Module 7
Linux Kernel Debugging

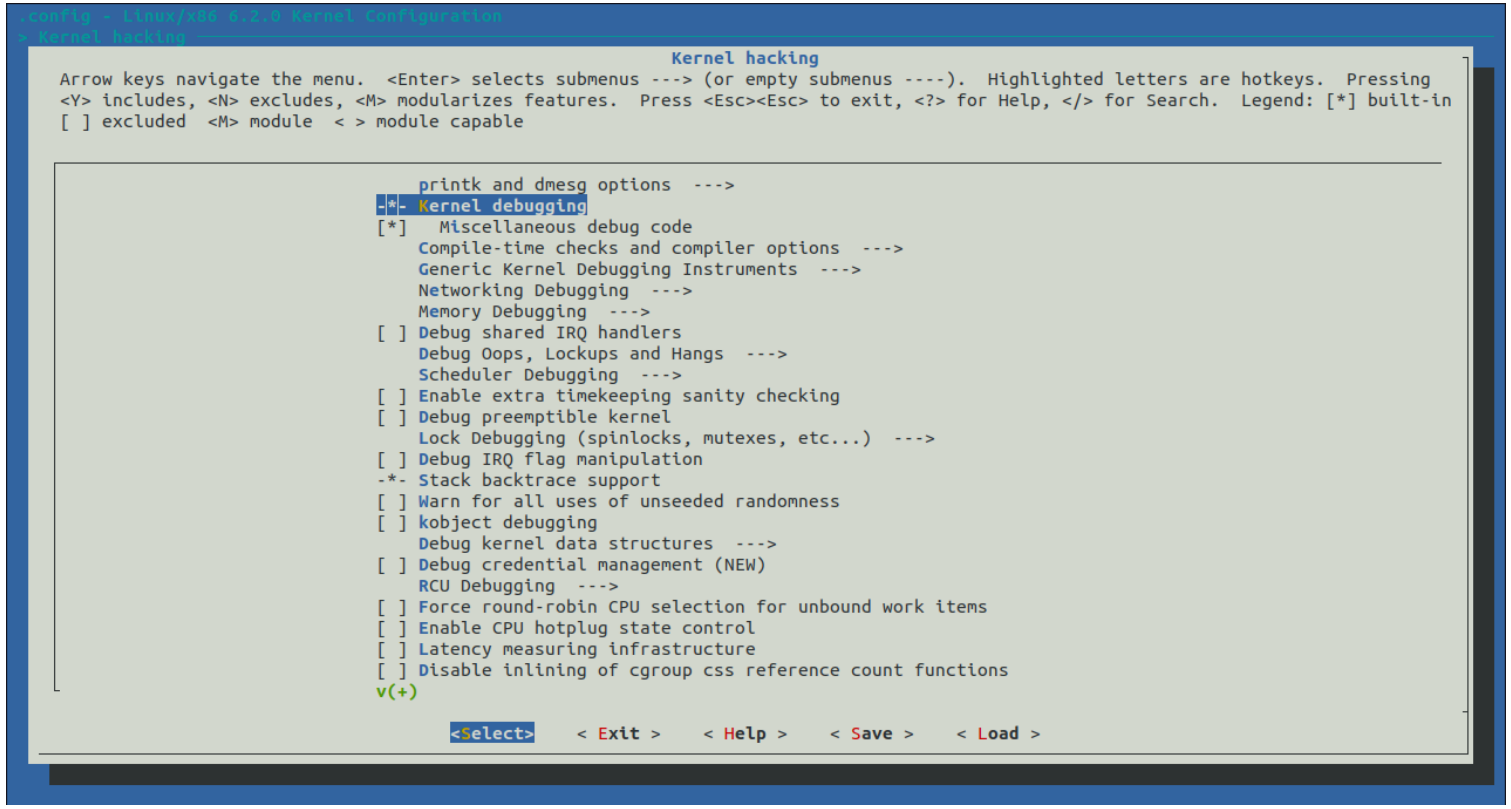
Introduction

- Debugging the Linux kernel is crucial for ensuring system stability, performance, and security, but it is more complex than userspace debugging due to its low-level operations and limited tools.
- Linux kernel provides a range of debugging tools that can be enabled through kernel configuration to aid in the identification and resolution of issues.

Kernel configuration (Kconfig)

- Kconfig is the configuration system used in the Linux kernel to enable or disable features and modules.
- It generates a `.config` file in the kernel source directory which contains all the selected configuration options.
- `make menuconfig` provides a text-based menu interface to configure kernel options through Kconfig.

Kernel Configuration



Kernel debugging support

vmlinux

- vmlinux is a uncompressed, statically linked kernel image contains all the debugging symbols and information needed for in-depth analysis.
- It is generally used with debuggers like GDB and crash utilities.

