available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

Treasure and tragedy in *kmem_cache* mining for live forensics investigation

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

[a] *Digital Forensics Solutions, LLC, United States*
[b] *Neohapsis Inc., United States*
[c] *Dept. of Computer Science, University of New Orleans, Lakefront Campus, New Orleans, LA 70148, United States*

ABSTRACT

This paper presents the first deep investigation of the *kmem_cache* facility in Linux from a forensics perspective. The *kmem_cache* is used by the Linux kernel to quickly allocate and deallocate kernel structures associated with processes, files, and the network stack. Our focus is on deallocated information that remains in the cache and the major contribution of this paper is to illustrate what forensically relevant information can be retrieved from the *kmem_cache* and what information is definitively not retrievable. We show that the *kmem_cache* contains a wealth of digital evidence, much of which was either previously unavailable or difficult to obtain, requiring ad hoc methods for extraction. Previously executed processes, memory mappings, sent and received network packets, NAT translations, accessed file system inodes, and more can all be recovered through examination of the *kmem_cache* contents. We also discuss portable methods for erasing this information, to ensure that private data is no longer recoverable.

1. Introduction

The field of live forensics has grown exponentially in the last few years due to the amount of information contained in volatile storage and the impact the information can have on an investigation. On systems properly utilizing full or partial disk encryption, recovery of files and data can become impossible once host machines are powered off and keying material is lost. In the most extreme example, diskless systems, such as those booted via a LiveCD, will lose all session information once volatile memory is cleared. Even on regular systems, a wealth of information can be obtained from system memory, such as processes, file system activity, network connections and data, loaded kernel drivers, and more. Depending on the system being analyzed, old and inactive information can be also be collected, which then forms the basis of a timeline of past system activity. Due to the importance of this information both from a security and forensics standpoint, the need for complete acquisition and analysis is vital.

With this motivation in mind, we present the results of our live forensics research targeting the Linux kernel's *kmem_cache* facility as a store capable of producing memory structures deleted long in the past. Currently, no published work utilizes this store for any security or forensics-related purpose, even though it has a number of desirable forensic traits, as we discuss later. Briefly, the *kmem_cache* is an allocation system used by the Linux kernel to quickly allocate C structures of the same size. It is strongly tied to the Linux kernel's virtual memory allocator and allows rapid allocation and deallocation of kernel structures, such as those responsible for process execution, networking, and

* *Corresponding author*:
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.2010.05.006

file system activity. As we will illustrate, a number of structures of forensic interest are backed by this facility, and freed objects with useful information are easily recoverable. Old processes, memory maps, open files, NAT tables, socket information, etc. was all successfully gathered during our research effort, all of which can be used by investigators to build a picture of a machine's past operation. Since forensics is concerned with both data recovery and privacy, a simple method for sanitizing objects upon deallocation was also developed. Created as a proof of concept, this technique could easily be extended into a complete privacy system or integrated into existing security and privacy enhancing security projects. As listed in detail later, the potentially large amount of sensitive and forensically interesting data left behind in freed entries requires special attention from a privacy standpoint.

The contributions of our work include documentation of which structures are available in *kmem_cache* stores, what data can be recovered from them and what cannot, a kernel module that recovers all usable data, and a privacy module that clears memory from caches upon the release of objects in the stores.

2. Related work

Research and interest in live forensics has seen a surge in recent years with a number of publications and projects aimed at collecting volatile memory, analyzing it for forensically interesting data, cataloguing it for future use, and more (Arasteh, 2007; Dolan-Gavitt, 2007; Halderman et al., 2008). One of the main research areas of this field is the identification and gathering of old and inactive objects within a system in order to gain an understanding of past actions and this is the area the in which the presented work focuses. Since live forensics deals with volatile data that can be quickly overwritten (Chow et al., 2004; Solomon et al., 2007), any mechanism that can acquire intact, deallocated information is greatly prized by investigators. Past research in this area has taken a number of approaches, the most prominent being to simply scan a memory image for address ranges that appear to match critical C structures used by the operating system (Betz, 2005; Case et al., 2008; Garner, 2005; Schuster, 2006; Volatility, 2009). Unfortunately, this type of scanning may result in false positives due to the amount of unclassified binary data in a memory image and false negatives due to sanity checking necessary to reduce the false positive rate. To alleviate this issue, a team from Georgia Tech published a novel virtual machine monitor (Dolan-Gavitt et al., 2009) capable of recording access to members of EPROCESS structures in a Windows guest. The values and ranges assigned to these structure members were then recorded and analyzed. This allows the system to accurately and dynamically validate structures since the ranges of used members are pre-calculated and unused or unnecessary members are skipped. Through use of *kmem_cache* caches, the presented system is capable of quickly identifying all structures backed by a cache without memory scanning or reliance on software outside the native operating system. This allows first responders to quickly get a complete structure listing without having to sift through false positive and negatives.

