Why it’s needed:

During the driver development, it may happen your driver may be buggy or you are trying to dereference a NULL pointer of wrong address that does not exist in the system.So all the monitoring and debugging techniques,that you used not able to detect sometimes bugs remain in the driver.The system faults when these kind of driver is executed.These kind of driver bugges may generate the the “oops”.

When this happens, it’s important to be able to collect as much information as possible to solve the problem.

System panic :

Note that “oops” doesn’t mean “panic”.The system becomes panic when a fault happens outside of a process’s context or if some vital part of the system is compromised. When the kernel panics, the kernel cannot continue running and the system must be restarted. In some cases the oops may cause a panic if something vital was affected. Oopses in device drivers don't normally cause panics--however, they may leave the system in a semi-useable state.

System OOPS :

Oopses are caused by the kernel dereferencing an invalid pointer.In a user-space program this would normally cause a segmentation fault,also known as a segfault.A user-space program cannot recover from a segfault. When this occurs in the kernel, however, it is called an oops and doesn't necessarily leave the kernel unuseable. An oops can be caused by both hardware problems and kernel programming errors.

Why does a Linux system not save a crash dump when the kernel panics ?
Ans:
The main reason why Linux does not save crash dumps by default is due to the nature of the x86 hardware architecture.When the kernel panics a dump must be written without kernel support.The PC BIOS does not have a means to save the state of memory when the system is rebooted,thereby preventing a reliable means of saving a crash dump.

The system fault occurs during the destruction of the current process while the system goes on working on it.But when the problem is due to a driver error, it usually results only in the sudden death of the process unlucky enough to be using the driver.The unrecoverable damage occurs when a process is destroyed and the memory allocated to the process’s context is lost. For example when a process destroys then,dynamic lists allocated by the driver through kmalloc might be lost.
The oops usually do not bring down the entire system, you may well find yourself needing to reboot after one happens.
A buggy driver can can do this :

- Leave hardware in an unusable state.
- Leave kernel resources in an inconsistent state.
- In the worst case, corrupt kernel memory in random places.

Often you can simply unload your buggy driver and try again after an oops. If, however, you see anything that suggests that the system as a whole is not well, your best bet is usually to reboot immediately.

We’ve already said that when kernel code misbehaves, an informative message is printed on the console. The next section explains how to decode and use such messages.

Oops Messages:

Most bugs show themselves in NULL pointer dereferences or by the use of other incorrect pointer values.The usual outcome of such bugs is an oops message.Almost any address used by the processor is a virtual address and is mapped to physical addresses through a complex structure of page tables When an invalid pointer is dereferenced, the paging mechanism fails to map the pointer to a physical address, and the processor signals a page fault to the operating system. If the address is not valid, the kernel is not able to “page in” the missing address; it generates an oops if this happens while the processor is in supervisor mode.

An oops displays the processor status at the time of the fault, including the contents of the CPU registers and other seemingly incomprehensible information.The message is generated by printk statements in the fault handler.

So now we are going to produced oops message and find try to debug:

Prerequisite :

We are going to experiment with this on 64 bit Ubuntu 18:10.Please update your machine so it contains an updated repository and is able to find the address of all prerequisite software . So we need some tools whose details I provide below. Let’s get started.

1->kdump:

Kdump is a utility that is used for capturing the “system core dump” when the system is crashed.These captured core dumps can be used later to analyze the exact cause of the system failure and implement the necessary fix to prevent the crashes in future.
when the production kernel crashes, another, separate kernel is booted via kexec, into a memory space reserved by the production kernel. The freshly booted separate kernel, also known as the crash kernel, then captures the state of the crashed system, and writes it to disk.

This all works because of kexec. kexec is a fastboot tool implemented in the kernel. It allows other kernels to be booted from an already running kernel, skipping the need to go through standard BIOS routines.

Once the crash dump is written to disk, or even sent over the network to a remote host, the system reboots so uptime can be restored. The crash dump can then be analysed later on.

2->Configure kdump:

2.1->Kernel Configuration:

If we read the kdump Documentation, kdump requires some kernel features to be configured and compiled into the production kernel they are given below :

- CONFIG_KEXEC=y
- CONFIG_CRASH_DUMP=y
- CONFIG_PROC_VMCORE=y
- CONFIG_DEBUG_INFO=y
- CONFIG_MAGIC_SYSRQ=y
- CONFIG_RELOCATABLE=y
- CONFIG_PHYSICAL_START=0x1000000

Details about these kernel configuration option:

CONFIG_KEXEC:

This enables the syscall required for kexec to function, and is necessary for being able to boot into the crash kernel.

