SLUB Artifact Retriever – Emphasizes retrieving forensic artifacts in SLUB allocator environments.

Author: Sunny Thakur

Table of Contents

Abstract

Memory forensics provides investigators with a way to analyze volatile memory (RAM), offering a snapshot of the system's state as it was actively running. Key elements of interest in memory samples include kernel data structures, which represent processes, files, and network sockets. The SLUB allocator, now the standard for small memory requests in modern Linux systems, allocates "slabs" — contiguous blocks of pre-allocated memory — to efficiently handle these requests. Unlike its predecessor, the SLAB allocator, which tracked every slab, SLUB does not always retain information on full slabs, posing a challenge for memory forensics. Without this tracking, investigators face difficulties in identifying and extracting allocated slabs.

To address this, we developed a method that combines memory carving with linked list enumeration to locate SLUB-allocated slabs. This approach involves locating allocated objects and exploring adjacent memory to find additional similar objects. We implemented this method as a Volatility plugin, named "slab carve," which can retrieve artifacts from memory. Adding this plugin to the Volatility framework provides investigators access to information previously inaccessible following Linux’s transition from SLAB to SLUB. This enhanced capability can aid in reconstructing system state, tracking malicious activity, and recovering traces of malware on compromised systems.

Chapter 1: Introduction

1.1 Memory Forensics

Digital forensics encompasses a range of techniques and tools aimed at providing insight into activities performed on computers. Memory forensics is a specialized branch of digital forensics that focuses on analyzing the volatile memory, or RAM, of a machine. Investigations typically start with a memory sample — a copy of a machine's volatile memory, which can be acquired using open-source or commercial tools on physical systems. For virtualized environments, such as virtual machines, the host system can capture a memory sample by suspending the virtual machine directly.

Memory analysis allows investigators to gain insights into the active state of the machine, capturing data that may not be written to disk. By examining memory, investigators can extract artifacts like active processes, network connections, passwords, open file handles, and system hooks. This enables a comprehensive reconstruction of activities, identifying which users performed which actions and whether malware played a role. Memory forensics can even detect malware that exists exclusively in memory without leaving traces on disk.

Specialized tools, such as Volatility, play a crucial role in memory forensics. Volatility is an open-source, plugin-based memory forensics framework with tools for analyzing memory samples from Windows, MacOS, and Linux systems. It provides key functionalities, such as translating between virtual and physical addresses, data overlays for kernel objects, and an interactive shell for memory sample inspection. Volatility plugins can also build on each other's results, allowing a sequential data analysis pipeline.

1.2 The Linux Operating System

Linux is widely used, powering approximately 25% of software development workstations, 80% of web servers, nearly all supercomputers, and Android-based devices. As an open-source operating system, Linux's codebase is accessible for both use and contribution, facilitating transparent analysis of its kernel behavior.

Linux is responsible for managing memory for both system processes and user applications. Memory allocation requests vary in size, and the operating system must efficiently allocate and track these resources. Memory is organized into 4096-byte pages, which serve as the basic units of memory management. Accessing data within pages requires both a page number and an offset within that page. For larger memory requests, Linux uses the Zone allocator, while smaller allocations are handled by one of several SL*B allocators (e.g., SLAB, SLUB, or SLOB), with the choice of allocator configured at kernel compile time.

1.3 Motivation

Prior to the adoption of SLUB, the Volatility framework included plugins like *pslist cache*, which used slabs allocated by the SLAB allocator to extract process information — a key component in forensic analysis for tracking system actions by users and malware alike. However, changes introduced in SLUB meant that the *pslist cache* plugin could no longer extract process information from systems running SLUB. Without automated tools to analyze SLUB-allocated memory, we aimed to develop a new tool to perform similar forensic artifact extraction as *pslist cache*.

1.4 Research Importance

While previous slab analysis plugins focused on extracting process information, slabs hold other critical kernel data structures, including socket and file system caches. Recovering these artifacts is essential for forensic analysis as it can uncover residual data from previously terminated processes or locate objects removed from kernel tracking lists — a behavior often employed by malware.

1.5 Notation

Terms in **bold** represent Linux kernel structures, helping to differentiate between the concept of a "page" as used in operating systems and the specific kernel data structure representation. Terms in *italics* refer to specific Volatility plugins or shell commands. Plugin names are referenced in their Linux-specific versions when relevant.

1.6 Outline

Chapter 2 provides an in-depth technical background on SLUB, Volatility, and relevant memory forensics research. Chapter 3 outlines the methodology and experimental setup. Chapter 4 presents the research results and their implications, and Chapter 5 concludes with a summary of the findings and suggestions for future work in Linux memory forensics.

