James Madison University

CS633 Project 3

Author:
Josh Feehs

April 3, 2014

1 Executive Summary

For this project, I was tasked with analyzing a memory dump from an acme machine that was running the linux 2.6.24 kernel. In order to analyze what processes were running and what files those processes had open, I had to write my own versions of the linux `ps` and `lsof` tools.

I chose to use python to create both of these programs, as I knew that it should run on any computer that has python installed on it without needing to worry about specific compilation flags or other platform-specific information.

2 Initial setup

In order to start ps (and eventually work on lsof), I first had to download the linux kernel source code. It was through careful inspection of the linux source code that I would be able to locate offsets for all of the various fields I needed to find.

Fortunately, I was also provided with a system map for the linux kernel. This map contained the offset for the first task_struct, the structure that would contain the information I needed for these programs, or a pointer to a structure that did. I did all of my searching for fields (both for ps or lsof) using a hex editor to identify potential offset values, and then ran my python program to attempt to verify if a specific potential offset was correct or not.

3 ps

Given my initial offset into the task_struct, I decided to start by focusing on the necessary fields that would be easy to find. The first field that would be easy to find was the name field. In Linux, it is highly likely that the first task is the "swapper" task. Therefore, in order to find the offset to get to the name field, I navigated in my hex editor to the top of the first task_struct, and searched downward until I found the bytes that corrresponded to the string "swapper". In the case of this memory dump, there were two such potential places. I would able to check this offset later once I had more task_structs to investigate.

From this point, the next important field for me to find would be the pointer to the next task_struct. The pointers to the next and previous tasks would be stored in a struct list_head called "tasks", and I found that those pointers were the first of at least eight kernel pointers that followed after some integers. I also knew that this would have to happen before the name. So, I started to look for a large enough block of pointers. When I found a potential candidate for the next pointer, I navigated to its address in my hex editor and looked for the word "init" to follow, as I knew it was highly likely that the second task was the init task. I had to check a few pointers, but I eventually found that the address of the first "next" pointer was 0x3ed42c, which meant that the offset from the top of a task_struct to the next field is 0x9c.

At this point, I had a working program that could go through each of the task_structs and print their name. Running this program allowed me to check that I had found the correct "swapper" when trying to find the name, and I determined that the offset from the top of a task_struct to the name field was 0x1cd.

The next field that I wanted to find way the PID. In this case, given the layout of the task_struct, I found that finding the pointer to the parent was going to be one of the easier ways to locate the PID, so I looked for them both in tangent. I found that the PID was a bit below the next pointer and right before a potential canary value. The parent field was a pointer that came right after the canary value. So, in order to find them, I started a bit below the next pointer and looked for a bunch of integers followed by at least 7 kernel pointers. It turned out that the kernel pointers were the easiest to find, so I located the parent at offset 0xd4 from the top of a task_struct. From there, I only had to determine if the integer right before the parent was a canary value or a tgid, which was almost always the same as the pid. I knew that the PID for swapper should be 0 and that the PID for init should be 1, so this allowed me to find that the offset for the PID was 0xc8, as the canary value was in fact present. Now that I had the PID offset and a pointer to the parent task_struct, getting the PPID was as simple as following the parent pointer and pulling it's PID field.

The next field that I wanted to find was the state of the process. This was easy to find, as it was the very top of the task_struct. I also knew from the demos in class that it was highly likely that there were only a few different states that a process would actually be in, so I pulled those from the header file and checked the state number against those. I verified that my checker did in fact label each of the processes in the virtual memory image with a valid state.

The next fields that I wanted to find were the uid and gid fields. According to the kernel file, these would be the first and fifth of eight integers that had kernel pointers before and after them. I also knew that they would be located close to the name, as they came a few lines before the name field i in the specification. Given the above information, I started at the name field and searched upwards until I found a set of integers. I decided the best way to verify that these were in fact the uid and gid was the run my program and look at the output values. Most user programs would have a uid and gid in the 1000's. Once I found the integers that had 0's for kernel processes and values in the 1000's for user processes like less, I knew that I had correctly found the uid and gid. Their offsets within a task_struct are 0x18c and 0x19c.

The remaining fields that I needed to find were the rss and the total vm. From class, I knew that these would be located in the mm_struct. Finding a pointer into the mm_struct was simple, as it was located after the next and previous pointers, seperated by two list_heads, which were both two pointers. This meant that the offset into the mm_struct from the top of the task_struct is 0xa4.