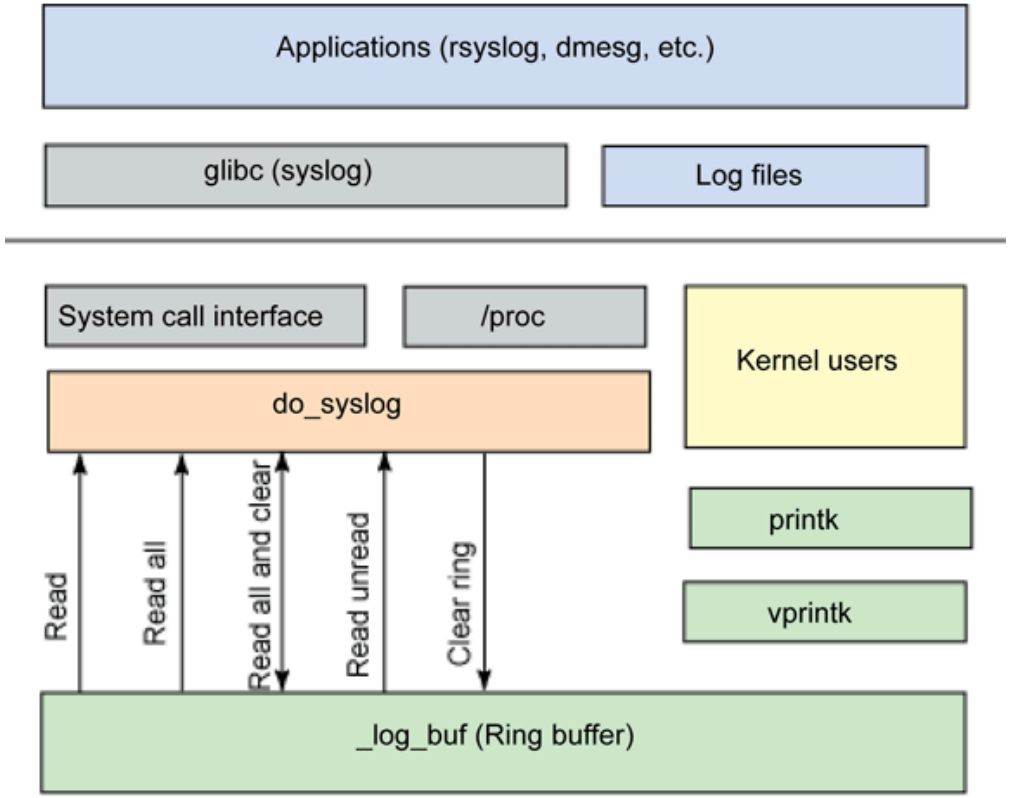
System.map

- System.map is a file containing linux kernel symbol name and its corresponding address.
- It is essential for interpreting kernel oops messages and stack traces.

GDB Scripts

- These are the scripts generated to assist with debugging the kernel using GDB. It automates and simplifies common debugging tasks and procedures.

Kernel logging system overview



Credit: IBM Developerworks

Kernel logging utilities (1/4)

printk() and pr_*() family

- printk() formats and print messages into kernel log buffer.
- `printk()` works similarly to printf() and uses a LOG MESSAGE level as a prefix string.

```
printk(KERN_ERR "%s: irq %u value %x status %d\n", __func__, irq, value, status);
```

- The pr_*() family of functions, namely pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warning(), pr_notice(), pr_info(), pr_debug() and pr_cont(), are the recommended logging functions over printk.

```
pr_info("Ftrace startup test is disabled due to %s\n", reason);
```

- The pr_*() family of functions are macros that wrap `printk` calls with specific log levels and are recommended for use in the kernel.

Kernel logging utilities (2/4)

dev_*() family

- The dev_*() family of functions, such as dev_emerg(), dev_alert(), dev_crit(), dev_err(), dev_warn(), dev_notice(), dev_info() and dev_dbg() are the recommended logging functions for use in device drivers.

Format specifier for kernel logging

- Refer to the printk-format for the full list of format specifiers.
  - %p: Display the hashed value of pointer by default.
  - %pe: Display the string corresponding to the error number.
  - %pS: Symbol pointer (versatile_init+0x0/0x110).
  - %pa: Display physical address.
  - %pK: Kernel pointer to display hashed pointer (depends on kptr_restrict sysctl value).
  - %pOF: Device-tree node format specifier.

Kernel logging utilities (3/4)

kptr_restrict

- kptr_restrict controls the restrictions placed on exposing kernel addresses via /proc and other interfaces.
- It is accessed through `/proc/sys/kernel/kptr_restrict` interface.
  - kptr_restrict = 0, No restriction (full address is printed either hashed or unhashed depending on the format specifier)
  - kptr_restrict = 1, Restricted visibility, root users can see the pointers.
  - kptr_restrict = 2, Kernel pointers are completely masked in all contexts, including for root users and replaced with Zeros such as 0x00000000.

Kernel logging utilities (4/4)

dmesg

- Print or control the kernel log buffer.
  - `dmesg` : Print the kernel log buffer
  - `dmesg -C` : Clear the kernel log buffer
  - `dmesg --level=err,warn` : Print error and warning message only.
  - `dmesg -n 5` : Set console logging filter to KERN_WARNING or more.

Demonstration

- Please refer to this link for sample code related to the demonstration of kernel logging utilities.

Kernel dynamic debugging (1/5)

- pr_debug and dev_debug macros are defined in the kernel with the lowest message level 7 - KERN_DEBUG.
- These functions are used to print debug messages only when the `DEBUG` compiler macro is defined, which can be enabled by either of the following methods:
  - Enable debug options in Kernel config.
  - Add -DDEBUG in Makefile ccflags.
  - Add "#define DEBUG" in source code.
- Every change in a DEBUG message requires rebuilding the kernel or kernel module.
- The kernel's dynamic debugging feature allows enabling or disabling debug messages at runtime without the need to rebuild the kernel or kernel module.
- Enable `CONFIG_DYNAMIC_DEBUG=y` in the kernel config to turn ON dynamic debug feature in the kernel.

Kernel dynamic debugging (2/5)

- Dynamic debug can be controlled through `/sys/kernel/debug/dynamic_debug/control` interface.
- Mount debugfs to access dynamic debugging feature.

```
root@sandbox:~# mount -t debugfs none /sys/kernel/debug/
```

- The feature allows enabling/disabling debug prints by matching any combination of:
  - Source filename
  - Function name
  - Line number (including ranges of line numbers)
  - Module name
  - Format string

Kernel dynamic debugging (3/5)

- Dynamic debugging syntax and usage.

```
echo "<matches> <ops><flags>" > <debugfs>/dynamic_debug/control
```

```
root@sandbox:~# cd /sys/kernel/debug/dynamic_debug/
# Enable dynamic debugging based on file.
echo "file svcsock.c +p" > control

# Enable dynamic debugging based on multiple file.
echo "file drivers/usb/core/* +p" > control

# Enable dynamic debugging based on line number.
echo "file svcsock.c line 1603 +p" > control

# Enable dynamic debugging based on function name.
echo "func svc_tcp_accept +p" > control

# Enable dynamic debugging based on kernel module name.
echo "module nfsd +p" > control

# Disable dynamic debugging based on file.
echo "file svcsock.c -p" > control

# view the currently configured behaviour of all the debug statements.
cat control
```