Chapter 2 : Background

This chapter covers crucial background information on Linux memory allocation, Volatility framework, and memory forensics, which form the foundation for understanding modern memory forensic analysis techniques. Each section delves into the mechanisms and tools integral to analyzing memory, especially when dealing with incident response and digital forensics for systems compromised by malware or other threats.

2.1 Linux Memory Allocation

The memory allocation process in Linux initiates with the kernel calling the kmalloc function, which checks the request size to route it to either the Zone allocator or the SL*B allocator. Among these allocators:

- **SLUB** supports up to 2-page large allocations.
- **SLAB** and **SLOB** support up to 1-page large allocations.

```
static __always_inline __alloc_size(1) void *kmalloc(size_t size, gfp_t flags)
{
	if (__builtin_constant_p(size)) {
#ifndef CONFIG_SLOB
		unsigned int index;
#endif
		if (size > KMALLOC_MAX_CACHE_SIZE)
			return kmalloc_large(size, flags);
#ifndef CONFIG_SLOB
		index = kmalloc_index(size);

		if (!index)
			return ZERO_SIZE_PTR;

		return kmem_cache_alloc_trace(
				kmalloc_caches[kmalloc_type(flags)][index],
				flags, size);
#endif
	}
	return __kmalloc(size, flags);
}
```

Figure 2.1. kmalloc implementation as of Linux 5.17

2.1.1 Linux Small Request Allocators

- **SLAB**: The original small memory allocator, SLAB, was the default until Linux kernel 2.6.23. It creates caches that contain slabs (represented by struct page), grouping objects of similar types within the same cache. It manages its slabs in three states: fully allocated, partially allocated, and free.

- **SLOB**: Designed for low-memory environments, SLOB uses a simple heap with aligned object return, making it lightweight and suitable for embedded devices.
- **SLUB**: The SLUB allocator, which replaced SLAB, optimizes allocation by grouping objects by size rather than type. It includes APIs to create custom caches, and it only tracks fully allocated slabs if the CONFIG SLUB DEBUG option is enabled. Forensically, this lack of tracking means that slabs' contents are not directly linked to their physical memory pages, requiring additional analysis.

2.1.2 Linux Memory Allocator Internals

Within SLUB:

- kmem_cache holds metadata like size, flags, and object order (oo), which optimizes the use of slabs by relating page size to object size.
- The kmem_cache_cpu structure, unique to each CPU, tracks the active slab and provides a freelist pointer for allocations.
- kmem_cache_node maintains lists of partially and fully allocated slabs, streamlining access and management.

2.2 Volatility and Memory Forensics

The Volatility framework is an open-source tool for analyzing volatile memory. By using custom plugins and profiles, Volatility can interpret OS memory layouts, handle virtual and physical address translations, and analyze kernel objects.

2.2.1 Linux Volatility Profile Creation

For accurate analysis, Volatility requires a kernel-specific profile. In Linux, profiles are created by compiling a kernel module with debug symbols, extracting type information, and combining it with System.map data.

2.2.2-2.2.4 Key Volatility Plugins

- **pslist**: Enumerates active processes in the kernel's task list.
- **pslist_cache**: Identifies inactive or hidden processes by carving task_struct objects from the SLAB cache.
- **netstat**: Examines network sockets, identifying connections to unusual addresses or suspicious processes.

2.3 Related Works

This section reviews research relevant to memory forensics, focusing on tools and techniques to ensure integrity during acquisition and analysis.

2.3.1 Resilient Memory Acquisition Analysis

To prevent malware from disrupting memory acquisition, **Stüttgen and Cohen** introduced hardware-based acquisition methods that bypass the OS, evading detection.

2.3.2 Acquiring Firmware Through Memory Acquisition

Firmware-level malware, like bootkits, requires specialized memory acquisition techniques. Stüttgen's team developed methods to capture firmware memory regions, enhancing firmware analysis for forensics.

2.3.3 Fuzzing Tools for Sample Corruption Resilience

Case et al. created a framework to simulate memory smearing, allowing for robust testing of forensic tools under typical corruption conditions.

2.3.4 Analyzing Compressed Memory

Modern OSes use compressed memory to extend physical memory. Work by **Richard and Case** and **Fire Eye** on reverse-engineering compressed pages ensures that crucial data remains accessible for analysis.

2.3.5-2.3.9 Additional Studies in Memory Forensics

These studies investigate various aspects of memory forensics, including:

- Extracting inactive objects from kmem_cache for forensic analysis.
- Fingerprinting user-space heaps to retrieve sensitive data.
- Analyzing processes in the Windows Subsystem for Linux (WSL), which runs Linux processes within Windows.
- Improving pool tag scanning efficiency in Windows memory.
- Identifying malware by analyzing suspicious memory page properties, such as those set to EXECUTE_READWRITE for code injection.

