IMPROVING KERNEL ARTIFACT EXTRACTION IN LINUX MEMORY SAMPLES USING THE SLUB ALLOCATOR

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Computer Science and Engineering

by
Daniel Donze
B.S., Louisiana State University, 2016
May 2022

Acknowledgments

I would like to thank Dr. Golden G. Richard III for all his help during my time at LSU. His expertise and knowledge of memory forensics as well as his advice was very helpful throughout my research. I would like to thank Andrew Case for providing his expansive knowledge. I would like to thank Dr. Feng Chen and Dr. Doris Carver for being members of my thesis committee. I would like to thank my friends in the applied cybersecurity lab for their support throughout my degree and in my research.

Table of Contents

List of Tables

List of Figures

Abstract

Memory forensics allows an investigator to analyze the volatile memory (RAM) of a computer, providing a view into the system state of the machine as it was running. Examples of items found in memory samples that are of interest to investigators are kernel data structures which can represent processes, files, and sockets. The SLUB allocator is the default small-request memory allocator for modern Linux systems. SLUB allocates "slabs", which are contiguous sections of pre-allocated memory that are used to efficiently service allocation requests. The predecessor to SLUB, the SLAB allocator, tracked every slab it allocated, allowing extraction of allocated slabs relatively easily from a memory forensics perspective. One of the changes introduced by SLUB, is that SLUB may not always track slabs once they become full. This has posed an issue with memory forensics, as it removes the tracking mechanisms previously leveraged to extract slabs. We researched and developed a technique that uses a mix of carving and linked list enumeration to locate slabs allocated by SLUB. This technique finds objects that are allocated by SLUB and carves in adjacent memory spaces to find similar objects. We implemented our technique in a Volatility plugin *slab_carve* and demonstrate its ability to extract artifacts from memory. The addition of the developed plugin to the Volatility framework will allow investigators to recover a wealth of information that has previously been missing since the Linux kernel's switch from the SLAB to SLUB allocator. This newly available information can aid recovery of further system state, reconstruct activities of attackers that abuse a system, and recover traces of malware.

Chapter 1. Introduction

1.1. Memory Forensics

Digital forensics encompasses techniques and tools which provide insight into the activities performed on computers. Memory forensics is a subset of digital forensics which focuses on analyzing the contents of the volatile memory of a machine, colloquially referred to as RAM. Typically analysis is done on a memory sample: a copy taken of a machine's volatile memory. Acquisition of memory samples can be performed through use of open-source or commercial software for systems running directly on hardware. Systems that are run in a virtualized environment, such as virtual machines, may be suspended by the host machine to obtain a sample directly.

Analysis of memory gives forensic insights into the state of the machine as it is running, rather than the data that is stored on a disk. Investigators can extract and analyze artifacts such as processes, sockets, passwords, file handles, and system hooks, which may not be written to disk. Memory forensics allows investigators to make a timeline of actions taken on the machine, which users performed those actions, and even determine if malware performed those actions. Memory forensics has been able to find some malware that exists entirely in memory and never writes to the disk[6].

Memory forensics is performed primarily through specialized tools, such as Volatility. Volatility is an open source, plugin based memory forensics framework[16]. Volatility has a range of plugins to analyze memory samples acquired from Windows, MacOS, and Linux systems. Volatility provides valuable features to analysts such as translation between virtual and physical address spaces, data structure overlays for kernel objects, and an interactive shell to inspect the memory sample. Volatility plugins may use results of other plugins to perform their own calculations, allowing a pipeline of data analysis to occur.

1.2. The Linux Operating System

The Linux operating system is used in approximately 25% of software developer workstations[11], 80% of web servers[18], nearly 100% of supercomputers[19], and acts as the booting operating system for Android phones. The Linux operating system is open-source, allowing anyone to view and contribute to the code-base. Linux's open-source status allows for simplified and accurate analysis of the kernel behavior since the source code is available. One responsibility of the operating system is to manage the memory for users and itself. Requests for memory allocations can vary greatly in size, and the operating system must efficiently handle these requests. Memory is divided up into subsections called pages, which can be individually referenced. Most modern hardware have page sizes of 4096-bytes, hereby referred to simply as pages. Data is written and read into pages through a reference to their page number, as well as an offset into the page. Linux has separate mechanisms to allocate and track memory depending on the size of the request. Larger requests are handled by the Zone allocator, and smaller requests are handled by a SL*B Allocator, which can be SLAB, SLUB, SLOB, or some other implementation. The choice of allocator is determined by a kernel configuration option at compile time.

1.3. Motivation

Prior to the adoption of SLUB, Volatility could extract process information through the slabs allocated by SLAB through its plugin, *pslist_cache*. Extraction of process information is vital for a forensic investigation as it gives leads to actions taken on a system, both by users and malware. Due to changes in functionality introduced by SLUB, the plugin was unable to extract any information from systems using SLUB. This left Volatility with no automated tools to analyze SLUB systems. We set out to analyze SLUB and create a tool which could extract forensic artifacts much like *pslist_cache*.

1.4. Research Importance

While previous plugins for slab analysis focused on extracting process information, slabs contain all of the smaller kernel data structures, such as sockets and file-system caches, making them a trove of forensic artifacts. Recovering artifacts through slabs is vital since there is the potential of finding objects that have been freed but not overridden, such as processes that terminated before the memory sample was taken, as well as finding objects that have been removed from kernel tracking lists, a behavior malware often exhibits.

1.5. Notation

Items in bold refer to the Linux kernel structure of that name; this is important to distinguish between the concept of a page in operating systems versus a **page**, which refers to the data structure representation. Items in italics refer to a Volatility plugin, or a command to a shell. All Volatility plugin names refer to the Linux version in instances of overlap.