Kernel dynamic debugging (4/5)

- When using dynamic debugging in kernel modules there are certain things we need to be aware of.
  - The module must be inserted before we can enable dynamic debug.
  - This means we will not see debug statements in the init section of a module.
  - We can confirm that a specific line has been enabled by observing the output of `cat /sys/kernel/debug/dynamic_debug/control` and looking for the `=p` indicator. For example:
  - `dynamic_debug_example.c:85 [dynamic_debug_example]dynamic_debug_function_two =p "dynamic_debug_example: In dynamic_debug_function_two, called with message: %s\n"`
  - This indicates that dynamic debug has been enabled
  - We can then observe the output in dmesg by using a suitable filter: `dmesg | grep dynamic_debug`

Kernel dynamic debugging (5/5)

- Dynamic debug can be enabled from U-boot to allow debugging kernel core code and built-in module during boot phase.
  - use dyndbg="QUERY" for kernel code.
  - use module.dyndbg="QUERY" for built-in module.

```
dyndbg="file ec.c +p"
```

- print_hex_dump_debug() and print_hex_dump_bytes() calls can be enabled dynamically.
- If neither `DEBUG` nor `CONFIG_DYNAMIC_DEBUG` are enabled, these messages are not compiled into the kernel.

Dynamic debuggging case study

- The code used in this example is available at: https://github.com/SpecialistLinuxTraining/linux-debug-training/blob/main/Examples/dynamic_debug

Kernel crash debugging

Kernel crash debugging

- The Linux kernel identifies and logs a "kernel oops" when it encounters a critical,non-fatal error, such as:
  - Invalid memory access (Null pointer dereference, out of bound access).
  - Deadlock detection.
  - Incorrect execution mode (sleeping in atomic context, Enabling Preemption in Critical Sections).
  - Voluntary oops (using WARN_ON() or WARN_ON_ONCE())
- Depending on the severity of the crash, the kernel may either continue running in a degraded state or completely hang.
- The oops message provides detailed information such as an error summary, CPU register states, stack dump, and backtrace, which helps developers in performing a "post-mortem analysis" of the kernel.

Kernel OOPS (1/2)

- The kernel oops message is displayed on the console and logged in dmesg.
- Below is an annotated kernel OOPS snapshot generated using a kernel module example.

```
manas@sandbox:~/work/Examples/crash_kernel_module$ sudo insmod crash-module.ko

========================== ERROR SUMMARY ==========================
[  234.823615] BUG: kernel NULL pointer dereference, address: 0000000000000000
[  234.823637] #PF: supervisor read access in kernel mode
[  234.823646] #PF: error_code(0x0000) - not-present page
[  234.823655] PGD 0 P4D 0
[  234.823663] Oops: Oops: 0000 [#1] PREEMPT SMP PTI

============== CPU# / PID# / KERNEL / Hardware Name ===============
[  234.823674] CPU: 0 UID: 0 PID: 457 Comm: insmod Tainted: G           OE      6.13.8-arch1-1 #1 456fce73f556a6ef8edc444db474316a8147249d
[  234.823700] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006

========================== CPU REGISTERS ==========================
[  234.823710] RIP: 0010:crash_module_init+0x15/0x30 [crash_module]
[  234.823722] Code: Unable to access opcode bytes at 0xffffffffc0a06ffb.
[  234.823730] RSP: 0018:ffffa32e812cbc48 EFLAGS: 00010246
[  234.823739] RAX: 0000000000000029 RBX: ffffffffc0a07010 RCX: 0000000000000027
[  234.823748] RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff89c19bc218c0
[  234.823756] RBP: 0000000000000000 R08: 0000000000000000 R09: ffffa32e812cbad8
[  234.823764] R10: ffffffffa46b5448 R11: 0000000000000003 R12: 000062a47eb11a1b
[  234.823773] R13: ffffa32e812cbc50 R14: ffff89c184d20cc0 R15: ffff89c1822399b8
[  234.823782] FS:  000070886016c740(0000) GS:ffff89c19bc00000(0000) knlGS:0000000000000000
[  234.823792] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  234.823800] CR2: ffffffffc0a06ffb CR3: 000000010a74a000 CR4: 00000000000506f0
```

Kernel OOPS (2/2)

```
==================== BACK TRACE  ====================

[  234.823813] Call Trace:
[  234.823819]  <TASK>
[  234.823824]  ? __die_body.cold+0x19/0x27
[  234.823834]  ? page_fault_oops+0x15c/0x2e0
[  234.823842]  ? search_bpf_extables+0x5f/0x80
[  234.823851]  ? exc_page_fault+0x81/0x190
[  234.823864]  ? asm_exc_page_fault+0x26/0x30
[  234.823872]  ? __pfx_crash_module_init+0x10/0x10 [crash_module 11a11b817c8e53686efb40be3365f6d893cbe205]
[  234.823884]  ? crash_module_init+0x15/0x30 [crash_module 11a11b817c8e53686efb40be3365f6d893cbe205]
[  234.823895]  ? crash_module_init+0x15/0x30 [crash_module 11a11b817c8e53686efb40be3365f6d893cbe205]
[  234.823905]  do_one_initcall+0x5b/0x310
[  234.823915]  do_init_module+0x60/0x230
[  234.823923]  init_module_from_file+0x89/0xe0
[  234.823932]  idempotent_init_module+0x115/0x310
[  234.823940]  __x64_sys_finit_module+0x65/0xc0
[  234.823948]  do_syscall_64+0x82/0x190
[  234.823956]  ? switch_fpu_return+0x4e/0xd0
[  234.823964]  ? arch_exit_to_user_mode_prepare.isra.0+0x79/0x90
[  234.823974]  ? syscall_exit_to_user_mode+0x37/0x1c0
[  234.823982]  ? do_syscall_64+0x8e/0x190
```

- The structure and content of the kernel oops message vary depending on the architecture used.

