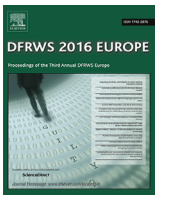
Digital Investigation 16 (2016) S11–S24

Contents lists available at ScienceDirect

Digital Investigation

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



DFRWS 2016 Europe — Proceedings of the Third Annual DFRWS Europe

Automatic profile generation for live Linux Memory analysis

Arkadiusz Socała [a, *], Michael Cohen [b]

[a] University of Warsaw, Krakowskie Przedmieście 26/28, Warsaw, Poland
[b] Google Inc., Brandschenkestrasse 110, Zurich, Switzerland

Keywords:
Memory analysis
Incident response
Memory Forensics
Compilers
Reverse Engineering
Malware
Linux Forensics
Digital Forensic Triaging

A B S T R A C T

Live Memory analysis on the Linux platform has traditionally been difficult to perform. Memory analysis requires precise knowledge of struct layout information in memory, usually obtained through debugging symbols generated at compile time. The Linux kernel is however, highly configurable, implying that debugging information is rarely applicable to systems other than the ones that generated it. For incident response applications, obtaining the relevant debugging information is currently a slow and manual process, limiting its usefulness in rapid triaging. We have developed a tool dubbed, the *Layout Expert* which is able to calculate memory layout of critical kernel structures at runtime on the target system without requiring extra tools, such as the compiler tool-chain to be pre-installed. Our approach specifically addresses the need to adapt the generated profile to customized Linux kernels – an important first step towards a general version agnostic system. Our system is completely self sufficient and allows a live analysis tool to operate automatically on the target system. The *layout expert* operates in two phases: First it pre-parses the kernel source code into a *preprocessor AST* (*Pre-AST*) which is trimmed and stored as a data file in the analysis tool's distribution. When running on the target system, the running system configuration is used to resolve the Pre-AST into a C-AST, and combined with a pre-calculated layout model. The result is a running system specific profile with precise struct layout information. We evaluate the effectiveness of the Layout Expert in producing profiles for analysis of two very differently configured kernels. The produced profiles can be used to analyze the live memory through the */proc/kcore* device without resorting to local or remote compilers. We finally consider future applications of this technique, such as memory acquisition.

Introduction

In recent times, Memory analysis has been used effectively in the wider context of digital forensics, and malware detection (Ligh et al., 2014). In essence, memory analysis strives to make sense of a computer's memory image – an exact copy of the physical memory used by a running system. As the size of physical memory increases, especially on large servers, memory analysis based triaging techniques are becoming more important (Moser and Cohen, 2013).

At first look, physical memory might appear as a large amorphous and unstructured collection of data. In fact, physical memory is used by the running software to store program state in a highly structured manner. The programmer employs logical constructs such as *C structs* to collect related data into logical units, representing abstract data types. The compiler than ensures that this struct is laid out in memory in a consistent way, and generates code to access various members of the struct according to this layout.

* Corresponding author.
E-mail address: as277575@students.mimuw.edu.pl (A. Socała).

http://dx.doi.org/10.1016/j.diin.2016.01.004

In order to successfully extract high level information from a memory image, one must extract and interpret the abstract *struct* objects that the program handles from the amorphous physical memory. In order to do this, one must have an accurate model of the physical layout of the structs and their individual member's data types.

Earlier memory analysis solutions relied on hand constructed layout models for each struct, obtained by trial and error (Schuster and Andreas, 2007). However, struct layouts change frequently between released versions (Cohen, 2015b), and the large number of structs of interest makes such maintenance difficult.

For open source operating systems, one might be tempted to examine the source code and from the source code, theorize the precise memory layout for each struct. However (as explained in detail in Section Layout model), such an analysis is not practical without intimate knowledge of the compiler's *layout model*. In practice there are many edge cases which are difficult to predict: For example, the compiler may add padding to ensure alignment of various struct members under different conditions.