Once I was in the mm_struct, I needed to find the hiwater_rss and total_vm fields. These were near the top of a large block of integers that followed a pair

of kernel pointers. Using this, I was able to find that the offset to the rss was 0x58, and the total_vm field was 8 bytes after it.

That concludes my search for all of the fields needed for ps.py. Now that I had found all of the correct offsets, my python program could simply follow the next pointers from task_struct to task_struct and collect and report all of the information.

4 lsof

Now that I completed ps, it was time to move on to lsof. I decided that I needed to complete most of ps before moving on because one of the inputs to lsof is a pid. With ps complete, I now had a list of all of the valid pid's for the memory dump.

The first part of lsof was handling the input and getting to the correct process. I was able to reuse the knowledge I had gained from ps, and use all of my pre-determined offsets to iterate through all of the task_structs and check to see if its pid was the same as the one I was searching for. If I found that I made a complete pass through all of the task_structs and had not found the pid that was asked for, I quit the program and reported that the requested pid was not valid.

The next field I wanted to find was the pointer to the files_struct as this would be the entry point for all of the other data that I needed. According to the Linux kernel, this was a pointer that came after the name field, and was the second of a block of five pointers. So, searched through the hex data, starting at the name field, and looked for those pointers. When I found the potential candidate, I went there to see if the data pointed to by the pointer started with an integer (atomic_t count) and two pointers. When it did, I knew that it was highly likely that I was correct. The offset to the files_struct is 0x484 into the task_struct.

From there, the next thing I needed was the pointer to the fdtable, as the fdtable stores the pointer to the file array and the bitmap of open files. Fortunately, the pointer to the fdtable is right near the beginning of the files_struct, so finding it was trivial, as it is 4 bytes in.

Once I was in the fdtable, the fields that were of interest to me were the pointer to the open_fd bitmap and the pointer to the files array. Given that the fdtable was straight-forward, I knew from reading the documentation that the pointer to the bitmap was 12 bytes in to the fdtable, and the pointer to the files array was 4 bytes in to the fdtable. From there, I was able to get the data from the open fds bitmap and know which files in the files array would be valid.

From this point, my lsof program loops over each valid file (determined by the open fd bitmap) and searches for the following data that lsof needs to report.

The first field that I wanted to find was the name and path of the file. This was the multistep process. Step 1 was to locate the struct path within the file itself. Fortunately, this struct, which contains pointers to the root and tail of the path, is located 8 bytes into the struct file that the fdarray points to. It's

first field is the pointer to the virtual file system mount, struct vfsmount, and the second field, 4 bytes in, points to a struct dentry for the end of the path for the file. The only field that is important out of the vfsmount is a pointer to a dentry that has the name information for the root directory for the particular file. This field is 16 bytes into a vfsmount, which like other fields earlier, could be calculated from the kernel documentation and header files.

From the Linux kernel files, the dentry has a struct qstr that has the name. This "qstr" has a few empty pointers at the beginning before it has the actual name of the file or directory that the dentry represents. Using knowledge of the dentry struct and the qstr struct, I was able to look through the hex and find that the name field started 36 bytes into the dentry. In the case of the vfsmount's dentry field, this name is the root of the virtual filesystem of the file, and was generally "/", "pipe:", or "socket:". This name field of the main dentry (the one pointed to by the path struct) was the actual name of the file, like "acme_secrets.txt". However, this was not the whole path to the file. In order to get the full path, I needed to find the parent directories of the file. Fortunately, there is a pointer within the dentry (right before the qstr struct; 24 bytes in) that points to the dentry's parent. This meant that after I grabbed the file name, I could pull the pointer to the parent dentry, go there, and get the parent's name. My program traverses up the tree until the parent pointer is null, the name pointer of the parent is null, or the parent's name is equal to the name of the root directory (grabbed from the vfsmount.) Because the parent was simply another dentry, finding its name and parent was exactly the same as finding the first name and first parent.