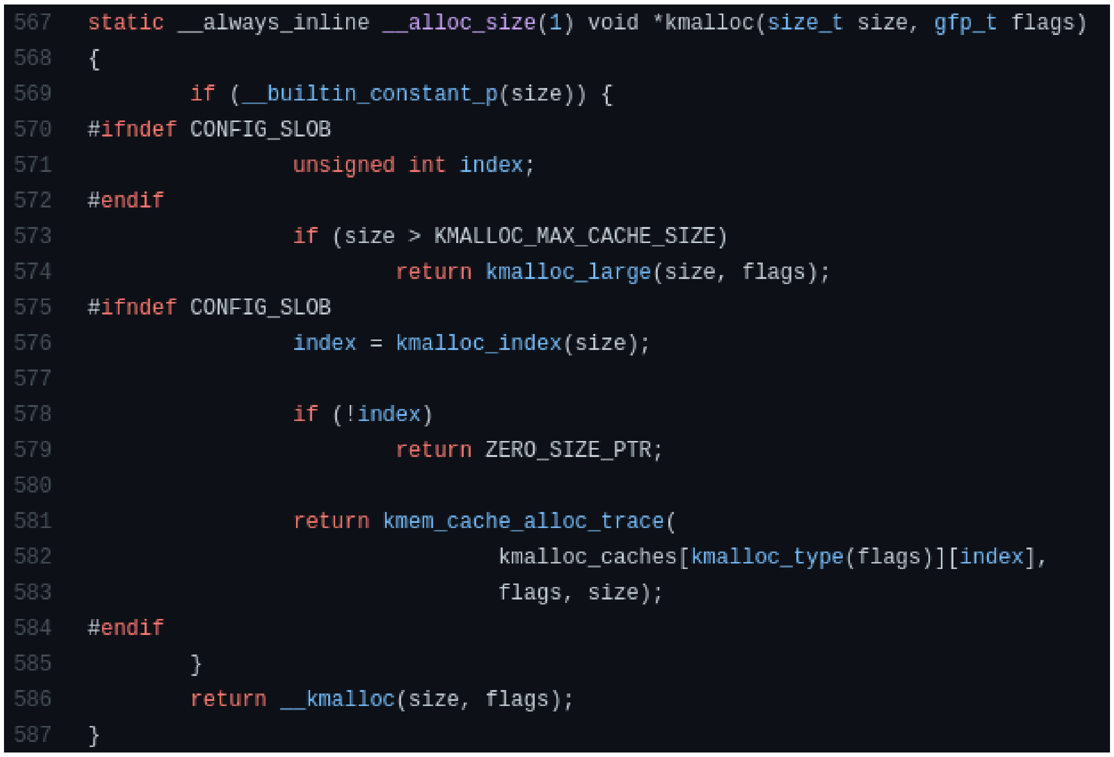
1.6. Outline

Chapter 2 provides a detailed technical background on SLUB, Volatility, as well as related work in memory forensics. Chapter 3 discusses our methodology and experimental setup . Chapter 4 presents the results we obtained and discusses their implications. Chapter 5 concludes on what has been done and discusses future work for Linux memory forensics.

Chapter 2. Background

2.1. Linux Memory Allocation

Linux memory allocation begins when some portion of the kernel calls kmalloc. The size of the request is checked to determine whether the request is sent to the Zone allocator, or to the SL*B allocator. This maximum size is determined by which SL*B allocator is present. SLUB supports up to 2-page large allocations, while SLAB and SLOB support 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**

SLAB is the original small memory allocator, and was the default memory allocator in Linux until kernel version 2.6.23 [8]. The SLAB allocator creates caches that contain slabs (represented by a struct **page**), which are divided to hold objects which are under a page in size. The SLAB allocator strives to group objects of the same type in the same cache. The SLAB allocator tracks all the slabs it manages in its caches in lists of fully allocated slabs, partially allocated slabs, and free (empty) slabs. Additionally each slab **page** contains a pointer to the first object in the slab represented.

- **SLOB**

  SLOB is a simplified implementation of SLAB that does not consume as much memory for management structures, designed to be used in low memory environments such as embedded devices. The implementation of SLOB is a Kernighan and Ritchie style heap with the ability to return memory aligned objects [9].

- **SLUB**

  The SLUB allocator replaced and upgraded the SLAB allocator through several optimizations, including changing its grouping policy from caches of a specific object to caches of similar size. SLUB organizes objects into caches, represented by the struct **kmem_cache**. Each default cache is responsible for handling requests of a certain size and allocation type. Slabs are allocated as a series of contiguous pages, which contain allocated objects and a freelist pointer, which points to where the next allocation will take place. The number of pages a slab takes is determined both by the kernel configuration and the size of the cache. SLUB exports an API to create additional caches separate from the generic caches. Both users and other kernel components have access to this API.

  The SLUB allocator does not track fully allocated slabs unless the CONFIG_SLUB_DEBUG setting is enabled during configuration. While this is a default setting, this means that memory samples may lack the fully allocated slab list, and its existence cannot be assumed. Additionally the slab **page** no longer contains a reference to the first object in the slab. This is forensically significant as there is not a direct link from a slab **page** to the actual page containing the objects, requiring the resolution of a struct **page** to the represented physical page in memory.

2.1.2. Linux Memory Allocator Internals

We will briefly touch upon the SLUB internals and how they're used. A **kmem_cache** has size, object_size,flags, object order (oo), and pointers to **kmem_cache_cpu** and **kmem_cache_node**. Size determines how objects are spaced within a slab, objects_size is used when metadata is stored along with objects in a slab, and flags represent the type of slab the **kmem_cache** handles. The oo represents a ratio between the page size and the object size, used to determine how many pages are used per slab to optimally fit an object.

The **kmem_cache_cpu** is a per cpu structure that contains a reference to the current slab, freelist pointer, and a transaction id (tid). These items are checked upon performing any allocation or de-allocation oepration to ensure that a process was not preempted during the update so that the values are updated correctly. The **kmem_cache_node** contains pointers to the partial and full lists for a given **kmem_cache**.

2.2. Volatility and Memory Forensics

The Volatility framework is a open-source memory analysis tool that can analyze the data found in the volatile memory (RAM) of a system [15]. Volatility performs analysis on a memory sample by matching it to a corresponding Volatility profile, which contains information needed to map out an operating system's memory [16]. Volatility is plugin based, which allows both standalone plugins to analyze specific parts of memory, as well as plugins to utilize the data from each other for a more complex analysis. Volatility handles physical to virtual address translation, kernel object interpretation, as well as providing a shell to manually analyze the memory.

2.2.1. Linux Volatility Profile Creation

Volatility needs a profile for the exact kernel version of a memory sample's operating system in order to obtain accurate analysis. The process of obtaining a profile varies between each major operating system (Windows, MacOs, Linux, Android, etc.); we will detail only the process of obtaining a Linux profile. Profile creation for Linux systems works by compiling a large, "dummy" kernel module that contains references or replicas of all needed kernel structures, and is compiled with debug symbols enabled. Afterwards, the dwarfdump utility is used to extract all of the type information of the kernel module, which is then combined with the System.map file that contains all of the symbol information (names and addresses). Once combined into a zip file, this profile can be then be copied into the Volatility installation directory and used to analyze samples.

2.2.2. The *pslist* Plugin

The *pslist* plugin gathers information on all of the **task_struct** instances by enumerating the kernel list tracking them. A **task_struct** represents a process in Linux, and contains information such as the process ID, the name of the process, when the process started, and the state of the process.

2.2.3. The *pslist_cache* Plugin

The *pslist_cache* plugin extracted **task_struct** instances by finding the "task_struct" **kmem_cache** in SLAB and carving out **task_struct** objects. This could find processes that are not present in the kernel task list, such as terminated processes or processes that have be de-linked from the kernel list. A process that appears to be active but only shows up through carving approaches is forensically interesting as this indicates there were attempts to hide said process.

2.2.4. The *netstat* Plugin

The *netstat* plugin takes a list of **task_struct** instances and iterates through the network information to pull out socket connections they may have. This network information includes remote address and port, local port, connection state, as well as what process has these connections. Networking information like this can be forensically interesting when there are connections to unknown remote addresses, or when applications that do not normally perform networking have sockets associated with them.

