A complex network diagram consisting of numerous small blue and green nodes connected by thin, light blue lines. The nodes are arranged in a roughly circular pattern, with a denser central area. A small, semi-transparent circular icon is positioned in the upper right quadrant of the network, containing a stylized globe or sphere with a few lines and dots. A light blue, rounded rectangular banner is overlaid horizontally across the center of the network, containing the text "Digital Forensics Solutions" in a dark, sans-serif font.

Digital Forensics Solutions

Linux Memory Analysis Workshop – Session 1

Andrew Case

Who Am I?

- Security Analyst at Digital Forensics Solutions
 - Also perform wide ranging forensics investigations
- Volatility Developer
- Former Blackhat, SOURCE, and DFRWS speaker
- Computer Science degree from UNO
- GIAC Certified Forensics Analyst (GCFA)

Format of this Workshop

- I will be presenting the Linux kernel memory analysis capabilities of Volatility
- Along the way we will be seeing numerous examples of Linux kernel source code as well as Volatility's plugins source code
- Following along with me while I use Volatility to recover data will get you the most out of this workshop

Setting up Your Environment

Agenda for Today's Workshop

- 1. Recovering Vital Runtime Information**
2. Investigating Live CDs (Memory Analysis)
3. Detecting Kernel Rootkits

Agenda for This Hour

- Memory Forensics Introduction
- Recovering Runtime Information
 - Will discuss kernel internals necessary to recover processes, memory maps, loaded modules, etc
 - Will discuss how these are useful/relevant to forensics & IR
 - We will be recovering data with Volatility as we go
- Q&A / Comments

Memory Forensics Introduction

Introduction

- Memory analysis is the process of taking a memory capture (a copy of RAM) and producing higher-level objects that are useful for an investigation
- A memory capture has the entire state of the operating system as well as running applications
 - Including all the related data structures, variables, etc

The Goal of Memory Analysis

- The higher level objects we are interested in are in-memory representations of C structures, custom data structures, and other variables used by the operating system
- With these we can recover processes listings, filesystem information, networking data, etc
- This is what we will be talking about throughout the workshop

Information Needed for Analysis

- The ability to:
 1. Locate needed data structures in memory
 2. Model those data structures offline
 3. Report their contents

Locating Data Structures

- To locate static data structures, we use the *System.map* file
 - Contains the name and address of every static data structure used in the kernel
 - Created in the kernel build process by using *nm* on the compiled *vmlinux* file

Modeling Data Structures

- The parts of the Linux kernel we care about are written in C
- All data structures boil down to C structures
- These have a very simple in-memory representation (next slide)

C Structures in Memory

- Source Code:

```
struct blah {  
    int i;  
    char c;  
    short s;    };  
struct blah *b =  
malloc(...);
```

- In Memory:

- Lets say we have an instance of 'b' at 0x0

- Then:

b->i goes from 0x0 to 0x4

b->c goes from 0x4 to 0x5

b->s goes from 0x5 to 0x7

Modeling Structures

- During analysis we want to automatically model each C structure of interest
- To do this, we use Volatility's *dwarfparse.py*:
 - Builds a profile of C structures along with members, types, and byte offsets
 - Records offsets of global variables
- Example structure definition
 - 'ClassObject': [0xa0, { Class name and size
 - 'obj': [0x0, ['Object']], Member name, offset,
 and type

Introducing Volatility

Volatility

- Most popular memory analysis framework
 - Written in Python
 - Open Source
 - Supports Windows {XP, Vista, 7, 2003, 2008}
 - Support Linux 2.6.9 to 2.6.3x on Intel and ARM
- Allows for analysis plugins to be easily written
- Used daily in real forensics investigations
- Will be the framework used in this workshop

Volatility Object Manager

- Once we have a model of a kernel's data structures (profiles) we can then just rely on Volatility
- Its object manager takes care of parsing the struct definitions, including types, and then providing them as requested
 - Example on next slide

Example Plugin Code

- Accessing a structure is as simple as knowing the type and offset

```
intval = obj.Object("int", offset=intOffset, ..)
```

- Volatility code to access 'descriptor' of an 'Object':

```
o = obj.Object("Object", offset=objectAddress, ..)
```

```
c = obj.Object("ClassObject", offset=o.clazz, ...)
```

```
desc = linux_common.get_string(c.descriptor)
```

Volatility Address Spaces

- Address spaces are used to translate virtual addresses to offsets within a memory capture
 - Same process used to translate to physical addresses on a running OS
- Plugin developers simply need to pass the given address space to functions that need it
 - Manual change only required to access userland (will see an example in a bit)

Current Address Spaces

- x86 / x64
- Arm (Android)
- Firewire
- Windows Hibernation Files
- Crash Dumps
- EWF Files

Recovering Runtime Information

Runtime Information

- This rest of this session is focused on orderly recovery of data that was active at the time of the memory capture
- We will be discussing how to find key pieces of information and then use Volatility to recover them

Information to be Recovered

- Processes
- Memory Maps
- Open Files
- Network Connections
- Network Data
- Loaded Kernel Modules

Recovering Process Information

- Each process is represented by a *task_struct*
- Once a *task_struct* is located, all information about a process can be quickly retrieved
 - Possible to do it through other methods, but much more convoluted

Locating Processes – Method 1

- *init_task* is the symbol for the *task_struct* of “swapper”, the PID 0 process
 - Statically initialized
- *task_struct->tasks* holds a linked list of all active processes
 - NOT threads! (more on this later)
 - Simply walking the list gives us a process listing

Locating Processes – Method 2

- Newer kernels have a Process ID (pid) namespace
- The hash table can be located through *init_pid_ns* and then enumerated
- These hash tables are used to populate */proc/<pid>*

Wanted Per-Process Information

- Name and Command Line Arguments
- UID/GID/PID
- Starting/Running Time
- Parent & Child Processes
- Memory Maps & Executable File
- Open Files
- Networking Information

Needed *task_struct* Members

- Name
 - char comm[TASK_COMM_LEN]; // 16
 - Command line arguments in later slides
- User ID / Group ID
 - Before 2.6.29
 - *uid* and *gid*
 - Since 2.6.29
 - struct cred *cred;
 - cred->uid and cred->gid

task_struct Members Cont.