At this point, I now had the name of each file. Now that I could see that I was getting reasonable names for files, I was confident that I had correctly found the offsets for the files_struct, the fd array, and the dentry, as all of those needed to be correct for me to have gotten valid names. From here, I decided to get all of the other requested fields that I could find in those structs. The next one of these fields was the f_mode. This is a number that tells what mode the file descriptor was opened in (read, write, etc). This field was simple to find, as there are no conditional variables in the file struct, so I could count down the fields and find that this was 28 bytes from the top of the file. I was able to interpret the value that was at this address using hard-coded values that were also included in fs.h.

From here, the only remaining fields I needed to find for lsof were stored in the inode struct. Earlier, I had found that the inode was stored in the dentry, and using the kernel files that defined a dentry, I was able to find that the pointer to the inode was 12 bytes into the dentry. So, in order to get any information from the inode, I had to go back to the main dentry (the second pointer in the struct path in the struct file), so 12 bytes in, and follow the pointer that was there.

Now that I was in the inode, I needed to find the inode number, the file size, the major device number, and the type of the file. From class, I knew that these fields were called i_no, i_size, i_rdev, and i_mode, respectively. Going through the struct and confirming that the values in the actual memory dump appeared

to be of the correct type, I found that the offset for the inode number was 32 bytes, the offset for the size was 60 bytes, the offset for the type was 106 bytes, and the offset for the device number was 52 bytes, all of which offsets from the top of the inode.

From here, I had collected all of the necessary fields. The remainder of my program simply prints out all of the collected values and then loops back around to collect the information for the next open file.

5 Answers to Questions

The question asked of me for this project was to determine which processes are running, and to describe some of those processes. The following is the output of my ps program. Following the raw output, I will discuss a few of the processes that were running.

kernel-addr	pid	ppid	uid	gid	name	state	rss	vsz
0xc03ed3a0	0	0	0	0	swapper	RU	-1	-1
0xcd8f0000	1	0	0	0	init	IN	426	717
0xcd8f05c0	2	0	0	0	kthreadd	IN	-1	-1
0xcd8f0b80	3	2	0	0	migration/0	IN	-1	-1
0xcd8f1140	4	2	0	0	ksoftirqd/0	IN	-1	-1
0xcd8f1700	5	2	0	0	watchdog/0	IN	-1	-1
0xcd8fe000	6	2	0	0	events/0	IN	-1	-1
0xcd8fe5c0	7	2	0	0	khelper	IN	-1	-1
0xcd966000	41	2	0	0	kblockd/0	IN	-1	-1
0xcd967140	44	2	0	0	kacpid	IN	-1	-1
0xcd967700	45	2	0	0	kacpi_notify	IN	-1	-1
0xcd941140	179	2	0	0	kseriod	IN	-1	-1
0xcda03700	217	2	0	0	pdflush	IN	-1	-1
0xcda0e000	218	2	0	0	pdflush	IN	-1	-1
0xcda0e5c0	219	2	0	0	kswapd0	IN	-1	-1
0xcda7eb80	260	2	0	0	aio/0	IN	-1	-1
0xcd8ff140	1542	2	0	0	ata/0	IN	-1	-1
0xcda7e5c0	1545	2	0	0	ata_aux	IN	-1	-1
0xcdb84000	1558	2	0	0	scsi_eh_0	IN	-1	-1
0xcd940000	1565	2	0	0	scsi_eh_1	IN	-1	-1
0xcf8d0b80	1580	2	0	0	ksuspend_usbd	IN	-1	-1
0xcd940b80	1585	2	0	0	khubd	IN	-1	-1
0xcfa9eb80	2484	2	0	0	scsi_eh_2	IN	-1	-1
0xcd9405c0	2692	2	0	0	kjournald	IN	-1	-1
0xcd8feb80	2896	1	0	0	udevd	IN	176	562
0xcda03140	3283	2	0	0	kgameportd	IN	-1	-1
0xcdb85140	3414	2	0	0	kpsmoused	IN	-1	-1
0xcd9665c0	4962	1	0	0	vmtoolsd	RU	984	6503
0xcd8ff700	5157	1	0	0	getty	IN	134	435
0xcdb85700	5160	1	0	0	getty	IN	133	435
0xcfa9f700	5166	1	0	0	getty	IN	134	435
0xcf8d0000	5169	1	0	0	getty	IN	134	435
0xcda7f140	5171	1	0	0	getty	IN	134	435
0xcda02b80	5326	1	0	0	acpid	IN	332	620
0xcf8d1140	5376	2	0	0	kondemand/0	IN	-1	-1
0xcda0f140	5433	1	102	103	syslogd	IN	176	490
0xcda0eb80	5490	1	0	0	dd	IN	132	474
0xcda02000	5492	1	103	104	klogd	IN	529	831
0xcea78b80	5514	1	107	116	dbus-daemon	IN	293	714
0xcea79700	5530	1	0	0	NetworkManager	IN	492	1151
0xcfa9e5c0	5544	1	0	0	NetworkManagerD	IN	296	853
0xcda0f700	5557	1	0	0	**system**-tools-ba	IN	280	1034
0xcda7e000	5579	1	0	0	sshd	IN	248	1335
0xcda7f700	5600	1	108	117	avahi-daemon	IN	454	814
0xcfa9e000	5601	5600	108	117	avahi-daemon	IN	119	696
0xcd966b80	5639	1	0	0	cupsd	IN	635	1519
0xce5ff140	5713	1	0	0	dhcdbd	IN	187	511
0xcf8d05c0	5732	1	110	120	hald	IN	2293	1496
0xce5fe5c0	5735	1	0	0	console-kit-dae	IN	584	1964
0xce23c000	5797	5732	0	0	hald-runner	IN	270	815