2.3. Related Works

2.3.1. Resilient Memory Acquisition Analysis

Most acquisition applications utilize the operating system in some way to access and read memory for a sample. Targeted corruption of operating system structures can thwart both acquisition and analysis of a memory sample. By corrupting specific kernel structures, malware can stop a successful acquisition of memory, but may also destabilize the operating system, leading to a crash. With this risk, malware would only want to corrupt the kernel structures if it detects behavior indicating an acquisition is in progress. Often malware can fingerprint this behavior as many acquisition tools are either popular or open source, thus their behavior is well known. Stüttgen and Cohen[12] developed a method of acquiring memory through hardware instead of the operating system, avoiding detection by malware.

2.3.2. Acquiring Firmware Through Memory Acquisition

Rootkits are sophisticated pieces of malware which attempt to subvert the operating system to gain control over a system. Bootkits in recent years have succeeded rootkits by attempting to subvert the system's firmware rather than the operating system. This has created the need for firmware acquisition in an incident response scenario. Stüttgen et el. [13] developed a method to safely acquire memory regions firmware resides in during a memory acquisition. In addition to methods for firmware acquisition, the authors developed plugins for Volatility and Rekall to analyze the Advanced Configuration and Power Interface (ACPI) environment in the resulting memory samples.

2.3.3. Fuzzing Tools for Sample Corruption Resilience

Memory smearing in memory forensics refers to corruption of memory samples through non-atomic acquisition. This is often the case when a live machine is getting sampled under heavy activity, as system structures such as page table may change after a tool has acquired them. Work by Case et al. [3] created a framework to mutate memory samples to test how different tools handle sample corruption. The framework imitated common corruption patterns that result from non-atomic acquisition of active systems.

2.3.4. Analyzing Compressed Memory

The amount of memory a system has is determined by the hardware installed. Despite a physical limit, often operating systems have to deal with situations where it needs more memory than is physically present. Traditionally, operating systems stored excess pages on the hard disk in a swap file, making way for more pages to be allocated and used. In more recent years, operating systems instead leveraged compression methods in lieu of using the swap file, as compressing memory is much faster than reading and writing from the hard disk. Pages in their compressed form, however, are useless for analysis as compression mangles the data into a smaller format. Work by Richard and Case [10] to reverse the page compression process performed by Linux and MacOs systems. Similar work was performed later by Fire Eye to retrieve compressed pages from Windows systems [5]. This removes the possibility of critical forensic information being lost inside of a compressed page.

2.3.5. Forensic Analysis of kmem_cache Inactive Objects

Historically, the **kmem_cache** has been target for forensic analysis as it provides access to kernel object instances. The work of Case et al. [4] aimed to extract "inactive" objects from slabs allocated in a **kmem_cache**. At the time of their work, SLAB was the predominant allocator in use, with SLUB recently added to the Linux kernel. They demonstrated extraction of inactive/free objects is possible, however the de-allocation process removed some forensically useful members from these objects.

2.3.6. Dissecting User-space heaps

User-space heaps store dynamically allocated program data, which can include forensically interesting data such as passwords or session data. Block and Dewald[1] worked on fingerprinting glibc's implementation of the heap, to allow reliable and efficient extraction of the heap data. This is accomplished through finding the main heap created by the library, and following the pointers found to locate the data of interest.

2.3.7. Analysis of the Windows Subsystem for Linux

In the "Anniversary" update for Windows 10, the Windows Subsystem for Linux(WSL) was introduced, allowing Windows users to run an emulated Linux environment. WSL added native support for Linux executables: ELF files. One of the forensic analysis challenges for WSL is that all the processes ran in WSL are Windows "pico" processes, which lack many of the informational data structures of regular processes. Work by Lewis et al.[7] reverse engineered parts of WSL in order to create versions of existing tools that can analyze WSL processes.

2.3.8. Windows Quick Pool Tag Scanning

Sylve et al.[14] developed a method to more efficiently scan Windows memory for pool tags. Existing scanning techniques at the time were to scan all of Windows' memory to find pool tags to locate kernel objects. A solution was proposed to utilize a bitmap Windows stored to find only the mapped pages that would have pool tags, and only scan those mapped pages. This method reduced scan time by at least two orders of magnitude.

2.3.9. Detection of Windows Malware

Several memory forensics approaches look at the security settings of pages to hunt for malware. Traditionally suspicious pages will have the bits for EXECUTE_READWRITE, as these pages can be read, written to, and executed. This combination of settings is used by malware to dynamically inject code into running processes as a method of hiding. Work by Block and Dewald[2] analyzed different page states for windows memory samples and tested the ability for forensic tools to identify potentially malicious pages.

Once malware has infected a system, it attempts to achieve persistence, where it can continue to run on a system even in the event of a reboot. For Windows, Uroz and Rodríguez [17] documented numerous Windows functionalities malware leverages to gain persistence. They also developed a plugin to analyze the Windows registry for registry entries malware uses for persistence.

Chapter 3. Methodology

3.1. Extracting Objects From Slabs

Given the location of the start of a given **kmem_cache** slab, objects will start at the beginning of the slab, and every **kmem_cache.size** bytes after. A stepped parsing approach will extract all potentially valid objects. While there is no information to indicate what type of object is extracted, it is generally rare for different kernel objects to have overlapping byte data that corresponds to valid values for a target object. Thus ”junk” objects can easily be identified from invalid values, and do not hinder analysis.

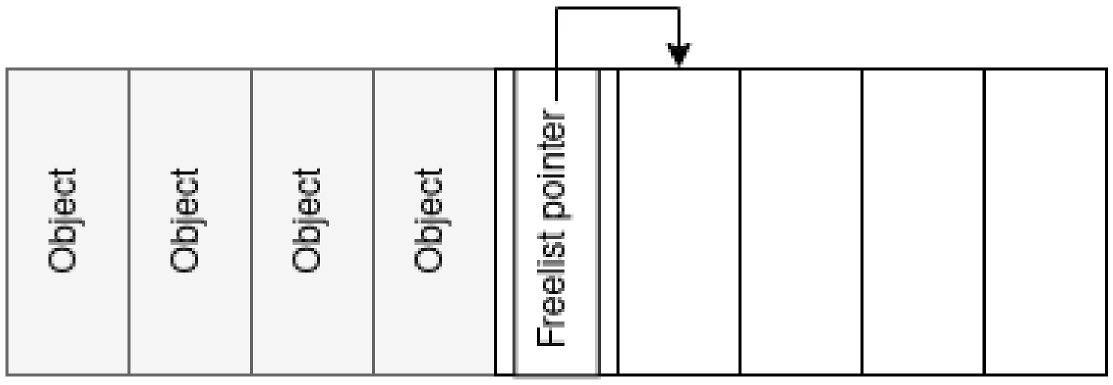


Figure 3.1. Diagram of slab layout

3.2. Getting the Needle into the Haystack

When we do not have access to slabs through their lists, we can access their objects through kernel lists, such as the process list. These object locations can be provided to us through other Volatiltiy plugins, such as *pslist* and *netstat*. From gathering the locations of these objects, we carve the slab starting at the objects rather than the beginning of the slab. While we do not know where an object resides inside of the slab, we can determine the farthest distance in memory the slab could extend. We can perform a stepped carving in both directions to guarantee that we will carve all the objects contained in the same slab as the object. The bounds of a slab can be calculated by multiplying the number of pages per slab, provided by the **kmem_cache**, and the page size, provided by the architecture.