In a past DFRWS publication (Schuster, 2008), forensic analysis of the Windows pool allocation algorithm was performed to determine how long kernel level objects stay active and in what order they are reused. While the next logical step in this would be to interact with the structures, we are aware of no work to date in this area. In a recent Phrack article (H L, 2009), the author describes in great detail the allocation and deallocation algorithms of all of the kernels allocators in a leadup to the description of his created kernel heap protection project, KERNHEAP, that is now part of the GrSecurity project. While describing these allocators, the *kmem_cache* facility is described as well as a short writeup on the sensitivity of data within the cache. Since this paper is attempting to deter reliable kernel exploitation under Linux, the author fears that the predictable, un-sanitized data within the caches could be used as static data to build a reliable exploit. He also mentions the existence of private information in other dynamic areas such as wireless keys, tty buffers, cryptographic information, and IPC. Like the previous work discussed, no specifics of the sensitive structures contained in the kernel are mentioned and no effort is made to extract the data. In (Chow et al., 2005), the persistence of sensitive data is also discussed in detail and a system presented that zeroes memory on deallocation.

3. *kmem_cache* internals

The inactive object recovery that we describe later in this paper relies on the Linux kernel's *kmem_cache* facility. This caching mechanism sits above the system memory allocator and provides quick access to C structures of the same size. Fig. 1 shows the definition of the *kmem_cache_create* function and a call to the function in the 2.6.26 Linux kernel for creating the cache that holds all *task_struct* (process descriptor) structures.

After the *kmem_cache_create* function returns, the *task_struct_cachep* structure can be used to quickly allocate *task_struct* structures within the kernel. The *kmem_cache* facility is intended to be used for structures that are allocated and deallocated often, such as those related to process handling, file system manipulation, and network processing. Under normal conditions, the use of freelists and per-cpu caches ensures that allocation can be done in O(1) time.

```
struct kmem_cache *kmem_cache_create(
        const char *,
        size_t, size_t,
        unsigned long,
         void (*)(struct kmem_cache *, void *));

task_struct_cachep =
kmem_cache_create("task_struct",
        sizeof(struct task_struct),
ARCH_MIN_TASKALIGN, SLAB_PANIC, NULL);
```

Fig. 1 – Prototype and sample call for *kmem_cache_create()* function.

The main reason that specific caches can be used for inactive object recovery is that all structures of the given type are stored in the cache and that active entries cannot be removed without compromising system stability. Entries must be allocated and kept in the cache until destruction because the built-in kernel API expects these structures to be managed by the *kmem_cache* facility. Not only would structures outside the cache have to be manually initialized, updated, and deallocated, but many built-in functions could not be used due to their expectation that the structure resides in the cache. As an example, *task_struct* has over fifty members that would have to be assigned on creation, and many of these are structures that have their own members and sub-structures that would have to be initialized. Deallocation would also be extremely difficult due to the number of subsystems that would need to be understood, affected, and modified. Concurrency control during both operations would be very complicated to achieve due to some code paths expecting to have exclusive rights over manipulation of structures and their members. Because of these issues, successfully handling a structure outside the cache for one kernel version would be a nearly impossible task, and handling in a kernel version agnostic manner would be extremely difficult. An attacker also cannot let the kernel initialize a structure and then remove it from the cache while it is still active. Since the deallocation routines treat the structure pointer as a *kmem_cache* member, the missing metadata and invalid pointer causes system instability, including kernel panics. Marking the entry as free would cause similar results as the next allocation of the structure type would overwrite the attacker's data. Since every instance of a structure is guaranteed to have originated and stayed in the cache, memory scanning is no longer necessary. Not having to carve for structures in memory, a technique used in most previous work, removes the chance of false positives and ensures that all entries are recovered.

