DIGITAL INVESTIGATION 7 (2010) S32–S40

available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

Dynamic recreation of kernel data structures for live forensics

Andrew Case [a], *Lodovico Marziale* [a], *Golden G. Richard, III* [b,*]

[a] *Digital Forensics Solutions, LLC, United States*

[b] *Dept. of Computer Science, University of New Orleans, Lakefront Campus, New Orleans, LA 70148, United States*

ABSTRACT

The role of live forensics in digital forensic investigations has become vital due to the importance of volatile data such as encryption keys, network activity, currently running processes, in memory only malware, and other key pieces of data that are lost when a device is powered down. While the technology to perform the first steps of a live investigation, physical memory collection and preservation, is available, the tools for completing the remaining steps remain incomplete. First-generation memory analyzers performed simple string and regular expression operations on the memory dump to locate data such as passwords, credit card numbers, fragments of chat conversations, and social security numbers. A more in-depth analysis can reveal information such as running processes, networking information, open file data, loaded kernel modules, and other critical information that can be used to gain insight into activity occurring on the machine when a memory acquisition occurred. To be useful, tools for performing this in-depth analysis must support a wide range of operating system versions with minimum configuration. Current live forensics tools are generally limited to a single kernel version, a very restricted set of closely related versions, or require substantial manual intervention.

This paper describes techniques developed to allow automatic adaptation of memory analysis tools to a wide range of kernel versions. Dynamic reconstruction of kernel data structures is obtained by analyzing the memory dump for the instructions that reference needed kernel structure members. The ability to dynamically recreate C structures used within the kernel allows for a large amount of information to be obtained and processed. Currently, this capability is used within a tool called RAMPARSER that is able to simulate commands such as ps and netstat as if an investigator were sitting at the machine at the time of the memory acquisition. Other applications of the developed capabilities include kernel-level malware detection, recovery of processes memory and file mappings, and other areas of forensics interest.

1. Introduction

Digital forensics comprises the set of techniques to recover, preserve, and examine digital evidence on or transmitted by digital devices and has applications in a number of important areas, including investigation of child exploitation, identity theft, counter-terrorism, intellectual property disputes, and more. Digital forensics tools typically examine and interpret data at a low level, because data of evidentiary value may have been deleted, partially overwritten, obfuscated, or corrupted. Furthermore, a forensics target must typically be examined exhaustively to reveal both incriminating and exculpatory

* *Corresponding author*. Tel.: +1 504 280 6045.
E-mail address: golden@cs.uno.edu (G.G. Richard III).

1742-2876/$ – see front matter © 2010 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2010.05.005

evidence. Most traditional digital forensics tools and techniques have focused on "dead" analysis, typically bit-perfect copies of storage media. From these copies, deleted files or file fragments are recovered, patterns of file access are determined, past web browsing activity is observed, etc. A number of factors have contributed to an increasing interest in "live" forensics, however, where the machine under investigation continues to run while forensic evidence is collected. These factors include a huge increase in the size of forensics targets (with commodity hard drives now exceeding 1 TB for under $100), increasing case backlogs as more criminal activity involves the use of computer systems, and the need to turn around cases very rapidly to counter acts of terrorism or other criminal activity where lives or property may be in imminent danger. In addition, a live forensics investigation can reveal a substantial amount of volatile evidence that would be lost if a traditional "pull the plug and makes copies of the hard drives" mentality prevailed. This evidence includes lists of running processes, network connections, fragments of volatile data such as chat messages, and keying material for drive encryption.

Early live forensics efforts typically involved running a number of statically linked binaries on the forensic target (e.g., *ls*, *ps*, *lsmod*, *lsof*, etc. under Linux) and capturing the output of these commands for later consumption. A physical memory dump was also captured, but analysis of the physical memory dump was often limited to simple string searches. In order to gather and analyze all the information contained within a physical memory dump, a complete understanding of the data structures and algorithms used by the original operating system must be understood. Under operating systems such as Linux, where new kernel versions are released frequently and distributions ship their own custom kernels, it is infeasible to write a tool that has knowledge of all kernel variations. Even in instances where the investigator has the complete source code for the kernel being investigated, there may not be enough information to accurately model the structures needed. While there are numerous problems faced when taking a source code based approach, the largest problem is conditional compilation during the kernel building process. Inclusion or exclusion of a kernel configuration option can cause the insertion or removal of a large number of members in key structures, which are required in the processing of an associated memory image. For example, inside the 2.6.27 Linux kernel's struct *task_struct*, which is the C structure used to track per process information, of the over one hundred members, at least forty depend on user-controlled compile time compilation options. This creates greatly different in-memory layouts of the structure between kernel versions, and breaks tools which rely on exact offsets of structure members to perform their work.