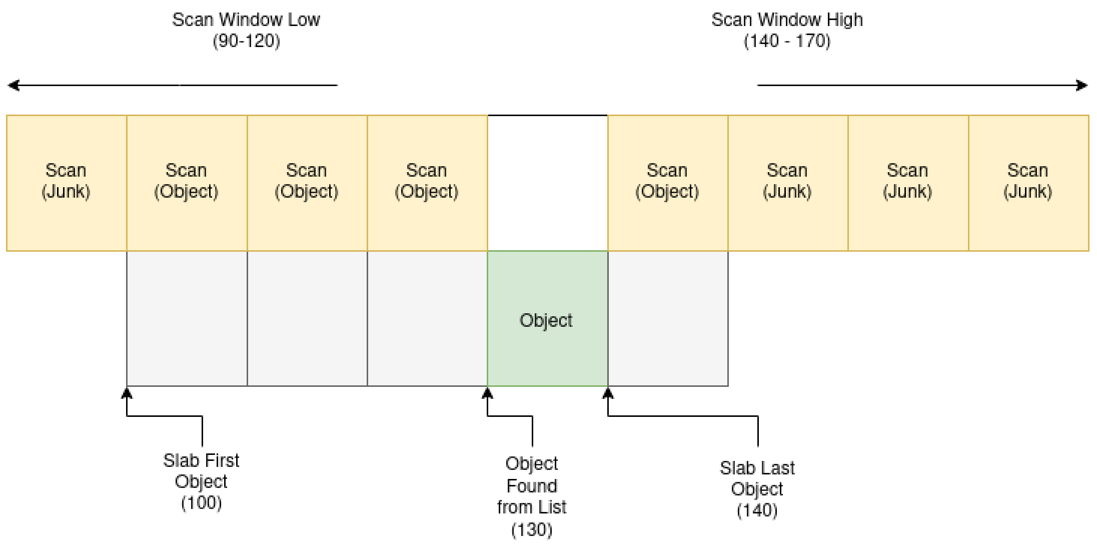


Figure 3.2. Diagram of carving approach

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.

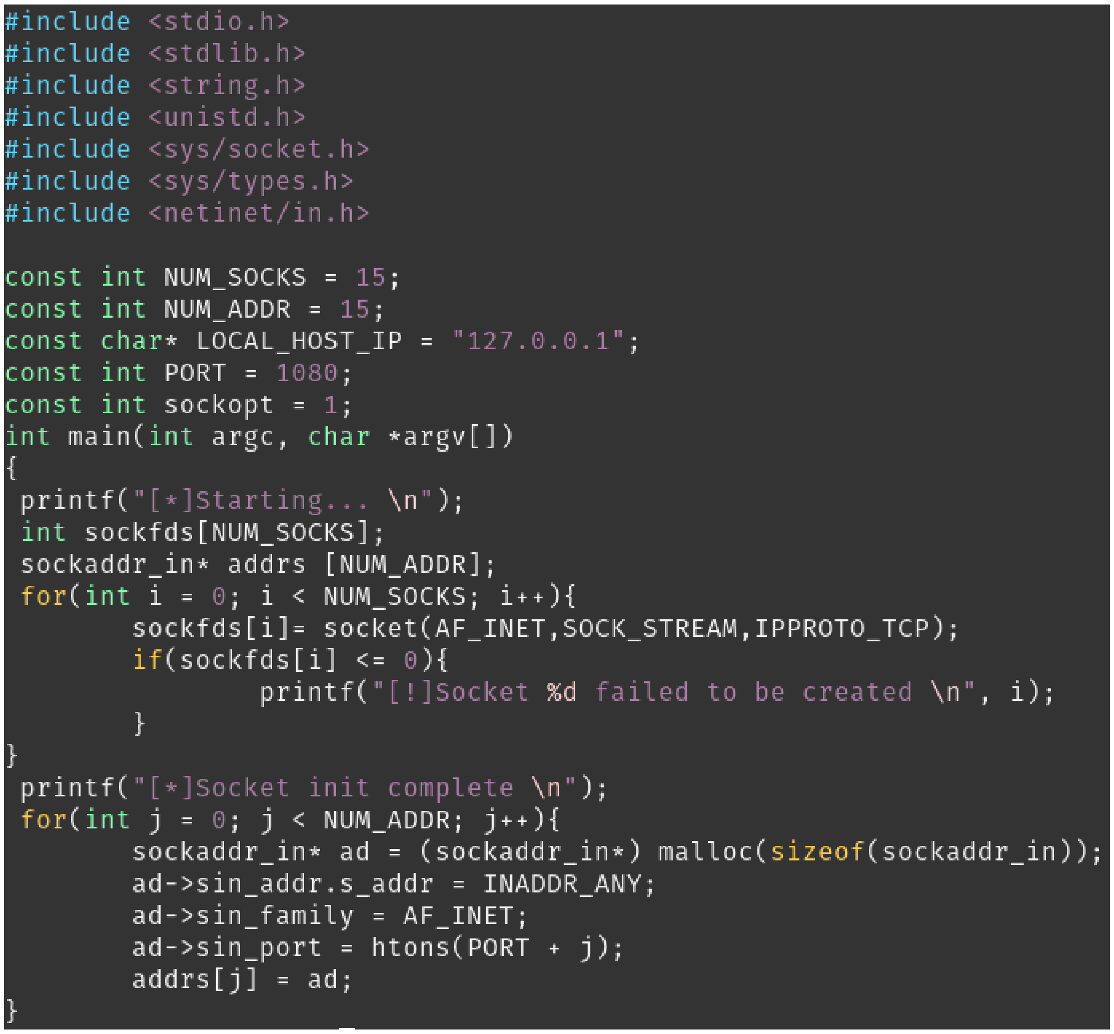
3.3. Experimental Setup

Linux targets were set up as virtual machines with Vmware. Memory samples were acquired by taking a snapshot of the virtual machine. The Linux targets used are shown in table 3.1. Analysis was performed on a Ubuntu 21.04 machine, running Volatility from the command line.

Table 3.1. Linux versions used for testing.

