CAN HACKERS CASH-IN ON THE SENSITIVE DATA CONTAINED IN CACHE?

Erich Rice, Dennis Guster, Li Dai

Department of Information Systems

St. Cloud State University

St. Cloud, MN 56301

eprice@stcloudstate.edu

Abstract

The threat to IT systems and applications continues to be a concern for organizations. While many of the current attacks are centered at the application layer, these types of attacks are not the only threat that requires protection. While it's easy for semi-sophisticated hackers to procure an organization's email addresses and target them by sending mass phishing emails, more sophisticated hackers may utilize techniques which allow for stealthier extraction of data. These more sophisticated hackers could seek to extract data directly from memory, thereby bypassing security controls placed to try and block or log unusual activity. By extracting the confidential data directly from the real or virtual memory a hacker could get what they desire without raising alerts, which might bring about efforts to try and stop their activities. This type of attack could be especially useful if it was on shared hardware, such as within a public computing cloud.

1 Introduction

Hackers are constantly looking for ways to compromise sensitive data. Certainly, the easiness of the method to compromise a system or application is a major consideration for them (Chng et al., 2022). Also of concern, is the probability of being detected in their hacking attempts. If standard file systems or databases are attacked, those attacks will typically be logged as well (Tayag, De Vigal Capuno 2019).

One potential way for hackers to bypass the logging system is to use a lower-level architecture of the computer system and pull the data they seek directly from memory. Although this method is more complex than the attack methods that are run through the application layer, such as through phishing emails, once they are developed and proven they are well shared and become viable for serious hackers (Koon, 2022). Therefore, these types of attacks should not be taken lightly as indicated by the nearly 70% of Microsoft security issues being memory safety related (Cimpanu, 2019). This type of scenario is especially of concern when it occurs on the backend server side. For example, a Linux host that is acting as a server used to support a company's e-commerce sales system. In this case it is not the data of one client that could be compromised, but the data of all the client devices that connect to the server to use the e-commerce system.

A good summary of how memory is allocated on a Linux host can be obtained by using the "free -h" command (Linuxize, 2020). In the example below taken from a Linux host on the authors' private computing cloud, the key is the first row that delineates the real memory. In that row it is found that the system has 31 GB of total real memory.

```
dennis.guster@eros2:~$ free -h

           total    used    free   shared/buff  cache   available

Mem:       31G      694M    24G       21M       5.9G      30G

Swap:      8.0G     0B      8.0G
```

However, only 694 MB are in use, leaving 24 GB of memory free. Of interest to this paper is the fact that there are 5.9 GB allocated for buffers and cache. In both cases, the use of a buffer and cache, are used to speed up the processing of data. A good example of a buffer would be the process of reading data from a sequential file from the disk. Disk access, even on a solid-state drive (SSD), is significantly slower than reading directly from memory. So, the read request goes directly into memory from disk and the process then pulls from the buffer speeding up the process. However, the process from the disk to the buffer is on-going and the read from the buffer goes on when the process gets the interrupt. The assumption is that the process may have to go through a series of wait states caused by reading other files, waiting for human input, or writing to a log file.