The research presented in this paper was conducted with the goal of creating a system that would be able to fully analyze Linux 2.6 memory dumps without relying on specific knowledge of kernel versions and distributions. Unlike other memory analysis systems which rely on information that is not always attainable, the presented system only needs the analyzed kernel's *System.map* file and a memory dump. To be fully kernel version independent, the memory analysis system depends on the ability to accurately and dynamically build representations of needed kernel structures. In the C programming language, when structure members are accessed, an offset-based memory deference is performed. By understanding the assembler instructions used to perform these references, the offset of a targeted structure member can be obtained dynamically, directly from the memory dump. By gathering the offsets of relevant structure members for important kernel structures, our tools can then access structure members in the memory dump in the same manner as the kernel does at runtime. This allows for replication of kernel functionality and algorithms which are needed to recreate the state of the machine at the time the memory dump was taken.

The remainder of this paper describes the techniques we developed to dynamically reconstruct the Linux kernel C structures needed to analyze a physical memory dump under Linux. The design of the system that allows it to be both architecture and kernel version independent is detailed and techniques for acquisition of key structure offsets explained. The culmination of this work is the RAMPARSER system, which can dynamically model kernel structures and provides an easy-to-use interface to view data in a Linux physical memory dump.

2. Related work

Recent interest in more powerful techniques for live forensics analysis has resulted in a number of proposed systems. To date, more attention has been made to machines running versions of Microsoft Windows than to Unix-based operating systems such as Linux, the BSD variants, and Mac OS X. The DFRWS 2005 memory analysis challenge was a primary catalyst for much of this work. A novel but fragile approach called BodySnatcher (Schatz, 2007) involves injection of a small, forensic OS that subverts and halts the running OS (a version of Windows) to allow analysis. The Volatility framework (Petroni et al., 2006, https://www.volatilesystems.com/default/volatility) extracts information from Windows XP SP2 and SP3 memory dumps, including a list of running processes, open network connections, loaded DLLs, and Virtual Address Descriptor (VAD) information. PyFlag (Cohen, 2008) is a correlation system for digital forensics investigations that now integrates with the Volatility framework to perform live forensics analysis of Windows machines, and on a more limited basis, Linux machines. The knTTools (http://www.gmgsystemsinc.com/knttools) dump information about processes, threads, access tokens, the handle table, and other OS structures from a Windows memory dump. MemParser (http://sourceforge.net/projects/memparser) performs similar functions, with cross-referencing used to detect hidden objects. Schuster's PTfinder tools (Schuster, 2006a,b) take a different approach from most of the previous tools and instead of walking OS structures, attempt to carve objects that represent threads and processes directly from the memory dump. This allows hidden processes to be more easily discovered and can also reveal information about recently terminated processes. Kornblum discusses some of the challenging issues in analysis of Windows physical memory dumps (Kornblum, 2006, 2007).

While there has been work in deep, live forensics analysis of Linux, all of the released tools rely on specific kernel versions and structures. iDetect (Burdach M, http://forensic.seccure.net/tools/idetect.tar.gz) is a simple proof of concept tool that parses 2.4–series memory dumps and enumerates page frames, discovers user mode processes, and provides detailed information about process descriptors. Urrea (2006) discusses many of the relevant OS structures that must be parsed to extract digital evidence from Linux memory dumps. Crash (Anderson, 2008) is a tool which allows extensive analysis of kernel core dumps and live systems. It has the ability to recover processes, memory mapped and opened files, networking related information such as routing tables and ARP cache, and a much more. It uses the same strategy as RAMPARSER, by first gathering the offsets of all needed structures and then performing analysis. Unlike RAMPARSER, it relies extensively on debugging information within the core dumps, and only works on Redhat-based Linux distributions. (Case et al., 2008) describes the FACE correlation engine that allows deep analysis of Linux memory dumps across a small range of 2.6 kernel versions. Beyond the capabilities mentioned in the previous related work, it also has the ability to recover queued packets and it can perform virtual to physical translation for recovery of userland data. Its usability in the field is limited because of the small number of kernel versions that are supported, a flaw that is addressed in the current work.

3. RAMPARSER

RAMPARSER was created to address some of the deficiencies in existing Linux memory analysis technology, the most important of which is portability. It has the capability of analyzing physical memory dumps for a wide range of kernel versions across multiple hardware architectures. It accomplishes this goal by utilizing knowledge of core kernel functions that remain nearly identical across a variety of kernel versions and which access kernel structures that RAMPARSER needs to model to support powerful live forensics queries. These core kernel versions are reverse engineered on-the-spot to generate accurate data structure models.

3.1. Testbed

RAMPARSER was developed using a testbed of a wide range of kernels across three different architectures. Linux kernel versions varying from 2.6.9 to 2.6.27 were tested to ensure that large changes in data structure layouts and kernel algorithms did not affect the validity of reported results. A sample of the kernels we used for testing during the development of RAMPARSER appears in Table 1.

After development was complete, other kernels were successfully tested against the original code. These are listed in Table 2.