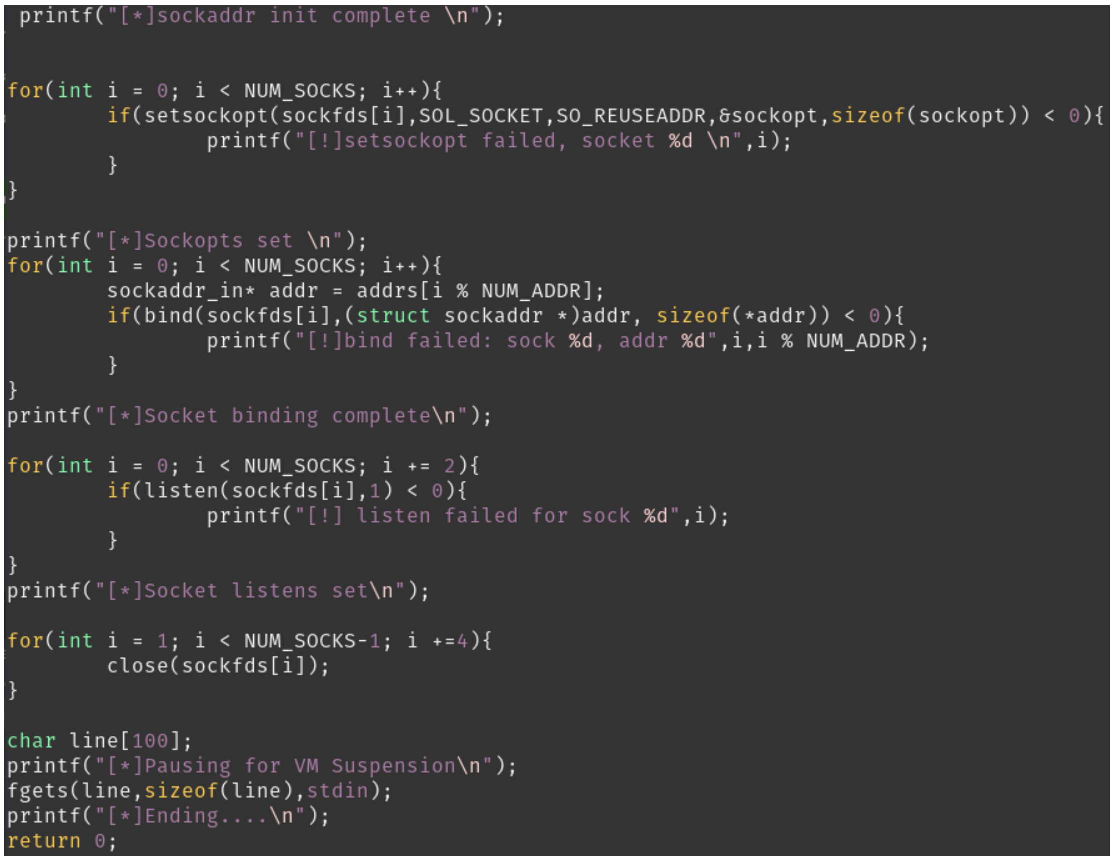
Sample	**Linux Kernel**	**Distribution**	**uname -a**
1	3.13	Ubuntu 14.04	Linux ubuntu 3.13.0-24-generic 46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
2	5.4	Ubuntu 18.04	Linux version 5.4.0-105-generic (buildd@ubuntu) (gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)) 119 18.04.1-Ubuntu SMP Tue Mar 8 11:21:24 UTC 2022

The Linux targets had some regular bash command line usage, including compiling and running a program to create some socket artifacts. The program was named ”sockFull” on the target machines. The artifact program created, initialized, and bound sockets on a number of incrementing ports, then closed every fourth socket. This was done to create identifiable ”hidden” objects to extract to demonstrate artifact recovery. For our examples, fifteen sockets were created, ranging on ports 1080 through 1094. The full implementation of ”sockFull” is shown 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



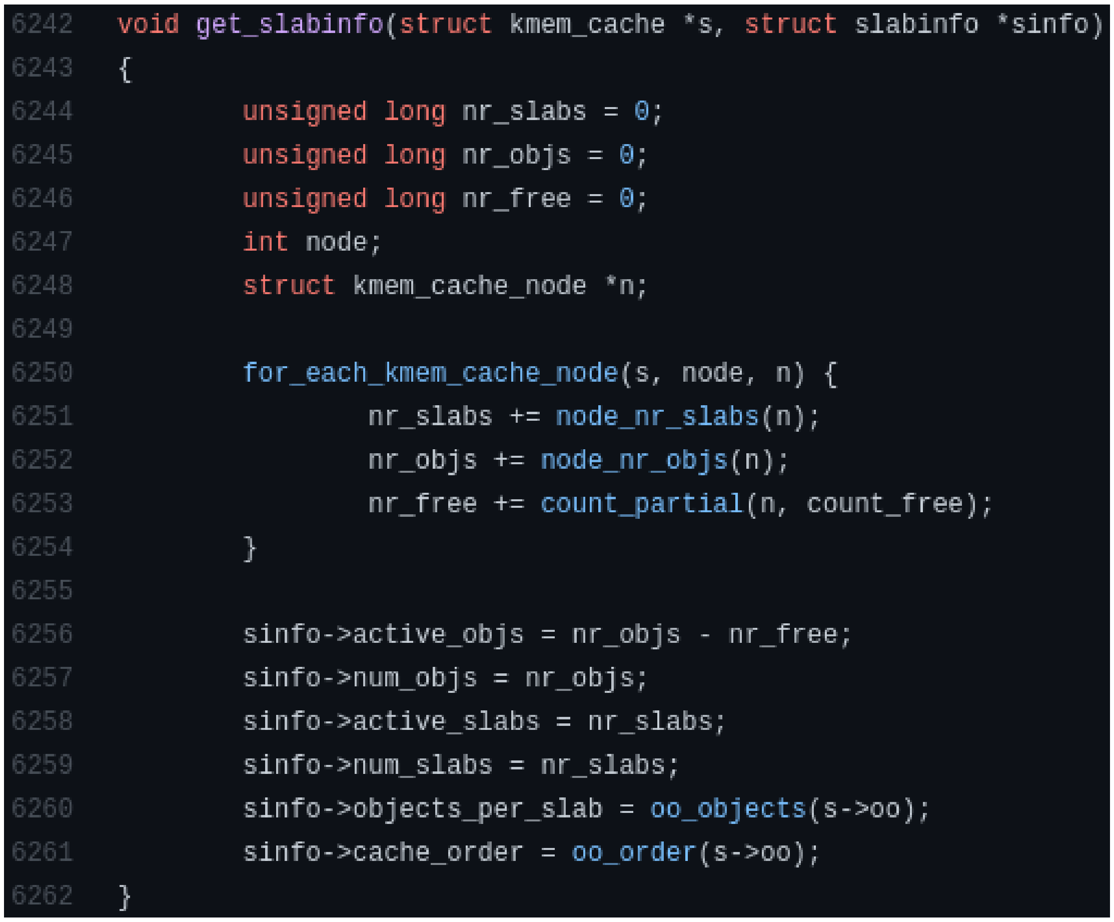
```
 printf("[*]sockaddr init complete \n");


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

Prior to using our approach, the *slab_info* plugin needed an update for SLUB support, as it provides the metadata for a **kmem_cache**. This plugin when ran directly emulates the output when reading /proc/slabinfo on a Linux machine, and provides API's for querying specific cache's data. Recreating the output for SLUB is fairly straightforward, as all we need is to re-create the implementation in fig 4.1 to create accurate data.

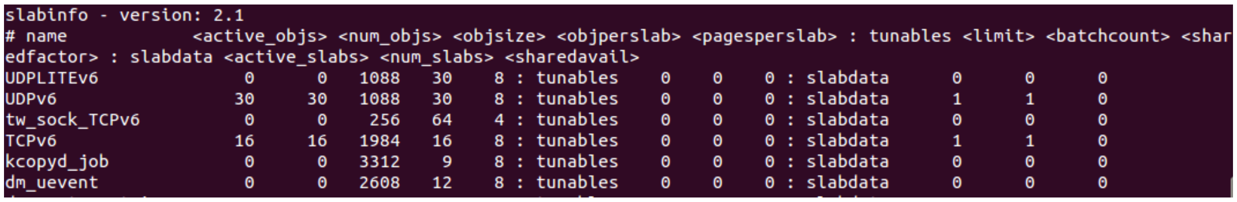