Kernel OOPS Analysis (1/4)

- Enable CONFIG_KALLSYMS to embed symbols in the kernel image and display them in backtraces using the following format.

```
<SYMBOL NAME>+<HEX OFFSET>/<SIZE>
```

- The program counter register (named RIP, IP, or EIP depending on the architecture) indicates the exact location in the code where the crash occurred.

```
[  234.823710] RIP: 0010:crash_module_init+0x15/0x30 [crash_module]
```

- There are tools available to analyze kernel oops and identify the exact fault location in the code.

Kernel OOPS Analysis (2/4)

GDB

```
:~/Examples/crash_kernel_module$ gdb crash-module.ko
Reading symbols from crash-module.ko...
(gdb) list *(crash_module_init+0x15)
0x25 is in crash_module_init (crash-module.c:10).
5       static int crash_module_init(void)
6       {
7           // Method 1: Explicitly assign NULL
8           int *p = NULL;
9           printk(KERN_INFO "Attempting to dereference NULL pointer...\n");
10          printk(KERN_INFO "Value at address p: %d\n", *p); // This line will cause the oops!
11          return 0;                                         // This line will likely not be reached
12      }
13
14      static void crash_module_exit(void)
```

- The `gdb` tool can be used to display the source code at the location indicated by the symbol in the program counter (PC) register.
- In the example above, line number 10 attempts to dereference a pointer with an invalid address.

Kernel OOPS Analysis (3/4)

addr2line and nm

- Identify the starting address of the `crash_module_init()` function in the kernel module file (crash-module.ko). The nm command can be used to list the symbols within a module.

```
ubuntu@sandbox:~/crash_kernel_module$  nm crash-module.ko | grep -w crash_module_init
0000000000000010 t crash_module_init
```

- Add the offset (0x15: Shown in crash report) to the address retrieved using nm (0x10: crash_module_init()). Then, use the addr2line tool to map this address to the corresponding line in the source file.

```
ubuntu@sandbox:~/crash_kernel_module$  addr2line -fe crash-module.ko 0x25 crash_module_init
crash_module_init
/home/developer/Examples/crash_kernel_module/crash-module.c:10 (discriminator 1)
__pfx_crash_module_init
crash-module.c:?
```

Kernel OOPS Analysis (4/4)

objdump

- Use objdump to disassemble the object file.

```
ubuntu@sandbox:~/work objdump -r -S -l --disassemble  crash-module.ko
...
crash_module_init():
/home/developer/Examples/crash_kernel_module/crash-module.c:6
static int crash_module_init(void) {
  10:   f3 0f 1e fa             endbr64
  14:   e8 00 00 00 00          call   19 <init_module+0x9>
  ...
  19:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi / 1c: R_X86_64_32S        .rodata.str1.8
  20:   e8 00 00 00 00          call   25 <init_module+0x15>
                        21: R_X86_64_PLT32      _printk-0x4
/home/developer/Examples/crash_kernel_module/crash-module.c:10 (discriminator 1)
    printk(KERN_INFO "Value at address p: %d\n", *p); // This line will cause the oops!
  25:   8b 34 25 00 00 00 00    mov    0x0,%esi
  2c:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
```

- The `crash_module_init()` function starts at address 0x10, and adding an offset of 0x15 brings us to the `printk` statement located at address 0x25.

Kernel Crash Analysis Using Sysrq

Kernel Crash Analysis [SysRq] (1/3)

- SysRq is a key combination (typically Alt + SysRq + , where SysRq is often the Print Screen key) that allows users to send commands directly to the kernel, bypassing the usual user-space layers.
- SysRq provides a way to interact with the kernel even when the system is mostly unresponsive. This is particularly valuable for diagnosing system hangs.
- The previous snapshot represents a **Kernel Panic**, which was intentionally triggered by crashing the kernel using the magic SysRq key.
- Please refer to the further slides or this document for more information about the Magic SysRq key.
- The call trace indicates that the kernel panic occurred after the `sysrq_handle_crash+0x1a/0x20` function was called.

Kernel Crash Analysis [SysRq] (2/3)

GDB

- Load the `vmlinux` file into GDB. The vmlinux file may have prebuilt paths that need to be replaced with your own Linux source code path using the `set substitute-path` command in GDB.
- Use the list command in GDB to identify the exact source line causing the panic.

```
root@sandbox:~# gdb /usr/lib/debug/boot/vmlinux-6.5.0-44-generic
Reading symbols from /usr/lib/debug/boot/vmlinux-6.5.0-44-generic...
(gdb) set substitute-path /build/linux-hwe-6.5-QkbMh4/linux-hwe-6.5-6.5.0/ /home/ubuntu/work/linux-kernel/linux-6.5/
(gdb)  l *(sysrq_handle_crash+0x1a)
0xffffffff81a8a4fa is at /build/linux-hwe-6.5-QkbMh4/linux-hwe-6.5-6.5.0/drivers/tty/sysrq.c:155.
150     static void sysrq_handle_crash(int key)
151     {
152             /* release the RCU read lock before crashing */
153             rcu_read_unlock();
154
155             panic("sysrq triggered crash\n");
156     }
```

Kernel Crash Analysis [SysRq] (3/3)

addr2line and nm

- Identify the starting address of the `sysrq_handle_crash()` function in the vmlinux file using nm command

```
root@sandbox:~# nm /usr/lib/debug/boot/vmlinux-6.5.0-44-generic | grep -wi sysrq_handle_crash
ffffffff81a8a4e0 t sysrq_handle_crash
```

- Add the offset (0x1a: Shown in crash message) to the address retrieved using nm (0xffffffff81a8a4e0: sysrq_handle_crash()). Then, use the addr2line tool to map this address to the corresponding line in the source file.