These studies provide a comprehensive understanding of Linux memory allocation, forensics, and related tools, setting a solid foundation for further analysis in this research.

Chapter 3. Methodology

3.1 Extracting Objects From Slabs

When analyzing a known kmem_cache slab, objects are positioned at the start and every kmem_cache.size bytes thereafter. By parsing in steps, we can extract all potential objects within the slab. While the exact type of extracted object is not immediately identifiable, overlaps of valid byte data across different kernel objects are rare. This rarity allows for easy identification of "junk" objects with invalid values, which do not interfere with further analysis.

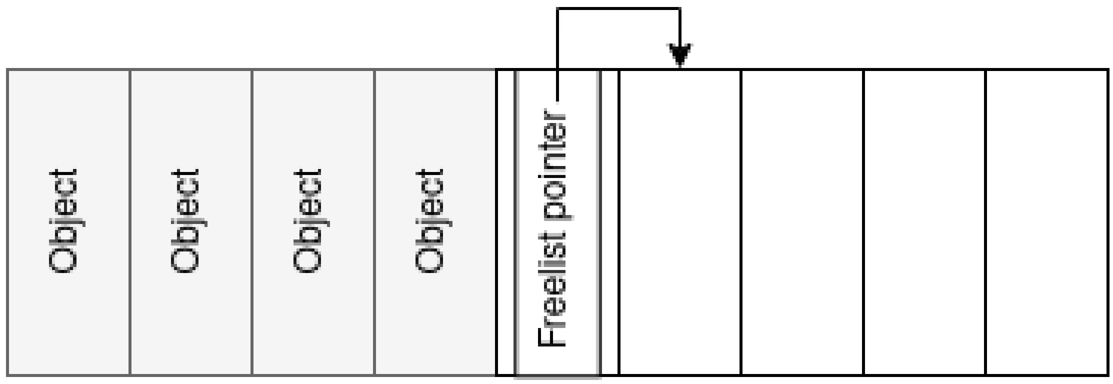


Figure 3.1. Diagram of slab layout

3.2 Getting the Needle into the Haystack

In cases where slabs are inaccessible via their lists, we can locate their objects using alternative kernel lists, such as the process list. These object addresses can be obtained through other Volatility plugins, like pslist and netstat. With the locations of these objects, slab carving can begin at the identified objects rather than from the slab's start. Although we lack precise object positions within the slab, the maximum memory range for the slab can be estimated. This range is calculated by multiplying the slab's page count (from kmem_cache) with the architecture-specific page size. A stepped carving in both directions from the object will ensure complete extraction of objects within the slab. Target objects, along with any "junk" data, can later be validated by the analyst or filtered.

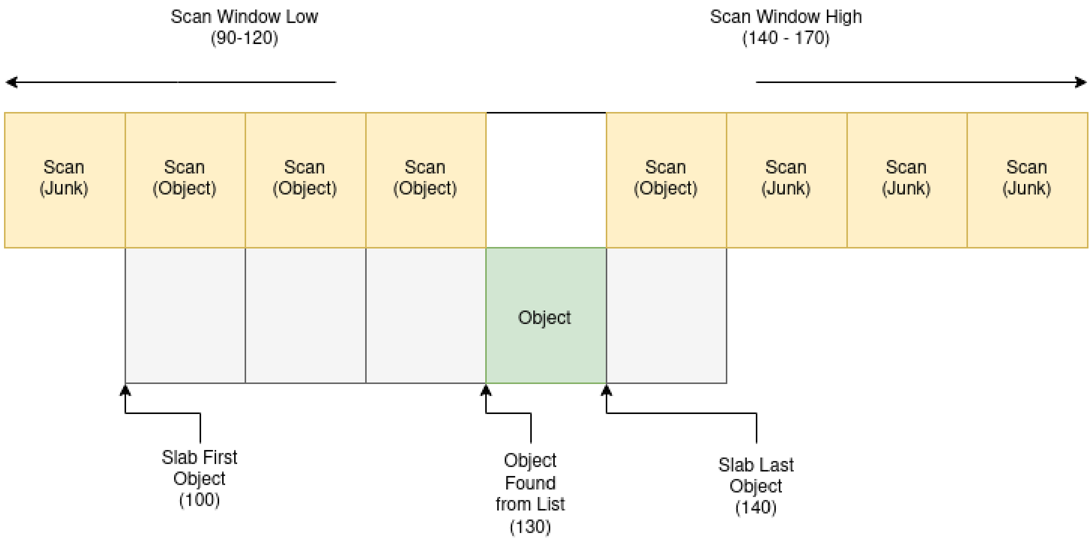


Figure 3.2. Diagram of carving approach

While this approach will lead to extracting both target objects as well as junk, the objects can be validated after, either by the analyst or a filter.