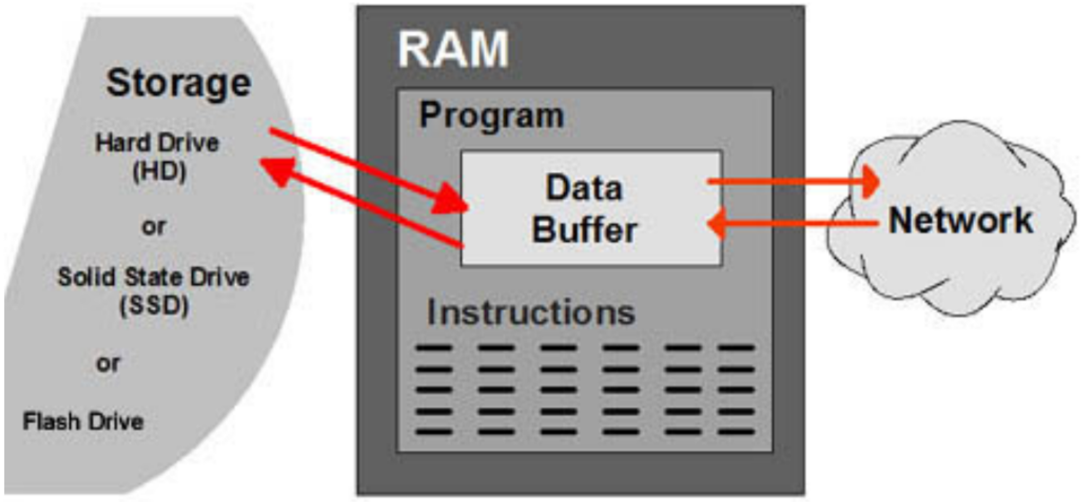

Figure 1: How a Memory Buffer Works (*Definition of buffer,* n.d.*)*

In the case of memory cache, also sometimes called a "CPU cache", information is either read or written quickly to the cache and periodically the cache is cleared (*Definition of cache,* n.d.). After this process has been completed then the cache space can then be reused. It is possible to evaluate some of the characteristics of a file that has been placed in memory on a Linux host. In the example below, a simple java program creates a file called Data.txt and two lines are sent to that file.

```
dennis.guster@eros2:~$ java ReadWriteZ

Type characters to write in File – Press Ctrl+z to end

Line 1: ps

Line 2: this file contains Data
```

However, when one looks in memory using the "lsof" command (Zivanov, 2022) at the open file for process ID 22177 there is no data in the file as indicated by the 0 in italics as seen below.

```
dennis.guster@eros2:~$ lsof -p 22177

java    22177 dennis.guster    6w   REG   0,52        0   27659826 /rhome/dennis.guster/Data.txt
(10.10.3.5:/exports/rhome)
```

A check on the file system level also reveals that no data has been written as well.

```
dennis.guster@eros2:~$ ls -l /rhome/dennis.guster/Data.txt

-rw-r--r-- 1 dennis.guster professors  0  Nov 10 11:46 /rhome/dennis.guster/Data.txt
```

After closing the java program, the data is then written to the file.

```
dennis.guster@eros2:~$ cat Data.txt

Line 1: ps

Line 2: this file contains Data
```

From the examples above it is clear that the data is initially going into a memory buffer. However, that buffer could be contained in real memory. As the free command displayed earlier, it indicated there was 5.9 GB allocated by the operating system for that purpose. It is also possible that the initial location of the data could be in virtual memory. Regardless, it is hoped that the data that is stored in memory will not be compromised by a hacker. For a hacker to accomplish such a feat they would need to know the relative address of the memory storage area where the pertinent data resides. The structure of those memory addresses is a hexadecimal number, which represents the relative bit address of the data. In some cases, those addresses are fixed, while in other cases (typically involving virtual memory) those address can vary (Sterling, Anderson, & Brodowicz 2018). In a 64-bit computer the memory addresses range from 0000000000000000 to ffffffffffffff or 16^16 possible bits (Vostokov, 2023). So even with fixed addresses, guessing where data might be stored is not a trivial matter for a potential hacker. With virtual memory the addresses are likely to change from execution to execution (Sterling, Anderson, & Brodowicz 2018). So, finding a virtual memory address to attack may have a very limited and short value.

However, the processes themselves need to know where the data is stored so it can map to the data needed to exist within the operating system. In the example below, one can see the memory segments related to the process ID 21087. Obviously, this is still a lot of memory to evaluate on a trial-and-error basis to find what you are looking for.

```
dennis.guster@eros2:/proc/21087/map_files$ ls

55f0371b3000-55f0371bb000  7fe994f19000-7fe994f1b000  7fe995137000-7fe995138000  7fe99513f000-
7fe995140000 55f0373ba000-55f0373bb000  7fe994f1f000-7fe994f48000  7fe995138000-7fe995139000
7fe995140000-7fe995147000
```