CONFIG_CRASH_DUMP and CONFIG_PROC_VMCORE :

This enables the crashed kernel to be dumped from memory, and exported to a ELF file.

CONFIG_DEBUG_INFO:

This builds the kernel with debugging symbols,and produces a vmlinux file which can be used for analysis of crash dumps.The production kernel runs a kernel stripped of debugging symbols for performance,so it is very important to match packages of production kernels and debug kernels.

CONFIG_MAGIC_SYSRQ

This is necessary to be able to use SYSRQ features, such as flushing buffers on kernel panic, and to be able to trigger crashes manually.

CONFIG_RELOCATABLE:

This is set since our kernel is relocatable in memory, so we must also set CONFIG_PHYSICAL_START as the deterministic address in which we can place the crash kernel in memory. Now, 0x1000000 is at 16mb in physical memory, and is the default set in Cosmic Cuttlefish’s kernel.

The nice thing is, all of these features are enabled by default on Ubuntu production kernels, so we don’t need to compile our own kernel today. You can verify that these features are enabled by looking at the config files in **/boot**:

```
$grep -Rin "CONFIG_KEXEC" /boot/config-4.18.0-25-generic
$grep -Rin "CONFIG_CRASH_DUMP" /boot/config-4.18.0-25-generic
$grep -Rin "CONFIG_PROC_VMCORE" /boot/config-4.18.0-25-generic
$grep -Rin "CONFIG_DEBUG_INFO" /boot/config-4.18.0-25-generic
$grep -Rin "CONFIG_RELOCATABLE" /boot/config-4.18.0-25-generic
$grep -Rin "CONFIG_PHYSICAL_START" /boot/config-4.18.0-25-generic
```

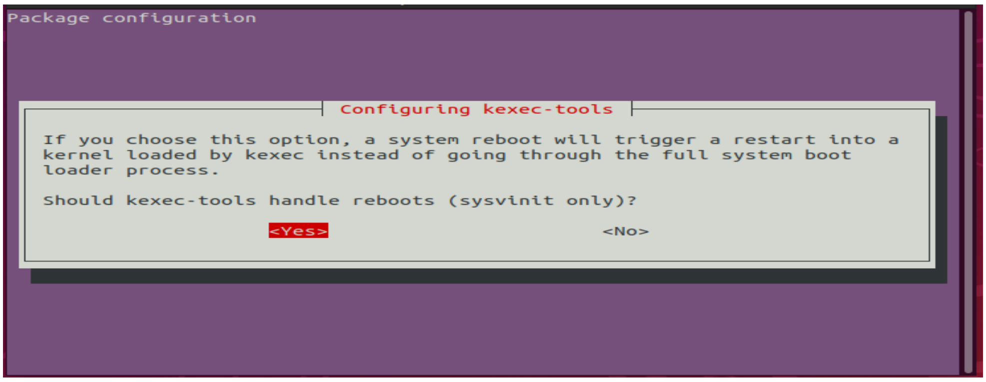
3->Installing Packages:

$sudo apt install linux-crashdump

This will install following given bellow package :