```c
void get_slabinfo(struct kmem_cache *s, struct slabinfo *sinfo)
{
        unsigned long nr_slabs = 0;
        unsigned long nr_objs = 0;
        unsigned long nr_free = 0;
        int node;
        struct kmem_cache_node *n;

        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

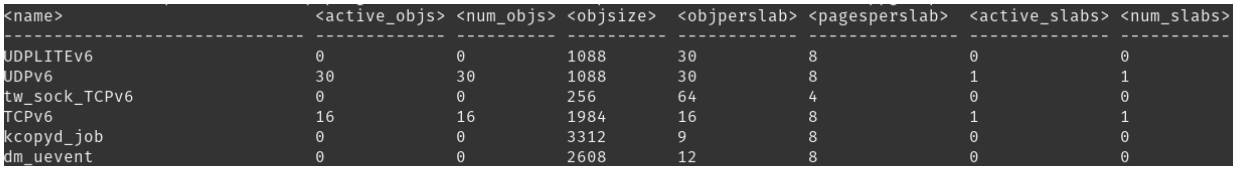
The implementation can be verified by comparing the results of reading /proc/slabinfo on a target machine and the output of the updated plugin.

Access to **kmem_cache** data from SLUB is crucial for optimizing our approach to



```
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



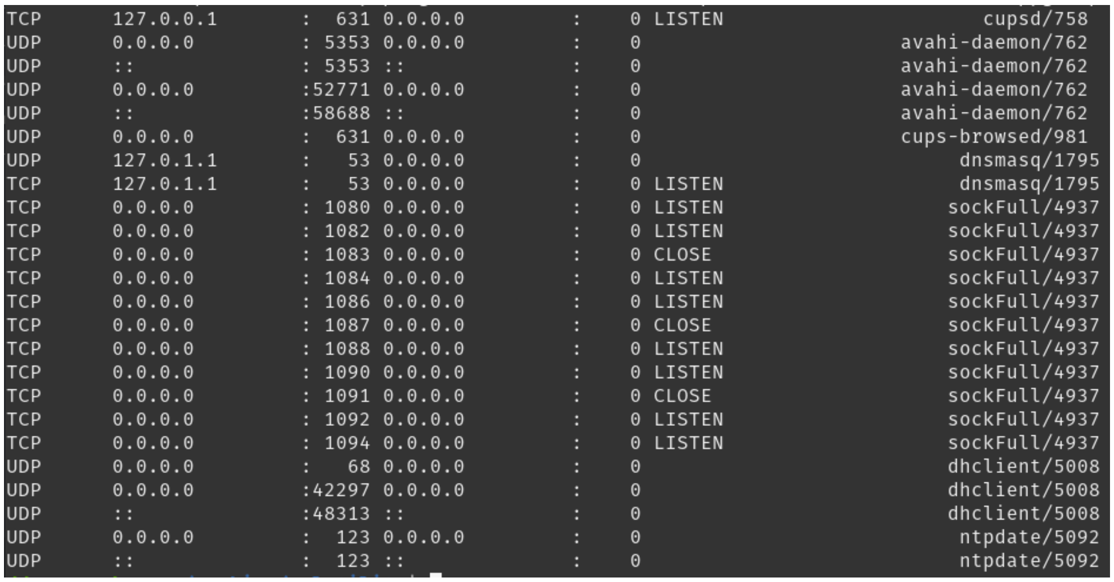
```
<name>                        <active_objs> <num_objs> <objsize>  <objperslab> <pagesperslab>  <active_slabs> <num_slabs>
----------------------------- ------------ ---------- ---------  ------------ --------------  -------------- ----------
UDPLITEv6                     0            0          1088       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

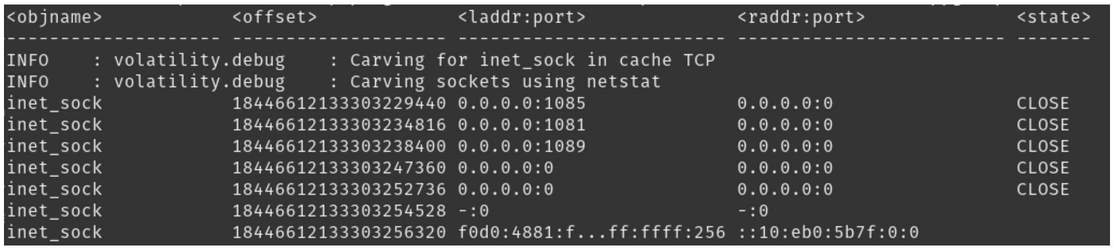
We can see the sockets found in our sample through a regular list walking approach by running the *netstat -U* Volatility command. The -U hides unix sockets from output, to simplify display.



```
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

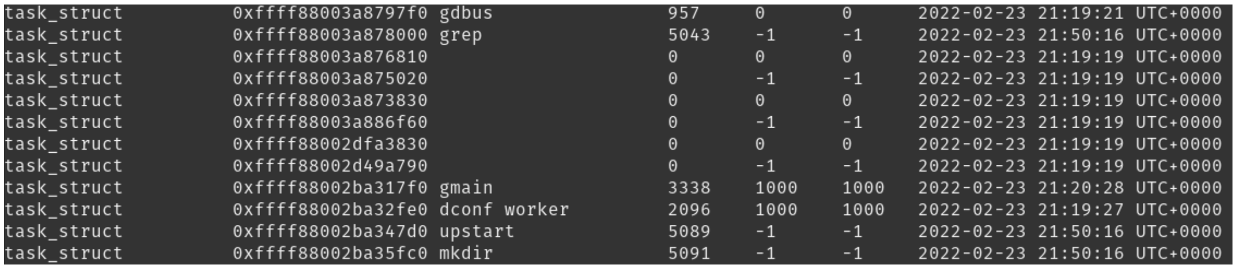
In this sample, the sockets for ports 1081, 1085, 1089, and 1093 are missing from the output, since they have been closed and freed by the artifact program ”sockFull”. When running the *slab_carve* plugin, the "hidden" sockets with ports 1081,1085, and 1089 were able to be extracted. While only three of the four original sockets were recovered in this case, these are artifacts that were all previously inaccessible.