To investigate the probability of a hacker finding the memory address of potentially sensitive data contained in either real or virtual memory, the example that follows uses an existing sequential file and appends data to it. Of course, the file could be attacked directly, however the probability of an attack like that being detected is quite high. So, the rationale of a hacker reading the sensitive data from memory would be that it would be stealthier, thus resulting in a lower chance of detection and an alert being raised. In the next example seen below, the file "filexyz" is opened and appended to by using the Linux "cat" command.

```
dennis.guster@eros2:~$ ls file*

file2011b  fileEX   filetcp483  filetoremove2  filexyz
```

```
dennis.guster@eros2:~$ cat >> filexyz

sample output

more sample output
```

By looking at the "io" file for process PID 21087 one can see that 36 characters have been appended to the file but note that the write bytes value is 4096 Bytes, which matches the page size currently being used by the Linux based operating system. This makes addressing and allocating memory a little cleaner.

```
dennis.guster@eros2:/proc/21087$ cat io

rchar: 4979

wchar: 36

syscr: 13

syscw: 4

read_bytes: 0

write_bytes: 4096

cancelled_write_bytes: 0
```

It is also possible to see if the file is open in conjunction with the process PID of the application. To do so, the list open files command "lsof" can be used (Zivanov, 2022). The file "filexyz" shows up as a regular file containing 62 Bytes of data at this point. Also of note is the fact that it is not currently on the Linux host, but on another as indicated by the private "10.10.3.5" IP address as the file is housed on an NFS (Network File System) server in the same private cloud environment. This hopefully would make it more difficult for a hacker to attack the file system directly, however it is important to remember that data will be temporarily stored in memory on the host computer prior to being sent across the network to the NFS server for storage.

```
dennis.guster@eros2:~$ lsof -p 19798 | grep filexyz

cat     19798 dennis.guster    1w   REG   0,52       62 27659830 /rhome/dennis.guster/filexyz
(10.10.3.5:/exports/rhome)
```

To determine if the file is in real or virtual memory the "vmtouch" command can be used to ascertain its location (Carrigan, 2020). In the example below it is checked to see if it is currently in the virtual memory, however it can be seen that it is stored in one page of memory, and it is used to store it in the real memory. Then the "filexyz" file is "touched" into virtual memory once again taking up one page of 4KB of space within the virtual memory space. On a side note, it is also possible to remove a file from virtual memory again using the "vmtouch" command (Carrigan, 2020).

```
dennis.guster@eros2:~$ vmtouch -v filexyz

filexyz

          Files: 1

     Directories: 0

  Resident Pages: 1/1  4K/4K  100%

        Elapsed: 0.001655 seconds


dennis.guster@eros2:~$ vmtouch -vt filexyz

filexyz

          Files: 1

     Directories: 0

   Touched Pages: 1 (4K)

        Elapsed: 0.001412 seconds


dennis.guster@eros2:~$ vmtouch -ve filexyz

Evicting filexyz

          Files: 1

     Directories: 0

   Evicted Pages: 1 (4K)

        Elapsed: 0.001375 seconds
```

A logical first step a hacker might take to try and resolve the actual relative address in memory is to take advantage of some of the files in the /proc directory. In the example below for process 30463 the entries for the "cat" command used to open and allow appending to the file "filexyz" are displayed. Note that it provides a beginning and ending relative bit address range in hexadecimal. The first line deals with execution related activities for the file. The second line with reads and the third with writes to the file. In all lines the memory is protected to prevent overwrites as indicated by the "p" flag.

It is really interesting to note that in line 3, if one subtracts the beginning address from the ending address a value of 1000 hex is obtained. If converted to decimal this would be a value of 4096 or 4KB. By coincidence could this be the 4KB page that contains the buffered data for the file "filexyz"? If so, it may be at risk and a hacker could use a program such as “gdb” to debug it and to read its contents (Stallman et al., 2002).

```
dennis.guster@eros2:/proc/30463$ cat smaps | grep /bin/cat

5647a41e3000-5647a41eb000 r-xp 00000000 08:02 1310762        /bin/cat

5647a43ea000-5647a43eb000 r--p 00007000 08:02 1310762        /bin/cat

5647a43eb000-5647a43ec000 rw-p 00008000 08:02 1310762       /bin/cat
```