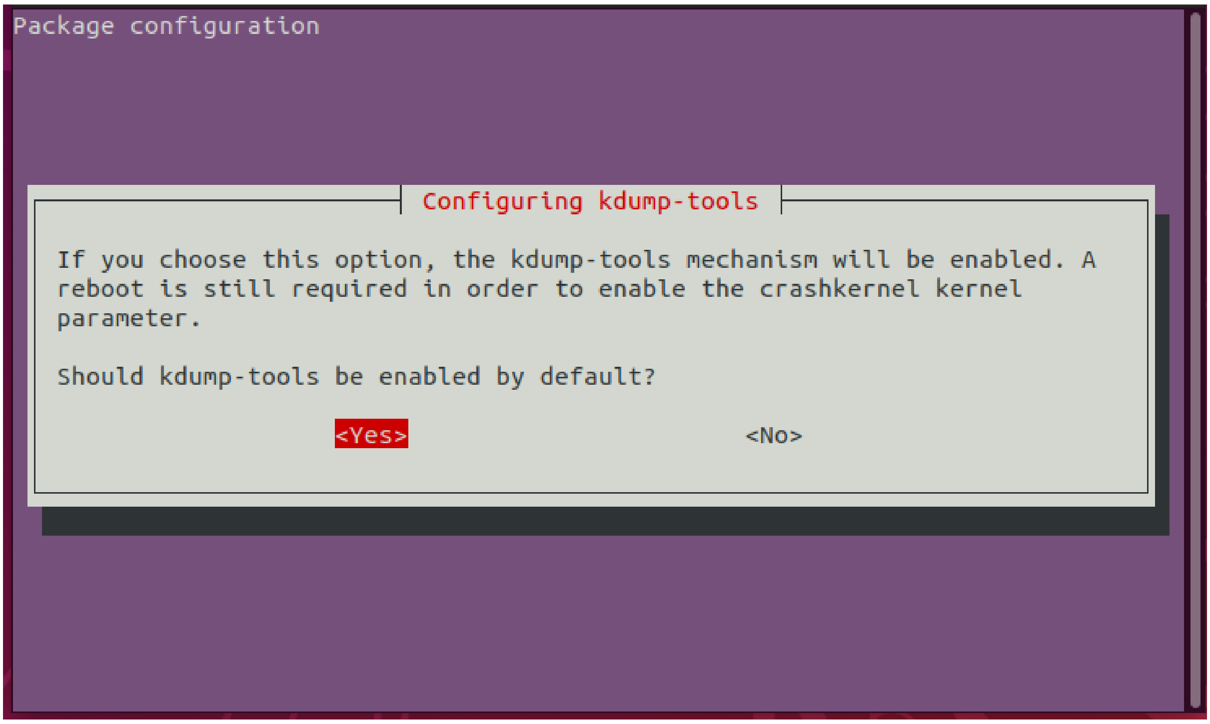
crash, kdump-tools, kexec-tools, makedumpfile

During the installation package they will asked following that given bellow



Should kexec-tools handle reboots (sysvinit only)?

Well, we do want kexec-tools to hook all reboots, so it starts the crash kernel when the system is forcefully rebooted. I’m not entirely sure about why the question says it applies to sysvinit only since Cosmic Cuttlefish uses systemd as its init system, but I am going to say yes anyway.



Should kdump-tools be enabled by default? Well, we are here to learn how to use kdump, so yes, we do want kdump-tools to be enabled by default.

We should probably make sure we have all the necessary kernel packages around as well, such as headers and tools, so run:

$ **sudo apt install linux-image-`uname -r` linux-headers-`uname -r` linux-tools-`uname -r`**

4->Setting up kdump-tools:

We need to tell the kernel where the crash kernel will be loaded in memory, and how much space it has. This happens by appending a **crashkernel=...** to the kernel command line. This has been done for us when we installed linux-crashdump, and we need to reboot the system so that the changes actually take effect. So go ahead and restart.

After restarting, we can see run **kdump-config** show, to see that all is well:

```
nullpointer@ubuntu:~$ kdump-config show
*****************************************************************************
DUMP_MODE:        kdump
USE_KDUMP:        1
KDUMP_SYSCTL:     kernel.panic_on_oops=1 vm.panic_on_oom=1
KDUMP_COREDIR:    /var/crash
crashkernel addr: 0x
   /var/lib/kdump/vmlinuz: symbolic link to /boot/vmlinuz-4.18.0-25-generic
kdump initrd:
```

```
   /var/lib/kdump/initrd.img: symbolic link to /var/lib/kdump/initrd.img-4.18.0-
25-generic
current state:    ready to kdump

kexec command:
 /sbin/kexec -p --command-line="BOOT_IMAGE=/boot/vmlinuz-4.18.0-25-generic
root=UUID=3d0b28d2-9a37-4d7f-9cfd-2da6f5dccb76 ro find_preseed=/preseed.cfg auto
noprompt priority=critical locale=en_US quiet nr_cpus=1 systemd.unit=kdump-tools-
dump.service irqpoll nousb ata_piix.prefer_ms_hyperv=0"
--initrd=/var/lib/kdump/initrd.img /var/lib/kdump/vmlinuz
```

****

A few interesting things to note:

1. Core dumps will be stored in /var/crash
2. The crash kernel vmlinuz is located at /var/lib/kdump/vmlinuz
3. The crash kernel initial filesystem is located at /var/lib/kdump/initrd.img

Verification:

To confirm that the kernel dump mechanism is enabled, there are a few things to verify. First, confirm that the crashkernel boot parameter is present:

nullpointer@ubuntu:~$ cat /proc/cmdline

****

```
BOOT_IMAGE=/boot/vmlinuz-4.18.0-25-generic root=UUID=3d0b28d2-9a37-4d7f-9cfd-
2da6f5dccb76 ro find_preseed=/preseed.cfg auto noprompt priority=critical
locale=en_US quiet crashkernel=512M-:192M
```

****

The above value means:

1. if the RAM is smaller than 384M, then don't reserve anything
2. if the RAM size is between 386M and 2G (exclusive), then reserve 64M
3. if the RAM size is larger than 2G, then reserve 128M

Second, verify that the kernel has reserved the requested memory area for the kdump kernel by doing:

nullpointer@ubuntu:~$ dmesg | grep -i crash

****

```
[0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.18.0-25-generic
root=UUID=3d0b28d2-9a37-4d7f-9cfd-2da6f5dccb76 ro find_preseed=/preseed.cfg auto
noprompt priority=critical locale=en_US quiet crashkernel=512M-:192M
[0.000000] Reserving 192MB of memory at 496MB for crashkernel (System RAM: 2047MB)
[0.000000] Kernel command line: BOOT_IMAGE=/boot/vmlinuz-4.18.0-25-generic
root=UUID=3d0b28d2-9a37-4d7f-9cfd-2da6f5dccb76 ro find_preseed=/preseed.cfg auto
noprompt priority=critical locale=en_US quiet crashkernel=512M-:192M
```

****

5->Testing the Crash Dump Mechanism:

Time for the main event, testing out kdump. We will do this via SYSRQ, and its ability to force a crash and then a reboot. Testing the Crash Dump Mechanism will cause a system reboot. In certain situations, this can cause data loss if the system is under heavy load. If you want to test the mechanism, make sure that the system is idle or under a very light load.

Verify that the *SysRQ* mechanism is enabled by looking at the value of the **/proc/sys/kernel/sysrq kernel** parameter :

nullpointer@ubuntu:~$ cat /proc/sys/kernel/sysrq
176
Note:

If a value of *0* is returned the dump and then reboot feature is disabled. A value greater than *1* indicates that a subset of sysrq features is enabled. See /etc/sysctl.d/10-magic-sysrq.conf for a detailed description of the options and the default value.
In my case 176 can be broken down into 128, 32 and 16. This means that SYSRQ can reboot the system, remount disks as read-only and sync buffers to disk.

Enable dump then reboot testing with the following command :
$ **sudo sysctl -w kernel.sysrq=1**

Once this is done, you must become root, as just using sudo will not be sufficient. As the root user, you will have to issue the command

```
$sudo -s
#echo c > /proc/sysrq-trigger
*****************************************************************************
[31.659002] SysRq : Trigger a crash
[31.659749] BUG: unable to handle kernel NULL pointer dereference at (null)
[31.662668] IP: [<ffffffff8139f166>] sysrq_handle_crash+0x16/0x20
[31.662668] PGD 3bfb9067 PUD 368a7067 PMD 0
[31.662668] Oops: 0002 [#1] SMP
[31.662668] CPU 1

*****************************************************************************
```

The rest of the output is truncated, but you should see the system rebooting and somewhere in the log, you will see the following line :

“Congratulation Your successfully generate system OOPS”

“Debugging the Panic with a crash”

Okay, so our system just crashed. Time to figure out what went wrong and to find wherein the source code the problem occurs. To be able to start a crash, we need a kernel with debugging symbols built-in. Now, this kernel has to exactly match the production kernel that we crashed, and cannot be compressed. For Ubuntu systems, we are looking for the kernel ddeb.