3.2. Review of functionality

Once a memory dump is analyzed to build models for kernel structures, RAMPARSER provides the following functionality: 'ps' command emulation, 'netstat' command emulation, per-process emulation of the contents of */proc/<pid>/fd*, and per-process emulation of */proc/<pid>/maps* contents. RAMPARSER is written in Python and other live forensics queries are easily added.

Table 1 – Sample of kernel versions used for testing during development of RAMPARSER.

Architecture	Distribution	Kernel version
x86	Ubuntu 8.10	2.6.27-7-generic SMP
x86	Debian 4.0	2.6.18-6-686 SMP
x86	CentOS 4.7	2.6.9-78.EL
x86_64	Ubuntu 8.10	2.6.27-7-generic SMP
x86_64	Debian 4.0	2.6.18-6-686 SMP
PS3 (PPC64)	Ubuntu 8.10	2.6.25-2-powerpc64-smp

When RAMPARSER is executed, it starts by gathering all the structures offsets needed to analyze the memory image. After the offsets are collected, information is gathered to support the features discussed above and this information is stored in a SQLite database. Upon completion of the script the database file contains all the information about the memory image that RAMPARSER is capable of retrieving.

An output module queries the database and displays requested information. The following illustrates output from the 'ps' emulation module:

NAME	UID	GID	PID
swapper	0	0	0
init	0	0	1
kthreadd	0	0	2
migration/0	0	0	3
ksoftirqd/0	0	0	4
watchdog/0	0	0	5
events/0	0	0	6
mixer_applet2	1000	1000	5998
evolution-alarm	1000	1000	6010
update-notifier	1000	1000	6016
tracker-applet	1000	1000	6018
nm-applet	1000	1000	6019
python	1000	1000	6020
trackerd	1000	1000	6021

With the information contained in the database, existing tools can integrate the results of RAMPARSER into their investigation suites. We are currently expanding the scope of live forensics queries supported by RAMPARSER.

3.3. Supported processor architectures

RAMPARSER currently supports the x86, x86_64, and PPC64 architectures. These architectures were chosen due to their

Table 2 – Post-development test kernels.

Architecture	Distribution	Kernel Version
x86	Debian 4.0	2.6.26-2-686 SMP
PS3 (PPC64)	Ubuntu 8.10	2.6.27 vanilla

wide-spread availability, with the Intel x86 and x86_64 being the most commonly available platforms. In addition to these Intel architectures, we targeted the PlayStation 3 (PS3), which is based on a PPC64 processor architecture and which in versions predating the "PS3 Slim" supports Linux natively. The combination of these architectures required that all code was both word size and endian independent. x86 has a 32 bit word size and little endian byte ordering, x86_64 has a 64 bit word size and little endian ordering, and PPC64 has a 64 bit word size and big endian ordering.

In order to work across multiple architectures, RAMPARSER requires a small amount of unique information about each architecture, including the kernel's virtual loading address and the page offset address. These addresses are used to convert virtual addresses to physical addresses. Since RAMPARSER analyzes physical memory dumps, the physical address obtained from the conversion is the offset into the file where the data is stored. On x86 and PPC64 these addresses are the same, but on x86_64 different addresses are used.

Under Linux, the *System.map* file is created at kernel compilation time and contains the address, type of symbol, and name for all defined symbols within the kernel. RAMPARSER uses this file to obtain the virtual addresses of many functions and data structures it needs to analyze to correctly model kernel data structures. Parsing *System.map* is relatively straightforward, but RAMPARSER does needs to accommodate some eccentricities in the format of *System.map*, such as the fact that some symbols for PPC64 are prepended with a period.

Upon initialization of the Linux operating system, a process named "swapper" with a process ID of 0 is created. The page global directory (pgd) used for this process is statically initialized and filled with entries that map the kernel. The address of this symbol is stored within *System.map*, but the name is different between architectures. It is named *swapper_pg_dir* on both x86 and PPC64, but named *init_level4_pgt* on x86_64.

The algorithms described in Section 3.4 depend on the ability to locate assembler instructions that access needed kernel structure members. To facilitate this, each architecture module in RAMPARSER contains the opcodes for the instructions that perform loading and storing of data and comparisons. The modules also contain the possible sizes and combinations of encoded structure offsets within the instructions. For instance both x86 and x86_64 can have either one byte or four byte offsets, while PPC64 uses two byte offsets.

Determining when a one or four byte offset is being used on Intel machines requires a precise understanding of the instruction format. Since RAMPARSER does not currently incorporate a complete disassembler to support its reverse engineering efforts, the offset size is determined by examining the size of the offset gathered by treating it as both a one byte and four byte offset and sanity checking the results. This method has correctly worked on all kernels and versions tested.

With the previously described information contained within the classes for each architecture, RAMPARSER's functionality can be implemented in an architecture independent way. This design format also allows for other architectures to be added into the system without requiring knowledge of RAMPARSER internals.

