolation of kernel drivers has been extensively studied in the past, it has not yet been widely adopted by mainstream operating systems. One of the key reasons is that it involves too much re-implementation effort. Instead of isolating kernel drivers, safeguarding a kernel driver itself looks more promising. As kernel drivers run at the same privilege level as the base kernel, one can achieve this goal by detecting and eliminating malicious code from kernel drivers before they are loaded into the kernel space. This technique is called kernel driver purification. Based on this design principle, Gu et al. [11] proposed and implemented a kernel driver purification framework, which aims to detect malicious/undesirable logic in a kernel driver and eliminate it without impairing the driver's normal functionalities. Their experimental results demonstrate that this technique can purify kernel drivers infected by various real world rootkits. However, we observe that there are two approaches which attackers can employ to defeat such a technique. The first approach uses self-checksum code to protect malicious kernel API calls, and the second approach is to simply avoid using kernel API calls altogether when writing a rootkit. We show that both approaches can effectively defeat current kernel driver purifiers.

The major contributions of our work are summarized as follows:

- We first present a self-checksum based rootkit that is able to evade the detection of current kernel driver purifiers. While self-checksum has long been proposed as a way to protect benign programs, as far as we know, we are the first to use it for hiding kernel rootkits. We also develop a compiler level tool, with which, attackers can automatically re-write existing rootkits and convert them into self-checksum based variants that are resistant to kernel driver purifiers.
- We present another approach of creating a more stealthy rootkit, which avoids using kernel API calls. While our first approach attempts to protect malicious kernel API calls from being removed by kernel driver purifiers, this new type of rootkit demonstrates that most kernel API calls can be avoided, and thus making the kernel driver purifier completely ineffective.
- We evaluate the functionality and performance of both rootkits. Our experimental results show that the presented rootkits maintain the same set of functionalities as most real world rootkits have and only incur minor performance overhead.

2 Background

Kernel drivers have always been a major source of kernel bugs and vulnerabilities, and improving their reliability has drawn significant attentions from the