- Parent Process
 - `struct task_struct *real_parent;`
- Child processes
 - `struct list_head children; /* list of my children */`
- Process times
 - `utime, stime, start_time, real_starttm`

Recovery with Volatility

- Option 1:
 - In: *volatility/plugins/linux_task_list_ps.py*
 - Walks the `task_struct->tasks` list
- Option 2:
 - In: *volatility/plugins/linux_task_list_psaux.py*
 - Reads command line invocation from userland
 - Will cover algorithm after discussing memory management structures

Process Gathering Demo/Hands On

- Will be using:
 - *linux_task_list_ps*
 - *linux_task_list_psaux*

Process Memory Maps

- Viewed on a running system within */proc/<pid>/maps*
- Lists all mappings within a process including:
 - Mapped file, if any
 - Address range
 - Permissions

Accessing the Mappings

- Each mapping is stored as a *vm_area_struct*
- Stored in two places:
 - `task_struct->mm->mm_rb`
 - Red black tree of mappings
 - `task_struct->mm->mmap`
 - List of mappings ordered by starting address

Needed Members of `vm_area_struct`

- unsigned long *vm_start*, *vm_end*
 - The starting and ending addresses of the mapping
- `vm_area_struct` *vm_next*
 - The next vma for the process (linked list from `mm->mmap`)
- struct file *vm_file*
 - If not NULL, points to the mapped file (shared library, open file, main executable, etc)

Recovery with Volatility

- Listing mappings implemented in *volatility/plugins/linux_proc_maps.py*
- Analyzing specific mappings implemented in *volatility/plugins/linux_dump_maps.py*
 - Can specify by PID or address

Getting the Command Line Args

- Each processes' *mm_struct* contains a number of variables that hold the address of the .text, .data, heap, stack, and command line args
- Command line args:
 - Beginning address stored in *arg_start*
 - End address stored in *arg_end*
 - Arguments are stored in a list of null terminated “strings”

Using `->mm` to get `**argv`

```
# switch pgd (address space)
```

```
tmp_dtb = self.addr_space.vtop(task.mm.pgd)
```

```
# create new address space
```

```
proc_as =
```

```
    self.addr_space.__class__(self.addr_space.base,  
                              self.addr_space.get_config(), dtb = tmp_dtb)
```

```
# read in command line argument buffer
```

```
argv = proc_as.read(task.mm.arg_start,  
                   task.mm.arg_end - task.mm.arg_start)
```

Gathering Open Files

- Want to emulate */proc/<pid>/fd*
- *task_struct->files->fdt->fd* is array of *file* structures
- Each array index is the file descriptor number
- If an index is non-NULL then it holds an open file
- Use *max_fds* of the *fdt* table to determine array size

Information Per-File

- Path information stored in the *f_dentry* and *f_vfsmnt* members
 - To get full path, need to emulate *__d_path* function
- Inode information stored in *f_dentry* structure
 - Contains size, owner, MAC times, and other metadata
- Recovering file contents in-memory requires use of the *f_mapping* member
 - Come back for session 2!

Memory Maps and Open Files Demo

- Memory Maps
 - Listing process mappings
 - Acquiring the stack and heap from interesting processes
- Open Files
 - Lists open files with their file descriptor number

Networking Information

- The kernel contains a wealth of useful information related to network activity
- This info is immensely helpful in a number of forensics and incident response scenarios

Netstat Plugin

- Used to emulate the *netstat* command
- This information is found on a running machine found in these */proc/net/* files:
 - tcp/tcp6
 - udp/udp6
 - unix
- We find open sockets by enumerating open files of processes and then finding those which are socket file descriptors

Volatility's linux_netstat.py

```
openfiles = lof.linux_list_open_files.calculate(self)
```

```
# for every open file
```

```
for (task, filp, _i, _addr_space) in openfiles:
```

```
    d = filp.get_dentry() # the files dentry
```

```
    if filp.f_op == self.smap["socket_file_ops"] or
```

```
    filp.d.d_op == self.smap["sockfs_dentry_operations"]:
```

```
        # it is a socket, can get the protocol information
```

```
        iaddr = d.d_inode
```

```
        skt = self.SOCKET_I(iaddr)
```

```
        inet_sock = obj.Object("inet_sock", offset = skt.sk, ...)
```

ARP Cache

- Emulates *arp -a*
- The ARP cache stores recently discovered IP and MAC address pairs
 - It is what facilitates ARP poisoning
- Recovery of this cache provides information on other machines the target machine was communicating with

Recovering the ARP Cache

- Implemented in *linux_arp.py*
- This code walks the *neigh_tables* and their respective *hash_buckets* to recover *neighbor* structures
- These contain the device name, mac address, and corresponding IP address for each entry

Routing Table

- Emulates *route -n*
- The routing table stores routing information for every known gateway device and its corresponding subnet
- The *linux_route* plugin recovers this information

Routing Cache

- Emulates *route -C*
- This cache stores recently determined source IP and gateway stores
- A great resource to determine recent network activity on a computer

Network Recovery Demo/Hands On

- Many plugins!