3.4. *Gathering offsets of structure members*

In order to dynamically build models of kernel structures, RAMPARSER must be able to properly find the offsets for important structure members. We utilize multiple algorithms to meet this need.

The first algorithm developed is the simplest, as it compares load and store instruction sequences from multiple functions which access the same structure member. After these matching instructions are found, an intersection is performed on the results to find the unique instructions. Small functions were chosen to limit the possibility of false positives and as many functions as possible were chosen for the comparison. While this technique is used in a few places in the system, it is not sufficient to properly model all kernel structure elements, since using exact matches will miss cases where, for example, a different register is used in different functions as the pointer to the base of a structure.

To be able to find the offsets for all the structure members needed, we explicitly handle the numerous ways in which C structure references can be compiled. Code fragments 1 through 3 show how equivalent statements can be compiled to radically different instruction sequences. The C source code in code fragment 1 is from the *insert_vm_struct()* function within *mm/mmap.c* of the Linux kernel source base. This function was used to help find the offset of the *vm_file* member of *struct vm_area_struct*.

CODE FRAGMENT #1 (C):
if (!vma->**vm_file**)
CODE FRAGMENT #2 (Ubuntu 2.6.27-11-generic):
8b 5a **48** mov ebx,DWORD PTR [edx+**0×48**]
85 db test ebx,ebx
CODE FRAGMENT #3 (Debian 2.6.18-6-686):
83 7a 48 00 cmp DWORD PTR [edx+0×48],0×0

Each disassembly accomplishes the same task, checking the *vm_file* member for a value of zero, but the methods chosen by the compiler are vastly different. In order to be able to work in both circumstances, the system must be aware of these possible methods and check for all of them until the proper offset is found. In the code fragments above, the constant 0×48 within the indexed instructions is the offset for the *vm_file* member. By understanding possible instruction formats, RAMPARSER is able to properly extract this offset.

Beyond having to understand all the possible combinations for one architecture, the system was designed in a way that could exploit the similarities of compiled code between architectures. Code fragment 4 shows the first line of the *sys_remap_file_pages()* function, and fragments 5 through 9 show the disassembly of the instruction that accesses the *mm* member of *task_struct* for a variety of architectures.

CODE FRAGMENT #4 (C)
struct mm_struct *mm = current->mm;
CODE FRAGMENT #5 (Playstation 3 2.6.28):
e9 2d **01 b0** ld r9,**432**(r13)
CODE FRAGMENT #6 (Ubuntu 2.6.27-7 x86_64):
4C 8B A0 **40 02 00 00** mov r12,[rax+**0×240**]
CODE FRAGMENT #7 (Debian 2.6.18-6 x86_64):
48 8B 80 D0 00 00 00 mov rax,[rax+0×d0]
CODE FRAGMENT #8 (Ubuntu 2.6.27-7 x86):

8B B0 **CC 01 00 00** mov esi,[eax+0×**1 cc**]
CODE FRAGMENT #9 (Debian 2.6.18-6 x86):
8B A8 **84 00 00 00** mov ebp,[eax+**0×84**]

Even with the differences between the opcodes, offset sizes, and locations in the functions, the information inside each architecture module allows RAMPARSER to properly locate and extract the needed offset. Throughout the process no extra knowledge of specific kernel or GCC constructs is used when analyzing the image. By limiting the amount of information to only that which is needed to define the architecture, the system is able to work on a very wide range of kernels, versions, and architectures.

3.5. *Important structures*

The combination of the structure offset gathering algorithms with other structure specific algorithms gives the system the ability to gather all needed offsets. The following section describes how offsets of the most important fields in a variety of useful kernel structures are gathered.

3.5.1. *task_struct*

To implement the per process functionality described earlier, the *task_struct* structure must be properly modeled, since this structure holds all information for a particular process. This information includes the name of the process, address space mappings, open files, signal information, and everything else needed by the kernel to track and schedule the process. To be able to quickly access information for the currently running process, a pointer to its *task_struct* is stored in a global variable *current* within the Linux kernel. Throughout 2.6 kernel development, access to this member has been optimized, since the operation happens so frequently. Code fragment 10 shows the C source for the *sys_getuid* system call, which extracts the user ID for the currently executing process. In earlier kernels the location of the pointer is found on the kernel stack for the process. Newer kernels store the location of *current* in the per-CPU variable, *current_task*, which is stored as *per_cpu__current_task* in the *System.map* file. The needed offset for the user ID field in the *task_struct* is shown in bold.