The other reason why the kmem_cache facility is useful for inactive object recovery is that individual caches can be easily enumerated. Assuming proper concurrency controls are used, this allows for quick gathering of all inactive structures of a cache, which can then be reported and investigated.

3.1. SLAB

During our research effort, our techniques were tested against the main two system allocators under Linux, with SLAB being the first. This is the oldest of the available 2.6 allocators, but has stood the test of time and is chosen by the more conservative Linux distributions. *kmem_cache* entities are tied directly to the allocator and optimized for it. Since traversal of a specific cache is not an operation supported by the built-in SLAB API, this functionality had to be developed for our use. Our algorithm works by first traversing a specific cache's *kmem_list3* member for each online CPU. A *kmem_list3* holds three circularly linked lists of each CPU's slabs. The lists contained are *full*, in which all entries are active, *partial* which contains both active and free entries, and *free*, which contains only inactive entries. In order to get inactive entries from the cache, the *partial* and *free* lists are walked and inactive entries are added into a hash table. To determine if entries in the partial list are inactive, a bitmap is first created for the maximum number of entries in a cache. Next, the free list of the slab structure for each partial list element is traversed and corresponding entries marked in the bitmap. Finally, the bitmap is traversed, and all marked entries are added into the inactive entries hash table.

Since the presented research revolves around inactive entries in caches, studying the lifetime and amount of free entries relative to a cache's size is of obvious importance. In this regard, SLAB is very generous and will maintain caches with well over half the entries free. This allows for effective recovery of old structures, and for systems that become extremely active, and then enter non-use states, freed entries may never be reused.

3.2. SLUB

The second allocator tested was SLUB. At the time of writing, this is the latest and most efficient allocator and is currently being integrated by almost all distributions. Unlike SLAB, which does not have a defined iteration algorithm, SLUB contains a *for_each_free_object* macro that can be used to effectively iterate over all free objects.

While this macro makes traversal of the free entries trivial, SLUB has an aggressive memory reclamation algorithm that leaves very few inactive objects in caches. Where SLAB may have over half of the cache free, SLUB leaves as little as three or four entries free out of dozens or hundreds. This hampers the recovery of historical data on busy systems, since new allocations will quickly use the few available free objects.

3.3. Handling freed structures

When analyzing freed structures, special care must be taken to ensure that pointers are valid before being dereferenced. Throughout the developed kernel module, special checks are made to ensure that objects point inside the kernel and that structure members contain sane values. For the structures stored in the cache, their non-pointer members such as integers and embedded arrays, will remain intact unless explicitly reset on free. Pointers are more troublesome, as many of them are set to NULL on deallocation, thus destroying the reference to potentially valuable data, or they point to a dynamic memory address that has since been overwritten. In Section 4 of this paper, we document the members that normally contain forensically interesting data, but are either set to NULL or point to unusable data after deallocation. This information is presented to aid efficient further development of live forensics techniques utilizing the *kmem_cache*.

4. Inactive entry recovery

The following section describes our effort to recover as much useful information as possible from a number of caches within the *kmem_cache* facility. Due to the issues with free structures and structure members discussed previously, each potentially interesting structure member had to be tested individually for valid information after free.

4.1. Processes

The ability to identify and collect terminated processes allows for partial reconstruction of past activity on a system. The Linux kernel's process descriptor structure, *task_struct*, contains all the information that links a process to its open files, memory mappings, signal handlers, network activity, and more. Fortunately for investigators, all *task_struct* structures are kept within the *task_struct_cachep* inside the *kmem_cache*. Unfortunately, a number of useful members, such as *mm* and *files*, are set to NULL upon process termination. As shown later, these memory management and open file structures are recoverable through other caches, but there is no known method to reliably link these structures back to their owning process for all kernel versions.

Members that are recoverable include those necessary to get a *ps* style listing that contains the process name, user and group ID, and process ID. Fig. 2 shows a snippet of terminated processes recovered on a test system.

Linking terminated process to other tasks is hampered since the *children* and *siblings* lists are cleared upon process termination. This hampers the discovery of child processes of terminated processes as well as the processes with the same parent process as the current one. The *parent* member is active in the structure, however, and can be used to walk backwards through the successive list of executed processes.