Dmesg

- The simplest plugin in all of Volatility
- Simply locates and prints the kernel debug buffer

Dmesg Plugin Code

```
ptr_addr = self.smap["log_buf"]
```

```
# the buffer
```

```
log_buf_addr = obj.Object("long", offset = ptr_addr, vm =  
    self.addr_space)
```

```
# its length
```

```
log_buf_len = obj.Object("int", self.smap["log_buf_len"], vm =  
    self.addr_space)
```

```
# read in the buffer
```

```
yield linux_common.get_string(log_buf_addr, self.addr_space,  
    log_buf_len)
```

Loaded Kernel Modules

- Want to emulate the *lsmod* command
- Each module is represented by a *struct module*
- Each active module is kept in the *modules* list
- We can simply walk the list to recover all needed information

Information Per Module

- `char name[MODULE_NAME_LEN]`
 - The name of the module
- `void module_init`
 - .text + .data of init functions
- `void module_core`
 - .text + .data of core functions
- `symtab/strtab`
 - Symbol and string tables
- `struct list_head list`
 - Entry within the list of loaded modules

Recovery with Volatility

- In: *volatility/plugins/linux_lsmmod.py*

- Volatility code:

```
mods_addr = self.smap["modules"]
```

```
modules = obj.Object("list_head", offset=mods_addr,)
```


```
for module in
```

```
    linux_common.walk_list_head("module", "list",  
    modules, ...):
```

```
        yield module
```

Questions/Comments?

- Please fill out the feedback forms!
- Contact:
 - andrew@digdeeply.com
 - @attrc



Digital Forensics Solutions

Linux Memory Analysis Workshop – Session 2

Andrew Case

Who Am I?

- Security Analyst at Digital Forensics Solutions
 - Also perform wide ranging forensics investigations
- Volatility Developer
- Former Blackhat, SOURCE, and DFRWS speaker
- Computer Science degree from UNO
- GIAC Certified Forensics Analyst (GCFA)

Format of this Workshop

- I will be presenting the Linux kernel memory analysis capabilities of Volatility
- Along the way we will be seeing numerous examples of Linux kernel source code as well as Volatility's plugins source code
- Following along with me while I use Volatility to recover data will get you the most out of this workshop

Setting up Your Environment

Agenda for Today's Workshop

1. Recovering Vital Runtime Information
- 2. Investigating Live CDs Through Memory Analysis**
3. Detecting Kernel Rootkits

Agenda for This Hour

- Discuss Live CDs and how they disrupt the normal forensics process
- Present research that enables traditional investigative techniques against live CDs
- We will be recovering files and data as we go along
- Q&A / Comments

Introducing Volatility

Volatility

- Most popular memory analysis framework
 - Written in Python
 - Open Source
 - Supports Windows {XP, Vista, 7, 2003, 2008}
 - Support Linux 2.6.9 to 2.6.3x on Intel and ARM
- Allows for analysis plugins to be easily written
- Used daily in real forensics investigations
- Will be the framework used in this workshop

Volatility Object Manager

- Once we have a model of a kernel's data structures (profiles) we can then just rely on Volatility
- Its object manager takes care of parsing the struct definitions, including types, and then providing them as requested
 - Example on next slide

Example Plugin Code

- Accessing a structure is as simple as knowing the type and offset

```
intval = obj.Object("int", offset=intOffset, ..)
```

- Volatility code to access 'descriptor' of an 'Object':

```
o = obj.Object("Object", offset=objectAddress, ..)
```

```
c = obj.Object("ClassObject", offset=o.clazz, ...)
```

```
desc = linux_common.get_string(c.descriptor)
```

Volatility Address Spaces

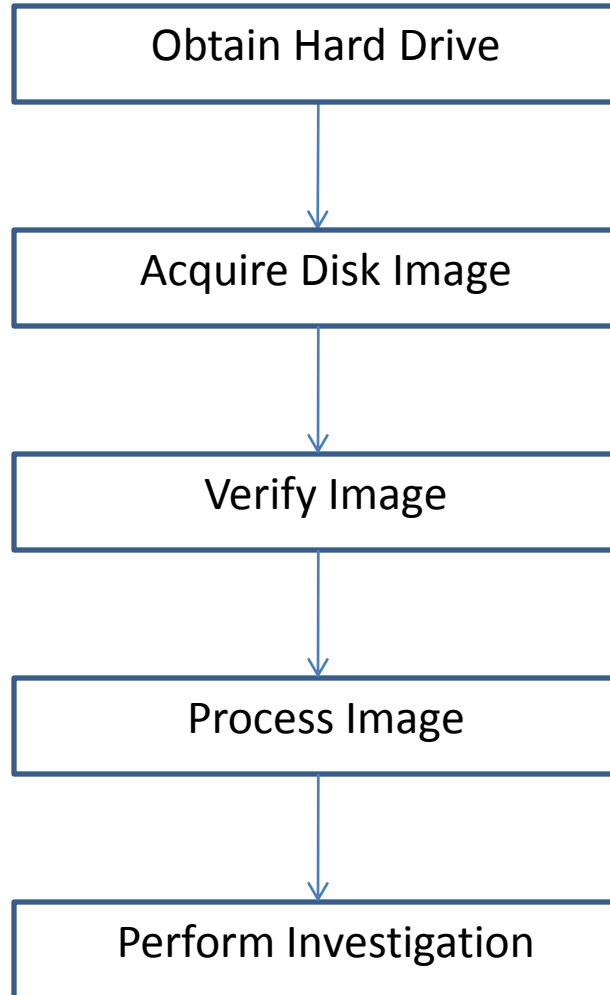
- Address spaces are used to translate virtual addresses to offsets within a memory capture
 - Same process used to translate to physical addresses on a running OS
- Plugin developers simply need to pass the given address space to functions that need it
 - Manual change only required to access userland (will see an example in a bit)

Current Address Spaces

- x86 / x64
- Arm (Android)
- Firewire
- Windows Hibernation Files
- Crash Dumps
- EWF Files