3.3 Experimental Setup

The testing environment consists of Linux virtual machines hosted on VMware. Memory samples were obtained by taking snapshots of these virtual machines. Analyses were conducted on an Ubuntu 21.04 machine, using the Volatility tool via command line. The Linux target systems are outlined in Table 3.1.

Table 3.1. Linux versions used for testing.

Sample	Linux Kernel	Distribution	uname -a
1	3.13	Ubuntu 14.04	Linux ubuntu 3.13.0-24-generic 46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86 64 x86 64 x86 _64 GNU/Linux
2	5.4	Ubuntu 18.04	Linux version 5.4.0-105-generic (buildd@ubuntu) (gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)) 119 18.04.1-Ubuntu SMP Tue Mar 8 11:21:24 UTC 2022

The Linux VMs were used for typical bash command-line operations, including compiling and executing a socket-managing program named sockFull. This program generated, initialized, and bound sockets across a range of incrementing ports, closing every fourth socket to create identifiable "hidden" objects. In this study, fifteen sockets were generated across ports 1080 to 1094. Full implementation of sockFull is provided in Figures 3.3 and 3.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>

const int NUM_SOCKS = 15;
const int NUM_ADDR = 15;
const char* LOCAL_HOST_IP = "127.0.0.1";
const int PORT = 1080;
const int sockopt = 1;
int main(int argc, char *argv[])
{
 printf("[*]Starting... \n");
 int sockfds[NUM_SOCKS];
 sockaddr_in* addrs [NUM_ADDR];
 for(int i = 0; i < NUM_SOCKS; i++){
        sockfds[i]= socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        if(sockfds[i] <= 0){
                printf("[!]Socket %d failed to be created \n", i);
        }
}
 printf("[*]Socket init complete \n");
 for(int j = 0; j < NUM_ADDR; j++){
        sockaddr_in* ad = (sockaddr_in*) malloc(sizeof(sockaddr_in));
        ad->sin_addr.s_addr = INADDR_ANY;
        ad->sin_family = AF_INET;
        ad->sin_port = htons(PORT + j);
        addrs[j] = ad;
}
```

Figure 3.3. sockFull code

```
 printf("[*]sockaddr init complete \n");

for(int i = 0; i < NUM_SOCKS; i++){
        if(setsockopt(sockfds[i],SOL_SOCKET,SO_REUSEADDR,&sockopt,sizeof(sockopt)) < 0){
                printf("[!]setsockopt failed, socket %d \n",i);
        }
}

printf("[*]Sockopts set \n");
for(int i = 0; i < NUM_SOCKS; i++){
        sockaddr_in* addr = addrs[i % NUM_ADDR];
        if(bind(sockfds[i],(struct sockaddr *)addr, sizeof(*addr)) < 0){
                printf("[!]bind failed: sock %d, addr %d",i,i % NUM_ADDR);
        }
}
printf("[*]Socket binding complete\n");

for(int i = 0; i < NUM_SOCKS; i += 2){
        if(listen(sockfds[i],1) < 0){
                printf("[!] listen failed for sock %d",i);
        }
}
printf("[*]Socket listens set\n");

for(int i = 1; i < NUM_SOCKS-1; i +=4){
        close(sockfds[i]);
}

char line[100];
printf("[*]Pausing for VM Suspension\n");
fgets(line,sizeof(line),stdin);
printf("[*]Ending....\n");
return 0;
```

Figure 3.4. sockFull code

Chapter 4. Results

4.1 Updating the slab info Plugin

To support our approach, the slab info plugin required updates for SLUB compatibility, as it retrieves metadata from the kmem_cache. Running this updated plugin emulates the output of reading /proc/slabinfo on a Linux machine and includes APIs for querying specific cache data. Recreating SLUB output is straightforward—simply implement the approach shown in *Figure 4.1* to ensure accurate data.

```c
void get_slabinfo(struct kmem_cache *s, struct slabinfo *sinfo)
{
        unsigned long nr_slabs = 0;
        unsigned long nr_objs = 0;
        unsigned long nr_free = 0;
        int node;
        struct kmem_cache_node *n;

        for_each_kmem_cache_node(s, node, n) {
                nr_slabs += node_nr_slabs(n);
                nr_objs += node_nr_objs(n);
                nr_free += count_partial(n, count_free);
        }

        sinfo->active_objs = nr_objs - nr_free;
        sinfo->num_objs = nr_objs;
        sinfo->active_slabs = nr_slabs;
        sinfo->num_slabs = nr_slabs;
        sinfo->objects_per_slab = oo_objects(s->oo);
        sinfo->cache_order = oo_order(s->oo);
}
```

Figure 4.1. get slabinfo implementation as of Linux 5.17

The implementation can be validated by comparing /proc/slabinfo results from the target machine with the output of the updated plugin. Access to kmem_cache data from SLUB is vital for accurately defining slab bounds, enhancing our approach, and improving future plugins using SLUB analysis.