In order to use the generated crash dump with **crash** one needs the *vmlinux* file which has the debugging information. This is part of the kernel ddeb package which can be found at .You need add these packet in your system:
****

```
sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/$(lsb_release -cs) main restricted universe multiverse
deb http://ddebs.ubuntu.com/$(lsb_release-cs)-security main restricted universe multiverse
deb http://ddebs.ubuntu.com/$(lsb_release -cs)-updates  main restricted universe
multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe
multiverse
EOF
```
****

We can then import the GPG key for the repo, refresh package lists and install the vmlinux package with:
****

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys ECDCAD72428D7C01
sudo apt-get update
sudo apt-get install linux-image-$(uname -r)-dbgsym
```
****

1->Starting crash:

- The crash dump is stored in **/var/crash/<timestamp>/dump.xxx**
  ```
  nullpointer@ubuntu:~$ ls -l /var/crash/201912100245/*
  -rw------- 1 root whoopsie   /var/crash/201912100245/dmesg.201912100245
  -rw------- 1 root whoopsie   /var/crash/201912100245/dump.201912100245
  ```
- The debug kernel is located in **/usr/lib/debug/boot/vmlinux-4.18.0-25-XX**
  ls -l /usr/lib/debug/boot/
  ```
  -rw-r--r-- 1 root root 675858456 Jun 24 01:53 vmlinux-4.18.0-25-generic
  ```
- We can start crash with the syntax: **crash <vmlinux> <dumpfile>**
  ```
  #cd /usr/lib/debug/boot
  #crash vmlinux-4.18.0-25-generic /var/crash/201912100245/dump.201912100245
  ```
- Wait sometime and you will see following on the console:

  ****

  root@ubuntu:/usr/lib/debug/boot# **crash vmlinux-4.18.0-25-generic /var/crash/201912100245/dump.201912100245**

```
crash 7.2.3
Copyright (C) 2002-2017  Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010  IBM Corporation
Copyright (C) 1999-2006  Hewlett-Packard Co
Copyright (C) 2005, 2006, 2011, 2012  Fujitsu Limited
Copyright (C) 2006, 2007  VA Linux Systems Japan K.K.
Copyright (C) 2005, 2011  NEC Corporation
Copyright (C) 1999, 2002, 2007  Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002  Mission Critical Linux, Inc.
This program is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it under
certain conditions.  Enter "help copying" to see the conditions.
This program has absolutely no warranty.  Enter "help warranty" for
details.

GNU gdb (GDB) 7.6
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu"...

WARNING: kernel relocated [802MB]: patching 106028 gdb minimal_symbol
values

      KERNEL: vmlinux-4.18.0-25-generic
    DUMPFILE: /var/crash/201912100245/dump.201912100245  [PARTIAL DUMP]
        CPUS: 1
        DATE: Tue Dec 10 02:45:42 2019
      UPTIME: 00:08:49
LOAD AVERAGE: 0.31, 0.26, 0.12
       TASKS: 452
    NODENAME: ubuntu
     RELEASE: 4.18.0-25-generic
     VERSION: #26-Ubuntu SMP Mon Jun 24 09:32:08 UTC 2019
     MACHINE: x86_64  (3408 Mhz)
      MEMORY: 2 GB
       PANIC: "sysrq: SysRq : Trigger a crash"
         PID: 2517
     COMMAND: "bash"
        TASK: ffff89f6d8dede00  [THREAD_INFO: ffff89f6d8dede00]
         CPU: 0
       STATE: TASK_RUNNING (SYSRQ)

crash>
```

****

When we first run a crash, we get a list of basic system configurations, such as the kernel version, system uptime, the date, number of tasks, the hostname, what process was running, and the PID of the process.

1->The processes of the crash system:

We can get a view of the processes the system was running with the ps command:

crash> ps | head -10

```
*****************************************************************************
   PID    PPID  CPU       TASK        ST  %MEM     VSZ    RSS  COMM
      0      0   0  ffffffffb4613740  RU   0.0       0      0  [swapper/0]
      1      0   0  ffff89f6faea1780  IN   0.4  131340   9020  systemd
      2      0   0  ffff89f6faea4680  IN   0.0       0      0  [kthreadd]
      3      2   0  ffff89f6faea2f00  UN   0.0       0      0  [rcu_gp]
      4      2   0  ffff89f6faea0000  UN   0.0       0      0  [rcu_par_gp]
      6      2   0  ffff89f6faee2f00  UN   0.0       0      0  [kworker/0:0H]
      8      2   0  ffff89f6faee5e00  UN   0.0       0      0  [mm_percpu_wq]
      9      2   0  ffff89f6faee1780  IN   0.0       0      0  [ksoftirqd/0]
     10      2   0  ffff89f6faee4680  UN   0.0       0      0  [rcu_sched]
crash>
******************************************************************************
```

2->The dmesg of the crash system:

The log command brings up the contents of dmesg, for that particular session. So the log will also show why the system crashed in the last snippet.

crash> log | tail -40