Live CD Introduction

Normal Forensics Process



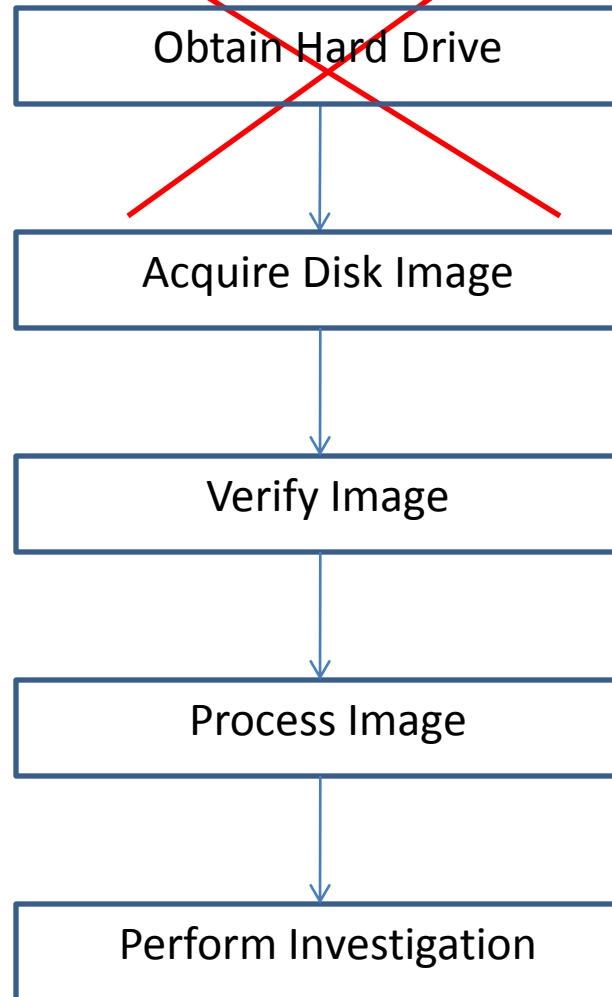
Traditional Analysis Techniques

- Timelining of activity based on MAC times
- Hashing of files
- Indexing and searching of files and unallocated space
- Recovery of deleted files
- Application specific analysis
 - Web activity from cache, history, and cookies
 - E-mail activity from local stores (PST, Mbox, ...)

Problem of Live CDs

- Live CDs allow users to run an operating system and all applications entirely in RAM
- This makes traditional digital forensics (examination of disk images) impossible
- All the previously listed analysis techniques cannot be performed

The Problem Illustrated



No Disks or Files, Now What?

- All we can obtain is a memory capture
- With this, an investigator is left with very limited and crude analysis techniques
- Can still search, but can't map to files or dates
 - No context, hard to present coherently
- File carving becomes useless
 - All fragmented in memory
- Good luck in court

People Have Caught On...

- The Amnesic Incognito Live System (TAILS) [1]
 - “No trace is left on local storage devices unless explicitly asked.”
 - “All outgoing connections to the Internet are forced to go through the Tor network”
- Backtrack [2]
 - “ability to perform assessments in a purely native environment dedicated to hacking.”

What It Really Means...

- Investigators without deep kernel internals knowledge and programming skill are basically hopeless
- It is well known that the use of live CDs is going to defeat most investigations
 - Main motivation for this work
 - Plenty anecdotal evidence of this can be found through Google searches

What is the Solution?

- Memory Analysis!
 - It is the only method we have available...
- This Analysis gives us:
 - The complete file system structure including file contents and metadata
 - Deleted Files (Maybe)
 - Userland process memory and file system information

Recovering the Filesystem

Goal 1: Recovering the File System

- Steps needed to achieve this goal:
 1. Understand the in-memory filesystem
 2. Develop an algorithm that can enumerate directory and files
 3. Recover metadata to enable timelining and other investigative techniques

The In-Memory Filesystem

- AUFS (AnotherUnionFS)
 - <http://aufs.sourceforge.net/>
 - Used by TAILS, Backtrack, Ubuntu 10.04 installer, and a number of other Live CDs
 - Not included in the vanilla kernel, loaded as an external module

AUFS Internals

- Stackable filesystem
 - Presents a multilayer filesystem as a single one to users
 - This allows for files created after system boot to be transparently merged on top of read only CD
- Each layer is termed a branch
 - In the live CD case, one branch for the CD, and one for all other files made or changed since boot

Walkthrough

- Explore AUFS on a running system

AUFS Userland View of TAILS

```
# cat /proc/mounts
```

```
aufs / aufs rw,relatime,si=4ef94245,noxino
```

```
/dev/loop0 /filesystem.squashfs squashfs
```

```
tmpfs /live/cow tmpfs
```

```
tmpfs /live tmpfs rw,relatime
```

```
# cat /sys/fs/aufs/si_4ef94245/br0
```

```
/live/cow=rw
```

```
# cat /sys/fs/aufs/si_4ef94245/br1
```

```
/filesystem.squashfs=rr
```

Mount
points
relevant
to AUFS

The
mount
point of
each
AUFS
branch

Forensics Approach

- No real need to copy files from the read-only branch
 - Just image the CD
- On the other hand, the writable branch contains every file that was created or modified since boot
 - Including metadata
 - No deleted ones though, more on that later

Linux Internals

Struct Dentry

- Represents a directory entry (directory, file, ...)
- Contains the name of the directory entry and a pointer to its inode structure
- Members used:
 - *d_inode* – pointer to inode structure
 - *d_name* – name of file

Struct Inode

- FS generic, in-memory representation of a disk inode
- Contains `address_space` structure that links an inode to its file's pages
- Also provides metadata:
 - MAC Times
 - Size
 - Owner
 - more...