```
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <shar
edfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
UDPLITEv6              0      0   1088   30    8 : tunables    0    0    0 : slabdata      0      0      0
UDPv6                 30     30   1088   30    8 : tunables    0    0    0 : slabdata      1      1      0
tw_sock_TCPv6          0      0    256   64    4 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                 16     16   1984   16    8 : tunables    0    0    0 : slabdata      1      1      0
kcopyd_job             0      0   3312    9    8 : tunables    0    0    0 : slabdata      0      0      0
dm_uevent              0      0   2608   12    8 : tunables    0    0    0 : slabdata      0      0      0
```

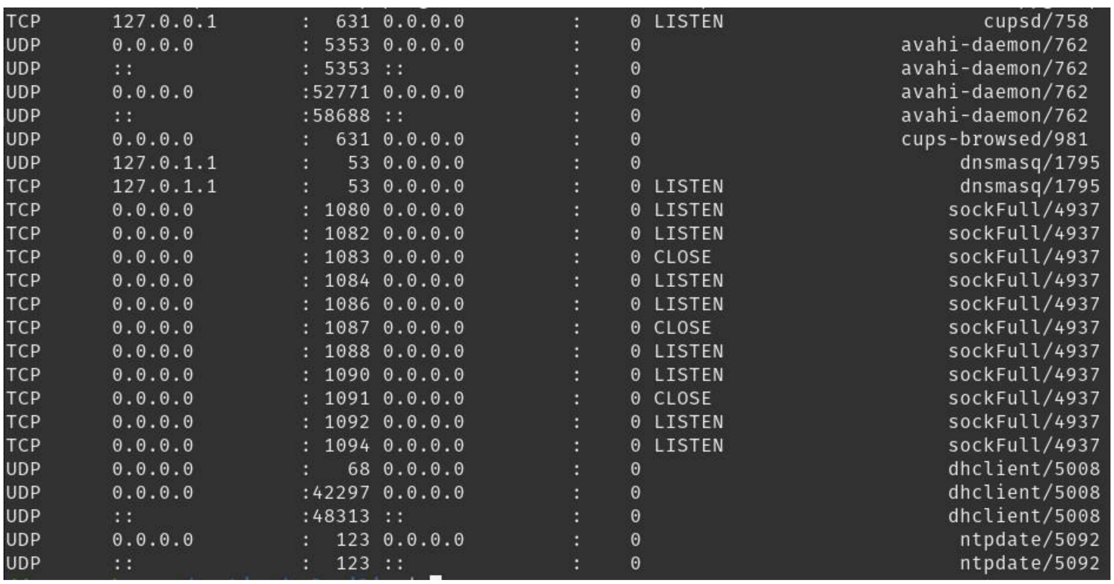
Figure 4.2. Result of running *cat /proc/slabinfo* on machine 1

```
<name>                         <active_objs> <num_objs> <objsize>  <objperslab> <pagesperslab>  <active_slabs> <num_slabs>
------------------------------ ------------- ---------- ---------- ------------ --------------- -------------- -----------
UDPLITEv6                      0             0          1088       30           8               0              0
UDPv6                          30            30         1088       30           8               1              1
tw_sock_TCPv6                  0             0          256        64           4               0              0
TCPv6                          16            16         1984       16           8               1              1
kcopyd_job                     0             0          3312       9            8               0              0
dm_uevent                      0             0          2608       12           8               0              0
```

Figure 4.3. Result of running the updated *slab info* plugin on sample 1 accurately target the bounds of a slab. Additionally any future plugins that incorporate SLUB analysis will benefit from the availability of this data.

4.2 Results of the New slab carve Plugin

Using a standard list-walking approach to view sockets within a sample, we executed the netstat -U Volatility command, where -U filters out UNIX sockets for clarity