CODE FRAGMENT #10 (sys_getuid in C):
```
asmlinkage long sys_getuid(void){
    return current->uid;
}
```
CODE FRAGMENT #11 (Ubuntu 2.6.27-11-generic sys_getuid):
64 a1 00 c0 50 c0 mov eax,fs:0×c050c000
8b 80 **c0 02 00 00** mov eax,DWORD PTR [eax+**0×2c0**]
CODE FRAGMENT #12 (Debian 2.6.18-6-686 sys_getuid):
89 e0 mov eax,esp
25 00 e0 ff ff and eax,0×ffffe000
8b 00 mov eax,DWORD PTR [eax]
8b 80 **50 01 00 00** mov eax,DWORD PTR [eax+**0×150**]
CODE FRAGMENT #13 (Centos 2.6.9-78.EL sys_getuid):
b8 00 f0 ff ff mov eax,0×fffff000
21 e0 and eax,esp
8b 00mov eax,DWORD PTR [eax]
8b 80 **a4 01 00 00** mov eax,DWORD PTR [eax+**420**]

RAMPARSER starts its analysis by gathering the offset of the *task_struct uid* member from the *sys_getuid()* function. Since accesses to *current* are almost always compiled the same way across a particular kernel version, the opcodes used to find *current* and reference a member are stored. This stored pattern is then used to analyze *sys_getgid()*, *sys_geteuid()*, and *sys_getegid()*, and find the offsets to important structure members that they contain.

Different approaches are taken to find other members of *task_struct*. For example, the *tasks* member of *task_struct* is only referenced within the *for_each_process* macro by the kernel. This means its code is embedded only within functions that call this macro. To extract the offset, the instruction intersection algorithm described previously is used.

To find the *pid* member of *task_struct*, the addresses of the first four processes are gathered, which have statically assigned process IDs of 0-3. The blocks of memory which holds these *task_structs* are then scanned for four byte integers which contain their pid. Once the offset of all these blocks are collected, an intersection of all four lists is performed. If the intersection returns multiple results, then any of them could be valid since they could represent the *pid* or *tid* member or the position within the PID hashtable for the specific process.

The remaining *task_struct* member's offsets are easy to gather using knowledge of *System.map* and knowledge of the INIT_TASK macro. This macro is used to initialize the *init_task* variable which contains the *task_struct* structure of the statically defined *swapper* process, which always has PID 0. Many of the structure members which are pointers are initialized to variables contained within *System.map*, and the *comm* member is initialized to "swapper". By obtaining the virtual address of *init_task* from *System.map* as well as the virtual addresses of the other symbols listed in the illustration below, such as *init_fs* and *init_files*, subtraction can be used to obtain the offsets of the *files*, *fs*, *user*, and *comm* members.

```
struct task_struct init_task =
INIT_TASK(init_task);
#define INIT_USER (&root_user)
#define INIT_TASK(tsk) \
{
   .stack = &init_thread_info,
   .user = INIT_USER,
   .comm = "swapper",
   .fs = &init_fs,
   .files = &init_files,
   /* not all fields shown */
}
```

3.5.2. *mm_struct*

struct *mm_struct* is used to hold all the information related to the memory management of a process. Of interest to RAMPARSER are the members needed to track memory mappings within the process' address space. For replication of the */proc* filesystem's per process *maps* file, information from this structure as well as the closely related *vm_area_struct* must be used. Each *vm_area_struct* holds information about one mapping within a process such as the starting and ending address, permissions, number of pages, and mapped file, if any. An *mm_struct* contains pointers to the starting and ending

address of the *vm_area_struct* for the process' code, heap, stack, and vdso (a type of shared object). It also contains pointers to the start and end of the command line arguments and environment variables.

Multiple members from both structures must be accessible in order to properly track mappings within a process. The first member of *mm_struct* across every 2.6 kernel is the *mmap* member which points to the first *vm_area_struct* of the process. Each *vm_area_struct* contains a pointer to the next one and this list is sorted by starting address. To properly emulate the */proc/<pid>/maps* file, the *vm_start*, *vm_end*, *vm_next*, *vm_flags*, and *vm_file* member offsets must be known. Since *vm_area_struct* is rarely changed, all of these members besides *vm_file* are at constant offsets throughout all kernel versions.

To find the offset of *vm_file*, the instruction sequence of the *insert_vm_struct* and *copy_vma* functions are compared for identical load instructions. If an exact match is found, the encoded offset is used. If a match is not found, then the comparison instructions of the functions are gathered and the offset of identical ones extracted. By using this two step approach, the *vm_file* offset is attainable even when differing compilation constructs are used between kernel versions.

3.5.3. *File*

struct *file* contains information related to a file, such as its path, the user ID and group ID of the owner, the permissions it was opened with, and other data needed by the kernel. To be able to build the full pathname for open and mapped files, RAMPARSER needs the offset of struct *file*'s *f_dentry* and *f_vfsmnt* members, struct *dentry*'s *d_parent* and *d_inode* members, and a complete understanding of struct *qstr*.

The one-line *posix_lock_file()* function in *fs/locks.c* accesses the *inode* member of a struct *file* through its struct *dentry* member. This deference of both structure members allows RAMPARSER to gather the offset of both *f_dentry* and *d_inode* from one small function. The following code fragments 14-16 show the C source for this function as well as the disassembly across multiple kernels.