Below an attempt is made to use the "gdb" debug program to dump the 3rd memory segment range from above (5647a43eb000-5647a43ec000) using the rights profile of the user that owns that process ID. However, this results in a "Cannot access memory" error.

```
dennis.guster@eros2:/proc/30463$ gdb

GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1

(gdb) dump memory ~/catlog 0x5647a43eb000 0x5647a43ec000

Cannot access memory at address 0xa43eb000
```

From the examples above one can see that finding the address of memory within the Linux operating system is quite possible. Its design is predicated on providing functionality to the system administrator and developers creating applications to be run on it. However, finding an address is one thing, but being able to compromise the data at that address is another. Remember that with virtual addressing the addresses are constantly changing so right off the bat there is a limited window of opportunity for a hacker to take advantage of it. There are also a number of other built in security precautions as well, most notably related to user rights on the profiles.

Therefore, the purpose of this paper is to evaluate the possibility of data being found in a buffer or virtual address space and actually being compromised. The experiments will be discussed from three different rights levels: user/owner, root and kernel module related.

2 Methodology

In the Introduction section an address was found in memory and an attempt was made to try to read that data, though to no avail when using the user level rights privilege. However, that experiment was then rerun and the sudo command was used to provide root level rights. But once again the data could not be read using the "gdb" command. Thus, a user space process, even running as root, is still limited in what it can do as it is running in "user mode" and the kernel is running in "kernel mode" which are actually distinct modes of operation for the CPU itself. In kernel mode a process can access any memory or issue any instruction. In user mode (on x86 CPUs there are actually a number of different protected modes), a process can only access its own memory and can only issue some instructions. Thus, a user space process running as root still only has access to the kernel mode features that the kernel exposes to it.

Thus, even the root user has limitations. Those limitations are imposed by the design of the operating system to differentiate between user space and kernel space. For instance, even though you are a root user, you can't change the speed at which the hard disk rotates if that option isn't provided to you through the driver (you can write a driver that will allow the function, but even then you are not accessing the hardware directly but through the driver), the reason for this is that the actual control of the hardware is all done in kernel space and the way user space accesses it is through system calls. A kernel space is not a place for a user!

Note below that when the "gdb" command was run via "sudo" that the address is slightly different than from the previous example. This is because virtual memory is currently being used. So therefore, a hacker trying to use past addressing of a process would be stymied due to the dynamic nature of the virtual memory addressing.

```
dennis.guster@eros2:~$ sudo gdb

(gdb) dump memory ~/catlog 0x000055c7b524c000 0x000055c7b524c100

Cannot access memory at address 0xb524c000
```

One might find this situation confusing because one is often taught erroneously that the root has all rights everywhere. Further, it is often taught that if the root doesn't have the rights needed that it has the right to give itself those rights. All of this is true except in entities controlled by the kernel module (Wazan et al., 2022).

To illustrate this concept, one might look at a zombie process, a zombie process refers to any process that is essentially removed from the system as "defunct", yet still resides in the CPU's memory as a "zombie". As one might expect a zombie process is one that cannot be killed even by using the root and the "kill" command with the "-9" option (Linuxize, 2019). So, if the root owns an apparent process that is a zombie, and if that is killed the zombie process would usually be killed as well. However, because the zombie process is often left over from some kernel function such as a remote procedure call to install the NFS client on a host the process is then owned by the kernel module and hence the root cannot directly kill it.

Fortunately, from a security perspective this concept of kernel ownership carries over to various memory segments as depicted in the example below in which not even the root can read the selected memory address. By using the program from (Kannan, 2018) it is possible to determine the virtual address of a memory variable and later link that virtual address to the actual physical address. When the C executable file named (vm-addr) is run the process ID of that executable is returned along with a virtual memory address.

```
dennis.guster@eros2:~/OLDHOME$ ./vm-addr
```

```
my pid: 29675

virtual address to work: 0x55dcd9526260
```

Once the program executes it remains in a wait state (as shown below) so that one can evaluate the memory area.

```
dennis.guster@eros2:~$ ps -al

F S     UID        PID    PPID   C  PRI  NI ADDR SZ  WCHAN   TTY    TIME       CMD

0 S 1895401321 29675  28848   0   80     0 - 1129          wait_w   pts/0  00:00:00   vm-addr
```