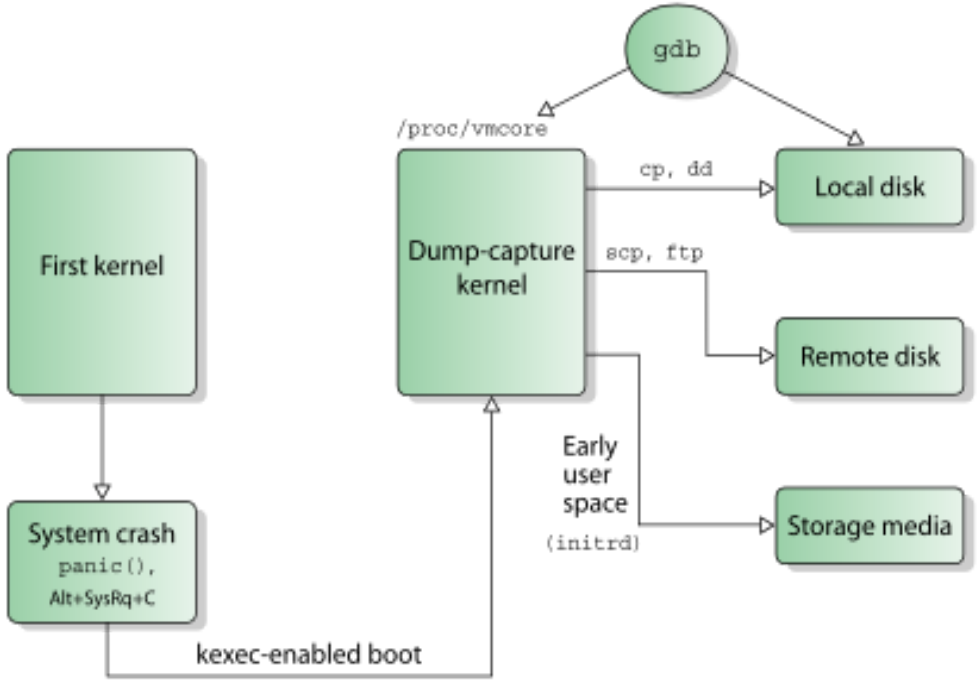
```
root@sandbox:~# addr2line -fe /usr/lib/debug/boot/vmlinux-6.5.0-44-generic 0xffffffff81a8a4fa sysrq_handle_crash
sysrq_handle_crash
/build/linux-hwe-6.5-QkbMh4/linux-hwe-6.5-6.5.0/drivers/tty/sysrq.c:155
```

```
155             panic("sysrq triggered crash\n");
```

Kexec and Kdump (1/3)

- Kexec and Kdump are mechanisms in the Linux kernel used to capture and recover the kernel memory snapshot (/proc/vmcore) after a kernel panic, without requiring a full system reboot.

- It enables the preservation and retrieval of the system kernel's memory image, which can later be used for detailed post-mortem analysis to diagnose the cause of the crash.



Kexec and Kdump (2/3)

- On panic, the kernel's kexec support allows the execution of a "dump-capture" kernel directly from the crashed kernel.
    - The "dump-capture" or Kdump kernel includes only the essential device drivers and features required to capture the crash dump.
- Kexec reserves a small section of memory in the RAM for the dump-capture kernel.
    - Use `crashkernel` kernel boot parameter to specify the memory region for dump-capture kernel.
- The `kexec -p` command loads the dump-capture kernel into the reserved memory.
- After booting from the "dump-capture" kernel, the kernel coredump found at `/proc/vmcore` can be stored as a file on the local filesystem or sent remotely over NFS or SSH.

Kexec and Kdump (3/3)

- You must ensure that the following kernel configuration options have been set in order to use the kexec and kdump functionality.

```
CONFIG_KEXEC_CORE=y (Core kexec functionality)
CONFIG_KEXEC_FILE=y (Newer kexec_file_load syscall)
CONFIG_CRASH_DUMP=y (Enables kdump functionality)
CONFIG_PROC_VMCORE=y (Exposes crash dump via /proc/vmcore)
CONFIG_DEBU_INFO=y (CRUCIAL for debugging symbols)
```

- Please refer to these instructions on configuring kdump and kexec on Ubuntu 22.04 and these instructions for configuring the same utilities on Arch Linux.

Crash Utility

- The crash tool is used to analyze state of the live kernel or memory dump (vmcore) collected from a crashed Linux system.
- The crash tool integrates with GDB, combining the kernel-specific functionality of the crash utility with GDB's source-level debugging capabilities to provide a "kernel-aware" debugging tool.
- It enables interactive post-mortem analysis of kernel data, including CPU registers, process states, memory, kernel symbols, backtraces, and loaded modules from the memory dump.
- The command set of crash utility generaly falls into four category.
  - Kernel data display: struct, union, sym, dis, etc.
  - System state: bt, dev, ps sys, task, vm, etc.
  - Utility functions: ascii, btop, ptob, eval, rd, wr, etc.
  - Session Control Commands: set, alias, exit, foreach etc.

Crash utility usage

- Refer to the instructions on installing the `vmlinux` file on Ubuntu 22.04, which is required by the crash utility to analyze core dumps. For more details on available command set consult the crash dump manual.

```
root@sandbox:/var/crash/202409061559# crash /usr/lib/debug/boot/vmlinux-6.8.0-40-generic dump.202409061559
...
      KERNEL: /usr/lib/debug/boot/vmlinux-6.8.0-40-generic
    DUMPFILE: dump.202409061559  [PARTIAL DUMP]
        CPUS: 32
        DATE: Fri Sep  6 15:59:05 IST 2024
      UPTIME: 00:34:52
LOAD AVERAGE: 0.20, 0.43, 0.36
       TASKS: 947
    NODENAME: sandbox
     RELEASE: 6.8.0-40-generic
     VERSION: #40~22.04.3-Ubuntu SMP PREEMPT_DYNAMIC Tue Jul 30 17:30:19 UTC 2
     MACHINE: x86_64  (2000 Mhz)
      MEMORY: 15.7 GB
       PANIC: "Kernel panic - not syncing: sysrq triggered crash"
         PID: 2711
     COMMAND: "bash"
        TASK: ffff952455685200  [THREAD_INFO: ffff952455685200]
         CPU: 14
       STATE: TASK_RUNNING (PANIC)
crash>
```

Magic SysRq

- Magic SysRq is a key combination used for low-level kernel commands to allow direct interaction with the kernel even when the system is unresponsive.
  - Enable it using `/proc/sys/kernel/sysrq`, where specific functionalities can be allowed or restricted.
- It trigger kernel functions like reboot, crash dump capture, process killing, or syncing disks.
  - On PCs: Trigger it using `ALT + SysRq + <command key>` key combination.
  - On Embedded Systems: Send a break character in the console using `([Ctrl] + a followed by [Ctrl] + \)`, then press command key.
  - Using command line: `echo <command character> > /proc/sysrq-trigger`.