In order to support debugging tools, which must also extract meaningful information from the program's memory, compilers typically emit the layout models for each struct used in a program into some kind of debugging stream, for example a PDB file, or DWARF streams (DWARF Debugging Information Format Committee, 2010).

The Volatility Memory analysis Framework (The Volatility Foundation, 2014) was the first open source memory analysis framework able to utilize information derived from debugging streams in order to analyze memory images from multiple versions of an operating system. In the Volatility framework, debugging information is converted into a *profile* specific to a particular version of the operating system. These profiles are embedded inside the tool and allow the user to specify which version of the operating system the image originated from.

On Microsoft Windows systems, debugging symbols are stored in external PDB files which may be downloaded from a central *symbol server* on demand (Okolica and Peterson, 2010). The Rekall memory analysis framework (The Rekall Team, 2014) is able to download debugging symbols for unknown kernels directly from the Microsoft debugging server. This feature is useful when operating in live mode since Rekall can parse the PDB files directly into *profiles* which are used to analyze the running system.

Unfortunately, memory analysis on Linux systems presents some practical challenges. Unlike Windows, the Linux Kernel is typically not compiled with debugging information (such as DWARF streams), nor is debugging information typically available on demand from a debug server. In order to obtain debugging information, one must recompile the kernel, or some part of the kernel (e.g. a kernel module) specifically with debug flags enabled. On a Debian based system, this also requires that a *linux-header* package be installed, containing kernel header files as well as important files that were generated during the kernel compilation step (e.g. *Modules.symvers* file) before a kernel module can be built (Hertzog and Mas, 2014). In practice, the kernel-header package for a custom compiled kernel is often not available or was never even created in the first place. At best, incident responders must scramble to identify the correct kernel-header package for the running kernel on the target system and hope that it matches.

Another complication is the high level of configurability of the Linux kernel. During the kernel build process, users may specify a large number of *configuration options* through the kernel's configuration system. These options affect the kernel build process by defining a large number of C preprocessing macros.

The Linux kernel source uses preprocessing macros heavily to customize the operation of the kernel itself – and in particular the kernel tends to include certain fields into critical structs only if certain functionality is enabled by the user. For example consider the code in Fig. 1 which shows the definition of *task_struct* – a critical struct maintaining information about running processes.

As can be seen, some of the struct members are only included if certain configuration parameters are set. For example, the *sched_task_group* pointer only exists when the kernel is compiled with support for task group scheduling – an optional feature of the Linux kernel. Similarly *CONFIG_SMP* controls the inclusion of several fields used by multiprocessing systems.

When the compiler generates the abstract struct layout model, it must allocate a position for every struct member in memory, sufficient to accommodate the size of the member, its alignment requirements and the alignment of members around it. Clearly if certain fields are not included in the struct definition (e.g. if the feature they implement is not chosen by the user), the compiler will not reserve any space for them, and therefore struct members that appear later in the struct definition will be located at different positions in memory.

The main problem that memory analysis tools encounter when parsing the Linux kernel's memory, is that the configuration of the kernel controls the resulting kernel structures' layout model, but this configuration is not constant. Since Linux users and distributions are free to reconfigure and recompile their kernels at any time, each specific kernel used in a given memory image can have vastly different configuration and therefore layouts (This is contrasted with commercial operating systems, such as Windows or OSX, where only a small number of officially released versions are found in the wild).

One solution to this problem is to maintain a large repository of common kernel configurations. The *Secondlook* product (Secondlook, 2015) maintain a large repository of profiles for every release of major distributions (e.g. Ubuntu, Redhat etc). Although this repository is large (supposedly over 14,000 profiles), if the user has recompiled the kernel and changed some configuration options themselves, the correct profile will not be found in the repository. A complete repository will have to account for every combination of configuration options and would therefore be impractically large.

A different approach, as taken by some memory analysis frameworks (The Rekall Team, 2014; The Volatility Foundation, 2014) requires the user to specifically build a profile for each target kernel in advance prior to analysis. The usual procedure is to obtain the kernel-headers package and use the target kernel's configuration to compile a