Treasure and Tragedy in kmem_cache Mining for Live Forensics Investigation

By

Golden Richard, Andrew Case, Lodovico Marziale and Cris Neckar

Presented At

The Digital Forensic Research Conference

DFRWS 2010 USA Portland, OR (Aug 2nd - 4th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment. As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http:/dfrws.org

Treasure and Tragedy in *kmem_cache* Mining for Live Forensics Investigation

Andrew Case, Lodovico Marziale, Cris Neckar, Golden G. Richard III

Senior Security Researcher, Digital Forensics Solutions

andrew@digdeeply.com

Current State Of Linux Memory Analysis

- Finding structures of interest requires walking lists, hash tables, trees, etc
- This is quite problematic
  - Requires understanding of a number of different kernel subsystems
  - Only can find allocated structures
- Finding de-allocated structures requires ad-hoc scanning based on patterns of what the structure may look like
  - Structure representation changes drastically between kernel versions
  - Slow – have to scan all physical memory

What is the *kmem_cache*?

- Facility that allows for quick allocation of C structures within the Linux kernel
- Used for allocation of a number of interesting structures related to process handling, networking, file system interaction, etc
- Implementation defined by chosen memory allocator (SLAB, SLUB, …)

Why Do We (Investigators) Care?

- On de-allocation of structures, memory is not (fully) cleared
- All sorts of forensically interesting information is contained within these structures after de-allocation
  - This is what the rest of the talk will focus on
- Specific caches (of structures) are trivial to locate and enumerate
  - All instances of a structure (both allocated and deallocated*) can be instantly found and analyzed
  - No more using patterns and other tricks to find structures in memory

Steps to Find Useful Information

- Search across the kernel for usage of the *kmem_cache* (LXR and grep)
- Using kernel internals knowledge, determine if a particular cache has any forensics use
- If so, walk the specific cache, and access members that contain the useful information
- Output this information in a human readable way
- …
- If only it were this easy

Challenges and Limitations

- When each structure is freed, a number of the pointer members are set to NULL or point to freed (and possibly overwritten) data
  - Before analysis can be, each member of interest must be checked for valid information
  - This issue erases a wealth of information
- Allocators have vastly different reclamation algorithms
  - Current version of project supports SLAB and SLUB (next slides)

SLAB

- Oldest available allocator, slowly being moved away from
- Leaves large caches with numerous free entries
- Enumeration algorithm
  - Not directly supported by API
  - Ours begins by traversing a cache's *kmem_list3,* which contains three lists
    - Full – all entries are allocated
    - Partial – mix of free and allocated entries
    - Free – all entries are deallocated
  - All entries from *free* are analyzed and entries from *partial* are checked against each slab's freelist and then processed if free

SLUB

- Newest kernel allocator and all distributions except Debian use it in latest kernel release
- Much more aggressive freed entry reclamation
  - For both SLAB and SLUB, information about each cache, including the number of allocated and free entries, can be obtained from */proc/slabinfo*
- Provides a built in enumeration function *for_each_free_object*

Recovering Old Processes

- Open files and memory mappings are not directly accessible since the pointers are NULL
- Process, user, group and other information seen in ‘ps’ is recoverable
- Timestamps are recoverable (illustrated next)
- Enumeration of the *task_struct* cache also reveals thread structures
  - Not talked about in many forensics publications

Timelining an Old Process

1) Running the application

```
# date; ./SomeApp
        Sun Apr 11 13:24:01 EDT 2010
        SomeApp pid: 1340
```

2) Later running the deleted process recovery module

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

3) Converting the start time

```
# date -d "@1271006642"
        Sun Apr 11 13:24:02 EDT 2010
```

Recovering Memory Mappings

- Contained in *struct mm_struct*
- Each mapping (*vm_area_struct)* can be recovered by walking *mmap* list
- Back link to task is broken on de-allocation
  - Can often be manually determined
- Stack, heap, etc are stored in userland, making their chance of recovery slim
  - Would have to parse page tables based on old CR3 value

Recovering “Open” Files

- Only memory mapped files used by multiple processes can files be recovered
- In all other cases, the directory entry structure is cleared
  - This removes the name, inode, and other file system information about the file
- Even if the file pointer in *task_struct* was valid, all file handles are closed on process exit

File System Inode Caches

- Every Linux file system driver keeps a cache of recently used inodes
- These inodes can be used to find deleted files and files that were recently in use on the system

Walking the Ext3 Inode Cache

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

Socket Buffers

- Each to be sent or received packet is represented by a *struct sk_buff*
- Normally a socket's queues of to be processed packets can be enumerated
- Unfortunately, these queues are emptied as all packets are handled or as the socket is closed
- The *sk_buff* structures can still be recovered by enumerating the *skbuff_head_cache,* revealing past packets
- Groups of *sk_buff* structures can be linked together by their *sock* structure if it is not overwritten

Gathering Old Socket Buffers

```
# perl -e 'print "SOMETHING_INTERESTING"x10240' | nc
    www.digdeeply.com 80 > /dev/null
# perl -e 'print "SOMETHING_BAD"x10240' | nc www.digdeeply.com
    80 > /dev/null
# insmod ./slabwalk.ko
# dmesg | tail -2
    [10326.272906] Found:
    SOMETHING_BADSOMETHING_BADSOMETHING_BADSOMETHIN
    G
    [10326.273733] Found:
    SOMETHING_INTERESTINGSOMETHING_INTERESTINGSOMETH
```

Netfilter NAT Table

- Netfilter is the underlying framework for packet filtering on Linux (Iptables)
- When NAT is in use, each translation is tracked by a *nf_conn* structure
- The *tuplehash* member of each *nf_conn* stores the translation's source and destination IP address and port pair
- Recovery of these entries reveals past connections on a machine

Examining NAT Output

```
# host digdeeply.com
   digdeeply.com has address 64.202.189.170
# nmap digdeeply.com 2>&1 > /dev/null
# insmod ./slabwalk.ko
# tail -4 /var/log/messages
   src: 192.168.181.132 226  dst: 64.202.189.170 247
   src: 64.202.189.170 247  dst: 192.168.181.132 226
   src: 192.168.181.132 255  dst: 64.202.189.170 190
   src: 64.202.189.170 190  dst: 192.168.181.132 255
```

Privacy Concerns

- This research raises some obvious privacy concerns
- Unsecure deletion is great for forensics investigators, but not so great for ordinary users
- We investigated a number of ways to securely erase cache entries on de-allocation, with JProbes being chosen as the eventual answer
- Unfortunately, its too much to cover in this talk, but full details of the work done is in the paper.

Future Work

- Exploration of more structures backed by the kmem_cache
- Integration of capabilities into Volatility
  - Current analysis is done by an LKM on the target system, but this could be easily ported to the Volatility plugin framework
- Automate manual linking of structures to overcome freed pointer issues

Conclusions

- *kmem_cache* entries contain a wealth of information
- On Linux systems, no longer need patterns to search for different structure types since the *kmem_cache* allows easy enumeration of both allocated and de-allocated structures
- Timelining of past information in memory can now be done much easier due to the number of time related information contained in cache backed structures

Questions/Comments

- andrew@digdeeply.com