Magic SysRq commands

- Here are some example commands available for use. For more details, please refer to the official documentation.
  - `h` : help
  - `c` : Perform a system crash and a crashdump will be taken if configured.
  - `e` : Send a SIGTERM to all processes, except for init.
  - `g` : Used by kgdb (kernel debugger).
  - `i` : Send a SIGKILL to all processes, except for init.
  - `l` : Shows a stack backtrace for all active CPUs.
  - `p` : Will dump the current registers and flags to your console.
  - `w` : Dumps tasks that are in uninterruptible (blocked) state.
- The functionality also offers an option to register custom commands, which can be triggered using SysRQ.

Kernel lock debugging

Lockdep (1/2)

- Lockdep (Lock Dependency Validator) is a built-in Linux kernel tool used for lock debugging. It detects and report potential deadlocks, lock order violations, and improper lock dependencies.
- Lockdep tracks lock acquisitions, releases, and their order during runtime to ensure safe locking practices. Information for debugging lock issues can be found in the lock debugging file located in `/proc`.

```
/proc/lockdep
/proc/lockdep_chains
/proc/lockdep_stat
/proc/locks
/proc/lock_stats
```

- Use `grep "lock-classes" /proc/lockdep_stats` to track the number of lock classes in use. If the count increases over time, it indicates a possible leak.
- Run `grep "BD" /proc/lockdep`, save the output, and compare it later to spot leaking lock classes or missing runtime lock initialization.

Lockdep (2/2)

- Enable the following kernel configurations for lock debugging to catch issues related to locking problems. Please refer to this demonstration code.

```
 Kernel hacking > Lock Debugging (spinlocks, mutexes, etc...)

[*] Lock debugging: prove locking correctness
[ ]   Enable raw_spinlock - spinlock nesting checks (NEW)
[*] Lock usage statistics
-*- RT Mutex debugging, deadlock detection
-*- Spinlock and rw-lock debugging: basic checks
-*- Mutex debugging: basic checks
-*- Wait/wound mutex debugging: Slowpath testing
-*- RW Semaphore debugging: basic checks
-*- Lock debugging: detect incorrect freeing of live locks
(15) Bitsize for MAX_LOCKDEP_ENTRIES (NEW)
(16) Bitsize for MAX_LOCKDEP_CHAINS (NEW)
(19) Bitsize for MAX_STACK_TRACE_ENTRIES (NEW)
(14) Bitsize for STACK_TRACE_HASH_SIZE (NEW)
(12) Bitsize for elements in circular_queue struct (NEW)
[*] Lock dependency engine debugging
[*] Sleep inside atomic section checking
[ ] Locking API boot-time self-tests
<M> torture tests for locking
<M> Wait/wound mutex selftests
< > torture tests for smp_call_function*()
[ ] Debugging for csd_lock_wait(), called from smp_call_function*()
```

Kernel Concurrency Sanitizer (KCSAN)

- KCSAN is a dynamic race condition detector for the Linux kernel that identifies data races in concurrent code, where multiple threads access shared data without proper synchronization.
    - Suitable for production environments due to low overhead.
- KCSAN uses compiler instrumentation to add watchpoints to memory locations accessed by different threads.
- It monitors these accesses and flags any unsynchronized or conflicting memory operations as potential race conditions.
- To enable KCSAN configure the kernel with CONFIG_KCSAN=y Kconfig option.
    - Use `/sys/kernel/debug/kcsan` interface to configure KCSAN.
- When a race condition is detected, KCSAN logs the memory location, conflicting threads, and a stack trace for debugging.
- Please refer to the KCSAN demonstration code

Kernel Memory debugging

Kmemleak (1/3)

- Kmemleak is a memory leak detector for the Linux kernel, designed to identify possible memory leaks that are no longer referenced but has not been freed.
    - Only suitable for debugging purposes (it should not be used on production systems).
- It uses an rbtree (red-black tree) data structure to efficiently scan and track memory allocations and deallocations in the kernel.
- Kmemleak can be enabled via:
    - CONFIG_DEBUG_KMEMLEAK Kconfig in "kernel hacking" option.
    - `kmemleak=on` kernel command line parameter.
- It can be controlled through debugfs `/sys/kernel/debug/kmemleak` .
- A kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found.

Kmemleak (2/3)

- To display the details of all the possible scanned memory leaks:

```
cat /sys/kernel/debug/kmemleak
```

- To trigger an intermediate memory scan:

```
echo scan > /sys/kernel/debug/kmemleak
```

- To clear the list of all current possible memory leaks:

```
echo clear > /sys/kernel/debug/kmemleak
```

- New leaks will then come up upon reading `/sys/kernel/debug/kmemleak` again.
- Please refer to the Kmemleak demonstration code.

Kmemleak (3/3)

```
echo scan > /sys/kernel/debug/kmemleak
[   62.888171] kmemleak: 1 new suspected memory leak (see /sys/kernel/debug/kmemleak)

cat /sys/kernel/debug/kmemleak

unreferenced object 0x83f92800 (size 1024):
  comm "insmod", pid 155, jiffies 4294939410
  hex dump (first 32 bytes):
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  backtrace (crc 9c453d41):
    kmemleak_alloc+0x44/0x4c
    __kmalloc_cache_noprof+0x2d4/0x310
    0x7f08b024
    do_one_initcall+0x50/0x214
    do_init_module+0x5c/0x228
    init_module_from_file+0x9c/0xd8
    sys_finit_module+0x1ac/0x304
    ret_fast_syscall+0x0/0x54
```

- Kmemleak found a memory object at address 0x83f92800 of size 1024 bytes that is not reachable by any known pointer - a likely memory leak.
- The hex dump shows the first 32 bytes of the leaked memory region, all zeroed; this is typical for an allocated but unused block.