Timelines of user activity make forensics investigations quicker and more accurate as the investigator can accurately gain the "big picture" of activity and join single actions into large groups of events. To help enhance timelining capabilities, the developed gathering module records the time the process started, how long it ran, and how much CPU time was used. This information can be gathered from the time members of *task_struct* and by replicating algorithms found in the kernel's task accounting code. The following figure shows the output of the task gathering module after having run a process named *SomeApp*.

As can be seen in Fig. 3a, *SomeApp* was run immediately after 13:24:01 and later on the recovery module was run. The reported output of the recovery module is shown in Fig. 3b and reports on the timing information of the previously terminated process. Extraction and conversion of the reported start time reports 13:24:02 which is 1 s off the *date* command. This 1 s discrepancy is expected since the date command ran before the *SomeApp* command.

NAME	PID	UID	GID
apt-cache	6306	0	0
ld	6386	1002	1002
lsb_release	6196	0	0
kstopmachine	6343	0	0
apt-cache	6298	0	0
lsb_release	6330	0	0
sh	6354	0	0
lsb_release	6353	0	0
as	6391	1002	1002
vim	6241	1002	1002
apt-cache	6403	0	0
as	6361	1002	1002
ld	6363	1002	1002

Fig. 2 – Listing of previously exited processes obtained by mining the *kmem_cache*.

a
```
Sun Apr 11 13:24:01 EDT 2010
SomeApp pid: 1340
```

b
```
debian kernel: [100187.829351] SomeApp 1340
1271006642 20512635
```

c
```
debian:~/slbwalk# date -d "@1271006642"
Sun Apr 11 13:24:02 EDT 2010
```

Fig. 3 – a. Running SomeApp which reports PID of 1340; b. The information reported by the gathering module about the process (start time in bold); c. Converting the reported start date to human-readable format.

Combination of the two process recovery modules would allow for not only reconstruction of processes, but also precise timelining of the recovered tasks.

4.2. Memory mappings

When a process is created, its stack, heap, main executable code, data, and shared libraries are mapped, using the *mmap* facility, into the new address space. Inside the kernel, each mapping is represented by a *vm_area_struct* and the set of mappings for a process are stored in a linked list that is contained in the process's *mm_struct*. All *mm_struct* structures are contained within the *mm_cachep* allocated by the *kmem_cache* and can be used to gather the mappings, command line arguments, environment variables, and other memory information for a single process. Unfortunately, until kernel release 2.6.26, there was no method for linking *mm_structs* to their respective *task_struct*. This means that while much information can be gathered about a process' memory, there is no automatic method to associate it with a specific process structure. In kernels since 2.6.26 with cgroups enabled, a new member, *owner*, points to the owning *task_struct* and can be used to link process and memory descriptors together programmatically. Even when the pointer to the process structure is missing, manual analysis of the contents can reveal a wealth of information. Manual linking of structures with processes may also be possible as the mappings contain the name of the binary loaded.

4.3. Open files

Gathering information about open files, including the process that opened a file, the full path of the open file, and its owner are potentially very useful pieces of evidence in a forensics investigation. Unfortunately, all of this information is lost during the deallocation process for file structures. While some information, such as the file's user and group ID and open mode can be retrieved, the more useful information like the file's name is unrecoverable since the directory entry pointer is set to NULL on free.

4.4. Filesystem inode caches

Every filesystem driver maintains a *kmem_cache* to track active inodes. Traversal of the free entries of these caches will reveal previously opened files and possibly the name of files that have

```
debian:~/slabwalk# insmod ./slabwalk.ko
debian:~/slabwalk# head -5 /var/log/messages
kernel: [35566.045181] inode: 106310 0 0
kernel: [35566.059469] inode: 106312 0 0
kernel: [35566.071471] inode: 139091 0 0
kernel: [35566.082007] inode: 106308 0 0
debian:~/slabwalk# ffind /dev/sda1 106310
/usr/share/zoneinfo/posix/America/Fortaleza/tmp
/cceZLcAc.o
debian:~/slabwalk# ffind /dev/sda1 139091
/var/run/sshd
debian:~/slabwalk# ffind /dev/sda1 106308
/usr/share/zoneinfo/posix/America/Fortaleza/tmp
/cceoInI5.o
```

Fig. 4 – Traversing the ext3 inode cache and using the Sleuthkit to obtain filenames.

been deleted since they were last used. Unfortunately, the normal algorithm for gathering the names of opened files, walking the list of the inode's directory entries, is not usable since the list is cleared on deallocation. Recovery of the names can still be achieved though with the help of the Sleuthkit, however, since the inode number is still accurate in the structure. Fig. 4 shows the results of traversing the free inode cache and then passing the inode number of free objects to *ffind* of the Sleuthkit, to perform a "dead" analysis of the inode in a filesystem image.