CODE FRAGMENT #14 (C posix_lock_file):

```
int posix_lock_file(struct file *filp,
struct file_lock *fl,
struct file_lock *conflock) {
  return
  __posix_lock_file(
  filp->f_path.dentry->d_inode,
  fl, conflock);
}
```

CODE FRAGMENT #15 (Playstation 3 2.6.28):

```
e9 23 00 18 ld r9,24(r3)
e8 69 00 38 ld r3,56(r9)
```

CODE FRAGMENT #16 (Ubuntu 2.6.27-7 x86_64, commented):

```
; move filp->f_path.dentry to rax
  mov rax,[rdi+0×18]
; move dentry->d_inode to rdi
  mov rdi,[rax+0×10]
```

Once the offset for *f_dentry* is found, the offset for *f_vfsmnt* can be calculated since the two members are adjacent in memory. By using *posix_lock_file*, both of the needed offsets within struct *file* can be obtained.

3.5.4. *Dentry*

struct *dentry* is used to cache the name and corresponding inode for a file. Of interest to RAMPARSER are the *d_parent* and *qstr* members. The offset of the *d_parent* member is found by analyzing the *d_alloc_anon()* function which accesses this member after a call to the *d_alloc()* function. Code fragment 17 shows the call to *d_alloc()* which returns a *dentry* pointer whose *d_parent* member is then referenced. The next two fragments, 18 and 19, show the disassembly of this sequence and how RAMPARSER can use the information to properly locate the needed instruction.

CODE FRAGMENT #17 (C):

```
tmp = d_alloc(NULL, &anonstring);
if (!tmp)
  return NULL;
tmp->d_parent = tmp;
```

CODE FRAGMENT #18 (x86 Ubuntu 2.6.27-7):

```
; call to d_alloc
E8 D9 F6 FF FF call dword 0×fffff720
85 C0 testeax,eax
89 C7 mov edi,eax ; non-indexed mov instruction
74 DD jz 0×2a
89 47 18 mov [edi+0×18],eax
```

CODE FRAGMENT #19 (x86 Debian 2.6.18-6):

```
; call to d_alloc
E8 07 F9 FF FF call 0×fffff92f
85 C0 test eax,eax
; non-indexed mov instruction
89 C3 mov ebx,eax
0F 84 A5 00 00 00 jz near 0×d7
89 43 18 mov [ebx+0×18],eax
```

Gathering this offset requires the ability to find call instructions to specific addresses. All architectures use relative addressing when encoding the destination of a control flow transfer. To account for this, the call instruction opcode and the size of the relative address are required in the architecture module. For both x86 and x86_64, four byte addresses are used while on PPC64 three byte addresses are used. Once the difference in the current instruction address and the destination is determined, it must be converted to its twos complement form. Once this converted number and the call opcode are combined, they can be used to search for the call instruction with a function.

Acquiring this offset also requires the ability of the system to distinguish between indexed and non-indexed instructions. As highlighted in code fragment 18 and 19, there are two *mov* instructions, with opcode 0×89, after the call instruction, but only one contains the needed offset. RAMPARSER's architecture modules use knowledge of the instruction format to determine when an offset is being used and skips instructions which do not access one. In this instance, the 0×C3 encoding signifies that a register to register transfer is being done while the 0×43 encoding signifies that an *[ebx]+disp8* reference is being done (http://download.intel.com/design/

processor/manuals/253666.pdf). By looking until an indexed based reference is used, RAMPARSER is able to select the proper instruction to analyze.

While the method described works to find the *d_parent* member for all tested x86 and x86_64 kernels, for some undetermined reason, the Playstation 3 kernels did not export the *d_alloc_anon* function in System.map. Since this symbol is not in the *System.map* file it cannot be used within RAMPARSER. Instead, the *d_alloc_root* function, which also accesses the *d_parent* member after a call to *d_alloc*, was used for the ppc64 architecture.

3.5.5. *Qstr*

struct *qstr* is a simple structure to track and maintain C-strings with an explicitly associated length. It contains a character pointer, an integer that holds the length of the character array, and a hash of the current string. The name of each *dentry* is stored within a *qstr* and is necessary to construct the full pathname of files. Since the *qstr* is stored inside the *dentry* and not referenced by a pointer, RAMPARSER has to walk the memory block of a valid *dentry* object to construct a representation of *qstr*. This procedure starts by first looking for a valid kernel pointer. After one is found, the pointer is followed to see if its referenced location contains a valid filename. If it does, then this pointer is known to be the *name* member of *qstr*. To find the ordering of the *hash* and *len* members relative to the character pointer, the two integers before and after the name are examined to see if they contain either the length of the string or the hash of the string as computed by the *partial_hash()* function within the kernel. Once these relative offsets are found, the offsets within struct *dentry* can be easily computed.