Kernel Address Sanitizer (KASAN)

- KASAN is a dynamic memory error detector for the Linux kernel which is designed to find issues like use-after-free, out-of-bounds memory accesses.
    - Only suitable for debugging purposes (it should not be used on production systems).
- KASAN uses shadow memory (metadata) to track the status of each byte in the kernel's memory to indicating whether it is accessible or has been poisoned.
- Compile-time instrumentation allows the insertion of shadow memory checks before each memory access.
- KASAN can be enabled using:
    - CONFIG_KASAN=y kconfig option
    - `kasan=on` Kernel command line parameter.
- When a memory error is detected, KASAN outputs a detailed report to the kernel log, including the memory location, error type, and call trace.
- Please refer to the KASAN demonstration code.

Kernel Electric-Fence (KFENCE) (1/2)

- KFENCE is a low-overhead sampling-based memory error detector, aimed at detecting heap out-of-bounds access, use-after-free, and invalid-free errors.
    - Suitable for production system due to near zero performance overhead.
- KFENCE works by surrounding specific memory allocations with guard pages (unmapped memory) which serve as protective barriers around the allocated memory to detect out-of-bounds access.
- Any access beyond the allocated memory triggers a page fault upon hitting a guard page.
- KFENCE intercepts this page fault and logs detailed debugging information, including the fault location, allocation size, and access pattern.
- It can be enabled using CONFIG_KFENCE=y Kconfig option.

Kernel Electric-Fence (KFENCE) (2/2)

- KFENCE uses a sampling-based approach to randomly choose memory allocations that it will monitor.
    - Only a subset of allocations will be protected by guard pages at any given time.
- The sampling-based approach significantly reduces performance overhead compared to tools like KASAN, making KFENCE suitable for production systems.
- Compared to KASAN, KFENCE prioritizes performance over precision, relying on extended uptime to catch bugs in less frequently exercised code paths that may be missed by non-production testing.
    - KASAN and KFENCE are complementary, with different target environments.
- Please refer to the KFENCE demonstration code.

Undefined Behavior Sanitizer (UBSAN)

- UBSAN is a runtime error detection tool that identifies undefined behavior in both userspace and kernel space code.
    - Signed integer overflow.
    - Invalid Shift operations for example, shifting by a negative or too large number.
    - Dereferencing misaligned or null pointers.
    - Type mismatch or invalid casts between different types.
    - Refer to the exhaustive list of undefined behaviors in this document.
- UBSAN instruments the code at compile-time by adding checks to detect undefined behavior at runtime.
- Enabled by configuring the kernel with CONFIG_UBSAN and the CONFIG_UBSAN* family for various specific checks.
- When an issue is detected, it logs a detailed message with the type of undefined behavior, source code location, and a backtrace.
- Please refer to the UBSAN demonstration code.

Kernel Debugger

Kernel GNU Debugger (KGDB)

- KGDB is a source-level debugger for the Linux kernel, used with GDB (GNU Debugger) to "break into" the kernel and analyze kernel behavior in real-time.
- Developers can place breakpoints in the kernel code and perform limited execution stepping to trace through kernel execution.
- KGDB runs within the kernel context and halts the kernel execution when a breakpoint is reached. It enables developers to inspect kernel memory, variables, and the call stack, similar to how GDB is used for debugging user-space programs.
- It requires another machine (host) to act as the debugger while the target machine (running the kernel) is being debugged.