When testing this module, the ext3 filesystem was used and its inode cache, *ext3_inode_cachep,* was traversed. This cache actually contains *ext3_inode_info* structures, and these structures embed a regular inode structure. The *inode* structure allows the module to gather an inode's owner, group, mode, and inode number. Investigators can use this information to discover recently opened files, the permissions the file was opened with, and which user opened the file.

4.5. Socket buffers

FACE (Case et al., 2008) illustrated that both un-transmitted and recently received network packets can be retrieved by walking the receive and send queues of a socket structure. The work described in that paper leverages this technique to tie data from socket buffers and network dumps together for further analysis. Unfortunately, these queues are emptied on free, but the individual socket buffers are stored in the *skbuff_head_cache* backed by the *kmem_cache*, which stores *sk_buff* structures. Enumeration of the inactive structures in this cache produces previously sent and received packets from the host machine and can be used in conjunction with FACE to directly match packets in a network dump with packets in the caches.

Inside of each *sk_buff* is a struct *sock* that can be used to obtain the *inet_sock* for the current socket buffer. Since the *sock* member is a pointer to dynamically allocated memory, there is a chance that the values inside of it could be corrupt. In the cases that it isn't corrupt, which is verified by ensuring the address is inside the kernel, the destination and source IP address and port can be recovered. This can be used to help tie the specific socket buffer to a particular process or daemon. The address of the *sock* structure can also be used to group socket buffers from a process together since each will point to the same address.

4.6. Bound sockets

Before network servers can start accepting connections, they must *bind()* to a network port. In order to facilitate fast lookups of open and in-use ports, the Linux kernel tracks each in-use port within a quickly searchable data structure. For the TCP protocol, each port is kept within the *bind_bucket_cachep* of the *tcp_hashinfo* structure. This cache holds *inet_bind_bucket* structures whose member *port* contains the listening network port. This structure also contains a list of *sock* structures that reference the port, but unfortunately this list is cleared on deallocation. Enumeration of inactive entries of this cache will reveal ports that were previously used to accept network connections. In investigations where network capable malware was either present on the system or suspected to be present, this cache may be useful to help prove or disprove its existence. The port numbers can also be used to quickly point an investigator to interesting data within a network capture. Finally, combined with the network caches presented in the rest of this section, an investigator can gather a large amount of information related to past network activity solely from the machine under investigation. This will save investigative time normally spent examining server, IDS, and router logs.

4.7. Netfilter NAT table

The last cache explored, *nf_conntrack_cachep*, stores the Netfilter connection tracking information in *nf_conn* structures. Netfilter (netfilter.org) is the underlying framework for packet filtering in the Linux kernel, and the popular firewall technology, IPTables, is built upon it. In order to provide NAT capabilities, the connection tracking module must store the source and destination IP address and port for each translated connection. In the current Linux implementation, the incoming and outgoing translation for each connection is stored within the *tuplehash* member of each *nf_conn* structure. By enumerating the connection tracking cache and walking each connection tuple, expired network translations can be fully recovered. The following figure shows a snippet of the output generated by the gathering module after having previously connected the test system to a number of different Google webserver IP addresses on TCP port 80 (Fig. 5).

The implications of recovering these entries are significant since this information could only be previously gathered reliably by monitoring network traffic at the time of the connections. By using these newly developed capabilities, investigators can instantly build a list of past connections on the target machine. In investigations involving network gateways, historical subnet network activity can be easily gathered from one module.

```
src: 192.168.181.132 255  dst: 74.125.95.35 80
src: 192.168.181.132 255  dst: 74.125.95.18 80
src: 192.168.181.132 252  dst: 74.125.95.21 80
src: 192.168.181.132 252  dst: 74.125.95.41 80
src: 192.168.181.132 252  dst: 74.125.95.42 80
src: 192.168.181.132 252  dst: 74.125.95.22 80
src: 192.168.181.132 252  dst: 74.125.95.17 80
src: 192.168.181.132 252  dst: 74.125.95.44 80
```

Fig. 5 – Freed NAT translation table entries.