Struct Address_Space

- Links physical pages together into something useful
- Holds the search tree of the struct *pages* for a file
 - This pages then lead to physical pages

Linux Internals Overview II

- Page Cache
 - Used to store *struct page* structures that correspond to physical pages
 - `address_space` structures contain linkage into the page cache that allows for ordered enumeration of all physical pages pertaining to an inode
- Tmpfs
 - In-memory filesystem
 - Used by TAILS to hold the writable branch

Enumerating Directories

- Once we can enumerate directories, we can recover the whole filesystem
- Not as simple as recursively walking the children of the file system's root directory
- AUFS creates hidden dentries and inodes in order to mask branches of the stacked filesystem
- Need to carefully interact between AUFS and tmpfs structures

Directory Enumeration Algorithm

- 1) Walk the super blocks list until the “aufs” filesystem is found
 - This contains a pointer to the root dentry
- 2) For each child dentry, test if it represents a directory

If the child is a directory:

- Obtain the hidden directory entry (next slide)
- Record metadata and recurse into directory

If the child is a regular file:

- Obtain the hidden inode and record metadata

Getting the Superblock

```
# get a reference the list of super blocks
sbptr = obj.Object("Pointer", offset =
    self.smap["super_blocks"], ...)
# the list
sb_list = obj.Object("list_head", offset = sbptr.v(),...)
# walk the list and find "aufs"
for sb in walk_list_head("super_block", "s_list", sb_list,
    if sb.s_id.startswith("aufs"):
        return sb.s_root
```

The Initial Code

```
# walk the directory and get the hidden dentry
```

```
def calculate(self):
```

```
    # get the superblock
```

```
    root_dentry = self.get_aufs_sb()
```

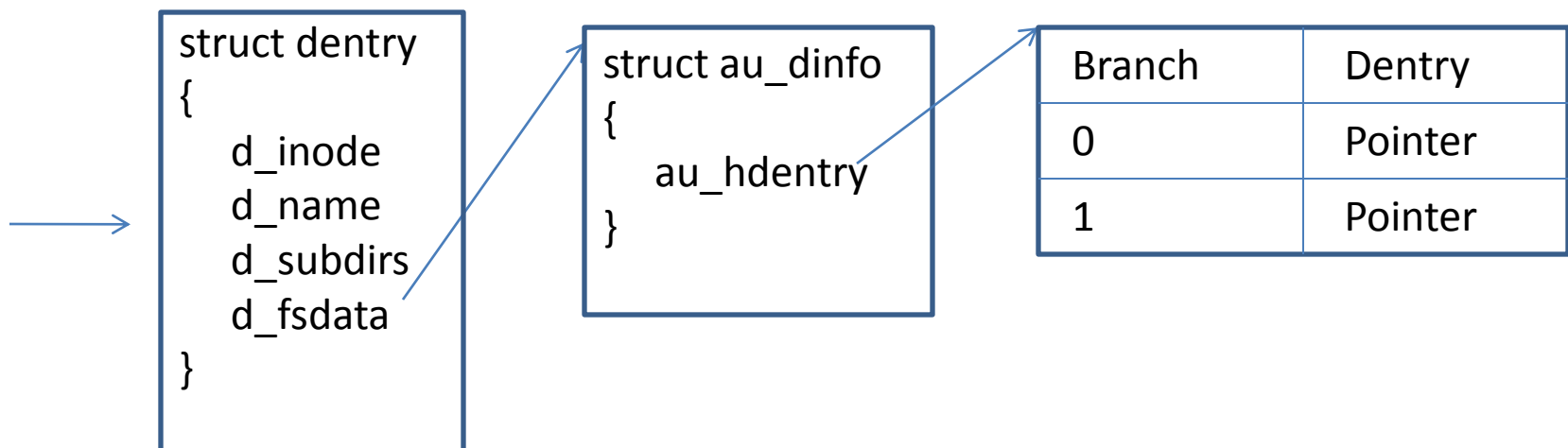
```
    # walk every dentry
```

```
    for dentry in walk_list_head("dentry", "d_u",  
                                root_dentry.d_subdirs, ...):
```

```
        h_dentry = self.get_h_dentry(dentry, 0)
```

Obtaining a Hidden Directory

- Each kernel dentry stores a pointer to an *au_dinfo* structure inside its *d_fsdata* member
- The *di_hentry* member of *au_dinfo* is an array of *au_hentry* structures that embed regular kernel dentries



get_h_dentry(dentry)

```
# the hidden "fsdata" pointer
```

```
dinfo = obj.Object("au_dinfo",  
offset=dentry.d_fsdata,
```

```
# get the array of hidden dentrys
```

```
dentry_array = obj.Object("Array",  
offset=dinfo.di_hdentry, "au_hdentry", count=2)
```

```
# grab the hidden dentry at branch 'index'
```

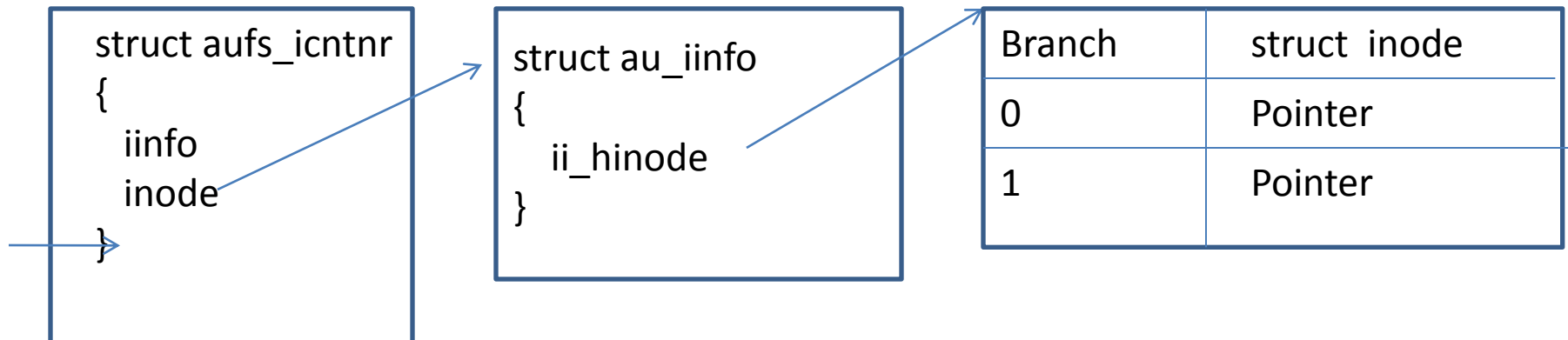
```
h_dentry = dentry_array[index].hd_dentry
```

