Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

# Robust Linux memory acquisition with minimal target impact

Johannes Stüttgen [a,*], Michael Cohen [b]

[a] Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstraße 3, 91058 Erlangen, Germany
[b] Google Inc., Brandschenkestrasse 110, Zurich, Switzerland

*Keywords:*
Memory forensics
Linux memory acquisition
Memory acquisition
Incident response
Live forensics

A B S T R A C T

Software based Memory acquisition on modern systems typically requires the insertion of a kernel module into the running kernel. On Linux, kernel modules must be compiled against the exact version of kernel headers and the exact kernel configuration used to build the currently executing kernel. This makes Linux memory acquisition significantly more complex in practice, than on other platforms due to the number of variations of kernel versions and configurations, especially when responding to incidents. The Linux kernel maintains a checksum of kernel version and will generally refuse to load a module which was compiled against a different kernel version. Although there are some techniques to override this check, there is an inherent danger leading to an unstable kernel and possible kernel crashes. This paper presents a novel technique to safely load a pre-compiled kernel module for acquisition on a wide range of Linux kernel versions and configuration. Our technique injects a minimal acquisition module (parasite) into another valid kernel module (host) already found on the target system. The resulting combined module is then relinked in such a way as to grant code execution and control over vital data structures to the acquisition code, whilst the host module remains dormant during runtime.

## Introduction

Memory analysis has rapidly become a very powerful tool in the arsenal of incident responders and forensic examiners. Frameworks such as Volatility (Walters, 2007) or Second Look (Raytheon Pikewerks, 2013) allow an in-depth analysis of operating system data structures and can be used to gain a thorough understanding on a live systems state like running processes, network connections and mapped files.

For this to work it is necessary to acquire a memory image. While multiple methods to do this using physical access to the hardware exist (Carrier and Grand, 2004; Boileau, 2006), physical access is often not available in an incident response scenario. Thus, a software based approach is sometimes the only viable option. Because current operating systems operate in protected mode for security and safety reasons, acquisition of the entire physical address space can only be achieved in system mode. For Linux this typically requires the injection of a Linux kernel module into the running kernel. Since the Linux kernel checks modules for having the correct version and checksums before loading, the kernel will typically refuse to load a kernel module pre-compiled on a different kernel version or configuration to the one being acquired. This check is necessary since the struct layout of internal kernel data structures varies between versions and configurations, and loading an incompatible kernel version will result in kernel instability and a potential crash.

For incident response this requirement makes memory acquisition problematic, since often responders do not

* Corresponding author.
*E-mail addresses:* johannes.stuettgen@cs.fau.de (J. Stüttgen), scudette@google.com (M. Cohen).

http://dx.doi.org/10.1016/j.diin.2014.03.014

know in advance which kernel version they will need to acquire. It is not always possible to compile the kernel module on the acquired system, which may not even have compilers or kernel headers installed.

Some Linux memory acquisition solutions aim to solve this problem by maintaining a vast library of kernel modules for every possible distribution and kernel version (Raytheon Pikewerks, 2013). While this works well as long as the specific kernel is available in the library, it is hard to maintain and can not cover cases where the kernel has been custom compiled or just is not common enough to award a place in the library. This is especially the case on mobile phones. Often phone vendors might publish the kernel version they used, but the configuration and details on all vendor specific patches are often not known, severely impeding memory acquisition (Sylve et al., 2012).

Rootkit authors also have encountered the same problem when trying to infect kernels where the build environment is not available. Recent work for Android shows that while it is trivial to bypass module version checking, it is still a hard problem to identify struct layout in unknown binary kernels (You, 2012). In the Android case this problem is solved by restricting dependencies to very few kernel symbols and reverse engineering their data structures on the fly using heuristics (You, 2012).

A solution for data structure layout detection could be live disassembly of functions which are known to be stable and use certain members in these structs. Recent work showed that it's possible to dynamically determine the offsets of particular members in certain structs used in memory management, file I/O and the socket API (Case et al., 2010).

Kernel integrity monitoring systems also have similar problems, as they have to monitor dynamic data and need to infer its type and structure to analyze it. Since this data layout changes with kernel version, these systems need to infer its data layout from external sources. The KOP (Carbone et al., 2009) and MAS (Weidong et al., 2012) frameworks, are exemplary systems designed to monitor integrity of dynamic kernel data structures. Their approach involves statically analyzing the kernel source code and debug symbols to infer type information for dynamic data. However, they rely on the kernel source-code and debug symbols for the exact running kernel being available in advance, which is exactly the dependency we can not guarantee in the incident response scenario.

*Contributions*. We have developed a method to inject a parasite kernel module into an already existing host kernel module as found on the running system. Most modern kernels have a large number of legitimate kernel modules, compiled specifically for the running kernel, already present on the system. Our approach locates a suitable existing kernel module (Host Module), injects a new kernel module into it (Parasite module) and loads the combined module into ring 0.

The resulting modified kernel module is fully compatible with the running kernel. All data structures accessed by the kernel are taken from the Host module, and were in fact compiled with compatible kernel headers and config options. However, control flow is diverted from the Host module to the Parasite module, by modifying static linking information. This allows the parasite module's code to use the hosts' structs for communication with kernel APIs.

## Anatomy of a Linux kernel module

Linux kernel modules are relocatable ELF object files and not an executable. The obvious difference is that executable ELF files are processed by a loader, while relocatable objects are intended for a linker.

The loader relies on the ELF Program Headers to identify the file layout and decide which parts to map into memory with which permissions. The linker instead relies on ELF section headers for this, with special sections containing symbol string and relocation tables, to identify and resolve inter-section symbol references (Fig. 1).

Dependencies on other objects in an ELF executable are resolved by dynamic linking. In this process, external symbols are referenced through the Global Offset Table (.got) and Procedure Linkage Table (.got.plt), and resolved by the dynamic linker at runtime (Fig. 2).

In contrast to this, relocatable ELF objects are statically linked using relocations. Each section with references to symbols in other sections or objects has a corresponding relocation table. Entries in these tables contain information on the specific symbol referenced, and how to patch a specific code or data reference with the final address of the symbol after it has been relocated.

One or more of these relocatable objects can be linked together by placing them into their final position in the final executable or address space, after which the linker applies all relocations to patch the now final references directly into the code.

In the context of the Linux kernel this means that loading a kernel module is actually the same thing as linking an executable, but with the executable being the running kernel image.

### *How is a LKM loaded and linked*

The actual loading process of a kernel module can be characterized by four steps, which begin in user space (Fig. 3):

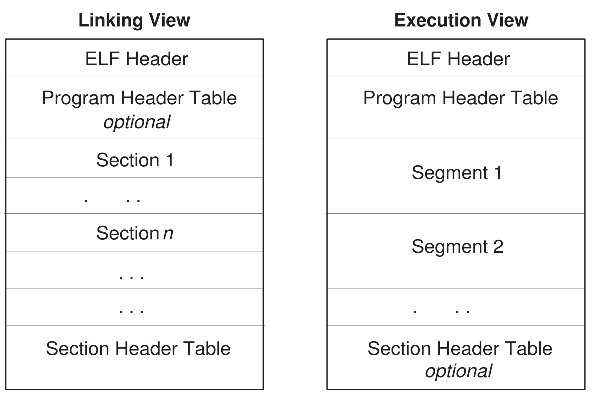

**Fig. 1.** ELF file layout (TIS Committee, 1995).