Currently this must be accomplished by gathering logs of upstream hosts such as ISP routers or end-servers.

5. Privacy implications/fixes

Besides recovery and analysis, digital forensics is also concerned with the privacy implications of using computer systems. With this in mind, techniques were developed to clear memory for *kmem_cache* backed structures upon deallocation. To facilitate quick development of this feature and mass portability, the *jprobes* (http://sourceware.org/systemtap/kprobes/) facility was used to hook the *kmem_cache_free* function. This function is responsible for freeing an instance of a structure from a specific cache. Jprobes allows a kernel module to gain control of execution before a specific function executes. It is much nicer than the closely related *kprobes* project in that it allows for regular C function parameters to be used in development instead of simply getting an instruction-only interface to the called function. Fig. 6a shows the code used to initialize our jprobe structure and 6.b illustrates how *clear_memory* can use the same prototype as the function it is hooking.

While this method looks promising for sanitization of freed structures, it is still under development. The original algorithm was designed to use the *objsize* member of the *kmem_cache* structure in order to get the size of objects in the cache. *Objsize* bytes starting at the *addr* parameter value would then be cleared. Since the *addr* parameter points to the structure being freed, it was believed that zeroing this memory range would effectively and safely clear freed structures. While the algorithm worked on all self-generated tests cases and also for clearing *task_struct* structures as they were freed, when set to clear structures of all types, memory corruption quickly caused kernel panics on both SLAB and SLUB systems. The cause of this memory corruption is still being investigated, but we believe we are a close to a full solution on SLAB systems and that SLUB will either require a different algorithm or may not be possible without modifying kernel source.

When the difficulties with hooking *kmem_cache_free* were discovered, another solution was simultaneously researched. The second approach developed borrowed ideas from the GrSecurity (http://www.grsecurity.org/) kernel configuration option that sanitizes *all* pages upon free with only a 3% performance penalty. This feature modifies both the *__free_pages_ok()* and *free_hot_cold_page()* functions in the kernel in order to clear each page before these functions return. Similar logic was implemented inside of a new jprobe hook in order to free all pages as the GrSecurity feature does. The developed solution has the obvious advantage over the GrSecurity project as it can be used on any running system without modifying kernel source code and without requiring kernel recompilation.

Once entries are cleared upon free, the data recoverable in Section 4 would no longer be present, hindering live forensics investigations greatly. This situation embodies the classical tradeoff between the need for privacy and the valid use of digital forensics to prosecute crime.

a
```
static struct jprobe my_jprobe = {
        .entry = clear_memory,
        .kp    = {
        .symbol_name = "kmem_cache_free",
        },
};
```

b
```
static void clear_memory(struct kmem_cache *s,
                         void *addr)
```

Fig. 6 – a. Illustration of jprobe initialization code. b. Illustration of *clear_memory* function prototype.

Distribution	Kernel Version	Allocator
Ubuntu	2.6.24–23-generic	SLUB
Debian	2.6.26-2-686	SLAB
Debian	2.6.26 vanilla	SLAB

6. Evaluation

Our research and development was performed on two VMware Workstation virtual machines. The following table shows the distribution, kernel version, and allocator used for each system.

A large number of tests systems were not needed since the main feature being tested was SLAB versus SLUB, and these allocators have not experienced major internal changes. An extra Debian kernel was used because the default 2.6.26-2-686 kernel did not have kprobes enabled, so a vanilla 2.6.26 kernel was compiled with this feature in order to test it on SLAB. Much of our effort involved observing the values of structure members of interest in order to determine if they could be reliably used to recover freed data. Once the set of usable members is determined, regular kernel algorithms, such as process enumeration, memory mapping walking, and open file handling, can be used to gather freed objects and determine their values.

Each capability was tested to ensure that reliable results were being reported. For processes, it was verified that each of the reported fields was accurate. Memory mappings were checked to ensure that reported information such as code, heap, and stack addresses were correct and that the conversion methods for modes and sizes were accurate. Inode recovery was verified using the Sleuthkit. Networking information was compared to netstat output collected while the connections were still active and similarly the netfilter translations were compared to output of */proc/net/nf_conntrack* before the translated connections had terminated.