Obtaining Metadata

- All useful metadata such as MAC times, file size, file owner, etc is contained in the hidden inode
- This information is used to fill the *stat* command and *istat* functionality of the Sleuthkit
- Timelining becomes possible again

Obtaining a Hidden Inode

- Each aufs controlled inode gets embedded in an *aufs_icntnr*
- This structure also embeds an array of *au_hinode* structures which can be indexed by branch number to find the hidden inode of an exposed inode



Demo/Hands-On

- Will recover the filesystem structure and metadata
- Will examine with a few FS analysis tools
- Will discuss how to keep forensically sound and discuss limitations

Goal 2: Recovering File Contents

- The size of a file is kept in its inode's *i_size* member
- An inode's *page_tree* member is the root of the radix tree of its physical pages
- In order to recover file contents this tree needs to be searched for each page of a file
- The lookup function returns a *struct page* which leads to the backing physical page

Recovering File Contents Cont.

- Indexing the tree in order and gathering of each page will lead to accurate recovery of a whole file
- This algorithm assumes that swap isn't being used
 - Using swap would defeat much of the purpose of anonymous live CDs
- Tmpfs analysis is useful for every distribution
 - Many distros mount /tmp using tmpfs, shmem, etc

Goal 3: Recovering Deleted Info

- Discussion:
 1. Formulate Approach
 2. Discuss the *kmem_cache* and how it relates to recovery
 3. Attempt to recover previously deleted file and directory names, metadata, and file contents

Approach

- We want orderly recovery
- To accomplish this, information about deleted files and directories needs to be found in a non-standard way
 - All regular lists, hash tables, and so on lose track of structures as they are deleted
- Need a way to gather these structures in an orderly manner
 - *kmem_cache* analysis to the rescue!

Recovery through *kmem_cache* analysis

- A *kmem_cache* holds all structures of the same type in an organized manner
 - Allows for instant allocations & deallocations
 - Used for handling of process, memory mappings, open files, and many other structures
- Implementation controlled by allocator in use
 - SLAB and SLUB are the two main ones

kmem_cache Internals

- Both allocators keep track of allocated and previously de-allocated objects on three lists:
 - *full*, in which all objects are allocated
 - *partial*, a mix of allocated and de-allocated objects
 - *free*, previously freed objects*
- The free lists are cleared in an allocator dependent manner
 - SLAB leaves free lists in-tact for long periods of time
 - SLUB is more aggressive

kmem_cache Illustrated

- */proc/slabinfo* contains information about each current *kmem_cache*
- Example output:

<i># name</i>	<i><active_objs></i>	<i><num_objs></i>
<i>task_struct</i>	<i>101</i>	<i>154</i>
<i>mm_struct</i>	<i>76</i>	<i>99</i>
<i>filp</i>	<i>901</i>	<i>1420</i>

The difference between *num_objs* and *active_objs* is how many free objects are being tracked by the kernel

Recovery Using *kmem_cache* Analysis

- Enumeration of the lists with free entries reveals previous objects still being tracked by the kernel
 - The kernel does not clear the memory of these objects
- Our previous work has demonstrated that much previously de-allocated, forensically interesting information can be leveraged from these caches [4]

Recovering Deleted Filesystem Structure

- Both Linux kernel and aufs directory entries are backed by the *kmem_cache*
- Recovery of these structures reveals names of previous files and directories
 - If *d_parent* member is still in-tact, can place entries within file system

Recovering Previous Metadata

- Inodes are also backed by the *kmem_cache*
- Recovery means we can timeline again
- Also, the dentry list of the AUFS inodes still have entries (strange)
 - This allows us to link inodes and dentrys together
 - Now we can reconstruct previously deleted file information with not only file names & paths, but also MAC times, sizes, inode numbers, and more

Recovering File Contents – Bad News

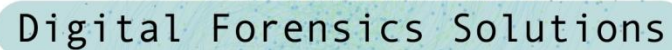
- Again, inodes are kept in the *kmem_cache*
- Unfortunately, page cache entries are removed upon deallocation, making lookup impossible
 - A large number of pointers would need to stay intact for this to work
- This removes the ability to recover file contents in an orderly manner
- Other ways may be possible, but will require more research

Summary of File System Analysis

- Can completely recover the in-memory filesystem, its associated metadata, and all file contents
- Ordered, partial recovery of deleted file names and their metadata is also possible
- Traditional forensics techniques can be made possible against live CDs
 - Making such analysis accessible to all investigators

Questions/Comments?

- Please fill out the feedback forms!
- Contact:
 - andrew@digdeeply.com
 - @attrc

A complex network diagram consisting of numerous small blue and green nodes connected by thin, light blue lines. The nodes are arranged in a roughly circular pattern, with a denser central area. A small, semi-transparent circular icon is positioned in the upper right quadrant of the network, containing a stylized globe or sphere with a few lines and dots. A light blue, rounded rectangular banner is overlaid horizontally across the center of the network, containing the text "Digital Forensics Solutions" in a dark, sans-serif font.

Digital Forensics Solutions

Linux Memory Analysis Workshop – Session 3

Andrew Case

Who Am I?