So, then step one is to ascertain if the virtual memory area can be accessed using the debug program "gdb". However, as is shown in the output below it cannot be found using the debug program as it again provides a "Cannot access memory" error.

```
(gdb) dump memory ~/vmmemlog 0x55dcd9526260 0x55dcd9526360

Cannot access memory at address 0xd9526260
```

Step two then is using the second program found on the East River Village utilities (mem-addr) from (Kannan, 2018), this is to get access to the physical address for the memory variable.

```
dennis.guster@eros2:~/OLDHOME$ ./mem-addr 30004 0x562c0bb28260

getting page number of virtual address 94747174797920 of process 30004

opening pagemap /proc/30004/pagemap

moving to 185053075776

physical frame address is 0x0

physical address is 0x260
```

Step three is then to determine if that memory can be accessed from both a user and root level using the "gdb" debug program. However, once again the memory is protected because it is effectively owned by the operating system kernel.

```
**(For both user and root accounts)**

(gdb) dump memory ~/phyaddlog 0x260 0x270

Cannot access memory at address 0x260
```

Even though certain ranges of memory addresses are protected, it is sometimes possible to gain access to data via registers and their related addressing. To illustrate this concept a C executable named "add" is run and returns the process PID 30222.

```
dennis.guster@eros2:~/OLDHOME$ ps -al
```

```
F S       UID          PID    PPID  C  PRI  NI ADDR SZ WCHAN  TTY      TIME     CMD

0 S 1895401321 30222  28848  0   80       0 - 1128       wait_w    pts/0    00:00:00   add
```

The utility asks for input via the keyboard and note that the last value entered is an ASCII "5" followed by the enter key. Also, note that the process is still running because only two integers have been entered instead of the three indicated in the first stage. Again, this provides an opportunity to examine memory addressing related to this program.

```
dennis.guster@eros2:~/OLDHOME$ ./add

Enter the number of integers you want to add

3

Enter 3 integers

4

5
```

Again, using the "gdb" debug program it is possible to get a summary of the registers and their associated addresses for this utility. This follows on the logic that was developed in (Farra, Guster, & Rice, 2017), where the source index register (rsi) is shown to contain the buffered data from keyboard entries. Which based on its name, the index contains what you would expect to see, for clarification on how it morphed from its original purpose see (Intel 64, rsi and rdi registers, 2014). The "info registers" command shows the starting relative address of the register related to the add program, which is running as process ID 30222. Note that root access was needed to do this as depicted by the use of the "sudo" command. Next, the first 16 Bytes of that register are dumped to a file called "regmem22".

```
dennis.guster@eros2:~/OLDHOME$ sudo gdb -p 30222

(gdb) info registers

rsi            0x1bd6670        29189744

(gdb) dump memory ~/regmem22 0x1bd6670 0x1bd6680
```

The contents of that file can now be evaluated using the "xxd" (hexadecimal dump) command. The first two characters of the dump are "35" which is an ASCII "5" the next two characters are ASCII "0a" which is a carriage return (or enter) on the keyboard. Off to the right in the interpreted part one can see that the "35" is depicted as a "5". While this only provides a piece of potentially sensitive data a bot could be programmed to record each data chunk as it happens and store them in a file to be read at a later time. There are of course other registers that could be monitored and potentially compromised as well, but this concept provides the basic scenario.

```
dennis.guster@eros2:~/OLDHOME$ xxd ~/regmem22

00000000: 350a 0000 0000 0000 0000 0000 0000 0000  5..............
```

The next question that needs to be addressed then is, will putting an open file that that is in append mode in virtual space, defeat compromising its data via the register attack scenario used above? In this example a file called Data.txt containing 26 Bytes is appended to via the keyboard using the "cat" command.

```
dennis.guster@eros2:~/OLDHOME$ ls -l Data.txt

-rw-r--r-- 1 dennis.guster professors 26 Jan  3  2023 Data.txt

dennis.guster@eros2:~/OLDHOME$ cat >> Data.txt

this is more data.
```

The file then shows up as a regular open file.

```
cat     21120 dennis.guster    1w   REG   0,52       45 27660056 /rhome/dennis.guster/OLDHOME/Data.txt
(10.10.3.5:/exports/rhome)
```