7. Future work

The goal of this research was to explore the feasibility of recovering inactive *kmem_cache* structures to aid live forensics investigations. While this effort proved successful, in order to be useful to less technical investigators, an application must be developed that provides an appropriate interface to use our techniques. A visual interface would not only help investigators quickly absorb the data, but would also help them

manually link together groups of structures that cannot currently be grouped programmatically (essentially, the human provides the "deductive leap" necessary for such grouping). Once this interface is developed, timelining of data will be our next task. Having an ordered view into past events will give investigators immediate insight and direction while analyzing complex cases.

During this project, we attempted to identify all forensically interesting information that could be gathered from *kmem_cache* objects. Due to the number of subsystems in the kernel and the rapid pace in which changes are made to the kernel, finding all relevant objects in one pass is difficult. After accomplishing our previously stated goals, we are now re-examining the entire set of objects backed by the *kmem_cache*. The greater the amount of historical information gathered, the less investigators have to guess to deduce past user activities and the greater the context current data can be placed in. All of this leads to quicker and more thorough investigations, which can be performed by ordinary (read: non-kernel hacking) investigators.

Determining the most appropriate algorithm to use to clear sensitive memory on release will also be a key priority. The amount and sensitivity of free information that can be gathered through *kmem_cache* analysis suggests the importance of a facility to sanitize data that is no longer in use. The choice of jprobes to perform the sanitization seems like the most promising technology as it is non-intrusive and is immediately portable to other systems with probes compiled in the kernel. Its C interface to function hooking also makes it extremely desirable since instruction level work is completely avoided.

8. Conclusion

We have presented, from both an investigative and privacy perspective, the importance of freed information contained within objects backed by *kmem_cache* structures. While previous work has been able to recover free information through unorganized methods, our techniques are the first to mine the *kmem_cache* facility in an organized way to support live forensics investigations. Previously executed processes, memory mappings, sent and received network packets, NAT translations, accessed file system inodes, and more can be recovered through enumeration of specific *kmem_cache* caches. The implications of this capability for live forensics are large, as this information was previously either completely unavailable or difficult to obtain, requiring ad hoc methods. In order to protect the privacy of users, we also discussed portable methods for erasing this information. This ensures that private information is no longer recoverable once it has served its purpose and cannot be leveraged by attackers to steal sensitive information.

REFERENCES

Arasteh A. Forensic memory analysis: from stack and code execution history. In: Proceedings of the 2007 Digital Forensic Research Workshop (DFRWS), 2007.

Betz C. MemParser, 2005. http://sourceforge.net/projects/memparser/

Case A, Cristina A, Marziale L, Richard G, Roussev V. FACE: automated digital evidence discovery and correlation. In: Proceedings of the 2008 Digital Forensic Research Workshop (DFRWS), 2008.

Chow J, Pfaff, B, Garfinkel T, Christopher K, Rosenblum M. Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX security symposium, August 2004.

Chow J, Pfaff, B, Garfinkel T, Rosenblum, M. Shredding your garbage: reducing data lifetime through secure deallocation. In: Proc. 14th USENIX Security Symposium. Aug. 2005, p. 331–346.

Dolan-Gavitt B, Srivastava A, Traynor P, Giffin J. Robust signatures for kernel data structures. In: Proceedings of the ACM conference on Computer and Communications Security (CCS), 2009.

Dolan-Gavitt B. The VAD tree: a process-eye view of physical memory. In: Proceedings of the 2007 Digital Forensic Research Workshop (DFRWS), 2007.

Garner G. kntlist, http://www.gmgsystemsinc.com/knttools/, 2005

H L. Linux kernel heap tampering detection. In: Phrack 66, Article 15, 2009.

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold boot attacks on encryption keys; 2008.

Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. In: Digital Forensic Research Workshop (DFRWS), 2006.

Schuster A. The impact of Microsoft Windows pool allocation strategies on memory forensics. In: Proceedings of the 2008 Digital Forensic Research Workshop (DFRWS), 2008.

Solomon J, Huebner E, Bem D, Szezynska M. User data persistence in physical memory. Digital Investigation June 2007;4(2): 68–72.

Volatility. https://www.volatilesystems.com/default/volatility/, 2009.

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

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

Cris Neckar is a Senior Application Security Consultant at Neohapsis. He is also an Adjunct Professor of Computer Science at DePaul University.

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.