```
TCP      127.0.0.1        :  631 0.0.0.0        :    0 LISTEN                  cupsd/758
UDP      0.0.0.0          : 5353 0.0.0.0        :    0               avahi-daemon/762
UDP      ::               : 5353 ::             :    0               avahi-daemon/762
UDP      0.0.0.0          :52771 0.0.0.0        :    0               avahi-daemon/762
UDP      ::               :58688 ::             :    0               avahi-daemon/762
UDP      0.0.0.0          :  631 0.0.0.0        :    0               cups-browsed/981
UDP      127.0.1.1        :   53 0.0.0.0        :    0                    dnsmasq/1795
TCP      127.0.1.1        :   53 0.0.0.0        :    0 LISTEN             dnsmasq/1795
TCP      0.0.0.0          : 1080 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1082 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1083 0.0.0.0        :    0 CLOSE             sockFull/4937
TCP      0.0.0.0          : 1084 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1086 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1087 0.0.0.0        :    0 CLOSE             sockFull/4937
TCP      0.0.0.0          : 1088 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1090 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1091 0.0.0.0        :    0 CLOSE             sockFull/4937
TCP      0.0.0.0          : 1092 0.0.0.0        :    0 LISTEN            sockFull/4937
TCP      0.0.0.0          : 1094 0.0.0.0        :    0 LISTEN            sockFull/4937
UDP      0.0.0.0          :   68 0.0.0.0        :    0                dhclient/5008
UDP      0.0.0.0          :42297 0.0.0.0        :    0                dhclient/5008
UDP      ::               :48313 ::             :    0                dhclient/5008
UDP      0.0.0.0          :  123 0.0.0.0        :    0                 ntpdate/5092
UDP      ::               :  123 ::             :    0                 ntpdate/5092
```

Figure 4.4. Result of running *netsat -U* on sample 1

In this sample, the sockFull artifact program closed and freed sockets at ports 1081, 1085, 1089, and 1093, resulting in their absence in the output. However, running the slab carve plugin successfully extracted "hidden" sockets at ports 1081, 1085, and 1089. While three of the four original sockets were recovered, each represents a previously inaccessible artifact.

```
<objname>           <offset>            <laddr:port>            <raddr:port>            <state>
------------------- ------------------- ----------------------- ----------------------- -------
INFO    : volatility.debug    : Carving for inet_sock in cache TCP
INFO    : volatility.debug    : Carving sockets using netstat
inet_sock           18446612133303229440 0.0.0.0:1085           0.0.0.0:0               CLOSE
inet_sock           18446612133303234816 0.0.0.0:1081           0.0.0.0:0               CLOSE
inet_sock           18446612133303238400 0.0.0.0:1089           0.0.0.0:0               CLOSE
inet_sock           18446612133303247360 0.0.0.0:0              0.0.0.0:0               CLOSE
inet_sock           18446612133303252736 0.0.0.0:0              0.0.0.0:0               CLOSE
inet_sock           18446612133303254528 -:0                    -:0
inet_sock           18446612133303256320 f0d0:4881:f...ff:ffff:256 ::10:eb0:5b7f:0:0
```

Figure 4.5. Result of running *slab carve* on sample 1 looking for sockets

The *slab carve* plugin can be utilized to look for **task struct** objects in sample 1.

```
task_struct         0xffff88003a8797f0 gdbus            957     0       0       2022-02-23 21:19:21 UTC+0000
task_struct         0xffff88003a878000 grep             5043    -1      -1      2022-02-23 21:50:16 UTC+0000
task_struct         0xffff88003a876810                  0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88003a875020                  0       -1      -1      2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88003a873830                  0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88003a886f60                  0       -1      -1      2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88002dfa3830                  0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88002d49a790                  0       -1      -1      2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88002ba317f0 gmain            3338    1000    1000    2022-02-23 21:20:28 UTC+0000
task_struct         0xffff88002ba32fe0 dconf worker     2096    1000    1000    2022-02-23 21:19:27 UTC+0000
task_struct         0xffff88002ba347d0 upstart          5089    -1      -1      2022-02-23 21:50:16 UTC+0000
task_struct         0xffff88002ba35fc0 mkdir            5091    -1      -1      2022-02-23 21:50:16 UTC+0000
```

Figure 4.6. Result of running *slab carve* on sample 1 looking for processes.

```
task_struct         0xffff88003c5e17f0 swapper/1        0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88003c5e2fe0 swapper/2        0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88003c5e47d0 swapper/3        0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct         0xffff88003c5e5fc0 swapper/4        0       0       0       2022-02-23 21:19:19 UTC+0000
```

Figure 4.7. Linux Swapper processes were also found on sample 1

Although we did not explicitly create processes for forensic hunting, this approach identified processes absent from pslist, such as historical shell commands like grep and mkdir. "Swapper" processes were also recovered, despite their intentional exclusion from the Volatility process list.

Following the same methodology, we verified the updated slab info plugin's effectiveness on the second sample.