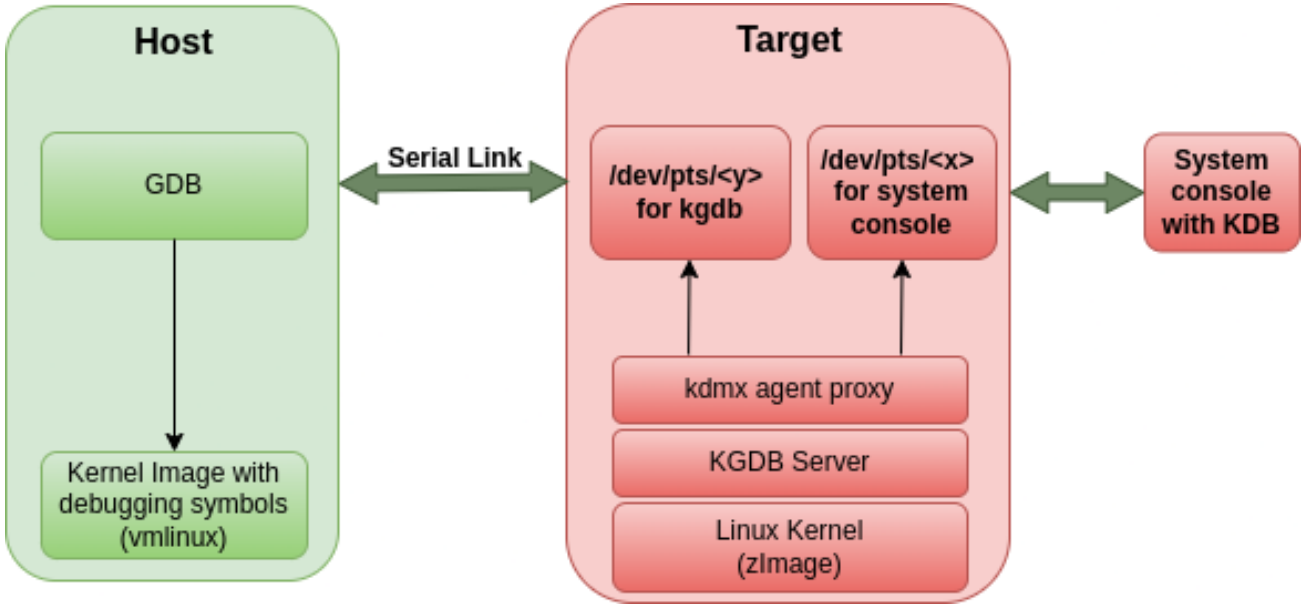
KGDB kernel config

- CONFIG_KGDB=y Enable KGDB support.
- CONFIG_DEBUG_KERNEL=y Enable Kernel debugging support.
- CONFIG_DEBUG_INFO=y Compile kernel with debugging symbols
- CONFIG_KGDB_SERIAL_CONSOLE=y Enable KGDB support over serial.
- CONFIG_KGDB_KDB=y Enable kdb frontend for kgdb.
- CONFIG_GDB_SCRIPTS=y Enable kernel GDB python scripts.
- CONFIG_MAGIC_SYSRQ=y Enable Magic SysRq support.
- CONFIG_PANIC_TIMEOUT=0 Disable reboot after panic
- CONFIG_STRICT_KERNEL_RWX=n Disable memory protection on the code section to allow setting breakpoints.
- CONFIG_FRAME_POINTER=y For reliable stacktraces.
- CONFIG_RANDOMIZE_BASE=n Disable KASLR.
- CONFIG_WATCHDOG=n Disable watchdog.

Kernel Debugger (KDB)

- KDB is a simpler, in-kernel debugger for performing basic debugging tasks without needing an external debugger like GDB.
- KDB offers basic debugging commands like memory examination, register checks, backtracing, and variable modification, but is more limited than KGDB, lacking source-level debugging and detailed inspection capabilities.
- It can be used directly on the target machine via a console (such as the kernel console or a serial console) and is useful for simple in-place debugging.
- KDB and KGDB can work together, allowing developers to switch between using KGDB for advanced debugging tasks and KDB for simpler tasks.
    - Use the `kgdb` command in KDB to enter kgdb mode.
    - Send a maintenance packet from gdb using `maintenance packet 3` to switch from kgdb to KDB mode.

KDMX (1/2)



KDMX (2/2)

- When a system has only a single serial port, it cannot use both KGDB and the console simultaneously because only one program can access the port at a time.
- KDMX is a user-space helper program that acts as a KGDB proxy/multiplexer.
- KDMX solves the issue of using both KGDB and the console simultaneously by splitting GDB packets and console traffic from the serial line into two separate pseudo-terminals (/dev/pts/x), allowing both KGDB and console output to function on a single serial port.

Setup and Configuration (1/4)

KDMX Configuration

- On the target side, ensure that any open connections to the serial port are closed. Then, run the following command to multiplex the serial port.

```
./kdmx -n -d -p/dev/ttyACM0 -b115200

serial port: /dev/ttyACM0
Initalizing the serial port to 115200 8n1
/dev/pts/4 is slave pty for terminal emulator
/dev/pts/5 is slave pty for gdb

Use <ctrl>C to terminate program
```

- Connect with target's serial console using `picocom -b 115200 /dev/pts/4`.
- Connect with GDB from host machine using `gdb <vmlinux File> -ex "set remotebaud 115200" -ex "target remote /dev/pts/5"`

Setup and Configuration (2/4)

- On the target device, enable kgdb over serial console `kgdboc` using CONFIG_KGDB_SERIAL_CONSOLE.
- Configure kgdboc at boot time by passing to the kernel:
  - kgdboc={tty-device},{bauds}.
  - For example: `console=ttyACM0,115200n8 oops=panic panic=0 kgdboc=ttyACM0`
- Or at runtime using sysfs:
  - echo ttyACM0 > /sys/module/kgdboc/parameters/kgdboc
  - If the console lacks polling support, an error will be generated when this command is executed.
- Add the `kgdbwait` parameter to the kernel to ensure it pauses and waits for a debugger connection.

Setup and Configuration (3/4)

Drop into the debugger

- Use Magic SysRq:
  - External keyboard: Alt-PrintScr-G
  - Command line shell: 'echo g > /proc/sysrq-trigger'
  - Send "BREAK-G" over serial port.
- Hardcode a breakpoint into your code: `kgdb_breakpoint()` .
- Cause an oops / panic.
- Create a custom debug trigger by inserting the `kgdb_breakpoint()` function into an IRQ handler which will force the kernel to break into the debugger when the interrupt is handled.

Setup and Configuration (4/4)

Connect with GDB

- On your host machine, start gdb as follows:
  `gdb {vmlinux File} -ex "set remotebaud 115200" -ex "target remote /dev/pts/5"`

KGDB scripts

- GDB provides a powerful Python scripting interface with helper scripts in the Linux kernel to simplify common debugging tasks.
- Enable CONFIG_GDB_SCRIPTS during kernel build and load the scripts into GDB via the `.gdbinit` file.

```
$ cat ~/.gdbinit
add-auto-load-safe-path /path/to/linux-build
<gdb> source vmlinux-gdb.py
```

- `apropos lx` List all GDB commands that contain "lx"
- `lx-lsmod` : Lists loaded kernel modules.
- `lx-symbols` : Automatically loads kernel symbols and module symbols.
- `lx-ps` : Displays the list of processes (`task_struct`) running in the kernel
- `lx-cpus` : Lists available CPUs.
- `$lx_task(pid)` : Fetches the `task_struct` for the Process ID (pid).
- `$lx_current()` : Returns the pointer to the current task in the kernel.

KGDB Demo

- Please refer to the KGDB demonstration code.

References

- https://elixir.bootlin.com/ : To navigate the kernel source code.
- Using Serial kdb / kgdb to Debug the Linux Kernel
- Raspberrypi-and-Buildroot
- Kernel Debugging How to debug the Linux kernel on a VM hosted on VirtualBox.
- Kexec and Kdump on Ubuntu 22.04
- Kexec and Kdump on Arch Linux