```
<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
```

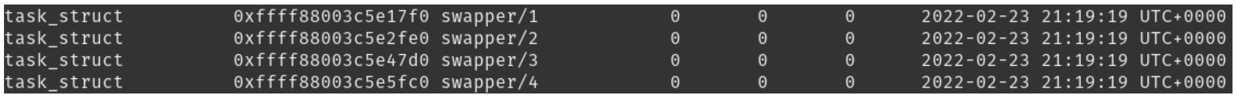
Figure 4.5. Result of running *slab_carve* on sample 1 looking for sockets

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.

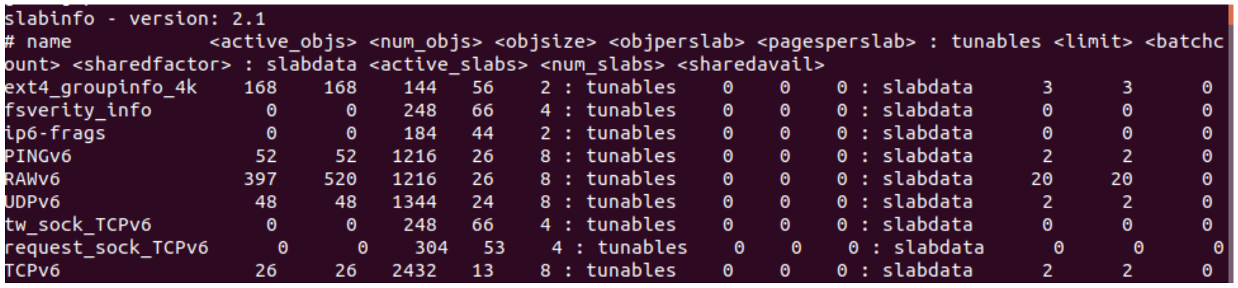


```
task_struct          0xffff88003c5e17f0 swapper/1            0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct          0xffff88003c5e2fe0 swapper/2            0       0       0       2022-02-23 21:19:19 UTC+0000
task_struct          0xffff88003c5e47d0 swapper/3            0      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

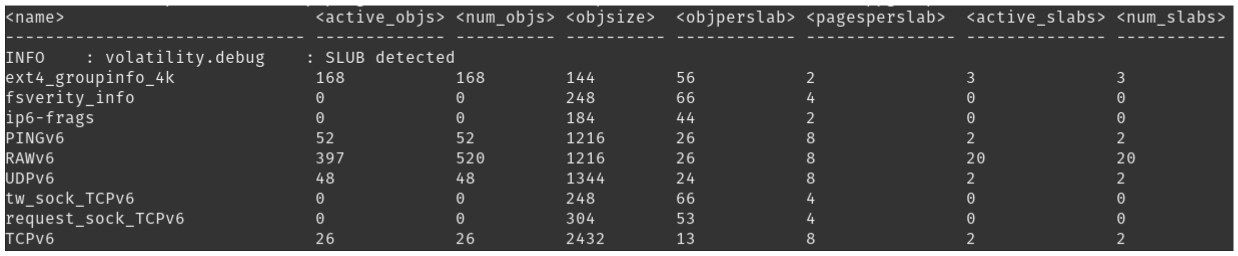
While we did not directly create processes to hunt for forensically, our approach is able to uncover processes not found through *pslist*, such as previously run commands on a shell like *grep* and *mkdir*. Additionally "swapper" processes were uncovered, which are not found within the Volatility process list by design.

We repeated our process on the other sample, first verifying that the updated *slab_info* works properly.



```
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchc
ount> <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
```

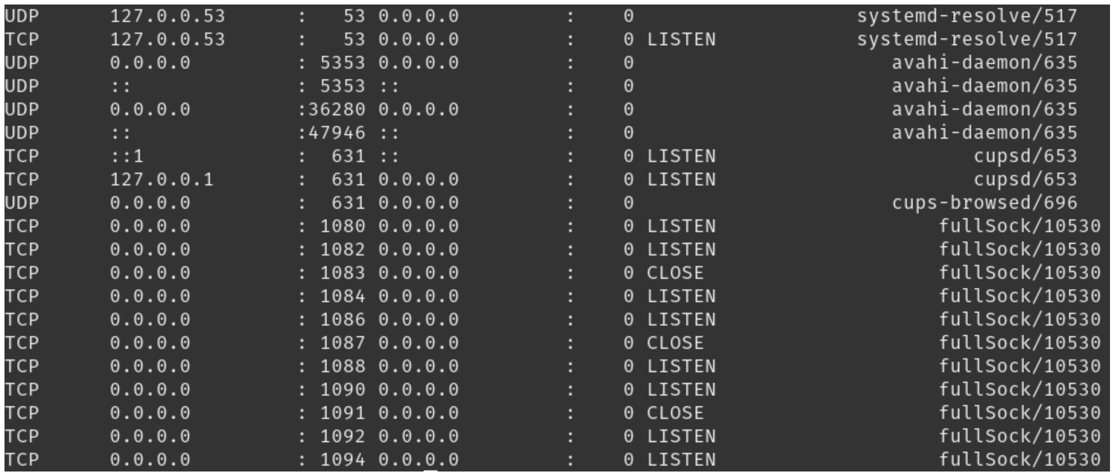
Figure 4.8. Result of running *cat /proc/slabinfo* on machine 2



```
<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

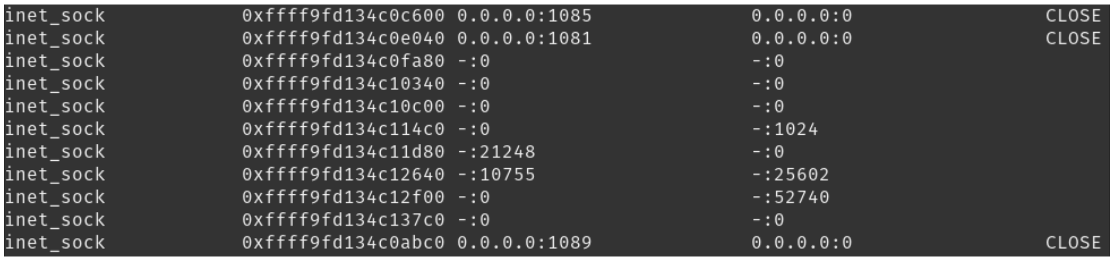
Then we gathered all of the sockets gathered from list enumeration through *netstat*, shown in figure 4.10



```
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
```

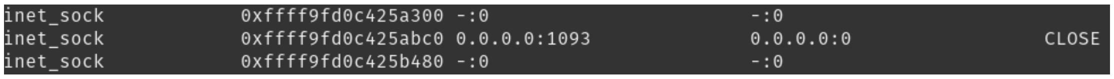
Figure 4.10. Result of running *netsat -U* on sample 2

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
```