```
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
ext4_groupinfo_4k    168    168    144   56    2 : tunables    0    0    0 : slabdata      3      3      0
fsverity_info          0      0    248   66    4 : tunables    0    0    0 : slabdata      0      0      0
ip6-frags              0      0    184   44    2 : tunables    0    0    0 : slabdata      0      0      0
PINGv6                52     52   1216   26    8 : tunables    0    0    0 : slabdata      2      2      0
RAWv6                397    520   1216   26    8 : tunables    0    0    0 : slabdata     20     20      0
UDPv6                 48     48   1344   24    8 : tunables    0    0    0 : slabdata      2      2      0
tw_sock_TCPv6          0      0    248   66    4 : tunables    0    0    0 : slabdata      0      0      0
request_sock_TCPv6       0      0    304   53    4 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                 26     26   2432   13    8 : tunables    0    0    0 : slabdata      2      2      0
```

Figure 4.8. Result of running *cat /proc/slabinfo* on machine 2

```
<name>                         <active_objs> <num_objs> <objsize>  <objperslab> <pagesperslab>  <active_slabs> <num_slabs>
------------------------------ ------------- ---------- ---------- ------------ --------------- -------------- -----------
INFO    : volatility.debug    : SLUB detected
ext4_groupinfo_4k              168           168        144        56           2               3              3
fsverity_info                  0             0          248        66           4               0              0
ip6-frags                      0             0          184        44           2               0              0
PINGv6                         52            52         1216       26           8               2              2
RAWv6                          397           520        1216       26           8               20             20
UDPv6                          48            48         1344       24           8               2              2
tw_sock_TCPv6                  0             0          248        66           4               0              0
request_sock_TCPv6             0             0          304        53           4               0              0
TCPv6                          26            26         2432       13           8               2              2
```

Figure 4.9. Result of running the updated *slab info* plugin on sample 2

Then we gathered all of the sockets gathered from list enumeration through *netstat*, shown in figure 4.10

```
UDP      127.0.0.53      :   53 0.0.0.0         :    0                    systemd-resolve/517
TCP      127.0.0.53      :   53 0.0.0.0         :    0 LISTEN             systemd-resolve/517
UDP      0.0.0.0         : 5353 0.0.0.0         :    0                       avahi-daemon/635
UDP      ::              : 5353 ::              :    0                       avahi-daemon/635
UDP      0.0.0.0         :36280 0.0.0.0         :    0                       avahi-daemon/635
UDP      ::              :47946 ::              :    0                       avahi-daemon/635
TCP      ::1             :  631 ::              :    0 LISTEN                       cupsd/653
TCP      127.0.0.1       :  631 0.0.0.0         :    0 LISTEN                       cupsd/653
UDP      0.0.0.0         :  631 0.0.0.0         :    0                       cups-browsed/696
TCP      0.0.0.0         : 1080 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1082 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1083 0.0.0.0         :    0 CLOSE                   fullSock/10530
TCP      0.0.0.0         : 1084 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1086 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1087 0.0.0.0         :    0 CLOSE                   fullSock/10530
TCP      0.0.0.0         : 1088 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1090 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1091 0.0.0.0         :    0 CLOSE                   fullSock/10530
TCP      0.0.0.0         : 1092 0.0.0.0         :    0 LISTEN                  fullSock/10530
TCP      0.0.0.0         : 1094 0.0.0.0         :    0 LISTEN                  fullSock/10530
```

Figure 4.10. Result of running *netsat -U* on sample 2

We were able to recover all four of the ”missing” sockets for sample 2, as shown in figures 4.11 and 4.12.

```
inet_sock           0xffff9fd134c0c600 0.0.0.0:1085            0.0.0.0:0                 CLOSE
inet_sock           0xffff9fd134c0e040 0.0.0.0:1081            0.0.0.0:0                 CLOSE
inet_sock           0xffff9fd134c0fa80 -:0                     -:0
inet_sock           0xffff9fd134c10340 -:0                     -:0
inet_sock           0xffff9fd134c10c00 -:0                     -:0
inet_sock           0xffff9fd134c114c0 -:0                     -:1024
inet_sock           0xffff9fd134c11d80 -:21248                 -:0
inet_sock           0xffff9fd134c12640 -:10755                 -:25602
inet_sock           0xffff9fd134c12f00 -:0                     -:52740
inet_sock           0xffff9fd134c137c0 -:0                     -:0
inet_sock           0xffff9fd134c0abc0 0.0.0.0:1089            0.0.0.0:0                 CLOSE
```

Figure 4.11. Result of running *slab carve* on sample 2 looking for sockets

```
inet_sock           0xffff9fd0c425a300 -:0                     -:0
inet_sock           0xffff9fd0c425abc0 0.0.0.0:1093            0.0.0.0:0                 CLOSE
inet_sock           0xffff9fd0c425b480 -:0                     -:0
```

Figure 4.12. Result of running *slab carve* on sample 2 looking for sockets

Carving for processes on sample 2, were again able to find both previous commands, as well as swapper processes, shown in figures 4.13 and 4.14.