- Security Analyst at Digital Forensics Solutions
 - Also perform wide ranging forensics investigations
- Volatility Developer
- Former Blackhat, SOURCE, and DFRWS speaker
- Computer Science degree from UNO
- GIAC Certified Forensics Analyst (GCFA)

Format of this Workshop

- I will be presenting the Linux kernel memory analysis capabilities of Volatility
- Along the way we will be seeing numerous examples of Linux kernel source code as well as Volatility's plugins source code
- Following along with me while I use Volatility to recover data will get you the most out of this workshop

Agenda for Today's Workshop

1. Recovering Vital Runtime Information
2. Investigating Live CDs Through Memory Analysis
- 3. Detecting Kernel Rootkits**

Agenda for This Hour

- This session will be a walkthrough of kernel-mode rootkits under Linux
- We will be discussing the techniques used by rootkits to stay hidden and how the Volatility modules uncover them
- I will also be presenting previously never disclosed rootkit techniques developed for this workshop
- Q&A / Comments

Introducing Volatility

Volatility

- Most popular memory analysis framework
 - Written in Python
 - Open Source
 - Supports Windows {XP, Vista, 7, 2003, 2008}
 - Support Linux 2.6.9 to 2.6.3x on Intel and ARM
- Allows for analysis plugins to be easily written
- Used daily in real forensics investigations
- Will be the framework used in this workshop

Volatility Object Manager

- Once we have a model of a kernel's data structures (profiles) we can then just rely on Volatility
- Its object manager takes care of parsing the struct definitions, including types, and then providing them as requested
 - Example on next slide

Example Plugin Code

- Accessing a structure is as simple as knowing the type and offset

```
intval = obj.Object("int", offset=intOffset, ..)
```

- Volatility code to access 'descriptor' of an 'Object':

```
o = obj.Object("Object", offset=objectAddress, ..)
```

```
c = obj.Object("ClassObject", offset=o.clazz, ...)
```

```
desc = linux_common.get_string(c.descriptor)
```

Volatility Address Spaces

- Address spaces are used to translate virtual addresses to offsets within a memory capture
 - Same process used to translate to physical addresses on a running OS
- Plugin developers simply need to pass the given address space to functions that need it
 - Manual change only required to access userland (will see an example in a bit)

Current Address Spaces

- x86 / x64
- Arm (Android)
- Firewire
- Windows Hibernation Files
- Crash Dumps
- EWF Files

Linux Kernel-Mode Rootkits

Introduction

- I promise not to bore you with information from ~2002 Phrack articles...
- Rootkits target two types of data:
 1. Static
 - Easy to implement and easy to detect
 2. Dynamic
 - Harder to implement and harder to detect

Static-Data Altering Rootkits

- These rootkits target data structures that are easy to modify, but are also effective at hiding activity
- Common technique types include:
 - Directly overwriting instructions in memory (.text)
 - Overwriting the system call & interrupt descriptor tables
 - Overwriting members of global data structures

Type 1: Overwriting .text

- Very popular as its easy to implement and makes hiding data easy
- Rootkits alter running instructions for a few reasons:
 - To gain control flow
 - To filter data (add, modify, delete) to stay hidden
 - To implement “triggers” so that userland code can make requests

Detecting Code Overwrites

- The compiled code of the kernel is static
 - One exception is covered next
- The compiled kernel (*vmlinux*) is an ELF file
 - All functions, including their name, instructions, and size can be gathered from debug information
- This information can then be compared to what is in memory
- Any alteration points to malicious (or broken) software

SMP Alternatives

- There is one circumstance when runtime modifications happen in the Linux kernel
- When the computer first boots and only one processor is active, all multi-core synchronization primitives are NOP'ed out
- When more than one CPU comes online, the kernel then has to rewrite these instructions with their SMP-safe counterparts to maintain concurrency

SMP Alts. Cont.

- These alternative instructions are kept for performance reasons
 - No reason to get, set, and check SMP locks if only one CPU is active
- The alternative instructions and their target location are stored within the vmlinux file
- We can gather this information and use it for accurate .text modification checking

Type 2: System Call & IDT Overwriting

- To avoid being detected when overwriting .text, rootkits started modifying the tables used to service system calls and interrupts
- This allows for a rootkit's code to easily filter the data received and returned by native kernel functions

Attack Examples

- Overwrite the *read* system call and filter out the rootkit's logging data unless a specific register contains a magic value
- Overwrite the *stat* system call to hide files from userland anti-rootkit applications
- Many more possibilities...

Detecting These Attacks

- The IDT and the system call table are simply C arrays
- They can be copied from the clean *vmlinux* file and then compared to the values in memory
- Will easily detect that the table has been altered and which entries were modified

Type 3: Overwriting Data Structures

- Popularized by the *adore*[1] rootkit, this attack overwrites function pointers of global data structures to filter information
- *Adore* overwrites the *readdir* member of the *file_operations* structure for the *proc* and root filesystems
 - The replacement function filters out files on a pattern used by the rootkit, effectively hiding them from userland

Other Common Attacks

- Overwriting structure members used to display information through */proc*
 - Info files in */proc* use the *seq_operations* interface
 - Hijacking the *show* member of this structure allows for trivial filtering of information
- Possible targets
 - Loaded modules list
 - Networking connections (netstat)
 - Open files (lsof)

Detecting these Attacks

- We take a generic approach
- During the profile creation stage, we filter for a number of commonly targeted structure types
 - For variables found, we then copy the statically set values of each member that may be hijacked
- This ensures that all instances of those structures are checked for malicious tampering

Targeted Structure Types

- *file_operations*
- *inode_operations*
- *vm_operations_struct*
- *address_space_operations*
- Operation structures used by */proc*

Hands On

- We will look at a memory image infected with a rootkit that uses a number of static-data altering techniques
- Volatility will show us the data structures infected

Dynamic-Data Modification Rootkits

- Rootkits that modify dynamic data are much more interesting than those that alter static data
 - Require more skill on part of the rootkit developer
 - Require more complicated analysis and detection capabilities on the detector
- Cannot be detected by using System.map or vmlinux
 - Need deep parsing of in-kernel data structures

Attacks & Defenses

- The rest of this session will cover attacks and defense related to dynamic data altering
 - Most of these attacks are new (developed for this workshop) to highlight the stealth ability of these types of attacks
- But first, we need to learn about the *kmem_cache*
 - Will be used extensively by our detection mechanisms

The *kmem_cache*

- The *kmem_cache* is a facility that provides a consistent and fast interface to allocate/de-allocate objects (C structures) of the same size
- The implementation of each cache is provided by the system allocator
 - SLAB and SLUB are the two main ones

kmem_cache Internals

- Both allocators keep track of allocated and previously de-allocated objects on three lists:
 - *full*, in which all objects are allocated
 - *partial*, a mix of allocated and de-allocated objects
 - *free*, previously freed objects*
- The free lists are cleared in an allocator dependent manner
 - SLAB leaves free lists in-tact for long periods of time
 - SLUB is more aggressive

kmem_cache Illustrated

- */proc/slabinfo* contains information about each current *kmem_cache*
- Example output:

<i># name</i>	<i><active_objs></i>	<i><num_objs></i>
<i>task_struct</i>	<i>101</i>	<i>154</i>
<i>mm_struct</i>	<i>76</i>	<i>99</i>
<i>filp</i>	<i>901</i>	<i>1420</i>

The difference between *num_objs* and *active_objs* is how many free objects are being tracked by the kernel

Utilizing the *kmem_cache*

- All of the allocated objects backed by a particular cache can be found on the *full* and *partial* lists
 - The one caveat is SLUB without debugging on
 - Every distro checked enables SLUB debugging
 - Might be possible to find all references even with debugging off

The Idea Behind the Detection

- Dynamic-data rootkit methods work by removing structures from lists, hash tables, and other data structures
- To detect this tampering, we can take a particular cache instance and use this as a cross-reference to other stores
- Any structure in the *kmem_cache* list, but not in another, is hidden
 - Inverse holds as well

Why the Detection Works

- *All* instances of a structure must be backed by the caches
- These caches work similar to an immutable store:
 - Structures of the specific type cannot be hidden from it
- A few possible attack scenarios exist, but will not work undetected

Detection Subversion Scenarios

1. Allocating outside the cache
 - Will be detected by the inverse comparisons
2. Allocating in the cache and then removing from it
 - Very difficult to do and will result in detection as with scenario #1
3. Allocating in the cache and then setting the entry as free
 - The structure will be overwritten on next allocation

Our First New Attack

- The first developed attack was hiding processes from */proc*
- A number of rootkit detection systems work by trying to enumerate */proc/[1-65535]* and then compare the output to *ps*
- The numbered proc directories are backed by their respective PID namespace and number

Process Background

- All tasks are allocated from the *task_struct_cache*
- Tasks are represented by a *task_struct*
- It is possible to remove a *task_struct* from the structures */proc* uses to fill */proc/<fd>/* while still allowing it to run

The Attack

- As simple as removing from the namespace
- Code, where p is the *task_struct* we want to hide:

```
pid = p->pids[PIDTYPE_PID].pid; // get the pid ref
detach_pid(p, PIDTYPE_PID);    // take out of PID
                                // group
```

- The process will no longer show up in */proc/<pid>* lookups

Detection

- We gather processes from a number of places before comparing to those in the cache
 - Implemented in *linux_check_processes.py*
- 1) Each *task_struct* holds a pointer into the *tasks* list
- 2) The run queue, where scheduled processes wait to execute
- 3) The PID cache, where we just removed our process from

Hands-on/Demo

- We will now investigate hidden processes and look at the corresponding Volatility detection code

Next Attack: Memory Maps

- The next attack hides memory maps from */proc/<pid>/maps*
 - This file is used to list every mapped address range in a process
- Each mapping is represented by a *vm_area_struct* and they are kept in two places:
 - The *mmap* list of the processes' *mm_struct*
 - The *mm_rb* tree of the *mm_struct*

mm_struct and *vm_area_struct*

- *mm_struct*
 - Used to represent the memory management structures of a process
- *vm_area_struct*
 - Represents a single memory mapping within a process

The Attack

- Inspection of a *maps* file makes attacks such as shared library injection very noticeable
 - The full path of the mapped binary plus its data and code sections will be visible
- To hide maps, we need to:
 - Remove the *vma* from the *mmap* lists
 - Fixup the structures that account for paging
- This will hide the map and allow for the targeted process to exit cleanly

Detection

- Implemented in: `linux_check_mappings.py`
- The first step is to gather all active VMAs for a process so they can be compared against those in the cache
- The problem is that the VMAs are anonymous
 - No immediate linkage to a specific process

Detection Cont.

- To work around this, we rely on the fact that *vm*s keep a back pointer to their owning *mm_struct* in the *vm_mm* member
- Using this, we can gather all the *vm*s for a specific process and then compare against the cache
- Can you think of a bypass in this detection?

Preventing Malware Tampering

- Since we rely on *vm_mm*, malware could try to avoid this detection by changing *vm_mm*
- Possible attempts:
 1. Set *vm_mm* to some invalid value (NULL, etc)
 2. Set *vm_mm* to another processes's *mm_struct*
- Will still be detected:
 - All *mm_structs* are also in a *kmem_cache*
 - Comparing the list of *vm_mm* values to this cache will reveal avoidance attempts

Next Attack: Open files

- The */proc/<pid>/fd* directory contains a symlink per open file
 - The symlink name is the file descriptor number
- Used by a number of utilities (lsof) and anti-rootkit applications to detect files being accessed
- To remain stealthy, this directory listing needs to be filtered

The Attack

- A processes' file descriptors are stored in an array of *file* structures indexed by file descriptor number
- All non-null indexes are treated as open files
 - NULL entries are skipped

The Hiding Code

```
idx = loop_counter; // the file desc to test
file = p->files->fdt->fd[idx]; // the file struct
if (file)
{
    fn = d_path(...); // get the full path of file
    if(!IS_ERR(fn) &&
        strcmp(fn, "/tmp/hidefile.txt"))
        fdt->fd[i] = NULL;
}
```

Detection

- As with the process detection algo, finding all open files requires gathering from a number of sources:
 - The (non-hidden) open files per-process
 - The *vm_file* structures used to memory map files
 - All swap files
- We then compare these against the *filp_cache* *kmem_cache*

Demo

- Will demo showing detection of dynamic modification of:
 - Mappings
 - Open Files

Questions/Comments?

- Please fill out the feedback forms!
- Contact:
 - andrew@digdeeply.com
 - @attrc

References

[1] <http://lwn.net/Articles/75991/>