Next, the file is placed in virtual memory using the "vmtouch" command which was used earlier. This provides another level of abstraction in regard to the true memory address of the data contained therein.

```
dennis.guster@eros2:~/OLDHOME$ vmtouch -vt Data.txt

Data.txt

[O] 1/1

    Files: 1

    Directories: 0

    Touched Pages: 1 (4K)

    Elapsed: 0.001294 seconds
```

By using “gdb” debug as before, it is possible to ascertain if all the data is hidden. The same logic as used in the prior example is applied herein and an address for the rsi register (which is being used as a keyboard buffer) is obtained. Note the hex address returned is different than in previous examples seen above. This is a good characteristic of the memory management of the Linux operating system.

```
(gdb) info registers

rsi            0x7f37f9297000   139878380171264
```

The first 48 Bytes of the register are dumped to a file called vmtouchfile (2nd address is 30x higher). When reading this file, it turns out that the register is unaffected by placing the online file in new virtual memory space. By looking at the interpreted portion of the dump one can see that the line "this is more data" is in fact readable. Again, by monitoring the ongoing transactions of the rsi register, data being written to the file Data.txt could potentially be compromised.

```
(gdb) dump memory ~/vmtouchfile 0x7f37f9297000 0x7f37f9297030

dennis.guster@eros2:~$ xxd vmtouchfile

00000000: 7468 6973 2069 7320 6d6f 7265 2064 6174  this is more dat

00000010: 612e 0a00 0000 0000 0000 0000 0000 0000  a..............

00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

These results illustrate an interesting property about computers. That data is often stored in several places before it reaches its ultimate destination, in this instance via the NSF service across the internal network of a private cloud to a SAN (Storage Area Network), as well as on the Linux host the file was being manipulated on. Further, some of that data is retained along the way. A good example of this is the propensity to create multiple replications of data in cloud computing. Other examples might be data contained in the keyboard or print buffers on a computer.

A computer system is a series of devices that are often operating at different speeds, hence the need to buffer data by various means. In the latest example, the online file's ultimate destination was hidden space in virtual memory. However, the data had to go from the keyboard to a register before it got to that point and hence was potentially vulnerable until it got to the hidden within the virtual memory space.

Therefore, simply using virtualization to obscure the location of data is not to be considered as a viable nor comprehensive security strategy. The literature touts the use of a multilayer security approach and certainly this is an imperative strategy (Koon, 2022). However, it may need to go beyond the rights of users that can access the sensitive data of an organization, as many IAM (Identity Access Management) regimes are currently set up in enterprises. The results herein indicate that monitoring memory usage and low-level access to data also merit consideration in a comprehensive security strategy, especially within a public cloud computing environment, which many organizations have moved towards over the course of the last decade and a half.

3 Discussion and Conclusions

As has been pointed out the threat to IT systems and applications continues to be a huge concern for most organizations, be they public, private or governmental organizations.

While some organizations still host much of their IT infrastructure on-premises, many others have moved to the “Cloud”. The continued push towards cloud computing, especially via large public cloud providers such as Amazon AWS, Microsoft Azure, Google and others means that organizations typically no longer have direct control over the hardware that their systems and applications reside on, although those providers do provide a high degree of physical and network security. However, you don’t know who your “neighbor” is that resides on the physical machines your virtual machines reside on in in the “Cloud”. Thus, the ability to protect against memory mining attacks becomes even more important. If a hacker could get onto an on-premises server how much data could they possibly collect? But if that same server was hosting multiple organizations sensitive confidential data, then the potential repercussions could be even greater, if the hacker was able to mine the memory on the machine. While many of the current attacks are centered at the application layer, the Verizon DBIR (Data Breach Investigations Report) typically lists phishing emails as the cause of roughly 80% of data breaches year-over-year, these types of attacks are not the only threat that organizations need to protect themselves against.