```
******************************************************************************
[ 1090.643267] rfkill: input handler disabled
[ 1134.097691] sysrq: SysRq : This sysrq operation is disabled.
[ 1150.081279] sysrq: SysRq : This sysrq operation is disabled.
[ 1217.586644] sysrq: SysRq : Trigger a crash
[ 1217.586659] BUG: unable to handle kernel NULL pointer dereference at
0000000000000000
[ 1217.586661] PGD 0 P4D 0
[ 1217.586665] Oops: 0002 [#1] SMP PTI
[ 1217.586669] CPU: 0 PID: 2517 Comm: bash Kdump: loaded Not tainted 4.18.0-25-
generic #26-Ubuntu
[ 1217.586671] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop
Reference Platform, BIOS 6.00 07/29/2019
[ 1217.586693] RIP: 0010:sysrq_handle_crash+0x16/0x20
[ 1217.586694] Code: 48 89 df e8 9c fb ff ff e9 b6 fe ff ff 90 90 90 90 90 90 90
0f 1f 44 00 00 55 c7 05 28 dd 35 01 01 00 00 00 48 89 e5 0f ae f8 <c6> 04 25 00 00
00 00 01 5d c3 0f 1f 44 00 00 55 bf 01 00 00 00 48
[ 1217.586774] Call Trace:
[ 1217.586781]  __handle_sysrq.cold.9+0x66/0x111
[ 1217.586785]  write_sysrq_trigger+0x34/0x40
[ 1217.586790]  proc_reg_write+0x41/0x70
[ 1217.586793]  __vfs_write+0x1b/0x40
[ 1217.586796]  vfs_write+0xab/0x1b0
[ 1217.586798]  ksys_write+0x55/0xc0
[ 1217.586801]  __x64_sys_write+0x1a/0x20
```

```
[ 1217.586805]  do_syscall_64+0x5a/0x110
[ 1217.586809]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[ 1217.586812] RIP: 0033:0x7f56bfbb3fd4
```
****

For this particular crash, the Oops message prints out more than enough debugging information to solve this particular problem. But for more complex bugs, the bt command is interesting.Here 2 red lines showing the why kernel crash.So getting more info what exactly terrible happens in this driver we are going to explore more using bt command.

3->The bt for particular culprit system call:

The bt(Backtrace) normally shows the backtrace of the task which caused the system crash, but you can select other tasks and fetch their backtraces as well.

crash>bt

****