53	0xce23c5c0	5819	5797	110	120	hald-addon-acpi	IN	227	557
54	0xce23d700	5823	5797	0	0	hald-addon-inpu	IN	262	831
55	0xce23d140	5833	5797	0	0	hald-addon-stor	IN	256	832
56	0xce31eb80	5840	5797	0	0	hald-addon-stor	IN	260	832
57	0xcf8d1700	5860	1	0	0	hcid	IN	304	769
58	0xce31e000	5866	2	0	0	btaddconn	IN	-1	-1
59	0xcdd5a000	5867	2	0	0	btdelconn	IN	-1	-1
60	0xce23cb80	5874	5860	0	0	bluetoothd-serv	IN	253	731
61	0xcfa9f140	5876	5860	0	0	bluetoothd-serv	IN	303	747
62	0xcddc8000	5882	2	0	0	krfcommd	IN	-1	-1
63	0xcddc9700	5915	1	0	0	gdm	IN	402	3519
64	0xcea785c0	5987	1	0	0	atd	IN	108	502
65	0xcdd5b700	6001	1	0	0	cron	IN	224	532
66	0xcdd8d700 14879	6069	1	65534	65534	monkey	IN	247	
67	0xcd8c1700	6099	1	0	0	getty	IN	134	435
68	0xcd8c05c0	6216	1	1000	1000	gnome-keyring-d	IN	246	1502
69	0xcdd5a5c0	6424	5915	0	1000	gdm	IN	740	3626
70	0xc0d1ab80 14297	6428	6424	0	0	Xorg	IN	6722	
71	0xcd8c1140	6463	6424	1000	1000	x-session-manag	IN	2870	7463
72	0xcdd8c000	6533	6463	1000	1000	ssh-agent	IN	132	1126
73	0xce31e5c0	6560	1	1000	1000	xfce-mcs-manage	IN	1420	6009
74	0xc0d1a5c0	6565	1	1000	1000	gconfd-2	IN	721	1462
75	0xc0d1b140	6566	1	1000	1000	gnome-keyring-d	IN	247	1502
76	0xcd8c0000	6568	1	1000	1000	xfwm4	IN	2396	4985
77	0xcdb845c0	6570	1	1000	1000	xfce4-panel	IN	2852	5260
78	0xcddc9140	6572	1	1000	1000	Thunar	IN	1228	3799
79	0xcddc8b80	6573	6570	1000	1000	xfce4-menu-plug	IN	2631	5792
80	0xce31f700	6575	1	1000	1000	gam_server	IN	291	727
81	0xce588000	6576	6570	1000	1000	xfce4-places-pl	IN	2684	5431
82	0xc0d1a000	6578	1	1000	1000	xfdesktop	IN	5624	7808
83	0xcfbfe000	6582	1	1000	1000	dbus-daemon	IN	239	648
84	0xcfbfe5c0	6583	1	1000	1000	dbus-launch	IN	154	758
85	0xcfbfeb80	6586	6570	1000	1000	thunar-tpa	IN	1849	4600
86	0xcd8c0b80	6591	1	1000	1000	update-notifier	IN	3441	6806
87	0xc3f54b80	6595	1	1000	1000	nm-applet	IN	3071	6947
88	0xc3f55140 14585	6596	1	1000	1000	vmtoolsd	RU	3973	
89	0xcda025c0	6598	1	1000	1000	python	IN	3021	6046
90	0xcc7625c0	6610	1	1000	1000	gnome-power-man	IN	1864	5770
91	0xc6378b80	24408	5915	0	1001	gdm	IN	692	3627
92	0xc6379700 15547	24411	24408	0	0	Xorg	IN	5631	
93	0xcb2d0b80	24472	24408	1001	1001	sh	IN	70	449
94	0xcb2d1700	24631	24472	1001	1001	ssh-agent	IN	133	1126
95	0xc6378000	24641	1	1001	1001	dbus-launch	IN	163	758
96	0xc6379140	24642	1	1001	1001	dbus-daemon	IN	242	648
97	0xc3f55700	24649	24472	1001	1001	xfce4-session	IN	1289	5763
98	0xcc762b80	24655	1	1001	1001	gconfd-2	IN	720	1462
99	0xcc763140	24656	1	1001	1001	gnome-screensav	IN	869	3807
100	0xcc763700	24657	1	1001	1001	xfce-mcs-manage	IN	1757	6792