3.5.6. *inet_sock*

struct *inet_sock* is the kernel's data structure to track INET sockets. Of interest to RAMPARSER are the members that hold the source and destination port and IP addresses of a socket, since these fields support live forensics queries involving network activity. To find these offsets, RAMPARSER takes advantage of the *udp_disconnect()* function, which has remained stable throughout the lifetime of the 2.6 kernel series development. Code fragment 20 contains a partial listing of the declaration of *inet_sock* in the proper order, showing that three of the four needed members are contiguous in memory. Code fragment 21 lists a piece of the *udp_disconnect()* function that accesses two of the needed members in consecutive instructions.

CODE FRAGMENT #20 (needed members in inet_sock)

```
__be32 daddr;
__be32 rcv_saddr;
__be16 dport;
__u16 num;
__be32 saddr;
__s16 uc_ttl;
__u16 cmsg_flags;
struct ip_options *opt;
__be16 sport;
```

CODE FRAGMENT #21 (accessing members):

```
sk->sk_state = TCP_CLOSE;
inet->daddr = 0;
inet->dport = 0;
```

CODE FRAGMENT #22 (Ubuntu 2.6.27-7 x86 disassembly):

```
C6 40 02 07 mov byte [eax+0×2],0×7
C7 80 7C 01 00 00 00 00 mov dword [eax+0×17c],0×0
66 C7 80 84 01 00 00 00 mov word [eax+0×184],0×0
```

This disassembly sequence of three indexed load instructions is constant across architectures. On x86 and x86_64, RAMPARSER's ability to recognize 16-bit constructs is used to properly determine the offset. The 0×66 preceding the last mov instruction signifies an operand size of 16-bits which corresponds to the 16-bit port number used by network protocols. By locating the two offsets referenced within the sequence, the *rcv_saddr* offset can be calculated since it is between the two in memory. The *sport* member stores the source port for a socket. For all kernel versions tested, the value can be found by finding the first kernel pointer after *dport*, the *opt* member, and examining the following 16-bit integer.

3.5.7. *Sock*

Many of the Linux kernel's network structures have a struct *sock* embedded as the first member. To be able to access members after this structure, RAMPARSER needs to determine the size of the *sock* structure. Within the *unix_copy_addr()* function, the *addr* member of *unix_sock* immediately after the embedded *sock* is accessed. This means that *addr*'s offset is also the size of the *sock* structure. The following code fragments show the declaration of *unix_sock* followed by the C source and disassembly that accesses the member.

CODE FRAGMENT # 23 (unix_sock declaration):

```
struct unix_sock {
  struct sock sk;
  struct unix_address *addr;
  /* fields omitted */
}
```

CODE FRAGMENT # 24 (accessing addr fields):

```
struct unix_sock *u = unix_sk(sk);
msg->msg_namelen = 0;
if (u->addr) {
/* … */
}
```

CODE FRAGMENT # 25 (Disassembly of the u->addr reference):

```
8b b2 78 01 00 00 mov esi,DWORD PTR [edx+0×178]
```

By determining the proper *mov* instruction and extracting the offset, RAMPARSER is able to gather the size of a *sock* in memory which is needed during later processing.

3.5.8. *Socket*

Within the kernel, *socket* and *inode* pairs are grouped together within a *socket_alloc* structure. Since these structures are placed on the stack and not referenced by a pointer, RAMPARSER must obtain the size of the socket structure to properly use the pair.

CODE FRAGMENT # 26 (socket_alloc declaration):

```
struct socket_alloc {
  struct socket socket;
```

```
    struct inode vfs_inode;
};
CODE FRAGMENT # 27 (SOCKET_I function):
static inline struct socket
*SOCKET_I(struct inode *inode) {
    return &container_of(inode,
    struct socket_alloc,
    vfs_inode)->socket;
}
CODE FRAGMENT # 28 (call to sock_release):
sock_fasync(-1, filp, 0);
sock_release(SOCKET_I(inode));
CODE FRAGMENT # 29 (SOCKET_I call disassembly):
e8 fa ec ff ff call c02923a0 <sock_fasync>;
8d 43 dc lea eax, [ebx-0×24]
e8 62 ff ff ff call c0293610 <sock_release>;
```

The *SOCKET_I()* function is used to quickly find the *socket* structure of a file when only given the *inode*. Since *socket_alloc()* is used, this transformation requires only a subtraction from the *inode* member address of the size of *socket* to acquire the corresponding *socket* structure. Code fragment 28 gives a partial listing of the *sock_close()* function that uses the return value of a call to *SOCKET_I()* as the first parameter to *socket_release()*. As shown in code fragment 29, this makes locating the instruction that performs the subtraction trivial. RAMPARSER first finds the call to *sock_fasync* within *sock_close()* and then extracts the offset from the following register-based arithmetic operation.

The *sk* member of *socket* contains a pointer to the *sock* structure of the file. To find the offset of this member, an intersection of indexed *mov* instructions is performed between the *udp_poll()* and *tcp_poll()* functions. Both of these functions immediately access the *sk* member of the struct *socket* passed as the parameter.