```
PID: 2517   TASK: ffff89f6d8dede00  CPU: 0   COMMAND: "bash"
 #0 [ffffa78c00b3baa0] machine_kexec at ffffffffb3265f43
 #1 [ffffa78c00b3bb00] __crash_kexec at ffffffffb3338a0e
 #2 [ffffa78c00b3bbe8]    oops_end at ffffffffb3231db7
 #3 [ffffa78c00b3bc10] no_context at ffffffffb3276a0b
 #4 [ffffa78c00b3bc70] __bad_area_nosemaphore at ffffffffb3276d10
 #5 [ffffa78c00b3bcb8] bad_area_nosemaphore at ffffffffb3276d74
 #6 [ffffa78c00b3bcc8] __do_page_fault at ffffffffb3277125
 #7 [ffffa78c00b3bd40] do_page_fault at ffffffffb327757e
 #8 [ffffa78c00b3bd70] page_fault at ffffffffb3c0108e
    [exception RIP: sysrq_handle_crash+22]
    RIP: ffffffffb38233c6  RSP: ffffa78c00b3be28  RFLAGS: 00010246
    RAX: ffffffffb38233b0  RBX: 0000000000000063  RCX: 0000000000000006
    RDX: 0000000000000000  RSI: 0000000000000086  RDI: 0000000000000063
    RBP: ffffa78c00b3be28   R8: 53203a7172737973   R9: 0000000000000638
    R10: 7265676769725420  R11: 6873617263206120  R12: 0000000000000007
    R13: 0000000000000001  R14: ffffffffb479a300  R15: 0000000000000000
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #9 [ffffa78c00b3be30] __handle_sysrq.cold.9 at ffffffffb3824096
#10 [ffffa78c00b3be60] write_sysrq_trigger at ffffffffb3823f24
#11 [ffffa78c00b3be78] proc_reg_write at ffffffffb351e181
#12 [ffffa78c00b3be98] __vfs_write at ffffffffb349e92b
#13 [ffffa78c00b3bea8] vfs_write at ffffffffb349eafb
#14 [ffffa78c00b3bee0] ksys_write at ffffffffb349eda5
#15 [ffffa78c00b3bf20] __x64_sys_write at ffffffffb349ee2a
#16 [ffffa78c00b3bf30] do_syscall_64 at ffffffffb32042da
#17 [ffffa78c00b3bf50] entry_SYSCALL_64_after_hwframe at ffffffffb3c00088
    RIP: 00007f56bfbb3fd4  RSP: 00007fffbe938f08  RFLAGS: 00000246
    RAX: ffffffffffffffda  RBX: 0000000000000002  RCX: 00007f56bfbb3fd4
    RDX: 0000000000000002  RSI: 000055d657fd98a0  RDI: 0000000000000001
    RBP: 000055d657fd98a0   R8: 000000000000000a   R9: 00000000ffffffff
    R10: 000000000000000a  R11: 0000000000000246  R12: 00007f56bfc8f760
    R13: 0000000000000002  R14: 00007f56bfc8b2a0  R15: 00007f56bfc8a760
    ORIG_RAX: 0000000000000001  CS: 0033  SS: 002b
crash>
```
****

4->The bt for first culprit system call:
Let’s try and find the root cause of this crash. bt shows us that a write was taking place, due to all the write related syscall functions and vfs write calls.

The bt shows us that the instruction pointer that caused the initial exception was inside the function sysrq_handle_crash, at offset +22.

We can get a better view of sysrq_handle_crash by disassembling the function with the dis command:

crash>dis
****
```
crash> dis sysrq_handle_crash
0xffffffffb38233b0 <sysrq_handle_crash>:        nopl    0x0(%rax,%rax,1) [FTRACE
NOP]
0xffffffffb38233b5 <sysrq_handle_crash+5>:      push    %rbp
0xffffffffb38233b6  <sysrq_handle_crash+6>:         movl     $0x1,0x135dd28(%rip)
# 0xffffffffb4b810e8
0xffffffffb38233c0 <sysrq_handle_crash+16>:     mov     %rsp,%rbp
0xffffffffb38233c3 <sysrq_handle_crash+19>:     sfence
0xffffffffb38233c6 <sysrq_handle_crash+22>:     movb    $0x1,0x0
0xffffffffb38233ce <sysrq_handle_crash+30>:     pop     %rbp
0xffffffffb38233cf <sysrq_handle_crash+31>:     retq
crash>
```
****

At offset +22, we see a call to MOV BYTE PTR ds:0x0,0x1, which tried to move 1 into the address 0x0. Now, you can’t write to a null pointer, since reading or writing to address 0x0 is forbidden.

We see that this causes a page fault, which eventually calls bad_area which then goes onto triggering a crash. From there, we can see __crash_kexec and machine_kexec take over after oops_end, and the system boots into the crash kernel to capture the dump.
****
```
PID: 2517   TASK: ffff89f6d8dede00  CPU: 0   COMMAND: "bash"
              #0 [ffffa78c00b3baa0] machine_kexec at ffffffffb3265f43
              #1 [ffffa78c00b3bb00] __crash_kexec at ffffffffb3338a0e
              #2 [ffffa78c00b3bbe8]    oops_end at ffffffffb3231db7
```
****

“ So ready for the next level of debugging ………!!!!”

Reference :

https://opensourceforu.com/2011/01/understanding-a-kernel-oops/
https://ruffell.nz/programming/writeups/2019/02/22/beginning-kernel-crash-debugging-on-ubuntu-18-10.html
https://www.dedoimedo.com/computers/crash-analyze.html#mozTocId175926

Vmlinux:

- The vmlinux is a statically linked executable file that contains the Linux kernel in one of the object file formats supported by Linux in ELF, COFF format.
- The vmlinux file required for kernel debugging, symbol table generation.
- The vmlinux is the uncompressed version of the kernel image which can be used for kernel debugging.
- The zImage or bzImage is the compressed version of the kernel image which is normally used for booting.
- The vmlinux can't directly be used by UBoot.By the addition of metadata info in the vmlinux is the process of creation of uImage for vmlinux.

Building a proper vmlinux with kdress
https://github.com/elfmaster/kdress
file:///home/byte/Downloads/[Ryan__elfmaster__O_Neill]_Learning_Linux_Binary_A(z-lib.org).pdf