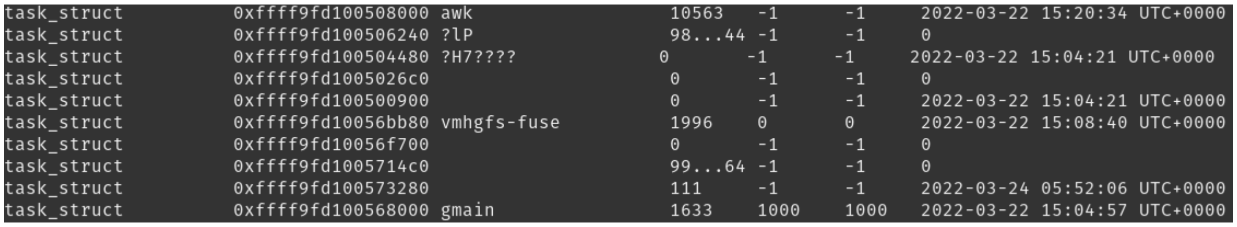
Figure 4.11. Result of running *slab_carve* on sample 2 looking for sockets



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

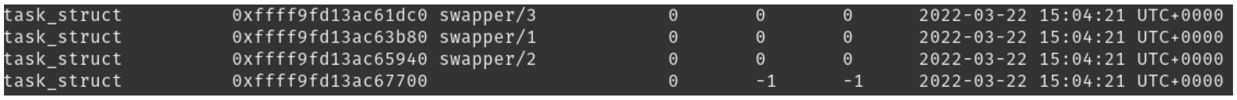
Figure 4.12. Result of running *slab_carve* on sample 2 looking for sockets

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

We were able to successfully recover three of the four sockets we had created to hunt for. This showcases the ability for our approach to recover forensic artifacts not found through list enumeration. Objects we created but were unable to find may be caused by one of the following scenarios: the free area the object once resided may have been overridden by another object after it was freed; there were no objects found by list enumeration that resided on the same slab the object was allocated on, leaving our approach blind to it; the slab had enough objects freed which triggered the freeing of the slab itself. The utility of our approach is further reinforced through the recovery of process information, providing a similar ability to the predecessor *pslist_cache* plugin. Finding "swapper" processes indicates that the carving approach can recover objects that are currently active on the system but are not acquired through list enumeration. Additionally for Linux systems finding old processes can assist an investigator in constructing a timeline of actions that occurred on a system.

Chapter 5. Conclusion and Future Work

5.1. Conclusion

With the introduction of our approach, investigators will be able to recover more forensic artifacts from SLUB systems than previously possible. This enables extraction of both recently freed and untracked objects. We implemented our approach in the *slab_carve* plugin which will allow investigators to analyze SLUB systems in Volatility. Additionally as a consequence of our work **kmem_cache** metadata is also able to be extracted from memory samples through the updated *slab_info* plugin, thus making enabling further analysis of SLUB systems.

5.2. Future Work

This research has provided a carving method to extract objects from SLUB's slabs. Future work would include resolving the slab **page**'s so that the partial list and full list can be used in conjunction with carving to extract as many objects as possible. Additional polish can be done to the work developed, such as adding the ability to target any object and cache on a Linux system. A filtering mechanism could be added to the plugin so analysts can ignore junk objects extracted without having to use another tool. The plugin was written with version 2.6 of Volatility, and we plan to adapt it to Volatility 3 when it releases from beta version.

As of this writing, Linux is currently introducing folios, a new representation of pages, which will impact memory forensics of all Linux systems. Analysis of folios and their forensic importance should be conducted after their full implementation.

References

[1] F. Block and A. Dewald, "Linux memory forensics: Dissecting the user space process heap," *Digital Investigation*, vol. 22, pp. S66 – S75, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301895?via%3Dihub

[2] F. Block and A. Dewald, "Windows memory forensics: Detecting (un)intentionally hidden injected code by examining page table entries," *Digital Investigation*, vol. 29, pp. S3 – S12, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287619301574

[3] A. Case, A. K. Das, S.-J. Park, J. R. Ramanujam, and G. G. R. III, "Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks," *Digital Investigation*, vol. 22, pp. S86–S93, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301986

[4] A. Case, L. Marziale, C. Neckar, and G. G. R. III, "Treasure and tragedy in kmem_cache mining for live forensics investigation," *Digital Investigation*, vol. 7, pp. S41–S47, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287610000332

[5] Fire Eye. (2019) Finding evil in windows 10 compressed memory. [Online]. Available: https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/finding-evil-in-windows-10-compressed-memory-wp.pdf

[6] Kaspersky. (2015) The mystery of duqu 2.0: a sophisticated cyberespionage actor returns. [Online]. Available: https://securelist.com/the-mystery-of-duqu-2-0-a-sophisticated-cyberespionage-actor-returns/70504/

[7] N. Lewis, A. Case, A. Ali-Gombe, and G. G. R. III, "Memory forensics and the windows subsystem for linux," *Digital Investigation*, vol. 26, pp. S3 – S11, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287618301944

[8] Linux. (2022) Kconfig. [Online]. Available: https://elixir.bootlin.com/linux/v2.6.23/source/init/Kconfig#L539

[9] Linux. (2022) slob.c. [Online]. Available: https://github.com/torvalds/linux/blob/master/mm/slob.c#L11

[10] G. G. Richard and A. Case, "In lieu of swap: Analyzing compressed ram in mac os x and linux," *Digital Investigation*, vol. 11, pp. S3 – S12, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287614000541?via%3Dihub

[11] StackOverflow. (2019) Developer survey results. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology-_-developers-primary-operating-systems

[12] J. Stüttgen and M. Cohen, "Anti-forensic resilient memory acquisition," *Digital Investigation*, vol. 10, pp. S105 – S115, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287613000583?via%3Dihub

[13] J. Stüttgen, S. Vömel, and M. Denzel, “Acquisition and analysis of compromised firmware using memory forensics,” *Digital Investigation*, vol. 12, pp. S50–S60, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287615000110#!

[14] J. Sylve, V. Marziale, and G. R. III, “Pool tag quick scanning for windows memory analysis,” *Digital Investigation*, vol. 16, pp. S25 – S32, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287616000062?via%3Dihub

[15] The Volatility Foundation. (2020) About the volatility foundation. [Online]. Available: https://www.volatilityfoundation.org/about

[16] The Volatility Foundation. (2020) Wiki home. [Online]. Available: https://github.com/volatilityfoundation/volatility/wiki

[17] D. Uroz and R. J.Rodríguez, “Characteristics and detectability of windows auto-start extensibility points in memory forensics,” *Digital Investigation*, vol. 28, pp. S95 – S104, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287619300362

[18] W3techs. (2022) Usage statistics of operating systems for websites. [Online]. Available: https://w3techs.com/technologies/overview/operating_system

[19] Wikipedia. (2022) Usage share of operating systems. [Online]. Available: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems

Vita

Daniel Ashton Donze was born in Silverspring, Maryland. He graduated with his B.S. in Computer Science, with minors in Mathematics and Physics, from Louisiana State University in 2020. His exposure to cyber-security during his bachelor's degree led him to pursue his master's degree in Computer Science at Louisiana State University. He plans to receive his master's in Computer Science in May 2022.