Once the size of *socket* and the offset of *sk* is obtained, RAMPARSER has the ability to get the *sock* structure of a file. This structure is used to populate the information needed during netstat emulation.

3.6. *Testing the system*

To ensure that changes and updates to RAMPARSER do not break existing functionality, an automatic testing system was created. Multiple memory dumps were collected for each of the test bed installations as well as the corresponding *System.map* file. Next, RAMPARSER was run against each of these images, and the resulting databases were verified by hand to be accurate. Once the databases were verified, they were saved along with the images and *System.map* files.

The testing module has a record of the names and locations of the database, memory image, and *System.map* for all the test bed installations. When run, the testing script invokes RAMPARSER for each saved memory dump and directs the database creation to a temporary file. Once the process is complete, it compares the new database to the saved valid one. Interestingly, SQLlite's minimalist file format and record keeping allows direct hashing of database files on disk for testing if two databases contain the same information. Using this knowledge, the testing script uses the Python *md5* module to hash each database file and then compare it.

4. Future work

While many techniques were developed for extracting kernel structure member offsets, there are still a number of compiled code constructs that RAMPARSER cannot analyze. Accurately locating and parsing these instructions will require integration of a disassembler into RAMPARSER, to fully support on-the-fly reverse engineering of kernel functions. The advantage of this approach will be that structure offsets that are only referenced deep inside large functions or within convoluted constructs can be precisely located by the searching algorithm.

In order for RAMPARSER to work beyond the 2.6.27 series of kernels, new logic must be added to accommodate substantial changes made in the mainstream kernel. The first major change is the replacement of the *uid*, *gid*, *euid*, and *egid* members of *task_struct* for members related to the new user credential system in 2.6.29. Other minor changes in functions used to extract offsets starting in 2.6.28 must also be accounted for, but they only pose a minimal problem. RAMPARSER development will always be a continuous endeavor due to the drastic changes made between versions in the kernel, but the current system ensures that the effort needed to make these changes, while still working on older kernels, is minimal.

Future integration into systems such as FACE or PyFlag is also desirable as this will immediately make these applications adaptable to a much larger range of kernels and architectures.

5. Conclusions

The main contribution of the work described in this paper is to illustrate that live forensics tools can be made substantially less brittle with respect to kernel versions than the current state-of-the-art. We do not claim that the techniques presented are simple to understand for a novice tool developer nor that they completely solve portability issues—rather, we claim that techniques like those presented *must* be incorporated if deep, live forensics tools are to be useful to digital forensics investigators. The current practice of supporting only limited kernel versions or requiring extensive manual configuration by investigators is not tenable given the large variety of kernel versions an investigator will regularly encounter.

With the capabilities provided by RAMPARSER, an investigator can gather a large amount of live forensics evidence from a target memory image. This evidence provides a substantial amount of information about the state of and the activities underway on a forensic target at the time of memory acquisition. RAMPARSER also provides future researchers a solid starting point to delve into memory forensics and develop even further reaching tools.

REFERENCES

Anderson D. Crash, http://people.redhat.com/anderson; 2008.

Case A, Cristina A, Marziale L, Richard III GG, Roussev V. FACE: automated digital evidence discovery and correlation. In: Proceedings of the 8th Annual digital forensics research workshop. Baltimore, MD: DFRWS; 2008.

Cohen M. PyFlag – an advanced network forensic framework. In: Proceedings of the 8th Annual digital forensics research workshop. Baltimore, MD: DFRWS 2008; 2008.
Kornblum J. Exploiting the rootkit paradox with Windows Memory Analysis. International Journal of Digital Evidence Fall 2006;5(1).
Kornblum J. Using every part of the buffalo in Windows Memory Analysis. Digital Investigation March 2007;4(1):24–9.
Petroni N, Walters A, Fraser T, Arbaugh W. FATKit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation December, 2006;3(4):197–210.
Schatz B. BodySnatcher: towards reliable volatile memory acquisition by software. In: Proceedings of the 2007 digital forensic research workshop. DFRWS 2007; 2007.
Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. In: Proceedings of the 2006 digital forensic research workshop. DFRWS; 2006a.
Schuster A. Pool allocations as an information source in Windows Memory Forensics. International Conference on IT-Incident Management and IT-Forensics; October 2006b.
Urrea JM. An analysis of Linux RAM forensics. Naval Post Graduate School Thesis; March 2006.

Andrew Case is a Senior Security Researcher at Digital Forensics, Solutions, LLC.

Lodovico Marziale is a Senior Security Researcher at Digital Forensics Solutions, LLC.

Golden G. Richard III is Professor of Computer Science and Director of the Greater New Orleans Center for Information Assurance (GNOCIA) at the University of New Orleans. He is also CTO of Digital Forensics Solutions, LLC.