101	0xcb2d1140	24659	1	1001	1001	gnome-keyring-d	IN	332	1502
102	0xcfbff700	24661	1	1001	1001	xfwm4	IN	2425	4985
103	0xcdd5ab80	24663	1	1001	1001	xfce4-panel	IN	3215	6047
104	0xce5885c0	24665	1	1001	1001	Thunar	IN	1221	3799
105	0xcb2d0000	24667	1	1001	1001	gam_server	IN	290	727
106	0xcfbff140	24668	24663	1001	1001	xfce4-menu-plug	IN	3421	6599
107	0xc3f54000	24669	24663	1001	1001	xfce4-places-pl	IN	3058	6208
108	0xcc2c4000	24671	1	1001	1001	xfdesktop	IN	5958	8588
109	0xcc2c4b80	24673	24663	1001	1001	thunar-tpa	IN	2178	5379
110	0xc14f0000	24677	1	1001	1001	update-notifier	IN	3774	7588
111	0xc14f1700	24682	1	1001	1001	nm-applet	IN	3491	7700
112	0xcc2c5140	24684	1	1001	1001	python	IN	3022	6048
113	0xc63785c0 13405	24685	1	1001	1001	vmtoolsd	RU	3101	
114	0xcb2405c0	24693	1	1001	1001	gnome-power-man	IN	1874	5771
115	0xc3d64b80 52048	24741	24668	1001	1001	firefox-bin	IN	13393	
116	0xcef22b80	24770	24668	1001	1001	thunderbird	IN	131	449
117	0xc1487700	24782	24770	1001	1001	run-mozilla.sh	IN	133	449
118	0xc1486000 38277	24786	24782	1001	1001	thunderbird-bin	IN	11058	
119	0xc74d8000	24866	24668	1001	1001	gcalctool	IN	4616	8115
120	0xc74d9140 16028	24872	24668	1001	1001	xfce4-terminal	IN	4890	
121	0xc74d9700	24875	24872	1001	1001	gnome-pty-helpe	IN	188	705
122	0xc74d8b80	24876	24872	1001	1001	bash	IN	277	1420
123	0xc1643140	24912	24876	1001	1001	less	IN	174	816

A vast majority of the processes running are system processes. Processes such as ssh-agent show that the computer was likely using ssh, which could make sense as a user bash shell is open, as is less, a bash utility. This also shows that the Thunderbird mail client was running, that a python script was running, and that firefox was also likely running. The existence of the process "vmtoolsd" likely means that this was a VMWare virtual machine running with VMWare tools installed. The many "gnome*" processes show that user 1001 was likely using the gnome desktop.

The ps output also shows that processes that belong to user 1000 are also running, such as python and other gnome utilities like the gnome keyring. This may suggest that multiple users were logged on the machine when the memory image was seized, or at least that some of user 1000's processes did not stop when they logged off. Given that none of user 1000's processes were programs with a GUI, if user 1000 was logged in, I think they were logged in remotely via ssh, whereas I am fairly sure that user 1001 was logged in on the physical machine at the time that the memory was seized.