The ability to monitor and place protections against the potential theft of data via memory mining or leakage is also a critical security control that organizations need to be aware of and put in place. More competent hackers or hacking groups might also make use of memory-based fileless methods, such as an Excel spreadsheet file embedded with macros or a website running Adobe Flash to carry out attacks that could be undetectable by conventional defenses, such as the (WAF) web application firewall or some form of endpoint protection, though this would still be delivered typically via an email. Without an adequate amount of cybersecurity in place, malware could be allowed to infect systems without end-users being aware of the fact that it has occurred. Because of the nature of computer memory, organizations need to be aware of the fact that memory-based attacks can be a potential threat vector that they need to be aware of, and place necessary controls and countermeasures in place to monitor and alert if malicious attacks do arise.

References

Carrigan, T. (2020, June 10). *Linux commands: Exploring virtual memory with vmstat*. Enable Sysadmin. Retrieved February 19, 2023, from https://www.redhat.com/sysadmin/linux-commands-vmstat

Chng, S., Lu, H. Y., Kumar, A., & Yau, D. (2022). Hacker types, motivations and strategies: A comprehensive framework. *Computers in Human Behavior Reports*, *5*, 100167. https://doi.org/10.1016/j.chbr.2022.100167

Cimpanu, C. (2019, February 11). *Microsoft: 70 percent of all security bugs are memory safety issues*. ZDNET. Retrieved March 18, 2023, from https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Definition of buffer. PCMAG. (n.d.). Retrieved February 10, 2023, from https://www.pcmag.com/encyclopedia/term/buffer

Definition of cache. PCMAG. (n.d.). Retrieved February 10, 2023, from https://www.pcmag.com/encyclopedia/term/cache

Farra, H., Guster, D. & Rice, E. (2017). Security Concerns of Registers in Linux Hosts: Using Debug to Find Memory Addresses of Sensitive Data. Proceedings of MICS 2017, https://www.micsymposium.org/mics_2017_proceedings/docs/MICS_2017_paper_2.pdf

Intel 64, rsi and rdi registers. (2014, April 29). Retrieved February 16, 2023, from https://stackoverflow.com/questions/23367624/intel-64-rsi-and-rdi-registers

Kannan, B. (2018, March 2). *Virtual memory to physical memory*. East River Village. Retrieved January 11, 2023, from https://eastrivervillage.com/Virtual-memory-to-Physical-memory/

Kernel Self-protection. (n.d.) Retrieved February 27, 2023, from https://www.kernel.org/doc/html/v4.14/security/self-protection.html

Koon, J. (2022, November 3). *Memory-based cyberattacks become more complex, difficult to detect*. Semiconductor Engineering. Retrieved March 1, 2023, from https://semiengineering.com/memory-based-cyberattacks-become-more-complex-difficult-to-detect/

Linuxize. (2019, December 2). *Kill command in linux*. Linuxize. Retrieved March 18, 2023, from https://linuxize.com/post/kill-command-in-linux/

Linuxize. (2020, July 18). *Free command in linux*. Linuxize. Retrieved February 8, 2023, from https://linuxize.com/post/free-command-in-linux/

Stallman, R., Pesch, R., & Shebs, S., et al. (2002, January). *Debugging with GDB*. Debugging with GDB - Table of Contents. Retrieved March 1, 2023, from https://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_toc.html

Sterling, T., Anderson, M., & Brodowicz, M. (2018, January 5). *Chapter 11 - Operating systems*. High Performance Computing. Retrieved February 24, 2023, from https://www.sciencedirect.com/science/article/pii/B9780124201583000113

Tayag, M. I., & De Vigal Capuno, M. E. (2019). Compromising systems: Implementing hacking phases. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3391093

Wazan, A. S., Chadwick, D. W., Venant, R., Billoir, E., Laborde, R., Ahmad, L., & Kaiiali, M. (2022). Rootasrole: A security module to manage the administrative privileges for linux. *Computers & Security*, 102983. https://doi.org/10.1016/j.cose.2022.102983

Vostokov, D. (2023). Bytes, Halfwords, Words, and Doublewords. In: Foundations of ARM64 Linux Debugging, Disassembling, and Reversing. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-9082-8_5

Zivanov, S. (2022, October 25). *LSOF command in linux {14 practical examples}*. Knowledge Base by phoenixNAP. Retrieved February 12, 2023, from https://phoenixnap.com/kb/lsof-command