```
task_struct         0xffff9fd100508000 awk                 10563   -1      -1      2022-03-22 15:20:34 UTC+0000
task_struct         0xffff9fd100506240 ?lP                 98...44 -1      -1      0
task_struct         0xffff9fd100504480 ?H7????            0       -1      -1      2022-03-22 15:04:21 UTC+0000
task_struct         0xffff9fd1005026c0                     0       -1      -1      0
task_struct         0xffff9fd100500900                     0       -1      -1      2022-03-22 15:04:21 UTC+0000
task_struct         0xffff9fd10056bb80 vmhgfs-fuse         1996    0       0       2022-03-22 15:08:40 UTC+0000
task_struct         0xffff9fd10056f700                     0       -1      -1      0
task_struct         0xffff9fd1005714c0                     99...64 -1      -1      0
task_struct         0xffff9fd100573280                     111     -1      -1      2022-03-24 05:52:06 UTC+0000
task_struct         0xffff9fd100568000 gmain               1633    1000    1000    2022-03-22 15:04:57 UTC+0000
```

Figure 4.13. Result of running *slab carve* on sample 2 looking for processes.

```
task_struct         0xffff9fd13ac61dc0 swapper/3           0       0       0       2022-03-22 15:04:21 UTC+0000
task_struct         0xffff9fd13ac63b80 swapper/1           0       0       0       2022-03-22 15:04:21 UTC+0000
task_struct         0xffff9fd13ac65940 swapper/2           0       0       0       2022-03-22 15:04:21 UTC+0000
task_struct         0xffff9fd13ac67700                     0       -1      -1      2022-03-22 15:04:21 UTC+0000
```

Figure 4.14. Linux Swapper processes found on sample 2

4.3 Discussion

Our approach successfully recovered three of the four originally created sockets, demonstrating its capacity to retrieve forensic artifacts overlooked by list enumeration. The inability to find certain created objects may be due to various factors: the area where an object resided may have been overwritten by another upon freeing; no list-enumerated objects were allocated on the same slab, rendering our approach ineffective; or a sufficient number of freed objects triggered the slab's deallocation.

The effectiveness of our approach extends to process information recovery, comparable to the former pslist cache plugin. Additionally, recovering "swapper" processes confirms the carving method’s ability to uncover active yet non-enumerated objects. For Linux systems, the discovery of old processes aids investigators in building a timeline of system activity.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

Our approach enhances forensic capabilities for SLUB systems, allowing investigators to recover artifacts that were previously difficult or impossible to extract. By implementing this approach into the *slab carve* plugin, forensic analysis on SLUB systems in Volatility is now improved. Furthermore, with updates to the *slab info* plugin, we have enabled the extraction of kmem cache metadata, paving the way for deeper analysis of memory samples from SLUB systems.

5.2 Future Work

This research offers a foundational carving method for extracting objects from SLUB slabs. Future efforts could include the resolution of slab pages so that partial and full lists can enhance carving by targeting a greater variety of objects. Improving the plugin's capabilities with an object-specific targeting feature and cache identification would expand forensic versatility across Linux systems. Additionally, filtering options could be added to help analysts focus on relevant objects without additional tools. Although this plugin was developed using Volatility version 2.6, we plan to adapt it to Volatility 3 once it exits beta.

As Linux begins implementing folios as a new page representation, it will likely have significant implications for memory forensics across Linux systems. Further study on folios and their forensic applications will be essential as they become more prevalent.

References

1. Block, F., & Dewald, A. (2017). Linux memory forensics: Dissecting the user space process heap. *Digital Investigation, 22*, S66–S75. [Online]. Available: ScienceDirect
2. Block, F., & Dewald, A. (2019). Windows memory forensics: Detecting (un)intentionally hidden injected code by examining page table entries. *Digital Investigation, 29*, S3–S12. [Online]. Available: ScienceDirect
3. Case, A., Das, K., Park, S.-J., Ramanujam, J. R., & G.G.R. III. (2017). Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks. *Digital Investigation, 22*, S86–S93. [Online]. Available: ScienceDirect
4. Case, A., Marziale, L., Neckar, C., & G.G.R. III. (2010). Treasure and tragedy in kmem cache mining for live forensics investigation. *Digital Investigation, 7*, S41–S47. [Online]. Available: ScienceDirect
5. Fire Eye. (2019). Finding evil in Windows 10 compressed memory. [Online]. Available: FireEye
6. Kaspersky. (2015). The mystery of Duqu 2.0: A sophisticated cyberespionage actor returns. [Online]. Available: SecureList
7. Lewis, N., Case, A., Ali-Gombe, A., & G.G.R. III. (2018). Memory forensics and the Windows Subsystem for Linux. *Digital Investigation, 26*, S3–S11. [Online]. Available: ScienceDirect
8. Linux. (2022). Kconfig. [Online]. Available: Bootlin
9. Linux. (2022). slob.c. [Online]. Available: GitHub